

---

# **Interactive Computational Support for Modeling and Generating Building Design Requirements**

**Halil I. Erhan**

Submitted to the School of Architecture of  
Carnegie Mellon University in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
December, 2003

**School of Architecture and  
Institute for Complex Engineered Systems (ICES)  
Carnegie Mellon University**

## **Thesis Committee:**

**Professor Ulrich Flemming, Ph.D. (Chair)**  
School of Architecture,  
Human-Computer Interaction Institute (HCI), and  
Institute for Complex Engineered Systems (ICES)  
Carnegie Mellon University

**Professor Ömer Akin, Ph.D.**  
School of Architecture and  
Institute for Complex Engineered Systems (ICES)  
Carnegie Mellon University

**Professor John R. Hayes, Ph.D.**  
Psychology Department and  
Center for Innovation in Learning (CIL)  
Carnegie Mellon University

---

---

I hereby declare that I am the author of this dissertation.

I authorize Carnegie Mellon University to lend this dissertation to other institutions or individuals for the purpose of scholarly research.

I further authorize Carnegie Mellon University to reproduce this dissertation by photocopying or by other means, in total or in part, at the requires of other institutions or individuals for the purpose of scholarly research.

---

Halil I. Erhan

Copyright © 2003 by Halil I. Erhan  
All rights reserved

---

---

# Abstract

---

Developing design requirements specification or an architectural program for a recurring building type offers unique opportunities for the programming phase in design. Recurring building types are repeated in different contexts, yet their general functional aspects do not change.

However, current practice makes only limited use of these opportunities. The use of passive programming media and manual methods together with non-standardized representation formats create problems with continuity, with upgrading programming knowledge, and with handling complex design requirements. Computer-based tools are generally limited to simple database or spread-sheet applications. Only few of these tools provide some generative mechanisms for formulating design problems separate from solution generation.

I believe that computer-assisted generative tools can assist programmers in partially alleviating the bottlenecks mentioned above and reduce the cognitive loads posed by using traditional manual techniques for handling complex programming information. Based on case studies and an extensive literature survey, I developed a general and flexible framework that models architectural programming knowledge as a generalized (extended) means-ends analysis; it can be made operational in the form of a computer-based support tool that can be adapted by users to any building type and is particularly suited to support programming recurring building types.

RaBBiT is a first prototype application. It is distinguished by the following features: (a) the ability to computationally capture reusable programming knowledge based on a set of concepts that are general enough to accommodate various programming styles while remaining operational; (b) simplification of the designer-computer interaction to make the application usable, even programmable to a degree, for non-computer programmers; and (c) the ability to generate design requirements as output that can be used by other generative design and decision support tools. The first prototype consists of an object-oriented application that is highly integrated with a direct-manipulation user interface.

---

---

# Acknowledgments

---

I thank my wife, Nesil, and our children, Emre and Efe, for their great patience and the sacrifices they have made for my education. I believe Nesil's name should be on this study instead of mine; she encouraged, supported, and motivated me at every turn even when I was ready to give up. I miss my family in Turkey every day I stayed away from them; I thank my mom and dad, my dear brother, Haluk, and my sister, Hatice; if I deserve to be proud of this study, I would like to share it with you.

I can't express enough my gratitude to Prof. Flemming for being my advisor. If it were not for him, I would not have been able to finish this research (although I also think more needs to be done). I learned many things from him, not necessarily all of them academic. Prof. Flemming will continue to inspire me as a professor, researcher, mentor, and friend. I am also deeply thankful to Prof. Akin and Prof. Hayes for their invaluable support, great understanding, and inspiring advice. I enjoyed every moment when I had a chat with Prof. Krishnamurti; his stories about Taiwanese culture are worth listening to.

I am indebted to my friends, Jonah, Hoda, Ipek, and Tanyel (from oldest to youngest). I am thankful for their support in all phases of my study, and for their friendship. Particularly, Jonah was a great help when it came to code writing and target practice. Ipek and Tanyel have been very kind by sharing their lunch with me, not to mention their remarkable comments on RaBBiT. I specially thank Ms. Fox and Ms. Davis for helping all of us graduate students when we needed it. I would need more pages if I wanted to list all of my friends who provided me with their support; they have become not only my friend but also family.

Izmir Institute of Technology has been very generous in providing me with financial support during the early years of my education; I want to thank each of the members of this institution who provided assistance to me. Prof. Flemming provided the rest of the financial support through his projects; another special "thank you" goes to him. I would also like to thank Ms. Kampert and Luis (Rico-Gutierrez) not only for their great help in finding some extra funds but also their invaluable encouragements during the most difficult times.

Thank you God, for everything!



---

# Table of Contents

---

<i>Abstract</i> .....	<i>i</i>
<i>Acknowledgement</i> .....	<i>ii</i>
<i>Table of Contents</i> .....	<i>iii</i>
<i>List of Figures</i> .....	<i>x</i>
<i>List of Tables</i> .....	<i>xiii</i>

## Chapter 1

### *Introduction*

1.1 Background .....	1
1.1.1. Programming as problem specification in architectural design .....	1
1.1.2. A brief review of programming for design .....	1
1.1.3. Characteristics of and bottlenecks in programming .....	2
<i>Programming media and manual programming methods</i> .....	2
<i>Non-standardized representations</i> .....	3
<i>Continuity of knowledge</i> .....	4
<i>Upgrade of knowledge</i> .....	4
<i>Level of complexity</i> .....	4
1.2 Motivation .....	5
1.2.1. Programming for recurring and non-recurring building types .....	5
1.2.2. Design support tools .....	5
1.3 Research Agenda .....	7
1.3.1. Overview .....	7
1.3.2. Scope .....	7
1.3.3. Objectives and Approach .....	8
1.4 Summary .....	10

## Chapter 2

### *Architectural Programming*

2.1 Architectural Programming in Design .....	13
2.1.1. Definitions .....	13
2.1.2. Common elements of definitions .....	14
2.2 Evolution of Architectural Programming .....	14

---

2.2.1. Brief History .....	14
2.2.2. A formal programming discipline .....	16
2.2.3. Emergence of design guidelines .....	16
2.3 Approaches to and Models of Architectural Programming .....	17
2.3.1. Approaches to programming and design .....	17
<i>Integrated approach</i> .....	17
<i>Separated Approach</i> .....	18
<i>Interactive-iterative approach</i> .....	19
2.3.2. Process models of architectural programming .....	20
2.4 Observations and Summary .....	22
2.4.1. Characteristics of the programming approaches and models .....	22
2.4.2. A common pattern .....	23

## Chapter 3

### *Case Studies: Three Recurring Building Types*

3.1 Programming Recurring Building Types .....	25
3.1.1. Overview .....	25
3.1.2. Selection of building types .....	25
3.2 Information Types used in Programming Recurring Building Types .....	26
3.2.1. Activity decomposition structures .....	26
3.2.2. Deriving spaces from activities .....	28
3.2.3. Deriving spatial information from organizational structures and building users .....	30
3.3 Program Parameters for Recurring Building Types .....	32
3.3.1. Roles of program parameters .....	32
3.3.2. Calculation of Area Requirements .....	32
<i>Incremental method</i> .....	33
<i>Choose-and-use method</i> .....	33
<i>Formula-based method</i> .....	34
3.3.3. Spatial relationships. ....	34
3.4 Parameters, Formulas, Logic Statements, and Procedures .....	35
3.4.1. Components and constructs .....	35
3.4.2. Graphical representation of component-construct relationships .....	37



---

3.5 Observations and Summary .....	39
------------------------------------	----

## Chapter 4      *Conceptual Framework*

4.1 Programming as Information Refinement Process .....	43
4.1.1. Step-wise refinement of requirements .....	43
4.1.2. The proposed model vs. existing models.....	43
4.2 The Model in Relation to Means-Ends Analysis.....	44
4.2.1. Means-Ends Analysis (MEA).....	44
4.2.2. Generalized MEA .....	45
4.2.3. Example Scenario .....	45
4.3 Applicability of the Framework.....	47
4.3.1. The framework and programming recurring building types .....	47
4.3.2. The framework at work: a partial example for a recurring building type .....	47
4.3.3. Framework at work: non-recurring building type test case .....	49
4.4 Summary .....	50

## Chapter 5      *System Definition: Features and Requirements*

5.1 Architectural Programming System.....	51
5.1.1. RaBBiT .....	51
5.1.2. User characteristics.....	51
5.1.3. Basic functionality of the system.....	52
5.2 Knowledge Modeling Features and Requirements.....	52
5.2.1. Overview .....	52
5.2.2. Structural requirements.....	52
5.2.3. Behavioral requirements .....	59
5.3 Program Generation Features and Requirements .....	60
5.3.1. Overview .....	60
5.3.2. Structural requirements.....	61
5.3.3. Behavioral requirements .....	62
5.4 Summary .....	62

---

## Chapter 6

## *Technology Selection for RaBBiT*

6.1 System Layers and Technologies .....	63
6.1.1. RaBBiT's system layers .....	63
6.1.2. The main concepts used in RaBBiT .....	63
6.1.3. Programming paradigm and technology .....	65
6.2 Object-oriented (OO) Programming.....	65
6.2.1. Overview .....	65
6.2.2. Sources of software quality .....	65
6.2.3. Modularity and Reusability for RaBBiT .....	67
<i>Class structure</i> .....	67
<i>Inheritance</i> .....	68
<i>Polymorphism</i> .....	69
6.2.4. Object configuration.....	69
6.2.5. OO representation of the knowledge models .....	70
6.3 Production Systems .....	71
6.3.1. Programming paradigm for program generation.....	71
6.3.2. Production systems.....	72
6.3.3. RaBBiT and Production Systems.....	72
6.3.4. Transforming programming knowledge model to a rule-base schema .....	74
6.3.5. User interface .....	75
6.4 Data Structures for Knowledge Modeling.....	75
6.4.1. Data structure requirements .....	75
6.4.2. Data organization techniques .....	75
6.4.3. The graph structure in RaBBiT .....	76
6.5 Model-View-Controller Architecture and System Layers.....	78
6.5.1. Model-View-Controller Architecture .....	78
6.5.2. RaBBiT's MVC Architecture.....	79
6.5.3. The model in RaBBiT .....	79
<i>Programming knowledge schema</i> .....	80
<i>Information nodes</i> .....	80
<i>Associations</i> .....	80
<i>Information categories</i> .....	80

---

<i>Architectural program</i> .....	80
<i>Program schema</i> .....	81
6.5.4. The controller sub-system of RaBBiT .....	81
<i>RaBBiT session manager</i> .....	81
<i>Information category manager</i> .....	82
<i>Programming knowledge model manager</i> .....	82
<i>Interaction manager</i> .....	82
<i>Association manager</i> .....	82
6.6 Implementation Constraints .....	83
6.6.1. Programming Language .....	83
6.6.2. Graph representation .....	83
6.6.3. Parametric associations .....	84
6.6.4. Program generation and information sharing .....	84
6.6.5. Production system shell .....	84
6.7 Summary .....	85

## Chapter 7      *Developing RaBBiT*

7.1 Behavioral and Structural Models .....	87
7.1.1. Overview .....	87
7.1.2. Behavioral models .....	87
7.1.3. Structural models .....	88
7.1.4. Usability Considerations .....	89
7.2 Use-case Driven Software Development .....	89
7.2.1. Overview .....	89
7.2.2. Software development process .....	90
7.2.3. USDP and Unified Modeling Language (UML) .....	91
7.3 Use-Case Descriptions .....	94
7.3.1. Overview .....	94
7.3.2. Session control use cases .....	96
<i>Start a new session</i> .....	96
<i>Open an existing project</i> .....	96
<i>Start a new project</i> .....	97

---

<i>Define a model for requirement information category levels</i> .....	97
<i>Save a project (knowledge model)</i> .....	98
<i>Close a project</i> .....	98
<i>Exit a session</i> .....	98
7.3.3. Knowledge modeling use cases.....	99
<i>Create (Insert) a component</i> .....	99
<i>Insert constructs into a component</i> .....	100
<i>Insert a construct with a resource value</i> .....	101
<i>Insert construct with a reference value</i> .....	101
<i>Insert construct with an expression value</i> .....	102
<i>Insert a global construct</i> .....	102
<i>Insert an association between two components</i> .....	103
7.3.4. Program generation use cases .....	103
<i>Provide project-specific information</i> .....	103
<i>Modify global parameters</i> .....	104
<i>Generate a program</i> .....	104
7.4 System-User Interaction .....	107
7.4.1. Conceptual and physical design of GUIs .....	107
7.4.2. Direct-manipulation paradigm .....	107
7.4.3. Interaction style in RaBBiT.....	107
7.4.4. Model-world metaphor for RaBBiT.....	108
7.5 GUI Design of RaBBiT .....	109
7.5.1. GUI Composition of RaBBiT .....	109
7.5.2. Usability heuristics for RaBBiT .....	111
7.6 Summary.....	114

## Chapter 8

## *Conclusions*

8.1 Observations and Summary.....	115
8.1.1. Overview .....	115
8.1.2. Architectural programming .....	115
8.1.3. The bottlenecks of architectural programming .....	115
8.1.4. Programming for recurring building types.....	115
8.1.5. Information refinement and GMEA .....	116

---

8.1.6. RaBBiT .....	116
8.2 Contributions .....	117
8.2.1. Contributions at the theoretical level .....	117
8.2.2. Contributions at the practical level .....	119
8.3 Future Work .....	120
8.3.1. Overview .....	120
8.3.2. Usability and usefulness analysis .....	121
8.3.3. Multiple system integration .....	121
8.3.4. Adapting framework and RaBBiT .....	121

<i>Bibliography</i> .....	123
---------------------------	-----

## Appendix A: *Case Studies*

## Appendix B: *Program Generation*

## Appendix C: *Object-Oriented Models and RaBBiT*

(in CD)

---

# List of Figures

---

FIGURE 2.1. Integrated approach.....	17
FIGURE 2.2. Segregated Approach. ....	18
FIGURE 2.3. Interactive approach. ....	19
FIGURE 3.1. The USARC activity structure .....	27
FIGURE 3.2. Partial AHCF activity structure.....	27
FIGURE 3.3. Partial ESPS activity structure. ....	28
FIGURE 3.4. Activity decomposition, dependency, and relationship network .....	28
FIGURE 3.5. Corresponding spaces to educational activities in a USARC. ....	29
FIGURE 3.6. Spatial decomposition of out-patient surgery area. ....	29
FIGURE 3.7. The spaces that the ESPS creative program activities take place .....	30
FIGURE 3.8. Non-spatial factors effecting spatial requirements in a USARC .....	31
FIGURE 3.9. Patient-volume and staffing pattern effect the spatial requirements. ....	31
FIGURE 3.10. School capacity and type effect activity composition and change spatial requirements. ....	32
FIGURE 3.11. Analysis of spatial area requirements of a study desk setting .....	33
FIGURE 3.12. Graphical symbols used in the representation.....	38
FIGURE 3.13. The graphical representation of components and constructs relationships in USARC programming. ....	38
FIGURE 3.14. The graphical representation of components and constructs relationships in AHCF programming. ....	39
FIGURE 3.15. Components and constructs in ESPS programming.....	39
FIGURE 4.1. The strictly-hierarchical refinement model (a) vs. the proposed model (b). ....	44
FIGURE 4.2. Decomposition of a problem into successive means and ends .....	45
FIGURE 4.3. Sample transition from high-level to low-level requirements .....	46
FIGURE 5.1. Overall system structure: users and functions.....	52
FIGURE 5.2. Sample dependency associations with and conditions.....	56
FIGURE 5.3. Complex dependency associations.....	57
FIGURE 5.4. Sample relational associations expressed with labels .....	58
FIGURE 5.5. Structural requirements for programming-knowledge model.....	58
FIGURE 5.6. (a) the sample model before remove operation and (b) after remove operation.....	60
FIGURE 6.1. The incremental and iterative programming process supported by the system.....	63
FIGURE 6.2. Programming knowledge concepts and their representations in the model .....	64
FIGURE 6.3. Objectives of a quality software and means for achieving them .....	66
FIGURE 6.4. Inheritance relations among requirement classes (UML notation) .....	69
FIGURE 6.5. Same composition and association (UML notation).....	70
FIGURE 6.6. The generic architecture of rule-based systems. ....	72
FIGURE 6.7. Partial example for programming school buildings. ....	73

---

FIGURE 6.8.A sample graph suitable for program knowledge modeling. ....	77
FIGURE 6.9.Model-View-Controller architecture.....	78
FIGURE 6.10.Modules of the system adapting the MVC architecture.....	79
FIGURE 7.1.Phases and products of use case-driven software development (Flemming et al., 2001) .....	91
FIGURE 7.2.Class structure organized in accordance with the MVC architecture .....	93
FIGURE 7.3.RaBBiT's main window (a) without a project and (b) with a project loaded. ....	97
FIGURE 7.4.Dialog for requirement information category level modeling .....	98
FIGURE 7.5.A sample component's views (a) without and (b) with constructs. ....	100
FIGURE 7.6.The GUI elements for invoking the insert construct use cases. ....	100
FIGURE 7.7.Views for inserting (a) reference and (b) expression values for constructs. ....	101
FIGURE 7.8.Boolean and numeric global constructs .....	103
FIGURE 7.9.Program generator dialog.....	104
FIGURE 7.10.A part of the programming schema in XML definition .....	105
FIGURE 7.11.Partial program data and its sample view as generated by RaBBiT .....	106
FIGURE 7.12.A snapshot from the graph representing a partial knowledge model.....	109
FIGURE 7.13.Snapshot from RaBBiT's main window.....	110
FIGURE 7.14.Mouse icon changes for error prevention (a) Illegal and (b) legal dependency association.....	112
FIGURE 7.15.Sample dialogs for (a) rule violation and (b) error prevention. ....	113
FIGURE 7.16.Shortcuts and accelerators for experienced users.....	113





---

# List of Tables

---

TABLE 2.1.	The terminology used in each of the architectural programming approaches.....	24
TABLE 3.1.	The averages for square footage and number of exam/patient treatment rooms. From MGMA (1999) survey of group practices and space planning. ....	34



---

# Chapter 1 Introduction

---

## 1.1 Background

### 1.1.1 Programming as problem specification in architectural design

Design studies define design as a process comprising overlapping patterns of design *problem specification*, *solution generation*, and *evaluation*. During this process, design progresses from more abstract descriptions of the whole (conceptual design) to more detailed description of the same whole (detailed design).

This research, in overall, addresses the design problem specification phase of architectural design. Design researchers state that *design problem specifications and representations* of such specifications play important roles in defining a framework for design and constitute critical information in design generation. (Reitman, 1964; Cross, 1997; Hinrichs, 1992, pg. 10; Akin and Akin, 1996). This is mainly because "...the problem solving begins with creating a problem formulation [representation and specification]" Simon (1998, pg. 108).

I agree with these authors and also believe that in architectural design, the problem specification phase is an important part of the overall design process and deserves special attention. The specification of an architectural design problem typically starts with collecting and analyzing high-level information about design requirements, which basically is non-computable and *soft*. Some of these requirements can be captured by a set of *computable* and *hard* concepts. For example, depending on higher-level requirements such as image, activity composition, site location, orientation, budget etc., the programmer structures lower-level requirements such as the number and size of spaces that have to be addressed in a design solution.

### 1.1.2 A brief review of programming for design

I refer to the problem specification process in architectural design as *architectural programming*. The process includes the following activities: (a) searching, filtering, and structuring the *information* relevant to the *needs* of a building, (b) generating the *design requirements* from the structured information, and (c) documenting the *requirements* in an *architectural program*. Akin et al. (1995, pp. 153) divide this process into four major steps:

1. Specifying all the design requirements
2. Deriving functional (and physical) descriptions of the architectural problem

3. Documenting the architectural program
4. Updating the architectural program during design.

Sanoff (1977) and Hershberger (1999) describe programming as the *dynamic* and *interactive* "definitional stage" of design. In this stage, an *architectural programmer* makes decisions about "the plan for the procedures and organization of all the resources necessary for developing a design within a specific context and with specific requirements" (Duerk, 1993, pg. 27). Programmers (or designers) document in a program what they understand about the *problem* before they attempt to *solve* it. By using the program, a designer investigates the design problem and its context (Pena et al., 1987).

A *program* is an organized collection of information in the form of *guidelines* and *statements* describing *desired* organizational, functional, and physical properties of a building to be designed—or a product to be designed in a more general sense. These statements may also include objectives and actions to be performed in order to accomplish a design. The program can be stated in many forms. For example, a client may give oral directions to a master-builder to build a house in a rural context, or the client may produce specifications and requirements in written form for a contemporary high-rise building in a big city. These examples represent opposite ends of the spectrum in which a program can be expressed. Designers communicate about a design problem and generate alternative design solutions in accordance with the design requirements stated in a program.

### 1.1.3 Characteristics of and bottlenecks in programming

Current architectural programming practice was investigated by Akin et al. (1995; Donia, 1998) who interviewed six national architectural firms. As part of their investigation, the researchers attempted to discover the basic characteristics of architectural programming, concentrating specifically on potential bottlenecks for design requirement specification. The study identifies some of these bottlenecks, which could be partially overcome through computational support. The following is a review of their findings in connection with the architectural programming literature investigated as part of this research (Chapter 2).

#### **Programming media and manual programming methods**

The two main causes that create bottlenecks in architectural programming are the use of *passive programming media* (such as paper, computer-based word processing and drawing files) and the *manual methods* employed in the process. These two bottlenecks are tightly coupled to each other (Akin et al., 1995, pp. 153; Donia, 1998, pp. 3, 24).

*Passive media* hinder the dynamic association of different pieces of program information with each other. A program—in all likelihood—comprises diverse types of information, which can be expressed in diverse formats such as *textual* (e.g. performance

requirements), *graphical* (e.g. bubble diagram), or *tabulated* (e.g. a spatial area requirement table) (Kumlin, 1995). Even though the information in a program describes interrelated ideas and decisions that form a whole, each piece of information is confined to its representation format due to the passive nature of the programming medium. For example, whenever a building function is added to or removed from the requirements, we expect that this change is propagated to and reflected in all of the associated information (for example, information in the spatial requirements table or functional relations diagrams may have to be updated). The passive medium does not facilitate these kinds of *change propagations* or *updates*, because there is no seamless information association between the different formats. Therefore, handling information updates becomes a complicated task (Akin et al. 1995, pg. 154).

The *use of manual methods* in the process is an outcome of the use of passive programming media—or vice versa. Each piece of information in a program is generated and compiled manually. The consistency and integrity of the program information are also manually maintained by the programmer(s) (Akin et al, 1995). A programmer updates a program document (text) by propagating any change that occurs in any part of the program to the other associated parts manually and one-by-one. Even though some computer applications are employed in generating partial program information, the programmer basically uses the *cut-and-paste* method in maintaining the consistency in a document<sup>1</sup>.

In design requirements specification, currently used representation media and manual methods are also not very efficient when it comes to adapting an existing program in order to generate a new program (Akin et al., 1995, pg. 153). Since the existing program in all likelihood exists in paper form or some other passive medium, its content has to be investigated by the programmer in its entirety to determine its applicability to the new case. This effort becomes even more *labor-intensive* if there are many applicable precedents. Investigating each precedent and establishing associations between relevant information demand considerable time and effort. I believe, in addition, that while adapting existing precedents to a new project, opportunities to improve the programmatic aspects of a particular building type may also be lost.

### **Non-standardized representations**

Another bottleneck in adapting precedents is related to *non-standardized* representation formats. This becomes obvious in the architectural programming literature in which

---

1. "Linking and embedding" data between different computer applications doesn't solve this problem, since these techniques are only useful for *displaying* the information in one file in one application to another file in another application. The semantic contents of the files are not associated and automatically updated.

different authors represent the same type of information in different formats (Pena, 1989; Palmer, 1981; Kumlin, 1995; Duerk, 1993; Hersberger, 1999). This has also been noted in (Akin et al., 1995, pp. 153), who report that they could not find a "consensus" about the use of (textual and visual) representations in architectural programming. If each precedent has been generated by different individuals or groups, the challenges in adapting program information from one case to another become even more pronounced due to non-standardized formats. If a programmer needs to adapt a program generated previously by another programmer, the information has to be translated from one format to another. In current practice, no automation tool addresses this aspect of programming.

### **Continuity of knowledge**

As a side-effect of the inefficiencies of manual methods and non-standardized representation formats, *continuity of the use of programming knowledge* cannot always be guaranteed. Regenerating program information becomes inevitable each time a new case emerges. Experience gained and lessons learned become difficult to generalize and record across similar projects.

Another continuity issue, which is partially tackled in Akin et al. (1995, pp. 153), is posed by human role players. Clients, programmers and designers possess the knowledge of and have experience with a specific programming method. If the involvement of one of these role players in the project is interrupted or terminated, the knowledge possessed by that player is lost, and a new role player has to start from the beginning. In addition, experience in programming is not easily shared across programming teams and firms for the reasons explained above.

### **Upgrade of knowledge**

It is essential for the success of future projects that the programming knowledge about a building type be continuously upgraded. This knowledge has to become available to other programmers working on similar cases as well. Insights and new knowledge discovered in each case should result in knowledge "upgrades" that are shared between and reusable by interested programmers. However, due to the methods used in practice, new insights are carried to the next project only through personal experience and the general literature, which is often not up-to-date and in any case requires interpretation if it is to be applied to a new design situation.

### **Level of complexity**

The level of design problem *complexity* is another factor that contributes to the difficulties of architectural programming. Managing complexity through manual methods and passive programming media becomes more challenging when the number of design elements, the

parameters used to specify these elements, and the relationships between these elements increase. This may also diminish the consistency of the program information.

---

## **1.2 Motivation**

### **1.2.1 Programming for recurring and non-recurring building types**

The case studies conducted as part of this research emphasize that the design requirements generation phase<sup>1</sup> is different for different building types [see Chapter 3]. These differences are especially pronounced between recurring building types and non-recurring building types. By a recurring building type, I mean *a class of buildings with specific functional (activity) and organizational patterns which are shared by other buildings in that class*. Some examples of recurring building types are health care facilities, schools, or public service buildings. A non-recurring building type has a unique identity and functional requirements; examples are fine-art centers, national monuments etc.

The case studies also demonstrated that recurring building types offer opportunities for more efficient and effective specification methods of design requirements. Recurring building types are repeated in different contexts, yet their general functional aspects do not change. Their program components and the relationship between these components are usually well-understood. Typical functions, user characteristics, and general organizational issues establish a common ground for each project. Most probably, a precedent architectural program already exists and can be adapted for a new project.

However, the same bottlenecks observed in architectural programming practice apply to programming recurring building types as well. The problems caused by the use of passive programming media and manual methods negatively affect the process of programming recurring building types as described above. In addition, non-standardized representation formats, problems with continuity, difficulties with upgrading programming knowledge, and complex design requirements impede the process.

I introduce in this research a novel computer-assisted *generative* method able to assist programmers in partially alleviating the bottlenecks identified above. The method is based on a careful investigation of the programming-related aspects of recurring building types.

### **1.2.2 Design support tools**

Design research mainly concentrates on three areas: *describing design*, *providing tools to support design*, and *automating design generation*, if possible. These studies are highly coupled with each other.

---

1. I will use the term "design requirements" synonymously with "design problem specification" from now on.

Design *support* research has these major concerns:

- how computers can assist designers, and in what area
- how design problems can be represented for computational support
- how computers can generate solutions using these representations
- how computers can help us to evaluate the quality of the generated solutions.

Some design researchers have addressed the problem of seamlessly integrating computational design support tools with each other. As an example, the SEED (Software Environment to support the Early phases in building Design) project (Flemming et.al., 2001) attempted to cover almost all of these areas in the domain of architectural design. It has to be noted that these tools are not intended to exclude human designers from the design process; as stated, they are meant to *support* design activities or to *assist* designers.

Much of design research concentrates on generative aspects of design computation; the problem specification phase is not addressed as often. In addition, I found not much work in the literature on support tools that can be employed for design problem specification, particularly for architectural programming. Some recent studies have focused on supporting this process in a computer environment. Maybe the most ambitious effort is SEED-Pro, which was developed as part of the SEED project (Akin et al. 1995; Donia, 1998). Other existing tools are limited to simple database or spread-sheet applications. Only few of these tools provide some *generative* mechanisms for formulating design problems separate from *design generation*. They usually deal with specific criteria of interest for a particular application integrated into a solution representation or implied by the solution algorithm (such as stacking-blocking (Zhang, 1999), cost estimate etc.).

I intend to show in the present work that the success achieved by SEED-Pro can be extended by a system with the following features: (a) the ability to computationally capture reusable programming knowledge of any recurring building type based on a set of concepts that are general enough to accommodate various programming styles while remaining operational; (b) simplification of the designer-computer interaction to make the application usable, even programmable to a degree, for non-computer programmers; and (c) ability to generate architectural programs as output that can be used by different generative design and decision support tools.



---

## 1.3 Research Agenda

### 1.3.1 Overview

The present research attempts to explore the following three main subjects: (a) architectural programming as problem specification; (b) specifying organizational, functional, and physical design requirements for architectural design of recurring building types; and (c) design and prototype implementation of a computer application supporting specifically architectural programming of recurring building types. Scope and objectives of this work are delineated below.

### 1.3.2 Scope

This study focuses on developing a *knowledge model* to be used for capturing *organizational*, *functional*, and *physical* information used in architectural programming in the context of specific building types (the types themselves are not predefined). Use of this model is intended to facilitate the generation of architectural requirements for a particular project of that building type. In order to arrive at such a model, I reviewed the body of general design knowledge as it is available today along with more specific knowledge in the architectural design domain. I then applied the model to develop a generative and computational support tool specifically for modeling requirements in architectural design.

For a successful implementation of the computational tool, I explored formal representations for the knowledge model and programming information that can be generated. The representations are view-independent and include only data and operations relevant for the model or program; they do not predetermine how the data can be presented (viewed) for a particular client application or by a user. While the formal representation enables the knowledge-model to be implemented in a computer environment, the view-independent content allows it to be displayed or exported in any desired format. We may call this *view* and *content separation*.

In order to narrow the scope of the study, I concentrated on design requirements specification for *recurring building types*. I did this for the following reasons:

1. There exists a considerable amount of knowledge about and precedents of recurring building types, which are well documented in design manuals, building-type literature, regulations, codes etc. Any architectural design that involves a recurring building type can take advantage of this knowledge and the experience gained from the precedents.
2. The information and methods used in design requirements specification for recurring building types are well-structured relative to the ones used in programming non-recurring building types. This is mainly because programmers employ more clearly

defined and frequently used information (data) and methods (operations). In addition, the types of information and methods used show similarities across different building types.

3. During this research, I have not encountered a study that explores programming different recurring building types and investigates their differences and similarities. An investigation of the design requirements specification for recurring building types could help programmers overcoming the challenges posed by the bottlenecks listed in previous sections. Furthermore, such an investigation can help design researchers more clearly define and improve requirement specification methodologies for other recurring design situations.

### **1.3.3 Objectives and Approach**

I suggest that the bottlenecks in architectural programming can be considerably alleviated through computer-aided requirements modeling based on a conceptual framework that is flexible enough to allow for

- modeling all types of computable program information,
- sharing program information with other computational tools, and
- providing a usable and interactive programmer-application interface.

In order to test this hypothesis, this research aims to develop a conceptual framework which is *general* enough to cover most of the processes described in the literature and other resources relating to recurring building types. It is crucial that the framework remain *operational* enough to guide the design of an experimental computer application.

Within this overall goal, this research pursued the following objectives:

- To conduct a detailed inquiry into architectural programming and investigate commonalities across recurring building types.

In order to achieve this objective, I conducted an extensive *literature review* [Chapter 2] and several *case studies* [Chapter 3]. The literature review includes findings about the nature of *architectural programming* in general. It establishes essential concepts and terms as well as a range of programming techniques employed. It also shows that different authors have different views of design requirement specification in architectural design. The case studies attempt to provide a detailed documentation on programming for three selected recurring building types and one non-recurring building type [see Chapter 4]. The case studies expose methods, techniques, patterns of use, as well as commonalities of programming for selected recurring building types.

- To delineate a conceptual framework for programming based on the findings from the first objective such that it can guide development of a prototype computational programming tool.

The literature review and case studies demonstrate that architectural programming is a hierarchical information-refinement process that is incremental and iterative in nature. During the process, *higher-level, non-spatial requirement information* is transformed into *lower-level and spatial requirements*. In this study, I describe a framework for capturing the refinement process as an *Extended (Generalized) Means-Ends Analysis* (EMEA), in which *one means to achieve a higher-level requirement becomes an end at the next level* and, unlike common means-ends analysis techniques, *one means can be used to achieve multiple ends*. Means and ends are structured such that taken together, they form an acyclic-directional graph. The nodes of the graph represent requirements that are means as well as ends (except for the leaf nodes). The links between nodes represent the means-ends relationships, i.e. dependencies among different requirements at different information levels.

- To explore how architectural programming (of recurring building types) can be supported by state-of-the-art computational tools in a generative computer environment and how the proposed framework can be adapted by that tool.

This objective is addressed by designing and implementing the prototype of an interactive architectural programming tool as a proof-of-concept application based on the conceptual framework. The application is named *RaBBiT: Requirement Building for Building Types*. RaBBiT is intended to assist architectural programmers or designers to interactively model the type-specific programming knowledge of any building type and to generate architectural programs for a project of this type.

RaBBiT's internal architecture is developed using object-oriented software engineering techniques—which are proven to be effective for many the software projects (Rumbaugh et al., 1991, pp. 9). The architecture encapsulates programming domain model in objects that represent requirements as *component* and parameters of the requirements as *construct* objects. The associations and relationships between the requirements information are captured through specialized *dependency* and *relationship* objects. An acyclic and directional graph data-structure captures these objects and defines a model which maps to a (building) type-specific domain model. This model can be stored persistently as well as shared with other applications through model transformation.

The object-oriented architecture of the system provides a *modular* and *extendible* structure. These two concepts are essential for *flexibility* in software design (Meyer, 1997, pp. 51). Object-oriented programming enables us to flexibly organize not only the system

architecture but also the domain knowledge model contained in the system. This domain model is used for generating organizational, functional, and physical design requirements encapsulated in domain objects [Chapter 5].

- To implement a human-computer interface which is easy to use and enjoyable to work with.

RaBBiT utilizes current interactive human-computer interface technologies. It provides a direct-manipulation style user interface, where domain objects can be created through simple interactions such as *point-and-select* and *drag-and-drop* in a modeling area. Complex operations—such as model and data consistency checking, and change propagation—are performed by the system hidden from the user. However, the user is informed of any such changes when necessary through simple dialogues. The graphical-user interface components, such as windows, buttons, menus, icons, and pointers, provide a consistent look and interaction across other window-based applications. In addition, the look-and-feel of RaBBiT can adapt to the native computer platform that the user operates on. RaBBiT's human-computer interface was designed and developed as an integral part of the software development.

An important feature of RaBBiT's user interface is that it enables architectural programmers to use terms of their choice when modeling a building type. Therefore, the application does not impose any terminology to be used for the identification of domain objects. For example, the user can call a *component* object a "requirement" or "requirement information."

---

## 1.4 Summary

In this chapter, I briefly investigated design requirements specifications in architectural design and identified some of the bottlenecks of current programming practice. I also described the differences between programming recurring and non-recurring building types. The challenges caused by the bottlenecks are common for both recurring and non-recurring building types. However, in most of the cases, programming recurring building types can be considered relatively well-structured in comparison to non-recurring building types; this may help us to define less ambiguously the information used and methods employed in design requirements specification of recurring building types. It will also be helpful in establishing the data and operations that can be used in the design and implementation of an experimental prototype of a computational tool for design requirements specifications, i.e. architectural programming.

The experimental computational tool that I propose, RaBBiT, adapts a framework for modeling domain knowledge in architectural programming. The framework is based on the findings of the case studies coupled with the literature review. Primarily, the framework shows us how to capture architectural programming knowledge—particularly reusable information—of any recurring building type in the form of an extended means-ends-analysis. A knowledge model based on the framework can be used to generate requirements for a particular project. In other words, the knowledge model contains generic and reusable information pertaining to a specific building type; and each time a program of that type is needed, the knowledge model is used in generating design requirements.

Programmers interact with the system through a *direct-manipulation style* human-computer interface. While the system maintains a view-independent knowledge model and program, the user-interface can provide different presentations of the model through a group of objects called *managers*.



## Chapter 2 Architectural Programming

---

### 2.1 Architectural Programming in Design

#### 2.1.1 Definitions

In early phases of architectural design, architectural programming is used to understand the requirements a proposed facility must satisfy. Based on these requirements, the preliminary design decisions are made (Palmer, 1981; Sprecklemeyer, 1982). Therefore, programming involves gathering, compiling, and evaluating information that aims at determining the needs of a facility and the facility's organizational, functional, and physical requirements.

The literature on architectural programming is heterogeneous. This starts with the terms used to denote the programming process itself. Some of the terms are "architectural analysis" (Pena and Caudill, 1959), "building programming" (Davis, 1969), "environmental programming" (Farbstein, 1976), "functional programming" (Davis and Szigeti, 1979), "facility programming" (Palmer, 1981; Preiser, 1985,1993; Sanoff, 1992), "design programming" (GSA, 1983), "project programming" (White, 1985), and "space programming" (Kirk and Spreckelmeyer, 1988). Hershberger (1999) states that the only reason for programming in architectural design is to achieve an "architecture" that responds effectively to the environment, to user (client and occupant) requirements, to functional and performance needs, and to many other factors. Since the main focus of the present research is architecture in this sense, I will stick with the term "architectural programming" in accordance with Hershberger.

Authors also cannot agree on a single general definition of architectural programming. Below are some of the definitions found in the literature:

Pena (Pena et al. 1987): Programming is *"a process leading to the statement of an architectural problem and the requirements to be met in offering a solution"*. Programming and design are two unique processes mutually exclusive. Design succeeds programming, and programming is a *separate* phase in the overall design process.

Duerk (1993): Architectural programming is “*the systematic process of gathering and analyzing information about a building or other setting, and then using that information to create guidelines for the performance of that setting.*”

Chery (1999): Architectural programming is “*...the research and decision-making process that defines the problem to be solved by design.*”

Hershberger (1999): Architectural programming is the first stage of architectural design. In this stage “*relevant values of the client, user, architect and society are identified; important project goals are articulated; facts about the project are uncovered; and facility needs are made explicit.*”

### **2.1.2 Common elements of definitions**

Even though these varying definitions reflect different attitudes about programming, almost all of the authors agree on the following points:

1. Architectural programming is the first step of the architectural design process in which a design problem is specified.
2. During programming, a wide range of information is gathered, compiled, and documented in the form of a program that represents a shared understanding of the design problem between design participants and other stakeholders.
3. The design solution is not the main concern for programming (Pena et al., 1987). Rather, programmers document what they understand about the problem.
4. The main goal of programming is to contribute to the achievement of *high-quality* architecture.

An architectural program has to define an architectural design problem in such a way that the description should be complete and specific enough for design generation. “A good architectural program does not anticipate what a project should look like or what it should be made of. It should describe the desired performance (requirements) and leave to the designer the development of forms to accommodate those performances” (Cherry, 1998). Other researchers and practitioners agree with this view (Pena, 1982; Hershberger, 1999; Duerk, 1993).

---

## **2.2 Evolution of Architectural Programming**

### **2.2.1 Brief History**

According to several authors, there has been always a *programming phase* in architectural design practice (Kumlin, 1995). Before the modern age, when people needed a new “shelter” (building or facility) for a particular purpose (dwelling, religious, defense etc.),



they gave *some form of brief and functional instructions* to the builder, who could also be the designer. Even the most complex (in historical context) buildings could be built with these instructions. This type of interaction and instruction was sufficient in pre-modern times for the following reasons:

- Buildings had less complex functions and design solutions were conventional;
- Builders could adapt well-known programs to new designs. A practicing architect, or even a builder, could heuristically decide functional needs and spatial requirements. There was little need for elaborate instructions.
- Technological changes were slow, and available technologies were sufficient and well-known. There was no need to specify them explicitly. If needed, this was done during construction.
- There were few institutions that needed complex facilities (such as religious buildings). The requirements for these facilities were also well-known.

Over time, architectural design and building types became more complex. In the early eighteenth century, nation states and government institutions needed more complex and specialized facilities to accommodate their newly emerging functions. The industrial revolution also required new complex building types. With the addition of new types of civic complexes, this created a need for a formal and elaborate programming phase (Preiser, 1985; Pena and Focke, 1987; Sanoff 1992; Kumlin 1995). Technological advancements added more complexity to building design and designers needed more detailed *design directions*.

Some of the earlier *architectural programs* were prepared for architectural design competitions and showed that architectural programming was turning into a prominent phase in design. These programs helped the creation of very famous architectural projects. Examples of these projects are the Paris Opera House by Garnier (1861-1874) and the Amsterdam Stock Exchange Building by H.P. Berlage (1898-1903) (Kumlin, 1995). These precedents lead to the recognition of architectural programming as a separate discipline in architectural design.

However, it was only in the 1960s that programming was formally introduced into the literature by authors such as Wheeler (1966), Horowitz (1967), and Agostini (1968) (Preiser, 1985). Pena and Focke of Caudill Rowlett Scott<sup>1</sup> (1975; 1987) established a more formal process for programming. Architectural programming was subsequently

---

1. This firm later became CRS and CRSS and is currently known as Hellmuth, Obata and Kassabaum.

investigated both at a theoretical level and as a process by Sanoff (1977; 1988), White (1985), Duerk (1993), Kumlin (1995), Cherry (1998), and Hershberger (1999).

### 2.2.2 A formal programming discipline

The initial rationale for programming was the need to establish effective communication among designers, builders, and users of the built environment (Preiser, 1977). This need became especially obvious for the design and construction of complex facilities. Most of the complex design cases were related to governmental, industrial, and institutional facilities, which eventually required substantial and sophisticated architectural programs.

This lead governments officially to accept architectural programming as a separate professional activity in the project delivery life cycle. In Canada, for example, the Public Works Department recognized programming as part of an eleven-phase building delivery life cycle (Preiser, 1985). In the USA, the General Services Administration prepared a two-part handbook for architectural programming that describes the process of programming in the governmental context (Zeisel, 1982). In the United Kingdom, architectural programming was similarly recognized as part of the building delivery life cycle (Hall, 1996).

### 2.2.3 Emergence of design guidelines

Architectural programming has become particularly important for large organizations and government agencies with highly "complex and substantial construction programs, frequently consisting of *repetitive building types*", such as offices, factories, schools etc. (Preiser, 1985; 1986). Therefore, government, industrial, or institutional agencies have started to compile *design guidelines* for their future facilities. These guidelines explicitly define the *general* design criteria that can be applied to *specific* projects. The "Design Guidelines on U.S. Army Services Schools" are an example. It contains state-of-the-art programming criteria at the level of individual space categories as well as the entire building and site development (Department of Army) (Preiser, 1985).

Other government agencies and large organizations, such as education departments, health organizations, or private corporations, also created design guidelines to be used as a reference in *programming* and *designing* their facilities. The Facilities Standards for the Public Buildings Service (PBS-PQ100.1) is an example. Another example is the "Postsecondary Education Facilities Inventory and Classification Manual" (Physical Facilities, 1992) that provides definitions and a coding structure for documenting and classifying spaces in higher-education facilities (colleges and universities). Similarly, the Ohio School Design Manual was developed for various types and sizes of schools to assure a certain level of quality across the school districts in the state (OSDM, 2002). The

United States Army Reserve also maintains a design manual for the construction of its training centers (AR 140-483, 1994).

Guidelines for programming various building types can also be found in the general programming literature. Most of these are not developed for or by a particular institution or agency. For example, the programming guidelines for ambulatory health care facilities published by Malkin (1982; 1989; 1997; 2002) provide general descriptions of these facilities including their space requirements. The guidelines also cover information relating to medical space planning, such as occupancy, equipment use, and esthetical concerns. Bobrow et al. (2000) also cover health care facilities and their characteristics. Legget et.al (1977), Ortiz (1994), and Perkins (2001) deal with elementary and secondary school facilities and provide various types of programming guidelines and criteria that address many of the functional and physical needs of these facilities. Other authors provide architectural programming resources for other buildings types.

---

## 2.3 Approaches to and Models of Architectural Programming

### 2.3.1 Approaches to programming and design

Approaches to architectural programming differ in two main areas: (a) when architectural programming starts and ends (stages of programming and design), and (b) the purpose of architectural programming in the general design process (Akin et al, 1995). These approaches can be categorized into three groups (Rabinson and Weeks, 1999; Hersberger 1999).

#### Integrated approach

In this approach, design itself is viewed as programming (Rabinson and Weeks, 1984). It is assumed that a design problem cannot be comprehensively understood before the design starts. Any definition of a design problem will be premature until there is an attempt to find a solution for it. Therefore, the entire design process integrates architectural programming and design generation (Figure 2.1). In this approach, a program (document) is not necessary; the design representations record both the program and the solution. Since programming is integrated into design generation, the outcome of programming is part of the final design documentation.

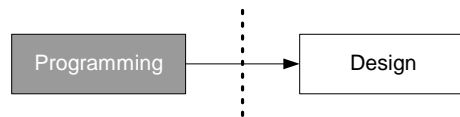


FIGURE 2.1. Integrated approach.

I agree with Cherry (1998, pp. 11) that this approach is applicable only to small projects, where the designer and programmer are the same person who has experience with the design problem at hand. In more complex design situations, the designer is usually given project information regarding the budget, organizational structure, functional and spatial requirements, etc., which describe a framework for design at the outset. This certainly requires a pre-design requirements specification effort to collect information and compile it for design.

### **Separated Approach**

In the second approach, architectural programming and design (solution generation) are *separated*. This separation is believed to be necessary for maintaining the integrity of each of the phases (i.e. programming and design) and for avoiding trial-and-error design alternatives (Figure 2.2).



**FIGURE 2.2.**Segregated Approach.

Pena (1977), who advocates this approach, states that "program is problem seeking, then design is problem solving". A problem cannot be solved unless it is understood at a sufficient level of details. Kumlin (1995) also agrees with this and states "programming must be based on a segregated (separated) *program-then-design* approach applied at the beginning of any defined task and completed more or less prior to the commencement of design".

In the segregated approach, an architectural program is a *utility for design* that is "regarded as a (isolated) step in the early stages of design" (Akin et al, 1995). The purpose of the program is to define user needs that will be addressed with design entities later. Program documentation stops as design starts.

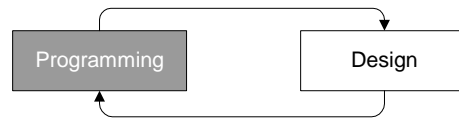
There are two basic rationales behind this approach. The first one is that programming should precede design so that inefficient design and prejudiced solutions are eliminated (because the problem formulation is known when design begins). The second rationale reflects the fact that there are numerous parties with different backgrounds involved in *design*. Separating programming from solution generation lets each participant concentrate on that aspect he or she is interested in. For example, the programmer concentrates on defining the problem and the designer concentrates on finding solutions for the defined problem. Programing involves more verbal, numerical, and relational information; whereas in design, esthetical, form-related, and spatial information

dominates. Separating these two phases allows different designers who are given the same design program to generate different design solutions. (Kumlin, 1995)

But in this approach, programmers have to anticipate almost every problem that can emerge during design, which is very difficult, if not impossible, in most cases. For example, the most feasible form to accommodate a function may conflict with the budget requirements. Programmers may not notice this conflict prior to design. But designers would have to revise the requirements in the architectural program, a situation not allowed in the separated approach.

### Interactive-iterative approach

This approach integrates programming with design in the form of repetitive program-design-feedback loops or iterations (Kumlin, 1995). The loops occur throughout the design process (Straub, 1980; McLaughlin, 1988) (Figure 2.3). At each iteration, the level of detail in the design solution increases. As design ideas become clearer, the corresponding program is revised accordingly (Cherry, 1998).



**FIGURE 2.3.** Interactive approach.

In the interactive approach, design is accepted as a *heuristic* process that evolves from the general to the specific (Duerk, 1993; Cherry, 1998). In contrast to the separated approach, programming is not a one-time, self-contained task. The iterative approach accepts programming and program documentation as a dynamic process extending through the entire design process. The program therefore evolves parallel to the design solution.

In this approach, an architectural program becomes a *framework for design*. The basic purpose of a program is to serve as "the complete inventory of design requirements and criteria of design evaluation" (Akin et al, 1995). The program includes the evolving and changing architectural design requirements and becomes an "informal contract" between the client and the designer.

The advantage of the integrated approach is that it covers "the most striking aspects of design, that it progresses from more abstract descriptions of the whole (an initial program and conceptual design) to more detailed description of the same whole (detailed design)."<sup>1</sup> Programming and design take place with many iterations and feedback loops. We see this

---

1. From discussion by my advisor Prof. Ulrich Flemming

most clearly when we realize how design progresses from functional requirements to sketches, and on to more detailed descriptions at different scales and at different levels of resolutions. At each scale, designers should have access to information appropriate for that scale—at different levels of resolution different types of requirements are needed. The interactive approach assures that the design requirements are generated in parallel with the solution as needed. Therefore, I believe that the interactive approach is more appropriate and feasible than other two approaches introduced in the literature.

### **2.3.2 Process models of architectural programming**

When it comes to specific process models, authors differ again, but all introduce a series of specific and mainly sequential (with or without feedback loops) programming activities. The steps deal generally with information gathering, program preparation, program documentation, and program evaluation.

One of these process models is outlined by Farbstein (1976) and later broadened by Preiser (1985). In the initial step in this model, a programmer investigates the design problem and states the main purpose of the project. In the following phase, the *organizational goals* and *objectives* of the client, which are derived from the building user characteristics and their expected behaviors and the *purpose* of the project, are documented. The stated objectives and goals are translated into specific building *functions* referring to the departmental names of the organizations, such as administration, meeting, security etc. The functions are broken down into *sub-functions* that need spatial allocations. As the functional requirements become clearer, functional relationships are determined, and each function is further detailed by adding *physical requirements*. The functional and physical requirements—which the authors refer to as *performance criteria*—are then used for determining *space specifications*. As part of space specifications, programmers determine spatial adjacency, accessibility, and constituency relations. Subsequent to the completion of the first draft of a program, programmers discuss alternative requirements to the ones specified in the first draft.

Pena and his colleagues (Pena, 1959; Pena et al., 1977; Pena et al., 1987) significantly influenced the theoretical foundations of programming practice. The function of programming, in their model, is "strictly limited to problem analysis, and synthesis is left to the designers". At the first step of Pena's programming model, a *brief problem description* is derived. This is followed by collecting, organizing, and analyzing the program information and sorting it into a standard program information table, which comprises function, form, economy, and time<sup>1</sup> categories. The categorization facilitates a systematic investigation of each piece of program information. The programmers derive *project goals* from each of the information categories. The goals yield to the definition of

project specific facts that usually refer to the project objectives. The determined facts are used in finding the *concepts* for the project that cover the performance requirements. The programmer determines the *project needs* encompassing specifics of the project requirements such as spatial requirements, space allocations, spatial relations, cost etc.

Sanoff (1977) introduces programming activities at two levels. At the first level, the programmer collects program information and establishes the project's main purpose. The compiled information is transformed into the *goals* and *objectives* of the design. The programmer derives the *performance criteria* of the project considering the stated goals and objectives. The performance criteria describe functional and physical requirements of the project as well as some basic parameters (such as area per occupant, project budget etc.). At the second level, the programmer determines the *quantitative requirements* of the project relating to each of the functional and physical requirements. The quantitative requirements capture equipment needs, space allocations, spatial relations etc.

Duerk advocates "issue-based programming", which is a hierarchical decision-making process (Duerk, 1993, pp. 20, 36). The first step derives the *mission statement* of a project. The *values*, which involve defining the special needs and overall qualitative features of the facility, are determined at the next step. The values form a base for defining *program issues*, such as image, function, structure, organization etc. and their priorities for a project. The issues are handled according to their priorities and evolve to project *goals*. The goals address the project requirements more specifically than values and issues. The programmer derives *performance requirements* from the specific goals. The performance requirements comprise both the quantitative and qualitative attributes of the facility to be designed. The *concepts and object specifications* detail the possible physical and spatial requirements. The information generated during this process is documented at an increasing level of detail.

Kumlin (1995) bases his approach on Pena's model, but proposes a more flexible model in which *priority issues*, *objectives*, *program concepts* and *design concepts* are developed from a standard information checklist, which is also used for management of the programming process. The checklist includes priorities, program objectives, concepts, space standards, organization diagrams, space list, affinities, grouping diagrams, flow diagrams, spatial data, and environmental requirements. The checklist is also used for compiling equipment data, site information, existing facility analysis, cost and budget (feasibility), and some other types of information. The granularity of the design requirements increases as the checklist becomes complete.

---

1. Category definitions are influenced by the Vitruvian triad *venustas* (delight), *commoditas* (utility), and *firmitas* (firmness).

Cherry (1998) proposes a six-step process, which starts with project background research. This is followed by a *mission statement* that captures the overall project description. The *context of the project* is studied and defined following the mission statement. The project *goals and objectives* are identified based on the project context information. The programmers select *strategies* for achieving the project goals and objectives. The outcome of the applied strategies leads to establishing the project's *quantitative requirements*. Each requirement becomes a *solution criterion* for the design generation.

In "value-based" programming, Hershberger (1999) proposes a sequence of transition from more abstract to more concrete programming decisions. The initial step in this process is to define the *purpose of the project*. At the next step, the programmer determines the *values* of the project pertaining to the client, users, site, climate, programmer, and even to the designer. The values lead to functional, social, physical, and physiological *issues*. The issues discovered at the previous stage yield to more specific project *goals and objectives*. Each objective leads to the definition of various project *facts*, in turn; facts are used to determine the specific (functional, budget, physical) *needs* (i.e. performance requirements) of the project. At the lowest level, *spatial requirements* are derived from the stated project needs.

These models cover the main process-related issues in architectural programming. Other models for architectural programming are mostly derivations or repetitions of these models with slight variations.

---

## **2.4 Observations and Summary**

### **2.4.1 Characteristics of the programming approaches and models**

I observed the following characteristics from the described approaches and model

- In programming, the programmers formulate a given design problem. In some approaches, problem formulation takes place as an integral part of design generation; in others, it is viewed as a separate process from design generation. However, a more comprehensive and more feasible approach is to interactively continue programming through out the design process (interactive approach).
- A common characteristic of the described models is that design requirements gradually evolve from higher-level information to lower-level (mostly spatial) and more detailed design requirements. At the higher-level, project goals and objectives are stated. At the lower levels, the programmers express the physical (spatial) and functional properties of facilities to be designed.



- Making *form-related* decisions, such as the shape of windows or rooms, is not part of programming—at least for the interactive and segregated approaches. In other words, programmers stay clear of any design decision.
- Process models differ in the methods by which the client's organizational goals and objectives are analyzed. The differences manifest themselves in how programmers collect information and how they sort the collected information into different information categories.

#### 2.4.2 A common pattern

Independent from the terms used and their meanings, a common pattern emerges: All authors treat programming as a process of step-wise refinement of program information, in which most abstract information is first derived—such as goals—and then transformed into more concrete and detailed requirements—such as spatial properties and relations.

Table 2.1 presents the transition from abstract goals to concrete requirements and the terms used by different authors. When I looked at the terms and their meanings used by different authors, I observed some fundamental similarities. That is, authors frequently use different terms to refer to similar concepts (steps). For example, "the mission statement" of a project is used in a similar sense to "purpose of the project". "Design concept" (Kumlin, 1995) and "solution criteria" (Chery, 1998) actually refer to "spatial requirements" (Hershberger, 1999). In the table, the background of terms has the same shade if the terms have similar meanings. The transition from higher- to lower-level requirements, on the other hand, is represented by changes in the density of the background shade from lighter to darker grey.

Pena (1969, 1977)	White (1972)	Markus (1972)	Farbstein (1977 and 1985)	Palmer (1981)	Verger and Kaderland (1993)	Duerk (1993)	Kumlin (1995)	Cherry (1998)	Hershberger (1999)
Problem description	Mission statement	Mission statement	Purpose	Mission of the facility	Main goal	Mission statement	Mission statement	Mission statement	Purpose of the project
Information categories	Fact categories	Systems	Goals	Factor categories	Design issues	Values	Priority statements	Context defini- tion	Values
Goals	Project facts	System goals	Performance criteria	Ascertain- ments	Goals	Issues	Issues	Goals and objective categories	Issues
Facts	Needs	Design issues	Space specifica- tions	Predictions	Needs	Goals	Program objectives	Strategies for goals and objectives	Goals and objectives
Concepts	Requirements	Spatial requirements		Recommen- dations	Spatial requirements	Performance requirements	Program concepts	Solution cri- teria	Facts
Needs						Concepts and object specifications	Design concepts		Needs
									Spatial Require- ments

**TABLE 2.1.** The terminology used in each of the architectural programming approaches.

---

## Chapter 3 Case Studies:

# Three Recurring Building Types

---

### 3.1 Programming Recurring Building Types

#### 3.1.1 Overview

Architectural programming, in general, involves *complex* information handling. The complexity is due to the huge amount of information handled, the variety of information types encountered, and the methods used in compiling a program. Program information ranges from the client organization's goals and objectives to the spatial needs to accommodate the client organization's activities. Programmers systematically take this information as input and produce an architectural program as output for a design project.

When programming a non-recurring building type, programmers must generate the program from scratch. On the other hand, recurring building types provide programmers with a relatively well-established collection of program information in the form of design guidelines, manuals, standards, and other published resources. They may also have access to precedent programs or are personally familiar with the building type at hand. Programmers are able to take advantage of this information because it eliminates the need for an *investigation of general programmatic issues in detail*. The present case studies explore basic commonalities of programming information and methods of recurring building types, to be used in forming a conceptual programming framework as indicated in Chapter 1. This chapter presents a summary of the case studies; a full description can be found in Appendix A.

#### 3.1.2 Selection of building types

I hypothesize that there are commonalities among the methods and concepts that programmers use when programming different recurring building types. In order to test this hypothesis, I investigated relatively well-established architectural programming processes as they occur for ambulatory health care facilities (AHCF), United States Army Reserve Centers (USARC), and elementary and secondary public schools (ESPS). I selected these types primarily for the following reasons:

- There is a huge demand for new facilities, at least when it comes to AHCFs and ESPSs.

- Each building type has unique functional, organizational, and social characteristics. This gives us a wide spectrum of requirement types to study.
- Each of the selected building types is relatively complex. Therefore, findings from studying these building types may be easily applied to less complex building types.<sup>1</sup>

Secondary reasons for selecting these building types:

- The organizational structures of the selected building types are well established, where recurring needs of the organizations can be relatively easily observed.
- Private and public organizations as well as individual researchers have documented design guidelines for each of these building types. These guidelines include the general missions of the client organizations that show substantial similarities.

---

## 3.2 Information Types used in Programming Recurring Building Types

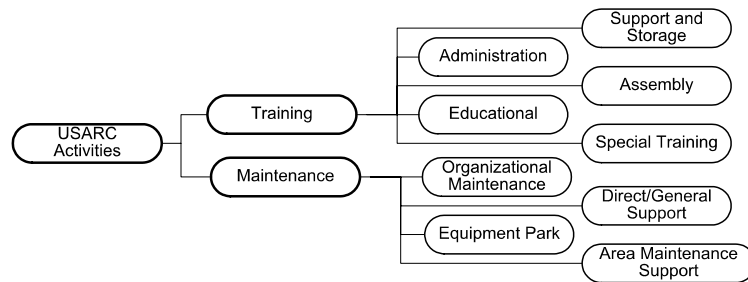
### 3.2.1 Activity decomposition structures

In this study, I use the term *activity structure* to denote the major functions and functional relationships that a facility accommodates. In the following, I give examples of the activity decompositions of the investigated building types.

In a typical USARC, two major groups of activities take place: training and training-related maintenance and support activities (DG, 1984). Training activities consist of five main groups: administration, instruction, assembly, storage-support and special training. In addition, special army equipment-use training (such as weapons, tank turret, or simulation) can take place as an extension of the special training activities. The maintenance group consists of maintenance activities that are training-oriented and of area maintenance support activities. Direct and general support activities consist of ancillary functions such as dressing, tool storage, shop management etc. These activities support both organizational and area maintenance activities. The activity structure for USARC is shown in Figure 3.1.

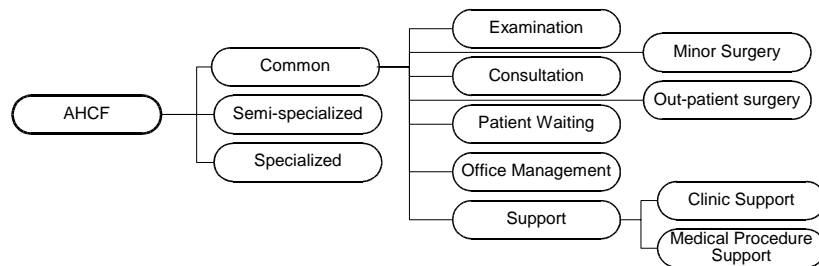
---

1. Fast-food stores that belong to a chain, for example, are also a recurring building type, but the complexity level is not as high as for the selected types.



**FIGURE 3.1.** The USARC activity structure

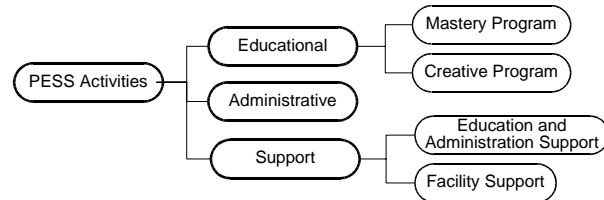
An AHCF comprises three groups of activities that form the backbone of its activity structure (Figure 3.2). The first group includes activities common to all medical specialties. The second group contains activities that are shared between one or more specialty, but not by all. The third group belongs to one particular specialty and is not required for the other specialties. For example, *examination of patients* is a common activity for all specialties, whereas *refraction test* is a specialized activity for ophthalmology. The medical specialties such as otolaryngology or internal medicine may require *specimen testing*, but psychiatry and ophthalmology do not. Thus, *specimen testing* falls into the second group of activities. Depending on the medical specialty, the activity structure changes. Figure 3.2 shows the common AHCF activities; the activities in other groups are shown in Appendix A in detail.



**FIGURE 3.2.** Partial AHCF activity structure

Formal school activities fall into three main groups: educational, administrative, and support activities (Figure 3.3). The educational activities consist of *mastery program* and *creative (group) program activities* (Legget et.al. 1977). In the mastery program, students learn through oral and written instructions and exercise reading, mathematics, and language skills. Creative activities place emphasis on discovery, creativity, and problem-solving skills. Social studies, science, art, music, and physical education form this activity category. Both mastery and creative programs are interwoven to form a complete educational program. The administrative activities, on the other hand, are separate and assure that the school's operation is planned and managed without any problem or delay.

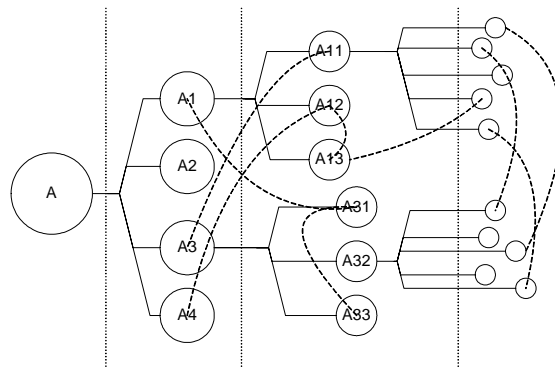
They range from record-keeping to curriculum and schedule assignments. Support activities provide basic services such as food service, library services, student health service, sanitation etc.



**FIGURE 3.3.**Partial ESPS activity structure.

The activity structure for ESPS can expand to include other activities (such as community education, parent meetings, extra-curricular sport activities etc.); or it can be modified to accommodate different approaches to education.

In each of the activity decompositions studied, I observe that the overall function of the facility is decomposed into specialized activities. The specialized activities, in turn, are hierarchically divided into more specialized sub-activities until a desired level of activity resolution is reached. This common pattern can be viewed as a hierarchical tree. However, when the dependencies and relationships between activities in the different branches of the tree are considered, the tree structures transform into more complex networks of activities (Figure 3.4). These networks specify a facility's spatial organization and can be represented with a graph.



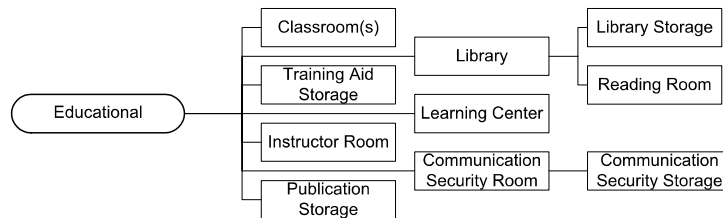
**FIGURE 3.4.**Activity decomposition, dependency, and relationship network

### 3.2.2 Deriving spaces from activities

Design guidelines pre-define spaces and describe them generically. If an activity can be decomposed into sub-activities, the space accommodating this activity is also decomposed into sub-spaces such that each sub-space corresponds to one specialized activity. Usually, however, design guidelines do not explicitly deal with activities at the lower levels; rather,

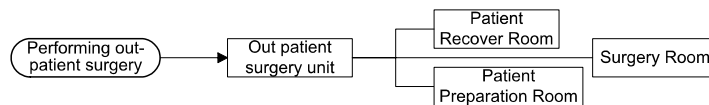
they tend to describe an activity in a broader sense and directly introduce the spaces in which all sub-activities take place.

In a USARC, the educational activity group involves instructional training of unit members. The group decomposes into instructional education and individual study. In instructional education, an instructor trains groups of 25-30 unit members in a single classroom. In the individual study, each unit member studies alone, for example, in a *library*, *reading-room*, or *learning-center*. The USARC design guidelines directly describe the required spaces for instructional education and individual studies without explicitly describing the sub-activities that these spaces accommodate (DG, 1984). However, the space names implicitly indicate which activity the spaces serve. The main spaces accommodating the educational activities are shown in Figure 3.5.



**FIGURE 3.5.** Corresponding spaces to educational activities in a USARC.

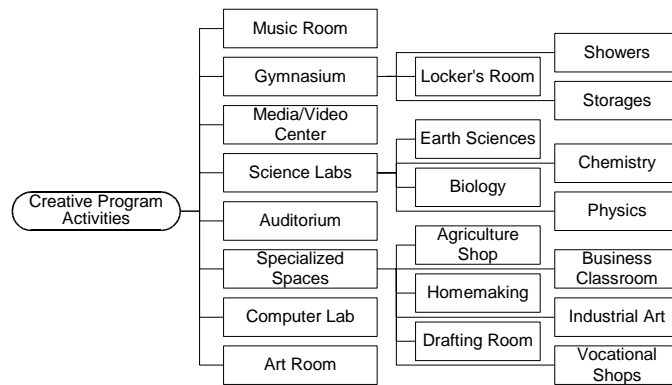
Similarly, the major AHCF activities are broken down into sub-activities that are assigned to respective spaces or zones. For example, out-patient surgery is common to all medical specialties. The space which accommodates this activity is called *out-patient surgery room* and requires a complex surgery setting. The spatial properties of this room are mainly derived from this setting. A patient who undergoes out-patient surgery is observed in a recovery room subsequent to the surgery and then released on the same day of the operation. Therefore, a *patient preparation* and a *patient recovery room* accessible from the *surgery room* are needed and complement the surgery area (Figure 3.6). AHCF spaces are broadly described in the respective design literature, which describes the activities with enough detail so that programmers are able to derive additional spatial needs (Malkin, 1989; 1997).



**FIGURE 3.6.** Spatial decomposition of out-patient surgery area.

The typical spaces in which ESPS activities take place are broadly pre-defined in design guidelines and the literature. For example, the creative group activities in ESPS take place in a specialized program area, which is decomposed into a music room, science lab, art

room, computer lab, gymnasium (if preferred, with a stage), media/video center, and library (Figure 3.7). The library and gymnasium are typically *required* spaces. The others are *recommended* spaces and can be added to the program if certain criteria are satisfied—such as budget, student population, site. In some cases, multiple activities can be accommodated in a single space. For example, a gymnasium can also serve as auditorium. Creative activities additionally may need an agricultural shop, business classroom, homemaking room, industrial art room, technical drafting room (or CAD room), vocational shops (e.g. woodworking, auto repair etc.), and band room. Other specialized spaces, such as a student lounge or parent education classrooms are also required when certain conditions are satisfied, such as budget.



**FIGURE 3.7.** The spaces that the ESPS creative program activities take place

When an activity is needed in a facility, the design guidelines circumvent a detailed investigation of the related sub-activities and sub-spaces and directly provide typical spatial requirements. Therefore, the selection of spaces can become a straight-forward and well-defined process.

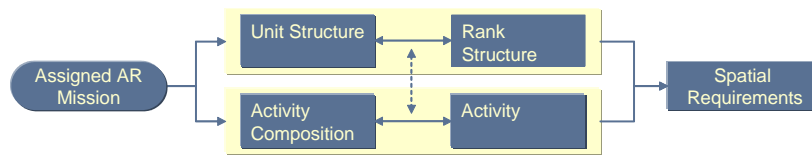
### 3.2.3 Deriving spatial information from organizational structures and building users

Organizational structures and the building's users are two other factors that effect the spatial needs in a facility. Design guidelines typically do not explicitly specify these and very often capture organization or user information through some higher-level parameters—such as the medical specialty, school type, army unit structure etc.—and lower-level parameters—such as the number of doctors, school population, number of unit members etc. In the following, I describe the effects of organizational structure and users on spatial requirements in relation to the activities that the users perform.

An army reserve unit and the ranks of its members are the outcome of the mission assigned to that army unit. The unit structure grows hierarchically as the mission of the unit gets more complex. In response to the changes in the unit structure, the hierarchy of

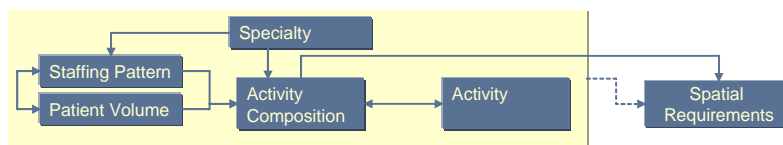


ranks changes. These changes result in more or less complex spatial requirements (Figure 3.8). For example, a training building's capacity is based on a parameter called *rated capacity*, which is the "aggregate authorized strength [number of troops] of all units programmed for assignment to the center" (AR 140-483, pg. 1). This parameter corresponds to the maximum number of reserves that a facility can accommodate at the same time. The changes in the rated capacity and rank structure are reflected in a program through changing values of the respective parameters in *formulas* that are used for deriving needed spaces and their spatial requirements.



**FIGURE 3.8.**Non-spatial factors effecting spatial requirements in a USARC

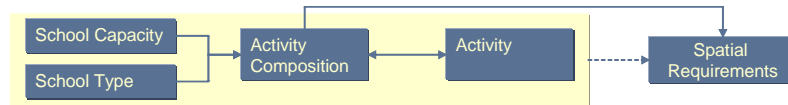
Programming an AHCF can commence either by setting an expected or projected patient volume—which influences staffing patterns—or with a pre-determined staffing pattern—which is used in formulas calculating how many patients can be served in a unit of time, say per hour. Therefore, the patient volume and staffing pattern are highly associated with each other (Figure 3.9). For example, a certain number of physicians with a certain specialty may predetermine the number of patients that can be examined in an AHCF. This association consequently influences the spatial requirements, such as adding rooms to or removing rooms from the program, changing area allocations or number of rooms. Therefore, both patient volume and medical staffing pattern, along with other higher-level pieces of information, play important roles in determining spatial requirements.



**FIGURE 3.9.**Patient-volume and staffing pattern effect the spatial requirements.

The spatial configuration of an ESPS changes depending on the type of a school (elementary, middle, or high school) and student population. For example, in elementary schools, each classroom is reserved for a particular class and grade, and the students only leave the class for creative program activities. In middle and high schools, on the other hand, students are not assigned to a particular classroom; rather, they attend each lecture in a different classroom, which is reserved for a particular subject (math, literature etc.). The students move from one classroom to another during breaks between classes. Therefore, the number of classrooms in an elementary school is given by the student capacity divided

by the class size; but in a secondary school, the maximum school capacity can be calculated by multiplying the average class size with the total number of instructional spaces including classrooms, labs, music room etc.



**FIGURE 3.10.** School capacity and type effect activity composition and change spatial requirements.

---

### 3.3 Program Parameters for Recurring Building Types

#### 3.3.1 Roles of program parameters

Parametric relationships between requirements play an important role in programming because they pose constraints for the generation of requirements. For example, the *number of patients that will be seen by a doctor during a definite time period* poses a constraint for the generation of spatial requirements for waiting spaces, circulation paths and the number of the required exam rooms (Malkin, 1989, pg. 20, 47). The *number of doctors that practice in a particular AHCF* influences design requirements by altering other parameters like the *number of nurses* or *number of exam rooms*.

The relations between parameters are both transformational and generative. They are transformational when a parameter is transformed to other parameter. They are generative when they help to generate other associated parameters or, at the lowest-level, result in design requirements.

The generation and transformation of parameters observed in the programming of AHCFs can also be found in programming other recurring building types. For example, the *number of students that attend school* is used together with standards about recommended class sizes for calculating the *number of classrooms*. In turn, the *number of classrooms* can influence programming decisions about other spatial requirements. By a similar reasoning technique, spatial requirements for each classroom are derived from the *number of students* and the *unit area required for each student*. Multiplication of the *number of students* in a classroom with the *allowable unit area for each student* gives the area required for each classroom.

#### 3.3.2 Calculation of Area Requirements

In general, the area of a space is parametrically derived from factors such as work flow, activities, status recognition of occupants (e.g. the principal of a school or the commanding officer in a USARC), and fittings (e.g. furniture, equipment, devices etc.).

Based on these factors, I divided commonly used methods for calculating the size of spaces into three basic groups.

### Incremental method

In the *incremental* area calculation, the areas required for equipment, furniture and devices and their operations (considering ergonomics and anthropometric data) are added up. For instance, as demonstrated in Figure 3.11, the area that a study desk occupies is, let's say, 13 sqf., and 4 sqf for a chair. The total area for a study desk setting is then 17 sqf. plus 15 sqf. required for the activity itself (a person sits in front of the desk and uses drawers, moves along a side of the desk on a chair etc.). The total area requirement then amounts to 32 sqf. However, the area required for the chair overlaps with the area of operation, and the area that the chair occupies must be subtracted from the 32 sqf. total area. The final figure then is 28 sqf.

As the example demonstrates, this method is analytical. In general terms, the total area requirement becomes the sum of all the areas where sub-activities taking place. The length and width of the required area are also determined through this method. The following algorithm implements this method:

```
for each required space S for an activity
  add S to required space list
  if S does not overlap with a space S' in the required space list
    add area of S to total required area
  else
    add difference of area of S minus area of S'
```

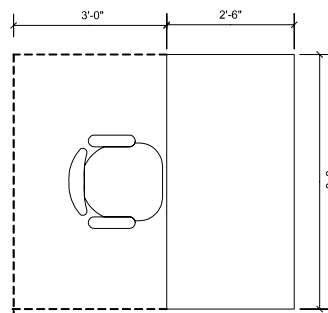


FIGURE 3.11. Analysis of spatial area requirements of a study desk setting

### Choose-and-use method

In the *choose-and-use* method, values for parameters that are related to spatial requirements are picked from well-established sources and plugged into formulas. The literature contains studies aiming at standardizing spatial requirements for different

building types. The provided data are usually based on personal experience (Kobus, 1997), surveys (MCMA, 2001), analytical techniques similar to the first method (Malkin, 1989), or well-established standards (BCHS, 1974-1 and 1974 - 2)(Chiara and Callender, 1990).

Type of practice	Square feet per physician	# of exam rooms per physician
Multi specialty	1497	2.12
Cardiology	1201	1.33
Family practice	1565	3.10
Internal medicine	1500	Not available
OB/Gyn	1653	2.78
Ophthalmology	1527	2.53
Orthopedic surgery	1843	2.32

**TABLE 3.1.** The averages for square footage and number of exam/patient treatment rooms. From MGMA (1999) survey of group practices and space planning.

### Formula-based method

The third method is based on pre-determined and tested formulas. In order to calculate an area, parameters that influence the calculation of a spatial requirement are incorporated into a set of formulas. By assigning values to each parameter in a formula, the required area can be calculated. For example, in order to calculate the area requirement for a waiting space in an AHCF, the number of required seats in the waiting room is first calculated. The number of seats is then multiplied by a coefficient, the *unit area per person* (Malkin, 1989, pg. 27). Formulas to calculate the number of seats ( $nS$ ) and the waiting area ( $aW$ ) can be expressed as follows:

$$nS = 3 \times [(P \times D) - E]$$
$$aW = nS \times A$$

where  $P$  is the average number of patients that a physician sees in an hour;  $D$  is the number of physicians;  $E$  is the number of exam rooms; and  $A$  is a *unit area per person* coefficient, which differs from one specialty to another.

### 3.3.3 Spatial relationships.

The case studies revealed four typical of spatial relations that programmers have to consider. The first one is that a space may need to be located at a certain physical distance from certain other spaces. This can be called a required *proximity*. For example, in school buildings, cafeteria and kitchen must be in close proximity so that the food from the kitchen area can be delivered to the service area located in the cafeteria without any interruption.

The second relation type is *accessibility*; it either limits or permits accessibility from one space to another. For example, in school buildings, only authorized personnel can access service areas (such as kitchen and mechanical rooms); students should not have direct access to these areas. In addition, accessibility can be of other types, such as *physical*, *visual*, or *acoustical*.

A third relation type is *spatial overlap*. This occurs when one space is used for multiple activities. Multi-purpose spaces, such as cafeteria or gymnasium, are examples for this relation. Activities such as ceremonies, presentations, large group meetings, and even some non-school activities can be performed in these spaces.

The final relation type is *containment* (or *constituency*). This occurs when a space contains other spaces. In an AHCF, for example, an out-patient surgery unit contains a preparation room, surgery room, and recovery room. Each contained space may have affinities to any one of the other spaces. For example, a surgery room has to have direct physical access from patient preparation and recovery rooms. The recovery room should be isolated from public spaces forming a proximity constraint between the recovery room and the spaces which have public access.

---

### 3.4 Parameters, Formulas, Logic Statements, and Procedures

#### 3.4.1 Components and constructs.

Requirement information found in the case studies can be composed of two basic groups: *components* and *constructs*.

*Components* are complex information bundles encapsulating related requirement information such as specialty, activity, and spaces. *Constructs* refer to parameters, constants (coefficients), formulas (expressions), procedures, and conditional statements. They are defined in this study as follows:

**Parameters:** A parameter refers to any factor that allows for a range of variations or restricts the result from a procedure or equation. Parameters can be *common* among building types (such as *number of occupants*, *budget*, *activities*, *site*, *area per occupant*) or *specific* to a building type, such as *specialty* for ambulatory health care facilities or *class size* for a public school building.

Parameters can be of different types, such as numerical (e.g. *number of physicians*, *students* or *the unit members*), boolean (true or false), or user-defined types (e.g. private, public, semi-public space use). Parameters can be *independent of* or *dependent on* a component (space, activity or specialty). For example, the *number of physicians* is a parameter which does depend on either a spatial requirement or a specialty, whereas

the *number of patient that a physician sees in an hour* is a parameter that changes from one specialty to another.

**Constants:** These are quantities with fixed values in a specified programming context (such as coefficients). For example, the area required to accommodate a specific piece of equipment or the minimum number of required toilet stalls is given as a constant. As another example, the exam room coefficient is a constant used in calculating the number of exam rooms for different specialties.

**Formulas:** Formulas take the form of equations or rules and are used to establish dependencies between parameters and constants. For example, the *number of exam rooms* ( $nE$ ) depends on the *number of physicians* ( $nF$ ) and the *exam room coefficient* ( $eEC$ ) for a *medical specialty* ( $S$ ). The following formula represents this relation:

$$\begin{aligned} f: & \quad S \rightarrow eEC \\ g: & \quad nE \leftarrow nF \cdot f(S) \\ g(nF, f(S)) &= nF \times f(S) \end{aligned}$$

Similarly, for calculating the administrative support area ( $aAS$ ) considering the number of units ( $nU$ ) in a USARC, the following formula is used:

$$\begin{aligned} f: & \quad nU \rightarrow aAS \\ f(nU) &= 120 \text{ sqf} + \left\lceil \left( \frac{nU}{50} \right) \times 60 \text{ sqf} \right\rceil \end{aligned}$$

**Conditionals:** This construct type represents logical relationships between parameters. For example, if the *number of physicians* ( $D$ ) is greater than or equal to 3, then a complex business office ( $Bd$ ) is required; otherwise a small business office will be sufficient. The first parameter ( $D$ ) is a numerical and the second ( $Bd$ ) a logical boolean (binary) parameter. The conditional statement can be expressed as follows:

$$if(D \geq 3) \begin{cases} Bd = true \\ Bd = false \end{cases}$$

**Procedures:** A procedure is a sequence of instructions that perform a specific decision-making task in programming (such as calculating the *number of classrooms in an elementary public school*). These constructs encapsulate one or many formulas and logical statements in a package that manipulates multiple parameters at a time. For example, the following formula can be used to calculate the *number of toilet stalls* in an AHCF:

$$\begin{aligned} f: & \quad S \rightarrow Tc \\ g: & \quad S \rightarrow Fx \\ h: & \quad TS \leftarrow S, D \\ h(S, D) &= g(S) + (D \times f(S)) \end{aligned}$$

where  $TS$  is the number of toilet stalls;  $Fx$  is the minimum number of toilet stalls required in medical office with a particular specialty  $S$  (expressed as function  $g$ );  $D$  is the number of physicians and  $Tc$  is a specialty-dependent coefficient (expressed as function  $f$ ).

However, this formula is not sufficient enough in most of the cases, such as for a urology clinic, for which the number of toilet stalls can be calculated with the following procedure incorporating the formula shown above.

$$u: \quad TS \leftarrow S, D$$
$$u(S, D) = \begin{cases} TS = 2 & D \leq 3 \\ h(S, D) & \text{else} \end{cases}$$

If the number of doctors is smaller than or equal to 3, then 2 toilet stalls are needed, else use the general method in calculating the number of toilet stalls as expressed in formula  $h$ .

This procedure is verbally described by Malkin (1989). I converted the description into pseudo-code as written above. Similarly, other parameters, formulas, procedures etc. can be derived from verbal statements in the design guidelines or literature related to programming a particular building type.

Constructs can be either encapsulated in a component or stand alone. For example, the *number of physicians* does not depend on any component (i.e. specialty, activity, or space), whereas the *number of patients that a physician sees in an hour* is a parameter that changes from one specialty to another. Similarly, the number of students and school type are independent parameters in programming an ESPS. But the required unit area per student is a dependent parameter because its value changes from one type of instructional space (such as lab or classroom) to another.

### **3.4.2 Graphical representation of component-construct relationships**

I represent the relationships between components and constructs graphically by using the symbols shown in Figure 3.12. In this representation, activities are shown as shaded rectangles with rounded corners and spaces are shown as rectangles with dark borders. Independent parameters are shown with the parameter name in a rectangle and a parameter symbol in a circle attached to the rectangle by a line. Dependent parameters and constants are shown as rectangles with light borders and attached to the related components by a line. Formulas and procedures are also attached to components by lines, but the line has an arrowhead pointing to the component to which they relate. All constructs are represented as rectangles with light border lines. The diamond-head arrow represents spatial aggregations (or spatial constituencies).

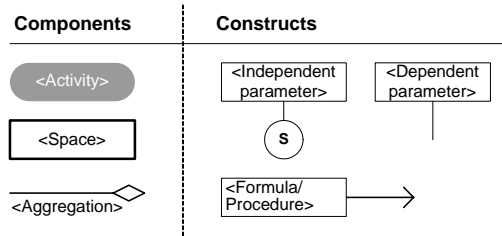


FIGURE 3.12. Graphical symbols used in the representation

An example of this graphical representation is shown in Figure 3.13. The figure represents partially the spaces accommodating the organizational maintenance activities for USARCs and how their physical properties are calculated through attached constructs. The program components and constructs are derived from the army building design guidelines and design manuals (see Appendix A). Similarly, Figure 3.14 and Figure 3.15 show partially how the components and constructs in AHCF and ESPS relate to each other. Each of these diagrams is explained in detail in Appendix A.

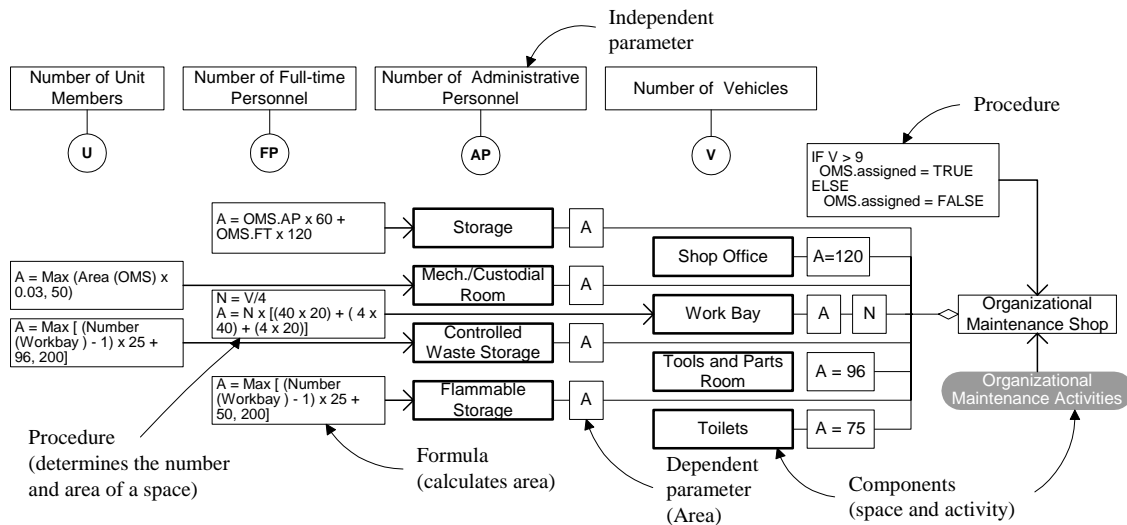
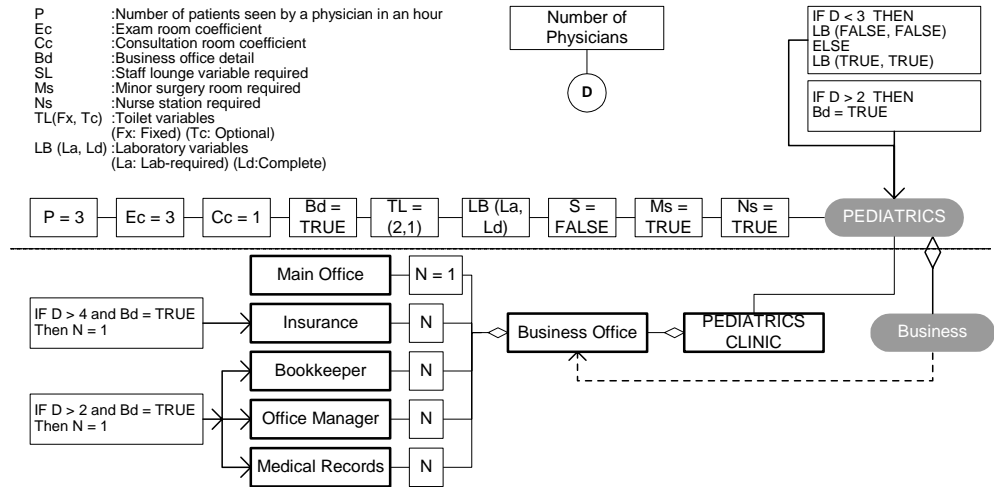


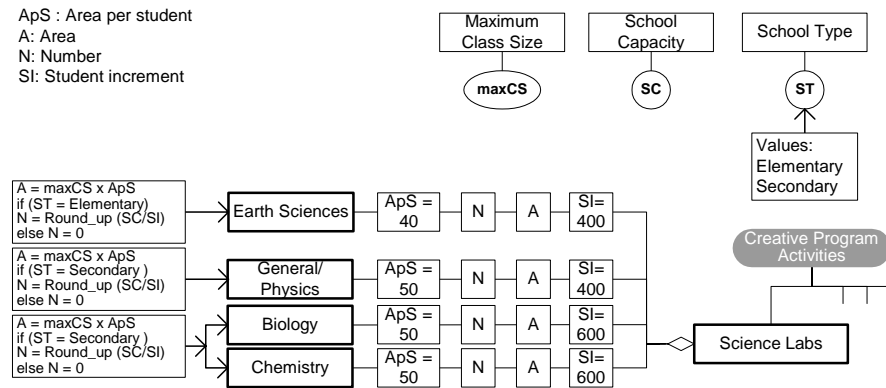
FIGURE 3.13. The graphical representation of components and constructs relationships in USARC programming.



## Observations and Summary



**FIGURE 3.14.** The graphical representation of components and constructs relationships in AHCF programming.



**FIGURE 3.15.** Components and constructs in ESPS programming.

## 3.5 Observations and Summary

Four general observations emerged from the case studies.

1. Programming recurring building types is similar to programming non-recurring building types in that design requirements at *higher-levels* are gradually refined into *lower-level* design requirements. Higher-level design requirements relate to abstract information, such as activities, organizational structure, occupant needs etc. At lower-levels, the spaces, physical properties of the spaces, spatial relationships are derived.
2. The information used in generating higher-level and lower-level design requirements (such as those regarding the client organization and its needs) is relatively predefined

and well-structured for recurring building types. Therefore, the refinement process in programming recurring building types can be less labor-intensive than for non-recurring building types. For example, the commonalities in spatial needs of school facilities can provide ready-to-use data. The organization of the school activities and building user characteristics do not need to be rediscovered every time a new public school design is needed. The only differences between ESPS projects may be in the school type, school (population) capacity, and some other contextual constraints (such as budget, site, climate etc.).

On the other hand, in programming a community center—as a non-recurring building type example—most of the needed information is not available in advance. Therefore, programmers have to *discover* the project goals and objectives based on the unique needs of the community (building users), activities to be accommodated, functions of the facility, spatial needs etc.

3. Requirements specifications are multi-directional. That is, one non-spatial requirement can lead to one or more spatial requirements as well as one or more non-spatial requirements. Similarly, a spatial requirement may generate additional spatial requirements or may cause other related non-spatial requirements to be reviewed.

Take a school building as an example. One non-spatial requirement could be that as part of the curriculum, *chemistry experiments* will be conducted. This requirement can generate other requirements such as *direct access from labs to all of the classrooms*. In addition, conducting chemistry experiments leads to the generation of multiple design requirements for a laboratory space with specific equipment, dimensions and layout. As other non-spatial requirements (such as scheduling the laboratory hours, number of students planned to be accommodated in school and budget limitations etc.) are considered, the resulting web of requirements and their influences on each other become hard to manage. The problem of defining this web becomes a considerably challenging task when the requirements are studied in their entirety.

In conventional programming, generation of requirements and propagation of requirement changes are manually performed. This potentially reduces the integrity and effectiveness of the requirements. Besides, even with manuals and written standards at hand, it is very labor-intensive to manage literally hundreds of dependencies that need to be observed between *building components* and *their underlying intent*. Programming professionals are able to manage these only due to their sustained experience with similar problems over years. On the other hand, I demonstrated that systematic and logical operations (methods) are at work during this process, such as *generate-propagate-update* cycles, which can be automated.

4. Different design requirements are based on different types of information, and each type of information is handled through disconnected representations. Working with disconnected information negatively affects the efficiency and effectiveness of generating design requirements. If a certain parameter at any given level changes, changes in other parameters at different levels have to follow. Take, for example, changing the *projected number of students* in a ESPS; a change in this parameter not only dramatically changes quantitative spatial requirements, but also causes certain activities to be *added to* or *removed from* the program. Current representation techniques are not seamlessly capable of accomplishing such change propagation.

Another example is the representation of requirement relationships. Relations of spaces can be represented in different formats, such as affinity matrix, diagrams, and lists. Each of these representation techniques shows the same concept, and a change in one of these has to be carried to others. Therefore, manual changes or updates of design requirements are frequently required. Maintaining up-to-date information (data) consistently becomes very tedious.

5. As the case studies have demonstrated, the process of generating lower-level requirements from higher-level requirements for programming recurring building types can be very structured, but at the same time, can be very complex. When other complex issues (such as equipment usage, engineering, codes, regulations) enter, the need for more enhanced generative methods for design requirements specification becomes even more pronounced. Similarly, the case studies demonstrate that the relatively well-structured nature of the process of programming recurring building types presents opportunities for supporting this process in a computational environment.



## Chapter 4 Conceptual Framework

---

### 4.1 Programming as Information Refinement Process

#### 4.1.1 Step-wise refinement of requirements

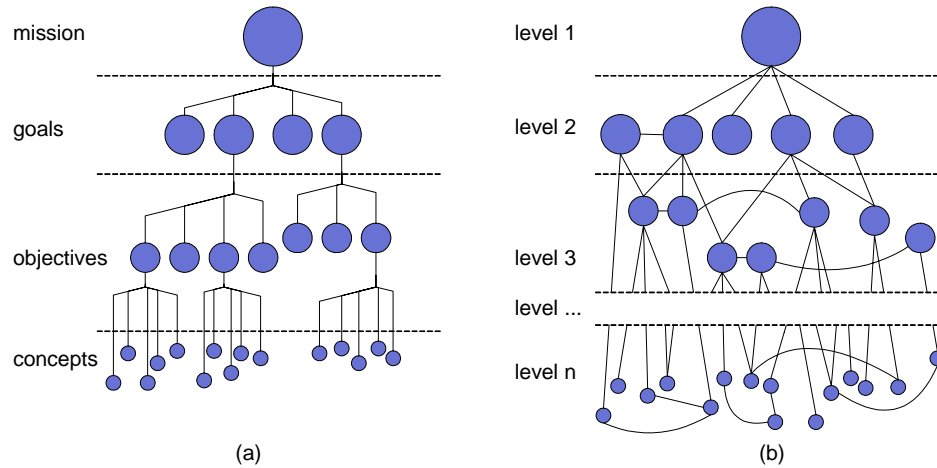
The case studies coupled with the literature review demonstrate that programming—in essence—can be generally characterized as an information refinement process where higher-level (non-spatial) requirements are gradually transformed into measurable and operational (spatial) requirements at the lower-levels. This transformation process is exemplified by various authors (Duerk, 1993, pp. 20, 36; Cherry, 1998, pp. 122; Hershberger, 1999, pp. 367; Kumlin, 1995, pp. 144). I generalize these approaches in the present chapter in the form of an open-ended and extensible model that can be used for any programming case, rather than only for a particular approach.

Figure 4.1a. shows the model in an abstract form. At the highest level, the main goal of a project, which is not refined at this stage, is depicted by a large circle. At each successive level, pieces of requirement information are recursively detailed to a granularity that provides enough information to specify requirements at the next level. The diagram uses smaller circles to show the refined information at each refinement level. The relationships between requirements are shown by connecting lines—which may refer to dependency, data use, parameter transformation, information generation etc.

#### 4.1.2 The proposed model vs. existing models

The main difference between the proposed model and the current approaches—such as Duerk’s (Duerk, 1993, pp. 20)—is that it does not require strictly hierarchical step-wise refinement; different design requirements at different levels can compose a web of relationships (Figure 4.1 a and b). However, strictly hierarchical refinement represents a special case that is also covered by the model. In addition, unlike current approaches, the proposed model provides a flexible level structure that does not fix the number of

information (refinement) levels; it also remains flexible with respect to terminology and degree of resolution, which remain under the programmer's control (Table 2.1).



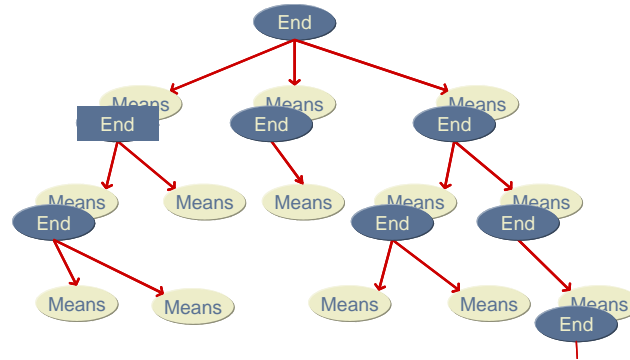
**FIGURE 4.1.** The strictly-hierarchical refinement model (a) vs. the proposed model (b).

Purely hierarchical refinement models allow only one-to-many relationships and fail to comprehensively represent webs of not strictly hierarchical dependencies and relationships. The need for these becomes apparent when we look, for example, at how spatial affinities are established among multiple spaces, which are the outcome of multiple requirements. In a more concrete example from ESPS programming, the *number of students* influences multiple requirements such as the *staffing pattern* and *number of classrooms*. Furthermore, the *number of classrooms* is not the only outcome of the *number of students*, but other requirements as well, such as *school type*—which, in turn, is derived from other requirements (see Chapter 3 and Appendix A for more details).

## 4.2 The Model in Relation to Means-Ends Analysis.

### 4.2.1 Means-Ends Analysis (MEA)

The proposed model resembles Means-Ends Analysis (MEA), a problem-solving method first introduced in the General Problem Solver (GPS) (Newell & Simon, 1963; Simon, 1989, pp. 36-37). MEA involves solving problems by successively reducing the differences between an *initial* and a *desired state* (Sternberg, 1996, pp. 483). In addition, the fundamental MEA strategy for solving a problem is to decompose the solution process into a series of steps each of which comprises its own initial and desired states and operators (Figure 4.2).



**FIGURE 4.2.**Decomposition of a problem into successive means and ends

A MEA problem solver starts by applying *means* to an *initial state*, which generates many *desired states* as new *ends*. These ends are then taken as initial states for the next refinement step. By applying associated *means* to the present initial states, the problem solver generates desired states at the next lower level. This step-wise resolution of means and ends recursively continues in a fashion that turns *means to achieve a higher-level goal* into *ends at the next level*. The process stops at the lowest desired level where required *ends* are derived.

#### 4.2.2 Generalized MEA

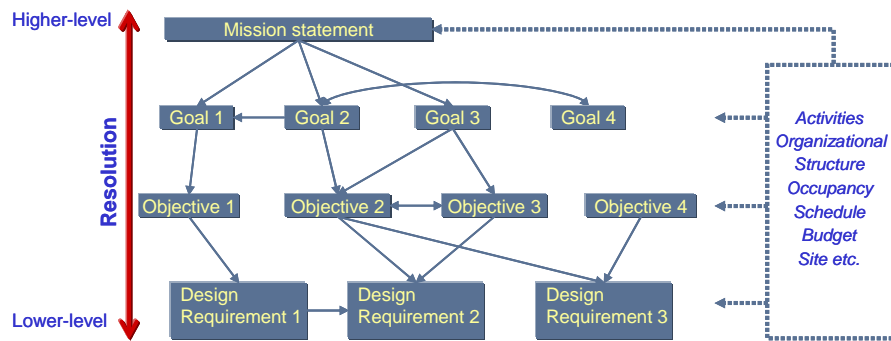
In (architectural) programming, *ends* at each level can be achieved by more than one *means*. Unlike in MEA, the transition from higher-level to lower-level requirements is not necessarily a strictly *hierarchical search* for a solution; it is rather a *multi-directional generative* and *hierarchical refinement process*. This multi-directionality creates possible cycles in the process. In order to manage these cyclic relationships among requirements, MEA needs to be extended so that one means can satisfy different ends. Therefore, the model presented here forms an extended (generalized) version of MEA (GMEA). GMEA has the following potential advantages: (a) it is general enough to capture all processes observed in the case studies or described in the literature, and (b) it is operational enough as basis for formalization and computer application.

#### 4.2.3 Example Scenario

In order to demonstrate the applicability of the proposed framework to established programming methods, I describe a scenario that adapts Duerk's (1993) method to derive missions, goals, objectives, and design requirements.

The process, as shown in Figure 4.3, starts with a *mission statement*, which can be divided into specific *operational goals*. However, the mission statement is not the only source of

generating goals because goals themselves can generate other goals; for example, *Goal 1* is generated mainly from the mission statement, but also in part from *Goal 2*. At the same level, *Goal 2* leads to the generation of *Goal 4*. In the next step, each goal is expressed in terms of specific *objectives*, such as performance requirements. These objectives become, at a lower-level, specific *design requirements (concepts)*, such as spatial properties. In deriving one *requirement*, information provided in multiple requirements may be used. For example, *design requirement 3* is generated using the information provided in *objective 2*, *objective 3*, and *requirement 1*.



**FIGURE 4.3.** Sample transition from high-level to low-level requirements

As illustrated by the diagram, there exist inter-requirement relationships between both higher and lower levels at each refinement step. A *goal* may generate another *goal*; an *objective* may generate another *objective*; and a *design requirement* may generate another *design requirement*. The activities, organizational structures, occupancy etc. are incorporated in the refinement process as requirement information.

When we apply GMEA to the structure described above, an *end* can be the *main goal* which can be *achieved* through the *sub-goals* as *means*. As the sub-goals become ends, the *objectives* become *means* to achieve them. At the following level, the *design requirements* become the means to achieve the objectives. Design requirements are specific and mostly spatial. Taken together, they specify concretely how the main goal (mission statement) can be accomplished by a proposed facility. By employing means-ends reasoning at each level, the programmer thus generates the lowest-level design requirements step-wise from the high-level ones.



---

## 4.3 Applicability of the Framework

### 4.3.1 The framework and programming recurring building types

Note that Duerk's method—like the methods proposed elsewhere in the general programming literature—is not restricted to recurring building types and intended to be generally applicable to programming. This means that the proposed model is also applicable to non-recurring building types. I will return to this at the end of the present section.

Programmers dealing with a recurring building type can take advantage of some *short-cuts*—if we may call them thus—extracted from design guidelines or gained through experience. The programmers can use the short-cuts to compile lower-level program requirements directly without the need of investigating a chain of intermittent requirements.

The short-cuts are fundamentally pieces of *reusable* program information in the form of *programming rules* derived from established *knowledge* and *expertise* about the organizational structures, functional requirements, and their physical implications for a recurring building type. A programming rule for programming an ambulatory health care facility, for instance, is the requirement of assigning a separate consultation room for each physician in internal medicine (i.e. the number of consultation rooms is equal to the number of physicians in an internal medicine clinic.) This is derived from the facts that (a) physicians need private offices, and (b) they consult with their patients before or after the examination, which is confidential and cannot take place in other spaces. Therefore, the established knowledge and expertise with this building type reduces programming complexities, and the rules provide fast and systematic decision-making mechanisms.

### 4.3.2 The framework at work: a partial example for a recurring building type

When we apply the framework to programming a school facility, the steps of a suitable *hierarchical refinement process* can be—partially—described as follows:

- The main goal (mission) is stated as end.

**Example:** *Design an elementary public school for 350 students in the State of Ohio.*

- The goals are stated as means to achieve the main goal.

**Example:** *In the school, the students should be grouped as specified in the Ohio School Design Manual (OSDM), which states that the students have to be distributed evenly in separate classrooms.*

- The means become ends, and sub-goals (objectives) are stated as means to achieve the higher-level goals. In the following, three sub-goals are derived from four rules taken directly from guidelines in the Ohio Design Manual.

**Rule 1:** The class size should not be larger than 25 students per class.

**Rule 2:** If there are more than 5 classrooms, they should be clustered in groups of maximally five.

**Rule 3:** The unit area per student in classrooms should be a minimum of 30 square feet and a maximum of 35 square feet.

**Rule 4:** The circulation area required in a classroom cluster should be at least 30% of the total area of the classrooms.

Application of these rules leads to the following spatial requirements as new lower-level ends.

**Example:** *Provide three clusters of 5 classrooms to accommodate 350 students.*

**Example:** *Each classroom should be 750 square feet.*

**Example:** *In each cluster, the circulation area should be 1125 square feet.*

The example shows how established programming knowledge and its capture through programming rules enable programmers directly to derive lower-level design requirements by just evaluating certain *critical programming parameters*. Based on these parameters, a programmer is able to derive quickly basic functional requirements of a facility and the physical attributes of its spaces.

The rules are based on intermittent goals and objectives that are not stated explicitly (like the rationale behind a maximum class size) and apply these to the critical parameters to derive directly lower-level specifications. Programmers who use the rules do not need to engage in extensive research to investigate intermediate functional and physical requirements of the facility. This would indeed be practically infeasible for any given project.

However, I do mean to imply that the goals and objectives implicit in guidelines for recurring building types should never be questioned. It is indeed the responsibility of their authors to monitor their implications on a continuous basis. All I want to point out is that they are ubiquitous for recurring building types, provide shortcuts for programmers, and can be incorporated into the proposed framework and captured by the components and construct concepts introduced in the preceding chapter.

#### 4.3.3 Framework at work: non-recurring building type test case

I wanted to test to what extent the model captures the programming information for non-recurring building types. For this purpose, I demonstrate in the present section how the abstract concepts mentioned in the framework can be mapped onto a real-world example: the program for the design of the National Humanities Center at Raleigh, NC, a unique facility for which no precedents existed at the time of its inception<sup>1</sup>.

The *mission statement* is a brief statement of the specific purpose of the project.

**Example:** *The mission of the project is to create a National Humanities Center which will house 40-50 scholars in residence and allow them to explore common issues from a variety of points of view.*

**Goals** are statements about the level of quality that is desired in the final project, yet general enough to be inclusive of a wide set of performance and design requirements.

**Example:** *The center should encourage interaction between scholars to stimulate new ideas and collaboration, and to provide new information and perspectives about issues.*

**Example:** *Each scholar should be provided with a setting where the scholar can isolate him or herself from other public or semi-public activities and concentrate on his or her study. The scholars should feel that they own this setting and be encouraged to modify as they wish.*

**Objectives** are performance requirements which define the measurable level of functions that a design must provide. Measurements can be binary, scalar, judgements or consist of an acceptable range of physically measurable values. Goals are refined into specific and operational objectives.

**Example:** *The researchers meet periodically to exchange ideas. For the meetings, formal and casual meeting settings (spaces) should be provided. The formal meeting spaces should provide privacy for the group as well as adequate spatial area to make the space reconfigurable for different needs (presentation, group discussion, hands-on training etc.). The meeting spaces should also have access to the spaces which store furniture and equipment.*

**Example:** *Each scholar conducts research individually. In addition, a scholar may invite other people to discuss their research in a private setting. Therefore, individual researchers should be given their own spaces where they can modify the space for their needs. These spaces should provide adequate visual stimulation without being distracting.*

---

1. The program was implemented and the design of the center was published in Architecture+Urbanism No. 135 December 1981 issue.

*Design requirements* are statements directly referring to the physical (spatial) characteristics of the building to be designed. Like performance requirements, these are more specific and measurable than goal statements and constitutes the lowest-level and spatial (or physical) requirements.

**Example:** *One seminar room for organized group meetings of at least 25 occupants should be provided. The seminar room should be square and allow flexible furniture layout and sitting configurations. The area requirement for the seminar room is minimum 30 sqf. and maximum 40 sqf. unit area per person.*

**Example:** *The seminar room should have access to a storage where chairs, tables, and presentation equipment are stored. The area for the storage should not be less than 10% or more than %20 of the seminar room.*

**Example:** *For each scholar, a private study area accommodating a 3' x 3' meeting table with three chairs, a 3' x 4' study desk, and two 3' x 8' shelves to store books should be provided. The furniture may be replaced by other furniture or the room layout may be changed by the individual scholar who occupies the room.*

**Example:** *The area required for each individual study room is 175 sqf. (possibly 12' x 14.5')*

These portions of the program document illustrate precisely the step-wise refinement process postulated by the proposed framework. The case therefore demonstrates that the proposed framework can also be utilized for capturing the information refinement process for non-recurring building types.

---

## **4.4 Summary**

In this chapter, I brought together my observations on architectural programming from both the case studies and the literature review. I redefined programming—for the purposes of this research—as a hierarchical information refinement process resembling MEA to accommodate multi-directional and not strictly hierarchical generative and configurative processes observed in programming.

I also demonstrated by example that the framework can capture programming for both recurring and non-recurring building types. Furthermore, I conclude that the proposed framework can be adapted for a computer-aided architectural programming tool because of its generality and formal structure.

## Chapter 5 System Definition: Features and Requirements

---

### 5.1 Architectural Programming System

#### 5.1.1 RaBBiT

The system that is developed as part of this research is called *RaBBiT*, which stands for *Requirements Building for Building Types*. I found the *requirements building* metaphor appropriate for the purpose of the proposed application because (a) the word "*building*" is commonly used in both construction and software engineering, and (b) the notion of *building* alludes to the most basic activity that the system is envisioned to facilitate: modeling building requirements in a structured form<sup>1</sup>.

#### 5.1.2 User characteristics

The target users of RaBBiT are those with a particular interest in architectural programming for specific building types. Specifically, the primary users are architectural programmers playing one of two roles: *architectural programming knowledge modeler* (APM) or *program composer* (APC). An APM will *design* type-specific knowledge models in RaBBiT; an APC will *use* the knowledge model to *generate* and *compose* a program for a particular project with RaBBiT's assistance. After a program is generated, the APC should be able to modify the generated program.

The primary users of RaBBiT are not assumed to be experts in *computer programming*. Therefore, other than writing basic mathematical expressions—such as the ones used in spreadsheets—an APC and APM should not be required to write computer programming code.

The secondary users are other CAD reasoning systems (users) and clients who want to utilize a model or a generated program in their own domain of interest—such as layout generators or budget planners.

---

1. The word "building" is defined in [www.dictionary.com](http://www.dictionary.com) as "*to form by combining materials or parts; to develop or give form to according to a plan or process*"

### 5.1.3 Basic functionality of the system

The basic function of RaBBiT is to provide interactive support for programming-knowledge modeling of and program generation for any recurring building type. Knowledge modeling will involve formulating and composing building requirements as a collection of re-usable and type-specific concepts and their relations. These concepts will be used in generating program information for a given project. Computational representations of the model and generated programs can be (a) shareable with other applications; (b) viewed—displayed and documented—in a format of a user’s choice, i.e. they are view-independent; and (c) persistently storable in and transferable between computers through network connections. Figure 5.1 shows an overview of the system functions.

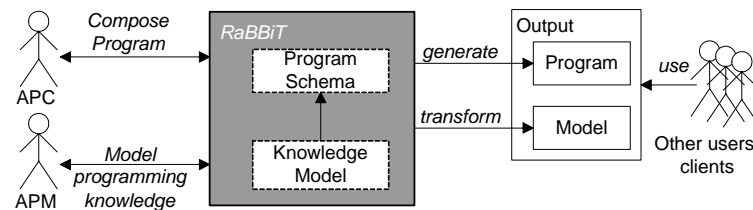


FIGURE 5.1.Overall system structure: users and functions

---

## 5.2 Knowledge Modeling Features and Requirements

### 5.2.1 Overview

The following sections outline the structural and behavioral requirements of RaBBiT for programming-knowledge modeling. These features are derived from the framework—the programming model and GMEA as introduced in Chapter 4—and are essential for guiding the system design. The structural requirements define the types of information (abstractions) that should be represented in the model; the behavioral requirements specify how the model should respond to any possible change. Note that these requirements are intended to guide the design and implementation of the system, not to describe the system in detail—later sections will be more specific.

### 5.2.2 Structural requirements

1. RaBBiT should assist programmers in assembling a *programming-knowledge model* that captures (building) type-specific programming information and parameters. Part of this information relates to activities, user characteristics, organizational structure, spaces, equipment etc. The model should be flexible enough to accommodate all possible types of information as observed in the case studies and found in the literature. The model should comprise the following information types (blocks)<sup>1</sup>.

- Programming information relating to a *salient* programming concept or a requirement should be captured in an information type called *component*. For example, a component can represent a space, a piece of equipment, a function, or a particular user. Each component should have a description and a name designating the concept or requirement it represents. Components should be able to contain parameters of arbitrary type and number, such as numeric, text, boolean, association, digital file links (e.g. to graphic images or internet addresses) etc. The order and number of parameters in a component should be determined by the users and not be imposed by the system. Components provide programmers with a data type generic enough to represent any concept used in architectural programming.
  - Parameters will be specified in an information type called *construct*. Each construct should have a name, data type (numeric, boolean, text etc.), and a value. The value of a parameter can be directly assigned by the users or derived from a function. It should be possible to specify relationships between parameters, or more precisely, between their values so that the value of one parameter can be derived from the value of other parameters. These dependencies will be called *parametric associations* in the following.
  - The case studies clearly point to the crucial role played by certain critical programming parameters for each building type. They constitute main decision nodes from which decisions are derived when composing an architectural program for a recurring building type. For example for programming an USARC, the number of unit members forms a critical parameter. The critical parameters in the knowledge model should be represented in a special information type called *global construct*. This type will ensure that the critical parameters can be represented independent from other information types and accessed without the need for an extensive search in the model. The programmer will enter the values of global parameters when generating programs for a given project or exploring alternative programs for the same project.
  - An association between components can belong to one of two information types: *dependencies* or *relationships*—called, respectively, *dependency* and *relational associations* in the following.
2. RaBBiT should provide a feature for categorizing programming concepts and requirements—contained in *components*—according to the information levels they belong to. The number of information levels and the name of each level (category) should be determined by the user and not be imposed by the system. This feature can

---

1. The naming conventions used here are intended only to make the system requirements easy to understand. They are internal to the present document and not necessarily intended to be used by the users or the system. During system design, these names can be used or changed as desired.

be used to sort requirements information into semantically related groups, such as defined in a particular (architectural) programming approach or in design guidelines. If the programmer chooses not use this function, RaBBiT should grant this choice.

3. The knowledge model in RaBBiT should be able to capture *functional*, *conditional*, and *nested associations* between pieces of programming information and parameters associated with them, including the parametric associations introduced above.
- **Functional associations** are an important structural feature that defines dependencies between pieces of program information, particularly between parameters. Take the simple rule in calculating the number of exam rooms in an AHCF as an example. In this rule, there is a functional association  $f$  between the number of doctors ( $nD$ ) and number of exam rooms ( $nE$ ) such that the number of exam rooms ( $nE$ ) is equal to the *ceiling* of the number of doctors ( $nD$ ) multiplied by a medical specialty coefficient ( $C$ ), i.e.  $nE$  is a function of  $nD$ .

$$\begin{aligned} f: \quad nD &\rightarrow nE \\ f(nD) &= \lceil nD \times C \rceil \end{aligned}$$

- **Conditional associations**<sup>1</sup> capture conditional dependencies between different pieces of program information. A conditional association is simply an *if-then-else* (If <conditions> then <actions> else <alternative actions>) statement or rule that establishes a conditional association between parameters. The *if* part or *antecedent expression* defines a condition that must be satisfied for the association to exist. It evaluates to *true* or *false*. The *then* part or *consequent expression* describes the action that must be performed if the antecedent expression holds true. The (optional) *else* part of the rule is similar to the *then* part; but it is processed when the antecedent expression evaluates to false.

As a simple example, assume that the area for the cafeteria ( $aC$ ) in a school building can be calculated based on the number of students ( $nS$ ) such that if the number of students is less than  $x$ , the area per student is  $k$  sqf., else it is  $m$  sqf. This association can be expressed by the following function  $f$ :

$$\begin{aligned} f: \quad nS &\rightarrow aC \\ f(nS) &= \begin{cases} nS \times k & nS < x \\ nS \times m & \text{else} \end{cases} \end{aligned}$$

---

1. The meaning of the term "conditional association" differs here from that used in statistics.



Conditional associations can also be used for decisions that change depending on whether or not a set of parameters satisfies certain conditions. For example, in programming an USARC, the following dependency condition can be used: An army reserve unit is given organizational maintenance mission (*OMM*) if 10 or more motorized vehicles (*nV*) are assigned to a USARC. If that becomes the case, program for that USARC should have organizational maintenance shops (*OMS*), else *OMS* are excluded (i.e. do nothing).

$$f: \quad nV \rightarrow \text{include } OMS$$

$$f(nV) = \begin{cases} \text{true} & nV \geq 10 \\ \text{false} & \text{else} \end{cases}$$

- **Nested associations** contain multiple functional and conditional associations configured such that they are evaluated together. For example, assume in the example given for the conditional association above that in addition to the mission, the number of officers (*nO*) becomes a factor in deciding whether or not we should include *OMS*. The number of officers in turn is a parameter with a functional association *f* to the number of unit members (*nU*). The project budget (*B*) is also a factor in effecting this decision such that it is compared to the actual cost (*Ca*) of the *OMS*: if the budget is greater than (*Ca*), *OMS spaces* are added to the program. A function *h* is used for calculating the total cost by adding the cost of each space *s<sub>i</sub>* given in a set of *OMS* spaces (*S*). The cost of each space is calculated by a *cost* function. The following statements describe the *nested association* (*g*) among these parameters and associations.

$$f: \quad nU \rightarrow nO \quad f(nU) = \lfloor (nU)/50 \rfloor$$

$$h: \quad S \rightarrow Ca \quad h(S) = \sum_{i=1}^{s_i \in S} \text{cost}(s_i)$$

$$g: \quad \text{include } OMS \leftarrow nV, nU, S, B \quad g(nV, nU, S, B) =$$

$$\left[ \left[ \left[ \begin{cases} \text{true} & B > h(S) \\ \text{false} & \text{else} \end{cases} \quad f(nU) > k \right] \quad nV \geq 10 \right] \right]$$

$$\left[ \begin{matrix} \text{false} & \text{else} \end{matrix} \right]$$

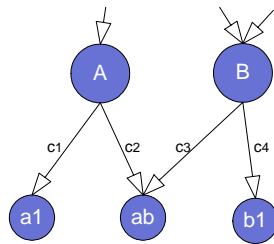
The expression *g* can also be  $g(nV, nU, S, B) = (nV \geq 10) \wedge (f(nU) > 10) \wedge (B > h(S))$

4. RaBBiT should provide mechanisms to specify *dependency associations* from higher-level requirements to lower-level requirements—or programming concepts for that

matter. This type of association is mainly needed to establish a requirement decomposition mechanism from higher-level information to lower-level information such that the resolution of the requirements can increase at the lower-levels.

From the perspective of the general framework introduced in the preceding chapter, the dependencies determine *which ends can be achieved with which means*.

A *dependency association* should be specified by an information type having references to a *source* and a *target component* and a *condition*. In Figure 5.2, assume that *A* and *B* are higher-level requirements and are achieved by lower-level requirements *a1* and *ab*, and *ab* and *b1* respectively. For *A* to be achieved by *a1*, the condition *c1* on this dependency must be *true*.



**FIGURE 5.2.**Sample dependency associations with and conditions

- The source component represents an *end* and the target component a *means* to achieve this end. One component can be a *source* for as well as be a *target* of many dependency associations. For example, in Figure 5.2 *ab* is a means to achieve both *A* and *B*.
- Each dependency should accommodate a *conditional association* that is used for evaluating whether or not a dependency is valid in a given situation. This feature is needed in cases where a requirement can be achieved by another requirement if a certain condition is satisfied.

As a real example in AHCF programming, we want to include a minor surgery room if the given medical specialty is *internal medicine*—or any other specialty that requires a minor surgery room for that matter. Therefore, the spatial requirements will include a minor surgery room if the specialty condition is satisfied, else the room will be excluded—and all the other lower-level requirements depending on this room. In this example, the *source* of the *dependency association* can be a component describing a spatial zone, the *target* is a component containing information of the surgery room, and the condition of the dependency is "*specialty should be equal to internal medicine*". Result of the condition will be used to decide if the zone should include this room or not.

- The conditional association in dependencies become especially useful when we want to make decisions based on the possible existence of certain other requirement in the program. Assume that in a school program, two higher-level (activity) goals have been created: (a) *extracurricular student activities* and (b) *community and parent education* should be accommodated (Figure 5.3). The student activities can be accommodated in two alternative ways; (1) in a *multipurpose hall* that can be used for *student-body meetings, graduation ceremonies, or dining* or (2) in separate spaces, an *auditorium* for student-body meetings and ceremonies, and a *cafeteria* for dining.

The selection of one of the alternatives for accommodating student activities can be based on some conditions—such as if the number of students is greater than a certain number, include a cafeteria and an auditorium, else include a multipurpose hall. Community education, on the other hand, can take place in the same multipurpose hall if the decisions about extracurricular activities have been made. Conversely, if community education is considered before extracurricular activities, a multi-purpose hall will be created, and when the extracurricular activities come under consideration, the existence of this space has to be taken into account.

This example illustrates that when we have dependencies among conditions that emerge dynamically during programming, the order in which the decisions are made matters, and conditions must be formulated such that the right decisions are always made. One implication of this example is that conditional expressions must be able to assert the existence or non-existence of certain components in the current state of the program. More generally, we have to conclude that cross-dependencies make the programming process non-monotonic; i.e. conditions that do or do not exist at one point may or may not exist later on. I will return to this issue in the next chapter.

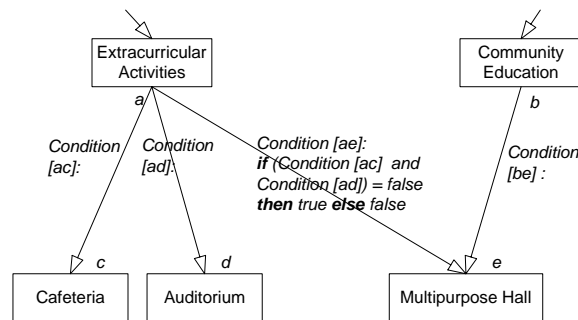


FIGURE 5.3. Complex dependency associations

5. RaBBiT should allow for *relational associations* between different requirements that belong to the same category or group: spatial affinities are such a group. For example, as shown in Figure 5.4, assume that *ab*, *a1*, and *b1* are rooms such that *a1* has an

*adjacency* relation to *ab* and a *proximity* relation to *b1*. RaBBiT should enable users to assert such relationships in the knowledge model.

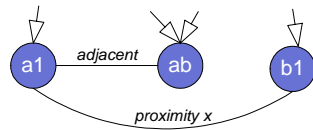


FIGURE 5.4. Sample relational associations expressed with labels

Figure 5.5 shows the information types outlined in the structural requirements above. A component typically contains multiple constructs. Each construct can have at most one parametric association that can be used to derive its value. Parametric associations can be of functional, nested, or conditional. A nested association may have references to many other parametric associations. A relational association keeps reference to two components and is expressed by a label. A dependency association also has references to two components, but unlike relational association, it specifies the direction of the dependency through target and source components to capture means-ends relations. Each information level may contain multiple components for the purpose of grouping (classification) of pieces of domain knowledge information.

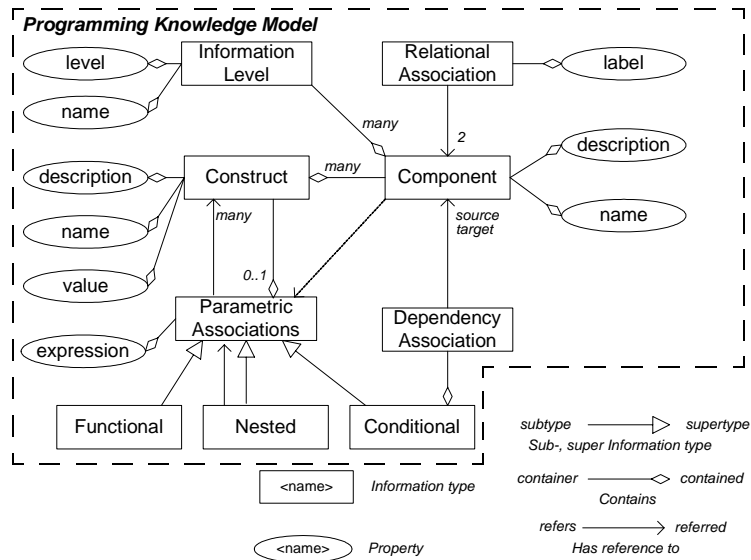


FIGURE 5.5. Structural requirements for programming-knowledge model

### 5.2.3 Behavioral requirements

1. RaBBiT should be able to *dynamically propagate* any changes made in one programming requirement to all other requirements connected through associations. A similar behavior can be observed in spreadsheets where a value changed in one *cell* is propagated to all other cells having a reference to the changed cell through formulas. However, unlike spreadsheets, RaBBiT will use parameters in associating pieces of information and will have a generative capability.

Change propagation especially becomes important for assuring the *continuity*, *completeness*, and *validity* of a knowledge model in RaBBiT. When the user changes part of the programming information, such as the name of a requirement or a parameter, all of the associations having reference to the changed information should be informed of the change so that they can update their content. Let's consider the following example; A *parameter x* is referenced in a *functional association f* determining the *parameter y* of *requirement R*. If the user changes the name of the *parameter x* to *parameter z*, *requirement R* will not be aware of this change without change propagation; therefore, the knowledge model will be invalid and inconsistent. The change propagation feature will ensure that *requirement R* and, in turn, the *functional association f* and *parameter y* will handle this change as it occurs and the model adjusts accordingly.

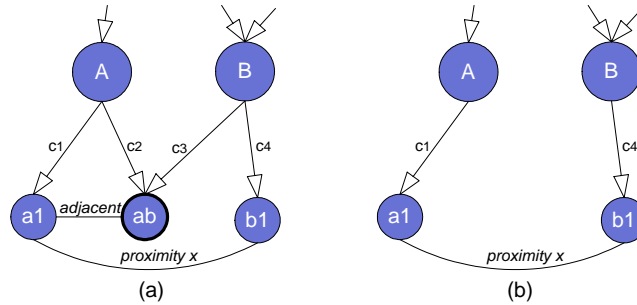
Before change: . . .  $f: \text{parameter } x \rightarrow \text{parameter } y \in \text{requirement } R$

After change: . . .  $f: \text{parameter } z \rightarrow \text{parameter } y \in \text{requirement } R$

Similarly, if the *requirement R* changes, all of the parameters associated to the program information captured in *requirement R* should be informed of this change.

2. RaBBiT should *dynamically update* the state of a working model during knowledge modeling. The programmer will add, remove, or modify requirement information and dependencies, represented in *components*, *constructs*, and *associations* respectively. Each of these actions will make the model state change; for example when the programmer removes a requirement connected with dependency associations to other requirements, all associations should also be removed from the model—with the user's permission. The model state change should be synchronized with the user input and visible to the user. In addition, the model's state should be persistently storable at a given time when the system is running (save feature).

Figure 5.6 shows an example of the state change of a model before and after a requirement is removed. Assume that the user removes *requirement ab*. RaBBiT should update the model by removing the associations from or to *requirement ab* as the operation is completed.



**FIGURE 5.6.**(a) the sample model before remove operation and (b) after remove operation

3. To maintain *consistency* and *correctness* of a model, RaBBiT should ensure that changes made in the model are consistent within the internal representation (data structure) and views. As programming information is entered, the knowledge model will gradually evolve in the form of a complex data structure. During this process, information added or removed should not interrupt the settled state of the model. For example, establishing a dependency association from one requirement to the very same requirement—or any cyclic relation, for that matter—violates the model and creates an inconsistency. Similarly, a parameter cannot have a functional or conditional association to itself.
4. When programming concepts and requirements are sorted according to an information level structure, RaBBiT should ensure that a lower-level requirement depends on only either the requirements in the same level or higher-levels, not on requirements in the lower-levels. This will ensure a proper transition from higher-level to lower-level programming information. Note that the information category structure is an optional feature: in case it is not used, all programming requirements are grouped under one level.

---

## 5.3 Program Generation Features and Requirements

### 5.3.1 Overview

This section outlines the requirements for program generation in RaBBiT. Like the requirements for knowledge modeling, the generation requirements are grouped under two types. Structural requirements specify the type of information and data used in program generation—and in the generated program itself. Behavioral requirements identify RaBBiT's functions for both generating program information and transforming a program model for sharing with other applications. Program information is captured in a computational representation called *program data*. The structural organization of a program will be based on a schema called *program schema*.

### 5.3.2 Structural requirements

1. As explained above, a knowledge model contains reusable (generic) program information and associations for a building type. A program is a composition of requirements generated by evaluating the associations defined in a knowledge model. A knowledge model is created for a specific building type, not for a specific project; a program is generated for a specific project and contains only the information relevant to that project.
2. RaBBiT should have a generic, modular *program schema* able to capture requirements as *program data*—possibly distributed over information levels. The program schema should allow for a mapping between information in the knowledge model and in the program. The schema should specify the structural organization of information in a program and the program should adhere to this schema. Neither program nor program schema should enforce a document view—i.e. both should be view-independent. In other words, program data and schema should be structure-oriented, not presentation-oriented.

A generated program will be composed of requirements following the program schema specifically designed for RaBBiT's program generation feature. Client applications that use a generated program have to transform the program data through schema transformation. This transformation can result in either a formatted architectural program document or another form of program data to be used for other purposes. Formatting of the program data can comply with a particular document style—such as defined in the architectural programming literature.

3. The information types used in the knowledge model, such as *components*, *constructs*, and *associations*, should be able to export their content as program data. For example, information specified in a component will be mapped to its corresponding representation in the program schema. Therefore, components should provide the program generator with queries for exporting its content in terms defined in the program schema.
4. The program schema should be in a shareable format, like the one used in Extensible Markup Language (XML) documents and schema definitions.
5. In addition to the information extracted from the knowledge model, the program schema should accommodate the following information (a) project name, location, description; (b) client name, address, contact name, e-mail address, and fax and phone numbers; (c) program version, date, description. When possible, a default or system-assigned values can be used in program generation, e.g. project date.

### 5.3.3 Behavioral requirements

1. RaBBiT should provide a program generation mode that can assist programmers in composing an architectural program for a particular project. The programmers will primarily enter or change values of critical programming parameters in the system; RaBBiT will update the knowledge model in accordance with these changes and generate all the relevant program information for the given project.
2. The operations of the program generation mode should not compromise a knowledge model's integrity, consistency, and correctness. Programmers should not be allowed to change the semantics (domain knowledge) of the program while RaBBiT is in program generation state (mode).
3. The program generation feature should be able to check the validity of the generated program (data) against the program schema. A generated program should be compared to the programming schema. If a generated program matches the program schema, it is said to be *valid*.
4. RaBBiT should create an output file for each program generated; the file will include only the data and not any view-dependent information. The content of the file should not require a particular application to open it; therefore, a text-based output file—such as in XML syntax—will be appropriate. Document formatting and view generation will be handled through schema transformation. For this purpose, RaBBiT should provide at least one schema transformation to demonstrate how the generated program data can be formatted for other uses.

---

## 5.4 Summary

In this chapter, I define the very essential system features and requirements of RaBBiT under two categories: knowledge modeling and program generation. For knowledge modeling, RaBBiT's architecture should contain three types of entities: (a) components representing requirements, (b) constructs representing parameters, and (c) associations representing dependencies and relations between components and their constructs. The relations between constructs are captured by parametric associations which can be conditional, functional, or nested. In program generation, a program schema follows the programming information structure defined in the knowledge model, and a program generation mechanism creates programs for specific projects using both the knowledge model and the program schema. The generated program should follow some standard data transfer formats such as XML.

The specified features and requirements are going to be used in the selection of appropriate software engineering technologies and in the design of the system.

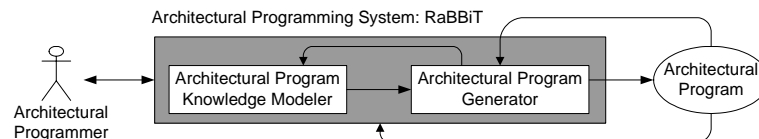


## Chapter 6 Technology Selection for RaBBiT

### 6.1 System Layers and Technologies

#### 6.1.1 RaBBiT's system layers

RaBBiT is intended to allow users to capture programming knowledge in dynamic knowledge bases through a set of graphical user interfaces that enable users to interactively define architectural programming models in their own terms. RaBBiT is also intended to allow users to generate and modify architectural programs. The entire process is incremental and iterative. RaBBiT supports this process through two system layers: a *programming knowledge modeler* and a *program generator* (Figure 6.1). By using the first layer, the user is able to define a type-based architectural program model for a recurring building type that includes higher- and lower-level program information, including associations among pieces of information. Using the second layer, a user is able to generate an architectural program by applying the knowledge model to a specific project whose critical parameters are input by the user. If needed, the user is able to re-enter the parameters and regenerate a new program; to modify the generated program; to add more details; or to change the programming knowledge model.



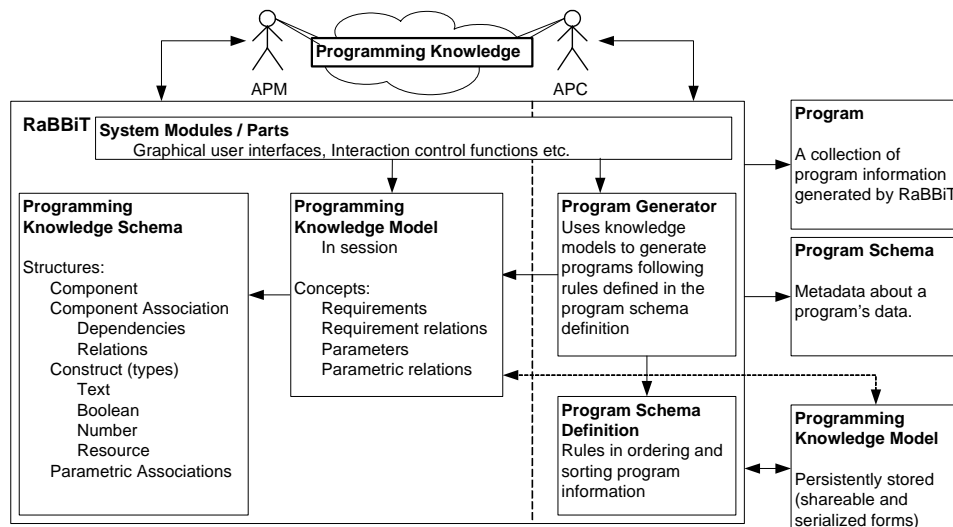
**FIGURE 6.1.** The incremental and iterative programming process supported by the system

#### 6.1.2 The main concepts used in RaBBiT

*Programming knowledge* refers to the domain knowledge about programming a building type as understood by architectural programmers outside the context of RaBBiT (Figure 6.2). This knowledge can be partially structured and captured in various media and representations.

A *programming knowledge model* captures programming knowledge in terms of RaBBiT concepts such as program requirements, parameters, requirement relations, and parametric relations. This model is internal to RaBBiT. *Program knowledge modeling*—or *knowledge modeling* for short—refers to the task of defining a programming knowledge model with the help of RaBBiT. Once it has been defined, such a model can be saved persistently in some form.

Knowledge modeling is based on a *programming knowledge schema* which comprises symbolic and generic computational representations each of which corresponds to one specific type of RaBBiT concept. The definitions of the schema and the representations used are integral parts of RaBBiT—users cannot change these definitions, only developers are allowed to do this. For example—in the object-oriented programming terms introduced below—a programming knowledge schema can be a collection of classes and their associations that can be instantiated to build a knowledge model.



**FIGURE 6.2.** Programming knowledge concepts and their representations in the model

A *program schema* defines a set of rules for ordering and sorting information captured in a programming knowledge model; it is internal to RaBBiT. The schema describes in what order the program information is sorted independent of the content of a program. A typical rule, for example, defines the order of how construct properties are sorted, such as name, description, (value) type, value, and a list of all associated constructs. The program schema also serves as a reference for transforming a program from its original form to another form in some other schema. For this purpose, every time a program is generated, a program schema definition containing metadata about the program data is also created.

A *program* is the collection of all relevant pieces of programming information for an instance of a particular recurring building type as generated by RaBBiT under the directions of the users; it is based on an underlying programming knowledge model and structured according to a program schema. Users can modify a program after it has been generated without the need to modify the underlying programming knowledge model or program schema.

### **6.1.3 Programming paradigm and technology**

In developing an application such as RaBBiT, the most appropriate programming paradigm or paradigms must be selected based on the nature of the application. It is obvious that the layers of RaBBiT must computationally represent robust knowledge models and implement generative functions reasoning on these models. The first layer is used for knowledge modeling and the second layer for *reasoning on* the acquired knowledge. The object-oriented (OO) programming paradigm is a promising candidate for the first layer, because we can capture programming knowledge by using objects corresponding to the domain concepts. A rule-based production system appears appropriate for the second, program generation layer because it offers convenient means to express the rules about how the objects are related and used in decision making along with the conditions under which these rules must be applied considering the objects at hand at any given time during program generation; production rules are a prime candidate for handling such dynamically changing conditions and the actions contingent on them. In the following, I discuss these technologies in the context of this research.

---

## **6.2 Object-oriented (OO) Programming**

### **6.2.1 Overview**

I favor the OO paradigm for the purposes of knowledge representation in RaBBiT. There are two basic reasons behind this decision: (a) OO programming is currently the most promising paradigm to guarantee general quality attributes of software (Meyers, 1988 and 1997); and (b) the paradigm provides convenient technologies—such as inheritance, polymorphism, object composition—for designing a system for capturing a programming knowledge model as envisioned here and for defining the underlying schema. I will elaborate each of these advantages below.

### **6.2.2 Sources of software quality**

Although there have been debates about the capabilities of OO programming and technologies (Ling, 1993; Hymes, 1995; Elrad et al., 2001), OO programming is currently the state-of-the-art for developing software systems because it provides advantages over

(a) conventional programming techniques—such as functional programming—by loosely coupling data structure and behavior; and (b) over alternative programming paradigms (such as aspect-oriented programming<sup>1</sup>) that are not commonly practiced and tested yet in their entirety. My own experience (Flemming et al, 2001) confirms the claims made by the pioneers of OO programming in its favor (Meyer 1988; Rumbaugh, 1991; Su and Chen, 1993; Oesterreich 1999)

A general discussion of programming paradigms is out of the scope of this research<sup>2</sup>. In the following, I summarize the arguments in favor of OO programming (Meyer, 1988) and relate them to the present context.

Meyer lists both *external* and *internal* factors for achieving quality software. Figure 6.3 outlines a hierarchy of these factors in terms of goals, means, principles, and approaches to satisfy general quality attributes. External factors refer to aspects of concern to users and fall into two main groups: *reliability* and *maintenance*. A software system is reliable if it is able to perform tasks as defined by requirements (*correctness*) and to function even in abnormal conditions (*robustness*). A system's maintainability is defined by its ability to (a) adapt to changes through use of simple system architectures with decentralized modules (*extendibility*), (b) reuse system modules for new applications (*reusability*), and (c) easily combine and interact with other systems (*compatibility*).

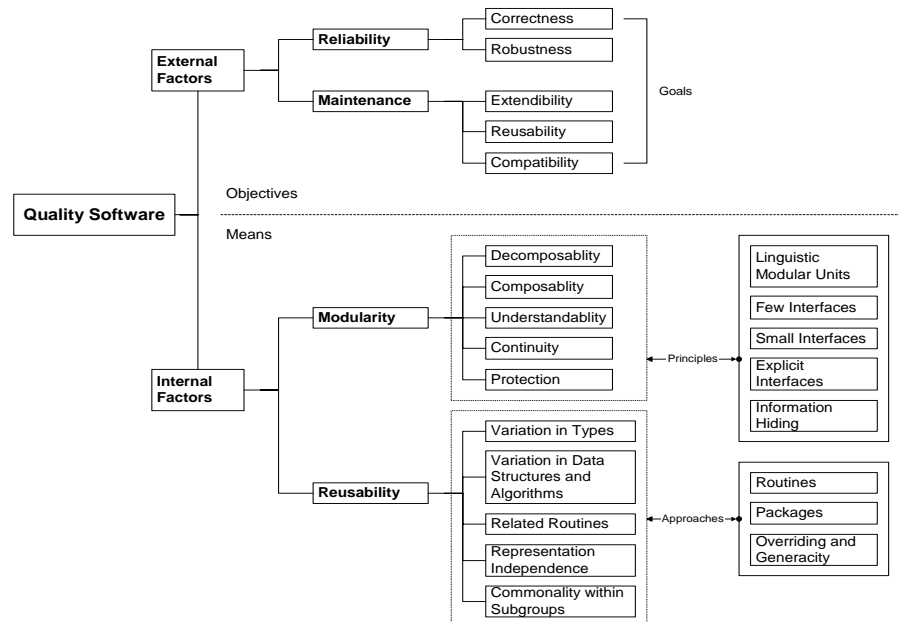


FIGURE 6.3. Objectives of a quality software and means for achieving them

1. Aspect-oriented programming is described in (Elrad et al. 2001a and 2001b; Katara, M. and S. Katz. 2003)
2. For detailed information about programming paradigms refer to (Czarnecki and Eisenecker, 2000).

Internal factors are the *means* to achieve external factors (goals) and “*perceptible*” only to software developers. These means are divided into two groups: *modularity* and *reusability*. Modularity is key to both reusability and extendibility and depends on the combined effects of five features: *decomposability*, *composability*, *understandability*, *continuity*, and *protection*. Decomposability relates to reducing the complexity of a system by dividing it into a set of less complex sub-systems or modules, while composability means that system modules can combine with each other to form a larger system. Software developers should be able to understand the function of a module by having to look only at a small set of related modules (*understandability*). Continuity means that a small change in a module must require only a change of just a few other modules. A module’s state can be altered only through certain specified methods (*protection*).

Modularity as defined by the above five features can be achieved by observing five principles: providing a modular structure where the modules represent meaningful concepts in the application domain or for application programmers; connecting modules with few, simple and well-defined interfaces; and information hiding such that only module interfaces are exposed to other modules and not their implementation. Reusability can be achieved through the following approaches: (a) a module should include all routines operating on that module; (b) module routines—also known as methods—must implement “well-defined” operations; (c) packages must be used for grouping all modules, routines, types, constants, and variables that are related to an important conceptual part of the system.

### **6.2.3 Modularity and Reusability for RaBBiT**

In order to deliver the features that are crucial for modularity and reusability, the OO paradigm relies crucially on *classes*, *inheritance*, and *polymorphism* (Rumbaugh et al., 1991, pp. 9). Below, I describe these concepts by giving examples on how they can be applied in the context of RaBBiT.

#### **Class structure**

Classes are used for *abstracting* domain concepts such that each *combines (packages)* the common *structural* and *behavioral* properties of a domain concept in a *class definition*. For example, in the RaBBiT context, program requirements and parameters can be represented through component and construct classes, respectively. A component class may describe a requirement, and a construct class may represent a parameter as defined in Chapter 5.

Objects are instances of classes and represent unique entities created at run-time. For example, a *secretary room (requirement)* can be an instance of a component class and the parameter *minimum area requirement* in the secretary room can be an instance of a

construct class associated with it. The *types* of attributes or properties an object can have are those of the class it instantiates, whereas the *values* of these attributes are determined in interaction with other objects or with the user. There can, of course, be many objects instantiating the same class, and they typically differ in terms of their attribute values.

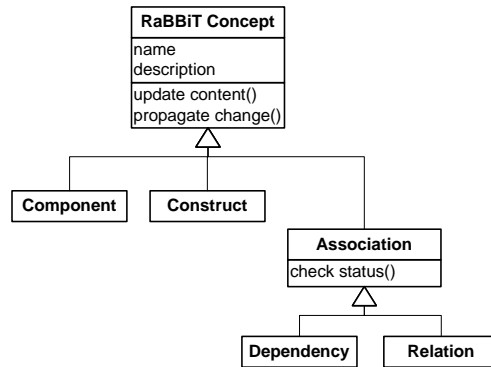
Note that a property of an object can be another object attached to it. In this way, behavior can be associated with object properties such that the responsibilities of an object can be delegated among its properties. For example, a change in the value of a parameter can be handled by the construct object representing it. This simplifies the structure of the object and of the class that defines its properties, thus increasing modularity. I will return to this in the section on object composition below.

The structural and behavioral properties of an object can be of two types. The first type directly corresponds to the properties of the domain concept it represents; the second type refers to implementation-oriented properties, such as how an object is saved by the system, which is of interest only to developers, not to users.

### **Inheritance**

Inheritance provides a mechanism through which the common properties of different classes can be factored out by establishing a super- and sub-class hierarchy among the classes. Each sub-class inherits properties it shares with other sub-classes from a common super-class, possibly over several levels of the hierarchy. For example, *name*, *description*, and *update* are common properties of the component and construct class. Therefore, each can be a sub-class of a more abstract class, *RaBBiT Concept* (Figure 6.4), which defines the common properties so that components and constructs can inherit them. This type of inheritance extends to shared behavioral properties. For example, components and constructs can inherit methods to update content or propagate change. But note also that a sub-class can have unique properties that are not inherited from a super-class (but can be inherited by a sub-class of the class).

Inheritance reduces code repetition. Another advantage is that every instance of a sub-class can be used in a place where an instance of one of its super-classes is expected. For example, all instances of component, construct, and association classes can be treated as instances of the RaBBiT Concept class. This feature lies at the core of polymorphism described briefly below.



**FIGURE 6.4.**Inheritance relations among requirement classes (UML notation)

### Polymorphism

Polymorphism allows (computer) programmers to redefine shared behavior in a sub-class. For example, components and constructs may need different update methods. Under polymorphism, each of these classes can implement its own version of the method and thus overwrite the definition in its super-class (provided all versions have the same signature). The proper version of the method is selected at run-time by a mechanism called *dynamic binding*. For example, when at a certain point the update method is called for an instance of a RaBBiT Concept, the selection of the method depends on whether this instance is, in fact, a construct or a component. In other words, programmers do not have to write explicit conditional or case statements to guarantee that the proper method is used.

Polymorphism and inheritance thus go hand-in-hand and are the main means by which OO achieves the modularity and flexibility that are its hallmark. For example, given a super-class RaBBiT Concept, polymorphism enables the programmer to write different implementations of the update or propagate change methods for any number of sub-classes, such as components, constructs, and associations. More importantly, sub-classes can be added to the program at any time and re-implement any shared method so that the addition does not require any (or only minimal) changes in the existing code: Under dynamic binding, no case distinctions have to be added to the program wherever instances of the new sub-class may appear at run-time.

#### 6.2.4 Object configuration

Objects can have relations with other objects. For example, a dependency relates a source with a target object. In order to capture relationships like these, we can use object composition techniques that are an integral part of OO programming: association and aggregation.

Object association is a technique that enables an object to refer to another object, where the two objects do not affect each other's existence; i.e. if one object is deleted, the other object survives—only the relation between them must be eliminated. In aggregation, an object becomes part of another object so that its existence depends on the existence of the container object—the two objects have to be deleted together.

For example, a dependency relation represented as an object (see below) can be *associated* with two component objects, where the two components can exist in the model independent from the dependency object. Furthermore, a condition object can be *aggregated* into this dependency object, where the condition object has no meaning outside the dependency relation, i.e. should be deleted when the dependency is deleted. Another example of object composition is a component object that is a *composition of* multiple constructs. In turn, the knowledge model itself can be an object that is composed of component, association, or other related objects.

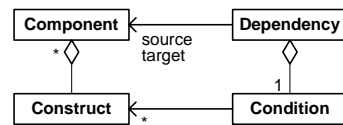


FIGURE 6.5. Same composition and association (UML notation)

### 6.2.5 OO representation of the knowledge models

A *programming knowledge model* in this research merges architectural programming rules and requirements to support a configurative and generative process, not a search for a solution<sup>1</sup> as used in other problem solving situations. The building blocks of a programming knowledge model can be defined as classes with features that are typical for OO programming.

In particular, types of requirement associations can be defined in association classes that can be used for representing architectural programming rules, i.e. each association class can implement how a programming rule effects program generation. This approach has advantages from the implementation and use perspectives.

From the implementation perspective, it enhances modularity and extensibility because it makes component classes as "light-weight" (simple) as possible: they have to represent only requirements, not how they relate to other requirements in the model. This can eliminate the need for modifying the component class—attributes and the methods that manage them—every time a new type of association (class) is introduced. New types of associations can be defined as sub-classes of a general association class through inheritance, for example, dependency and relational associations. Another advantage is

1. Particularly this feature is addressed in GMEA, which separates it from general MEA.



that the relations between requirements can be managed at the model level, not in the components, which allows users to modify—during run time—association objects independently of component objects, for example, to change conditional associations or labels attached to them. As a result, instances of associations become part of the model as distinct entities that are able to take over various responsibilities during modeling and model updates; for example, they can adjust themselves following a change made in the model because—unlike link attributes—they can be aware of other associated objects such as dependency conditions.

From the use perspective, users can manage the requirements and their relations as distinct entities independently from each other. This leads to a "plugable" structure that can be dynamically created and changed during interactive modeling without effecting the internal states of the related component objects. Once an association object is inserted into a model, the model has to respond to this change. This is a plausible approach since during program generation, a model will be queried for its content and has to know all of its parts. For instance, the user may want to see all the dependency associations: Instead of asking every component object to expose its dependencies, the model will have them ready for the user as independent objects. In addition, the associations and the conditions that they contain can be copied and modified together without disturbing other objects. The copies can then be used for associating other components that have the same association type. For example, assume  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $E$  are component objects;  $A \rightarrow B$  ( $A$  depends on  $B$ ), but other relations are not established yet. The dependency association can be interactively modified such that the users can copy this object and use the copy to set a relation between  $C$  and  $D$  ( $C \rightarrow D$ ). This will create a new dependency association between  $C$  and  $D$  by using the same conditional logic as the one between  $A$  and  $B$ . If the user wishes, this condition object can be altered independent from the component objects. In addition, one end of the association can also be edited; for example  $A \rightarrow B$  can be changed to  $A \rightarrow E$  or  $E \rightarrow B$ , if this is logically applicable.

---

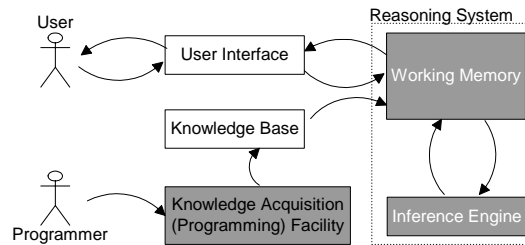
## **6.3 Production Systems**

### **6.3.1 Programming paradigm for program generation**

RaBBiT's system layer for generating programs can be implemented using a programming paradigm that allows for flexible definitions of program generation rules that can be applied on programming knowledge models described above. The need for programming rules whose applicability depends on conditions—together with the GMEA structure—offers an opportunity for a rule-based production system—a class of knowledge-based expert systems in artificial intelligence.

### 6.3.2 Production systems

A production system (Figure 6.6) supports flexible and modular programming structures based on conditional knowledge expressed in the form of productions or rules. A rule is simply an *if-then* (*If* <conditions> *then* <actions>) statement that establishes a relationship between a condition or a context and the actions that can take place under these conditions or in this context. The *if* part of a rule, also called its *left-hand side* or *antecedent* part, describes the condition, typically as a logical expression or clause that evaluates to true or false. The *then* part, also called the *right-hand side* or *consequent* part, specifies the actions that can be performed if the antecedent part holds *true*. A rule whose antecedent clause is true, given the current state of working memory, is said to be ready to *execute* or to *fire* (Flemming, 1994, pp. 5).



**FIGURE 6.6.** The generic architecture of rule-based systems.

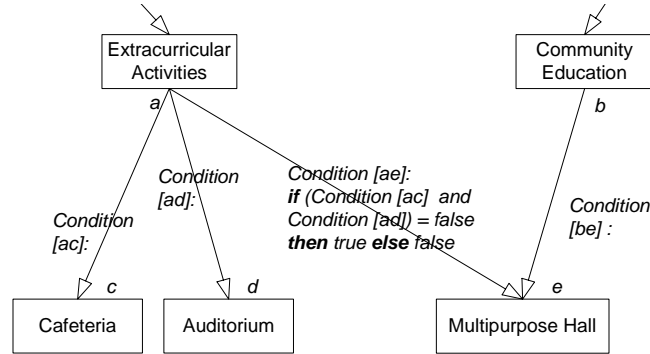
A collection of rules that captures some domain expertise is the domain-dependent *knowledge* or *rule base* of a production system. In addition, a production system contains a domain-independent *inference engine* as an integral part (Russel and Norvig, 1995, pp. 297 and 313). It is able to "match" at any given time the left-hand sides of all rules against the current state of working memory, i.e. to determine which rules can fire. If there is more than one such rule, the inference engine applies some conflict resolution strategy to select a single rule for execution, which is likely to change some portion of working memory. After this, the match-select-execute cycle is initiated again.

The order in which the rules appear in a rule base is arbitrary, which leads to a flexible organization that can incrementally expand, whereas in another type of system, changing a condition may trigger revisions over the complete system (Haley, 2001). Production systems therefore allow programmers to acquire and refine domain knowledge in an incremental, step-by-step fashion (Rychener, 1976).

### 6.3.3 RaBBiT and Production Systems

Since production systems can directly support the incremental refinement and evolution of knowledge bases as required for RaBBiT, they constitute *prima facie* a promising programming paradigm for implementing the program generating layer in RaBBiT. Let's

consider the example taken from programming school buildings in previous chapters (Figure 6.7)



**FIGURE 6.7.**Partial example for programming school buildings.

We can state one of the rules (condition [ae]) leading to the requirement for a multi-purpose hall as follows:

**IF** extra-curricular activities are required  
**and** the other conditions to have multi-purpose hall are satisfied (such as budget)  
**THEN** the school will have a multi-purpose hall.

This rule can be defined in RaBBiT terms as follows:

```

rule generate multi-purpose hall for extra-curricular activities {
  IF: there is a component (object) representing extra-curricular activities
    and there is a dependency from extra-curricular activities to multi-purpose hall
    with dependency condition true
    and there is NO component representing a multi-purpose hall
      (in working memory)
  THEN:
    generate a Multi-purpose hall
    (insert a component representing Multi-purpose hall into working memory)
}
  
```

The condition "if there is no component representing a multi-purpose hall" ensures that the multi-purpose hall is not duplicated. This is needed because the dependency association [be] also will try to generate a multi-purpose hall, i.e. there is a chance that at a certain time during program generation, the requirement for a multi-purpose hall is generated twice (or multiple times).

Consider the condition on dependency [ae]: the multi-purpose hall for extra-curricular activities will be added to the program if cafeteria and auditorium are not added. Therefore, we need another rule that removes the multi-purpose hall from the program when cafeteria and auditorium are added. However, this rule also must consider the

dependency [be], since the multi-purpose hall requirement could have been generated as a result of dependency [be].

```
rule switch to Multi-purpose hall alternative {  
  IF:  there is a component representing cafeteria  
        and there is a component representing auditorium  
        and there is a dependency from a component to multi-purpose hall  
        with a dependency condition false  
  THEN:  
    remove Multi-purpose hall from the working memory)  
}
```

#### **6.3.4 Transforming programming knowledge model to a rule-base schema**

The OO model representing programming knowledge does not include "rules" in a form that can be used in a production system directly. Therefore, the parts of the knowledge model that possibly contain rules must be transformed into a rule-base schema. This can be achieved by a program (or a subsystem of RaBBiT) that searches the conditional knowledge captured in the model for dependency and conditional association objects and translates these conditionals into rules. For example, the dependency [ae], in the example above, can be a dependency object containing a condition that implies a rule. The transformation of all conditional knowledge will provide a rule-base that can be used for reasoning in program generation. During this process, objects such as component and constructs in the programming knowledge model can be inserted into the working-memory of the production system, which in turn will fire the relevant rules.

However, it must be noted that transforming the conditional knowledge as defined in the programming knowledge model into rules is not a trivial task and poses some risks, because the transformation will force the knowledge model to change from its intended state defined by architectural programmers to a different form that will be used by a different technology. Some of these risks are loss of information and errors caused by the transformation (Noy and McGuinness, 2001). Indeed, we may even need an additional set of transformation rules apart from the rules in a programming knowledge model. This may require some extra afford and will be non-trivial, given the complexities inherent in programming knowledge models.

For program generation, one alternative programming paradigm is a rule-based production system. However, using such a system will require a complex model-transformation and rule-translation mechanisms, as mentioned. Before directly implementing program generation module by using production system, I would like to test another alternative that is purely based on OO paradigm. This part will be designed and implemented with OO

programming that it can generate programs, possibly, by simulating the behavior of a production system.

#### **6.3.5 User interface**

In production systems, the user provides information about the problem to be solved; the system then attempts to provide insights derived (or inferred) from the rule-base. This interaction between user and the system may occur in two basic forms: (a) domain-experts (and rule-based programmers) define and modify the application domain knowledge (rules) in the system and (b) the system asks users to direct the process by answering questions or inputting data while the system is running. The second interaction is simple and straight-forward if the system is already built and running. The first type of interaction, however, still poses challenges to both the programmers of the rule-based system and the users because it depends on knowledge acquisition as an independent process. The main challenge to programmers is to provide a facility through which the users can actually define the domain knowledge in the system without having low-level programming expertise (Clark et al., 2001; Barker et al., 2001). The first layer in RaBBiT can serve this purpose.

---

### **6.4 Data Structures for Knowledge Modeling**

#### **6.4.1 Data structure requirements**

A collection of requirements and dependencies among them form the backbone of a knowledge model. The selected structure should directly support the incremental refinement and evolution of a knowledge model. In other words, the structure should allow architectural programmers to incrementally acquire and refine domain knowledge without the need to reorganize the entire collection of requirements and associations when a new condition or special case is added to the system. The structure should provide means to define, modify, and remove a requirements or dependencies between two existing requirements at any given time and without a substantial change in the knowledge model.

#### **6.4.2 Data organization techniques**

The organization of data in RaBBiT can be managed by different data representation techniques such as decision trees, flow charts, or graphs. However, not all of these are equally efficient in meeting the data structure requirements mentioned above.

Control of the depth and breadth of the branching pose a challenge in decision trees especially when a programming knowledge model is interactively defined and complex—which is the case at least for the three recurring building types studied in this research. In addition, multiple dependencies may have to be executed in some conditions, but decision

trees allow only one branch to be selected among given alternatives, i.e. only one consequent rule is executed along the branches. This can be overcome by using highly-coupled multiple decision trees, but this solution reduces efficiency—if it can be made to work at all in the present context. In addition, if a requirement is removed from the decision trees, all subsequent branches are removed recursively, which poses problems for multiple conditionals as they exist in the dependency web of requirements (Section 4.1.2).

Flow charts exhibit the same shortcomings. Especially, as the flow chart grows larger, conditionals become more complex, which leads to rigidity and a program composition that is hard to revise.

If there is only a small number of rules, the use of flow charts or decision trees can be efficient. This could be the case, for example, in programming a simple recurring building type, such as a fast-food store. However, as the program requirements become more complex, like in the building types studied in the case studies (see Chapter 3), the number of programming rules gets larger, which makes flow charts and decision trees less efficient.

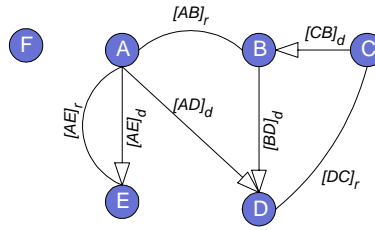
Graphs, on the other hand, present a fairly flexible structure. A graph consists of vertices (nodes) and edges connecting certain pairs of vertices and, unlike decision trees, the exact geometric pattern is not specified. As a generalization of lists and trees, graphs come into play where lists and trees are not sufficient to model more complex relations. For example, each node has at most one parent, and zero or more children in a tree, whereas in a graph, each vertex simply has zero or more *neighbors*. In representing programming knowledge, each vertex can represent a requirement, and edges between vertices can represent the associations between different requirements. A graph can be modified easier than a tree or a flow chart as changes effect only the modified requirement locally and, if they exist, the (neighboring) requirements connected with associations to the modified requirement. The rest of the structure—and therefore the model it represents—stays intact.

#### **6.4.3 The graph structure in RaBBiT**

Note that not any graph structure is appropriate for representing programming knowledge models that comprise an ordered assembly of requirements with *dependency associations* from higher- to lower-levels (see Section 4.1.1). In addition, *relational associations* can be set regardless of the information level of two requirement. Therefore, the graph should be able to accommodate *directed edges* for *dependencies* and *bidirectional edges* for *relation associations*. Furthermore, higher-level information cannot depend on lower-level information; therefore, edges representing dependencies as specified in this research should form an acyclic graph, i.e. a graph without cycles. For relational associations, there is no such restriction.

The graph that can be used in representing a programming knowledge model in RaBBiT can be formally described as  $G(V,E)$  such that  $V$  is a set of all component nodes (requirements) and  $E$  is a set of the relational associations (dependencies and relations) between these nodes. Dependency associations form a sub-graph  $G_d(V_d,E_d)$  of  $G$  ( $G_d \subseteq G$ ) such that  $V_d \subseteq V$  and  $E_d \subseteq E$ , where  $E_d$  represents the set of directional dependencies as one type of edge. In addition a set of dependency associations  $P \subseteq E_d$  showing a *path* from any requirement  $v \in V$  to the same requirement  $v$  is an empty set ( $P = \emptyset$ ), i.e. cycles and loops of dependencies are not allowed. Relation associations also form a sub-graph  $G_r(V_r,E_r)$  of  $G$  ( $G_r \subseteq G$ ) such that  $V_r \subseteq V$ ,  $E_r \subseteq E$ , and a dependency association  $e_d \in E_d$  is not a member of  $E_r$ .  $E_r$  contains the second type of edges. Briefly, the graph for representing a programming knowledge model can be defined as  $G_d(V_d,E_d) \cup G_r(V_r,E_r) = G(V,E)$ ,  $E_d \cap E_r = \emptyset$ , and  $G_d$  is acyclic and directional.

Consider the sample graph given in Figure 6.8, where the nodes represent requirements and edges dependency and relational associations.



**FIGURE 6.8.** A sample graph suitable for program knowledge modeling.

The set of all requirements is  $V = \{A, B, C, D, E, F\}$ , and the set of all the relations is  $E = \{[AE]_r, [AE]_d, [AD]_d, [AB]_r, [BD]_d, [CB]_d, [DC]_r\}$ . The set containing the nodes in the dependency graph is  $V_d = \{A, B, C, D, E\}$  and the edges of dependency graph are contained in the set  $E_d = \{[AE]_d, [AD]_d, [BD]_d, [CB]_d\}$ . The set of the nodes in the relation graph is  $V_r = \{A, B, C, D, E\}$  and the set of relation edges is  $E_r = \{[AE]_r, [AB]_r, [DC]_r\}$ . An advantage of graphs is that nodes may not have any connecting edges but still can be part of a graph, such as the node  $F$ .

## 6.5 Model-View-Controller Architecture and System Layers

### 6.5.1 Model-View-Controller Architecture

The Model-View-Controller (MVC) architecture was first introduced by Trygve Reenskaug in SmallTalk-80—an object-oriented programming language (Krasner and Pope, 1988; Burbeck, 1992). MVC was originally developed to map the traditional input, processing, and output roles into the graphical user interface realm. However, MVC and the other architectures derived from it (such as Presentation-Abstraction-Control) have evolved into a common pattern used generally in current software engineering practice (Buschmann et al., 1996, pp. 134; Bosch, 2000, pp. 142; Veit and Herrmann, 2003).

The MVC architecture insists on a clear separation among the object *model* (abstraction) of the real-world concepts in an application; the *views*, which are visualizations of the state of the model; and the *controlling mechanism*, which establishes communication between the model and the view and offers facilities to change the state of the model. This separation creates the benefits of *reusability*, *modularity*, and *flexibility* of the system or its parts because designing, adding, removing, or modifying a system component can be independently managed; specifically, changes made in one part remain local, i.e. do not lead to ripple effects throughout the system, and the different parts are reusable independently of each other.

The controller is the only part that connects a view with the model. Ideally, the model cannot have a direct reference to either a view or a controller. The basic MVC architecture is shown in Figure 6.9. Note that more than one view can be attached to the same model, and all views are automatically updated when the model changes. Note also that although the model, view and controller are isolated in design, they are in constant communication with each other during execution.

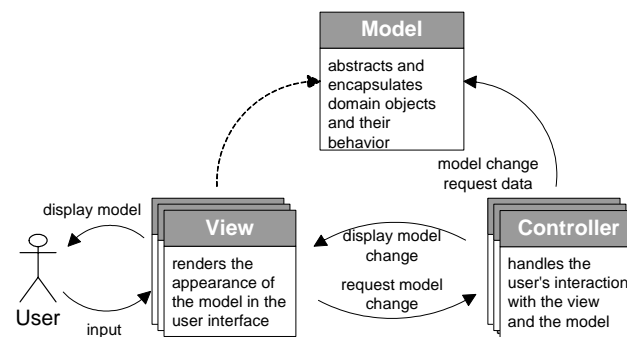
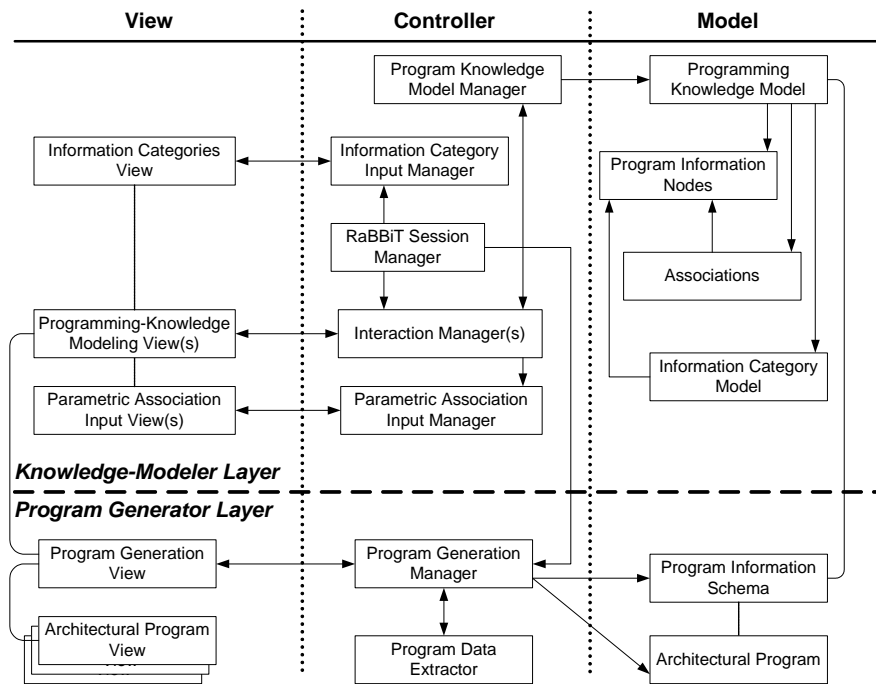


FIGURE 6.9. Model-View-Controller architecture



### 6.5.2 RaBBiT's MVC Architecture

RaBBiT should adapt the Model-View-Controller (MVC) architecture for clear separation of sub-systems and layers that show semantic and logical differences and have loosely coupled responsibilities. RaBBiT should have two layers—as mentioned in Section 6.1—based on its functionality: (a) the programming-knowledge modeling layer, and (b) the program information generation layer. If we implement these based on the MVC architecture, we arrive at the following sub-systems for RaBBiT: (a) the model (M) representing programming knowledge models captured in representations as defined in the programming knowledge schema, (b) the controllers (C) consisting of manager objects that control the flow of information, delegate responsibilities, and facilitate interactions between the system and the user; and (c) the views (V) which expose the system functions and present the knowledge models to the user or to other systems (Figure 6.10).



**FIGURE 6.10.** Modules of the system adapting the MVC architecture.

### 6.5.3 The model in RaBBiT

This sub-system is responsible for capturing knowledge models and program information. It contains the input and output of knowledge modeling and program generation processes and only persistent components of RaBBiT. The controllers and views are transient and recreated each time a RaBBiT session starts.

### **Programming knowledge schema**

This part of the model sub-system contains classes as the internal representation of the programming knowledge of a building type. Taken together, these classes provide a flexible, reusable, and modular object-oriented schema as described in Section 6.2.4. This schema should be able to capture a programming knowledge as specified in Chapter 5. A programming knowledge schema contains program information in components, constructs, and associations which, together these are combined in a graph structure. Therefore, this schema supports the graph representation of the knowledge models (see Section 6.4.3).

### **Information nodes**

These parts contain requirements in the form of *components* and *constructs* and are part of the graph as nodes (vertices). The parametric associations are encapsulated in these nodes along with the properties (attributes) of each component and construct.

### **Associations**

These parts of the model represent dependency and relational associations and form the edges of the graph. The programming knowledge model keeps a record of which dependency or relational association relates to which components. To make the structure flexible, these associations must have their unique representation *as objects*, as specified in Section 6.2.5.

### **Information categories**

The programming information nodes can be classified within groups that semantically separate requirements. Each category has a unique name and a category level number. The system is able to import, export, and save a category model independent from the knowledge model that it is used for. In this way, users may choose a predefined information category model when a new programming knowledge modeling process begins.

### **Architectural program**

A program must meet the requirements specified in Section 5.3. The content of a program will be generated by the system based on a programming knowledge model. A generated program will follow the structure defined in the program schema. Although an architectural program is an output generated by the system, it must be treated as part of the model sub-system in the MVC architecture.

**Program schema**

This part of the model sub-system must meet the requirements specified in Section 5.3. The schema will be generic enough to be used for program (data) generation and program schema transformation purposes, and it will contain metadata about the program data. Every time a program is generated, RaBBiT will provide the program schema in some form such that other users who are interested in transforming the program data from one form to another will have a metadata reference .

**6.5.4 The controller sub-system of RaBBiT**

Users will interact with RaBBiT and manipulate the knowledge model through graphical user interfaces, which are parts of the view sub-system of MVC. This interaction is always channeled through *managers*, which are part of the controller, not the view. This separation allows the interface (view) classes in the view sub-system and model classes in the model sub-system to be independently modified, actually reused in other applications. As an advantage of this separation, the views will not directly manipulate the model; all of the interactions will be managed by the managers, which will also ensure that the model and its views are synchronized at any given time, and that the consistency of the model is maintained during knowledge modeling. Therefore, the managers, or the control sub-system as a whole, are responsible for connecting views to the model and delegating *responsibilities* to instances of specialized classes in any sub-system in response to user input; but they are not allowed to execute programming tasks as such.

Distributing the responsibilities of the controller over several managers—as opposed to combining them in a single class—increases the modularity and extendibility of the controller and of RaBBiT as a whole. Taken together, the RaBBiT managers have three responsibilities. The first is to manage the interaction between RaBBiT and the computer (system). A second responsibility is to communicate with each other to make sure that the user-system interaction is synchronized. For example, managers controlling the views and managers controlling the model communicate with each other to assure that the system runs without any problem. The third responsibility is to change the mode of the system from knowledge modeling to program generation, each of which requires a unique setting of the system state. For knowledge modeling, the knowledge model can be changed dynamically, but when the generation phase starts, the knowledge model should not be altered but remain accessible.

**RaBBiT session manager**

An instance of this control class is created at the start of a RaBBiT session. The session manager is responsible for instantiating view classes and connecting a knowledge model with these view objects. The session manager is also responsible for persistently saving or

retrieving saved knowledge models. When a new knowledge model is created, the session manager delegates information category definition to other specialized managers. In addition, the session manager coordinates the system running in a particular computer platform (hardware and software) with platform-specific tasks such as file storing, file loading, external resources invocation etc.

**Information category manager**

This manager is responsible for managing information level categories prior to starting the definition of a programming-knowledge model—for a particular building type. The session manager asks this manager to open the views in which the user will define information categories. As the information categories are defined by users, the information category manager creates an information category model (and its computational representation) and passes it to the session manager—or a sub-manager—that will make the category model a part of the programming knowledge model. The information category manager also can (a) import an existing information category model from storage; (b) save the current model so that it can be used in modeling a different knowledge-model; or (c) rename an existing information category model and persistently store it.

**Programming knowledge model manager**

This manager is specifically responsible for coordinating actions or events that directly change a programming knowledge model. It is in direct communication with the interaction manager (see below) and the session manager. It receives messages from the knowledge model and passes them to relevant managers when they go beyond its own responsibility.

**Interaction manager**

This manager is responsible for coordinating the user-system interaction during knowledge modeling with the rest of the system, including parts of the model and controller sub-systems. Each knowledge modeling-related action that a user performs in a view is handled by this manager, except for parametric association definitions, which are handled by the respective parametric association managers.

**Association manager**

This manager is instantiated by the session manager when RaBBiT starts and is invoked by the interaction manager when the user wants to establish a parametric association between different programming parameters, i.e. between constructs. Mainly, the association manager is responsible for creating a parametric association view in which the

user can interactively compose functional, conditional, or nested associations. After an association has been defined, the association manager delegates updating the model to the interaction manager, which, in turn, delegates this task to the programming knowledge model manager.

---

## **6.6 Implementation Constraints**

### **6.6.1 Programming Language**

RaBBiT should be written in Java, which uses a pure object-oriented Application Programming Interface (API) providing a set of routines, protocols, and tools as building blocks for developing an application. Along with complex and primitive data types, such as maps, lists, sets, and hash-tables, the Java API comes with UI building components, such as frames, panels, buttons, tables, trees etc., packaged in the Abstract Widget Toolkit (AWT) and Swing libraries. The Swing library is more sophisticated and provides the most commonly used features for building a GUI. The Swing components adapt the MVC pattern/architecture mentioned above—but note that MVC generally applies to the system architecture overall, whereas in Swing, each GUI is treated as one module made up of MVC parts (classes).

Furthermore, Java and its run-time environments are platform-independent, which makes porting an application between different platforms, operating systems, and hardware easy and reliable. Therefore, design-support systems designed and implemented in other platforms can be incorporated into design through an uninterrupted exchange of information.

In addition to the standard Java API, I use available third-party Java libraries that make it easy to extend the software programming environment so that the first prototype of RaBBiT can be designed and implemented in a shorter time and with tested and robust software subsystems.

### **6.6.2 Graph representation**

An example of a third-party programming library I plan to use is JGraph<sup>1</sup>, a fully standards-compliant graph component for Java that supports the extended display and editing of graphs (Adler, 2002; JGraph, 2003). JGraph is appropriate for implementing RaBBiT because it is fully compatible with Java and provides a clear and efficient (GUI component) design adapting the MVC pattern. In JGraph, "...the basic architecture is the same as for standard Swing components, and the method and variable naming complies with Java code conventions. This has the advantage of reduced learning costs, and existing

---

1. JGraph and JEP libraries and application frameworks are open-source and come with public-use licenses.

source code can be reused, resulting in shorter development time" (Adler, 2002, pp. 1). JGraph enables in-place editing and vertex handling; it also provides mechanisms for data transfer and marquee selection. JGraph enables dynamically creating a graph with its edges and vertices through extendable Java classes. In implementing RaBBiT, the flexible components provided in the JGraph library should be used to their full extend.

#### 6.6.3 Parametric associations

RaBBiT should use the Java Expression Parser (JEP) (Kolaroff, 2002) to input expressions of parametric associations (in text form) and parse these expressions into architectural programming constructs. JEP allow users to enter an arbitrary formula as a string, and instantly evaluate the expression. It also supports user-defined variables, constants, and functions. A number of common mathematical functions and constants are also included. However, JEP does not provide a function for evaluating *if-then-else* conditional expressions; i.e. it is not intended to implement conditional statements as required in *conditional associations*. However, I used JEP's extensibility to implement a sub-system which allows users to enter *conditional associations*. I also added other functions that are used for calculating minimum or maximum values for two given numbers, getting the floor or ceiling of a calculated value, and rounding a given decimal number.

#### 6.6.4 Program generation and information sharing

I intended to specify system requirements at a level of detail that does not prescribe a specific technology for program data structuring and schema transformation. But for the first prototype, the Extensible Markup Language (XML) should be considered (W3C, 2003)—or a similar technology that is capable of implementing the features specified in the system requirements. XML is a meta markup language for text-based documents which contain data as strings of text. The types of schema and data configurations (organizations) mentioned in the system requirements are commonly used in the Internet for electronic publishing using XML. In addition, data and schema transformations are possible with XML that are increasingly being utilized in software systems that exchange a wide variety of data for online or local computing (Harold and Means, 2002).

#### 6.6.5 Production system shell

In case the experimental prototype designed and implemented by using OO technologies becomes ineffective in generating programs, the second layer of RaBBiT should build on a production system shell that can be relatively easy to integrate to the system. For the purpose of this study, I preferred to use the OPSJ<sup>1</sup> production system development environment (shell) from Production Systems Inc. (Forgy, 1998). The main criterion in

selecting the OPSJ is that it enables a full system integration of production rules with different object-oriented schemas defined in Java. The domain rules can recognize each object definition without having an interface application between rules and objects—however it still uses wrapper classes for representing domain objects in working memory. Secondly, the OPSJ—like the Java programming environment—is platform-independent and can run on different platforms. Third, like any application using Java, software programmers can take advantage of rich Java programming libraries. Finally, the inference engine of OPSJ uses the proprietary Rete II algorithm, a very advanced version of the Rete algorithm, which is the accepted industry standard for a pattern-matching algorithm needed by production system inference engines (such as CLIPS and ART uses Rete). The algorithm handles a large number of rules and data faster and more accurately than earlier versions (Forgy, 1998).

---

## **6.7 Summary**

In this chapter, I discussed software development technologies considering the unique needs of RaBBiT. For programming knowledge models and the definition of such models, I favor OO programming because of the technologies that it provides for flexible, extensible, and modular system architectures.

The data structure for representing requirements (or programming knowledge models) is defined as a directed acyclical graph. The MVC architecture is chosen as basic system architecture of RaBBiT. In addition, this chapter sets certain constraints on the implementation such as use of Java as programming language with its API, integrating JGraph and JEP subsystems. If the experimental prototype cannot perform program generation as specified, OPS/J will be used as production system shell for implementing a rule-based system for program generation.

---

1. A full version of the OPSJ is provided to me with no charge to be used in this study by the Production Systems Inc. ([www.psd.com](http://www.psd.com)). Other libraries and application frameworks are open-source and come with public-use licenses.





## Chapter 7 Developing RaBBiT

---

### 7.1 Behavioral and Structural Models

#### 7.1.1 Overview

One of the most challenging parts in designing systems for interactive knowledge modeling is providing effective and efficient interactions between the system and its users. This challenge is even more pronounced when the system has to provide a facility through which users can actually build a model of domain knowledge without having low-level computer programming expertise (Clark et al., 2001; Barker, Porter, and Clark, 2001). This is clearly the case with RaBBiT, a main objective of which has been to enable users to *build* programming knowledge models for building types and to generate architectural programs, using the terms of their choice without entering low-level (computer) programming code.

Ideally, the user-system interaction issues are addressed in the very early stages of software development and continue to be addressed throughout the process. Developers typically handle these issues during the specification of the *behavioral* and *structural models* of a system parallel to the design of its *graphical user interfaces* (GUIs). The present chapter briefly describes how I developed these models for RaBBiT and how I addressed interaction issues through a set of GUIs in the software development process. An investigation of software development (engineering) itself is out of the scope of this research.

#### 7.1.2 Behavioral models

Behavioral models capture "time-dependent" dynamic behaviors of a system defined by "logically correct sequences of interactions" and activated by "permissible events—externally visible stimuli [from users or automated actors] and responses [from the system]" (Booch et al. 1999, pp. 169). Developing behavioral models is one of the most essential part of object-oriented software design and implementation (Meyer, 1991). These models are typically depicted in multiple diagrams that show the system from different views at different resolutions—such as activity, sequence, state, collaboration (Meyer, 1991, pp. 85; Booch et. al., 1999, 233).

There are various software engineering strategies proposed for capturing a system's high-level behaviors: use-cases, scripts, scenarios, mechanisms, walk-through etc. (Coyne et al., 1993). Among these, the most reliable and proven-to-be-effective one is the *use-case* approach, particularly for the object-oriented design and development of an interactive system (Coyne et al., 1993; Armour and Miller, 2001; Flemming et al., 2001). Use cases are typically written in plain English following a pre-determined format, and their logical order and relations can be captured in a more formalized use case diagram, which also becomes the central part for modeling the behavior of a system—or a subsystem (Booch et.al. 1999, pp. 226 and 233). Based on my previous experiences in both teaching object-oriented programming and implementing various types of applications, I found the use-cases appropriate for developing RaBBiT as well.

A use case describes a "sequence of actions an actor (user) performs using a system to achieve a particular goal" (Rosenberg 1999, pp. 38) or "a sequence of actions a system performs that yields an observable result of value to a particular actor" (Booch et al. 1999, pp. 19). Leffingwell and Widrig (2000, pp. 135) describe the use-case approach as an "... integral to the software methodology [of] object-oriented software engineering... [which] is a way of describing a [complex] system's behavior from the perspective of how the various users interact with the system to accomplish their objectives." I will discuss the use-cases I developed for RaBBiT below.

### **7.1.3 Structural models**

Structural models describe the types of concepts in the system and the various kinds of static relationships that exist among them—for this reason they are also sometimes called *static models*. A structural model is gradually developed through three stages. In the first stage, domain concepts are explored in relation to the system requirements. This stage is followed by defining the classes corresponding to the domain concepts. However, it must be noted that there is often no direct mapping between domain concepts and the software that might implement them (Fowler and Scott, 1999); a domain concept can be addressed by multiple system parts. In this second stage, relationships and interfaces between classes are also studied. This includes formulating the associations and generalizations of classes. The third stage aims at increasing the granularity of the structural model for a particular implementation. Therefore, each class is described in detail (methods and attributes) and how its instances communicate with the instances of other classes.

Behavioral models and structural models are highly coupled with each other; structural models show a system's static view, while behavioral models represents the dynamics of a system. These two models are developed parallel to each other, except that the use cases come before the static model and govern the whole process and each of these models feeds

information to the other one. A class discovered in the structural model is studied in the behavioral models to determine its interfaces for communicating with other classes in reference to the related use case.

#### 7.1.4 Usability Considerations

Nielsen describes usefulness as "the issue of whether a system can be used to achieve some desired goal [task]" (Nielsen, 1995, pp. 24). Usefulness is broken down into two aspects: *utility* and *usability*. Utility addresses the system's functionality with respect to the user's needs. Usability concerns how well the functionality of a system can be used by the users. Usability is measured with respect to five quality factors: *learnability*, *efficiency*, *memorability*, *error* prevention and handling, and subjective *satisfaction* of the users (Nielsen, 1995). Dix et al. (1998) group these factors under three categories: *learnability*, *flexibility*, and *robustness*.

**Learnability** is a measure of how fast a system's users begin to use effective interactions to achieve maximal performance. It relates to the principles of predictability, familiarity, generalizability, consistency.

**Flexibility** refers to the multiplicity of ways the user and the system exchange information. There are several principles at work when it comes to flexibility: taking dialog initiative, supporting multiple tasks at a time, delegating the control between system and user, representing one concept in multiple ways, and customizing the GUI for different situations (adaptability and adaptivity).

**Robustness** relates to how successfully a system supports the achievement and assessment of the user goals. The core principles for robustness are observability of the internal state of the system, responsiveness of the system to the users actions, visibility and completeness of the system functions, and recoverability from an error—if error can't be prevented.

A good GUI design considers these factors and makes system use intuitive. It also corresponds to the user's mental model of how the system behaves. In the design of RaBBiT, these factors (and the principles they rely on) provide attributes for the measurement for usability.

---

## 7.2 Use-case Driven Software Development

### 7.2.1 Overview

In use case-driven software development, use cases are the primary information used in designing the GUIs of a system. They specify both the tasks a user performs and how the system behaves under these tasks (Armour and Miller, 2001). They help GUI designers in

understanding the sequence of the actions in accomplishing a task along with when and what information is needed by the system or by the users. Use cases also can be used as a benchmark for measuring the level of usefulness of a system along the aspects outlined above.

Use cases provide a common thread that runs through all software development phases (Jacobson et al. 1999). They specifically guarantee the desired system functionality through all development phases and to integrate all phases by giving them a shared focus that always keeps this functionality utmost in the developers' mind (Flemming et al, 2001).

### **7.2.2 Software development process**

The Unified Software Development Process (USDP)<sup>1</sup> provides effective methods for OO system development (Jacobson et al. 1992; 1999). This process is *use-case driven, architecture-centric, iterative, and incremental*. It is based on four phases in the following order: *inception, elaboration, construction, and transition*. Developers produce different "products" (defining the system) at every phase, and at every cycle the detail of the products increases.

I adapted an *agile*<sup>2</sup> version of USDP for developing RaBBiT. A full-scale use of USDP would be more appropriate for large development teams and for complex projects (Cockburn, 2002a). Although RaBBiT is also a complex project, managing its complexity doesn't require using USDP to its full extend: RaBBiT has been developed as a proof-of-concept prototype by one person in a limited amount of time. In this context, "agile" means effective, maneuverable, light-weight, and sufficient (Cockburn, 2002a).

In this agile version, I do not apply all the methods to document the software design in its entirety as it is prescribed in (Jacobson et al. 1999); rather, I keep the documentation just sufficient to guide the implementation of the system. For example, I intentionally drop most of the overlapping methods used in USDP or replace the formal representations of software concepts—or products in USDP terms—with simple (hand) sketches and notes; in most cases, these sketches were sufficient enough for implementation. I prefer to place the primary emphasis on the use-cases; and a secondary emphasis on implementation and incremental testing in small loops. I would like to emphasize that my approach (agile

---

1. A discussion of USDP and other processes is out of the scope of this study. For more information see (Jacobson, 1992; Jacobson et al., 1999; Leffingwell and Winrig 2000; Cockburn, 2002a; Cockburn, 2002b). For the UML see (Booch et al., 1998; Jacobson et al., 1999; UML, 2002; Fowler and Scott, 2002).

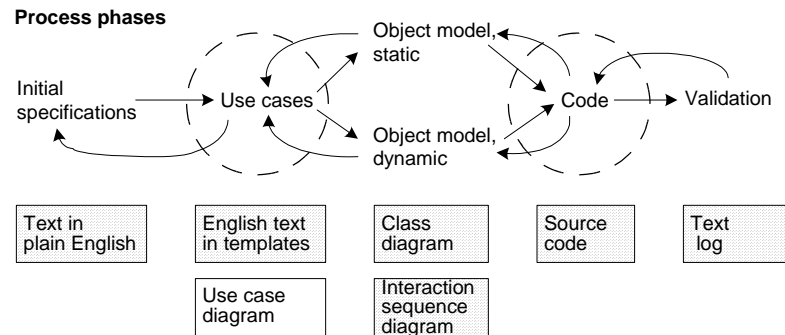
2. For agile software development refer to <http://agilemanifesto.org/> (last visited on November, 12 2003) and <http://www.agilemodeling.com/> (last visited on December, 12, 2003)

USDP) is not novel; it is inspired by other similar (and mostly overlapping) approaches such as described in (Cockburn, 2002b; Agile Modeling, 2003).

### 7.2.3 USDP and Unified Modeling Language (UML)

USDP uses the UML notation to specify a framework for system development and implementation (UML, 2002; Booch et al., 1998; Fowler and Scott, 2002). The notation comprises graphical figures to represent software architecture, modules, and functionality, where each of the representations embodies behavioral and structural aspects. These representations include use-case models, classes, behavioral and structural (object) models, interfaces etc.

Figure 7.1 shows the products that I obtained in each phase of the agile USDP. Each product corresponds to a concept in UML. It must be noted that I used these products for rapid system implementation; the quality of the formats I use is not of professional grade. In most of the cases they are sketches that are clear enough to capture an idea quickly and test its applicability in the implementation. Only after I implemented an idea successfully, did I clean up the corresponding product to be used in the next iteration. For example, I never worked on a complete set of use-cases that specify every aspect of the system, but documented only the essential use-cases. I will talk about this in the coming sections.



**FIGURE 7.1.**Phases and products of use case-driven software development (Flemming et al., 2001)

**Initial specification:** The initial specifications for RaBBiT include basic system and context requirements identifying the overall structural and behavioral features and constraints. I determined these specifications by studying the domain of discourse through the case studies and literature review<sup>1</sup>. Note that the final form of the initial specifications for RaBBiT has been established after several iterative loops, even after the system

1. The literature review is in Chapter 2, case studies is in Chapter 3, and system requirements is in Chapter 4 and 5.

implementation started. Therefore, they reflect the system requirements as they have evolved, not what was specified very early in the process. For example, initially I specified a *weight attribute* to be attached to each dependency association, but later I found out—after discussing with my advisor and implementation—that this attribute was not needed.

**Use case development:** The initial specifications yield to use-cases at the next phase. I described the use-cases semi-formally through several refinement iterations (see Section 7.3.1 for an example use case). These iterations provided a description of the desired system functionality in architectural programming terms. I even discovered new ideas or problems while developing the use-cases that I was not aware of during the initial specifications or implementation phases. For example, through a couple of iterations, I noticed that certain initially specified interactions between the users and RaBBiT were not possible to implement—or would increase implementation complexity. An example is the use-case for the expression editing. When users are editing an expression (parametric associations) they need to interactively select constructs located in components. This requires keeping both the expression editing dialog and the main window, where the component is located, active. However initially I specified in the use case that the expression editing dialog would be modal so that when expressions were edited, the other parts of the knowledge model could not be accessed. In the redefinition of the use case I removed the modal dialog constraint and added that only construct selection should be allowed on the model during expression editing.

**Object model (structural):** I defined the classes that are required in implementing RaBBiT in object-oriented programming. I arranged the classes following the MVC architecture as described in Chapter 6. The classes representing a programming knowledge model constituted the programming knowledge schema; the view classes defined the GUI part; and the manager classes are internal elements that manage the interaction between the instances of model and view classes. As a part of this phase, I developed a class diagram defining a schema for the application<sup>1</sup>. Note that the class diagram introduces a considerable number of "helper" classes that do not directly fall in any MVC sub-system. For example, for change propagation, I needed specialized objects capturing a *change* when it occurs. These objects are passed between all the other objects having an interest in this change. Discovering these kind of objects takes place during the iterative loops between system development phases.

The class structure of RaBBiT contains more than 180 classes (excluding the classes in third-party libraries). These classes are "packaged" in accordance with their

---

1. A complete set of class diagrams are included in the RaBBiT CD attached to this document.

responsibilities and the MVC architecture. Figure 7.2 shows some of the classes in their respective packages and the relationship between these packages.

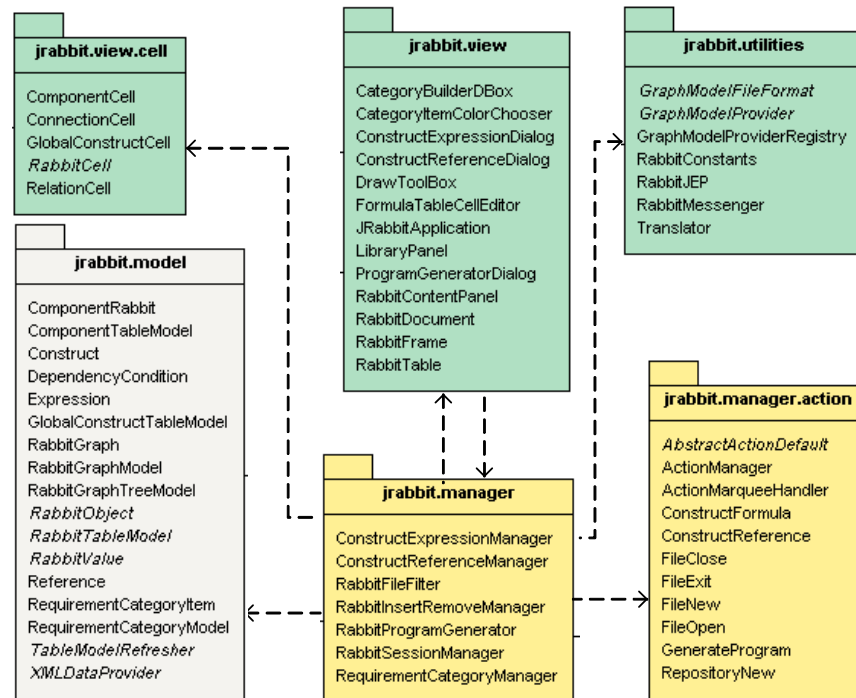


FIGURE 7.2. Class structure organized in accordance with the MVC architecture

**Object model (behavioral):** I developed the most essential interaction or sequence diagrams that depict which objects are involved in which use case and what part of their public interfaces are being used. These diagrams are particularly useful in defining the interaction between manager objects and view and model objects when users are performing a particular task. Through these diagrams, I was able to capture the required public interfaces (methods) in each class along with the information that needed to be passed between these interfaces. For example, again in change propagation, the objects in the knowledge model receive the change object through their public interfaces (method calls)—*public void update (ChangeObject change) {...}* is an example.

**Code:** In this phase, the code for all classes in the class diagrams was produced, *use case-by-use case*. This was crucial because I was able to test even partial implementations of use cases without considering the other parts. For example, when a GUI element was implemented, I was able to test its look-and-feel by actually displaying the GUI without implementing the functions (tasks) it delivered—i.e. without connecting the GUI element to its corresponding manager object, which may not exist at the time. I incrementally

implemented the steps of the action sequence defined in each use case in the GUI by connecting its methods to the manager objects' call-back methods. This way, I was also able to track the effects of each step of the use case on the system. When an unexpected effect occurred, I only checked the partial implementation, where the problem occurred most of the time.

**Validation:** This phase tests the code, again use case-by-use case.

These phases do not follow each other in a strict "waterfall" model. Rather, the process is meant to be highly iterative and incremental with numerous feedback loops.

---

## **7.3 Use-Case Descriptions**

### **7.3.1 Overview**

In our view (Flemming et al, 2001), use cases describe a meaningful task or result of value an actor (i.e. a user of the system in a specific role) may achieve. A meaningful task, in turn, is a sequence of actions or operations that must be executed together to achieve some goal. The task is self-contained in the sense that after the last task has been performed, an actor has choices in selecting a next task to execute. An example is the *Insert a Component* use case, which determines a specific task in knowledge modeling in RaBBiT. This task consists of a sequence of actions (select command, point location, define constructs, enter a name etc.) that must be executed if the goal of the task is to be achieved. It is self-contained because it does not determine what went on before or can go on after its execution. On the other hand, adjusting the view settings, in itself, would not be a use case because it is a task that has meaning only within another, larger task.

What constitutes a meaningful task for an application depends very much on the level of granularity at which the application is considered and use cases are formulated (Flemming et al., 2001). Selecting this level is a major design decision. In order to make system and GUI implementation flexible, I particularly favor the use of a *coarsely-grained* and *casual format* for studying use cases as opposed to *finely-grained*, *fully-dressed* and *formalized* formats. The selected format for the use-cases addresses neither the GUI component nor implementation, rather, for the sake of flexibility, it leaves these details to be resolved as implementation progresses. This is exemplified by the "*Insert A Component*" use case below.

**Use case name:** *Insert a component (requirement)*

**Use case category:** Knowledge modeling

**Primary Actor:** Architectural Program Modeler (APM).



**Description:** The APM inserts a new requirement (component) into the knowledge model by using the (graph) modeling area.

**Preconditions:** RaBBiT is running with a project open. If there is an active task (a task that is in progress before this use case invoked), it can be interrupted by this task, i.e. **the** active task is not modal. If information category levels are used, the APM has already selected a proper level for the new requirement.

**Flow of events (basic course):**

1. The APM selects the *insert component* command.
2. If there is an active task and cancellation requires confirmation, RaBBiT asks the APM's to confirm cancellation. If the APM confirms, RaBBiT cancels the active task; else the use case ends.
3. The APM indicates where to insert the new component on the GUI element used as knowledge modeling area<sup>1</sup>. The location itself is computationally not important since the geometric pattern of the graph is irrelevant.
4. RaBBiT creates a view and a model object for the new component with a default unique name (such as Requirement N).
5. RaBBiT switches to the component editing state.
6. The APM enters a name and description for the new component, if needed.
7. The APM can continue with the "*insert construct into a component*" use case or complete the present use case.
8. RaBBiT checks the integrity of the new component—such as uniqueness of name. If the new component is invalid, RaBBiT asks the APM to correct the problem by specifically pointing to the source of the error such as "The entered name <name> is already used in another component, please change the name".
9. RaBBiT inserts the new component into the knowledge model and updates all the views attached to the knowledge model.
10. RaBBiT registers the change in the graph for reverse operation (undo-redo) and switches to the normal modeling state.
11. The APM can insert another component by returning to step 3. The APM can also cancel the use case by selecting a new command.

**Post-condition:** A new component becomes part of the model and is displayed in views.

**Alternative flows or exceptions:** (not shown here)

---

1. Note that the physical design of the UI elements are not specified yet.

The use cases defining the behavioral model of RaBBiT are grouped under four categories: (a) session control, (b) knowledge modeling, (c) program generation, and (d) graph view manipulation (including view control and formatting). Below, I summarize the use cases under first three of these categories. Note that the following describes only the critical use cases; there are other use cases that provide accessory functions such as component repository building, automatic graph layout, grid-snap setting etc. that are not mentioned here.

### **7.3.2 Session control use cases**

These use cases specify the interaction between the users (the APM or Architectural Program Composer (APC)) and RaBBiT during a knowledge modeling session. Each knowledge model is called a *RaBBiT Project* in the session, or in short *a project* in these use cases. The APM can save a project at any time during the session. In addition, a project can be loaded, saved, or renamed in different formats, such as Extensible Markup Language (XML), at any given time during a knowledge modeling session. The session control use cases also specify printing a knowledge model to the system's printer or capturing the graph as image. The following use cases describe how these tasks are accomplished by the APM in interaction with RaBBiT.

#### **Start a new session**

The APM or APC initializes a new session in the computer by giving the operating system a command reserved for RaBBiT. In a window-based system, the users can start RaBBiT by double-clicking on the icon representing a short-cut for accessing the RaBBiT executable file. When a RaBBiT session starts, RaBBiT displays the main window with no project loaded (Figure 7.3a). The APM can initiate a new programming knowledge model definition or open a persistently stored knowledge model for editing at this point. Therefore, this use case is "uses"—in UML terms—two additional use cases: *start a new project* and *open an existing project*. The APC is not responsible for creating a new project, but opening an existing one, which is similar to opening a text file in a text editor.

#### **Open an existing project**

The APM or APC can open an existing project by locating the project in the computer's persistent storage devices in a standard file-dialog. In case a project is already loaded before issuing the open command, the APM is given options to close the current project; if the current project has changed since last save operation, the APM can save the current project before opening another project or can cancel opening the existing project. If no knowledge model is being edited as the open command is issued, RaBBiT loads the selected project (Figure 7.3b).

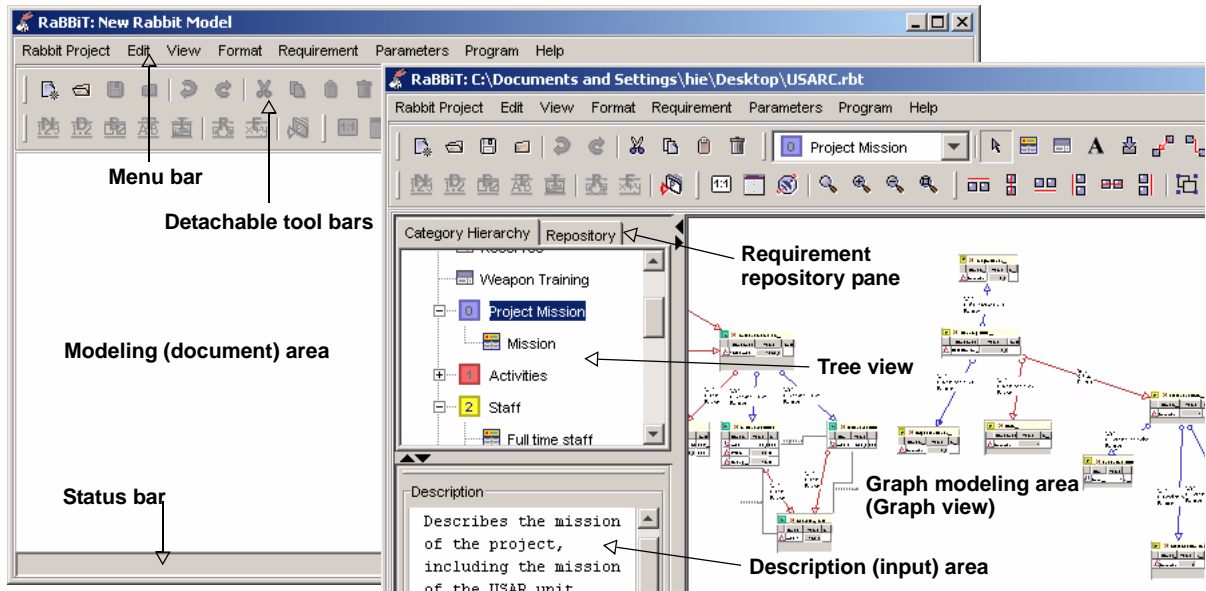


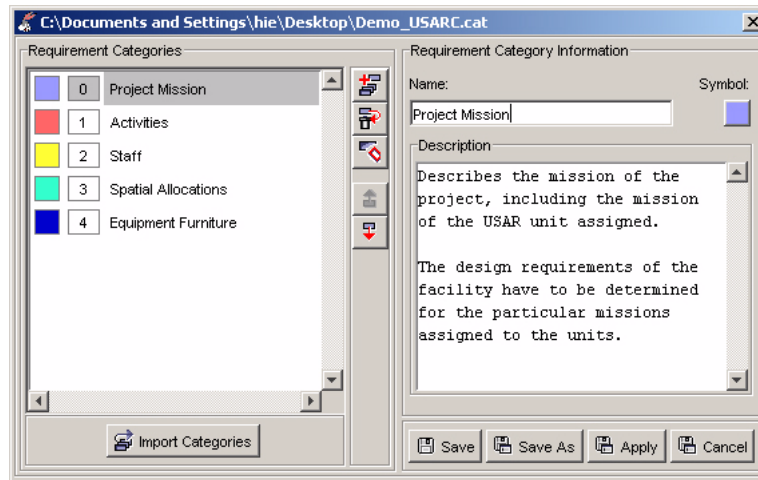
FIGURE 7.3. RaBBiT's main window (a) without a project and (b) with a project loaded.

### Start a new project

After a session starts, the APM selects the *new project* command from the given GUI options (a button in a toolbox or a menu item from a pull-down menu). As a new project is initiated, RaBBiT displays a dialog for defining a model for *requirement information category levels*. If the APM prefers this, the use case relating to modeling information category levels can be skipped and knowledge modeling starts without using category levels.

### Define a model for requirement information category levels

The APM can either build a new category level model or import an already defined and persistently stored one from the system. The APM can design a category level model according to any architectural programming approach or design guidelines; RaBBiT can save this model to be used in other projects, if the APM chooses. If category levels are required, they are used to group requirements information from higher to lower levels. This use case is extended by various other use cases: *add a category level*, *remove a category level*, *change the level of a category*, *save (or save-as) category level model*, *import a category level model*, and *apply the current category level model to the new project*. When the last use-case is invoked, RaBBiT starts a new knowledge model by adopting the current category model. Figure 7.4 shows the dialog box designed to implement these use-cases.



**FIGURE 7.4.**Dialog box for requirement information category level modeling

### Save a project (knowledge model)

The APM can save a project at any given time when the active function is not modal. If the project is saved for the first time, the APM is asked to choose a location (in the system's persistent storage devices) and enter a name for the project. Only when both pieces of information have been entered, RaBBiT saves the project and displays the name of the project, for instance, on the window title. The APM must be able to save the knowledge model at least in two formats: a binary or XML serialization. The extension of the file for the first format must be unique, such as *rbt*; the extension of the file for the second format must be *xml*. For renaming an open file, the APM interacts with the system in a similar way.

### Close a project

The APM can close a project at any given time when the active function is not modal. When the APM chooses the close command, RaBBiT checks if the project has changed since the last save operation. If so, the APM is given a chance to save the project or cancel *close a project* use case. When the project is closed, RaBBiT displays an empty modeling area and becomes ready to initiate editing a new or existing project.

### Exit a session

The users (the APM or APC) can stop an active session at any given time by choosing an *exit* command, provided that the active function is not modal. When the exit command is issued, RaBBiT checks if any change has been made on the model. If this is not the case, RaBBiT terminates; otherwise, the APM or APC can save the file before exit, return back to editing, or proceed with exiting without saving the project.

### 7.3.3 Knowledge modeling use cases

These use cases specify the interaction between the APM and RaBBiT when the APM is modeling programming knowledge. The terms used are defined as follows:

**Component:** an object representing a requirement or a programming concept

**Global construct:** an object representing a critical programming parameter

**Construct:** an object representing a programming parameter that belongs to a component

**Value:** an object of simple type (text, number, boolean) or complex type (resource location, reference, or parametric association) assigned to a construct to represent its value.

**Dependency:** a connection object between two components showing dependencies with direction and condition.

**Relation:** a connection object describing a relationship between two components

**Expression:** an object representing a parametric association assigned as a value to a construct.

**Reference:** a value assigned to a construct that is read from another construct's value.

**Category level:** an object that is used to classify a component according to requirement information or programming concept level

**Category hierarchy:** an object model defining an order for classifying programming information represented by components from higher- to lower-levels.

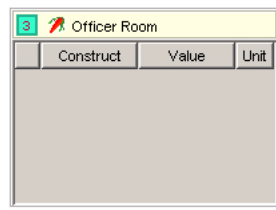
**Knowledge model:** a complex object model with a graph structure that is used for composing programming information represented through components, constructs, global constructs, dependencies, relations, and, if defined, category levels ordered in a category model.

#### Create (Insert) a component

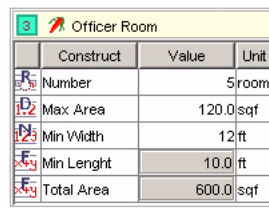
The APM selects the *create component* command from one of the GUI element handling this function. The APM locates the position in which the new component will be inserted. The command is observed by a corresponding manager object in RaBBiT that creates a view of the component along with a component object to be stored in the knowledge model. As the component is inserted, RaBBiT switches to the component editing state (uses *edit component use case*). The component has to have a unique name describing the concept or the requirement that it represents. The use case can continue with the *insert constructs* use case or the APM can *abort* this use case. In case the APM prefers to define constructs later, the APM can do this in another editing step. The APM can also enter a

description for the new component. This can take place either when the component is inserted or in a separate editing use case. In case category levels are used, RaBBiT assigns the new component to the category level selected before this use case initiated. RaBBiT updates all the views of the knowledge model as the APM inserts the new component.

The GUI object representing a component object is determined in the physical design phase of the GUI; sample component views are shown in Figure 7.5. The *header* shows the name of the component, and the box with a number next to the name shows the category level for this component with the color code of that level.



(a)



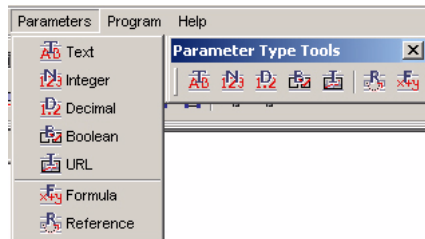
(b)

	Construct	Value	Unit
Number			5 room
Max Area		120.0	sqft
Min Width		12	ft
Min Length		10.0	ft
Total Area		600.0	sqft

**FIGURE 7.5.**A sample component's views (a) without and (b) with constructs.

### Insert constructs into a component

The APM can initiate this use case either during the *insert component* or *edit component* use cases; that is, a component must be in the editing state. In UML terms, this use case is an *abstract use case* and specialized by the *insert a construct with* (a) a numerical, (b) a boolean, (c) a text, (d) a resource, (e) a reference, and (f) an expression (formula) value use cases. The APM initiates all of these use-cases in a similar way by invoking an action on the corresponding GUI component from a RaBBiT view (such as menu item or button). The GUI elements initiating these use case are implemented as shown in Figure 7.6.



**FIGURE 7.6.**The GUI elements for invoking the insert construct use cases.

Use cases a-c are straight-forward and follow an identical interaction sequence: The APM selects the corresponding GUI object to initialize one of these use cases and RaBBiT inserts a construct object both in the view and the component being edited—with a default value and name. The APM can change the default value or name or edit them during a separate editing step. These use cases are not described here in greater detail. Figure 7.5b

shows a component containing a reference, a numeric (with decimal points), and two expression constructs in a table below the header part.

### Insert a construct with a resource value

The APM can insert a construct into a component that has a value pointing to a Unified Resource Locator object (URL) (W3C, 2003)—a file of any type persistently stored in the system, including the local network or Internet. Essentially, this action can be used if an external resource describes the programming concept or requirement (component) in more detail in a data format that RaBBiT cannot handle, such as in external text documents, images, computer-aided design (CAD) files etc. After the APM inserts the construct, the APM can enter the location of the URL as the value of the construct; this also can be done through a standard file browsing dialog.

### Insert construct with a reference value

This use case defines how the APM assigns a value to a construct that it is going to be read from an existing construct. The use case starts similar to inserting any other construct and continues with the APM interactively selecting a component that contains the construct whose value will be read. RaBBiT makes this component and its constructs selectable by pointing. The APM points to the construct, and a new construct referencing the value of the selected construct is inserted into the component currently being edited. The view that displays the component with the referenced construct is shown in Figure 7.7a.

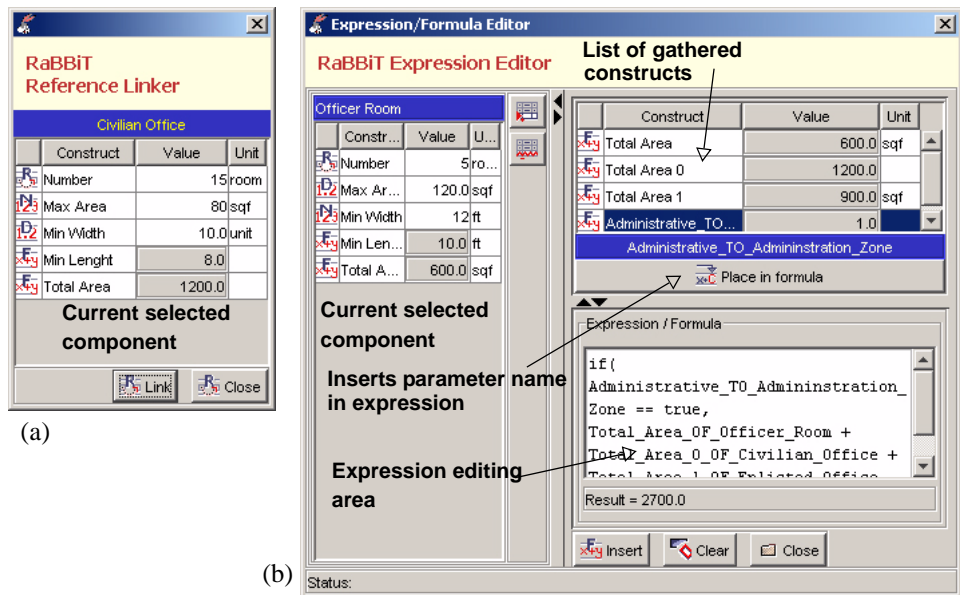


FIGURE 7.7. Views for inserting (a) reference and (b) expression values for constructs.

### Insert construct with an expression value

The APM issues the insert expression command and RaBBiT opens the expression editor dialog box. The APM gathers constructs to be used in the expression by selecting the appropriate components and constructs from the model. In the dialog box, (Figure 7.7b) RaBBiT shows a list of the constructs gathered interactively. The APM can add other constructs to the list at any time during expression editing. Expressions are formulas with simple syntax similar to the one used in spreadsheets. The APM can either type these expressions or build them interactively by selecting constructs from the construct list, after which RaBBiT inserts the name of the selected construct into the expression. The result of the expression is computed as it is defined and becomes the value of the new construct. RaBBiT continuously informs the APM of any errors in the expression.

An example is the expression for calculating the total area of the *administrative zone* shown in Figure 7.12. This zone may contain *officer rooms*, *civilian offices*, *enlisted offices* based on the dependency conditions in the dependency associations from the administrative zone to each of these spaces.

$$\begin{aligned} &(\text{if (Administrative Zone TO Officer Rooms == true, Total Area OF Officer Rooms, 0) +} \\ &\text{if (Administrative Zone TO Civilian Offices == true, Total Area OF Civilian Offices, 0) +} \\ &\text{if (Administrative Zone TO Enlisted Offices == true, Total Area OF Enlisted Offices, 0)}) \times \\ &\text{Circulation Area Coefficient OF Administrative Zone} \end{aligned}$$

This expression shows how a nested parametric association can be formed. Administrative Zone to Officer Rooms refers to the conditional association included in the dependency association from the Administrative Zone to the Officer Rooms. If this condition is satisfied, the (parameter) Total Area of requirement Officer Rooms is added in the Total Area of Administrative Zone. At the end of the expression, the sum of the areas is multiplied by the Circulation Area Coefficient included in the Administrative Zone (requirement), which returns a final value for the Total Area of Administrative Zone. Note that the expression editor manager is responsible for reading the names of the parameters or the requirements that were interactively selected by the APM—which reduces the cognitive load in remembering the names of the parameters. However, the APM can also enter these names manually. Note also that when the name or value of a parameter changes at any time during knowledge modeling, RaBBiT propagates these changes to each expression having reference to that parameter.

### Insert a global construct

The APM inserts global constructs in a way that is similar to inserting components, but with one difference: A global construct contains only one construct object. Please note that global construct objects are not allowed to have dependency or relation associations, but can be used in parametric associations.





**FIGURE 7.8.** Boolean and numeric global constructs

#### **Insert an association between two components**

This use case defines how the APM inserts a dependency or relation association between two existing components. In general, the APM follows the same steps for creating either type of associations. First, the APM activates the appropriate association command, then selects an existing component representing the source of the association. As the APM moves the mouse over the target objects, RaBBiT checks if the component under the pointer is a legal component for this associations. RaBBiT continuously (dynamically) informs the APM of this and tells the APM from which component to which component the association will be created (Figure 7.14 on page 112). When the APM locates the target component, RaBBiT creates a model and a view object for the new association and switches to association editing mode for (a) the dependency condition, if a dependency association is created; or (b) the relation label, if a relation association is created. The APM edits dependency conditions by using the expression editing dialog box and the relation labels "in-place". If the dependency condition evaluates to "false", the dependency symbol (lines) are rendered blue and red otherwise.

#### **7.3.4 Program generation use cases**

These use cases describe how the Architectural Program Creator (APC) and RaBBiT work together in generating programs for a project by using a previously defined knowledge model. Since the generation process is carried out internal to RaBBiT, the interaction between the APC and the system is very simple and can be described in three main use cases: *provide project-specific information*, *modify global parameters*, and *generate a program*. The precondition for all of these use cases is that a knowledge model must be open and the APC has issued the program generation command, which displays the program generator dialog box (Figure 7.9).

##### **Provide project-specific information**

The APC enters project name, location, and ID in RaBBiT's program generator dialog box. In addition, the APC inputs client information such as name, phone number, e-mail, and address. Each program generated for a project contains also version information. The generation date is assigned by RaBBiT, and the APC must enter version number and description. When RaBBiT generates a program, it places the project-specific information at the beginning of the document.

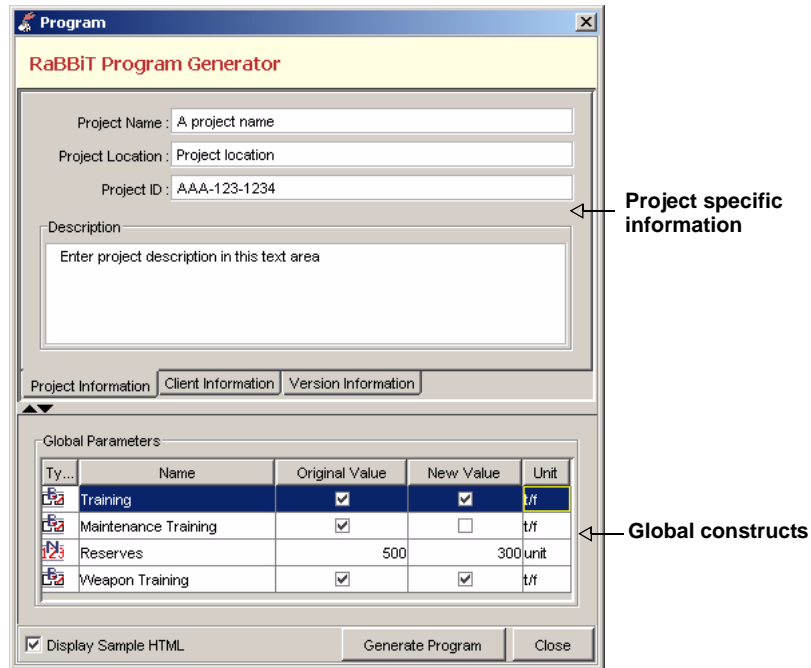


FIGURE 7.9. Program generator dialog

### Modify global parameters

Before RaBBiT displays the program generator dialog box, it gathers all of the global construct objects (critical parameters) that govern the program generation. The box presents these constructs in the rows of a table such that each row contains information about one construct. The original values of global constructs as defined in the knowledge model and a new value input area placed side by side. RaBBiT directly links the new values to the global constructs in the knowledge model so that when the APC enters a new value for a construct, RaBBiT updates the model and its views. Therefore, the APC interactively can investigate the results of changes in the model.

### Generate a program

The APC issues the generate program command in the program generator dialog box. By applying its internal program generation algorithm<sup>1</sup>, RaBBiT generates a program as output. RaBBiT asks the APC to select a location and a name for the file in which the program will be stored. Note that this file is not in a view-dependent format; it contains only plain program data. In order to view the program in a formatted display, the APC can choose to produce a sample view as well. If this occurs, RaBBiT generates a sample view

1. Appendix B includes program generation algorithm, a sample generated program for a USARC, and its view in html.

of the program (in html form), finds an application by which to display the view (e.g. default web browser), and opens the view in the application.

I designed a (architectural) program schema (Figure 7.10) using XML following the system requirements and the use cases defining program generation. RaBBiT generates programs (data) in XML format following this schema. In addition, I designed a sample view schema in XML Style Sheet format (i.e. xslt) for program (data) presentation in HTML form. After a program is generated, RaBBiT uses Java's XML transformation package to transform the generated program into HTML form as defined in the XML Style Sheet. Figure 7.11a shows a part of the program data ("full-time staff") in XML form, and Figure 7.11b shows a view of that part in a web browser—which RaBBiT opens to display the generated HTML file.

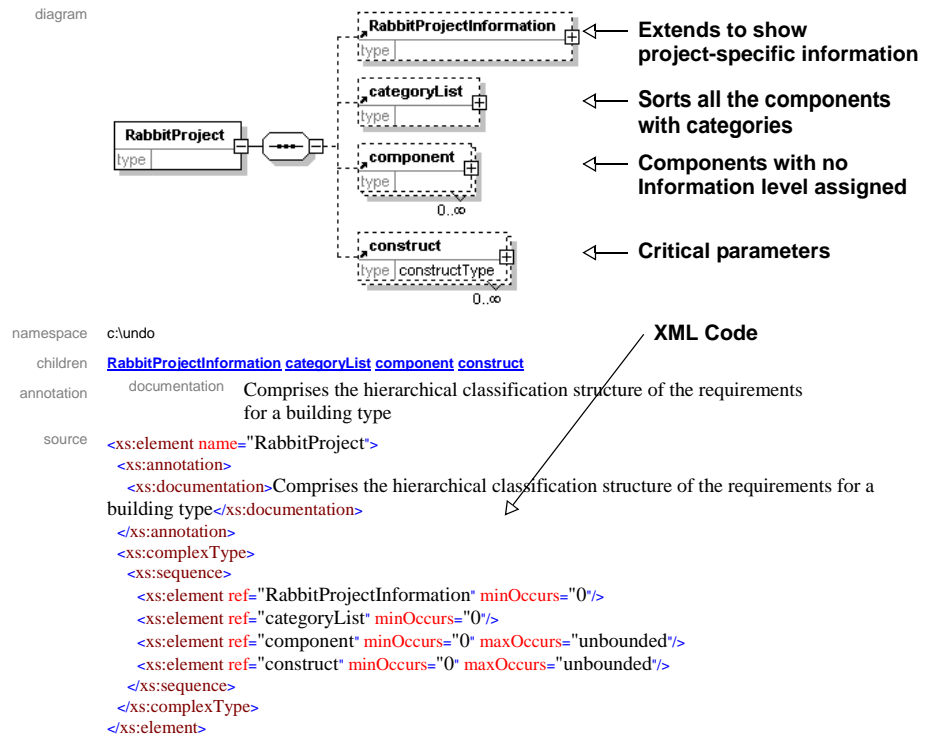


FIGURE 7.10.A part of the programming schema in XML definition

## Developing RaBBiT

```

- <category>
  <categoryName>Staff</categoryName>
  <categoryDescription>An Army unit ...</categoryDescription>
  <categoryLevel>2</categoryLevel>
- <component>
  - <componentInformation>
    <componentName>Full time staff</componentName>
    <componentDescription />
    <categoryName>Staff</categoryName>
  - <constructLocalAs>
    <constructName>Officers</constructName>
    <constructDescription />
  - <constructObservedBy>
    <componentName>Full time staff</componentName>
    <constructName>Civilians</constructName>
  - <value>
    - <valueComplex>
      <type>Expression</type>
    - <valueExpression>
      - <value>
        - <valueSimple>
          <type>integer</type>
          <integer>6</integer>
        </valueSimple>
      </value>
      <expression> roundup (Officers_OF_Full_time_staff * 2)
      </expression>
    - <valueReadFrom>
      <componentName>Full time staff</componentName>
      <constructName>Officers</constructName>
    - <value>
    - <valueComplex>
      <type>Expression</type>
    - <valueExpression>
      - <value>
        - <valueSimple>.....

```



Constr...	Value	U...
Officers	3	
Civilians	6	
Enlisted	9	

Category: Staff

Information Level: 2

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full time staff

Requirement: Full

FIGURE 7.11. Partial program data and its sample view as generated by RaBBiT

---

## 7.4 System-User Interaction

### 7.4.1 Conceptual and physical design of GUIs

In order to achieve a high usability level for RaBBiT, the UI design process is divided into two distinct phases: *conceptual* and *physical design* as described in (Jacobson, 1999). Conceptual design focuses on the features necessary to reflect the functionality of the system and is based on the needs of the system and the user in their interaction with one another (Armour and Miller, 2001). Physical design addresses the selection and composition of user interface components based on the conceptual design. Although logically distinct, the two phases are coupled with each other and form a larger iterative process.

### 7.4.2 Direct-manipulation paradigm

The direct-manipulation paradigm was first introduced by Shneiderman (1982). It is defined as an interaction technique characterized by rapid feedback, visible domain objects, and incremental and reversible actions. The most significant feature of direct-manipulation interfaces is the replacement of complex command languages with actions that manipulate *directly visible (domain) objects*. Later, Hutchins and his colleagues provided a more cognitive justification for direct manipulation based on a *model-world metaphor* (Hutchins et al, 1986). In this metaphor, the UI is not only a medium between the user and system, but represents the system from the user's perspective. They call this a *user-centered UI* that "is itself a world where the user can act, and which changes state in response to user actions...Appropriate use of model-world metaphor can create the sensation in the user of acting upon the objects of the task domain themselves." (Hutchins et al., 1987, pp. 87). They call this form of interaction *direct engagement*.

The direct-manipulation paradigm is commonly used in current computer applications, especially in those that heavily rely on (visual) graphical information. The benefits of using direct manipulation are not limited to manipulating domain objects interactively, but extend to interactive interactions that allow users to simultaneously observe the consequences of an action (Dix et al, 1998).

### 7.4.3 Interaction style in RaBBiT

I believe that for the GUI of RaBBiT, direct manipulation is appropriate for the following reasons: (1) the system can take advantage of inherent benefits provided in the direct-manipulation paradigm and model-world metaphor; and (2) architectural programmers with a design background are mainly trained how to organize visual information—such as affinity diagrams and spatial layouts. A well-configured user interface and appropriately

designed system-user interaction based on direct manipulation paradigm can contribute to increasing the usability of the system for these types of users.

RaBBiT's components, constructs, and associations can be treated as domain objects created by the user through a set of direct-manipulation GUIs. We can observe similar types of interaction in the tools used for object-oriented system modeling—such as supporting UML—where classes, their attributes, class generalization, and associations are defined interactively in a graphical *modeling area*. Using a modeling area similar to these, RaBBiT users can specify requirements and their relationships pertaining to a building type by interactively manipulating the knowledge model.

#### **7.4.4 Model-world metaphor for RaBBiT**

I think that the most appropriate model-world metaphor for RaBBiT is *graph modeling of requirements*. A graph is made up of nodes and edges connecting nodes as mentioned in Chapter 5. In representing a programming knowledge model, there will be two types of nodes: components and global constructs. The former encapsulate a requirement or a programming concept; the latter represent the critical programming parameters. Dependency and relational associations can be shown as lines connecting nodes on the graph. From the associations point of view, the graph will be made up of two overlapping sub-graphs. The dependencies form an acyclic directional graph and are shown by special lines indicating the direction from a source to a target requirement. Each dependency will also include a dependency condition. The relational associations, on the other hand, form a sub-graph that is bidirectional and cyclic. The relational associations are represented by lines with labels indicating the relationship between two requirements.

In RaBBiT's UI, the user will be able to manipulate a programming knowledge model through the nodes and lines of a graph. Manipulation of the graph includes creating nodes, setting connections (associations) between nodes, composing nodes and connections in groups, relocating existing groups, changing the nodes or associations individually etc. Related to graph manipulation are ancillary functions for *view manipulation and formatting*—such as zoom in and out, and pan.

Parametric associations, on the other hand, are too complex to be shown using nodes and connections in the same graph. These associations could *pollute* the view of the graph such that users can no longer assess the integrity of the view—and the model for that matter. A solution is to use *tables* that can be modified through a *spread-sheet-like* interaction. The parameters of each requirement can be represented as the rows of a spreadsheet table; the tables, in turn, can be shown in the nodes of the graph. This can contribute to the usability of the system because (a) spread-sheets are commonly used in

architectural programming and a similar interaction will increase learnability, and (b) it provides a sub-metaphor that reflects the nature of parametric associations.

Figure 7.12 shows a part of the graph representing the knowledge model in programming USARCs. Each node represents a requirement that encapsulates its parameters in a spreadsheet-like table. Dependencies are represented as lines with arrow heads pointing in the dependency direction. Each line has an attached a *box* showing the dependency condition (C), and result of the condition (R). Relationships are shown with routing lines and attached labels.

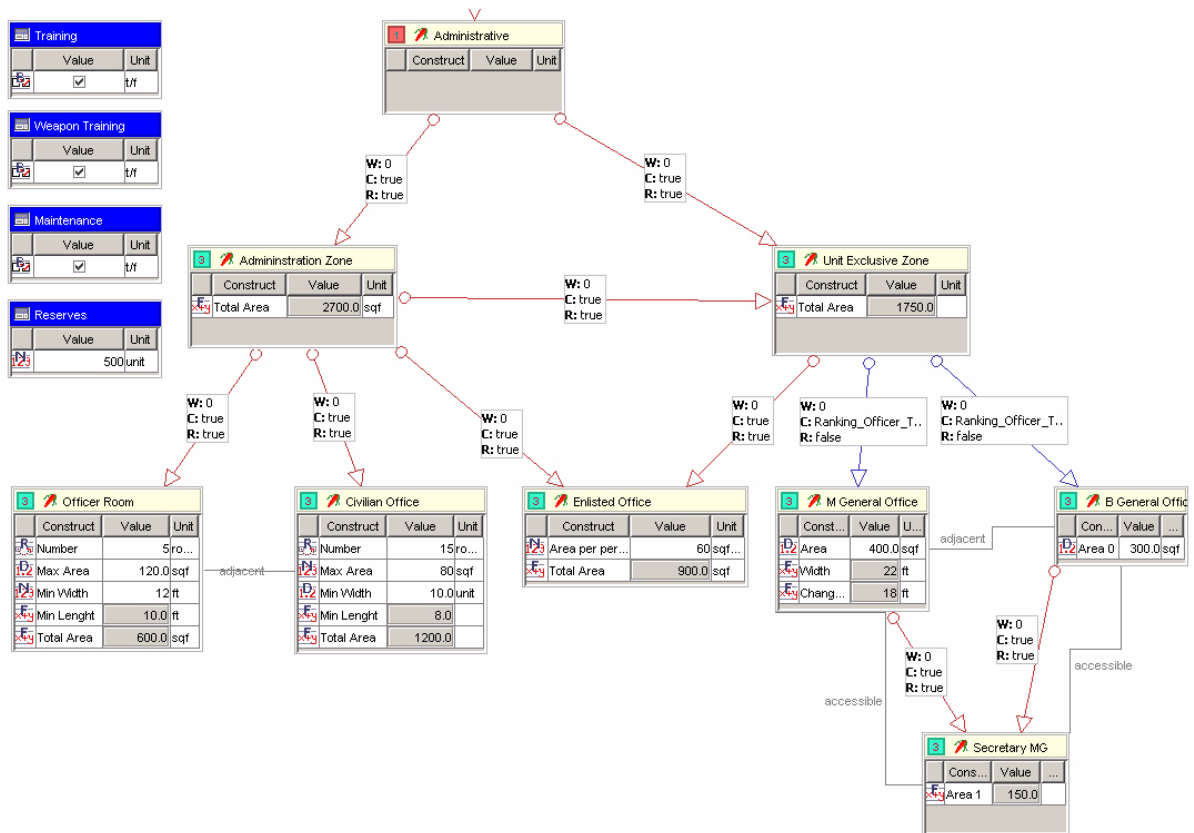


FIGURE 7.12.A snapshot from the graph representing a partial knowledge model.

## 7.5 GUI Design of RaBBiT

### 7.5.1 GUI Composition of RaBBiT

The GUIs designed for RaBBiT correspond to the *view* sub-system of the MVC architecture mentioned in Section 7.1.3. Each function or set of functions has a corresponding *view*. When the use-case model is considered along with these views, both

the conceptual and physical design of RaBBiT's GUI emerge. A snapshot of the physical design of RaBBiT's user interface is shown in Figure 7.13.

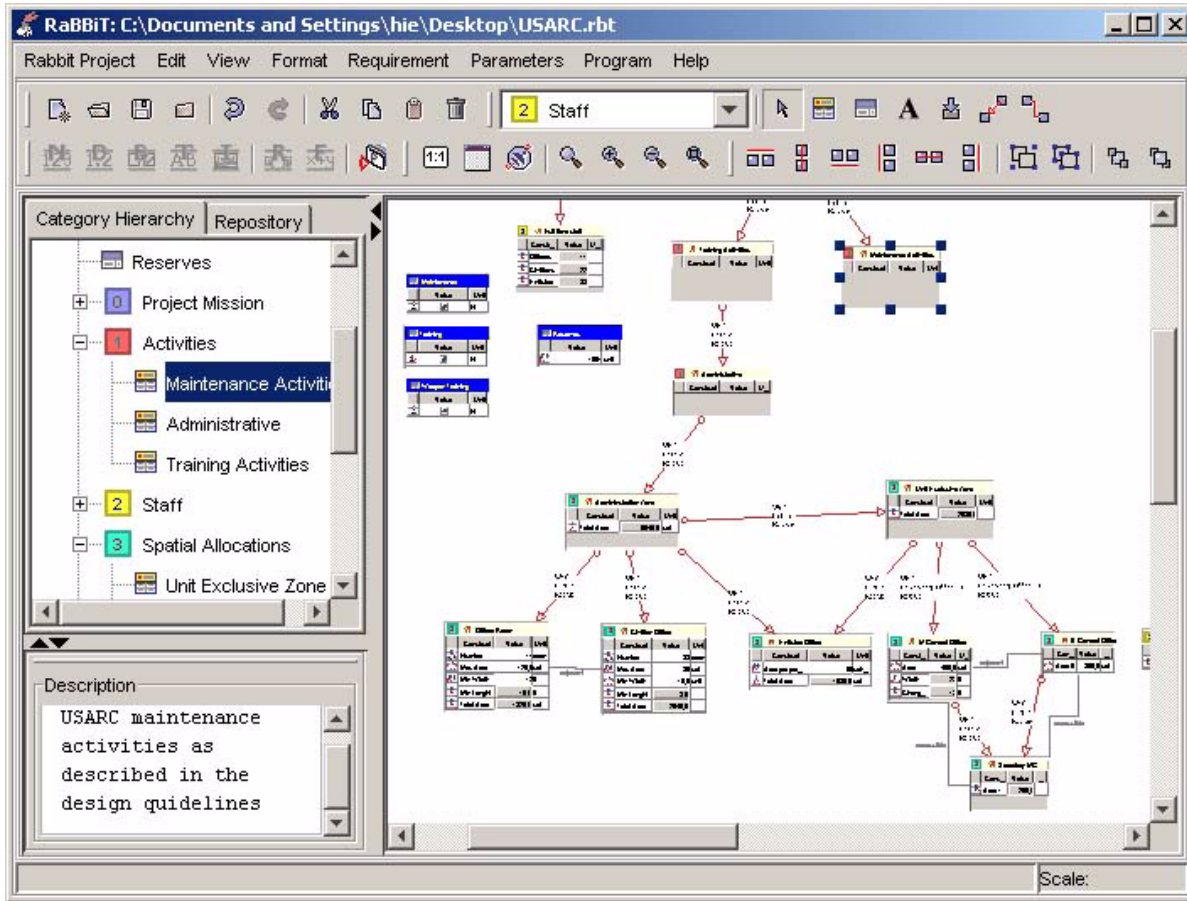


FIGURE 7.13. Snapshot from RaBBiT's main window.

Like any standard window-based application, RaBBiT includes pull-down menus and (detachable) toolboxes for invoking a desired function (described in the respective use case in the conceptual design of the GUI). The graph modeling component is divided into two parts for displaying different views of the graph model: a tree-view and a graph view. Both of these views can be directly manipulated.

The tree-view area is used for displaying all requirements with respect to the information category level under which they fall. The same area contains a tab labeled *repository* that displays a *requirements repository*<sup>1</sup> view. The users can store and retrieve requirements

1. Since this is an ancillary feature, it is not specified in the system requirements. It emerged as a side effect of the iterative system design process.



across different projects by using this area. A detachable (floating) area that can be used to enter descriptions is placed below the tree-view area.

The graph-view is the main interaction area where a knowledge model is build. The status bar at the bottom of the window informs the user of the state of the system or the consequences of actions. During knowledge modeling, several dialog boxes are displayed to assist the interaction between RaBBiT and its users. I will talk about these in the next chapter.

### **7.5.2 Usability heuristics for RaBBiT**

In developing RaBBiT, I followed GUI design principles and heuristics that are essential for forming a base-line for a "user-friendly" system; these principles and heuristics are based on Nielsen (1995) and Dix et al (1998). Coupled with the use cases, they provided the guidelines for developing the UI and system-user interaction for RaBBiT.

The following usability heuristics were observed in designing RaBBiT's GUI.

**Visibility of the system status:** The users of RaBBiT are kept informed by continued feedbacks such that most of the changes done on a model are accompanied by simultaneous updates in the relevant user interface components. For example, the active function is both highlighted on the selected tool and displayed on the status bar. When editing a requirement is completed, the result of editing synchronously becomes visible in the GUI object representing the requirement.

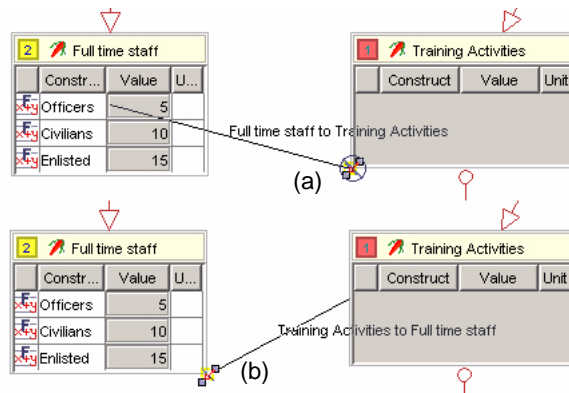
**Match between system and the real word:** The system uses a language and concepts familiar to the user. The users are given flexibility to use their own terms in naming requirements and their associations. In addition, RaBBiT uses *graph-modeling* and *spread sheet table* metaphors for the requirements building process, which, I think, will help the user to form an *appropriate* mental model of the system (see Figure 7.12 on page 109).

**User control and freedom:** The users are able to cancel any active tasks through standard actions that are common to other window-based applications—such as pressing the escape key or mouse-click on an empty area cancels the active function. Furthermore, most of the actions that have been implemented are reversible through undo-redo functions. Due to performance-related considerations, I implemented RaBBiT to handle undo-redo actions only 10 steps back; however, this can be increased up to an arbitrary number of steps by modifying a value in a resource file.

**Consistency and standards:** Although I have not explicitly specified guidelines for consistency, I intuitively followed a schema that most window-based applications use. The user interface design is consistent in terms of using symbols, colors, words, situations, and actions across the application. However, consistency is not blindly followed; if there is an

exception, the consistency is violated. In addition, users are given an option to choose their own color schema for color-coding information categories. The color code is displayed both on the branches of the tree-view of the model and on individual components (see Figure 7.13 on page 110).

**Error prevention and recovery:** RaBBiT's internal logic is designed to follow certain error prevention and recovery rules in modeling. For example, a cyclic dependency relation is not allowed when the user is configuring dependencies between multiple requirements (Figure 7.15a). If the user's action violates one of these rules, the system interactively informs the user of the violated rule by displaying visual (GUI) clues. For example, the icon at the mouse pointer changes based on whether a dependency association being created is legal or illegal (Figure 7.14a and b). In this way, RaBBiT first tries to prevent an error from occurring, and if an error cannot be prevented, users can recover following the suggested solution displayed in error or warning dialog boxes. For example, the box shown in Figure 7.15b is displayed when the user attempts to delete a parameter that is being referenced in parametric associations. However, this feature is implemented for only the essential modeling functions. For example, error prevention is not provided for automatic graph layout, although error recovery (through undo-redo functions) is available.



**FIGURE 7.14.** Mouse icon changes for error prevention (a) Illegal and (b) legal dependency association.

The users can interactively change the source or target of a dependency association by dragging either end of the association object onto the new requirement. If the new association is illegal—because of a cyclic relation or duplicate associations between two requirements—the user is warned of the situation, and RaBBiT automatically changes the model back to the state before the illegal operation.

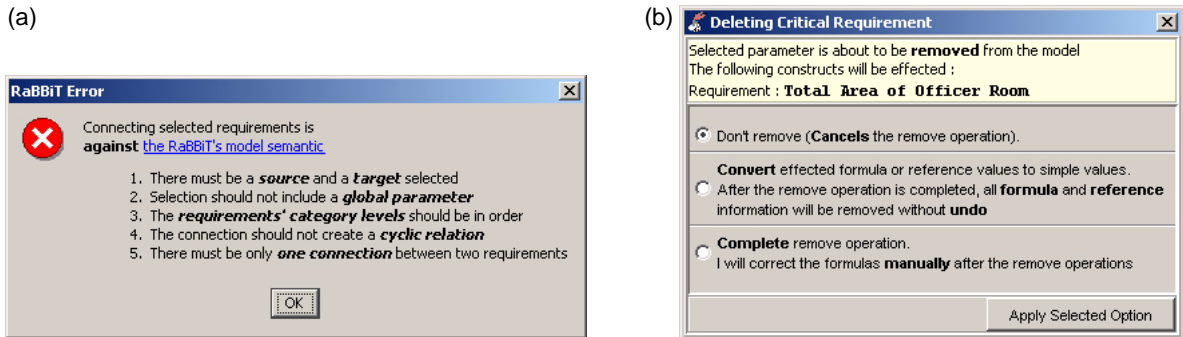


FIGURE 7.15. Sample dialogs for (a) rule violation and (b) error prevention.

**Recognition rather than recall:** Interaction between the user and RaBBiT's UI is based on the direct manipulation of visible objects; hence, users do not have to remember the syntax of the commands. All the actions and options become visible to the user during visual object manipulation. For example, when creating a requirement node (component), the user selects a button (or menu item) with an icon that is a visual abstraction of the UI object to be created. The mouse pointer icon also changes with respect to what the user and system are doing at a given state.

RaBBiT can recognize the active function and its context, enable all the relevant functions, and disable the irrelevant ones that may cause an error if activated.

**Flexibility and efficiency of use:** The system tries to accommodate both experienced and inexperienced users. Therefore, RaBBiT's UI provides shortcuts or accelerators for invoking functions by experienced users. Figure 7.16 shows a pull-down menu with menu items and their accelerator key-combinations.

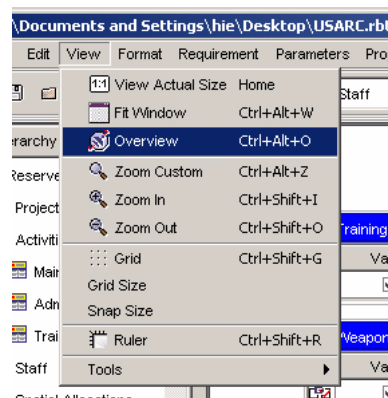


FIGURE 7.16. Shortcuts and accelerators for experienced users

**Aesthetic and minimalist design:** RaBBiT's GUI displays only relevant information; redundant information that may confuse the users is hidden; for example, parametric associations are only visible when requested. The users can also hide functions (on the tool bars) which are not used as often. Colors, size of interface components and their configuration are carefully chosen to form a simple, yet pleasing design. The APM can adjust the size of the component and construct (visual) objects on the screen.

**Help and documentation:** Although this is an important feature to consider, due to time limitation, RaBBiT's documentation (manual) has not been completed.

Testing the usability of RaBBiT for a real-life case has also not been completed due to time and resource limitations.

---

## 7.6 Summary

In this chapter, I described the strategies that I used in developing RaBBiT. One of the most important decision was to use a custom-tailored software development process that provides methods for OO programming, fits the limitations and the nature of the application, and offers flexibly where needed. This process was based on agile (software) modeling and methods used in USDP in connection with UML. RaBBiT's structural and behavioral models were produced by using a selected set of USDP methods. I paid special attention to usability and, therefore, to the GUI design of RaBBiT. The entire process was driven by use cases describing the interactions between RaBBiT and its users.

RaBBiT uses two metaphors in building knowledge models: graphs building knowledge models and spreadsheets for defining parametric associations. The first metaphor presents a graphical environment in which users can visually observe the result of their actions. The second metaphor reduces the complexity of defining programming rules through parametric associations to interactively defining simple equations.

## Chapter 8 Conclusions

---

### 8.1 Observations and Summary

#### 8.1.1 Overview

The present research focuses on three basic subjects: architectural programming in general; design requirements specifications for recurring building types as a specific programming task; and design and prototype implementation of a computer application supporting specifically architectural programming of recurring building types.

#### 8.1.2 Architectural programming

Note that my conclusions about architectural programming are all based on a extensive literature survey and three case studies. I also used results obtained from interviews conducted by other members of the CMU community.

Architectural programming, as part of architectural design, is an incremental and iterative process by which architectural design problems—ill-structured in essence—become better defined (although they never become well-structured in their entirety) and are specified through architectural programs. A program is typically refined to a desired level of detail through several levels of decreasing abstraction and then becomes a starting point for design exploration. As often as needed, designers and programmers go back and forth between design and programming to either modify the established requirements or to increase the level of details.

#### 8.1.3 The bottlenecks of architectural programming

Regardless of the building type to be programmed (recurring or non-recurring), the bottlenecks of the programming process do not change and are based on several factors: (a) the use of passive programming media and of manual methods, (b) non-standardized representations of program information, (c) the obstacles to continuity and upgrading of programming knowledge, (d) the lack of domain-specific tools for managing complex requirements and information-sharing.

#### **8.1.4 Programming for recurring building types**

Programmers derive, filter, and structure almost all of the required program information from scratch when they generate a program for a non-recurring building type. On the other hand, for programming recurring building types, programmers are able to take advantage of established *knowledge* and *expertise* about the organizational structures, functional requirements, and their physical implications for the planned facility. In many cases, the knowledge and expertise are documented in design guidelines addressing most common and recurring design requirements in a well-structured form. As a result, programming becomes less labor-*intensive*.

Reusable pieces of information (requirements) can be systematically organized and well-structured to form a set of programming rules and parameters for each recurring building type. Rules and parameters can be stated in the form of procedures, formulas, and variables, which can be represented computationally. These expressions can be general enough to be reused repeatedly as needed.

#### **8.1.5 Information refinement and GMEA**

Architectural programming is an information refinement process through which higher-level non-spatial information is transformed into lower-level and spatial requirements. However—and in contrast to some of the descriptions found in the literature—this process is not strictly hierarchical since different design requirements at different levels can compose a web of relationships. Strictly hierarchical models allow only one-to-many relationships and they fail to comprehensively represent webs of not strictly hierarchical dependencies and relationships.

The refinement process resembles Means-Ends Analysis, in which *means to achieve a higher-level goal become ends at the next level*. However, in (architectural) programming, ends at each level can be achieved by more than one means. Therefore, the transition from higher-level to lower-level requirements is rather a multi-directional generative process creating cyclic relations among requirements. In order to manage these cyclic relationships, the model presented in this research forms an extended (generalized) version of MEA (GMEA).

#### **8.1.6 RaBBiT**

I believe computational tools can assist architectural programmers to partially alleviate the bottlenecks of programming—particularly for recurring building types. These tools can particularly capture *functional* and *physical* performance requirements in formal representations which can be used in other generative computational design-support environments. For this purpose, I developed a prototype application to test to what extend

and which part of the architectural programming process can be supported by computational tools. I call the application *RaBBiT*. The major characteristic of RaBBiT is its capability to interact with its users during programming knowledge modeling. RaBBiT provides the following main functions: (a) computationally capturing the programming knowledge of any recurring building type, (b) simplifying the designer-computer interaction, especially in the modeling stage, to make the application usable for non-computer programmers, and (c) computationally generating architectural programs as output in a flexible format that can be adapted by different *generative* design and decision support tools.

In developing RaBBiT, I followed a systematic software development process. After studying the domain of discourse, I developed a conceptual framework formalizing (or structuring) the domain knowledge in a computable form. The framework became a reference for the definition of the system features and requirements. Based on these requirements, I selected appropriate software technologies, such as use cases for defining the system's functionality, the OO paradigm for system design and implementation, MVC for designing the system architecture, production systems for program generation, graphs for representing programming knowledge models, and XML for program data definition and program data transformation. These technologies are used for implementing RaBBiT through an agile-USDP, a custom-tailored software development process combining agile software development approaches with USDP. One important experience that I gained in using agile-USDP is that the formality of strict methodological approaches (such as in USDP) sometimes limits developers, who sometimes must ignore the formalism considering the complexity and nature of the system.

---

## **8.2 Contributions**

### **8.2.1 Contributions at the theoretical level**

- Investigation of the bottlenecks of the programming process and the methods used by the programmers.

I have not encountered a study—except (Akin, et al., 1995)—that documents the general bottlenecks of programming process in detail at the theoretical level. The findings from this study may initiate a discussion among not only the programming community, but also design researchers who wish to develop computational programming support tools to alleviate these bottlenecks. This research is intended to demonstrate the value of such endeavors.

- A better understanding of architectural programming, specifically programming recurring building types, as the design problem specification phase of architectural

design. This understanding is captured in a conceptual framework that is based on an generalized (extended) means-ends-analysis and general enough to cover all processes documented in the literature and in the case studies, while remaining operational enough to guide the design of a computer application supporting architectural programming.

Basic design considerations and program characteristics of different building types have been studied and documented in the literature. However, I did not find a general study investigating the higher-level characteristics of programming recurring building types, nor did I find a procedural model general enough to cover all methodologies presented in the literature. In fact, one of the more startling findings is that no two authors agree on the terms and concepts to be used, although the overall information processing structure is very similar across methods. I believe that the conceptual framework helps to elucidate this underlying common structure and hope that this work may start a discussion in this direction.

- The developed framework for programming knowledge modeling and program generation also reflects cognitive processes of a programmer (or designer) during architectural programming. The framework outlines a general model on how programmers make decisions while structuring a given ill-defined (design) problem towards a well-defined problem.

The generalized means-ends-analysis presented here can be considered a form and part of design problem-solving. Design researchers (and possibly cognitive psychologists who are interested in design) can use this framework for defining more general cognitive models for design problem structuring. Indeed, the validity of the presented framework can also be tested through these studies. Although this is not meant to be a direct contribution to the design cognition area, the observations made can improve our understanding of design.

- The exploration of how computer-assisted problem specification can take place in computational design problem-solving.

Even though I address a very specific question—namely how to support architectural programming through a computer application—this study falls into a more general category of design problem specification in computational form. Computational design generation and design decision support are two open-ended research areas in the design research community. This study can become a test-bed for evaluating how computer-assisted problem specification can contribute to the overall design process. The experience gained and lessons learned from this study may inspire and motivate other design researchers—including myself.



From the software development perspective, discovering computationally representable domain concepts can be trivial when these concepts are given in an appropriate form. However, casting domain concepts in this form requires intimate knowledge about the domain *and* about computer programming. Software engineers without knowledge about the domain of discourse can fail to understand the actual problems in that domain. As part of this research, I developed software for programming knowledge modeling and program generation as someone who understands the domain and its needs relatively better than someone who is not familiar with architectural programming. On the other hand, when it came to making decisions on software development issues, I needed to study the software engineering domain, which required some extra effort on my part. Combining the two areas of expertise, I developed a conceptual framework for the development of (architectural) programming software that I consider sound and clean from both the computational and architectural perspective. I believe this actually demonstrates the positive contributions that domain experts with some software engineering background can make to application development. This idea, of course, is not novel, but I believe this study is another contribution to both design and software engineering in this sense.

### **8.2.2 Contributions at the practical level**

At the practical level, the present research makes the following contributions:

- Demonstration how to create a useful and usable computational design support system that can assist architectural programmers in generating architectural programs, and partially, if not completely, can alleviate the bottlenecks of the current architectural programming process.

The proposed application was developed at a level that allows its feasibility to be tested in real-world programming situations. However, detailed field tests of the proposed application fall outside the scope of this study due to time limitations. Instead, I attempted to follow the usability considerations proposed in the human-computer interaction literature to the greatest possible degree when I developed the application.

- Design of an interaction process and a set of Graphical User Interfaces that enables the users to define interactively both rules and domain concepts of a programming knowledge model in the users' own terms, and to generate and modify architectural programs.

Knowledge modeling, in general, is an important research question in both the cognitive sciences and software engineering. Particularly, capturing procedural knowledge (rules) has been an issue in designing generative systems. I tackled this issue in the context of architectural programming and developed a flexible definition for programming knowledge models through which each knowledge category in programming recurring

building types can be defined by architectural programmers who are not computer programming experts. The system is flexible enough to allow users to customize the architectural program models according to their own preferences and to use their own terms for defining rules and objects.

- Implementation of a computer system that provides a seamless integration of an object-oriented application with human-computer interaction interfaces, aimed to perform a series of tasks delegated between the user and the system itself and to facilitate change propagation.

I initially planed to integrate two different system architectures; namely the *Architectural Program Modeler*, an object-oriented application, and the *Architectural Program Generator*, a rule-based production system. However, such a system would require a complex model-transformation and rule-translation mechanisms from object-oriented model to rule-based schema. Instead of directly implementing the program generation sub-system in a rule-based production system, I experimented with an alternative that is purely object-oriented. The sub-system uses the graph-traversal algorithm I developed for generating programs and does not require model transformation or rule translation. The experimental programs generated by this sub-system showed evidence that the rule-based system may not be needed. However, making this decision will require a more comprehensive testing of the system in a real-world context. But regardless of the outcome, I believe this was an interesting experiment also from the software engineering perspective.

---

## **8.3 Future Research Directions**

### **8.3.1 Overview**

During the course of this research, three important directions for future investigations emerged. The first relates to the usability and usefulness of the system developed. The second direction could focus on one of the most fundamental issue in computer aided design: multi-directional integration of different computational design tools. The third direction is the applicability of the developed framework and RaBBiT to other domains, such as system configuration by using components and associations.

### **8.3.2 Usability and usefulness analysis**

RaBBiT provides mechanisms to define programming knowledge models and generate programs by using these models. However, the effectiveness of RaBBiT in capturing these models needs to be tested from usability and usefulness perspectives. Usability tests are required to measure how RaBBiT meets the usability criteria. In order to make unbiased

conclusions, these tests should be performed with real users and in real programming situations. The aim would be to establish the effectiveness of RaBBiT in generating valid architectural programs, i.e. to see if the programs can be used for real-life design situations. A promising experimental design would be to see if some existing programs could be regenerated by using RaBBiT and to compare the existing and the generated programs in terms of both quality and ease of generation.

### **8.3.3 Multiple system integration**

Although programs generated as output from RaBBiT are in a shareable data format, data sharing is currently possible only in a unidirectional way: from RaBBiT to other client applications. This data sharing is modal and requires data transformation. For example, if a client application requests a model or generated program to be changed, this can be done only manually with the current implementation. In order to take all of the advantages of using multiple computational support tools for design, RaBBiT would have to provide a multi-directional "dynamic link" mechanism between itself and other client applications such that RaBBiT can respond to change-requests from client applications and the client applications can directly link to RaBBiT, for example, to trace the reasoning behind program generation. As another example, a client application used for budget optimization may link to RaBBiT to generate programs several times after modifying critical parameters without any human interruption.

### **8.3.4 Extensibility to other domains**

Dependency and parametric associations are not unique to architectural programming. For example, product modeling or system configuration use similar concepts. It would be interesting to test if the framework defined in this study can be generalized (or tailored) so that similar problems in other domains can be addressed in this framework. Theoretically, this is possible. We can use the framework to configure a system that is composed of multiple parts with different variations; the parts can be selected based on certain conditions. The concepts in the framework, components, constructs, dependencies, relations, and parametric associations are generic enough to model such a system, for example, to configure computer (hardware). As a future research direction, I like to test with experts from other domains to what extent the framework (and RaBBiT) can be used their domains.

---

## Conclusions

---

# Bibliography

- Agile Modeling 2002. [www.agilemodeling.org](http://www.agilemodeling.org)
- Agostini, E.J. 1968. Programming: Demanding Specialty in a Special World. *Architectural Record*, September.
- Akin O. and C. Akin. 1996. "Frames of Reference in Architectural Design: Analyzing the Hyper-Acclamation (Aha!). *Design Studies* 17(4): 341-361.
- Akin, Ö. 1986. *Psychology of Architectural Design*. London, Pion Limited.
- Akin, Ö. 1994. *Psychology of Early Design*; Technical Report, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.
- Akin, Ö. 1994a. *Psychology of Early Design*; Technical Report, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.
- Akin, Ö. 1994b. Creativity in design. *Performance Improvement Quarterly* 7, 3, 9-21.
- Akin, Ö. 1999. Variants of design cognition. Proceedings of the Knowing and Learning to Design Conference held at Georgia Institute of Technology, April 27-28, 1999. Ed. into a book by Chuck Eastman, Mike McCracken, Wendy Newsletter.
- Akin, Ö. and C. Akin. 1996. Frames of reference in architectural design: analysing the hyper-acclamation (A-h-a!) *Design Studies*, 17, 341-361.
- Akin, Ö., 1993. Architects' reasoning with structures and functions. *Environment and Planning B: Planning and Design* 20 (1993) 273-294.
- Akin, Ö., B. Dave, and S. Pithavadian. 1992. Heuristic generation of layouts (HeGel) based on a paradigm for problem structuring. *Environmental and Planning B: Planning and Design*. Vol. 1
- Akin, Ö., Sen, R., Donia, M. and Zhang, Y., 1995. SEED-Pro: Computer assisted architectural programming in SEED, in *Journal of Architectural Engineering*, ASCE, 1(4): 153-161.
- Alder, G. 2002. Design and Implementation of the JGraph Swing Component. White-paper in [www.jgraph.com/documentation](http://www.jgraph.com/documentation). Last visited: November, 15 2003.
- AM (Agile Modeling), 2003. The Official Agile Modeling (AM). <http://www.agilemodeling.com/> (Last visited on 11.12.2003)
- AMGA, 2001. The American Medical Group Association. <http://www.amga.org/>
- AR 140-483. 1994. Army Reserve Land and Facilities Management, Space Guidelines for U.S. Army Reserve Facilities, Army Publications and Printing Command. (also available at [http://books.usapa.belvoir.army.mil/cgi-bin/bookmgr/BOOKS/R140\\_483/CCONTENTS](http://books.usapa.belvoir.army.mil/cgi-bin/bookmgr/BOOKS/R140_483/CCONTENTS)).

- 
- Armour, F. and G. Miller. 2001. *Advanced Use Case Modeling: Software Systems*. New York, NY: Addison Wesley Pub. Co.
- Army, 2001. <http://www.army.mil/usar/overview.htm>. Last visited on August 22, 2002
- Asimov, M. 1962. *Introduction to Design*. Englewood Cliffs, NJ: Prentice-Hall
- Barker, K., B. Porter, and P. Clark. 2001. A library of generic concepts for composing knowledge bases. *Proceedings of the international conference on Knowledge capture* October 2001
- BCHS. 1974 - 1. *Equipment guidelines for ambulatory health centers*. United States. Health Services Administration. Bureau of Community Health Services. DHEW publication ; no. (PHS) 79-50066
- BCHS. 1974 - 2. *Space guidelines for ambulatory health centers*. United States. Health Services Administration. Bureau of Community Health Services. DHEW publication ; no. (PHS) 79-50066.
- Blythe, J., J. Kim, S. Ramachandran, and Y. Gil. 2001. An integrated environment for knowledge acquisition. In *International Conference on Intelligent User Interfaces*, 13–20, 2001.
- Bobrow, M. (Editor), T. Payette, R. Skaggs, R. Kobus, J. Thomas. 2000. *Building Type Basics for Healthcare Facilities*. John Wiley & Son Ltd., New York, NY.
- Booch, G., I. Jacobson, James Rumbaugh, and Jim Rumbaugh. 1998. *The Unified Modeling Language User Guide*. Addison-Wesley Publications Co. Reading, MA.
- Booch, G., J. Rumbaugh and I. Jacobson., 1999. *The Unified Modeling Language User Guide*. New York: Addison-Wesley.
- Bosch, J. 2000. *Design and Use of Software Architectures*. Addison-Wesley Publication Co., Reading, MA.
- Buchanan, Richard. 1993. *Wicked Problems in Design Thinking* in V. Morgolin and R. Buchanan (eds), *The Idea of Design*. Cambridge, Massachusetts. The MIT Press.
- Burbeck, S. 1992. *Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)*. <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>. Last visited on August 7, 2002.
- Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Son Ltd., New York, NY.
- CDC and NCHS. 1998. *National Hospital Ambulatory Healthcare Survey*. [www.cdc.gov/nchs/about/major/ahcd/outpatientcharts.htm](http://www.cdc.gov/nchs/about/major/ahcd/outpatientcharts.htm)
- Chandrasekaran, B. 1986. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3):23-30.
- Cherry, E. 1998. *Programming for design: from theory to practice*. New York, NY: John Wiley and Sons Inc.
- Chiara, J.D. and J. H. Callender, (Contributors). 1990. *Time-Saver Standards for Building Types*. New York, NY. McGraw Hill Text.
- Chimaera, 2000. *Chimaera Ontology Environment*. [www.ksl.stanford.edu/software/chimaera](http://www.ksl.stanford.edu/software/chimaera)
- Clark, P., J. Thompson, K. Barker, B. Porter, V. Chaudhri, A. Rodriguez, J. Thomere, S. Mishra, Y. Gil, P. Hayes, and T. Reichherze. 2001. Knowledge entry as the graphical assembly of components. *Proceedings of the international conference on Knowledge capture* October 2001.
- Cockburn, A. 2002a. *Agile Software Development*. New York, NY: Addison-Wesley Pub Co.

- 
- Cockburn, A. 2002b. *Writing Effective Use Cases*. New York, NY: New York, NY: Addison-Wesley Pub Co.
- COSMIC, 2002. <http://www.openchannelfoundation.org/projects/CLIPS-ADA>. Last visited on July 23, 2002.
- Coyne, R.F., U. Flemming, P. Piela, and R. Woodbury. 1993. Behavior Modeling in Design System Development. in U. Flemming and S.V. Wyk (eds.), *CAAD Futures '93 (Proceedings of the Fifth International Conference on Computer-Aided Architectural Design Futures)*. Amsterdam, Netherlands: Elsevier Science Publishers,. 1993 pp. 335-354.
- Cross, N. 1996. *Analysing Design Activity* (N. Cross, H. Christiaans and K. Dorst; eds.), John Wiley and Sons Ltd., Chichester, UK.
- Cross, N. 1997. Descriptive Models of Creative Design: application to an example. *Design Studies*, Vol. 18, No. 4.
- Cross, N. 1999. Protocol and Other Formal Studies. In *Knowing and Learning to Design*. (ed) Chuck Eastman, Mike McCracken and Wendy Newstetter. Georgia Institute of Technology.
- Cross, N. and A. Clayburn Cross. 1998. Expertise in Engineering Design. *Research in Engineering Design* 10 (3). pp. 141-149.
- Czarnecki, K. and Eisenecker, U.W. 2000. *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison Wesley.
- David, Gerald and Francoise Szigeti. 1979. Functional and Technical Programming: When the Owner/Sponsor is a Large or Complex Organization. Paper presented at the Fourth International Architectural Programming Symposium.
- Davis, Gerald. 1969. *The Independent Building Program Consultant*. Building Research
- Davis, Gerald. 1982. "The Relationship of Evaluation to Facilities Programming." In *Symposium of Evaluation of Occupied Designed Environments*. Atlanta, GA: Georgia Institute of Technology.
- DG, 1984. *Design Guide DG 1110-3-107*, U.S. Army Reserve Facilities, Department of Army, Corps of Engineers.
- DGH79 Duda, R. O., Gaschnig, J. G., and Hart, P. E., "Model Design in the Prospector Consultant System for Mineral Exploration," in *Expert Systems in the Microelectronic Age*, ed. D. Michie, Edinburgh University Press, Edinburgh, 1979.
- Dix, A., J. Finlay, G. Abowd, and R. Beale. 1998. *Human Computer Interaction* (2nd. ed). Europe: Prentice Hall.
- Domingue, J., 1998. Tadzebao and webonto: Discussing, browsing, and editing ontologies on the web, WebOnto. In *Proceedings KAW'98*.
- Donia, M. 1998. *Computational Modeling of Design Requirements for Buildings*. Doctoral Thesis. School of Architecture, Carnegie Mellon University. Pittsburgh, PA.
- Duerk, D.P. 1993. *Architectural Programming: Information Management for Design*. New York NY: John Wiley & Sons, Inc.
- Dym, C.L. 1994. *Engineering desing: a synthesis of views*. New York, N.Y. Cambridge University Press.

- 
- Eastman, C. 1970. On the Analysis of Intuitive Design Processes, in G.T. Moore (ed.), *Emerging Methods in Environment Design and Planning*. Cambridge, MA. MIT Press.
- Eastman, C. M. 1975. *Spatial Synthesis in Computer-Aided Building Design*. London, England. Applied Science Publisher Ltd.
- Elrad, T., M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. 2001b. Discussing aspects of AOP. *Communications of the ACM* Volume 44, Number 10 (2001), Pages 33-38. New York, NY: ACM Press.
- Elrad, T., R. E. Filman, A. Bader. 2001a. Aspect-oriented programming: Introduction. *Communications of the ACM*, Volume 44, Number 10 (2001), Pages 28-32. New York, NY, USA: ACM Press.
- Farbstein, J. 1977. Assumptions in Environmental Programming. In Suedfeld, P. et. al. (eds.). *The Behavioural Basis of Design*, EDRA Proceedings, Stroudsburg, PA. Dowden, Hutchinson and Ross.
- Farbstein, Jay D. 1976. Assumptions in Environmental Programming. In *the Behavioural Basis of Design: Proceedings of the Seventh International Conference of the Environmental Design Research*
- Farbstein, J. 1985. Using the Program, Applications for Design, Occupancy, and Evaluation. In Preiser, W.F.E. (ed.) 1985. *Programming the Built Environment*. New York: Van Nostrand Reinhold.
- Farquhar, A. , R. Fikes, and J. P. Rice. 1997. A Collaborative Tool for Ontology Construction. *International Journal of Human Computer Studies*, 46:707–727.
- Flemming U. and R Woodbury. 1995. Software Environment to support early phases in building design (SEED): Overview. *Journal of Architectural Engineering* 1, 4 (December).
- Flemming U., J. Adams, C. Carison, R. Coyne, S. Fenves, S. Finger, R. Ganeshan, J. Garret, A. Gupta, Y. Reich, D. Siewiorek, R. Sturges, D. Thomas, R. Woodbury. 1992. *Computational Models for Form-Function Synthesis in Engineering Design*. Technical Report EDRC 48-25-92. Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.
- Flemming, U. 1994. Artificial Intelligence and Design: A Mid-Term Review, in *Knowledge-Based Computer-Aided Architectural Design*, 1994. Carrara G. and Y.E. Kalay (eds.). Elsevier, New York, pp. 1-24.
- Flemming, U. and R. Woodbury, 1995. Software Environment to Support Early Phases in Building Design (SEED): Overview. In *Journal of Architectural Engineering*, December 1995. Volume 1. No. 4. pp. 147-152. American Society of Civil Engineers, Architectural Engineering Division.
- Flemming, U. and R.F. Baykan, 1992. Hierarchical Generate-and-Test vs. Constraint-Directed Search", in *Artificial Intelligence in Design 1992*. J.S. Gero (ed.), Kluwer Academic Publishers, Boston, pp. 817-838.
- Flemming, U. et al. 2000. The SEED Experience. Internal Report. Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA.
- Flemming, U. et. al., 1992. *Computational Models for Form-Function Synthesis in Engineering Design*. Technical Report. Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.
- Flemming, U., H.I. Erhan, I. Ozkaya. 2001. *Object-Oriented Application Development in CAD*. Technical Report 48-01-01. Pittsburgh, PA: Carnegie Mellon University, Institute of Complex Engineered Systems.



- 
- Forgy, Charles L., 1982. Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. *Artificial Intelligence* 19(1):17-37.
- Forgy, Charles L., 1998. Benchmarking OPSJ. <http://www.pst.com/benopsj.htm>. Last visited on July 29, 2002.
- Fowler, M. and K. Scott. 2002. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. New York, NY: Addison Wesley Pub. Co.
- Freeman, P. and A. Newell, 1971. A Model for Functional Reasoning in Design. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 621-633.
- French, M.J., 1992. *Conceptual Design for Engineers*. The Design Council, London.
- Gamma E., R. Helm , R. Johnson , and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.
- Gamma, E. and T. Eggenschwiler. 2002. JHotDraw as Open-source Project. <http://jhotdraw.sourceforge.net>. Last visited on August 7, 2002.
- Gero, J. S. 1998. Conceptual designing as a sequence of situated acts, in I. Smith (ed.), *Artificial Intelligence in Structural Engineering*, Springer, Berlin, pp. 165-177.
- Gero, J. S. and Maher, M. L. 1997. A framework for research in design computing, in B. Martens, H. Linzer and A. Voigt (eds), *ECAADE'97*, Österreichischer Kunst und Kulturverlag, Vienna
- Gero, J. S., 1998. Towards a model of designing which includes its situatedness, in H. Grabowski, S. Rude and G. Green (eds), *Universal Design Theory*, Shaker Verlag, Aachen, pp. 47-56
- Glegg, G. 1969. *The Design of Design*. New York, N.Y. Cambridge University Press.
- Gero, J. S. 1990. Design prototypes: a knowledge representation schema for design, *AI Magazine*, 11(4): 26-36.
- Gordon, Douglas E. and M. Stephanie Stubbs. 1988. *Technology & Practice: Programming, ARCHITECTURE*, May 1988.
- Griffin, N. L. and F. D. Lewis, 2002. A Rule-Based Inference Engine which is Optimal and VLSI Implementable, <http://cs.engr.uky.edu/~lewis/papers/inf-engine.pdf>. Last visited on July 2002.
- GSA. 1983. *Design Programming*. PBS 3430.2. Washington, D.C.: General Services Administration
- Guindon, R., H. Krasner, and B. Curtis. 1987. Cognitive Process in Software Design: Activities in Early Upstream Design, *Proceedings of the Second IFIP Conference on Human Computer Inter*
- Haley, 2001. Answers to Common Questions About AI: A Reasonign Technology. [www.haley.com](http://www.haley.com). Last visited: August 17, 2002.
- Hall, A., 1996. *Design Control: Towards a New Approach*. Oxford, GB: Butterworth-Heinemann.
- Harold, E.R. and W.S. Means. 2002. *XML in a Nutshell (Second Edition)*. Sebastopol, CA: O'Reilly and Associates Inc.
- Hershberger, R. G. 1985. A Theoretical Foundation for Architectural Programming, in *Programming the Built Environment* edited by Wolfgang F.E. Preiser, New York: Van Nostrand Reinhold.
- Hershberger, R. G. 1999. *Architectural programming and predesign manager*. NY: McGraw-Hill Co.
- Hinrichs, T.R., 1992. *Problem Solving in Open Worlds: A Case Study in Design*. Hillside, NJ. Lawrence Erlbaum Associates, Publishers.

- 
- Horowitz, H. 1967. The Program's the Thing. The American Institute of Architects Journal, May.
- Hutchins, E.L., J.D. Hollan, and D.A. Norman. 1986. Direct-manipulation interfaces. In D.A. (Eds.). User Centered System Design. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
- Hymes, C. M., 1995. Conflicting class structures between the object oriented paradigm and users concepts. Conference Companion on Human Factors in Computing Systems. Denver, Colorado, United States. Pages: 57 - 58. New York, NY, USA: ACM Press.
- Jackson, M. 1995. Software requirements and specifications: a lexicon of practice, principles and prejudices. New York, NY. Addison-Wesley Publication Co. and ACM Press.
- Jacobson, I., G. Booch, J. Rumbaugh. 1999. The Unified Software Development Process. Addison-Wesley Publication Co. Reading, MA.
- JESS, 2002. <http://herzberg.ca.sandia.gov/jess/>. Last visited on August 9, 2002.
- JGraph. 2003. [www.jgraph.com](http://www.jgraph.com). Last visited: November, 15 2003.
- Jonassen, David H. 2000. Educational technology research and development : ETR & D. 48, no. 4
- Katara, M. and S. Katz. 2003. Architectural views of aspects. Proceedings of the 2nd international conference on Aspect-oriented software development Boston, Massachusetts March 17 - 21, 2003. pp. 1-10. New York, NY: ACM Press.
- Kirk, S. and K. F. Spreckelmeyer. 1988. Creative design decisions : a systematic approach to problem solving in architecture. New York, NY: Van Nostrand Reinhold
- Kobus, R., R. L. Skaggs, M. Bobrow, J. Thomas and T.M. Payette. 1997. Building Type Basics for Healthcare Facilities. New York. NY. John Wiley and Sons Ltd.
- Kolaroff, S. 2002. Java Expression Parser (JEP). <http://jep.sourceforge.net/index.html>. Last visited on August 7, 2002.
- Krasner, G. and S. Pope. 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk -80. The Journal of Object-Oriented Programming (JOOP), August/September 1988.
- Kroes, P. 2002. Design Methodology and the Nature of Technical Artefacts. In Design Studies 23 (2002) pp. 287-302.
- Kumlin, R.R. 1995. Architectural programming: creative techniques for design professionals. NY: McGraw-Hill Co.
- Lawson, B. 1979. Cognitive Strategies in Architectural Design. Ergonomics 22 (1). pp. 59-68.
- Leffingwell, D. and Widrig, D. 2000. Managing Software Requirements: A Unified Approach. New York, NY: Addison-Wesley.
- Ling, TW, Peo, TK. 1993. Towards Resolving the Inadequacies in Object Oriented Data Models. In: Information and Software Technology, Vol: 35, Issue: 5, pp.267-276.
- Maher, L. M., M. B. Balachandran and D. M. Zhang. 1995. Case-Based Reasoning in Design. Lawrence-Erlbaum Associates, Mahwah, New Jersey.
- Malkin, J. 1982. The Design of Medical and Dental Facilities. New York, NY. John Wiley and Sons Ltd.
- Malkin, J. 1989 (1st ed.). Medical and Dental Space Planning for the 1990s. New York, NY: John Wiley and Sons Ltd.

- 
- Malkin, J. 1997 (2nd ed.) Medical and Dental Space Planning: A Comprehensive Guide to Design, Equipment, and Clinical Procedures. New York, NY: John Wiley and Sons Ltd.
- Malkin, J. 2002 (3rd ed.). Medical and Dental Space Planning: A Comprehensive Guide to Design, Equipment, and Clinical Procedures. New York, NY: John Wiley and Sons Ltd.
- Markus, T. 1972. Building Performance. New York, NY: Halstead Press.
- Mayer, R.E. 1992. Thinking, problem solving, cognition (2nd ed.). New York. Freeman
- McNeill, T., J. S. Gero, . and J. Warren. 1998. Understanding conceptual electronic design using protocol analysis, Research in Engineering Design10: 129-140
- Medical Group Managment Association (MGMA). 1999. Medical Office Space Planning Survey. [www.mgma.com/infocenter/faq-web.html#4](http://www.mgma.com/infocenter/faq-web.html#4)
- Meyer, B. 1997. Object-oriented Software Construction. 2nd Edition. Upper Saddle River, N.J. : Prentice Hall.
- Middleton, S. 2001. Training Decision Making in Organizations: Dealing with Uncertainty, Complexity, and Conflict. Special Report. (<http://www.workteams.unt.edu/reports/smiddltn.htm>).
- Newell, A. and H. A. Simon. 1972. Human Problem Solving. Englewood Cliffs: Prentice-Hall.
- Nielsen, J. 1994. Usability Engineering. Boston, MA : Academic Press.
- Noy, F.N. and D.L. McGuinness. 2001. Ontology Development 101: A Guide to Creating Your First Ontology. SMI technical report SMI-2001-0880 (2001), Stanford University.
- Oesterreich, B 1999. Developing Software with UML. Object-Oriented Analysis and Design in Practice. Reading, MA: Addison-Wesley.
- Paley, S. M., J. D. Lowrance, and P. D. Karp. 1997. A Generic Knowledge Base Browser and Editor, GKB. In Proceedings of IAAI'97. AAAI Press.
- Palmer, M. A., ed. 1981. The Architect's Guide to Facility Programming. Washington, DC. The American Institute of Architects; New York: Architectural Record Books.
- Pazzani, M. and D. Kibler. 1992. "The role of prior knowledge in inductive learning", Machine Learning 9:54-97.
- PBS-PQ100.1. 1996. Facilities Standards for the Public Buildings Service, US government General Services Administration. [www.gsa.gov/pbs/pc/tc\\_files/stds/pq100.pdf](http://www.gsa.gov/pbs/pc/tc_files/stds/pq100.pdf).
- Pena, W., W. Caudill and J. Focke. 1977. Problem Seeking: An Architectural Programming Primer. Boston, MA. Cahnners Books International, Inc.
- Pena, W.M. and J. Focke. 1969. Problem Seeking. Houston. Caudill Rowlett Scott.
- Pena, W.M., and W.W. Caudill, 1959. Architectural Analysis: Prelude to Good Design. Architectural Record, May 1959. (pg. 178-182).
- Pena, W.M., S. Parshall, and K. Kelly 1987. Problem Seeking: An Architectural Primer. Washington, DC. American Institute of Architects Press
- Perkins, B. 2000. Building Types Basics for Elementary and Secondary Schools. John Wiley & Sons, Inc. New York, NY.

- 
- Pierce, C., 1997. Group Practice Personnel Policies Manual. Medical Group Management Associations, New York, NY.
- Pierce, Courtney. 1997. Group Practice Personnel Policies Manual. Medical Group Management Associations, New York, NY.
- Preiser, W. F. E. (ed). 1978. Facility Programming: Methods and Applications. Stroudsburch, Pa.: Dowden, Hutchinson and Ross
- Preiser, W. F. E. 1993. Professional Practice in Facility Programming. New York: Van Nostrand Reinhold.
- Preiser, W.F.E. (ed) 1985. Programming the Built Environment. New York: Van Nostrand Reinhold.
- Protégé, 2000. The Protege Project. <http://protege.stanford.edu>
- Rabinson, Julia, and J. Stephen Weeks. 1984. Programming as Design. Minneapolis, Minn.: Department of Architecture, University of Minnesota.
- Reitman, W.R. 1965. Cognition and thought: An information processing approach. New York: Wiley.
- Rosenberg, D. (1999). Use Case Driven Object Modeling with UML. A Practical Approach. Reading, MA: Addison-Wesley
- Rothenfluh, T.R., Gennari, J.H., Eriksson, H., Puerta, A.R., Tu, S.W. and Musen, M.A. 1996. Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. International Journal of Human-Computer Studies 44: 303-332.
- Rowe, P.G. 1987. Design Thinking. Cambridge, MA. The MIT Press
- Rumbaugh, James, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. Object Oriented Modeling and Design. Enlewood Cliffs, NJ: Prentice-Hall.
- Russell, S. and P. Norvig. 1995. Artificial Intelligence: A Modern Approach. Upper Saddle River, New Jersey: Prentice Hall.
- Rychener, M. D., 1976. Production systems as a programming language for artificial intelligence applications. Carnegie Mellon University, School of Computer Science. Ph.D. Thesis.
- Sanoff, H., 1990. Methods of Architectural Programming. Van Nostrand Reinhold.
- Sanoff, H., 1992. Integrating Programming, Evaluation and Participation in Design: A Theory Z Approach (Ethnoscapes:, V. 7). Henry Sanoff.
- Sheiderman, B., 1982. The future of interactive systems and the emergence of direct manipulation. Behavior and Information Technology, 1(3): 237-256.
- Sheiderman, B., 1987. Designing the User Interfaces: Strategies for Effective Human-computer Interaction. New York: Addison-Wesley.
- Simon, H. A. 1981. The Sciences of the Artificial (2nd ed.). Cambridge, MA. The MIT Press
- Simon, H. A. 1996. The Sciences of the Artificial (3rd ed.). Cambridge, MA. The MIT Press
- Simon, H.A. 1989. Models of Thought, Vol. I. New Haven, CT. Yale University Press.
- Simon, H.A. 1989. Models of Thought. Volume II. New Haven, CT. Yale University Press.
- Sims, W., & Becker, F., 2000. Planning and Programming Process, in Smith, P., Facilities Engineering and Management Handbook: an Integrated Approach, McGraw-Hill Book Company .

- 
- Sims, W., 1978. Programming Environments for Human Use: A look at some approaches to generating user oriented design requirements. in Rogers and Ittelson, *New Directions in Environmental*.
- Smith, M.U. 1991. A view from biology. In M.U. Smith (ed.), *Toward a unified theory of problem solving*. Hillsdale, NJ. Lawrence Erlbaum Associates.
- Sprecklemeyer, K., 1982. *Architectural Programming as an Evaluation Tool in Design*. Environmental Design Research Association, Maryland
- Sternberg, R.J. and P.A. Frensch. (Eds.) 1991. *Complex problem solving: Principles and mechanisms*. Hillsdale, NJ. Lawrence Erlbaum Associates
- Sternberg, R.J., 1996. *Cognitive Psychology*. Harcourt Brace College Publishers, Philadelphia, PA, 1996.
- Straub, C. C. 1980. Lecture on architectural programming at Arizona State University, Tempe, Arizona
- Su, S.Y.W. and H.-H.M. Chen. 1993. Temporal rule specification and management in object-oriented knowledge bases. In N. Paton and M. Williams, editors, *Rules in Database Systems*, Workshops in Computing, pages 73-- 91. Springer, September 1993.
- Sumner, T., Bonnardel, N., & Kallak, B. H., 1997. The Cognitive Ergonomics of KnowledgeBased Design Support Systems. Paper presented at the Proceedings of Human Factors in Computing (CHI '97), Atlanta.
- UML 2002. [www.uml.org](http://www.uml.org)
- Van Melle, W., Shortliffe, E. H., and Buchanan, B. G., "EMYCIN: A Domain-Independent System that Aids in Constructing Knowledge-Based Consultation Programs," *Machine Intelligence 3* (1981).
- Veit, M. and S. Herrmann. 2003. Model-view-controller and object teams: a perfect match of paradigms. Proceedings of the 2nd international conference on Aspect-oriented software development Boston, Massachusetts March 17 - 21, 2003. pp. 140-149. New York, NY: ACM Press.
- Verger, M., N. Kaderland. 1993. *Connective Planning*. New York, NY. McGraw-Hill
- W3C, 2003. <http://www.w3c.org>. Last visited on November 5, 2003
- WGPPF. 1992. *Postsecondary Education Facilities Inventory and Classification Manual*. Working Group on Postsecondary Physical Facilities. [bacweb.the-bac.edu/~michael.b.williams/](http://bacweb.the-bac.edu/~michael.b.williams/)
- Wheeler, C. G. 1966. *Emerging Techniques of Architectural Practice*. Washington, DC. The American Institute of Architects.
- White, E.T. 1972. *Introduction to Architectural Programming*. Tucson, AZ. Architectural Media.
- White, Edward T. 1985. *Project Programming: A Growing Architectural Service*, Architectural Programming and Design Option, Graduate Program, School of Architecture, Florida A&M University,
- WK81 Weiss, S. M. and Kulikowski, C. A., "EXPERT Consultation Systems: The Expert and Casnet Projects," *Machine Intelligence 3* (1981).
- Zeisel J. and P. Welch. 1982. *Administrative Handbook for Design Programming vol. 2*. Cambridge, MA. Building Diagnostics
- Zeisel, J. and P. Welch. 1982. *Technical Handbook for Design Programming, vol.1*. Cambridge, MA. Building Diagnostics
- Zimring, C., D.L Craig, 2001. Defining Design Between Domains: An Argument for Design Research a la Carte (penultimate draft), in *Design Knowing and Learning: Cognition in Design Education*.



---

## Appendix A: Case Studies

---

1. United States Army Reserve Centers
2. Elementary and Secondary Public Schools
3. Ambulatory Health Care Facilities

---

# *Case Studies on Programming Recurring Building Types*

**Halil I. Erhan  
School of Architecture and ICES  
Carnegie Mellon University**

**May, 2001**

---



---

## Table of Content

<b>1. Introduction</b>	<b>1</b>
Motivation	1
Objectives	2
 <b>2. Case Study: The United States Army Reserve Centers</b>	 <b>3</b>
Introduction	3
Basic USARC activity types and spatial designations	4
<i>Training activities and spatial designations</i>	4
<i>Maintenance activities</i>	11
Effects of army unit structure on a USARC architectural program.	13
Effects of army equipment and vehicles on the program	15
Area requirements calculations for USARCs	16
A graphical framework delinating spatial requirements generation for USARC.	18
USARC activity affinities and their effect on design requirements	19
<i>Low-level affinities: USARC spatial relationships.</i>	19
<i>Higher-level affinities: USARC activity relationships.</i>	20
<i>Representation of the affinity structures.</i>	20
Summary and discussions.	22
 <b>3. Case Study: Elementary and Secondary Public Schools</b>	 <b>25</b>
Introduction	25
ESPS activities	26
<i>Overview</i>	26
<i>Educational activities and spaces.</i>	26
<i>Administrative activities and spatial requirements</i>	28
<i>Support activities and spaces.</i>	29
Effects of school type and school capacity on spatial requirements.	31
Spatial requirements generation for ESPS.	34
ESPS activity affinities and spatial requirements.	35
<i>Overview</i>	35
<i>ESPS activity affinities.</i>	36
<i>ESPS spatial relationships.</i>	38
Summary and discussions.	39

---

#### 4. *Case Study: Ambulatory Health Care Facilities* 41

Introduction 41

Effect of the medicine speciality on the program 41

Activities in ambulatory health care facilities 43

*Common activities* 43

*Semi-specialized activities and descriptions:* 47

*Specialized activities* 49

Effect of a staffing pattern and patients on a program 56

    Effects of furniture, and medical instruments, equipment, tools on the program. 59

Spatial area requirements calculation methods: 61

Synthesis of components in a framework 63

*Framework components and constructs.* 63

*The number of physicians: The main independent variable for AHCF* 65

*Variables attached to specialty components.* 65

*Sample specialty with attached constructs* 67

*The sample space component with attached constructs* 68

Activity affinities and their effects on spatial relationships 69

Summary 72

#### *Bibliography* 73

#### *Appendix A: USARCs Space Assignments*

#### *Appendix B: ESPS Space Assignments*

#### *Appendix C: AHCFs Space Assignments*

---

## **List of Figures**

- FIGURE 2.1.USARC major activity structure 4
- FIGURE 2.2.Unit structure-rated capacity-space relation. 14
- FIGURE 2.3.Unit structure-rank structure-spatial requirements relation. 14
- FIGURE 2.4.Unit mission-activity composition-space requirements relation. 15
- FIGURE 2.5.Combination of non-spatial factors effecting spatial requirements in a USARC. 15
- FIGURE 2.6.Army equipment and vehicles determine spatial requirements 16
- FIGURE 2.7. Analyzing a computer programming instruction setting for two unit members. 17
- FIGURE 2.8. OMS spatial area decision structure in the framework. 18
- FIGURE 2.9.An army unit member's day sequence diagram based on the same use-case. 21
- FIGURE 3.1.ESPS activity structure. 26
- FIGURE 3.2.ESPS educational activities and spaces. 28
- FIGURE 3.3.ESPS spatial designations of administrative activities. 29
- FIGURE 3.4. ESPS spatial designations of support activities. 31
- FIGURE 3.5.School capacity and type affect activity composition and change spatial requirements. 33
- FIGURE 3.6. A partial view of the graphical framework for ESPS programming 34
- FIGURE 3.7.A diagram selected from (OSDM, 2001, pg. 4103-1) as an example of how spatial relationships (affinities) are represented in general. 36
- FIGURE 3.8. Sequence diagram describing the activities, spaces and participants during the lunch break. 38
- FIGURE 4.1.Categories of ambulatory health care services. 42
- FIGURE 4.2.Activity-staff-space interaction 57
- FIGURE 4.3.Staff-requires-staff relation changes staffing pattern and spatial requirements. 57
- FIGURE 4.4.Patient-volume is determined by staffing, and staffing pattern is affected by patient volume; the interaction affects the spatial requirements. 58
- FIGURE 4.5.An activity requires equipment and it is used by staff; this changes spatial requirements 59
- FIGURE 4.6.Complete schema for program elements. 60
- FIGURE 4.7.Analysis of spatial area requirements of a study desk setting 61
- FIGURE 4.8.Components and constructs 65
- FIGURE 4.9.Pediatrics specialty and attached constructs. 67
- FIGURE 4.10.Updating the schema by considering proximity and accessibilities between spaces. 69
- FIGURE 4.11.Activity affinity analysis by UML interaction diagram 71

---

### List of Tables

- TABLE 2.1. Women's latrine spatial-area calculation table (DG, 1984, pg. 46) 11
- TABLE 2.2. Rated capacity for total authorized strength (AR 140-483, pg. 1). 14
- TABLE 3.1. Area requirements are selected considering the school capacity ranges. 32
- TABLE 4.1. Activities and medical specialty (X represents required activities, C represents the activity is required if certain conditions are satisfied). 55
- TABLE 4.2. MGMA (1999) conducted an informal survey of group practices and their space planning. The averages for square footage and number of exam/patient treatment rooms are listed in the table 62

---

# 1. *Introduction*

---

## 1.1 **Motivation**

We believe that recurring building types can offer opportunities for more efficient and effective specification of design requirements. Recurring building types are repeated in different contexts (including geographic location or social environments), yet their general functional aspects do not change; their program components and the relationship between these components are usually well-understood. Typical functions, user characteristics, and general organizational issues form a common ground for each project. Most probably, many precedent architectural programs already exist and can be adapted for new projects.

However, problems with use of manual methods and passive programming media, difficulties with handling complex information, and non-standard representation techniques used in current practice lead to inefficiencies and ineffectiveness in programming process. A computer-assisted architectural programming can be one of the means to partial overcome these difficulties. However, we believe that the system which will assist programming must be based on findings of a careful investigation of programming-related characteristics of recurring building types.

One way of discovering these findings is through an investigation of relatively well-established architectural programming processes for recurring building types. As far as the building types are concerned, we selected ambulatory health care facilities (AHCF), United States Army Reserve Centers (USARC) and elementary and secondary public schools (ESPS). The primary reasons for selecting these types are as follows:

- The different functionality of each building type provides a wide range of contextual differences. This can help comparing different design requirements generation methods and finding their commonalities.
- In these building types, the institutional structures are well established and organizational issues are clear. Effects of these factors on programming and program can be observed relatively easier than for non-recurring building types.
- Occupants (and users) of a recurring building type carry basic characteristic similarities. Activities that occupants and users perform are independent from the location (context) that the building is designed for.
- Each of the selected building types is relatively complex; therefore findings from studying these building types may be easily applied to less complex building types. (Fast-food stores that belong to a chain, for example, are also a recurring building type, but the complexity level is not as high as for the selected types.)
- There is a huge demand for new such facilities, at least AHCFs and ESPS.
- Studies performed by private or public parties document standard requirements and design guidelines for each type. These are easily accessible. However, most of these studies do not address methodologies for programming, but present certain criteria that a design can be evaluated against.

---

## 1.2 Objectives

Specifically, the objectives of the case studies can be stated as follows:

1. To investigate architectural programming for selected recurring types, namely USARC, AHCF and PESS, in a structured way.
2. To provide data for a detailed inquiry of commonalities in architectural programming for recurring building types. Some of these commonalities will deal with generative techniques of programming that we hope will help us answer the following questions:
  - Is there a programming pattern that can be generalized for different types of recurring building?
  - What are the common leading concepts among programming different recurring building types?
3. To determine if this process can be supported by state-of-the-art computational tools so that the process becomes more seamless, and if so:
  - to investigate if it is possible to interactively manage the generation of programmatic information.
  - to systematically examine how such a computational tool can help to improve architectural programming, both in conventional ways and in transferring generated data to other generative computational tools (e.g. layout generator, budget or scheduling applications etc.).
  - Investigate the usage of such tools not only as part of support for early design phases, but also for the detailed design.

Results of these studies are expected to improve our understanding of generative mechanisms of design requirements specification. These mechanisms can be employed for two purposes: first, they can help us improve the architectural programming process. Second, we can support programming by implementing enhanced methods in computational tools. Such a computational tool can be used for:

- Providing seamless links between different types of information through enabling each piece of information of one type to communicate with related pieces of information of other types.
- Enabling consistent and synchronized updates of the design requirements specification of a particular program
- Generating design requirements in different formats
- Taking requirements available to other computational design support tools, such as generating schematic layout, analyzing budget, staffing and scheduling
- Providing improved knowledge consistently about a specific building type.

## 2. *Case Study: The United States Army Reserve Centers*

---

### 2.1 Introduction

Training is the backbone of the army activities. The new and updated military defence systems, methods and technologies create an increasing need for continuous training of the army personnel. Also, as new recruits join to the army or promotions take place, the importance of training becomes more clear. Therefore, training becomes an ongoing and well-structured process to keep the army ready to accomplish given assignments with the highest success rate and minimum damage.

As part of the army structure, the army reserve units (ARU) are not less focused in terms of training. Because of the same reasons listed above, the ARUs have to be trained to the same degree as any full-time army personnel. The *reserves* are the members of an Army Reserve Unit defined in (Army 1, 2001) as follows:

"The U.S. Army Reserve is the active Army's federal reserve force (more than 1.100.000 reserve organized in 2000 units). It is made up of highly trained and ready-to-go combat support and combat service support forces that can move on short notice to give the active Army the resources it needs to deploy overseas and to sustain combat troops during war-time, contingencies or other operations. It is the Army's main source of transportation, medical, logistical and other units, and it is the Army's only source of trained individual soldiers to augment headquarters staffs and fill vacancies in units."

The training of the ARUs takes place in the United States Army Reserve Centers (USARC), which are strategically located in different geographic areas across the United States of America. Along side their primary purpose, which is to assure *training* ARUs, these facilities are planned to operate as a regular army facility (AR 140-483). The activities accommodated in a USARC are described in the following sections.

The reserves in a certain geographic area are assigned to a USARC facility located in the same geographic area. Each group of reserves is scheduled to use the facility at a certain drill weekend. Depending on the mission of the units, one or more of the units can use the facility simultaneously. Besides the reserves, the facility accommodates a full-time staff who performs administrative and support activities.

As a need for a new USARC facility emerges due to the different strategic decisions made by the armed forces (such as the shifting a unit from one location to an other, upgrading or extending the training activities, or increasing the number of reserves at a certain location etc.), these facilities are designed and built by spending a significant amount of resources and employing many architecture, engineering and construction specialists. Each time a new USARC is to be built, the design requirements of the facility have to be determined for the particular missions assigned to the units. A well-configured architectural program

can help to eliminate the misuse of the resources and assures that USARC activities are performed effectively and efficiently in an appropriate architectural environment. The resources spent and the challenges of configuring architectural programs for these facilities make the USARCs an interesting recurring building type.

In this study, we are interested in the architectural programming of these facilities. In order to understand the dynamics of programming and managing design requirements of USARCs, we need to investigate typical parameters and methods used during the programming process. The following sections will describe these issues in detail.

---

## 2.2 Basic USARC activity types and spatial designations

As stated before, in a typical USARC, two major activities take place: training and training-related maintenance and support activities. Training activities consist of four main groups: administration, instruction, assembly, storage-support and special training (Figure 2.1). In addition to these, if required, special army equipment-use training (such as weapons, tank turret or simulation) can take place in these facilities. Maintenance group includes organizational maintenance activities, which are training oriented; and area maintenance support activities, which relate to maintenance of army equipment and vehicles by full-time mechanics. Direct and general support activities consist of ancillary functions which take place in dressing rooms, tool rooms, shop offices etc. and these activities support both organizational and area maintenance activities (DG, 1984).

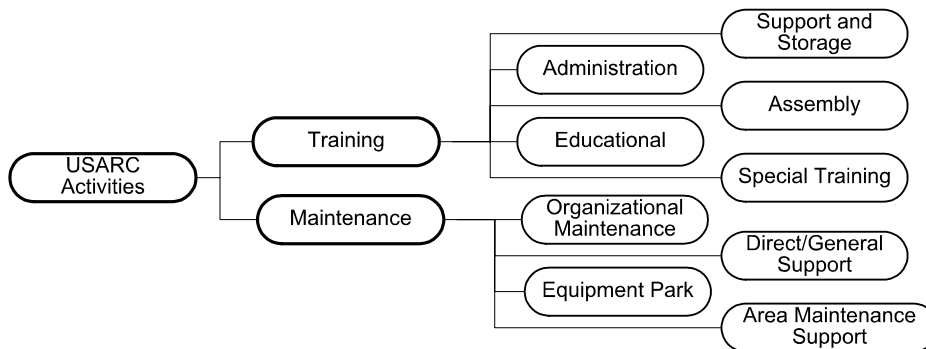


FIGURE 2.1.USARC major activity structure

The activities along with sub-activities are explained in the following section. Assignments of activities to spatial units (e.g. room, zone, gathering area etc.) are also described. A complete list of space definitions and required area calculations is represented in Appendix A.

### 2.2.1 Training activities and spatial designations

**Administration:** Administrative activities are divided into two specific categories: daily administration by permanent personnel and periodic administration by AR personnel. The spaces in which these activities take place have two types of use: *exclusive* by officers and *common* by enlisted personnel. The exclusive-use spaces are reserved for ranking officers and personnel with special tasks (recruit-



ing, consulting etc.). The common-use and related activities take place in an open-office space setting where workstations for office tasks are configured. The common-office space is preferred to be surrounded by exclusive-use offices.

Space area allowances and designations of administrative activities depend on two major factors. The first of these factors is the rank of the personnel who will occupy the space, and the second one is the nature of the activity. The activity descriptions are highly coupled with the ranking structure in USARCs. For example, each full-time personnel who participates in administrative activities is authorized to have 120 sqf. office area (open or closed space). During the drill period, depending on the number of reservees (troops) to be trained, the rank structure of the ARU changes. The officers with different ranks are authorized to have office spaces of different size depending on their ranks. Therefore the rank structure is reflected in the design requirements. If an ARU requires a major general, he or she is allowed to have an office as large as 400 sqf, a brigadier general is authorized to have 300 sqf. of space; for a colonel, 200 sqf of office space area are authorized. The allowable office area for an officer increases parallel to the the rank of the officer in the hierarchy.

The rank structure also reflects the number of troops in a particular army unit. For example, in a company unit the highest ranked officer is a captain. The unit contains minimum two platoons, which each is led by a lieutenant. In a platoon there are minimum two squads consisting of 12 to 20 troops. Therefore, in a company led by a captain there are more then 50 soldiers. The maximum numbers is specified by the army authorities according to the mission of the unit. The same structure also applies to the ARUs.

In ARUs, the maximum number of reserves is defined as *total authorized drilling strength (number of reserves) of the largest drill (training) weekend*. The largest drill strength is headed by the highest ranked officer in the ARU. Usually, the highest ranking officer during the largest drill weekend can represent how large the training unit is and how reservees are grouped (namely platoon, company, battalion, regiment etc.). It also has to be noted that the missions of the units who are being trained in the same drill weekend can be different. Therefore the ranking structure may not necessarily imply the type of the training taking place during a certain drill weekend but the assigned missions do. During specifying the design requirements of an USARC, the rank structure and its affects on the number of troops becomes a frequently used variable.

The common-use office spaces are also defined in terms of the maximum number of personnel who will use these spaces during a drill period. For these spaces, each personnel who will be trained as office personnel is authorized to have 60 sqf. open-office area. This number is standard for each USARC and calculated after the analysis of the activities, which considers required furniture and equipment. For example, in unit-common office each personnel is given a study desk, two file cabinets, and one visitor chair. Space allocation incorporates the required area for each of these furniture per unit.

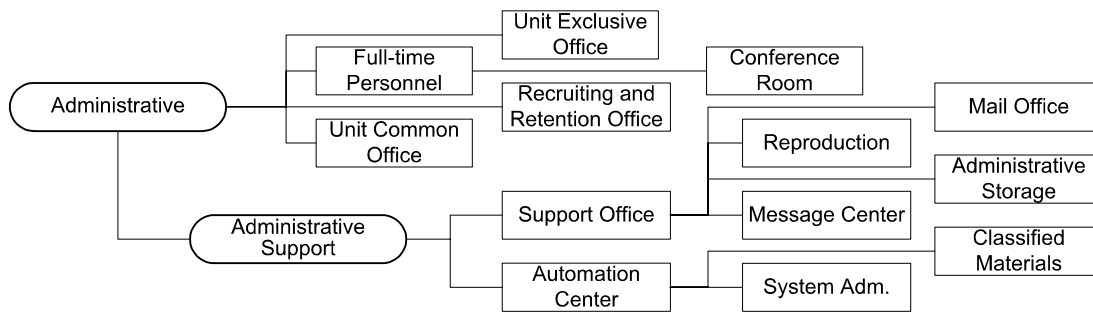
After adding the circulation paths to the open office areas, an initial area requirement for these office spaces can be determined. The area which is required for intra-activity circulation must be equal to 15 percent of the total area required for office spaces. Inter-activity circulation paths (e.g. from open-office space to exclusive space) are specified as individual spaces.

Some of the administrative activities, such as recruiting and retention, require a fixed spatial area which is independent from the rank of the personnel who occupy the space. For example, a recruiting office in a typical USARC is authorized to be 250 sqf. In this office officers, civilian and enlisted personnel work together for recruiting activities.

**Administrative support:** The administrative activities include receiving and distributing all inter- and intra-office correspondences, reproducing of army documents and paperwork, and automation-support (computer-based). The area requirements for these spaces are based on the maximum number of reserves being trained during a typical drill period. The required area decision is made in a range which consists of a minimum and maximum value. For example, the area requirements for support-offices are calculated as follows:

Area = 120 sqf + (Round\_up (number of members/50) x 60 sqf).  
if the required area is greater than 360 sqf then the area is 360 sqf.

The relationships of activities in office-spaces can be represented in the following figure.



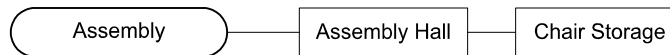
**Assembly:** In a typical ARU, one of the main activities is to assemble the unit-members in an organized and ordered manner in an assembly hall. Usually, the officers address the units about a particular issue involving the whole unit. The assembly hall must be flexible enough to accommodate other functions such as dining, full-unit training, special events, and ceremonies. The hall is used by large groups for different activities which can't take place in other spaces. The space must have direct access to the spaces where administration, food service, and instruction activities take place. The area requirement is calculated by incorporating the total authorized drill strength of the largest drill weekend into a formula. As part of the assembly hall, a chair-storage space is required. The range of

---

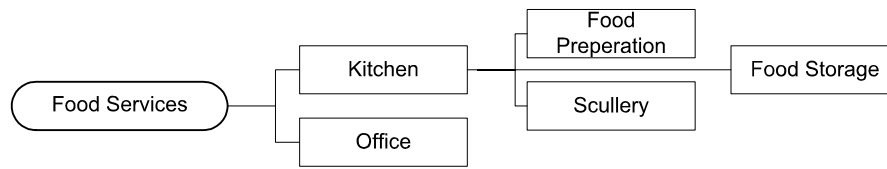
## Case Study: The United States Army Reserve Centers

area authorization is between 3000 and 6200 sqf. The area of assembly hall is calculated as follows:

Area =  $3000 + (\text{Round\_up}(\text{Number of members} / 50) \times 600)$   
if (Area > 6200) then Area = 6200 sqf.



**Food service:** The activity consists of preparing, cooking and serving the food held in the food storage. It also includes cleaning and storing all utensils, pots, dishes, trays and silverware. The area requirement for the spaces in which food service activities take place is standard for all USARCs: 300 sqf. for food preparation, 150 sqf. for food storage, and 200 sqf. for scullery activities. In addition to these, a 80 sqf. of office-space area is needed.



**Instruction:** These activities involve instructional training of unit members. The activities can be grouped under two different settings: group instruction and individual study. In group instruction, a certain number of personnel are educated by an instructor. This activity usually takes place in the classrooms, which are planned to accommodate 25-30 trainees and an instructor. The classrooms are required to be flexible so that they can be divided by a movable partition. In the individual study activities, each individual studies alone in a space such as library, reading-room, or learning-center.

The area requirements for the classrooms can be calculated by two different methods. The first one uses the total number of personnel (troops or unit members) of the largest drill weekend. In this method, the following formula is applied:

Area =  $3000 \text{ sqf.} + (\text{Round\_up}(\text{Number of members} / 50) \times 300 \text{ sqf.})$

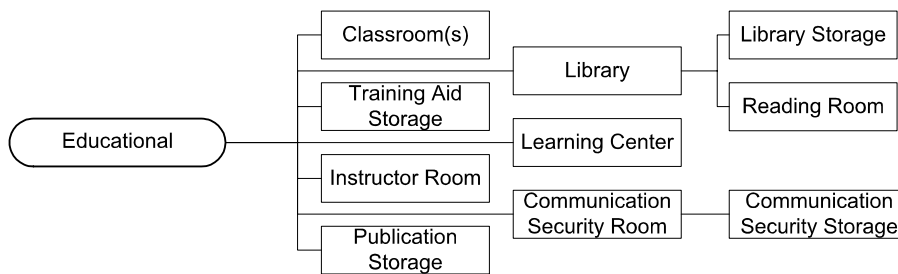
The second method is similar to the first one, but the formulation is different. In this method, each individual who will be trained is authorized a 20 sqf. of classroom area. The number of trainees is multiplied by this value. However, the first method is preferred in the recent design guidelines (AR 140-483).

The number of classrooms can be calculated by dividing the number of members of the largest drill weekend by the ideal classroom size, which is 25-30 reserves. The classrooms must have direct connection to a training aid storage. The storage

area is 10 percent of the total classroom areas. Another requirement is that direct daylight (e.g. window open to outside) should be provided to the classrooms.

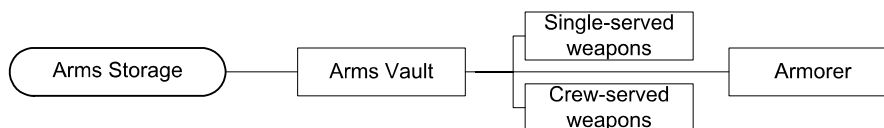
If a USARC is authorized a COMSEC account (COMMunication SECurity Training), a communication security training classroom and storage area are additional required spaces. Each of these is authorized to have a 100 sqf. area.

For each USARC, an instructor room of 300 sqf. and a publication room are additional required spaces.



**Weapon storage:** If a USARC is assigned weapon training, a special weapon storage area is required. The storage is divided into two distinct zones: single-served weapon storage (such as pistols or rifles) and crew-served weapon storage (such as machine guns). The storage must be secure enough to prevent unauthorized entry and exit. A place where the weapons are repaired and maintained is also required. The area calculation for the required spaces can be calculated as follows:

Single-served weapon storage =  $220 + (\text{Round\_up}(\text{number of members} / 100) \times 110)$  sqf.  
Crew-served weapon storage =  $\text{Round\_up}(\text{number of crew\_served weapons} / 50) \times 110$  sqf.  
Armorer = 110 sqf.  
Total Area = Single-served weapon storage + Crew-served weapon storage + Armorer



**General storage:** Unit and individual equipment items are stored in specially designated areas where standard 96 square foot storage cages are located. The number of cages depends on two factors: the number of members to be trained and the unit type. For example, if the unit type is "None school TDA" (i.e. if there is not any training activity which take place in classrooms), one cage per increment or portion of 20 unit members is authorized. Fifteen percent of the storage area is added as intra-functional circulation area. In addition to these areas, a staging area, which must be equal to the unit storage area, is required. The stor-

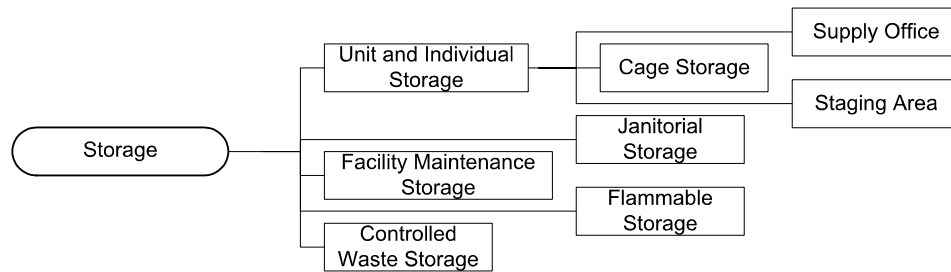
age is managed by a supply officer, who is located in a 120 sqf. of office adjacent to the storage.

Fifty square feet of janitorial storage area are required for each USARC. If the facility is not planned to include organizational and area maintenance activities, additional storage such as flammable storage and controlled waste storage is also required. Facility maintenance and storage for custodial contractors are other required spaces and their area calculation is as follows:

Area for facility maintenance and custodial contractors =  $200 + (\text{Round\_up}(\text{number of members} / 10) \times 100 \text{ sqf.})$

if (Area > 800) then Area = 800 sqf.

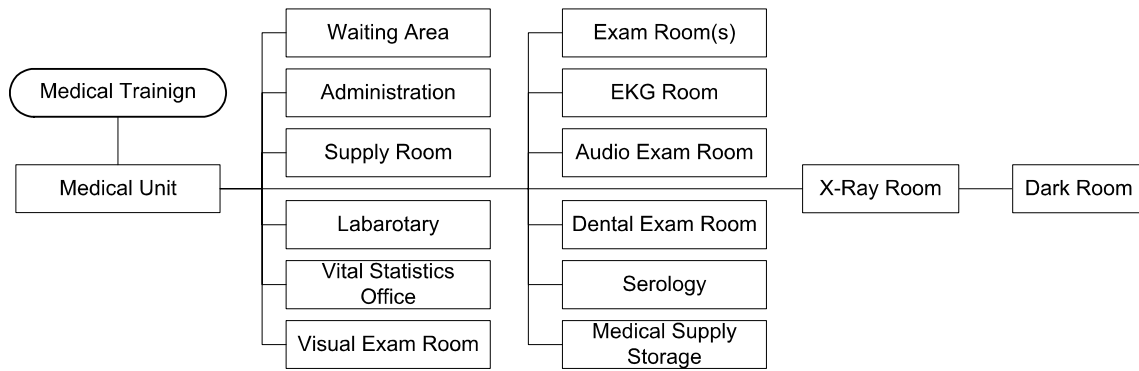
Similar formulas are used for calculating area requirements for storage. A complete set of the formulas and functions are included in Appendix A.



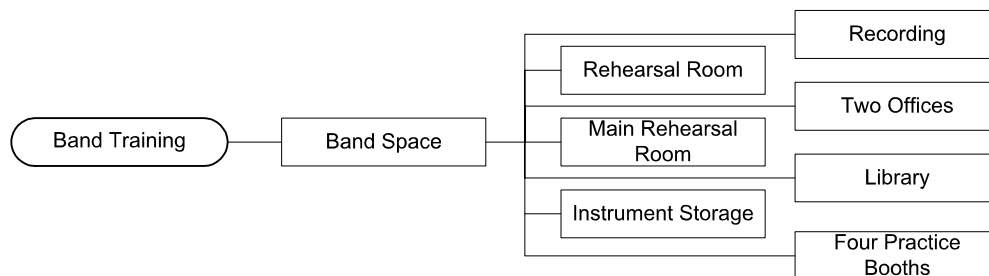
**Special training:** These activities prepare the unit members for special missions. Some of these activities are the followings: rifle shooting practice, photo processing and analyzing, medical training, physical exercise, sensitive information processing, solid material testing, drafting etc. The area requirement calculations either depend on the number of members who participate in these training activities or a fixed area is authorized for each activity.

Among these activities, medical training is the most complex one and it requires specialized spaces. However, this complexity is solved by a standard medical facility program, which the overall space requirements of medical training facilities are defined. This standard program is applicable for each USARC. For example, in order to accommodate medical training activities, a 400 sqf. area is needed. An additional 2500 sqf. area is required for physical examination activities. The training and physical examination activities are further broken down into smaller activities, and each of these activities is assigned to a specific space. Their relationships (and even spatial layouts) are defined in the medical zone program.

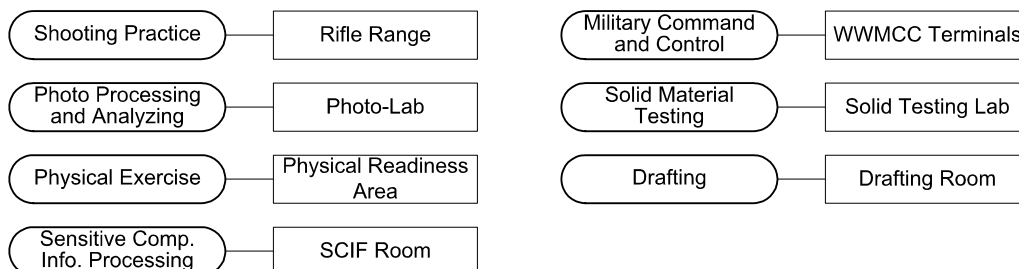
## Case Study: The United States Army Reserve Centers



Another activity which requires complex spatial features is band-training. The spaces required for this activity include offices, practice booths, storage for instruments, rehearsal room, recording room, library, and main rehearsal space. Some of the design requirements (including the area requirements) for each of these spaces are mentioned in (DG, 1984, pg. 58) and are applicable to all USARCs.



Other special activities and their respective activity-to-space mapping are represented in the following figure.

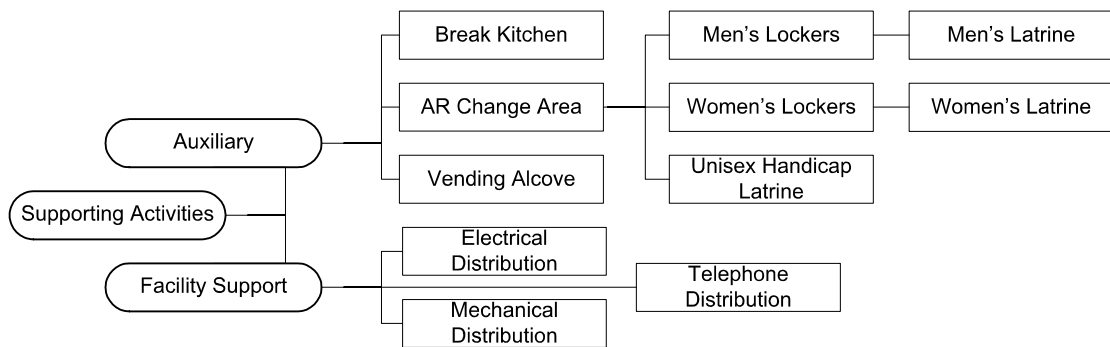


**General support:** These activities are divided into two types: auxiliary and facility support. The first group of activities requires men's and women's locker rooms and latrine facilities. An alcove to locate vending machines for refreshments and a kitchenette for full-time personnel are additional required spaces. The second type of support activity deals with electrical-telephone-mechanical distribution, which requires continuous maintenance.

The required area calculations for the first type of activities are based on two different methods. The first method consists of a formula in which the number of members of the unit is incorporated. The second method uses standard data-tables to select spatial requirements. A sample table for planning women's latrine is represented in Table 2.1. The table takes the peak occupancy as the guiding variable, which assumes that 30 percent of the largest drill weekend would be female. The spaces supporting the second type of activities have constant (fixed) area requirements.

Peak Occupancy	Water Closets	Lavatories	Showers	Total Fixtures	Spatial area (sqf)
1 to 15	1	1	1	3	150
16 to 35	2	2	1	5	175
36 to 55	3	3	1	7	225
56 to 60	4	3	1	8	250
61 to 80	4	4	1	9	275
....	...	...	...	...	...

TABLE 2.1. Women's latrine spatial-area calculation table (DG, 1984, pg. 46)

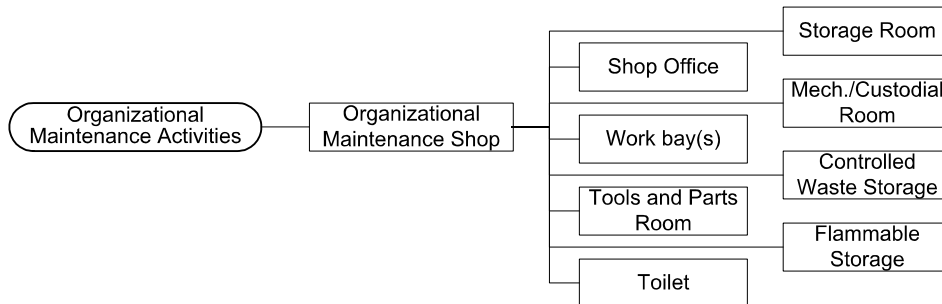


### 2.2.2 Maintenance activities

**Organizational-maintenance:** The purpose of organizational maintenance is to train ARU assigned to mechanic positions. The maintenance training takes place in the organizational maintenance shop(s) (OMS), which is required if more than

10 vehicles are assigned to a USARC. The number of vehicles are calculated by adding the number of all assigned wheeled and tracked vehicles, engineering equipment. The OMSs have their unique spatial requirements. All of the maintenance activities must be connected to work bay area, which provides a space for servicing and repairing the assigned mobile and stationary army equipment. For every increment of four assigned vehicles, one work bay is needed. The dimensions of each work bay is standard (20 feet wide and 40 feet deep with a 14 feet clear ceiling height). Four feet-wide walkways along each side of the work bays are also required for circulation. The configuration of work bays depends on the number of work bays needed, which should allow each equipment to enter and exit without blocking the other vehicles.

Each OMS should provide outside work bays in front of the vehicle entrances, which are the extension of work-bay area. The area requirements for outside work bays are similar to the area requirements for indoor work bays, except they don't need an enclosure.



The area of work bays is calculated as follows:

Number\_of\_workbays = Round\_up (number of vehicles / 4)

Area of work bays = [number of work bays x (40 feet x 20 feet)] + [(4 feet x 40 feet) + (4 feet x 20 feet)]

Other activities are performed in a space surrounding the work bay cluster. Needed spaces are an office, a tools and parts room, a small latrine, a storage, a battery room, a flammable storage room, a controlled waste storage, and a mechanic/custodial room. The area requirements of these spaces depend on the number of work bays required in a OMS. For example, in order to calculate the area requirement for the battery room, the following formula is applied:

Area of battery room = 50 + [(number of work bays - 1) x 25 sqf]

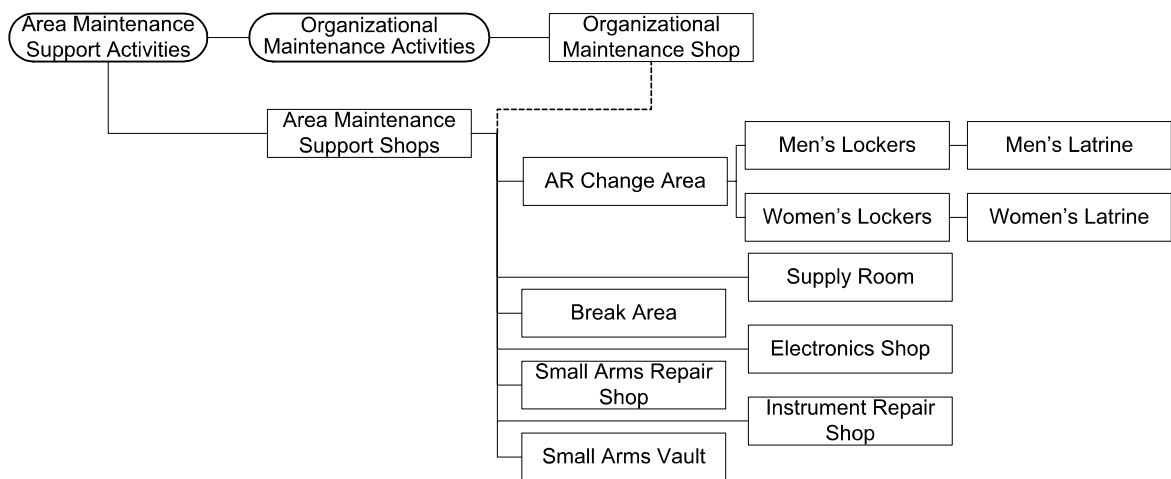
if (Area > 200 sqf) then Area = 200 sqf.

The area of shop-offices is calculated based on the number of assigned full-time administrative officers and the number of administrative personnel. The formula which is used to calculate the area requirements of office space is defined as follows:

Shop office area = [number of adm. personnel x 60 sqf] + [number of adm. officers x 120 sqf]



**Area maintenance support:** As opposed to OMS, the primary purpose of these activities is to service vehicles by the full-time personnel. Even though training ARU mechanics is not an objective of these type of activities, they require settings similar to OMSs. The *area maintenance support activity shops* (AMSA) are the space in which the majority of the maintenance work is performed. If required, these shops can be located in a separate location. If located in a USARC facility, AMSA shares OMS's work bays and support spaces. If this is the case, some other unique spaces which are not listed in OMS are needed to accommodate full-time personnel. These spaces are a break area, men's and women's locker rooms, a supply room, an electronics shop, a small arms repair shop connected to an arms vault, and an instrument repair shop. Outside of each AMSA, a service or access apron as wide as work bay clusters is required. In order to clean vehicles, a wash platform is also required. A standard service apron and wash platform is also needed.



The area calculations of work bays and other spaces in AMSA facilities are similar to the spatial area calculations for OMS facilities.

An equipment park is also required for parking army equipment and vehicles in a safe place. Depending on the climate and the needed security level, the equipment park can be either open or closed. A complete description of the spatial area calculations is described in Appendix A.

### 2.3 Effects of army unit structure on a USARC architectural program.

The army is a highly organized institution. The unit structure hierarchically grows as the mission of the unit gets more complex. The smallest part of an army unit is a soldier. A certain number of soldiers (usually between 12 to 20) form a squad; two or more squads form a platoon; two or more platoons form a company; two or more company form a battalion etc. As the structure grows by addition of more units into regiment, brigade, division, corps and army.

The structure of the army units is reflected in the physical facilities that the army uses (Figure 2.2). For example, a training building capacity is based on a variable called *rated capacity*; it is based on "aggregate authorized strength (number of troops) of all units programmed for assignment to the center" (AR 140-483, pg. 1). This variable corresponds to "the maximum number of reserves that a facility can accommodate at all training assemblies." The rated capacity is selected from a standard table Table 2.2. The rated capacity is used in formulas to calculate area requirements for different spaces. In previous section, the *number of members* variable points to the value assigned to the rated capacity.

Total authorized strength	Rated capacity
Under 55	N/A
55 to 75	60
76 to 125	100
126 to 175	150
176 to 250	200
...	...

TABLE 2.2. Rated capacity for total authorized strength (AR 140-483, pg. 1).

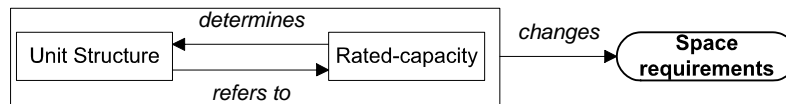


FIGURE 2.2. Unit structure-rated capacity-space relation.

The hierarchical rank structure is also reflected in the physical requirements of a USARC. As described earlier, the best example of this is that the spatial area of office spaces increases as the rank of the officers assigned to these offices increases. For example, a general is authorized to have 400 sqf. office space, and a lower rank, colonel, is authorized to have 200 sqf. office space. Parallel to the increase in the rank, the number of members of a unit increases as well (or vice versa). The increase in the number of members, as described in activity-space designations, causes changes in the spatial area designations and, consequently, affects activity-space assignments (Figure 2.3).

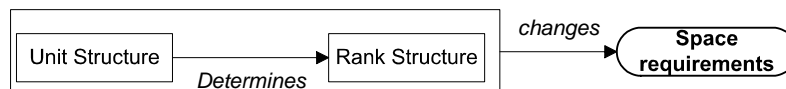


FIGURE 2.3. Unit structure-rank structure-spatial requirements relation.

The mission requirement of the units assigned to a USARC is another determinant that affects the spatial requirements. The mission, furthermore, is broken down into specific activities, which in turn constitute an activity composition.

The mission assigned to an ARU determines the activities which take place in a USARC. Each activity to be performed requires certain spaces (Figure 2.4). For example, if an ARU trains to maintain army equipment, then a OMS facility becomes part of the spatial requirements. Another example is when a unit is assigned for weapon-use training. In this case, an army vault, a shooting range with a certain number of shooting lanes, an arm repair space etc. are added to the program. Beside this interaction, the spatial-area and dimensional requirements are calculated considering other variables, such as rated capacity and space allowances per member.

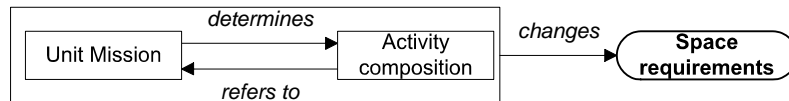


FIGURE 2.4. Unit mission-activity composition-space requirements relation.

We observe that an ARU structure is the result of the mission assigned to the unit. As the unit structure changes, the ranking structure of the unit members changes as well. The combination of all of these relations is represented in Figure 2.5.

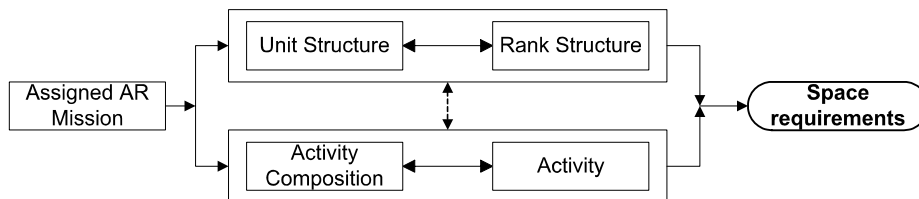


FIGURE 2.5. Combination of non-spatial factors effecting spatial requirements in a USARC.

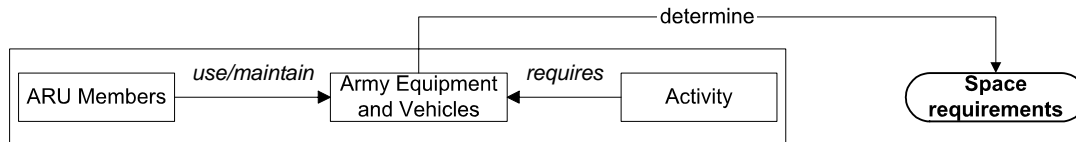
## 2.4 Effects of army equipment and vehicles on the program

The equipment and vehicles that are used during an ARU's mission directly affect the decisions made about the spatial requirements. We observed that each equipment or vehicle can be in one of three basic states: stored in a location, being maintained/repared by unit members, or being used for training purposes. In each case, it occupies a space with certain requirements. When equipment is not used, it is located in a parking area with varying sizes depending on the equipment and vehicle types. Another example are army weapons that are stored in arms vaults where the vaults area is determined considering the sizes of the weapons, security concerns, accessibility to arm-repair shops or shooting practice ranges.

The maintenance spaces must accommodate equipment and vehicles of different sizes, therefore flexibility of these spaces is a desired feature. An example for this is the work bays of an OMS. The vehicle that is maintained in these spaces can be of different types, such as an armored vehicle or a multi-terrain vehicle (like a Hambee). The work bay's spatial requirements are optimized to accommodate all possible types of vehicles. Wireless communication equipment is maintained or repaired as part of USARC activities. The spaces where the equipment and vehicles are used are specified similar to spaces they are maintained in (i.e. equipment and vehicle size- and operation-related features are incorporated into the spatial requirements).

The furniture which is used in both open- and private-office spaces is also a determining factor of spatial requirements. In an open office, each individual (such as a clerk) is assigned to a workstation with a study desk, a guest chair, an office chair, and two file cabinets. A standard furniture layout in a standard-size space is specified in army design guidelines (DG, 1984, pg. 24).

Each equipment that the ARU members uses is associated with one or more activities. Therefore, the effects of army equipment on the space requirements can be represented as shown in Figure 2.6



**FIGURE 2.6.**Army equipment and vehicles determine spatial requirements

## **2.5 Area requirements calculations for USARCs**

The area requirements of the activities which take place in a USARC are determined considering basically four types of information:

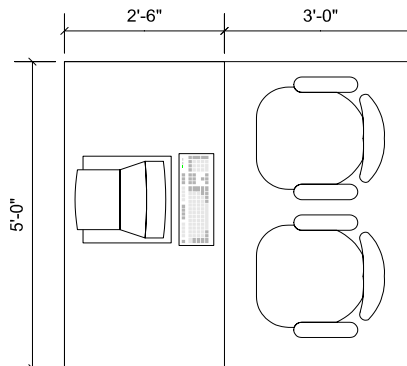
- number of members who perform the activity,
- rank of members who perform the activity,
- activity-specific description (including ergonomic and antropometric data),
- physical data about the equipment, vehicles, and furniture to be used, maintained or stored.

As noticed, non-spatial information such as ergonomic data, or number or rank of unit members, are considered in calculating area requirements of spaces in a USARC. Some of the area calculation formulations are explained in Section 2.2. When these formulations are generalized, we observe that each formula utilizes similar variables which refer to the above listed four basic types of information. For example, in calculating area requirements for an office-space for a ranked officer, we determine the area by considering the rank of the officer, the office layout considering furniture area requirements, and the use of the office itself in relation to other activities. However, for each office space and for each

officer rank, a predefined spatial area with its dimensions is listed in associated army documents such as (DG, 1984) and (AR 140-483, 1994). Selecting the value from these sources eliminates the area calculation phase in most of the cases.

The area requirements of an OMS, on the other hand, are calculated by taking *the number of work bays* as one of the reference variables. The number of work bays depends on the number of pieces of equipment and vehicles which are assigned to the unit. The variables in formulas to calculate area requirements can be extracted from the army standard documents as done in this study (such as (DG, 1986)(AR 140-143, 1994)). The area requirements either can be picked from the listed sources or the listed sources include standard formulas in which values for each variable are assigned and the formulas result in the area requirement. Therefore, the methods for area calculation present an organized systematic pattern.

Spaces required by activities which aren't standard for a USARC may not be included in design guidelines. In such a case, the army provides the description of the activity, which contains the typical participants of the activity, special equipment to be used during the activity and affinities to other activities are contained. Design requirements for the spaces which accommodate non-standard activities can be decided with an analytical method. For example, let's assume that a non-standard activity would be training the unit members in computer programming. The number of members who will be trained in a session could be a reference variable to start with. For each increment of two members, one computer may be assigned (which is usually a decision made by the army planners). Therefore, the number of computers will be equal to half of the number of unit members who will be trained. An area to accommodate two unit members, a computer and a work station (desk) can be calculated by analyzing the activity-space requirements (Figure 2.7). The number of workstations and the unit area requirements are multiplied. In order to finalize the area requirements, the result is added to the sum of the required circulation area (which is standard 15% of the total activity area for USARC) and ancillary spaces (such as storage, printer, and instructor desk etc.). During the analysis, additional equipment and furniture which could be used during the activity can also be determined.



**FIGURE 2.7.** Analyzing a computer programming instruction setting for two unit members.

As described above, the methods to be used for area requirements can be one of three types: selecting a value from a table, using a pre-determined formula, or analytical reasoning. Depending on the situation, the programmer can choose to use one of these methods.

## 2.6 A graphical framework delineating spatial requirements generation for USARC.

In the previous sections, components which play an important role in the transition from non-spatial requirement to spatial requirements are discussed. Activities and their composition, army unit structure and rank structure, and activity-dependent equipment-vehicle-furniture are some of these components. In order to understand all of these components in their entirety, we delineate a framework in which these components and their relationships can be structured.

The framework that we intend to delineate is represented in Appendix A. The framework is described by two types of representations complementing each other. The first representation contains formulas, functions, variables and conditional statements when used calculating the number and spatial requirements of spaces which accommodate USARC activities. The second representation graphically describes the hierarchical structure of the spaces. Both representation can be joined to describe the framework as shown in Figure 2.8 as an example. However, we preferred to simplify this in Appendix A by dividing it into the representations that we mentioned here.

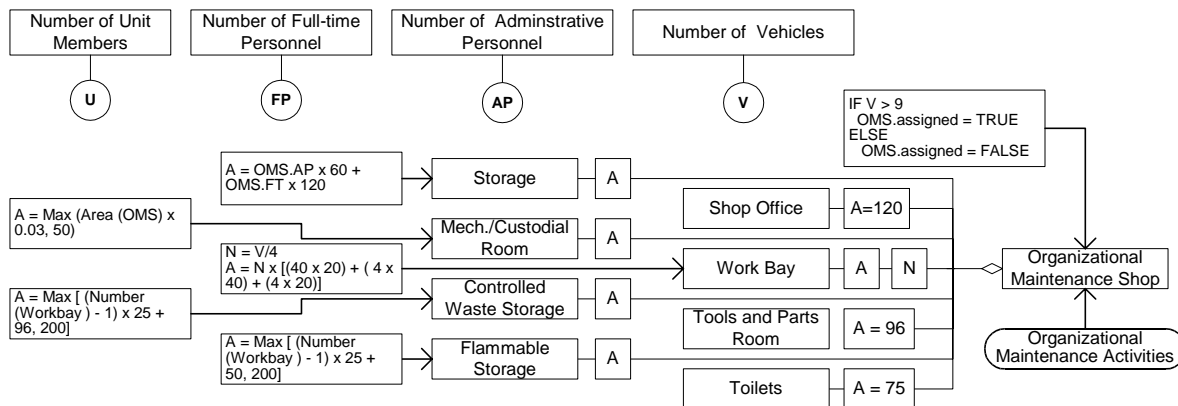


FIGURE 2.8. OMS spatial area decision structure in the framework.

In this framework, independent variables (such as the number of unit members, number of army equipment-vehicles etc.) are used along with dependent variables (such as the number of work bays) in formulas and functions to make decisions about spatial requirements. For example in Figure 2.8, an OMS and associated spaces are shown. If the number of army equipment and vehicles assigned to a unit is more than 10, a USARC is assigned to an OMS. The number of vehicles and equipment is decided by army authorities. After this decision, spatial requirements for an OMS can be derived. Each requirement involves one or more spaces, and each space's spatial requirements are decided by applying associated

formulas and functions. For example, to decide the number of required work bays, we divide the number of assigned vehicles and equipment by four, i.e. for each work bay, four vehicles are assigned. The formula is not directly given in the standard documentations, but can be derived from the descriptions. Similarly, area requirements for each work bay are also provided in these standards. These requirements can be formulated in a function where the area variables (such as the number and area) of work bays are calculated. The demonstrated process can be extended to include generating other design requirements (such as equipment assignment, mechanical, electrical, or structural requirements).

## 2.7 USARC activity affinities and their effect on design requirements

In previous sections, the USARC activities and their effects on spatial requirements have been investigated. At the next level, we intend to investigate the USARC activity affinity patterns and their reflection in the spatial requirements and affinities. This is because, we believe that the spatial affinities are a way of describing how activity affinities in a given organizational structure can be constructed and satisfied by a facility.

### 2.7.1 Low-level affinities: USARC spatial relationships.

The affinities between spaces are the outcome of the activity affinities. That is, if two or more activities relate to each other, the spaces which accommodate these activities relate to each other in some way. We observe that this can happen in three basic situations.

The first one is that a certain space may need to be located at a certain physical distance to other related spaces. This can be called *proximity*. For example, In order to avoid the noise, the rifle range should be located away from the medical wing. It also has to be located close to the arms vault and arms repair shop to provide an isolated and safe access to arms. The magnitude of the closeness (distance) can be specified depending on the size of the facility and other spatial requirements.

The second situation occurs when different types of *accessibility*, such as visual, physical, acoustic etc., from one space to other spaces are required. For instance, from the exclusive offices to the common-use office area a direct access is required. Also, between the work-stations located in the common-use office area, a certain level of visual accessibility is needed to observe the trainees. The noise access from the assembly hall to the classrooms is not a desired feature. Therefore, a zero level acoustic accessibility from assembly hall to the classrooms can be specified.

In the third situation, a space may be used for multiple purposes (activities) at different times. Therefore the spatial requirements including spatial affinities have to be specified considering each of the activities. For example, the assembly hall is used for assembly, large group training, ceremonies, and dining. Each of these activities may require the assembly hall furniture and layout to be arranged differently. If the dining activity is scheduled, the assembly hall is furnished with dining tables, and the space has to have access to the food service area. If a ceremony takes place, the arrangement will change and may be the tables and chairs will put in a storage area. Each of these activities consist of their unique affinities to other activities and they are projected to space affinities. The affinities between different activities taking place in the same space at different times imply a spatial *overlapping* relation from one space to the very same space.

### 2.7.2 Higher-level affinities: USARC activity relationships.

We believe that the spatial affinities are derived from higher-level affinities such as relationship from one activity to other activities in an organizational structure. In the army design guidelines (DG, 1984), affinities between activities are not explicitly mentioned, but affinities between spaces are. From the spatial affinities and their descriptions, we could derive the activities and their affinities to each other. Therefore, the spatial affinities can be used to analyze the logic and patterns underlying behind spatial affinity requirements in army design guidelines. This process is similar to reverse engineering and the results could provide us a more adaptable method in generalizing the USARC spatial affinity structure.

In addition to spatial affinities, the design guidelines refer to the activity participants (unit members, officers, mechanics etc.) who contribute to an activity and the activity's relation to other activities. For instance, the specification of an ranked-officer's office refers to the officer's rank and role in the facility. If the officer supervises clerical training, then this implies an affinity between what the officer does and what the clerical personnel does. This relation can be interpreted as the officer's office must have a direct access and visual access to the clerical personnel training area.

The equipment and vehicles are also incorporated into activity affinity structure, and therefore into spatial affinities. There are cases where a spatial affinity is required only because an equipment or a vehicle is needed to be shared by two or more activities. For example, the vehicle and equipment maintenance activities in work bays of an OMS share the same space with AMSA activities. The main spaces (such as work bays) have an overlapping affinity to itself and the support spaces.

### 2.7.3 Representation of the affinity structures.

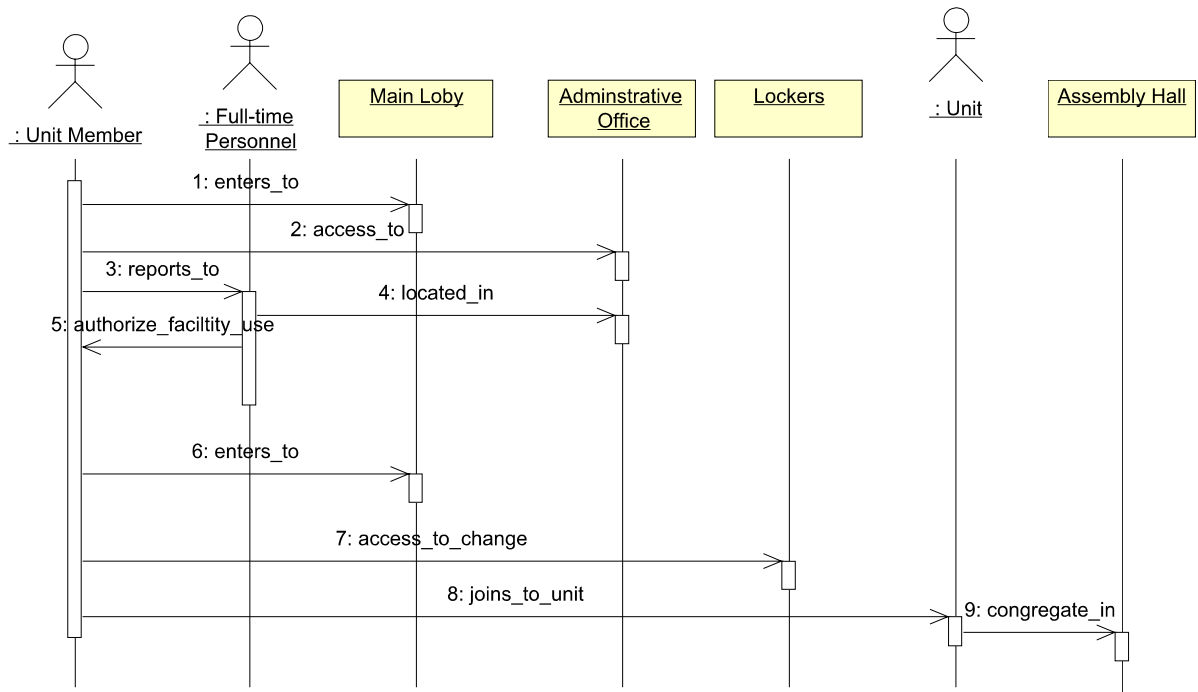
In order to represent affinities between spaces, affinity matrix, affinity diagrams or affinity charts are used in conventional practice. Army design guideline (DG, 1984) uses spatial adjacency diagrams to illustrate the spatial affinities. However, these techniques only represent the final state of spatial relationships, and make it difficult, if not impossible, to understand the logic behind them. Understanding the complete affinity structure requires more comprehensive methods where the activities are described, and their implications on activity participants, equipment and vehicles, and spaces are explored. Also, as another dimension, synchronization (time) of activities could be represented. These methods also can help us manipulate the affinity structure without changing the essence of the requirements. Such an analysis can be done using UML methods and notations. Each space, performer (as role players), equipment and vehicles can be analyzed with their interactions and interrelations. In order to illustrate how this could be achieved, a unit member's day is described in the following use-case.

1. The unit member enters to the main lobby
2. The full-time personnel is located in the administrative office
3. The unit member enters to the administrative office
4. The unit member reports to the full-time personnel
5. The full-time personnel authorizes the facility-use for the unit member



6. The unit member enters to the lockers through the main lobby and gets ready for training activities
7. The unit member joins to the unit
8. The unit congregates in the assembly hall.

The same use-case can also be represented by using a sequence diagram. The beginning of an army unit member's day at the simplest form is represented by this method in Figure 2.9.



**FIGURE 2.9.**An army unit member's day sequence diagram based on the same use-case.

Figure shows the first activities that a unit member performs as the day starts. The diagram incorporates both the role-players and spaces in a time-based manner. In addition, it explicitly depicts the activity sequences and involved factors (space, user, vehicle etc.). Equipment and vehicles can also be added to the sequence diagram. This analysis technique can help us to establish a mechanism by which we evaluate an activity sequence with the participants, equipment and vehicles, and reason about possible spaces and space descriptions. For our sample case, direct access from administrative office to lockers is not a required feature. The access to these spaces is provided through the main lobby, which can also be the main entrance to the facility. The unit member can access the assembly hall directly from the locker area. Where we don't have a direct access, we may choose to specify circulation spaces which are not mentioned before. However, this sequence may change if the constraints on activity requirements change. For example, an electronic sign-

in and identification equipment may eliminate reporting to the full-time personnel. Instead, a computer recognizes who requests entry to the facility's restricted zones and grand permission or not.

The army design guidelines include a set of spatial adjacency diagrams. However, as we noted earlier, they are not as comprehensive as needed to understand the involvement of role players, equipment, vehicles, and spatial areas in activities.

---

## 2.8 Summary and discussions.

Specifying design requirements for USARCs can be considered as a decision making process, which starts with the decision of building one of these facilities. After the strategic goals of the facility are established early in the process, decisions about non-spatial requirements based on these goals are defined. These requirements determine the scope of the project. Among the decisions are the type of the facility, activities and their composition, the ARU structure, and budget etc. In the next step, the non-spatial requirements lead to a set of spatial requirements in the form of a space list, definitions and affinities.

In decisions about the spatial or non-spatial needs of an USARC, the activities and their composition play an especially important role. If one of the goals requires an activity to be accommodated in a USARC, spatial (quantitative and qualitative) needs are derived from the description of the activities. In deed design guidelines are established to assume that each activity requirement can be satisfied in the form of a space and its description (AR 143-144)(DG, 1984). Therefore, activity selection and composition constitute one of the very first programming phases.

Adding a space as a requirement to the program results from the decision of adding an activity to the activity composition. A gradual shift from required activities to required spaces occurs during the programming process. This also incorporates equipment and vehicles and activity participants. At each decision step, the resolution of the specified requirements increases to provide a more concrete description of what should be satisfied by the design. In this process, an activity's relation to other activities (i.e. activity affinities) is reflected in the spatial program in the form of spatial affinities.

Design requirements that can be stated with some type of quantitative measures are formulated by using variables, formulas, and procedures. For example, the army unit structure is considered in design requirements through variables like the number of units, number of full-time personnel, number of mechanics, the number of officers with different ranks etc. Equipment and vehicle needs are also reflected in the design requirements as variables such as number of vehicles, their physical and operational features (encapsulated in variables such as area required to store or operate an equipment or a vehicle), and the number of crew members who use the equipment. These and other variables are used in formulas and procedures, which are employed in making decisions about the needs of a USARC facility.

The design guidelines provide information about the variables and their use in formulas, procedures, and logical statements. We observed two basic limitations with the information provided. The first one is that the objective of the formulations are confined to determine limited types of needs, such as the space sizes.<sup>1</sup>

The second limitation is the presentation and the use of the information, which is based on some paper form. Obviously, the guidelines pages have to be manually browsed and the relevant formulation has to be manually selected. The overwhelming task of information compiling and update can become so complex that managing the information manually results in a cumbersome situation.

In order to assist the decision making process by a computer application, the army engineers designed a spread-sheet<sup>2</sup> based on the information provided in design guidelines. However, its use is limited to calculating certain values such as total area, gross area, circulation area etc., and it is not capable of generating design requirements which are interconnected with each other. Consistency and correctness of the entered information is not checked. Furthermore, conflicts between different formulations are not handled. A change in one formula is not propagated through the spread-sheet. These are shortcomings partially due to the challenges with developing spread-sheet applications, and partially due to the limited implementation of the formulas.

We believe that the process of generating design requirements for USARCs can be modelled and simulated in a specific computer application. This especially seems possible given to the highly structured nature of the process. This process contains stages of deciding the activities of a USARC, activity related spaces, and spatial requirements of these spaces. We also believe that implementing the process in a computational environment can provide opportunities for making the process more efficient and effective, and for reproducing the final program in different formats which can be used by other computer applications or for other documentation purposes. In this way, the re-use of the computer-generated programs also becomes possible.

- 
1. The application of these variables to the decision making process is included in Appendix A.
  2. The spread sheet file is provided by the U.S. Army Construction Engineering Research (CERL) Center. The spread sheet calculates the gross area, net area and circulation area in USARCs.



---

# 3.

## *Case Study: Elementary and Secondary Public Schools*

---

### 3.1 Introduction

A statistical study conducted by the US Department of Education (DoE, 2000) and National Center for Education Statistics (NCES) showed that the elementary and secondary education institutions constitute about 93% of the national education institutions. 73% of these schools are supported by state and federal funds, i.e. these are public schools. As of 2000, some 42.5 million of 52 million students from kindergarten through 12th grade attend the elementary and secondary public schools (ESPS).

In order to provide school facilities to accommodate increasing student numbers, the amount spent for school construction has increased 98% in the last six years (1994-2000). The public funding spent for ESPS in the year 2000 alone exceeded 26.6 billion dollars nationwide. These figures are projected to continue increasing parallel to the growth and geographic shifts in the school-age-population. However, the public schools's capacity and quality have not reached a sufficient level to cover existing demand (Hebert and Meek, 1990, pg. 11). Changes in the education systems and programs, and the completion of old school's life cycles are other factors that emphasize the need for new schools.

Currently, the major road-blocks in expanding the capacity and increasing the number of ESPS facilities are the following: the lack of funding, the cost of accommodating new requirements, which emerges from the adaptation of modern educational programs and technology, and bureaucracy (Ortiz, 1994, pg.10; Hebert and Meek, 1990, pg. 11-16). We believe that in addition to these, the conventional project delivery process forms another road-block. The project delivery process passes through the stages of planning, design, construction, occupancy and post-occupancy evaluation. However, as one of the major components of the systems, architectural programming and design requirement specification, in particular, are not emphasized enough. At this stage, most of the cost-effective planning and design approaches can be developed. In the current process, the quality (aesthetic, symbolic, environmental issues) of the school facilities are more emphasized than how the school facility can be functional and economic, and yet satisfy the quality concerns as well (Ortiz, 1994, pg.16,63-74; Cherry, 1999, pg. 132-138). Stating that, we don't mean that qualitative concerns are less important. On the contrary, in order to achieve them, a rational and realistic approach should be adapted, particularly, for determining the physical (spatial) needs. Most importantly, the time factor in planning such facilities should be seriously considered. This includes the time spent on architectural programming.

In the following sections, we investigate the programmatic issues of this recurring building type. We believe that the investigation will help us formulate a pattern in programming ESPS facilities, determine the unique design requirements of ESPS facilities (such as the main variables which can be used in determining the physical descriptions of required spaces), and uncover the similarities and differences of programming these facilities with the programming other recurring building types.

### 3.2 ESPS activities

#### 3.2.1 Overview

The formal school activities are organized in three groups; educational, administrative and support activities (Figure 3.1). The educational activities consist of mastery program activities and creative (group) program activities (Legget et.al. 1977). The mastery program activities deal with reading, mathematics, and language skills. Students learn through oral and written instructions or they study educational material in a formal setting. The creative activities place emphasis on discovery, creativity and problem solving skills. Social studies, science, art and shop, music, and physical education form this group of activities. Both mastery and creative programs are interwoven to form a complete educational program. The administrative activities, on the other hand, assure that the school's operation is planned and managed without any problem or delay. They span from record keeping to curriculum and schedule assignments. The support activities provide basic services such as food service, library services, student health service, sanitation etc.

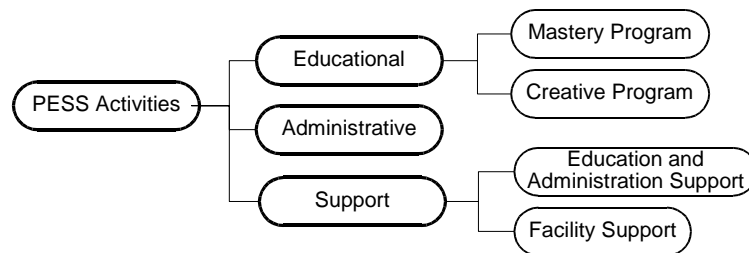


FIGURE 3.1.ESPS activity structure.

The described activity structure can expand to include other activities (such as community education, parent meetings, extra-curricular sportive activities etc.) or it may be altered in different education programming approaches. But, for the purpose of this study, we can use a simplified structure as represented in Figure 3.1. In any case, these basic activity groups exist in each ESPS, but can be defined or accommodated differently.

#### 3.2.2 Educational activities and spaces.

As mentioned before, educational activities are grouped under two categories. Activities in the first category, which are part of the mastery program, take place in two types of areas: general-purpose classrooms and specific education spaces. The general-purpose classrooms are typically sized to accommodate 28 students and a teacher (maybe with an assistant); but this number can change for different school districts or projects. Most of the modern education approaches suggest keeping the number of students in a classroom between 22-24 (Perkins, 2000, pg. 28).

Elementary and secondary school classroom requirements change depending on the changing needs of and education programs for the students in different age groups. For example, for an elementary school classroom, a project area within the classroom which can be used for science, computer, and other equipment-intensive activities is required. On

the other hand, for secondary schools, specialized program areas (such as science rooms and computer lab) replace project areas. In other words, these activities become more complex and take place in separate locations. A locker area for secondary schools is needed while it can be replaced with a wall-hanger unit in an elementary school classroom. In addition, secondary school students are provided with individual tables and chairs, whereas elementary school students can sit either at individual tables and chairs or at group tables. In either case, the required space to accommodate classroom furniture changes due to the change in the number of different age groups.

In order to accommodate students with special needs (such as students with learning or physical disabilities, gifted-children etc.) who cannot attend school in regular classrooms, the special education classrooms are provided. These classrooms have the same or half the size of general classrooms in both elementary and secondary schools. For example, in state of New York, Virginia, Ohio or Florida this number ranges from 6 to 12 (OSDM, 2001, pg. 4100; Perkins, 2001, pg. 35).

The number of classrooms needed in a ESPS equals to the projected school population divided by the maximum number of students in a classroom stated in the education policies. In general, the required classroom area is calculated by the number of students in a class multiplied by the allowable unit area per student. However, usually, each state provides guidelines and codes in calculating classroom areas. For instance, in New York, the minimum standard for an elementary school classroom is 770 sqf based on 27 students. In California, minimum space required in a classroom for each student must be not less than 30 sqf per student, and a classroom can not be smaller than 960 sqf. The Florida State Education Board recommends 25 students per classroom for grades 1 through 3, at a range of 36 to 40 sqf per student, and 28 students per classroom for grades 4 through 6, at a range of 30-34 sqf per student (Perkins, 2001, pg. 30). In the Ohio School Design Manual (OSDM) of the Ohio Department of Education, the maximum class size is suggested to be 25 students and the minimum area for each classroom is specified to be 900 sqf. (OSDM, 2001, pg. 6101-3).

The creative group activities take place in specialized program areas. Their spatial requirements may differ from one school to another depending on the budget and school population. The spaces that accommodate these activities are music room, science lab, art room, computer lab, gymnasium, (if preferred with a stage), media/video center, and library. The library and gymnasium are typically *required* spaces by code, the others are *recommended* spaces to be included in the program if the budget and student population allows. In order to make use of the budget more efficiently, multiple activities can be accommodated in a single space; such as a gymnasium, which can also serve as auditorium. If the school is a high school, the program includes an agricultural shop, business classroom, homemaking room, industrial art room, technical drafting room (or CAD room), vocational shops (e.g. woodworking, auto repair etc.), and band room. As other activities are required by the education program, depending on the budget, other specialized spaces, such as student lounge or parent education classrooms, could be included.

The area requirements and number of these spaces differ in elementary schools and secondary schools. For example, in an elementary school with 400 or less students, one science room with 1,000-1,400 sqf. area (or 40 sqf area per student in a class) is sufficient. A 1,000-1,200 sqf. science room (or 50 sqf area per student in a class) for every increment of

400 students is needed for a middle school. In high schools, science rooms become specialized laboratories such as biology, chemistry, or physics labs, which could be around 1,200 sqf. each (or 50 sqf per student). The number of labs is calculated based on the student population. For each increment of 400 students, one additional physics lab and for each increment of 600 students, one additional chemistry and one biology lab are needed (OSDM, 2001, pg. 2300-2; Perkins, 2001, pg. 34).

The following figure shows the spaces needed for mastery and creative activity groups which are mentioned above.

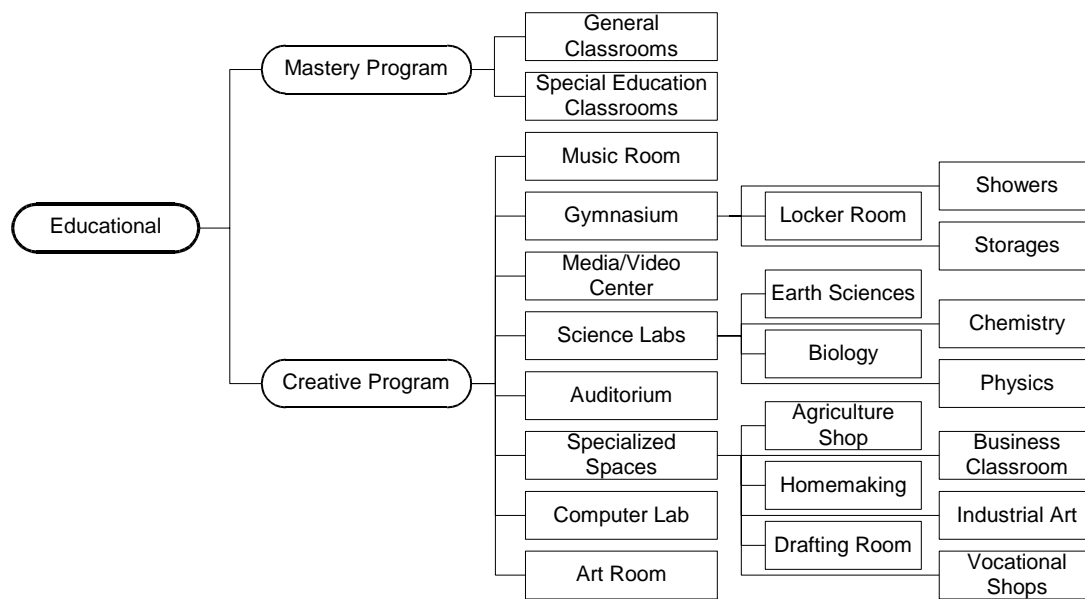


FIGURE 3.2.ESPS educational activities and spaces.

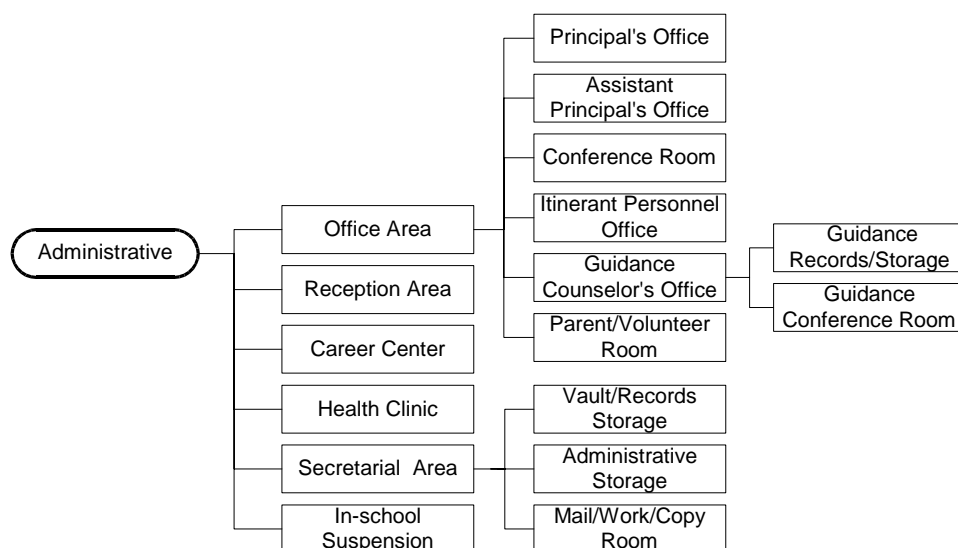
### 3.2.3 Administrative activities and spatial requirements

Administrative activities include, but are not limited to, record keeping and reviewing, budget development, curriculum development and class scheduling, counseling, public (parents) and school relations, student registration and graduation, faculty (teacher) meetings. Essentially these activities take place in an office area which become the control center for the school and contact point for parents, students, and faculty alike (De Chiarra and Callender, 2001, pg. 225). Generally, the administration area is placed near the school's main entrance to enable direct visual supervision of visitors. Security has become an increasing concern for schools and consequently for the architectural program of school facilities. A reception area controlled from the administration area, security check point and, if needed, security equipment area become important for a safe school environment.

There are two factors which determine the complexity of the functions in a school administration: the student population and the type of the school. First, if the student population



increases, the school administration functions become more complex. Second, the responsibilities of a high school administration are different than a middle school; and a middle school administration functions are different than an elementary school administration. Therefore, these complexities are reflected on the spatial requirements of administrative spaces.



**FIGURE 3.3.**ESPS spatial designations of administrative activities.

The administrative office is described in the Ohio School Design Manual (OSDM, 2001) as represented in Figure 3.3. The figure shows the suggested spaces that may be required in an ESPS facility. Deciding which space will be added to the program depends on the budget, school type, and school population. If one of these spaces is added to the program, it should conform a minimum required area or a unit area per student requirements stated in the school design manuals. Perkin (Perkins, 2001, pg. 34, 35) provides a table which shows the typical spaces in the administrative area, spatial requirements, and whether or not they are typically required.

### 3.2.4 Support activities and spaces.

Support activities in an ESPS aim at providing services for education, administrative, and facility maintenance activities, yet not directly relate to scholastic instruction of the students. The supporting activities tend to be accommodated in a centralized location in the school building due to the need of students and teachers to access to these spaces easily. Some of these activities are library services, food services and dining, personal and health services (such as showers and restroom), teacher support services, large group meetings and presentations. The spaces accommodating these activities are learning resource center (also referred as media center which includes library, media room, and computer cluster etc.), cafeteria, kitchen, storages, restrooms and showers, and faculty area.

Facility maintenance activities assure that the building systems (HVAC, mechanical, electrical etc.) function properly and don't cause any interruptions in education due to system malfunctions. The basic spaces which support the facility maintenance activities are electrical and mechanical closets, technology closet, central storage, sanitation supply storage, loading and receiving area. The spatial needs for these activities are not limited to the listed spaces. In addition the space names may change from one school program to another. However, the listed spaces form a core of required spaces for facility support activities and represented in Figure 3.4.

The calculation of area requirements of support and maintenance spaces are derived from the number of students in the school, a fixed area suggested in design manuals, or a certain percentage of the area of other related spaces (the space A relates to the space B and the area A is equal to %X of the area B). For example, the area requirements for cafeteria in an ESPS is calculated in OSDM as one-third of the student capacity multiplied by 20 sqf per student or 3000 sqf, whichever is greater.

$$\text{Area} = (\text{Student Population} / 3) + 20$$

if (Area < 3000) then Area = 3000 sqf.

Similarly, the size of the reading room including inner-circulation area is equal to 10% of the student capacity multiplied by 35 SF per student.

$$\text{Reading Area} = \text{Student Population} \times \%10 \times 35 \text{ sqf.}$$

As an example of the third method, the area of the mechanical and electrical space is equal to the %7 percent of the sum of the program areas excluding building services.

$$\text{Mechanical/Electrical Area} = \%7 \times (\text{sum (program areas)} - \text{building services})$$

The OSDM suggests 30 sqf for each of the custodial, electric and technology closets. The size of these spaces is independent of the number of students and other spaces (OSDM, 2001, pg. 2200-11, 2400-14).

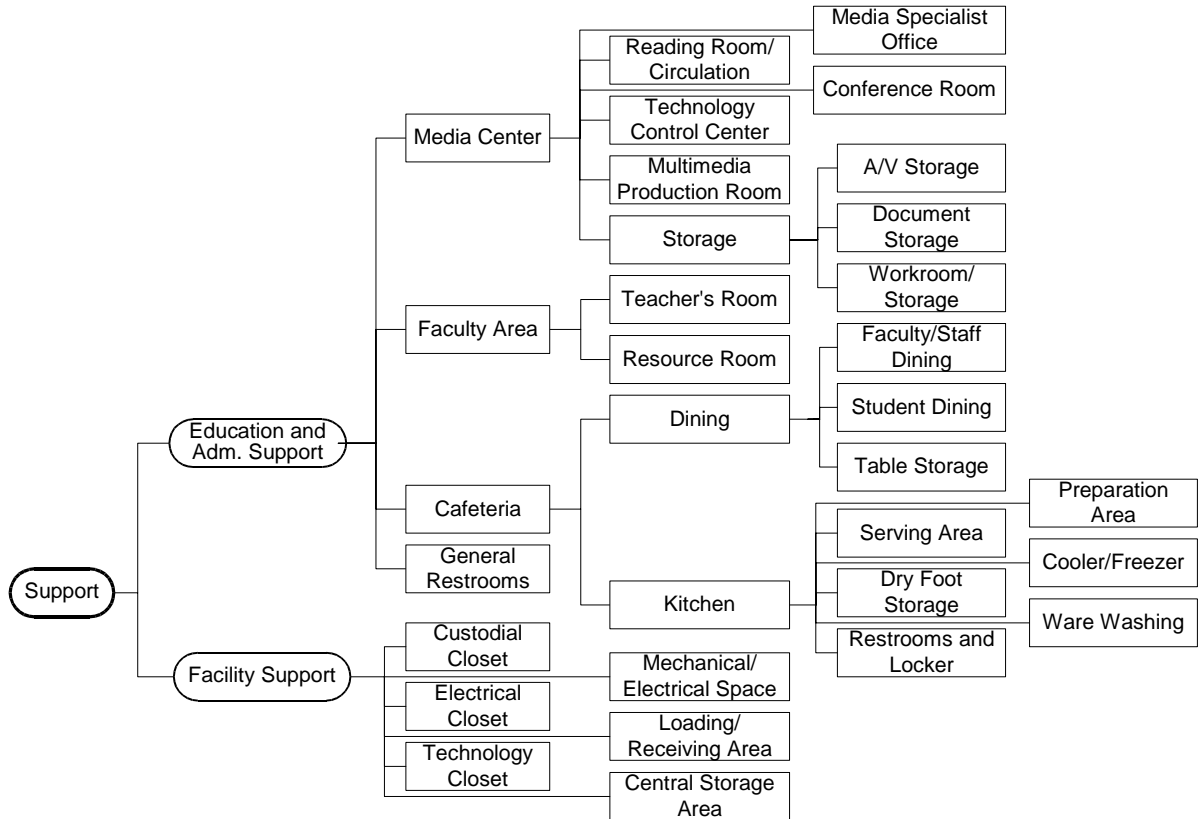


FIGURE 3.4. ESPPS spatial designations of support activities.

It has to be noted that in programming for ESPPS, the spaces listed above may not be included in their entirety. Due to budget or site limitations, many large school spaces are designed for shared functions, such as one single space for auditorium and cafeteria. Another example is that the administration and resource areas are programmed on a school-by-school basis and depend on staffing, operating budget, and other factors. Therefore, determining the spaces of a ESPPS program involves often a trade-off between different space configurations and budget options. The trade-off decisions are made considering the unique situation of the school project.

A complete list of ESPPS space requirements can be found in Appendix B.

### 3.3 Effects of school type and school capacity on spatial requirements.

In the USA education system, the elementary schools consist of kindergarten through 5th grade, and the secondary schools include grade 6 through grade 12. Grades 6, 7, and 8 are taught in middle schools, and grades 9 through 12 in high schools. Depending on the school type (elementary, middle, or high school) the spatial requirements change to

accommodate changing activities. For example, in elementary schools, each classroom is reserved for a particular class of a particular grade, and the students only leave the class for creative program activities. In middle and high schools, the students are not assigned to a particular classroom; rather they attend each lecture in a different classroom, which is reserved for a particular subject (math, literature etc.). The students, therefore, move from one classroom to another during breaks between classes. The number of classrooms in an elementary school equals the student capacity divided by the class size; yet in a secondary school the school population is distributed over classrooms and specialized areas (such as labs). The maximum school capacity for secondary schools can be calculated by multiplying the average class size with the total number of instructional spaces including classrooms, labs, music room etc.

The activity composition of a school is also affected by school type. This is mainly because, at each grade, the students incrementally learn more complex subjects, and complexity of subjects leads to different instruction methods. In turn, each school type requires different space types and requirements. For example, science in elementary schools can be taught in regular classrooms, or in a general purpose lab. In middle schools, more specialized general purpose science labs are needed, while in high schools, each science branch is instructed in a related lab which accommodates the special needs of that subject. In elementary schools, science students learn basic and simple concepts, and the experiments do not require sophisticated settings. In high schools, on the other hand, more complex experiment settings are needed, and each lab is equipped so that students can conduct these experiment in a safe and pedagogically positive environment.

The school capacity is treated as a factor in school design manuals which affects the selection of activities, and therefore the selection of spaces and their sizes. The public authorities may set an upper and lower limits of school capacities. For example, the Ohio Revised Code requires that each classroom facility in an approved project must have a projected or actual enrollment of at least 350 students. Therefore, the minimum required spaces for a school are selected to accommodate at least 350 students.

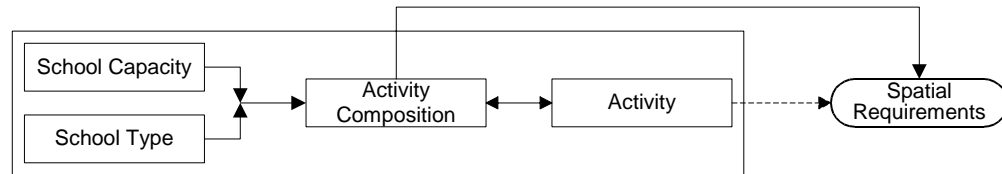
The area requirements of certain spaces are calculated based on certain ranges of school population. For example, in calculating the area requirements of instructional material storage the following table is used. As the school capacity increases, the required area for instructional materials increases: for increment of each approximately 400 students, an additional 50 sqf of storage space is needed.

School capacity (students)	Area for Instructional material storage (sqf)
350-450	50
451-800	100
801-1200	150
1201-1600	200

**TABLE 3.1.** Area requirements are selected considering the school capacity ranges.

As another example, the OSDM requires one physics lab for each increment of 400 high school students accommodated in a school (OSDM, 2001, 2300-2). Thus, if the school capacity is 800 students, at least two physics labs are required. Similarly, the area require-

ments of the other spaces are derived from different formulas incorporating the school capacity as one of the major parameters.



**FIGURE 3.5.** School capacity and type affect activity composition and change spatial requirements.

Finally, an estimate of the total area of the school facility can be derived by using the unit area per student, which is assessed based on existing school facilities and the school capacities. In the OSDM (OSDM, 2001), the unit area range for elementary schools is recommended to be from 115 sqf per student to 125 sqf per student; for middle schools it ranges from 140 to 150 sqf per student; and for high schools, the recommended area per student falls into the range of 160 to 180 sqf. When the school capacity is multiplied by the unit area per student, an approximate target area for a school facility can be obtained. The result is used as a benchmark during allocating the budget and architectural programming.

The school type and capacity, which are non-spatial parameters (as initial requirements), determine the activity composition, which comprises school activities and their relations. Each activity requires a specifically assigned space in a facility. However, due to the some restrictions such as budget or site, more than one activity can be accommodated in one space (e.g. cafeteria is used both for dining and large group gatherings). As seen in Figure 3.5, these non-spatial requirements (such as class size, school population, school type, budget etc.) play an important role in determining the spatial requirements of a school facility.

We can summarize these non-spatial parameters as follows:

- School capacity (the number of students that can or should be accommodated in a ESPS),
- School type (elementary, middle, high schools, combination of elementary and middle schools, combination of middle or high schools, or combinations of all types in one facility),
- Maximum number of students allowed in one classroom (maximum class size),
- Activity composition and activity relationships,
- Budget.

These parameters are used in formulas and procedures along with some spatial parameters in order to calculate spatial area and number of spaces required in a ESPS.

### 3.4 Spatial requirements generation for ESPS.

In order to summarize the observations that we made during the analysis of architectural programming of ESPS, we presented below a graphical framework. The framework contains the activity-space couples, parameters which are used in spatial requirements calculation, and formulas and procedures. Each space name implicitly refers to the activity which it accommodates, and each space (entity) is represented in a box, which may be attached to one or more parameters, formulas and procedures. Formulas may include logical predicates or mathematical equations. If more than one formula is needed for making a particular decision, the formulas are encapsulated in one procedure. In addition, procedures can refer to other procedures. The parameters are divided into two groups: independent and dependent. Independent parameters are those which are not directly attached to an activity or spatial entity, and dependent variables are the ones which relate to an activity or spatial entity.

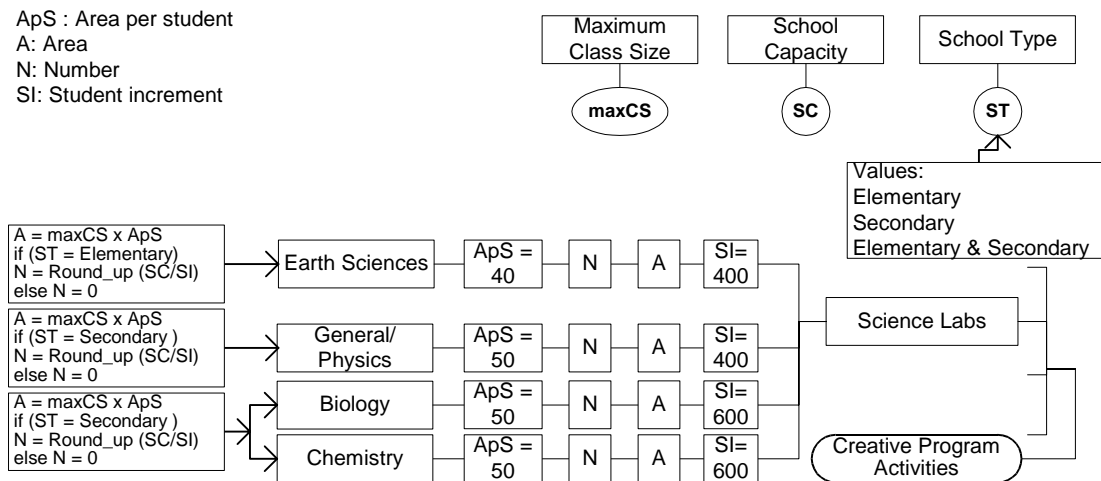


FIGURE 3.6. A partial view of the graphical framework for ESPS programming

The Figure 3.6 partially shows how this framework represents programming ESPS. In the figure, as part of creative program activities, students are instructed science (physics, biology, chemistry etc.) in a lab environment, where they can also conduct experiments (learn-by-doing). The labs for these activities are physics lab, biology lab, chemistry lab, and general purpose lab. In the figure, each of these labs are represented with a rectangular box, and the activity which they relate to is represented with a rounded-corner rectangle. We define independent parameters as the maximum class size (maxCS), the school capacity (SC), and the school type. The class size and the school capacity are numeric parameters, and the school type is a string parameter with a limited set of values. The dependent parameters are attached to spatial entities, and each is assigned a specific value. The required unit area per student for a particular space, the required area, the number of a space, and the increment of the number of students are the dependent parameters in the

example. The class size, the school capacity, the school type, the area per student, and the student increment are parameters which are required from the programmers. The formulas are used to calculate values for, such as, the number of a particular space and its area. The procedures are represented in a box with an open-arrow head attached to the space box. The procedure which is used to calculate the area requirements for a physics lab requires values for the area per student, the student increment from the physics lab space box. It calculates the area requirements by multiplying the unit area per student by the class size. The number of physics lab is calculated if the school type is secondary school. If so, the school capacity is divided by the student increment and the result is rounded to the next integer value, which is assigned to the number of space parameter. For an elementary school, no physics lab is needed.

This framework provides a graphical representation for describing how non-spatial and spatial information is used in generating a design requirements for an ESPS facility.

---

### **3.5 ESPS activity affinities and spatial requirements.**

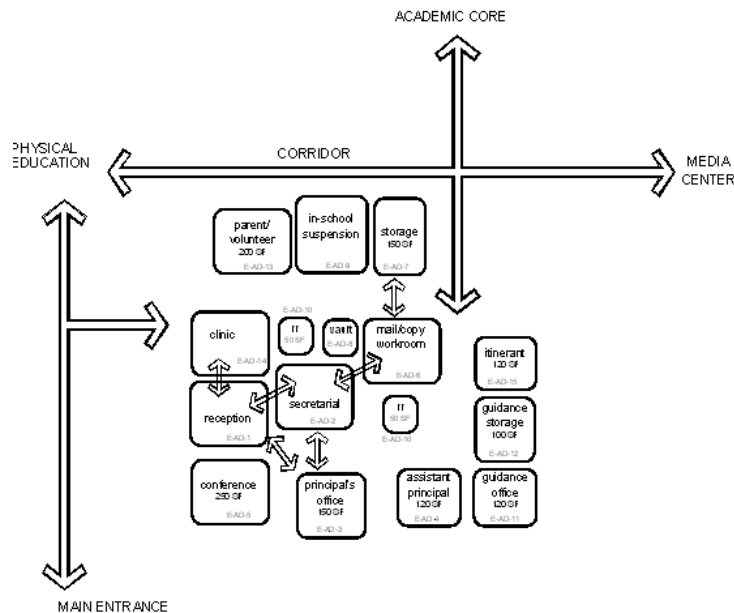
#### **3.5.1 Overview**

In the ESPS programming literature, some spatial-affinity schemes for the ESPS facilities are prescribed (OSDM, 2001; Perkins, 2001; De Chiara and Callender, 2001; Brubaker, 1998; Leggett, 1977). These schemes covers either very abstract spatial-affinities, which are very ambitious and cannot apply directly in programming; or very specific-affinities, which can apply to only one particular school facility. In both schemes, the actual motivations for why one space should relate to another is not described; which we believe that this is more important than rigidly establishing spatial relationships without knowing the rationale behind.

The spatial affinities is mainly derived from activity affinities. In the literature, there is not an affinity schema which explains their consequences on spatial affinities. However, we observed that the activities and their affinities (along with organizational structure) are the major factors in establishing spatial-affinities. For example, the principle's office, general office, secretarial area, record storage, meeting rooms are some of the typical spaces allocated in the school administration area, where different and related activities take place. These spaces are allocated in relation to each other because the activities which take place in them require a close proximity relationship (such as the principle supervises the administrative activities the office personnel meet periodically in the meeting rooms, the secretarial personnel updates the student records and store them in the storage area etc.) Similarly, the service activities (such as food preparation) are highly coupled with the activities depending on these service activities (such as dining); therefore the service spaces (such as kitchen) is located near by the served-spaces (such as cafeteria). The activity relationships may require occupants move from one location to another, and this may imply a proximity relationship between these locations.

In a school facility, simultaneously, a wide range of activities take place. This creates a very dynamic environment in which any interruption may cause inefficient functioning of the school facility. The establishing spatial affinities based on activity affinities assures that this dynamic environment functions properly. There may be other factors, such as privacy, acoustic isolation, security etc., which directly relate to activities and affect the spatial relationships.

Symbolic representations of spatial affinities in the architectural programs also become an issue when generating or interpreting the spatial-affinity requirements. The current representation techniques of spatial-affinities (such as affinity diagrams as shown in Figure 3.7) may cause designers to misinterpret the spatial relationships. For example, the representation of space symbols closer or at a distant to each other may imply that these spaces should be located actually as they are depicted in the diagram. However, the location of the main entrance could be from some other direction depending on the site and other activity relations, or assistant principle's office may also need to have access to secretarial area or be located on the other side of the principle's office. In addition, the representation technique doesn't include non-spatial information such as activity relations. However, activity affinities combined with spatial affinities may convey a more comprehensive and clear description of why a space should relate to another.



**FIGURE 3.7.**A diagram selected from (OSDM, 2001, pg. 4103-1) as an example of how spatial relationships (affinities) are represented in general.

In the following, we analyze activity affinities and spatial affinities of ESPS facilities.

### 3.5.2 ESPS activity affinities.

In the literature, the ESPS activity relationships are not explicitly documented. However, most of the programming guidelines and design manuals provide lists of required space and their relationships. From the descriptions of spatial relationships and space names, we understand what are the activity taking place in a school building and what kind of affinities can exist between different spaces. By this way, we can establish a logic of transition from spatial affinities to activity affinities.



This provide us the actual reasons for the need of a space or the relationship between different spaces. For example, the education activities are broken down into mastery and creative activities, which each of them complements the others in education program. The classrooms and creative activity spaces (such as labs) are self contained in that the activities taking place in each of them does not directly relate to each other. In other word, the students in a classroom do not interact with the students in a science lab. However, one class leaves the classroom (or lab) and moves to another (class or lab) during a recess. Therefore, students and teachers circulate periodically between different zones and spaces. Unlike this example, in the administration area, secretarial activities directly relate to the activities taking place in office areas. The staff and teachers located in these two zones share common activities very often (such as teacher's or staff meetings). In determining the accessibility and proximity affinities between the mentioned spaces above, we evaluate how the activities are performed and what possible activity affinities could exist.

In analyzing and representing activity affinities, we believe using the Unified Modeling Language (UML) notations, particularly use cases and sequence diagrams, would be an appropriate method (for details refer to Booch, G., J. Rumbaugh and I. Jacobson., 1999). For example, in order to model the activities being performed during a lunch break, we sequentially list each of the interaction between the parties and spaces involve in this activity in a use case (Booch, G., J. Rumbaugh and I. Jacobson., 1999). A use case describes a particular use of a system and during the use what kind of activities can occur and what types of objects involve during the activity. Therefore, an ESPS is the system we intend to partially model, and we analyze the activities which take places during the *lunch break*. The participants who perform the activities and spaces are also included in the use case.

1. The students are in the classroom.
2. The administration informs the lunch time
3. The teacher releases the students.
4. The students leave the class and move through the corridor.
5. The students line in front of the food serving counter in cafeteria.
6. The students sit and eat in the cafeteria.
7. The students dispose the food trays near kitchen.
8. The students leave cafeteria.
9. The students move through corridor to attend next class.

The same activity sequence is demonstrated by an activity sequence diagram (Figure 3.8). In the diagram each space represented as an object (rectangle) and building users are represented with an actor symbol of the UML. The arrows connecting the objects or actors describe the interactions between them, and the arrow head shows which object is active during the interaction.

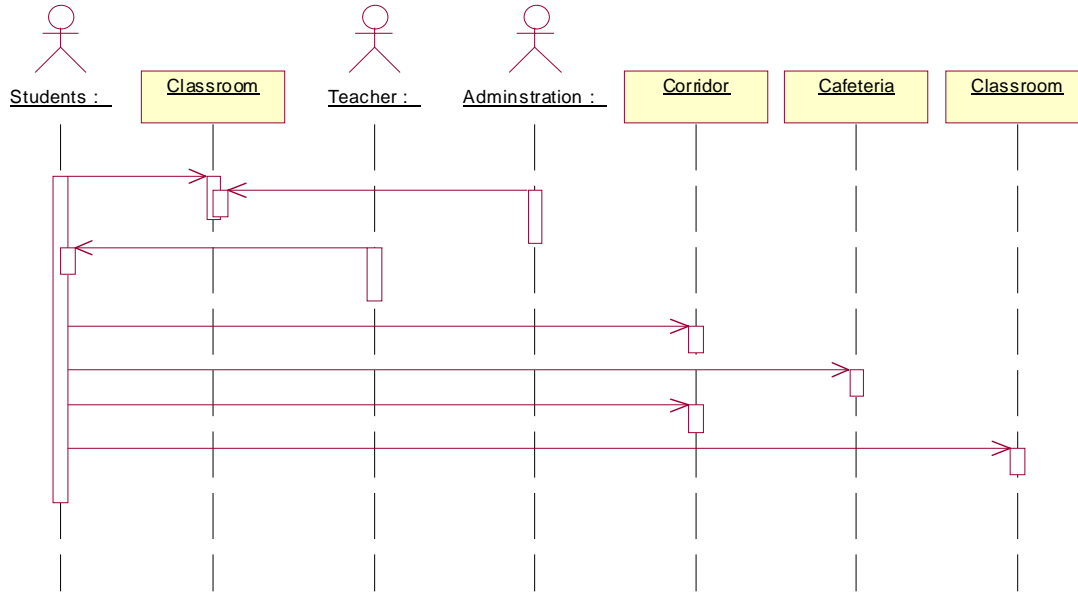


FIGURE 3.8. Sequence diagram describing the activities, spaces and participants during the lunch break.

### 3.5.3 ESPS spatial relationships.

In ESPS facilities, we have observed that there are three basic spatial affinities. The first one is that a space in a school may need to be located at a certain physical distance to other related spaces. This can be called *proximity*. For example, cafeteria and kitchen has a close proximity relation so that the food from kitchen area can be delivered to the service area located in cafeteria without any interruption. The second is that accessing from one space to another need to be limited for certain building users. For example, only authorized personnel can access to service areas such as kitchen and mechanical rooms. The students should not directly access to the kitchen from cafeteria. Between the cafeteria and the spaces where education activities take place (such as classrooms) there must be a noise barrier. Therefore, the access type could be *physical*, *visual*, and *acoustic*. Other than proximity and accessibility (physical, visual, and acoustical) affinities, a third one could be the *overlapping* affinity, which occurs when one space is used for multiple activities. For example, the cafeteria or gymnasium is a suitable example for this affinity type. The activities such as ceremonies, presentations, large group meetings, and even some non-school activities can be performed in these spaces.

---

### 3.6 Summary and discussions.

ESPS programming starts with determining the non-spatial requirements (needs) of a facility based on the education programs, projected and current student enrollment, school type, construction and operation-sustaining budgets etc. As non-spatial requirements become clear, the spatial requirements of the facility are derived from these requirements. For example, the activities which are planned to take place in the school, are specified in education programs and an activity composition is formed by considering the relationships between each activity. The activities, in turn, yield to space requirements. The spaces are usually named after the activity names that they accommodate. The activity composition is transformed into spatial relationships, which could be proximity (distance between two spaces), accessibility (acoustic, physical, and visual access), and overlapping (two activities take place in the same space).

Other than activities, the school organization structure plays an important role in determining the spatial requirements. For example, the roles of the building users are reflected on the requirements such that the principle of the school and the principle's assistants usually require to have their own offices, while regular office personnel share the same office area. Teachers, on the other hand, use the classrooms for both instruction and study purposes.

The school population, suggested class size, and school type are the other parameters which are used in calculating the numbers and physical properties of the spaces.

The design guidelines for ESPS programming provide most of the ESPS programming information in a structured way. The information presented in these guidelines describes the general quantitative and qualitative features of a school facility. For example, each guideline or design manual describes the ESPS facilities by listing some general non-spatial requirements (such as class sizes, the school capacity etc.) and spatial requirements (such as the spaces to be allocated, their sizes). The most of the information is provided in this documents and programmers only need to focus on each facility's project specific (unique) requirements.



---

# 4.

## *Case Study: Ambulatory Health Care Facilities*

---

### 4.1 Introduction

AHCFs, also called medical office facilities in the literature, have gained importance because of increasing demand for outpatient treatments. The basic objective of such facilities is to provide health care services for patients who do not need over-night hospitalization or long-term care. AHCFs, therefore, are distinct from hospitals, on the one hand, and extended care facilities on the other hand. AHCFs are aimed at reducing the cost of providing medical services. Some of the basic reasons for this demand are the following (Kobus et.al., 2000, pg. 1-7)(CDC and NCHS, 1998):

- There is a shift towards managed care because the cost of prevention is less than the cost for healing acute illnesses.
- Both the increasing aging population and younger population are seeking for more preventive medical care. The demand for preventive medicine will require building more AHCFs.
- Improvements in the medical field and technology create a necessity of redefined spatial requirements. These improvements also change patient-physician relations and definitions of activities. New facilities have to accommodate these changing aspects of care.
- Employers are providing insurance options to their employees, and most of their coverage requires a step-by-step treatment procedure, where a patient would first see a general physician before requesting service from a specialist.
- Rising cost of both providing and receiving medical treatments require a careful planning of medical facilities.

Health care is also becoming a universal service. And wherever quality health care is offered, the patients from different places of the world come there to seek treatments. All of these reasons make programming and designing such facilities interesting.

In the following sections, we wish to outline the variables which are considered in architectural programming (design requirements specification) of AHCF. Following this, we would like to demonstrate how these variables gradually become spatial requirements during the programming phase.

---

### 4.2 Effect of the medicine speciality on the program

The top-level variable for programming AHCF facilities is the health care service type, which can be grouped under four classes. These classes are specialty-dependent and can be listed as *primary care*, *specialized medicine*, *diagnostic medicine* and *group practice* (Malkin, 1989, pg. 17). Each of these classes comprises both common and unique activities. The staffing pattern also shows differences in each class.

*Primary care* is composed of *general practice*, *internal medicine* and *pediatrics*. Basic responsibilities of these specialities are to examine patients and, depending on the seriousness of the complaints, to provide treatment or to refer the patient to a specialist; to keep patients' health records; to consult or educate patients for issues related to preventive medicine. The second class *specialized medicine* (such as *obstetrics* and *gynecology*, *urology* or *dermatology*) involves with specialized patient care (such as pregnancy and fertility), consultation (such as about birth control and sexual diseases) and treatment of illnesses occurring in certain parts of the human body. Diagnostic medicine, as the third class, includes diagnostic activities. *Performing medical tests*, *radiology*, *ultra-sound* are some of the activities carried in the third class health care services. Results of the tests are interpreted by a diagnostic medicine specialist and documents are transferred to the physician who referred the patient. Group practice includes single- or multi-speciality activities. The American Medical Group Association (AMGA) (AMGA, 2001) defines group practice as

"...the provision of health care services by licensed physicians and /or dentists practicing in a structured organizational setting that has an identifiable and functionally integrated system for both business and practice management. This includes shared facilities, equipment, personnel, records and professional staff responsible for quality of care."

The objective of this practice is to provide more efficient staffing and space planning through sharing resources for one or more types of specialties. Economics plays a very important role for establishing a group practice. For example, providing diagnostic medicine activities in a group practice is more cost-effective than providing the same services in a single-speciality small medical office. This also brings advantages of receiving diagnostic results as soon as needed and helps keeping the records in the same organization.

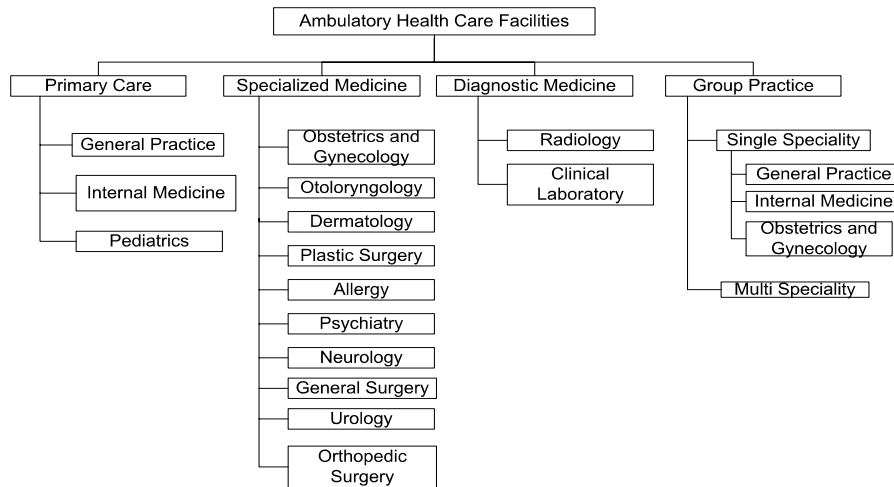


FIGURE 4.1.Categories of ambulatory health care services.

Specialty is one of the main variables for the selection of activities and staffing. When we decide an activity composition for a specialty, initially the services that this specialty will

offer are decided. For example, a dermatologist's office generally doesn't need to include a specimen testing lab, but it may need to have an ultraviolet treatment room. However, the specialty is not the only criteria in making decisions about an activity composition. Staffing, patients volume, number of physicians also play an important role. Take internal medicine as an example: if the number of patients that will be seen is not sufficient enough to afford an in-house radiology facility, this activity can be excluded from the requirements even though it is a required activity for this medicine speciality. On the other hand, for economical reasons, an internal medicine facility may have to provide improved specimen testing activities along with regular testing procedures, even though they are not required generally. Therefore medicine speciality can determine an outline of basic activity composition for a AHCF, but it cannot enforce accommodation of all activities. Refinement of an activity composition continues by reviewing the effects of other variables.

---

### 4.3 Activities in ambulatory health care facilities

From the architectural programming perspective, AHCF activities can be grouped into three: The first group includes activities common among all specialties. The second group contains activities that are shared between one or more specialty, yet not all of the specialties require these activities. The third group of activities are highly coupled with a specific specialty and not required for other specialties. For example, *examination of patients* is a common activity for each specialty, whereas *testing of refraction of eyes* is a specialized activity only for ophthalmology. Also, some of the medical specialties such as otolaryngology or internal medicine may require *specimen testing*, but psychiatry and ophthalmology do not. Thus, activities like *specimen testing* fall into the second group of activities.

These activities are a major factor in determining the spatial requirements, because each activity needs a space. From activities, essential spatial requirements can be derived through evaluating required participants, required instruments, devices or tools, or how often these activities occur. These aspects will be explained in the following sections.

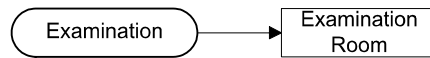
In addition, each activity is known by an activity-related name. The space names are highly coupled with the activity that they accommodate. Mapping from activity descriptions to space names also needs to be investigated to understand the transition from an activity to a space in the programming process. The following section describes the activities that directly affect architectural design requirements and mapping from activity descriptions to space designations.

#### 4.3.1 Common activities

**Patient examination:** Patient, physician and, if required, a nurse are involved in this activity. The physician investigates the patients' complaints. If preferred and allowed by the physician, patient's company can also join but plays a passive role. The nature of the activity is private.

Examination requires a space where examination and routine consultation with patients take place. There are two basic considerations. The first one is related to the area requirements. If a general purpose examination room is needed, area requirements depend on allocating a full-size exam table, a built-in sink cabinet with drawers and cabinets above, a dressing alcove (which is optional), a desk for

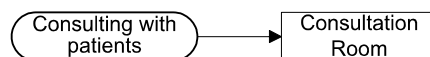
note taking, one mobile and one regular chair. Standard area for such a room is about 96 sqf (8 x 12 feet); however, again it can be increased or decreased depending on the situation. If special equipment, such as EKG or treadmill, is needed, the area is recalculated by incorporating the area required to accommodate the equipment.



*The number of exam rooms* depends on the medical specialty and the number of physicians. For example, for an internal medicine practice an average of 2.5 exam rooms is needed for each physician (Malkin, 1989, pg. 47). If there are three physicians specialized in internal medicine, there is a need for 8 exam rooms.

**Consultation:** Depending on the stage of the treatment, the physician explains the possible reasons for the patient's complaint, refers patient to other specialists or to diagnostic services. If the results from diagnostic services are at hand, the physician explains the findings to the patient. The activity is a private discourse between the patient and the physician.

The activity takes place in a *consultation room*, which is also the private office of the physician. The typical furniture of the office is a study desk with chair, book shelves, and depending on the budget, a couch or chairs for patients to seat. The minimum size for this room is about 96 sqf. (8x12 feet). Each physician is assigned a consultation room in an AHCF. *The number of the consultation rooms* must be equal to *number of physicians*.



**Waiting:** Patient and patient's company wait for being called for examination, completion of paperwork, pre-examination activities (such as weight and height measurements), post-examination activities (waiting for prescription or paperwork to be completed), and examination-independent activities (e.g. shots, payments, or even resting after the examination).

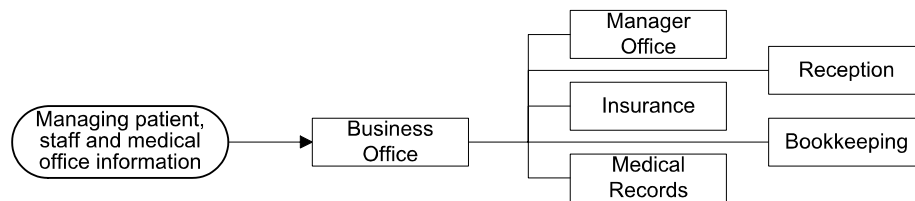
*The waiting room* is where the activity takes place. Area requirements for waiting spaces depend on the *number of physicians* and the *number of patients that a physician can see in an hour*. The area is calculated by multiplying the number of seat requirements and unit area requirement per seat. The calculation of area requirements is explained in detail in Section 4.6.





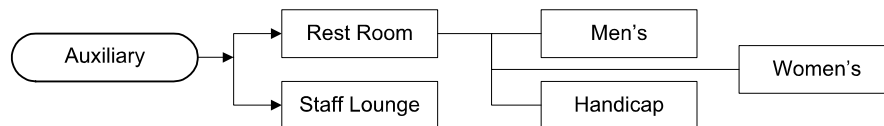
**Patient-, staff- and medical office- information management:** Managerial or secretarial staff keeps medical, insurance and payment records of patients. For a new patient, a new file is created; for a current patient, patient's record is retrieved before the examination. One of the managerial activities is to schedule staff working hours and schedule patient visiting times.

The *business office* is the location where these activities take place. The business office may contain other spaces such as insurance office, bookkeeper office, office manager's office, reception, medical record archive. The existence of each space depends on the *number of physicians*. In a normal-volume office, there is usually one office manager, one insurance person, two receptionists, and one staff member for every two physicians (Malkin, 1989, pg. 65).



**Auxiliary:** Activities in this designation are supported by rest rooms, water fountains or vendor machines. These spaces serve waiting patients and their companies, and staff. There is also a need for a staff lounge where staff members can rest, eat or socialize.

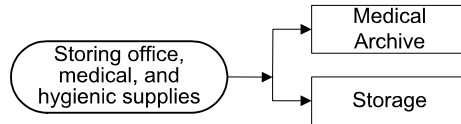
Rest rooms constitute toilet stalls, urinals and sinks. The number of these constituents depends on the specialty and number of physicians. Rest room area to be allocated is calculated considering the number of toilet stalls and sinks. For handicap access, a rest room design incorporates design criteria instructed in the American with Disabilities Act (ADA).



Any office with more than three employees has generally a staff lounge with the average size of 96 sqf. The staff lounge includes a counter with cabinets and sink, under-counter or regular-size refrigerator, microwave oven and coffee machine.

**Office, medical, and hygienic supplies storage:** Different storage facilities are needed for different supplies. Each storage is maintained by different staff; for example, nurses are responsible for medical supply, office staff takes care of storing office supplies and archiving out-dated paperwork.

For each increment of three physicians, as a standard, there must be a storage space allocated with a minimum size of 5 sqf.



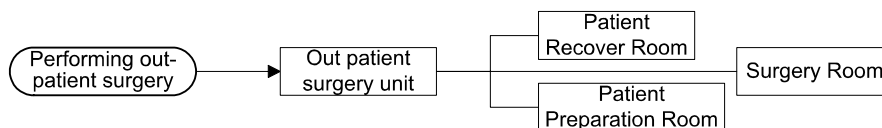
**Minor surgery:** Minor surgery such as treating accident cases, removing burned tissues, treating cuts is performed by a specialized physician who is accompanied by one or more nurses. Sometimes local anesthetics is also needed. People accompanying the patient may stay with the patient during the operation if it is allowed by the physician.

The activity takes place in a large exam room (12 x 12 feet). The examination table is replaced with an operation table with a suspended operation light. Cabinets and counter are designed to accommodate minor surgery-related equipment and tools. For small AHCFs, *immobilization treatment by casting* also takes place in this room. Therefore, the space must be sized accordingly.



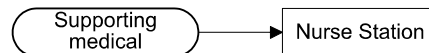
**Out-patient surgery:** This activity requires more complicated surgery settings and devices than minor surgery. Local or general anesthetic may be needed during the surgery. Also, a patient is kept in a recovery room following the a surgery. The patient is released on the same day of the operation. Internal medicine, otolaryngology, plastic surgery, general surgery, and orthopedic surgery specialties may perform this activity. If the volume of the office is relatively low, (for example, there are only two physicians practicing in the office) this activity and minor surgery can take place in the same space.

Out-patient surgery requires a motorized operation table and an operation light suspended from the ceiling over the table. Built-in cabinets to store surgery instruments, linens, and a sterilization area are other factors effecting spatial requirements. The activity also needs a patient recovery room and a patient preparation room connected to the surgery room.



**Medical procedures support:** Physician's aides or nurses assist the physician by performing pre-examination and post-examination tasks. Some of the tasks are weighing patients, sterilizing instruments, sorting or dispensing drug samples, giving injections, performing routine lab tests or contacting patients over the phone. Depending on the speciality, the activity can expand to include other tasks such as nurse practitioners' role in obstetrics gynecology.

The nurse station (room) is the space where staff supporting medical activities are take place. The station size depend on the number of aides who will use the station. The number of supporting staff can be estimated on the basis of each physician requiring one or two aides depending on the practice. For each additional supporting staff over three, 10 sqf. additional space is added to a standard 8 x 12 feet space. The typical space furniture is a 6 feet long counter with wall and base cabinets and a study desk for each supporting staff.

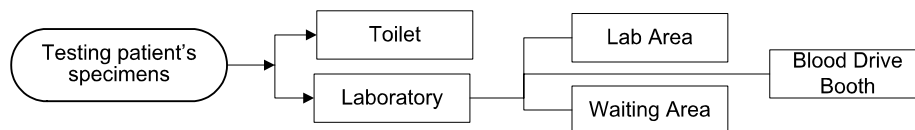


#### 4.3.2 Semi-specialized activities and descriptions:

Semi-specialized activities are performed by some of the medical specialties (which are explained in Section 4.2.) and not a necessary part of all specialties like common activities. Semi-specialized activities include the following:

**Specimens testing:** Sample specimens from patients are tested by expert staff. The patient provides tissue, blood, or urine samples to be used for analysis and diagnosis. Nurses or technicians take a blood or tissue sample and compile the analysis results. The physician may also want to examine the samples during the testing stage.

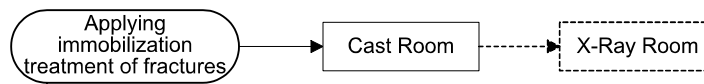
The activity takes place in a laboratory. Depending on the specialty, a waiting area in front of the lab, a blood drive room, and a lab storage can be added. The lab must have easy access to a toilet. Area requirements for these spaces are also affected by specialty-specific additional activities. Lab equipment and cabinets play an important role in the layout of the spaces.



**Immobilization treatment of fractures:** This activity consists as *putting a cast on a fractured area of the body*. The activity may require diagnosing a fractured area before applying a cast. if the *number of immobilization treatments performed in an office* (like number of patients seen by the physicians or number of physicians) justifies it, a local X-ray may be required; otherwise, the patient is

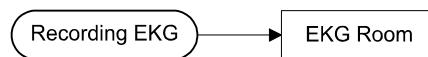
referred to a radiology provider. This activity is specific to general practice and orthopedic surgery.

The activity takes place in the same room where minor surgery takes place in a general practice. For orthopedic surgery, the space has to be furnished to accommodate a full-size table in the center of the room as well as cast materials and storage. If the room has an X-ray machine, its area requirement has to be considered. If the *X-ray taking* happens in a separate space, the casting space must be located close to the X-ray room.



**Electrocardiogram (EKG):** This activity records the electrical activity of the heart on a moving strip of paper. The electrocardiogram detects and records the electrical potential of the heart during contraction. The activity is usually performed by a specialist technician or by a nurse. The patient is either asked to lie on an exam table or perform certain exercises such as walking on a treadmill. This activity requires mobile and stationary high-tech devices. The activity is performed in general practice and internal medicine.

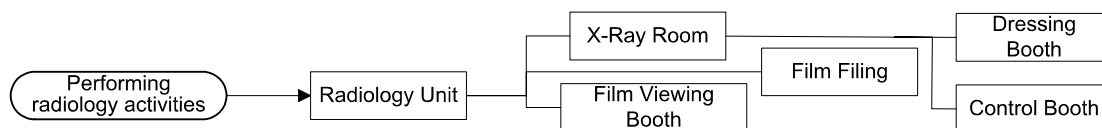
The activity takes place in a room slightly bigger than an average exam room. A treadmill, an EKG device, and a full-size exam table are accommodated in the room. The room is designated as EKG room.



**Radiology:** This activity consists of taking and studying of X-rays. The activity takes place at different stages, and these stages involve technical staff, physician and patient. Preparation of a patient (undressing and, if necessary medication giving), X-ray taking, providing protection for the technician, film processing, film storing, evaluating and diagnostics are related activities. This activity is part of general practice, internal medicine, otolaryngology, urology and orthopedic surgery.

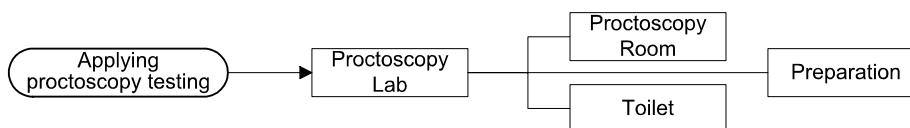
The main activity occurs in an X-ray room, which is supported by a dark room, a control alcove, a film filing archive and a film viewing alcove. A dressing area, which could be 3x3 sqf area enclosed by curtains, for patients is also required. The combination of all of these spaces is called radiology service. The facility's spatial requirements include some technical complexities which are due to advanced technology or devices used in the activity. A room with a full size X-ray machine occupies a minimum of 120 sqf. The room must be insulated to prevent the danger of radiation. Depending on the specialty and the patient volume,

smaller X-ray machines can be used. The size of the X-ray room changes in response to changes in the spatial requirements for the equipment.



**Proctoscopy:** Proctoscopy is used to examine the rectum by using endoscopic techniques. "It is used to locate, identify, and photograph pathologic alterations, to obtain biopsy material and perform other surgical interventions, and for delivery of medication" (Cancer Medical Dictionary). Biopsy material is sent to testing labs for further analysis. A patient, a nurse and a specialist physician are involved in the activity. The activity must be connected to preparation for testing activities (both for physician and patient), and access to a rest room must be provided. Internal medicine, urology, and obstetrics and gynecology are specialties which require proctoscopy testing.

A proctology room is the space where the activity take place. The room includes a full-size exam table in the center of the room, a dressing area, a counter and cabinets. A 20 sqf. preparation area, which is adjacent to the proctology room, is also required. The room must have access to an ADA-compliant rest room. As a pre-condition for this room's addition to the program, there must be at least four physicians practicing in the same internal medicine office or three physicians urology.



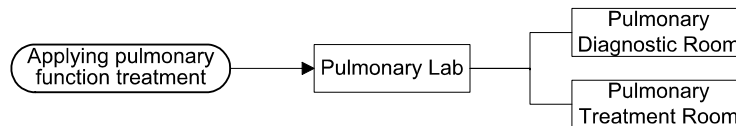
### 4.3.3 Specialized activities

Specialized activities are unique to only one of the medical specialty as introduced in Section 4.2. These activities can consist of the following:

**Pulmonary function treatment:** Treatment occurs after patient screening. The patient's pulmonary dysfunctions is treated by letting him or her breath pressurized air with medication. A positive-pressure (breathing) device, an EKG device and some other exercise devices are used during the treatment. A pulmonary technician, a physician and a patient participate in the activity. This activity is unique to internal medicine.

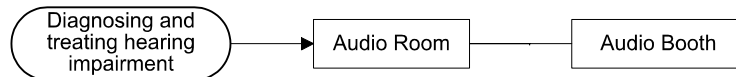
The activity takes place in *pulmonary lab*. There are three possible configurations for the lab. In the first setting, pulmonary screening and treatment take place in one space. In the second one, pulmonary screening is combined with car-

diovascular screening. In the third setting, pulmonary screening is accommodated in a separate space from pulmonary treatment. The selection of configuration depends on the patient volume. The first setting is useful for low-volume medical offices. The second and third settings could be selected if the volume is medium and high, respectively. The area requirement for the activity is about 10 x 10 sqf.



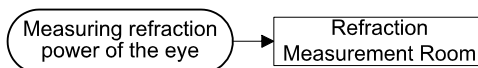
**Hearing-impairment diagnosis and treatment:** A patient's hearing ability is measured or treated in a soundproofed space. A technician, a patient and if necessary a physician participate in the activity. This activity is specific to otolaryngology.

The space for this activity is called an *audio room*, which includes a soundproofed booth about 30 sqf. and an observation space that will accommodate a study desk and audio equipment. The technician visually observes the patient's reactions through a window to the audio booth. The area required for these activities is about 120 sqf.



**Refraction power measurement of the eyes:** This activity is special to ophthalmology. Complicated devices are used for refraction testing of the eye, where the patient is asked to read several sizes of shapes from a certain distance (20 feet from chart to patient). Before certain other tests, patients are given drop medication to dilate the pupils of the eyes; medication requires a certain time to show effect. The activity requires special lighting configuration that can be adjusted from dark to different levels of lighting. A nurse or a physician and a patient participate in the activity.

This activity and patient examination take place in the same room for ophthalmology. The main constant that affects the room size is the distance of 20 feet for reading the symbols on a chart. This can be reduced to half by using a special setting with a mirror and a projector. The width of the space can be the same as for a regular exam room. The furniture and equipment accommodated in the room are refraction instruments, a special refraction measurement chair with attached optical devices, and computer-controlled eye-screening equipment. The room size can be 10 x 14 sqf.

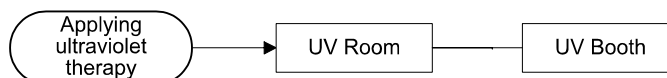


**Electrolysis and facials:** The activity is usually associated with cosmetic surgery. The removal of unwanted body hair by electrocuting the hair roots with an electrified needle and applying cosmetic medicine to the operated areas are two major tasks performed by a nurse or a physician.

A typical exam room with an additional area to accommodate an electrolysis device is sufficient to support this activity. This space is called as *electrolysis room*.

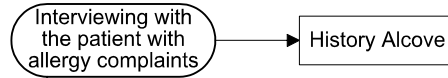


**Ultraviolet therapy:** Certain skin diseases can be treated by subjecting the skin to ultraviolet light. The ultraviolet light source is usually located in a silver foil-lined box in a room that a patient can fit in. If the light source is located in a room, therapist also can enter to the room with the patient. The area requirement for this activity depends on the configuration. An ultraviolet booth (box) occupies about 4x5 sqf. in a room which must be as large as an exam room. A mobile ultraviolet treatment box can also be used if the number of physicians in the office is less than three (i.e. a low-volume medical office).



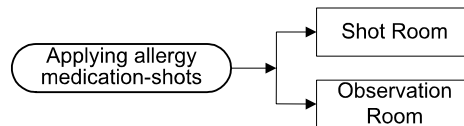
**Interview patients with allergy complaints:** As the activity name explicitly states, this is specific to immunology and allergy specialties. When a patient visits a physician's office for the first time, a nurse asks the patient several questions to gain knowledge about the patient's health history. The answers of the patient can directly be recorded on a paper form or entered into a computer application by the nurse. The nature of this discourse is private.

The space in which an interview takes place must accommodate a desk (2x4 feet) and three chairs (one for the nurse, one for the patient and one for the patient's company). It is called as *interview room* or *history alcove*. A 6x6 sqf. space is sufficient for this activity. The number of interview rooms is determined by allocating one room for each increment of two physicians.



**Allergy medication-shots:** Allergy medication is given to a patient, and the patient is expected to stay in the ACHF until the physician makes sure that the patient's body does not show adverse reaction to the medication.

The activity requires two distinct spaces, one where the shots are applied and one where the patient is observed after the shot. These spaces are called shot room and observation room, respectively. They must be adjacent to each other.



**Allergy symptoms diagnostics:** Usually allergy reasons are not clear at the outset, and the patient must undergo several allergy tests to reveal possible causes of the complaints. The patient lies on an examination table and a nurse or a technician performs the tests by applying allergens. The patient's reaction to each allergen is recorded and passed to the physician for further diagnosis and treatment decisions. Sometimes, the allergist observes the reactions directly.

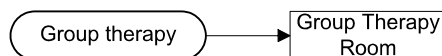
The activity takes place in a *test room*, which requires an area to accommodate an exam table adjacent to a wall, a counter with cabinets, and enough space to use a mobile table for allergens. The optimum area for these is about 96 sqf.



**Group therapy:** Psychiatry requires a space for group therapy. The space must accommodate several patients, a physician and, if needed, a recording nurse or typist. The setting should be relaxing, conversation-motivating and not distracting.

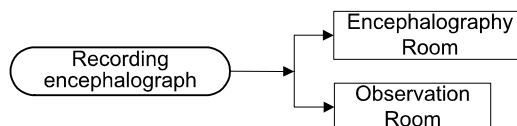
The group therapy area is calculated by considering the number of patients attending a session at a time. This number changes from one psychiatrist to another; however, between 15 to 20 sqf area per patient is accepted as a reasonable standard. The room must be designed to accommodate seating for each patient, a chair for physician, and possibly an aide to take notes. The session can be recorded on video or audio media. The layout must consider these factors as well.





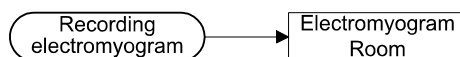
**Encephalography:** The electrical activity of a patient's brain is measured with an instrument that records electrical potentials on the scalp. Physicians and technicians who practice neurology perform the activity.

In an encephalograph recording room, the patient lies on a full-size exam table. Connections to an encephalograph are attached to the patient's head. The patient is observed from an adjacent observation room. In order to accommodate the required furniture and equipment, the observation room needs an area as large as an exam room (12 x 8 sqf.).



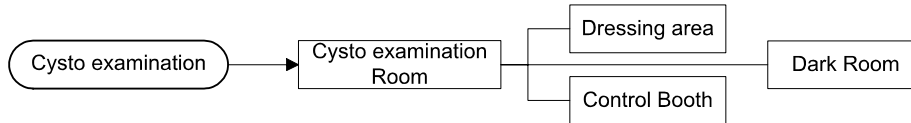
**Electromyogram:** An instrument is used to measure if a muscle is deteriorating and if it can be rehabilitated. It is used by a physician or technician (nurse) in a special setting. The results of the measurement show the strength of the muscle and indicate if the nerves are effected. Neurologist or nurses perform the activity.

A room that accommodates electromyogram device mounted on a movable table is required. The room can be slightly smaller than an exam room (10 x 10 sqf.)



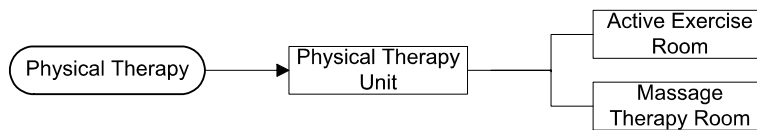
**Cysto examination:** A patient with a urinary complaint is examined by cystoscopy. If required by physician, an X-ray device is used in the same room along with a cystoscopic table. A patient and an urologist are involved in this activity.

If there is only one physician practicing in the office, one cysto-exam room is sufficient. The room accommodates X-ray equipment mounted over a full size table on the ceiling. A control area, dressing area and a dark room to process X-rays are additional spaces required. For each increment of two doctors, an additional cycto-exam room without X-ray is required along with a cycto-exam room with X-ray. The cysto-exam room with X-ray requires about 12 x 18 sqf. Without X-ray, the room size can be as big as an exam room.



**Physical Therapy:** The activity involves the treatment of physical dysfunction or injury by the use of different types of exercises at different intensity levels. It is intended to restore or facilitate normal function or development of the patient. The activity is specific to orthopedic surgery.

There are two different methods for physical therapy. The first method requires a patient to lie on a special table with a stress motor attached at one end. A therapist performs hot-cold and stress therapies. These occur in a massage therapy room. The second method uses special devices and equipment for treatment. The space which accommodates this activity can be called *active exercise room*. The area requirements depend on the selection of devices or equipment. Generally, for a one-physician practice, there is a 16 x 18 sqf. area requirement. The area can be increased by 1.5 times for each additional physician to accommodate the increased number of patients.



In this section, the activities of AHCFs and their relationships to medical specialties have been summarized. These are also represented in the following table Table 4.1. The required activities for each medical specialty are marked by an "X". As described before, some activities are required if certain conditions are met (such as the number of physicians must be equal to a certain value etc.). The conditional activities are marked by a "C" in the table. Conditional activities must be included in an architectural program when these conditions in the activity descriptions are satisfied. For example, in order to accommodate a detailed business activity (with insurance, bookkeeping, office managing etc.), the number of physicians practicing internal medicine in an AHCF should be equal to or more than three; else the program's business will be performed by only one or two staff members. This will be discussed in the following section (Section 4.4) in greater detail.

We also observe that activity descriptions are used in the initial phase of design requirements specification for deciding the needed spaces. In the following phase, required spaces are named and their spatial requirements are concluded. An activity's name is usually reflected in the name of the space where it takes place (mapping from activity names to space names).

## Activities in ambulatory health care facilities

	Primary Care			Specialized Medicine										
	General Practice	Internal Medicine	Pediatrics	Obstetrics and Gynecology	Otolaryngology	Ophthalmology	Dermatology	Plastic Surgery	Allergy	Psychiatry	Neurology	General Surgery	Urology	Orthopedic Surgery
Common Activities	Patient Examination	X	X	X	X	X	X	X	X	X	X	X	X	X
	Consultation	X	X	X	X	X	X	X	X	X	X	X	X	X
	Waiting	X	X	C	X	X	X	X	X	X	X	X	X	X
	Information Management	C	C	C	C	C	C	C	C	C	C	C	C	C
	Auxiliary	C	C	C	C	C	C	C	C	C	C	C	C	C
	Medical Storage	X	X	X	X	X	X	X	X		X	X	X	X
	Minor Surgery	C		C	C	C	C							
	Out-Patient Surgery		C		C			X				X		
	Medical Support	X	X	X	X	X			X		X	X		X
Semi-special Activities														
	Specimen Testing	C	C	C	X	C	X				X			
	Immobilization Treatment	X												X
	EKG	X	X											
	Radiology	C	C		X								C	X
Specialized Activities	Prostoscopy		X		X								X	
	Pulmonary Function Treatment		X											
	Hearing Treatment				X									
	Refraction Power Measurement					X								
	Electrolysis						X	X						
	Facials						X	X						
	Ultraviolet Therapy						X							
	Allergy Patient Interview								X					
	Allergy Shots								X					
	Allergy Diagnosis								X					
	Group Therapy									X				
	Encephalography										X			
	Electromyogram										X			
	Cysto Examination												X	
	Physical Therapy													X

**TABLE 4.1.** Activities and medical specialty (X represents required activities, C represents the activity is required if certain conditions are satisfied).

---

#### 4.4 Effect of a staffing pattern and patients on a program

A staffing pattern (as one of the main variables in shaping an architectural program) is highly coupled with the activities taking place in each medical specialty. The staffing pattern specifies the people and their organizational structure (who does what and who works for whom).

Characteristic staffing pattern of an AHCF contain the following professionals<sup>3</sup>:

**Physicians:** A physician is at the center of all medical activities. The core responsibilities of a physician are to examine a patient, diagnose causes of patient's complaints, consult with patients, prescribe treatments and possibly, administer them.

**Nurses:** Nurses are located at the second level of health care. A nurse is a staff member who makes initial examination of complaints. To a certain degree, nurses also make decisions about diagnosis and treatment of patients' complaints. They can also be specialized in a certain medical specialty like physicians. For example, nurse practitioners are practicing in obstetrics and gynecology. Before the examination, nurses take the patients vital signs (blood pressure, heart beat rate etc.)

**Technicians:** The operators of equipment and devices used in diagnosis and treatment are called technicians. While some of the technicians can have direct contact with patients, some work without any contact. For example, lab technicians usually don't need to interact with patients, whereas the technicians who run ultra-sound instrument directly communicate with patients. Nurses or physicians instruct the technicians for the treatment or diagnostic procedures.

**Medical assistants:** They help nurses and physicians in preparing patients for examination or treatment. In addition, they take the patients' file from archive, call the patients from waiting room and help them to move in the medical facility. Preliminary examination or screening of the patients' vital signs also can be their responsibility.

**Office personnel:** This is the staff that works in the business office of an AHCF. They are responsible for managing patient information, and office-insurance, office-patient, and office-business relationships. They also place purchase-orders for necessary supply or equipment to be used in the facility. The staffing pattern for a business office in a medium-volume AHCF includes an office manager, an insurance clerk, a bookkeeper, and a medical record keeper. Most of the AHCF hire cleaning companies to clean the medical facility since medical and bio-hazard materials have to be handled by authorized specialists.

The staffing pattern for an AHCF affects the decisions about general spatial requirements (such as number of spaces or spatial areas). This can be explained by the maxim that *an activity is performed by staff and staff is accommodated in a space* Figure 4.2. There are

---

3. AHCF's staff and their job descriptions can be found in (Pierce, 1997) in greater detail. We only included basic types of staff and their characteristics in this section.

basic standards or guidelines in deciding these requirements (Malkin, 1989). For example, the number of physicians determines how many consultation rooms are needed. Each physician consults with his or her patients and to prevent overloading of one space, each physician is assigned to one consultation room. This room can also be used as the private office by the physician. As the example states, the number of consultation rooms and the number of exam rooms depend on the number of physicians.

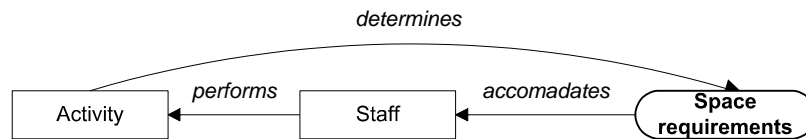


FIGURE 4.2. Activity-staff-space interaction

A physician's patient-examination schedule can be designed efficiently if he or she is assisted by a certain number of nurses. The number of nurses depends on the medical specialty and the type of the AHCF (single-specialty or multi-specialty). For example, in general medicine, a nurse prepares a patient for examination in one room while a physician is examining another patient in another room. For an efficient work flow, a physician practicing in general medicine must be assisted by two nurses. Therefore, two examination rooms must be assigned per physician. If we generalize this example, we may state that a type of staff is assisted by a certain number of another type of staff (*staff requires staff*), and this is reflected in the staffing pattern; in turn, this relation affects the spatial requirements Figure 4.3.

The type and number of *assisting-staff* change for different specialties. For example, a psychiatrist does not need an assisting-staff for patient's examination. However, an office assistant who keeps records and schedules is needed in an office with two psychiatrists. An obstetrics gynecologist, on the other hand, requires a nurse in the exam room during the examination of patients. In an obstetrics gynecology practice, for each physician, there must be two nurses and one medical assistant in the staff. In the first example, two psychiatrist who are practicing in the same office are assisted by an office personnel. In the second example, an obstetrics gynecologist requires three more medical personnels; two nurses and one medical assistant. The number of a specific type of staff (such as physicians) can be an independent variable, which (with other variables) is used to make decisions about other staffing requirements Figure 4.3.

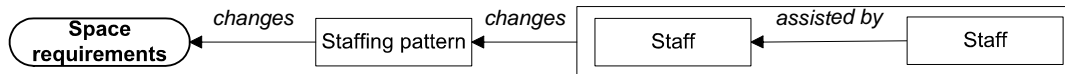
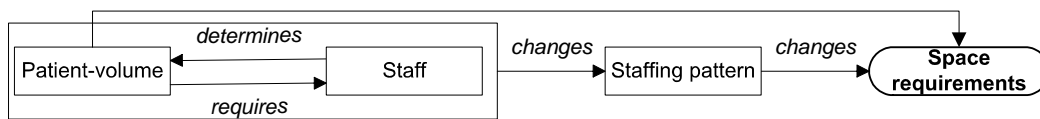


FIGURE 4.3. Staff-requires-staff relation changes staffing pattern and spatial requirements.

**Patients** are directly or indirectly served by medical office staff. In design requirement specification of AHCFs, patient-related variables (such as the number of patients, or num-

bers and types of complaints) affect the decisions about the needed staffing pattern, and consequently the spatial requirements. The overall volume of a medical office is determined by considering the number of patients who visit the office in a unit-time. In general, this type of variable depends on the *number of physicians* and the *medical specialty*. For example, a physician who is specialized in family (general) practice can see four to six patients per hour, while a psychiatrist can see one to two patients in an hour. In order to accommodate more patient, the number of physicians must be increased, therefore additional supporting staff members also will be needed. This relationship can be explained by the maxim that *staffing pattern determines patient-volume* Figure 4.4.



**FIGURE 4.4.** Patient-volume is determined by staffing, and staffing pattern is affected by patient volume; the interaction affects the spatial requirements.

Depending on the architectural programming approach, the number of patients can be either a dependent or an independent variable. In other words, we can start programming with a proposed number of patient to be examined in a unit-time and decide the number of staff and staffing pattern of an AHCF; or with a pre-determined staffing pattern and conclude how many patients can be served in a unit-time (say per hour). Therefore, interaction between patient-volume and staffing pattern can be bidirectional. Programming can start from either end. For example, if an AHCF is proposed to have a certain number of physicians with a certain specialty, the patient volume is limited with the staffing pattern i.e. *staffing pattern determines the patient volume and spatial requirements, and spatial requirements can accommodate a certain patient-volume*. If the patient-volume (demand) is high, the solution would be increasing the number of physicians, nurses, technicians or office personnel. Consequently these increases affect required number of exam rooms and other spatial requirements. This can be stated with the maxim that *patient-volume determines staffing pattern and spatial requirements, and staffing pattern determines spatial requirements*.

In most of the cases, a patient is accompanied by another person. Therefore, while calculating spatial requirements for general-use spaces, this also must be considered. For example, in order to calculate the area requirement for a waiting space, the required area for one patient must be multiplied by two. For other spaces where the patient's accompany is admitted, area and furniture requirements must take this factor into consideration.

As a summary, in determining spatial requirements of an AHCF (the number, size and types of spaces) basically four variables play an important role: *medical specialty*, *activity-composition*, *staffing pattern*, and *patient-volume*. Activities are the initial variables which determine both the required staff who performs the activity, and consequently, spatial requirements. The composition of the activities depends on the specialty of the medical practice. Staffing pattern is determined after the evaluation of activity composition, required staff, and the patient-volume. At any point, a change in the patient-volume even-

tually may effect the decisions about the current and the projected staff requirements. However, as a managerial decision, a fixed patient-volume can also be targeted.

#### 4.5 Effects of furniture, and medical instruments, equipment, tools on the program.

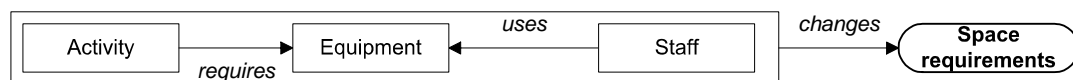
Furniture, instruments, devices, equipment or tools which are required for office-use, diagnostics and treatment purposes in an AHCF are accommodated in three typical situations. In these situations, they could be:

- being operated
- idle and waiting to be operated or maintained
- stored and not used.

The first situation involves an equipment actively being used during an activity. In the second situation, an equipment is ready to be used but the current activity doesn't require it. In the third situation, the equipment is stored and it is not ready to be used for an activity. Each situation is considered in making decisions about the spatial requirements. Each of these requires three basic types of spatial requirements: storage area, area to operate devices and equipment, area required when they are not used and not in the storage.

Take, for example, a medium-size X-ray machine; it is mounted on a ceiling, and when it is operated, an area with a certain radius has to be clear from obstacles. Since it is fixed in a room, the same room becomes the machine's storage area. Due to the dangerous effects of the X-rays, the machine has to be located in a room which is isolated from other spaces by special treatment of surrounding walls, ceiling, and floor. Other instruments such as refraction measuring tools require a space to have a specific layout. Operating refraction measurement devices requires a certain level of lighting and a minimum depth of space. The spatial requirements depend on two different specifications of equipment and devices: the use (the area required to operate them) and body-size (the area required to store them). These can be explained by the maxim that *an activity requires equipment, device, tools, and furniture and these are used by staff; and spatial requirements are affected by these relations* Figure 4.5.

Each equipment, device or furniture can be selected from different sizes. These could be due to the manufacturers' specifications or the usage-purpose. In addition, in parallel to the changes in technology, the size-related specifications of the devices change. An equipment's changing features can indirectly affect the spatial requirements.



**FIGURE 4.5.**An activity requires equipment and it is used by staff; this changes spatial requirements

A mobile equipment is stored in a room. The design requirements due to transferring a mobile-equipment from one location to another are considered in programming stage.

These affect the room sizes, opening dimensions, accessibilities, and adjacency relationships of spaces.

In this study, design requirement specifications related to instruments, equipment and devices are incorporated into activity-related requirements. Therefore, a change in an activity description and relationships to other activities may cause the inclusion or exclusion of furniture, equipment, devices etc. This indirectly affects the design requirements in an architectural program.

The effects of activities, specialty, staffing pattern and equipment on the architectural program of an AHCF can be brought together in a complete schema as shown in Figure 4.6. The schema is represented by using Unified Modeling Language (UML) notations (for details refer to Booch, G, J. Rumbaugh and I. Jacobson., 1999).

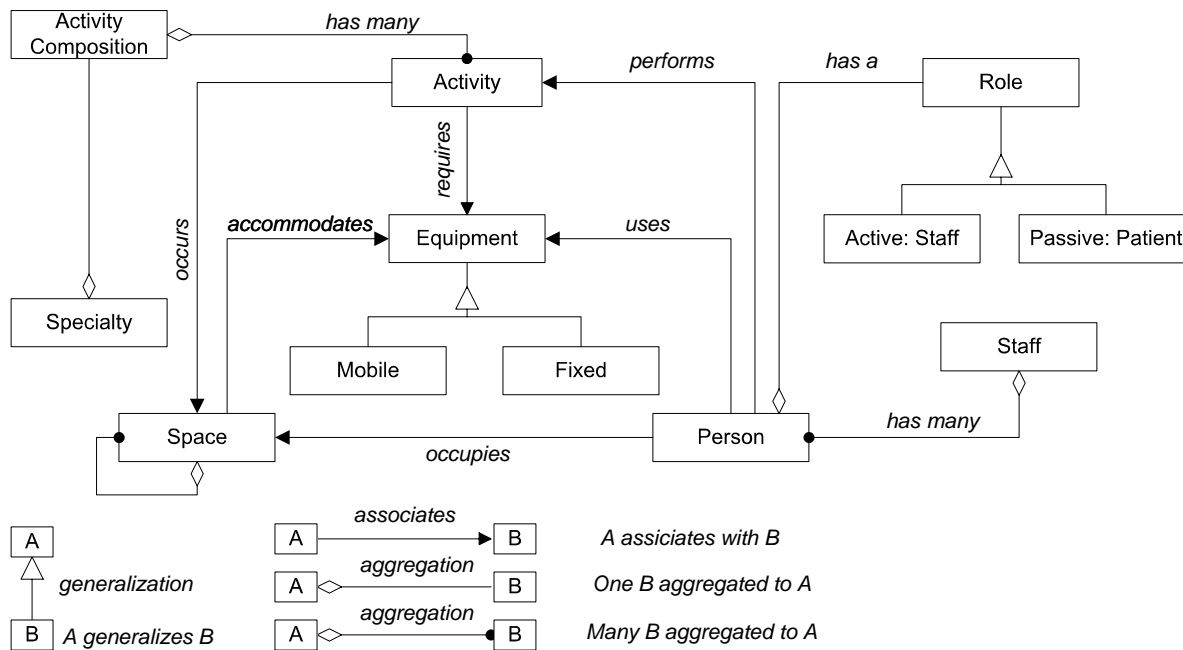


FIGURE 4.6. Complete schema for program elements.

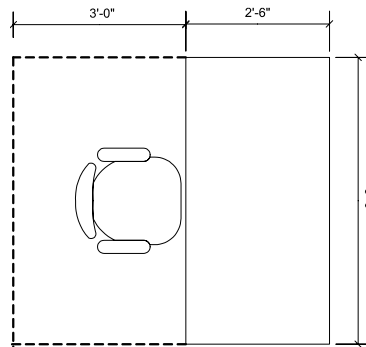
In the schema, equipment is divided into two subclasses: mobile and fixed. Each equipment (device, instrument, furniture etc.) is accommodated in a space and is used by staff. Each activity occurs in a certain space, and each space may consists of many other spaces (for example, business office consists of insurance office, manager's office, record keeping etc. spaces). The required activities in an AHCF are composed in an activity composition. The activity composition is associated with a specialty. The people who occupy an AHCF can have two different roles: active (doctors, nurses etc.) and passive (patients and their companions). The combination of staff members makes the AHCF's staff. People perform activities, which require equipment in a certain space.



#### 4.6 Spatial area requirements calculation methods:

Each space in an AHCF is known by an activity-related name as listed in the Table 4.1. The initial decisions about the size of these spaces can be made by using three basic methods. Generally these methods are based on activity-related performance requirements, but in decision making process each follows different techniques.

In the first method, which can be called *accumulated-area calculation*, the required area for equipment, furniture and devices and their operations (considering ergonomics and anthropometric data) are added up. For instance, as demonstrated in Figure 4.7, the area that a study desk occupies is, let's say, 13 sqf., and for a chair it is 4 sqf. The total area for a study desk setting is going to be 17 sqf. plus a 15 sqf. area required for the operation (a person sits in front of the desk and uses drawers, moves along side of the desk on a chair etc.). The total area requirement would be 32 sqf. However, the area required for the chair overlaps with the area of operation, and the area that the chair occupies must be subtracted from the 32 sqf. total area. We conclude that the total area requirement for the study desk is going to be 28 sqf. As the example demonstrates, this method is analytical, and each variable affecting the spatial requirements is considered individually. By analyzing the operation and operation-related furniture (equipment), we assign areas for each component. The length and width of the required area are also determined during adding each of these areas to the total. In the process, overlapping areas are removed from the total area.



**FIGURE 4.7.**Analysis of spatial area requirements of a study desk setting

In the second method, values for different variables, which are related to spatial requirements, are picked from well-established sources. In the literature, there are studies (sources) that aim at standardizing spatial requirements for AHCFs programming. These studies incorporate occupant, activity, and related equipment and furniture factors into the data they provide. In addition, general information that can be applied to specific cases is presented in these studies. Area requirements of different spaces are prescribed in different formats. The data provided in these studies are based on personal experiences (Kobus, 1997), surveys (MCMA, 2001), analytical techniques (similar to the first method) (Malkin, 1989) or well-established standards (BCHS, 1974-1 and 1974 - 2)(Chiara and Callender, 1990). The second method is generally applied for space planning purposes.

Even though data provided in these sources can be applied to most of the cases, they have to be revised and modified as needed.

Type of practice	Square feet per physician	# of exam rooms per physician
Multi specialty	1497	2.12
Cardiology	1201	1.33
Family practice	1565	3.10
Internal medicine	1500	Not available
OB/Gyn	1653	2.78
Ophthalmology	1527	2.53
Orthopedic surgery	1843	2.32

**TABLE 4.2.** MGMA (1999) conducted an informal survey of group practices and their space planning. The averages for square footage and number of exam/patient treatment rooms are listed in the table

The third method is based on pre-determined and tested formulas. In order to calculate an area, variables which contribute to the generation of a spatial requirement are incorporated into a set of formulas. By assigning values to each variable, the required area can be calculated. For example, in order to calculate area requirements for a waiting space in an AHCF, first the number of required seats in the waiting room is calculated. Following this, the number of seats is multiplied by a *unit area per person* constant (Malkin, 1989, pg. 27). The formula to calculate the number of seats is:

$$\text{Number of required seats} = (3 \times P \times D) - E$$

where: P represents the average number of patients that a physician sees in an hour; D is the number of physicians and E is the number of exam rooms, which differs from one specialty to another. For example, for general medicine there must be three exam rooms for each physician, and for allergy specialty, this variable is equal to two. Also, the *number of exam rooms* can be stated in terms of the *number of physicians*. However, in some cases where there are budget limitations or un-expected spatial constraints, the number of exam rooms could be more independent of the number of physicians. The area required for seats is calculated by the following formula:

$$\text{Area} = \text{Number of seats} \times \text{Area per seat.}$$

The area-per-seat coefficient can be determined by using either the first method or the second one. If the second method is chosen, the area requirement-per-seat is listed as 15 to 20 sqf per seat in (Malkin, 1989). In order to calculate the area for a waiting space in an AHCF with one physician, we initially apply the first formula to find required number of seats;

$$\text{Number of seats} = (3 \times 4 \times 1) - 3 = 9$$

and the result (9) is multiplied by the selected value for required area per seat.

$$\text{Area requirement for waiting room} = 9 \times 18 = 162 \text{ sqf.}$$

These three methods can be employed individually or combined. This depends on the needed data for design requirement specification. There are times when these methods become ineffective, because they may not cover some variables which play an important role in spatial requirements generation. It also is a trade off between staffing pattern, activities, budget, and site- and regulation-related limitations which are almost difficult to integrate into these general information sources or formulas. In addition, in determining initial requirements, rather than accepting one specific value, a range of acceptable values is targeted. When these trade-offs are evaluated, the final selected area can be chosen from this targeted range.

The described methods in this section exclusively cover area calculation techniques. There may be some other custom-built information sources by different programmers which are not mentioned in this study.

---

#### **4.7 Synthesis of components in a framework**

In the previous sections, we demonstrated how a medical specialty affects an activity composition; how we derive required spaces and space names from activity descriptions; how generally a staffing pattern influences an architectural program; and how equipment (instruments, devices, tools) affect the spatial requirements. In addition, three basic methods for calculating area requirements are explained. However, the missing part in this picture is how these different components can be combined into a complete framework. The framework must delineate the interaction between the components and it must contribute to representing design requirements for AHCF.

##### **4.7.1 Framework components and constructs.**

The framework represented in this section is composed of two basic groups: components and constructs.

By components, we mean the medical specialties, activities, and spaces that are described in the previous sections. The spaces are spatial-volumes which accommodate activities. A medical specialty, on the other hand, plays an important role in activity composition, and the activities are reflected in the architectural program in the form of spaces.

The second group consists of the constructs which are variables, constants, formulas, procedures, and conditional statements. They are defined in this study as follows:

**Variables:** These are the pointers to values with different formats, which can be numerical, logical (true or false) or user-defined types. They are used in formulas and procedures. Their values are manipulated through formulas and procedures. An example of a variable is the *number of physicians* which holds a numerical value (integer) representing the number of physicians that work at a given time in an AHCF. In the framework, these variables can be independent from or dependent on a component (space, activity or specialty). For example, the *number of physicians* is a variable which doesn't depend on either spatial requirements or specialty, whereas *number of patient that a physician sees in an hour* is a variable that changes from one specialty to another. Therefore, the second variable depends on the medical specialty. The user-defined variables are used to classify or group selected components. For example, medical specialty itself is a user

defined variable which defines the medical practice. Similarly, user-defined variables can determine if a space is private, public or semi-public.

**Constants and coefficients:** These are fixed values (data) that can be substituted in other constructs mentioned in this study. For example, the area required to accommodate a specific equipment or the minimum number of required toilet stalls is a constant. As another example, the exam room coefficient is a value that is used to decide the number of exam rooms for different specialties.

**Formulas:** These are statements which connect a set of variables and constants with each other so that if a change in one of the variables is made, there will be a consequent change in at least one of the other remaining variables. The formulas are expressed as equations. For example, *number of exam rooms* variable is connected to the *number of physicians* and *exam room coefficient for a medical specialty*. The formula for this connection can be represented as:

$$[\text{\# of exam rooms}] = \text{\# of physicians} \times \text{\# of exam rooms per physician}$$

*Number of exam rooms per physician* can be taken from Table 2. The value for this variable is different for each specialty.

**Conditional statements:** As the name implies, these constructs represent logical relationships between variables. For example, to state if the *number of physicians* (D) is greater than 3 then a detailed business office (Bd) is needed. The first variable (D) is a numerical variable and the second (Bd) is a logical (binary) variable. The conditional statement can be state as:

```
if D >= 3 then Bd = true
else Bd = false
```

**Procedures:** These constructs are used to encapsulate one or many formulas and logical statements in a package that manipulates multiple variables at a time. For example, to calculate the *number of toilet stalls* we use the following formula:

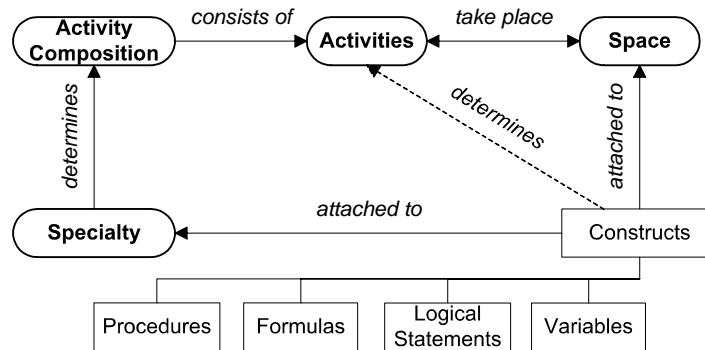
$$TS = Fx + (D \times Tc)$$

where: TS is the number of toilet stalls; Fx is the minimum number of toilet stalls; D is the number of physicians and Tc is a specialty-dependent coefficient. In order to determine the required number of toilet stalls for a urology specialty, the function can be stated as:

```
Calculate_number_of_toilet_stalls_for_urology (D <number_of_physicians>) {
  if D = 1 then
    Fx = 2; Tc = 0
    TS = 2 + (1x0) = 2
  else
    Fx = 1; Tc=1
    TS = 1 + (Dx1)
}
```

If needed, in one procedure, other procedures also can be invoked.

The constructs can be either attached to a component (Figure 4.8) or they can stand alone affecting other constructs. For example, the *number of physicians* does not depend on any component (i.e. specialty, activity, or space) whereas the *number of patient that a physician sees in an hour* is a variable that changes from one specialty to another.



**FIGURE 4.8.**Components and constructs

#### 4.7.2 The number of physicians: The main independent variable for AHCF

We stated that each specialty contains different activity compositions and the activities in a composition take place in spaces. Some of the activities are added to or removed from the composition if certain conditions are satisfied. These conditions can be based on many factors such as budget, schedule, staffing pattern, patient volume etc. However, for AHCF programming, the general tendency is to take the number of physicians as the main variable. This variable is also used to arrive gradually at spatial requirements. For example, the number of physicians in an AHCF is used for determining the number of exam rooms or calculating the area for the waiting room. The number of physicians is taken as an independent variable and incorporated into the required conditions by which the entire decision making process is influenced.

#### 4.7.3 Variables attached to specialty components.

The following variables and constants are common across specialties. Each of these is assigned to different values for different specialties. In the following section, we will describe these common variables and constants. The use of each of them is included in Appendix A.

**Exam room coefficient (Ec):** The constant (or coefficient) depends on the medical specialty and determines how many exam rooms an AHCF with a specific specialty requires, if any. The number of exam rooms is determined by using this coefficient in a formula in which the value is multiplied by the *number of physicians who work in the facility during an average day*. In the framework represented in Appendix C, the formula is attached to the space (component) which has *examination room* tag. If the result of the multiplication is not an integer, it is rounded to the next integer value.

**Consultation room coefficient (Cc):** Like the exam-room coefficient, this is used to determine the number of consultation rooms required for a particular medical specialty in an AHCF. The number of consultation rooms is determined by multiplying this coefficient with the *number of physicians practicing in an AHCF*. The multiplication formula is attached to the space (component) which has the *consultation room* tag.

**Business office detail (Bd):** This is a binary variable determining if a specialty requires a business office with insurance, bookkeeping office, manager office and media record store. The binary value is determined by considering the *number of physicians*. For example, if the *number of physicians* is greater than 2 and the specialty is general practice, then the variable value is assigned to *true*, else the value for this variable, as default, is *false*. This is encapsulated in a conditional statement and attached to each individual space listed under business office. If the value is true for a specific business office component, then the space is added to the program.

**Toilet requirement variables (TL (Fx, Tc)):** These are a pair of variables which are used to calculate the number of toilet stalls. The first one of the pair (Fx) is the *minimum-number of toilet stalls* and this is a constant. The second variable (Tc) determines the variable number of toilet stalls. The *number of toilet stalls* is used to calculate area requirement of a rest room. It is also incorporated into a formula which is used to calculate the number of rest rooms. The formulas used for a rest room's area and the number of rest rooms in a facility are attached to the spaces (components) tagged *rest room* in the framework.

**Laboratory variables (LB (La, Ld)):** An AHCF may need a lab if (a) a specialty requires, and (b) the conditions for lab are satisfied. If a lab is required, then conditions for adding the supporting spaces (waiting area, blood drive booth etc.) to the program are checked. Two variables, as a pair, are attached to a component tagged laboratory space, and evaluated together. The first variable of the pair, *lab needed*, is a binary variable and it represents if the lab is required by a medical specialty. The second variable, *supporting lab spaces required* is also a binary variable and used for deciding if supporting spaces will be needed. The evaluation continues in the constructs which are attached to lab-supporting spaces (constructs).

**Staff lounge variable (SI):** The decision to add a staff lounge to a program is incorporated into this variable, and the variable is attached to medical-specialty components. It is a binary variable and used for deciding if a lounge is needed in an AHCF. The other variable effecting this decision is the *number of physicians*. Together, these variables are contained in a logical statement which is attached to a space-component named *staff lounge*.

**Minor surgery room variable (Ms):** This is a binary variable which depends on the specialty. It is used for determining whether or not a minor surgery room must be included in the program. Depending on the number of physicians, adding

a minor surgery room to the program is decided if the variable's value is *true*. For each specialty, there is a certain threshold value for the number of physicians in order to add a minor surgery room to the program.

**Nurse station variable (Ns):** This is similar to the previous two variables, and it is used to determine if a *nurse station* is needed. The variable is encapsulated in the space-component named *nurse station* and evaluated in a logical statement.

**Number of patients seen in an hour (P):** This is an integer variable and determines how many patients can be seen by a physician in an hour. Due to the nature of the specialties and the medical procedures, each specialty has a unique value for this variable. This variable is evaluated in procedures attached to the different spaces.

#### 4.7.4 Sample specialty with attached constructs

In order to demonstrate attachment of variable-constructs to specialty-components in the framework, *pediatrics specialty* can be used as an example (Figure 4.9).

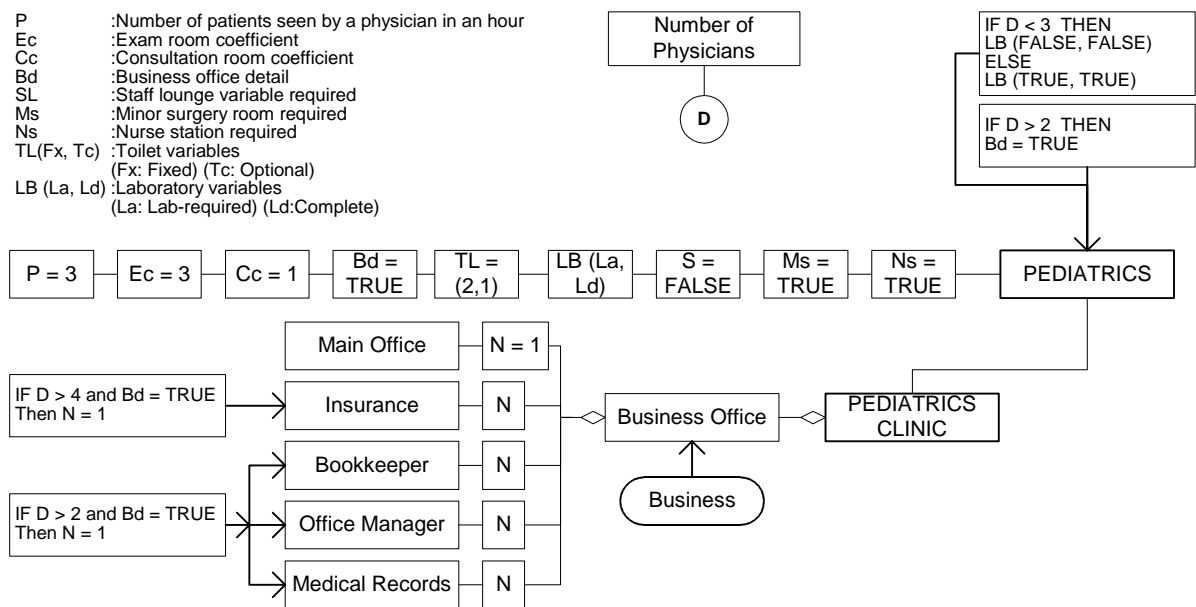


FIGURE 4.9. Pediatrics specialty and attached constructs.

According to this diagram, the attached variable-constructs and their respective values are as follows:

The *number of patients* seen by a physician in an hour (P) is equal to 3. The variable can be modified by the programmer if required. This will affect the area calculation for waiting spaces.

The *exam room coefficient* ( $E_c$ ) value is 3. That means, for each physician, 3 exam rooms must be included in the program.

The *consultation room coefficient* ( $C_c$ ) is equal to 1. That is, for each physician one consultation room is required.

The default value for *business office detail* ( $B_d$ ) variable is *false*. However, if the number of physicians is greater than 3 then the value becomes true. This change is decided through the function which includes a conditional statement attached to the business office space-component.

The *toilet variables* ( $T_L$  ( $F_x$ ,  $T_c$ )) indicate that there must be at least two toilet stalls required for this specialty ( $F_x$  is equal to 2) and the number of additional toilet stalls ( $T_c$ ) is equal to zero. However, the programmer can adjust these values to increase the number of toilets if preferred.

The *laboratory variables* ( $L_B$  ( $L_a$ ,  $L_d$ )) are determined by a function which is attached to each specialty component. This function evaluates the value of the number of physicians, and then assigns values to these pair variables.

The *staff lounge variable* ( $S_l$ ) is *false*. That means a staff lounge is not needed in this specific case. However it is up to budget and the client to decide if a staff lounge should be added to the program.

The *minor surgery room variable* ( $M_s$ ) indicates that at least one minor surgery room is required for this specialty.

The *nurse station variable* ( $N_s$ ), similar to the minor surgery, is required for this specialty.

A complete variable-construct assignments to each specialty-component is described in a diagram and it is presented in Appendix C.

#### **4.7.5 The sample space component with attached constructs**

We also included space components in the framework. In Figure 4.9, spatial components and their attributes are included as well. The complete framework is attached as Appendix C. The figure represents business office space-components and their attached constructs. However, it should be noted that even though the business office is represented as connected to pediatric specialty, it could be attached to any other specialty, and its representation will not change.

The main variable construct attached to each space component is the number of spaces ( $N$ ), which is assigned to an integer. Each of the spaces has one or more attached function constructs that evaluate the conditions. These function-constructs are also used for assigning values to the variable  $N$ . If the conditions are satisfied, the particular supporting space is aggregated to the business office space-component. For example, to have a main office in the program, the only condition is to have at least one physician. On the other hand, in order to add an office manager space-component, the *business office detail* variable-construct ( $B_d$ ), which is attached to the specialty-component, must be *true* and the number of physicians has to be equal to or greater than 3. If and only if these conditions are satisfied, the space-component for manager office is attached to the main business office. In other words, the constituent spaces of the business office are selected after the evaluation of the



conditional statements. The programmer can adjust the values and conditions if required for each specific case.

Other variable-constructs such as area, dimensional sizes, list of equipment can also be attached to each space-component as required. The procedure or formula constructs can be modified to accommodate new variables.

A complete diagram showing the spaces, activities and their composition with attached constructs can be found in Appendix C.

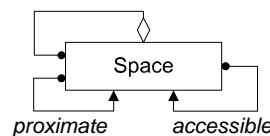
#### 4.8 Activity affinities and their effects on spatial relationships

There is another important issue that we need to investigate: the affinities of activities and their effects on the spatial relation requirements (such as adjacencies or proximity). We tackled this issue when we discussed the designation of spaces for each activity and their spatial configurations in terms of constituencies. For example, some spaces such as business office or radiology unit consist of other spaces which are related to these activities.

The affinities between activities and their compositions require respective spaces to relate to each others in three ways. These relationships may be listed as following:

- access from one space to another (physical or visual)
- allocating one space adjacent to another space
- maintaining a certain physical distances between two or more spaces.

The first relationship is called accessibility. The other two can be described as proximity relationship. In Figure 4.6, integration of spaces into a schema is represented. The schema also describes how and what types of components interact with each other. After the evaluation of proximity and accessibilities, the space component in this schema can be altered as shown in Figure 4.10.



**FIGURE 4.10.** Updating the schema by considering proximity and accessibilities between spaces.

In an AHCF, these relationships can be determined after analyzing the activity affinities. The analysis can be based on interaction between occupants and spaces. It also can be described by different scenarios and represented by different methods such as bubble diagrams or affinity matrixes. However, we prefer to use Unified Modeling Language (UML) interaction diagrams (Booch, G., J. Rumbaugh and I. Jacobson., 1999) as the representation technique. The main reason for this is that we not only need to represent spaces but also occupants. Bubble diagrams and affinity matrices can only be used to show static and space-to-space relationships. By using UML notations, we can incorporate other factors, such as occupants, equipment etc. into the activity scenarios. In order to illustrate how this

could be achieved, let's analyze the scenario of a patient entering an AHCF for examination. The initial activity sequence for this can be described with the following steps:

**Pre-conditions:** The patient made an appointment earlier. The record staff removed the patient's file from the archive and physician reviewed the file. The receptionist receives the file and places on the daily file stack.

- a patient comes to the AHCF, enters to the waiting area
- the patient signs in the sing-in form placed on the reception counter (desk)
- the patient takes a seat in the waiting area
- the receptionist controls the sign-in page
- the receptionist communicates with the patient and completes the registration
- the patient waits to be called
- the receptionist removes the file from the file stack and places it on the desk for the nurse
- the nurse picks up the file and goes to waiting area and calls the patient
- the nurse escorts the patient to the nurse station
- the nurse examines the patient's vital signs.
- the nurse escorts the patient to an exam room and places the file in a rack attached to or by the door of the exam room.
- the patient waits for the physician.

This interaction graphically can be represented by a UML interaction diagram as shown in Figure 4.11.

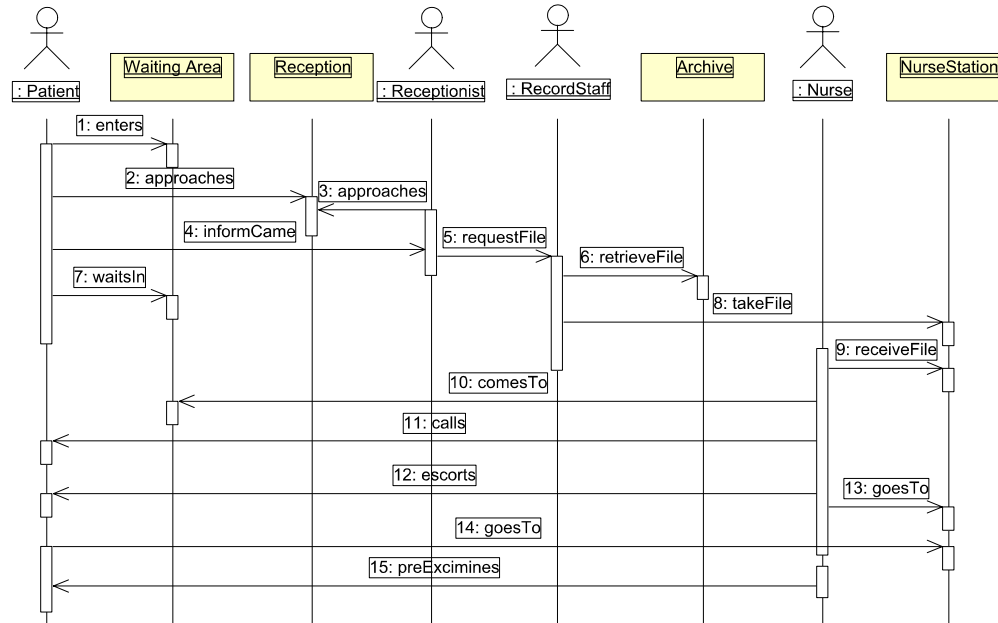


FIGURE 4.11. Activity affinity analysis by UML interaction diagram

From the scenario and the diagram, we can infer certain relationships between spaces. A physical and visual access from the waiting area to reception is required. The degree (nature) of accessibility is up to the programmer or the designer. There is a close proximity requirement between the waiting area and the nurse station. However, in between them the reception area becomes a control point. The nurse and patient access from the nurse station to an exam room. This requires a proximity and accessibility requirement from nurse station to the exam room. The accessibility requirement can depends on the programmer's choice; it can be connected via a circulation path or via a direct access (for example a door if the exam rooms surrounds the nurse station). The relationships can be further detailed as the programming process continues.

As a side-effect of this analysis, other spatial requirements can be discovered. For example, up to this point, we have not mentioned circulation or entrance activities, because these are generic activities for every facility. However, depending on the context, they can be defined differently. Therefore, their spatial requirements change in each context. From the sample analysis, we may decide to have a connection space as a corridor from the waiting room to the nurse station. This also creates a relation (constraint) of proximity. The minimum width of the corridor is determined by the standards defined in ADA. In addition we can also decide the furniture or equipment requirements. Such as the desk for the receptionist or a receptionist counter are mentined in the scenario.

Similar to this decision-making process, other activity affinities (as higher-level design requirements) can be resolved and can yield lower-level design requirements in the form of spatial relations. This could be managed by describing scenarios and representing them

with sequence list and interaction diagrams which not only incorporate spaces but also other factors into the analysis.

#### **4.9 Summary**

In this case study, we have investigated architectural programming and design requirement specification of an AHCF. As a contrast to the conventional methods, we tried to concentrate on gradual transition from higher-level design requirements to lower-level design requirements. In the process, AHCFs are divided into specialty-related classes. Each class is described with its unique activity compositions (including common activities). These activity compositions help us determine required spaces for each activity. Following this, spaces are described and located in a spatial composition for each medical specialty (Appendix C). In connection with the space descriptions, methods for the computation of area requirements are described. Constructs such as variables, formulas, logical statements and procedures are used to detail spatial-composition diagrams. Finally, the effects of activity affinities over design requirements in the form of spatial relations are discussed.

The case study initially suggests that the process of generating design requirements of AHCF can be simplified if:

1. For each specialty, there is a pre-existing activity selection mechanism which is based on independent variables such as the number of physicians.
2. Pre-determined spatial designations and their attached constructs are available and connected to each activity.
3. There is a mechanism which transfers activity affinities to spatial relationships. Also this mechanism generates additional spaces if required.

As mentioned earlier, this process (with different methods and techniques) is repeated each time a new AHCF is needed to be programmed. There is a considerable potential for employment of a computational tool. This is implicitly demonstrated by the appropriateness of software engineering representation techniques and analysis methods (such as UML for representing the static schema and interaction diagrams). Each of the variables of an AHCF program (such as specialty, activities, activity composition, spaces, occupants, constructs and components) can be modeled by using Object Oriented Modeling methods, and eventually they can be programmed with a software language. This way, the propagation from higher-level design requirements to lower-level spatial design requirements can be managed seamlessly.

---

## *Bibliography*

- Akin, Ö., Sen, R., Donia, M. and Zhang, Y., 1995. SEED-Pro: Computer assisted architectural programming in SEED, in *Journal of Architectural Engineering*, ASCE, 1(4): 153-161.
- AMGA, 2001. The American Medical Group Association. <http://www.amga.org/>
- AR 140-483. 1994. Army Reserve Land and Facilities Management, Space Guidelines for U.S. Army Reserve Facilities, Army Publications and Printing Command. (also available at [http://books.usapa.belvoir.army.mil/cgi-bin/bookmgr/BOOKS/R140\\_483/CCONTENTS](http://books.usapa.belvoir.army.mil/cgi-bin/bookmgr/BOOKS/R140_483/CCONTENTS)).
- Army 1. 2001. <http://www.army.mil/usar/overview.htm>.
- BCHS. 1974 - 1. Equipment guidelines for ambulatory health centers. United States. Health Services Administration. Bureau of Community Health Services. DHEW publication ; no. (PHS) 79-50066.
- BCHS. 1974 - 2. Space guidelines for ambulatory health centers. United States. Health Services Administration. Bureau of Community Health Services. DHEW publication ; no. (PHS) 79-50066.
- Booch, G., J. Rumbaugh and I. Jacobson., 1999. *The Unified Modeling Language User Guide*. New York: Addison-Wesley.
- Cherry, E. 1998. *Programming for design: from theory to practice*. NY: John Wiley and Sons Inc.
- Chiara, J.D. and J. H. Callender, (Contributors). 1990. *Time-Saver Standards for Building Types*. New York, NY. McGraw Hill Text.
- Cross, N. 1996. *Analysing Design Activity* (N. Cross, H. Christiaans and K. Dorst; eds.), John Wiley and Sons Ltd., Chichester, UK.
- CDC and NCHS. 1998. National Hospital Ambulatory Healthcare Survey. [www.cdc.gov/nchs/about/major/ahcd/outpatientcharts.htm](http://www.cdc.gov/nchs/about/major/ahcd/outpatientcharts.htm)
- DG, 1984. Design Guide DG 1110-3-107, U.S. Army Reserve Facilities, Department of Army, Corps of Engineers.
- Duerk, D.P. 1993. *Architectural Programming: Information Management for Design*. NY. John Wiley & Sons, Inc.
- Farbstein, J. 1977. Assumptions in Environmental Programming. In Suedfeld, P. et. al. (eds.). *The Behavioral Basis of Design*, EDRA Proceedings, Stroudsburg, PA. Dowden, Hutchinson and Ross.

- 
- Farbstein, J. 1985. Using the Program, Applications for Design, Occupancy, and Evaluation. In Preiser, W.F.E. (ed.) 1985. *Programming the Built Environment*. New York: Van Nostrand Reinhold.
- Flemming, U. et al. 2000. *The SEED Experience*. Internal Report. Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA.
- GSA. 1983. *Design Programming*. PBS 3430.2. Washington, D.C.: General Services Administration
- Hershberger, R. G. 1985. *A Theoretical Foundation for Architectural Programming*, in *Programming the Built Environment* edited by Wolfgang F.E. Preiser, New York: Van Nostrand Reinhold.
- Hershberger, R. G. 1999. *Architectural programming and predesign manager*. NY: McGraw-Hill Co.
- Jackson, M. 1995. *Software requirements and specifications: a lexicon of practice, principles and prejudices*. New York, NY. Addison-Wesley Publication Co. and ACM Press.
- Kobus, R., R. L. Skaggs, M. Bobrow, J. Thomas and T.M. Payette. 1997. *Building Type Basics for Healthcare Facilities*. New York, NY. John Wiley and Sons Ltd.
- Kumlin, R.R. 1995. *Architectural programming: creative techniques for design professionals*. NY: McGraw-Hill Co.
- Markus, T. 1972. *Building Performance*. New York, NY. Halstead Press.
- Malkin, J. 1982. *The Design of Medical and Dental Facilities*. New York, NY. John Wiley and Sons Ltd.
- Malkin, J. 1989. *Medical and Dental Space Planning for the 1990s*. New York, NY. John Wiley and Sons Ltd.
- Medical Group Management Association (MGMA). 1999. *Medical Office Space Planning Survey*. [www.mgma.com/infocenter/faq-web.html#4](http://www.mgma.com/infocenter/faq-web.html#4)
- Middleton, S. 2001. *Training Decision Making in Organizations: Dealing with Uncertainty, Complexity, and Conflict*. Special Report. (<http://www.work-teams.unt.edu/reports/smiddltn.htm>).
- Palmer, M.A. 1981. *The Architect's Guide to Facility Programming*. Washington D.C. The American Institute of Architects.
- PBS-PQ100.1. 1996. *Facilities Standards for the Public Buildings Service, US government General Services Administration*. [www.gsa.gov/pbs/pc/tc\\_files/stds/pq100.pdf](http://www.gsa.gov/pbs/pc/tc_files/stds/pq100.pdf).
- Pena, W., W. Caudill and J. Focke. 1977. *Problem Seeking: An Architectural Programming Primer*. Boston, MA. Cahners Books International, Inc.
- Pena, W.M., S. Parshall, and K. Kelly 1987. *Problem Seeking: An Architectural Primer*. Washington, DC. American Institute of Architects Press.

- 
- Pena, W.M., and W.W. Caudill, 1959. Architectural Analysis: Prelude to Good Design. Architectural Record, May 1959. (pg. 178-182).
- Pena, W.M. and J. Focke. 1969. Problem Seeking. Houston. Caudill Rowlett Scott.
- Perkins, B. 2000. Building Types Basics for Elementary and Secondary Schools. John Wiley & Sons, Inc. New York, NY.
- Pierce, Courtney. 1997. Group Practice Personnel Policies Manual. Medical Group Management Associations, New York, NY.
- Preiser, W.F.E. (ed) 1985. Programming the Built Environment. New York: Van Nostrand Reinhold.
- Preiser, W. F. E. 1993. Professional Practice in Facility Programming. New York: Van Nostrand Reinhold.
- Preiser, W. F. E. (ed). 1978. Facility Programming: Methods and Applications. Stroudsburch, Pa.: Dowden, Hutchinson and Ross.
- Sims, W. 1978. "Programming Environments for Human Use: A look at some approaches to generating user oriented design requirements". in Rogers and Ittelson, New Directions in Environmental.
- Verger, M., N. Kaderland. 1993. Connective Planning. New York, NY. McGraw-Hill
- White, E.T. 1972. Introduction to Architectural Programming. Tucson, AZ. Architectural Media.
- WGPPF. 1992. Postsecondary Education Facilities Inventory and Classification Manual. Working Group on Postsecondary Physical Facilities. [bacweb.the-bac.edu/~michael.b.williams/](http://bacweb.the-bac.edu/~michael.b.williams/)
- Zimring, C., D.L Craig (2001). Defining Design Between Domains: An Argument for Design Research a la Carte (penultimate draft), in Design Knowing and Learning: Cognition in Design Education.

---

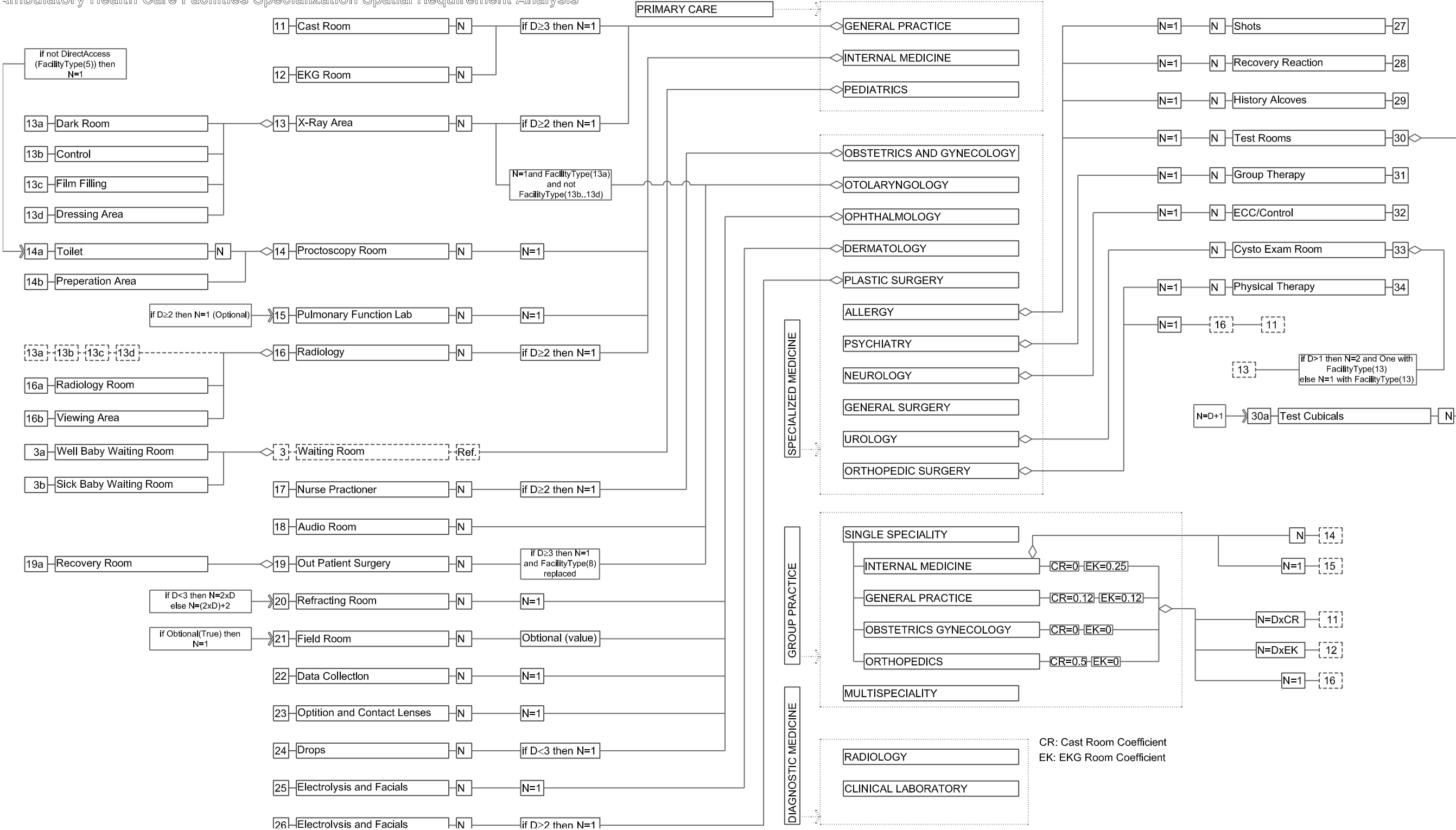


---

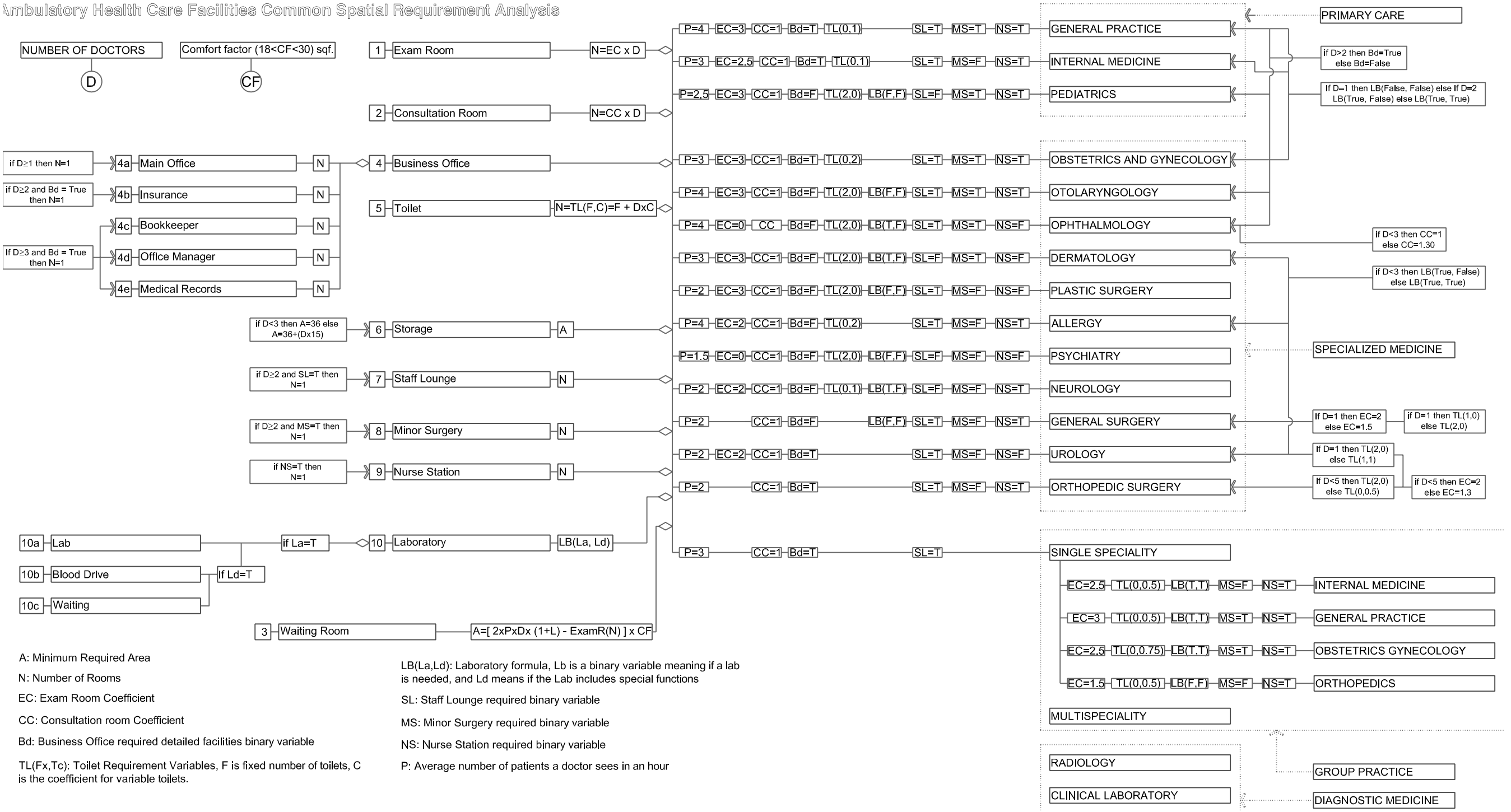
## *Appendix A: AHCFs Space (of Case Studies Assignments)*

- AHCFs common spatial requirements analysis
- AHCFs specialization specific spatial requirements analysis

# Ambulatory Health Care Facilities Specialization Spatial Requirement Analysis



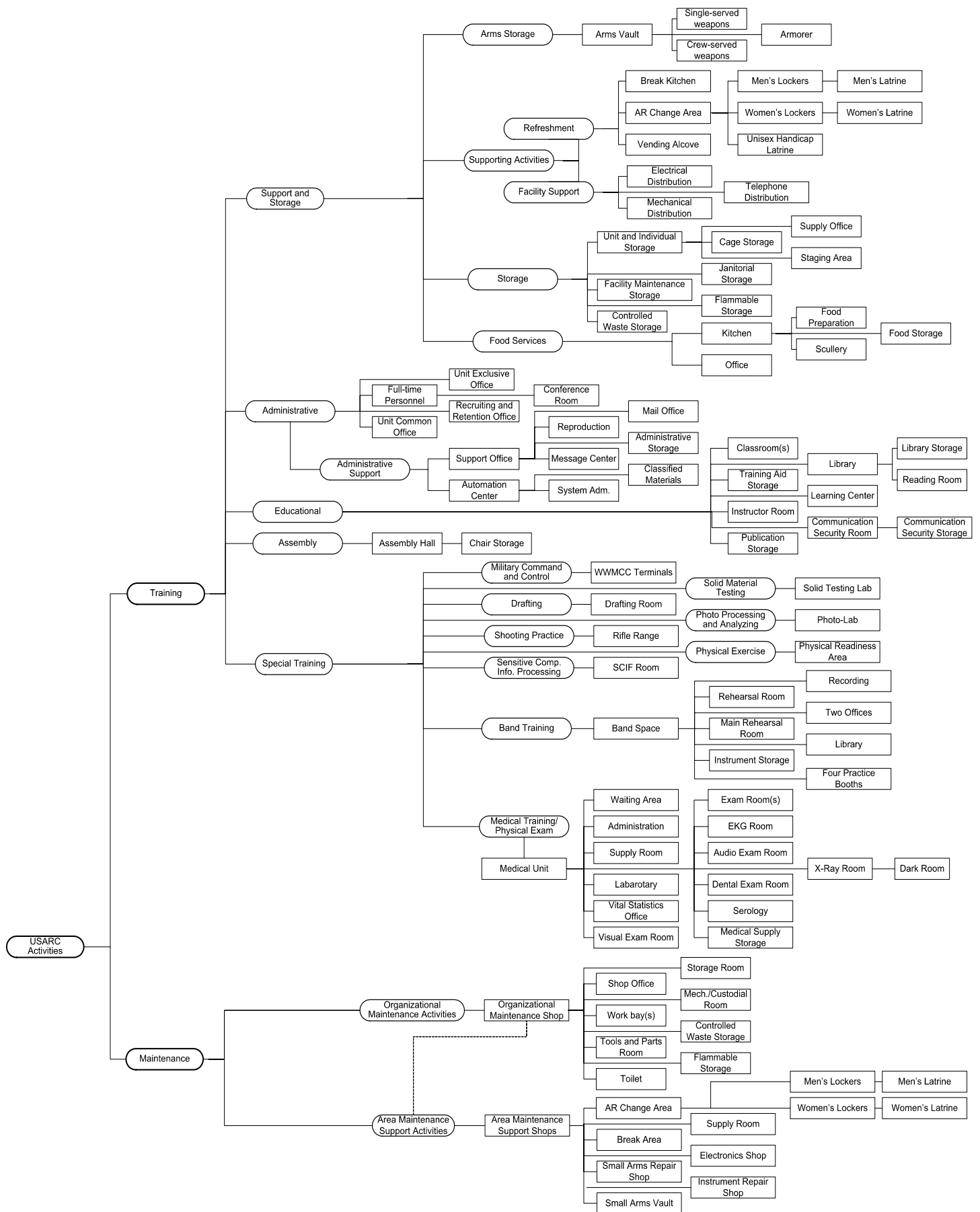
Ambulatory Health Care Facilities Common Spatial Requirement Analysis



---

## *Appendix A: USARCs Space Assignments (Case Studies)*

- USARC activities and spaces.
- USARC activity to space transition tree.
- USARC spatial requirements formulations.



---

## 1 TRAINING FACILITY AREA CALCULATION

Area Reduction Tolerance = - % 10

### 1.1 *Administrative Area*

Full time personnel

*Officers 120 sqf*

*Civilians 120 sqf*

*Enlisted 120 sqf*

### 1.2 *Unit Exclusive*

General Officers

*Major General 400 sqf*

*Brigadier General 300 sqf*

Commanders Exclusive

*Colonel 200 sqf*

*Lieutenant Colonel 150 sqf*

*Lower Ranks 150 sqf*

Battalion And Higher Units

Exclusive Space

*Deputy Commander 120 sqf*

*Executive Officer 120 sqf*

*Chief of Staff 120 sqf*

*Staff Section Chiefs 120 sqf*

*Command Sergeant Major 120 sqf*

Company and Lower Units

Exclusive Space

*First Sergeant 120 sqf*

### 1.3 *Unit Common*

if not provided in Unit Exclusive each positions authorized 60 sqf

Used by all units on their respective drill weekend.

Determined for each drill weekend

Intra-functional circulation = Area x %15

Total Area = Area + Intra-functional Area

Recruiting and retention

*250 sqf office space to be used by all units*

---

## **1.4 Administrative Support**

### Reproduction

MailArea=  $120 + (\text{ROUNDUP}(\text{Number of Members}/50) \times 60)$

Message Center if (Area > 360) then Area = 360

### Administrative Storage

### Reserve Component Automation System (RCAS)

**Condition:** Required strength or type of unit identified by unit's approved MTOE or TDA

### System Administrator

### Processing Classified Materials

Area =  $120 +$

$(\text{ROUNDUP}(\text{Number of Workstation}/8) \times 60 \text{ sqf})$

if (Area > 480) then Area = 480 sqf

### Lobby

Area = 80 sqf

### Assembly Area

Assembly Hall Based on the total authorized drilling strength of the largest drill weekend.

Area =  $3000 + (\text{ROUNDUP}(\text{Number of members} / 50) \times 600)$

if (Area > 6200) then Area = 6200 sqf

### Chair Storage Area

Area =  $\text{Area}(\text{Assembly Hall}) \times \%10$

### Kitchen (Standard Design)

*Kitchen 730 sqf*

*Office 81 sqf*

## **1.5 Weapons**

### Arms vault

Based on the total authorized center strength for units that are authorized to have weapons.

Area = single-served weapon storage + crew-served weapon storage

Single-served Weapon Area =  $220 + (\text{ROUNDUP}(\text{Number of members} / 100) \times 110)$

Crew-served Weapon Area =  $\text{ROUNDUP}(\text{Number of Crew-served Weapons} / 50) \times 110$

### Armorer

Area = 110

---

## 1.6 Education

### Classrooms

Area based on the total authorized drilling strength of the largest drill weekend

$$\text{Area} = \text{ROUNDUP} ( \text{Number of members} / 50 ) \times 300$$

### Library-Reading

$$\text{Area} = \text{ROUNDUP} ( \text{Number of members} / 50 ) \times 75$$

### Library Storage

$$\text{Area} = \text{Area} ( \text{Classrooms} ) \times \%10$$

### Learning Center

$$\text{Area} = \text{ROUNDUP} ( \text{Number of members} / 50 ) \times 50$$

$$\text{if } (\text{Area} < 100) \text{ Area} = 100 \text{ sqf}$$

### Training Aid Storage

$$\text{Area} = \text{Area} ( \text{Classrooms} ) \times \%10$$

### Communication Security Training (COMSEC account issued)

$$\text{Area} = 100$$

### Communication Security Storage

$$\text{Area} = 100$$

### Instructor Classroom

$$\text{if } ( \text{TYPE} ( \text{USARF} ) )$$

$$\text{Area} = 300$$

### Publication Storage

$$\text{if } ( \text{TYPE} ( \text{USARF} ) ) \text{ then Assign} = \text{true}$$

### Storage

### Unit and Individual Storage

Area based on total authorized station strength

$$( \text{TYPE} ( \text{Unit} ) ) \Rightarrow \text{number of standard 96 sqf cages}$$

$$\text{if } ( \text{TYPE} ( \text{Unit} ) = \text{USARF} )$$

$$\text{If } ( \text{TYPE} ( \text{Unit} ) = \text{NonSchool} - \text{TDA OR Training Division Unit} )$$

$$\text{Number of Standard Cage} = \text{ROUNDUP} ( \text{Number of Members} / 20 )$$

$$\text{If } ( \text{TYPE} ( \text{Unit} ) = \text{MTOE} )$$

$$\text{Number of Standard Cage} = \text{ROUNDUP} ( \text{Number of Members} / 10 )$$

$$\text{Area} = \text{Intrafunctional Circulation Area} + \text{Unit Storage Area}$$

$$\text{Number of Standard Cage} = \text{ROUNDUP} ( \text{Number of Members} / 6 )$$

$$\text{Unit Storage Area} = \text{Number of Cages} \times 96 \text{ sqf}$$

$$\text{Intrafunctional Circulation Area} = \text{Unit Storage Area} \times \%15$$



---

#### Staging Area

Area = Unit Storage Area x %10

#### Supply office

Adjacent to Unit and individual storage

If (Full-time supply technician assigned)

Area = 120

else

Property Account assigned

Area = 96

#### Janitorial Storage

if ( Two level building)

Area = 25 sqf each floor

else Area = 50 sqf

#### Flammable Storage

if ( maintenance Shop is not collocated)

Area = 150 sqf

else Area = 0

#### Controlled Waste Storage

if ( maintenance Shop is not collocated)

Area = 96 sqf

#### Facility Maintenance and Storage for the custodial contractor

Area = 200 + ( ROUNDUP ( Number of Members / 10 ) - 1 ) x 100

if ( Area > 800 ) Area = 800

### **1.7 Spacial Training Area**

#### Rifle Range

if ( Assigned (Rifle-Range) )

Area = Number of Fire Range Lane x 375

#### Photo-Lab

If (Assigned (Photo-Lab) )

Area = 250

#### Band-Room

if ( Assigned (Band-Room) )

Area = 2850

#### Medical

if (Assigned ( Medical Training) )

Area = 400

---

Physical Exam

if ( Assigned ( Physical Exam ) )

Area = 2500

SCIF

if ( Assigned ( Sensitive Compartmented Information Facility ) )

Area = 500

Solid Testing Lab

if ( Assigned ( Solid Testing ) )

Area = 150

Conference Room

Area = 400

Drafting

if ( Assigned ( Drafting Equipment ) )

if ( Number of Draftsperson > 4 )

Area = 250 + ( Number of Draftsperson - 4 ) x 60

else Area = 250

Physical Rediness Area

Area = 200 + ( ROUNDUP ( Number of Members / 10 ) x 100 )

if ( Area > 1600 )

Area = 1600

Worldwide Military Command and Control System Terminals

Area = 200

**1.8 Specialized Areas**

Required Information: Sketch detailing room dimensions

Location of MTOE or TDA equipment to be installed or used in the area

Proposed individual workspace for people training or working in the area

**1.9 Support Areas**

Men's Toilets and Showers

Area = 350 + ( ROUNDUP ( ( Number of Members x %90 ) / 50 ) - 1 ) x 100

Area would be finalized according to required number of fixtures

Women's Toilets and Showers

Area = 225 + ( ROUNDUP ( ( Number of Members x %30 ) / 50 ) - 1 ) x 25

Area would be finalized according to required number of fixtures

Unisex Handicap Toilet

Area = 75

---

Locker Room

Area =  $1100 + (\text{ROUNDUP} (\text{Number of Members} / 10) - 1) \times 100$   
if ( Area > 2100 )  
Area = 2100

Vending Machine Alcove

Area = 48

Full-time Personnel Break Kitchen

Area = 218

Electrical Distribution

Area = 100 or  
Area required by services and electrical equipment

Telephone Distribution

Area = 100 or Area required by services and telephone equipment

Mechanical Room

Area ( Area (Support-Areas) x %2 )

Circulation

if ( Area ( Training-Building) > 20000 )  
Story ( Training Center) = Multi Story  
Area = Area (Training Center) x %22  
else  
Story ( Training Center) = Single Story  
Area = Area (Training Center) x %15

Structural Allowances

Area = Area (Training Building ) x %10

**1.10 Special Purpose Facilities**

Equipment Concentration Sites

Hardstand area

Area = Number of Items x ( 50 + 5 )

Fuel Storage and Disensing Systems

One system per type of fuel

Military Equipment Loading Ramp

if ( Justified )

Wash Platform

Number = ROUNDUP ( Number of Equipments / 100 )

Warehouse

---

If ( Assigned ( Equipment requires Indoor storage ) )

Arms Vault

If ( Assigned ( Arm ) )

## **2 MILITARY EQUIPMENT PARKING AND TRAINING FACILITIES**

### **2.1 Organizational Maintenance Shop**

Area Reduction Tolerance =  $\pm 10\%$  ( Area ( OMS ) - Area ( Workbay ) )

if ( Number of Vehicles > 10 )

Assign (OMS) = TRUE

Shop Office

Area = ( Number of Administrative Person x 60 )

+ ( Number of Full-time Adm. Officer x 120 )

Workbay

Area = Number of Bays x ( 40 x 20 ) + ( 4 ft. x 40 + 4 ft. x 20 )

Number of Bays = ROUNDUP ( Number of Vehicles ) / 4

Number of Vehicles = Wheeled + Tracked + Eng. Equipments

if ( Collocatedwith ( AMSA ) )

Additional Bays

else No Additional Bay

Install\_in ( FOR EACH workbay )

- Compressed Air system
- Hose bibb
- Bench
- Hotwater heater

Tools and Parts

Area = 96

Install\_in ( FOR EACH tools/parts room )

- Shelving

Toilets

Unisex Toilet

Area = 75

if ( REQUIRED ( Men's AND Women's )

Storage Room

Area = Number of Workbays x 96

Battrey Room

Area = 50 + ( ( Number of Workbays - 1 ) x 25 )

---

if ( Area ( Battery Room ) > 200 )  
Area = 200

Flammable Storage

Area = 50 + ( ( Number of Workbays - 1 ) x 25 )  
if ( Area ( Flammable Storage ) > 200 )  
Area = 200

Controlled Waste Storage

Area = 96 + ( ( Number of Workbays - 1 ) x 25 )  
if ( Area ( CWS ) > 596 )  
Area = 596

Mechanical/Custodial Room

Area = NetArea ( OMA Building ) x %3  
if ( Area ( Mechanical/Custodial ) < 50 )  
Area = 50

**2.2 Area Maintenance Support Activities**

Shop Office

Area = ( Number of Reserve Administrative Person x 60 )  
+ ( Number of Full-time Adm. Officer x 120 )

Men's Toilet

Nominal Area = 200

Women's Toilet

Nominal Area = 150

Locker Room

Area = Number of AMSA Person x 10  
if ( Area ( LockerRoom ) < 100 )  
Area = 100

Class Room / Break Area

Area = Number of AMSA Personnel x 10  
if ( Area ( Class Room ) < 200 )  
Area = 200

Workbay

Area = Number of Bays x ( 40 x 20 ) + ( 4 ft. x 40 + 4 ft. x 20 )  
Number of Bays = ROUNDUP ( Number of Vehicles ) / 2

Number of Vehicles = Wheeled + Tracked + Eng. Equipments

Install\_in ( FOR EACH workbay )

- 
- Compressed Air system
  - Hose bibb
  - Bench
  - Hotwater heater

#### Tool Room

Area = Number of Bays x 96

#### Supply Room

Area = Number of Bays x 96

#### Battery Room

Area = Number of Bays x 50

if ( Area ( BatteryRoom ) > 400 )

Area = 400

#### Electronics Shop

Area = Number of Electronic Technicians x 150

#### Instrument Repair Shop

Area = Number of Instrument Repair Technician x 100

#### Small Arms Repair Shop

Area = Number of Small Arms Repair Technician x 100

#### Small Arms Vault

Area = 100

#### Flammable Storage

Area = Number of Workbays x 25

if ( Area ( Flammable Storage ) < 50 )

Area = 50

#### Controlled Waste Storage

Area = Number of Workbays x 96

if ( Area ( Controlled Waste Storage ) > 596 )

Area = 596

#### Mechanical/Custodial Room

Area = NetArea ( AMSA Building ) x %3

if ( Area ( Mechanical/Custodial ) < 50 )

Area = 50

#### Privately Owned Car Parking Area

if ( Collocated ( AMSA, USARC ) )

USES ( USARC parking )

---

else

Area = Number of Members x %80 x 315

Service or Access Apron

Area = 36 x WIDTH ( Workbays )

Wash Platform

if ( Collocated ( AMSA , OMA ) )

USES ( OMA wash Platform)

else

One wash Platform

### **3 Direct Support/General Support Maintenance**

Specified by specialized units and support activities

Military Equipment Storage

if ( NOT EQUIPMENT.Parked\_at OMS )

Area = Number of Equipment x 450

if (NOT EQUIPMENT.Parked\_at AMSA )

Area = Number of Equipment x %10 x 450

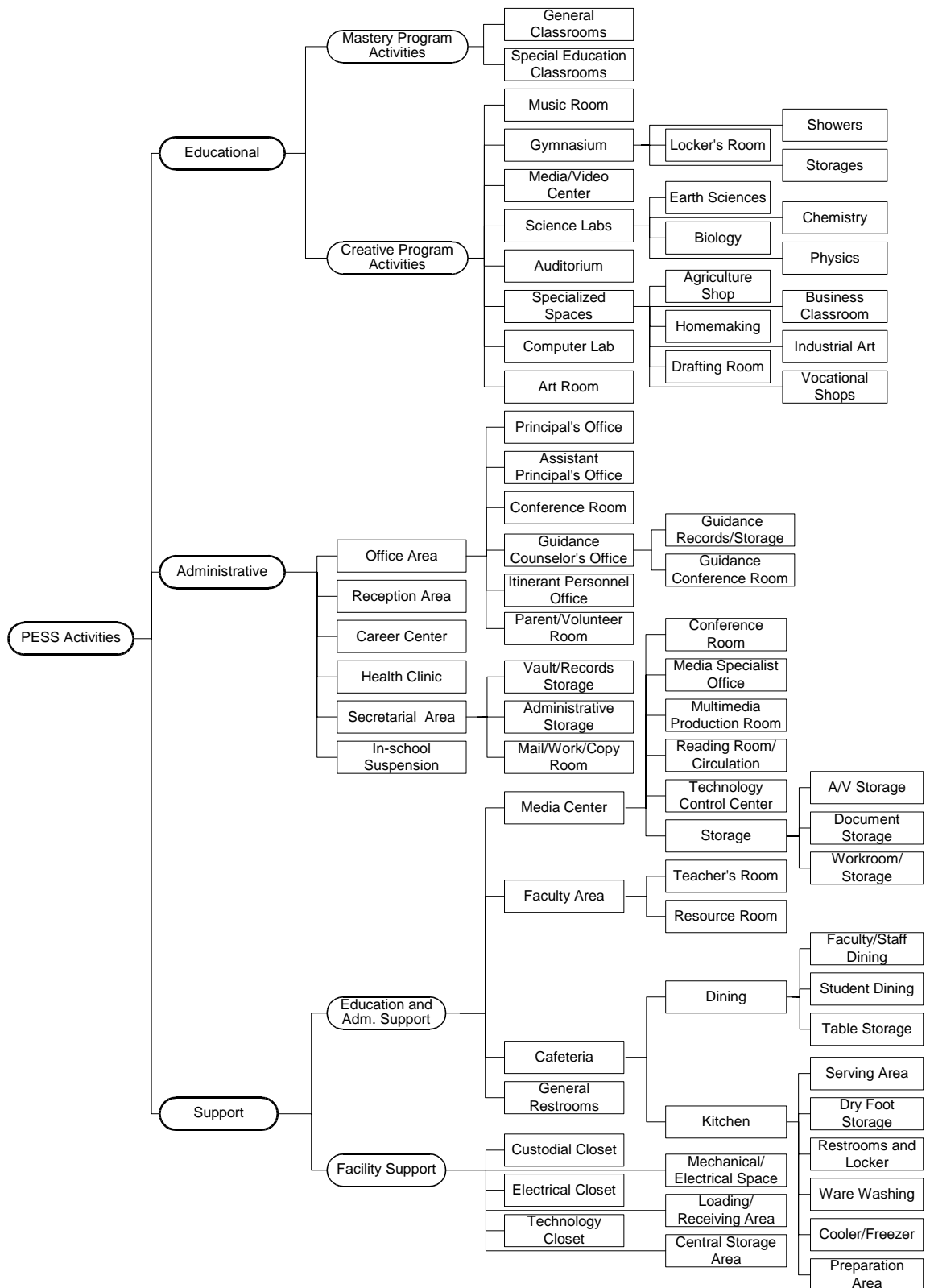
---

## *Appendix B: ESPS Space Assignments (Case Studies)*

- ESPS activities and space.
- ESPS activity-to-space transition tree.

<b>1</b>	<b>Academic Core Spaces</b>
<b>2</b>	<b>Special Education Spaces</b>
<b>3</b>	<b>Administrative Spaces</b>
<b>4</b>	<b>Media Center Spaces</b>
<b>5</b>	<b>Visual Arts Spaces</b>
<b>6</b>	<b>Music Spaces</b>
<b>7</b>	<b>Technology Education Spaces</b>
<b>8</b>	<b>Business Education Spaces</b>
<b>9</b>	<b>Family and Consumer Science Spaces</b>
<b>10</b>	<b>Physical Education Spaces</b>
<b>11</b>	<b>Student Dining Spaces</b>
<b>12</b>	<b>Food Service Spaces</b>
<b>13</b>	<b>Custodial Spaces</b>
<b>14</b>	<b>Building Services</b>





---

## Appendix B: Program Generation

---

1. RaBBiT Architectural Program Generation Algorithm
2. Program schema in XML
3. Schema transformation from XML to HTML through XSLT (style sheet)
4. Partial generated program in XML
5. Partial generated program view in HTML after schema transformation

---

## Program Generation Algorithm

```
FUNCTION generate component order ( Rabbit Graph graph ) → Map
  hash map := Map { ( (Component component), (Value ( boolean condition, int weight ) ) }
  roots := get all root components (graph) →
    ( Set { roots | root IS Component AND  $\neg$  depend on other components } )

  FOR  $\exists$  root  $\in$  { roots }
    visit (root, value (true, 0), hash map )
  END:FOR
  RETURN hash map
END:FUNCTION

FUNCTION visit (Component component, Value value, Map hash map )

  insert in hash map (component, value)

  IF condition of value = TRUE
    edges := get all dependency edges sourcing from ( component ) →
      (Set { edges | edge IS Dependency Edge AND source of edge IS component } )

    FOR  $\exists$  edge  $\in$  { edges }

      target := get target (edge) → (Component target component )

      value current := get value clone ( edge ) →
        (Value copy of dependency in edge )
      value previous := get from hash map ( target ) →
        (Value value of target from hash order )

      IF value previous  $\neg$  null

        IF weight of value previous > weight of value current
          visit ( target, value previous, hash map )

        ELSE IF weight of value previous = weight of value current
          condition of value current := condition of value previous OR
            ( condition of current value AND condition of value )
          visit ( target, value current, hash map )

        ELSE IF weight of value previous < weight of value current
          condition of value current :=
            condition of value AND condition of value current
          visit ( target, value current, hash map )

        END:IFELSE

      ELSE
        condition of value current := condition of value AND condition of value current
        visit ( target, value current, hash map )

      END:IFELSE
    END:FOR
  END:IF
END:FUNCTION
```

---

**RaBBiT Generated Architectural Program View**

**Project Details**

**Project Information**

Name:	A project name
Location:	Project location
ID:	AAA-123-1234
Description:	Enter project description in this text area

**Client Information**

Name:	Client A
Contact Name:	Emre Efe Rabbit
Phone:	555-555-5555
Email:	rabbitkus@netscape.com
Adress:	Address information form will be redesigned

**Version Information**

Generation Date:	2003-12-14 00:02:48
Version Number:	v0.0.0
Description:	Write the version description here

# Global Parameters

## Parameter: Training

Description:	None Available
Parameter Observers:	No Observer
Unit:	t/f
Value:	boolean: <b>true</b>

## Parameter: Maintenance Training

Description:	None Available
Parameter Observers:	No Observer
Unit:	t/f
Value:	boolean: <b>false</b>

## Parameter: Reserves

Description:	None Available				
Parameter Observers:	Requirement	Parameter	Value		
	Full time staff	Officers	Expression Result integer: <b>3</b>	Expression / Formula roundup( Reserves / 100 )	Referenced Parameters Reserves of Global
	Officer Room	Number	Referenced Value integer: <b>300</b>		Referenced Parameter Reserves of Global
	Ranking Officer_TO_Major General	Ranking Officer_TO_Major General	boolean: <b>false</b>		
	Ranking Officer_TO_Brigadier General	Ranking Officer_TO_Brigadier General	boolean: <b>false</b>		
	Ranking Officer_TO_Colonel	Ranking Officer_TO_Colonel	boolean: <b>true</b>		
	Ranking Officer	Batalion or higher	Expression Result double: <b>0.0</b>	Expression / Formula if (Reserves > 500, true, false)	Referenced Parameters Reserves of Global
	Lietenant Colonel	Number	Expression Result integer: <b>0</b>	Expression / Formula rounddown ( Reserves / 400 )	Referenced Parameters Reserves of Global
			Expression Result	Expression / Formula	Referenced Parameters

Colonel	Number	integer: 1	rounddown (Reserves / 300)	Reserves of Global
Brigadier General	Number	Expression Result  double: 0.0	Expression / Formula  if ( Ranking_Officer_TO_Brigadier_General, rounddown( Reserves / 600), 0)	Referenced Parameters Ranking Officer_TO_Brigadier General of Ranking Officer_TO_Brigadier GeneralReserves of Global
Staff Section Chief	Number	Expression Result  integer: 1	Expression / Formula  roundup( Reserves / 300 )	Referenced Parameters  Reserves of Global
Staff Section Chief	Number	Expression Result  integer: 2	Expression / Formula  roundup( Reserves / 200 )	Referenced Parameters  Reserves of Global

Unit: unit

Value: integer: 300

## Parameter: **Weapon Training**

Description: None Available

Parameter Observers: No Observer

Unit: t/f

Value: boolean: true

## Requirement Information by Categories

### Category: **Project Mission**

Description: Describes the mission of the project, including the mission of the USAR unit assigned. The design requirements of the facility have to be determined for the particular missions assigned to the units.

Information Level: 0

### Requirements:

#### Requirement: **Mission**

Description: Design a USARC for a number of reserves

##### Depended **by** Requirements

Requirement	Condition		
Full time staff	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes
Maintenance Activities	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes
Training Activities	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes

##### Relationships **with** Requirements

### Category: **Activities**

Description: The requirement information fall under this group includes the activities that a USAR unit is assigned. Basically, training and training-related maintenance activities.

Information Level: 1

### Requirements:

#### Requirement: **Administrative**

Description: None available

##### Depends **on** Requirements

Requirement	Condition		
Training Activities	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes

##### Depended **by** Requirements

Requirement	Condition		
Administration Zone	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes
Unit Exclusive Zone	Relative Weight	Expression/Formula	Is Satisfied?

	0	true	yes	
--	---	------	-----	--

Relationships **with** Requirements

## Requirement: **Training Activities**

Description: None available

Depends **on** Requirements

Requirement	Condition			
	Relative Weight	Expression/Formula	Is Satisfied?	
Mission	0	true	yes	

Depended **by** Requirements

Requirement	Condition			
	Relative Weight	Expression/Formula	Is Satisfied?	
Administrative	0	true	yes	

Relationships **with** Requirements

## Requirement: **Maintenance Activities**

Description: USARC maintenance activities as described in the design guidelines

Depends **on** Requirements

Requirement	Condition			
	Relative Weight	Expression/Formula	Is Satisfied?	
Mission	0	true	yes	

Relationships **with** Requirements

## Category: **Staff**

An Army unit structure is hierarchical and composed of smaller to larger groups. Smallest unit is a "reservee", an USAR personnel who is under training. Each larger group is trained by a ranking officer, which his/her rank changes as the number of reserves changes in the group. During the drill period, depending on the number of reservees (troops) to be trained, the rank structure of the ARU changes. The rank structure and number of troops in an USAR unit is reflected in the design requirements. The maximum number of reserves is defined as total authorized drilling strength (number of reserves) of the largest drill (training) weekend. The largest drill strength is headed by the highest ranked officer in the ARU.

Information Level: 2

## Requirements:

### Requirement: **Full time staff**

Description: None available

Parameter Name	Unit	Value	Parameter Observer			Note
Officers		ExpressionExpression /	Requirement	Parameter	Value	



		<div>Result</div> <div>integer: 3</div>	<div>Formula</div> <div>( Reserves / 100 )</div>	<div>Referenced Parameters</div> <div>Reserves of Global</div>	Full time staff	Civilians	<div>Expression Result</div> <div>integer: 6</div>	<div>Expression / Formula</div> <div>roundup (Officers_OF_Full_time_staff * 2)</div>	<div>Referenced Parameters</div> <div>Officers of Full time staff</div>	
					Full time staff	Enlisted	<div>Expression Result</div> <div>integer: 9</div>	<div>Expression / Formula</div> <div>roundup ( Officers_OF_Full_time_staff * 3)</div>	<div>Referenced Parameters</div> <div>Officers of Full time staff</div>	
					Officer Room	Number	<div>Referenced Value</div> <div>integer: 3</div>	<div>Expression Formula</div> <div>roundup ( Reserves / 100 )</div>	<div>Referenced Parameters</div> <div>Reserves of Global</div>	<div>Referenced Parameter</div> <div>Officers of Full time staff</div>
Civilians		<div>Expression Result</div> <div>integer: 6</div>	<div>Expression / Formula</div> <div>roundup (Officers_OF_Full_time_staff * 2)</div>	<div>Referenced Parameters</div> <div>Officers of Full time staff</div>						
Enlisted		<div>Expression Result</div> <div>integer: 9</div>	<div>Expression / Formula</div> <div>roundup ( Officers_OF_Full_time_staff * 3)</div>	<div>Referenced Parameters</div> <div>Officers of Full time staff</div>	Requirement	Parameter	Value			
					Civilian Office	Number	<div>Referenced Value</div> <div>integer: 9</div>	<div>Expression Formula</div> <div>roundup ( Officers_OF_Full_time_staff * 3)</div>	<div>Referenced Parameters</div> <div>Officers of Full time staff</div>	<div>Referenced Parameter</div> <div>Enlisted of Full time staff</div>
					Enlisted Office	Total Area	<div>Expression Result</div> <div>double: 540.0</div>	<div>Expression / Formula</div> <div>Area_per_person_OF_Enlisted_Office * Enlisted_OF_Full_time_staff</div>	<div>Referenced Parameters</div> <div>Area per person of Enlisted Office</div>	<div>Referenced Parameter</div> <div>Enlisted of Full time staff</div>

Depends on Requirements

Requirement	Condition		
Mission	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes

Relationships with Requirements

Requirement: **Lietenant Colonel**

Description: None available

Parameter Name	Unit	Value			Parameter Observer	Note
Number		Expression Result	Expression / Formula	Referenced Parameters		
		integer: 0	rounddown ( Reserves / 400 )	Reserves of Global		

Depends **on** Requirements

Requirement	Condition			
Ranking Officer	Relative Weight	Expression/Formula	Is Satisfied?	
	0	true	yes	

Depended **by** Requirements

Requirement	Condition			
Chief of Staff	Relative Weight	Expression/Formula	Is Satisfied?	
	0	false	no	
Staff Section Chief	Relative Weight	Expression/Formula	Is Satisfied?	
	0	false	no	
Staff Section Chief	Relative Weight	Expression/Formula	Is Satisfied?	
	0	false	no	
Executive Officer	Relative Weight	Expression/Formula	Is Satisfied?	
	0	false	no	
Deputy Commander	Relative Weight	Expression/Formula	Is Satisfied?	
	0	false	no	

Relationships **with** Requirements

Requirement: **Colonel**

Description: None available

Parameter Name	Unit	Value			Parameter Observer	Note
Number		Expression Result	Expression / Formula	Referenced Parameters		
		integer: 1	rounddown (Reserves / 300)	Reserves of Global		

Depends **on** Requirements

Requirement	Condition			
Ranking Officer	Relative Weight	Expression/Formula	Is Satisfied?	
	0	true	yes	

Relationships **with** Requirements

Requirement: **Ranking Officer**

Description: None available

Parameter Name	Unit	Value	Parameter Observer	Note
----------------	------	-------	--------------------	------

Batalion or higher	Expression Result	Expression / Formula	Referenced Parameters	Requirement	Parameter	Value		
	double: 0.0	if (Reserves > 500, true, false)	Reserves of Global	Ranking Officer_TO_Deputy Commander	Ranking Officer_TO_Deputy Commander	Expression Result	Expression / Formula	Referenced Parameters
						double: 0.0	Batalion_or_higher_OF_Ranking_Officer	Batalion or higher of Ranking Officer
				_TO_	Ranking Officer_TO_Deputy Commander	Expression Result	Expression / Formula	Referenced Parameters
						double: 0.0	Batalion_or_higher_OF_Ranking_Officer	Batalion or higher of Ranking Officer
				_TO_	Ranking Officer_TO_Deputy Commander	Expression Result	Expression / Formula	Referenced Parameters
						double: 0.0	Batalion_or_higher_OF_Ranking_Officer	Batalion or higher of Ranking Officer
				Lietenant Colonel_TO_Deputy Commander	Lietenant Colonel_TO_Deputy Commander	boolean: false		
				Lietenant Colonel_TO_Executive Officer	Lietenant Colonel_TO_Executive Officer	boolean: false		
				Lietenant Colonel_TO_Deputy Commander	Lietenant Colonel_TO_Deputy Commander	Expression Result	Expression / Formula	Referenced Parameters
						double: 0.0	Batalion_or_higher_OF_Ranking_Officer	Batalion or higher of Ranking Officer
				Lietenant Colonel_TO_Chief of Staff	Lietenant Colonel_TO_Chief of Staff	boolean: false		
				Lietenant Colonel_TO_Staff Section Chief	Lietenant Colonel_TO_Staff Section Chief	boolean: false		
				Lietenant Colonel_TO_Staff Section Chief	Lietenant Colonel_TO_Staff Section Chief	boolean: false		

Depended by Requirements

Requirement	Condition		
Lietenant Colonel	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes
Major General	Relative Weight	Expression/Formula	Is Satisfied?
	0	false	no

Colonel	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes
Brigadier General	Relative Weight	Expression/Formula	Is Satisfied?
	0	false	no

Relationships **with** Requirements

## Category: **Spatial Allocations**

Description: Space area allowances and designations of USARC activities.

Information Level: 3

### Requirements:

#### Requirement: **Admininstration Zone**

Description: None available

Parameter Name	Unit	Value	Parameter Observer
Total Area	sqf	<div>Expression Result</div> <div>double: <b>1620.0</b></div> <div> <div>Expression / Formula</div> <div> Total_Area_OF_Officer_Room  +  Total_Area_OF_Civilian_Office  +  Total_Area_OF_Enlisted_Office </div> </div> <div> <div>Referenced Parameters</div> <div> Total Area of Officer Room  Total Area of Civilian Office  Total Area of Enlisted Office </div> </div>	

Depends **on** Requirements

Requirement	Condition
Administrative	<div>Relative Weight</div> <div>0</div> <div>Expression/Formula</div> <div>true</div> <div>Is Satisfied?</div> <div>yes</div>

Depended **by** Requirements

Requirement	Condition
Civilian Office	<div>Relative Weight</div> <div>0</div> <div>Expression/Formula</div> <div>true</div> <div>Is Satisfied?</div> <div>yes</div>
Unit Exclusive Zone	<div>Relative Weight</div> <div>0</div> <div>Expression/Formula</div> <div>true</div> <div>Is Satisfied?</div> <div>yes</div>
Enlisted Office	<div>Relative Weight</div> <div>0</div> <div>Expression/Formula</div> <div>true</div> <div>Is Satisfied?</div> <div>yes</div>
Officer Room	<div>Relative Weight</div> <div>0</div> <div>Expression/Formula</div> <div>true</div> <div>Is Satisfied?</div> <div>yes</div>

Relationships **with** Requirements

#### Requirement: **Officer Room**

Description: None available

Parameter Name	Unit	Value			Parameter Observer			
Number	room	Referenced Value			Requirement	Parameter	Value	
		Expression Result	Expression / Formula	Referenced Parameters			Expression Result	Expression / Formula
		integer: 3	roundup ( Reserves / 100 )	Reserves of Global	Officer Room	Total Area	double: 360.0	Number_OF_Officer_Room * Max_Area_OF_Officer_Room
Max Area	sqf	double: 120.0			Requirement	Parameter	Value	
							Expression Result	Expression / Formula
					Officer Room	Min Lenght	double: 10.0	Max_Area_OF_Officer_Room / Min_Width_OF_Officer_Room
					Officer Room	Total Area	double: 360.0	Number_OF_Officer_Room * Max_Area_OF_Officer_Room
Min Width	ft	integer: 12			Requirement	Parameter	Value	
							Expression Result	Expression / Formula
					Officer Room	Min Lenght	double: 10.0	Max_Area_OF_Officer_Room / Min_Width_OF_Officer_Room
Min Lenght	ft	Expression Result	Expression / Formula	Referenced Parameters				
		double: 10.0	Max_Area_OF_Officer_Room / Min_Width_OF_Officer_Room	Max Area of Officer Room Min Width of Officer Room				
Total Area	sqf	Expression Result	Expression / Formula	Referenced Parameters	Requirement	Parameter	Value	
				Number of			Expression Result	Expression / Formula

		double: <b>360.0</b>	Number_OF_Officer_Room * Max_Area_OF_Officer_Room	Officer RoomMax Area of Officer Room	Admininstration Zone	Total Area	double: <b>1620.0</b>	Total_Area_OF_Officer_Room + Total_Area_OF_Civilian_Office + Total_Area_OF_Enlisted_Office	T O R A C C A E C
--	--	----------------------	------------------------------------------------------	--------------------------------------------------	-------------------------	------------	-----------------------	--------------------------------------------------------------------------------------------------------	-------------------------------------------

Depends **on** Requirements

Requirement	Condition		
Admininstration Zone	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes

Relationships **with** Requirements

Requirement	Relation Type
Civilian Office	adjacent,

## Requirement: **Civilian Office**

Description: None available

Parameter Name	Unit	Value	Parameter Observer
Number	room	Referenced Value	Requirement
		Expression Result	Parameter
		Expression / Formula	Value
		Referenced Parameters	Expression Result
		Enlisted of	Expression / Formula
		Officers of Full time staff	Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result
			Expression / Formula
			Result
			Expression / Formula
			Result
			Parameter
			Value
			Expression Result

Min Width	unit	double: 10.0		Requirement	Parameter	Value		C
						Expression Result	Expression / Formula	R P M o C M C C
				Civilian Office	Min Lenght	double: 8.0	Max_Area_OF_Civilian_Office / Min_Width_OF_Civilian_Office	
Min Lenght		Expression Result double: 8.0	Expression / Formula Max_Area_OF_Civilian_Office / Min_Width_OF_Civilian_Office	Referenced Parameters Max Area of Civilian Office Min Width of Civilian Office				
Total Area		Expression Result double: 720.0	Expression / Formula Number_OF_Civilian_Office *Max_Area_OF_Civilian_Office	Referenced Parameters Number of Civilian Office Max Area of Civilian Office	Requirement	Parameter	Value	R P T o R A C C A E C
				Admininstration Zone	Total Area	double: 1620.0	Total_Area_OF_Officer_Room + Total_Area_OF_Civilian_Office + Total_Area_OF_Enlisted_Office	

Depends **on** Requirements

Requirement	Condition		
Admininstration Zone	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes

Relationships **with** Requirements

Requirement	Relation Type
Officer Room	adjacent,

## Requirement: Enlisted Office

Description: None available

Parameter Name	Unit	Value	Parameter Observer				
Area per person	sqf/person	integer: 60	Requirement	Parameter	Value		
					Expression Result	Expression / Formula	Refe Para Are: pers
			Enlisted				

				Office	Total Area	double: 540.0	Area_per_person_OF_Enlisted_Office * Enlisted_OF_Full_time_staff	Enli Offic of F staff
Total Area	sqf	Expression Result	Expression / Formula	Referenced Parameters	Requirement	Parameter	Value	
		double: 540.0	Area_per_person_OF_Enlisted_Office * Enlisted_OF_Full_time_staff	Area per person of Enlisted Office of Full time staff	Admininstration Zone	Total Area	double: 1620.0	Total_Area_OF_Officer_Room + Total_Area_OF_Civilian_Office + Total_Area_OF_Enlisted_Office
					Unit Exclusive Zone	Total Area	double: 1390.0	Total_Area_OF_Enlisted_Office + Area_OF_M_General_Office + Area_0_OF_B_General_Office + Area_1_OF_Secretary_MG

Depends on Requirements

Requirement	Condition		
Unit Exclusive Zone	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes
Admininstration Zone	Relative Weight	Expression/Formula	Is Satisfied?
	0	true	yes

Relationships with Requirements

## Requirement: Unit Exclusive Zone

Description: None available

Parameter Name	Unit	Value	Parameter Observer
Total Area		Expression Result	
		Expression / Formula	
		Referenced Parameters	
		Total_Area_OF_Enlisted_Office + Area_OF_M_General_Office	Total Area of Enlisted OfficeArea of M



		double: 1390.0	+	Area_0_OF_B_General_Office	General OfficeArea 0 of
			+	Area_1_OF_Secretary_MG	B General OfficeArea 1
					of Secretary MG

Depends **on** Requirements

Requirement	Condition			
Admininstration Zone	Relative Weight	Expression/Formula	Is Satisfied?	
	0	true	yes	
Administrative	Relative Weight	Expression/Formula	Is Satisfied?	
	0	true	yes	

Depended **by** Requirements

Requirement	Condition			
Enlisted Office	Relative Weight	Expression/Formula	Is Satisfied?	
	0	true	yes	
B General Office	Relative Weight	Expression/Formula	Is Satisfied?	
	0	Ranking_Officer_TO_Brigadier_General	no	
M General Office	Relative Weight	Expression/Formula	Is Satisfied?	
	0	Ranking_Officer_TO_Major_General	no	

Relationships **with** Requirements

Category: **Equipment Furniture**

Description: Description:

Information Level: 4

**Requirements:**

```

<?xml version="1.0" encoding="UTF-8" ?>
<!-- XML file generated RabbitProgramGenerator -->
- <RabbitProject xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="C:\Documents and
  Settings\hie\Desktop\RabbitXMLSchema.xsd">
- <RabbitProjectInformation>
  - <project>
    <projectName>A project name</projectName>
    <projectLocation>Project location</projectLocation>
    <projectID>AAA-123-1234</projectID>
    <projectDesc>Enter project description in this text area</projectDesc>
  </project>
  - <client>
    <clientName>Client A</clientName>
    <contactName>Emre Efe Rabbit</contactName>
    <contactPhone>555-555-5555</contactPhone>
    <contactEmail>rabbitkus@netscape.com</contactEmail>
    <contactAdress>Address information form will be
      redesigned</contactAdress>
  </client>
  - <versionInformation>
    <date>2003-12-14 00:02:48</date>
    <versionNumber>v0.0.0</versionNumber>
    <versionDescription>Write the version description
      here</versionDescription>
  </versionInformation>
</RabbitProjectInformation>
- <constructGlobalAs>
  <constructName>Training</constructName>
  <constructDescription />
  <unit>t/f</unit>
- <value>
  - <valueSimple>
    <type>boolean</type>
    <boolean>true</boolean>
  </valueSimple>
</value>
</constructGlobalAs>
- <constructGlobalAs>
  <constructName>Maintenance Training</constructName>
  <constructDescription />
  <unit>t/f</unit>
- <value>
  - <valueSimple>
    <type>boolean</type>
    <boolean>>false</boolean>
  </valueSimple>
</value>
</constructGlobalAs>
- <constructGlobalAs>
  <constructName>Reserves</constructName>
  <constructDescription />

```

```

- <constructObservedBy>
  <componentName>Full time staff</componentName>
  <constructName>Officers</constructName>
- <value>
  - <valueComplex>
    <type>Expression</type>
    - <valueExpression>
      - <value>
        - <valueSimple>
          <type>integer</type>
          <integer>3</integer>
          </valueSimple>
        </value>
        <expression>roundup( Reserves / 100 )</expression>
      - <valueReadFrom>
        <componentName>GLOBAL PARAMETER</componentName>
        <constructName>Reserves</constructName>
        - <value>
          - <valueSimple>
            <type>integer</type>
            <integer>300</integer>
            </valueSimple>
          </value>
        </valueReadFrom>
      </valueExpression>
    </valueComplex>
  </value>
</constructObservedBy>
- <constructObservedBy>
  <componentName>Officer Room</componentName>
  <constructName>Number</constructName>
- <value>
  - <valueComplex>
    <type>Reference</type>
    - <valueReference>
      - <value>
        - <valueSimple>
          <type>integer</type>
          <integer>300</integer>
          </valueSimple>
        </value>
      - <valueReadFrom>
        <componentName>GLOBAL PARAMETER</componentName>
        <constructName>Reserves</constructName>
        - <value>
          - <valueSimple>
            <type>integer</type>
            <integer>300</integer>
            </valueSimple>
          </value>
        </valueReadFrom>
      </valueReference>
    </valueComplex>
  </value>
</constructObservedBy>

```

```

        </valueComplex>
    </value>
</constructObservedBy>
- <constructObservedBy>
    <componentName>Ranking Officer_TO_Major General</componentName>
    <constructName>Ranking Officer_TO_Major General</constructName>
    - <value>
        - <valueSimple>
            <type>boolean</type>
            <boolean>>false</boolean>
        </valueSimple>
    </value>
</constructObservedBy>
- <constructObservedBy>
    <componentName>Ranking Officer_TO_Brigadier
        General</componentName>
    <constructName>Ranking Officer_TO_Brigadier General</constructName>
    - <value>
        - <valueSimple>
            <type>boolean</type>
            <boolean>>false</boolean>
        </valueSimple>
    </value>
</constructObservedBy>
- <constructObservedBy>
    <componentName>Ranking Officer_TO_Colonel</componentName>
    <constructName>Ranking Officer_TO_Colonel</constructName>
    - <value>
        - <valueSimple>
            <type>boolean</type>
            <boolean>>true</boolean>
        </valueSimple>
    </value>
</constructObservedBy>
- <constructObservedBy>
    <componentName>Ranking Officer</componentName>
    <constructName>Batalion or higher</constructName>
    - <value>
        - <valueComplex>
            <type>Expression</type>
        - <valueExpression>
            - <value>
                - <valueSimple>
                    <type>double</type>
                    <double>0.0</double>
                </valueSimple>
            </value>
            <expression>if (Reserves > 500, true, false)</expression>
        - <valueReadFrom>
            <componentName>GLOBAL PARAMETER</componentName>
            <constructName>Reserves</constructName>
        - <value>

```

```

        - <valueSimple>
            <type>integer</type>
            <integer>300</integer>
        </valueSimple>
    </value>
</valueReadFrom>
</valueExpression>
</valueComplex>
</value>
</constructObservedBy>
- <constructObservedBy>
    <componentName>Lietenant Colonel</componentName>
    <constructName>Number</constructName>
- <value>
    - <valueComplex>
        <type>Expression</type>
        - <valueExpression>
            - <value>
                - <valueSimple>
                    <type>integer</type>
                    <integer>0</integer>
                </valueSimple>
            </value>
            <expression>rounddown( Reserves / 400 )</expression>
        </valueExpression>
        - <valueReadFrom>
            <componentName>GLOBAL PARAMETER</componentName>
            <constructName>Reserves</constructName>
            - <value>
                - <valueSimple>
                    <type>integer</type>
                    <integer>300</integer>
                </valueSimple>
            </value>
            </valueReadFrom>
        </valueExpression>
    </valueComplex>
</value>
</constructObservedBy>
- <constructObservedBy>
    <componentName>Colonel</componentName>
    <constructName>Number</constructName>
- <value>
    - <valueComplex>
        <type>Expression</type>
        - <valueExpression>
            - <value>
                - <valueSimple>
                    <type>integer</type>
                    <integer>1</integer>
                </valueSimple>
            </value>
            <expression>rounddown (Reserves / 300)</expression>
        </valueExpression>
    </valueComplex>
</value>

```

```

- <valueReadFrom>
  <componentName>GLOBAL PARAMETER</componentName>
  <constructName>Reserves</constructName>
- <value>
  - <valueSimple>
    <type>integer</type>
    <integer>300</integer>
    </valueSimple>
  </value>
</valueReadFrom>
</valueExpression>
</valueComplex>
</value>
</constructObservedBy>
- <constructObservedBy>
  <componentName>Brigadier General</componentName>
  <constructName>Number</constructName>
- <value>
  - <valueComplex>
    <type>Expression</type>
  - <valueExpression>
    - <value>
      - <valueSimple>
        <type>double</type>
        <double>0.0</double>
        </valueSimple>
      </value>
      <expression>if ( Ranking_Officer_TO_Brigadier_General,
        rounddown( Reserves / 600), 0)</expression>
    - <valueReadFrom>
      <componentName>Ranking Officer_TO_Brigadier
        General</componentName>
      <constructName>Ranking Officer_TO_Brigadier
        General</constructName>
    - <value>
      - <valueSimple>
        <type>boolean</type>
        <boolean>>false</boolean>
        </valueSimple>
      </value>
    </valueReadFrom>
  - <valueReadFrom>
    <componentName>GLOBAL PARAMETER</componentName>
    <constructName>Reserves</constructName>
  - <value>
    - <valueSimple>
      <type>integer</type>
      <integer>300</integer>
      </valueSimple>
    </value>
    </valueReadFrom>
  </valueExpression>

```

```

    </valueComplex>
  </value>
</constructObservedBy>
- <constructObservedBy>
  <componentName>Staff Section Chief</componentName>
  <constructName>Number</constructName>
  - <value>
    - <valueComplex>
      <type>Expression</type>
      - <valueExpression>
        - <value>
          - <valueSimple>
            <type>integer</type>
            <integer>1</integer>
            </valueSimple>
          </value>
          <expression>roundup( Reserves / 300 )</expression>
        - <valueReadFrom>
          <componentName>GLOBAL PARAMETER</componentName>
          <constructName>Reserves</constructName>
          - <value>
            - <valueSimple>
              <type>integer</type>
              <integer>300</integer>
              </valueSimple>
            </value>
          </valueReadFrom>
        </valueExpression>
      </valueComplex>
    </value>
  </constructObservedBy>
- <constructObservedBy>
  <componentName>Staff Section Chief</componentName>
  <constructName>Number</constructName>
  - <value>
    - <valueComplex>
      <type>Expression</type>
      - <valueExpression>
        - <value>
          - <valueSimple>
            <type>integer</type>
            <integer>2</integer>
            </valueSimple>
          </value>
          <expression>roundup( Reserves / 200 )</expression>
        - <valueReadFrom>
          <componentName>GLOBAL PARAMETER</componentName>
          <constructName>Reserves</constructName>
          - <value>
            - <valueSimple>
              <type>integer</type>
              <integer>300</integer>
            </valueSimple>
          </value>
        </valueReadFrom>
      </valueExpression>
    </valueComplex>
  </value>
</constructObservedBy>

```

```

        </valueSimple>
      </value>
    </valueReadFrom>
  </valueExpression>
</valueComplex>
</value>
</constructObservedBy>
<unit>unit</unit>
- <value>
  - <valueSimple>
    <type>integer</type>
    <integer>300</integer>
  </valueSimple>
</value>
</constructGlobalAs>
- <constructGlobalAs>
  <constructName>Weapon Training</constructName>
  <constructDescription />
  <unit>t/f</unit>
- <value>
  - <valueSimple>
    <type>boolean</type>
    <boolean>true</boolean>
  </valueSimple>
</value>
</constructGlobalAs>
- <categoryList>
  - <category>
    <categoryName>Project Mission</categoryName>
    <categoryDescription>Describes the mission of the project, including the mission of the USAR unit assigned. The design requirements of the facility have to be determined for the particular missions assigned to the units.</categoryDescription>
    <categoryLevel>0</categoryLevel>
  - <component>
    - <componentInformation>
      <componentName>Mission</componentName>
      <componentDescription>Design a USARC for a number of reserves</componentDescription>
      <categoryName>Project Mission</categoryName>
    </componentInformation>
    - <dependedByComponents>
      - <componentDependency>
        <componentName>Full time staff</componentName>
        - <dependencyCondition>
          <weight>0</weight>
          <expression>true</expression>
          <isSatisfied>true</isSatisfied>
        </dependencyCondition>
      </componentDependency>
      - <componentDependency>
        <componentName>Maintenance Activities</componentName>

```



```

- <dependencyCondition>
  <weight>0</weight>
  <expression>true</expression>
  <isSatisfied>true</isSatisfied>
</dependencyCondition>
</componentDependency>
- <componentDependency>
  <componentName>Training Activities</componentName>
- <dependencyCondition>
  <weight>0</weight>
  <expression>true</expression>
  <isSatisfied>true</isSatisfied>
</dependencyCondition>
</componentDependency>
</dependedByComponents>
</component>
</category>
- <category>
  <categoryName>Activities</categoryName>
  <categoryDescription>The requirement information fall under this group
    includes the activities that a USAR unit is assigned. Basically, training
    and traningin-related maintenance activities.</categoryDescription>
  <categoryLevel>1</categoryLevel>
- <component>
- <componentInformation>
  <componentName>Administrative</componentName>
  <componentDescription />
  <categoryName>Activities</categoryName>
</componentInformation>
- <dependsOnComponents>
- <componentDependency>
  <componentName>Training Activities</componentName>
- <dependencyCondition>
  <weight>0</weight>
  <expression>true</expression>
  <isSatisfied>true</isSatisfied>
</dependencyCondition>
</componentDependency>
</dependsOnComponents>
- <dependedByComponents>
- <componentDependency>
  <componentName>Admininstration Zone</componentName>
- <dependencyCondition>
  <weight>0</weight>
  <expression>true</expression>
  <isSatisfied>true</isSatisfied>
</dependencyCondition>
</componentDependency>
- <componentDependency>
  <componentName>Unit Exclusive Zone</componentName>
- <dependencyCondition>
  <weight>0</weight>

```

```

        <expression>true</expression>
        <isSatisfied>true</isSatisfied>
    </dependencyCondition>
</componentDependency>
</dependedByComponents>
</component>
- <component>
- <componentInformation>
    <componentName>Training Activities</componentName>
    <componentDescription />
    <categoryName>Activities</categoryName>
</componentInformation>
- <dependsOnComponents>
- <componentDependency>
    <componentName>Mission</componentName>
    - <dependencyCondition>
        <weight>0</weight>
        <expression>true</expression>
        <isSatisfied>true</isSatisfied>
    </dependencyCondition>
    </componentDependency>
</dependsOnComponents>
- <dependedByComponents>
- <componentDependency>
    <componentName>Administrative</componentName>
    - <dependencyCondition>
        <weight>0</weight>
        <expression>true</expression>
        <isSatisfied>true</isSatisfied>
    </dependencyCondition>
    </componentDependency>
</dependedByComponents>
</component>
- <component>
- <componentInformation>
    <componentName>Maintenance Activities</componentName>
    <componentDescription>USARC maintenance activities as
        described in the design guidelines</componentDescription>
    <categoryName>Activities</categoryName>
</componentInformation>
- <dependsOnComponents>
- <componentDependency>
    <componentName>Mission</componentName>
    - <dependencyCondition>
        <weight>0</weight>
        <expression>true</expression>
        <isSatisfied>true</isSatisfied>
    </dependencyCondition>
    </componentDependency>
</dependsOnComponents>
</component>
</category>

```

- <category>
  - <categoryName>Staff</categoryName>
  - <categoryDescription>An Army unit structure is hierarchical and composed of smaller to larger groups. Smallest unit is a "reservee", an USAR personnel who is under training. Each larger group is trained by a ranking officer, which his/her rank changes as the number of reserves changes in the group. During the drill period, depending on the number of reservees (troops) to be trained, the rank structure of the ARU changes. The rank structure and number of troops in an USAR unit is reflected in the design requirements. The maximum number of reserves is defined as total authorized drilling strength (number of reserves) of the largest drill (training) weekend. The largest drill strength is headed by the highest ranked officer in the ARU.</categoryDescription>
  - <categoryLevel>2</categoryLevel>
- <component>
  - <componentInformation>
    - <componentName>Full time staff</componentName>
    - <componentDescription />
    - <categoryName>Staff</categoryName>
  - <constructLocalAs>
    - <constructName>Officers</constructName>
    - <constructDescription />
  - <constructObservedBy>
    - <componentName>Full time staff</componentName>
    - <constructName>Civilians</constructName>
  - <value>
    - <valueComplex>
      - <type>Expression</type>
    - <valueExpression>
      - <value>
        - <valueSimple>
          - <type>integer</type>
          - <integer>6</integer>
          - </valueSimple>
        - </value>
        - <expression>roundup  
(Officers\_OF\_Full\_time\_staff \* 2)  
</expression>
      - <valueReadFrom>
        - <componentName>Full time  
staff</componentName>
        - <constructName>Officers</constructName>
      - <value>
        - <valueComplex>
          - <type>Expression</type>
        - <valueExpression>
          - <value>
            - <valueSimple>
              - <type>integer</type>
              - <integer>3</integer>
              - </valueSimple>
            - </value>

```

<?xml version="1.0" encoding="UTF-8" ?>
- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <xsl:variable name="fo:layout-master-set">
  - <fo:layout-master-set>
    - <fo:simple-page-master master-name="default-page" page-height="11in"
      page-width="8.5in" margin-left="0.6in" margin-right="0.6in">
      <fo:region-body margin-top="0.79in" margin-bottom="0.79in" />
    </fo:simple-page-master>
  </fo:layout-master-set>
</xsl:variable>
- <xsl:template match="/">
  - <fo:root>
    <xsl:copy-of select="$fo:layout-master-set" />
    - <fo:page-sequence master-reference="default-page" initial-page-number="1"
      format="1">
      - <fo:flow flow-name="xsl-region-body">
        - <fo:block>
          - <xsl:for-each select="RabbitProject">
            - <fo:block font-size="18pt" font-weight="bold" space-
              before.optimum="1pt" space-after.optimum="2pt">
              - <fo:block>
                <fo:inline color="#BE3232">RaBBiT Generated
                  Architectural Program View</fo:inline>
              </fo:block>
            </fo:block>
            <xsl:apply-templates select="RabbitProjectInformation" />
          </fo:block>
          <fo:leader leader-pattern="space" />
        </fo:block>
        - <fo:block break-after="page" color="#BE3232" font-
          size="22">
          <fo:leader leader-pattern="space" />
        </fo:block>
        <fo:inline color="#BE3232" font-size="22">Global
          Parameters</fo:inline>
        <xsl:apply-templates select="constructGlobalAs" />
      </fo:block>
      <fo:block break-after="page">
        <fo:leader leader-pattern="space" />
      </fo:block>
      <xsl:apply-templates select="categoryList" />
    </fo:block>
    <fo:block break-after="page" color="#BE3232" font-
      size="22">
      <fo:leader leader-pattern="space" />
    </fo:block>
  </fo:choose>
  - <xsl:when test="component != """>
    <fo:inline color="#BE3232" font-size="22">Global
      (Uncategoriezed) Requirement
      Information</fo:inline>
    - <fo:block>
      <fo:leader leader-pattern="space" />

```

```

        </fo:block>
        <xsl:apply-templates select="component" />
    </xsl:when>
    <xsl:otherwise />
</xsl:choose>
- <fo:block>
    <xsl:text></xsl:text>
</fo:block>
</xsl:for-each>
</fo:block>
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
- <xsl:template match="RabbitProject">
    <xsl:apply-templates />
</xsl:template>
- <xsl:template match="RabbitProjectInformation">
    <fo:inline color="#BE3232" font-size="22">Project Details</fo:inline>
    - <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
        width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
        <fo:table-column />
        - <fo:table-body>
            - <fo:table-row>
                - <fo:table-cell background-color="#3232BE" border-after-
                    color="#3232BE" border-after-width="3pt" border-before-
                    color="#3232BE" border-before-width="3pt" border-end-
                    color="#3232BE" border-end-width="3pt" border-start-
                    color="#3232BE" border-start-width="3pt" padding-start="3pt"
                    padding-end="3pt" padding-before="3pt" padding-after="3pt"
                    display-align="center" text-align="start" border-style="solid"
                    border-width="1pt" border-color="#3232BE">
                - <fo:block>
                    <fo:inline color="#FFFE1" font-size="18" font-
                        weight="bold">Project Information</fo:inline>
                    </fo:block>
                </fo:table-cell>
            </fo:table-row>
        </fo:table-body>
    </fo:table>
    - <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
        after.optimum="2pt">
        <fo:table-column column-width="proportional-column-width(30)" />
        - <xsl:for-each select="project">
            <fo:table-column />
        </xsl:for-each>
        - <fo:table-body>
            - <fo:table-row>
                - <fo:table-cell border-after-color="#3232BE" border-before-
                    color="#3232BE" border-before-style="solid" border-before-
                    width="3pt" border-end-color="#3232BE" border-start-
                    color="#3232BE" font-size="12" padding-end="5pt" padding-
                    start="5pt" background-color="#FFFE1" text-align="right"

```

```

width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Name:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectName">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
  <fo:block>Location:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectLocation">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFFE1" text-align="right"

```

```

width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>ID:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectID">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-after-style="solid"
    border-after-width="3pt" border-before-color="#475874" border-
    before-style="solid" border-before-width="1pt" border-end-
    color="#475874" border-start-color="#475874" font-size="12"
    padding-end="5pt" padding-start="5pt" background-
    color="#FFFE1" text-align="right" width="30%" padding-
    before="3pt" padding-after="3pt" display-align="center" border-
    style="solid" border-width="1pt" border-color="white">
    <fo:block>Description:</fo:block>
  </fo:table-cell>
  - <xsl:for-each select="project">
    - <fo:table-cell border-after-color="#475874" border-after-
      style="solid" border-after-width="3pt" border-before-
      color="#475874" border-before-style="solid" border-before-
      width="1pt" border-end-color="#475874" border-start-
      color="#475874" font-size="12" padding-end="5pt" padding-
      start="5pt" padding-before="3pt" padding-after="3pt" display-
      align="center" text-align="start" border-style="solid" border-
      width="1pt" border-color="white">
    - <fo:block>
      <xsl:if test="projectDesc = "">None available</xsl:if>
      - <xsl:for-each select="projectDesc">
        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
  width="100%" space-before.optimum="1pt" space-after.optimum="2pt">

```

```

</fo:table-column />
- <fo:table-body>
- <fo:table-row>
  - <fo:table-cell background-color="#3232BE" border-after-
    color="#3232BE" border-after-width="3pt" border-before-
    color="#3232BE" border-before-width="3pt" border-end-
    color="#3232BE" border-end-width="3pt" border-start-
    color="#3232BE" border-start-width="3pt" padding-start="3pt"
    padding-end="3pt" padding-before="3pt" padding-after="3pt"
    display-align="center" text-align="start" border-style="solid"
    border-width="1pt" border-color="#3232BE">
  - <fo:block>
    <fo:inline color="#FFFE1" font-size="18" font-
      weight="bold">Client Information</fo:inline>
    </fo:block>
  </fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
  after.optimum="2pt">
  <fo:table-column column-width="proportional-column-width(30)" />
- <xsl:for-each select="client">
  <fo:table-column />
</xsl:for-each>
- <fo:table-body>
- <fo:table-row>
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Name:</fo:block>
  </fo:table-cell>
- <xsl:for-each select="client">
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="clientName">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>

```



```

- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Contact Name:</fo:block>
</fo:table-cell>
- <xsl:for-each select="client">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="contactName">
  <xsl:apply-templates />
</xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Phone:</fo:block>
</fo:table-cell>
- <xsl:for-each select="client">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="contactPhone">
  <xsl:apply-templates />
</xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>

```

- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" background-color="#FFFE1" text-align="right" width="30%" padding-before="3pt" padding-after="3pt" display-align="center" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>Email:</fo:block>
   
</fo:table-cell>
  - <xsl:for-each select="client">
    - <fo:table-cell border-after-color="#475874" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" padding-before="3pt" padding-after="3pt" display-align="center" text-align="start" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>
      - <xsl:for-each select="contactEmail">
   
<xsl:apply-templates />
   
</xsl:for-each>  
</fo:block>  
</fo:table-cell>  
</xsl:for-each>

  
</fo:table-row>
 
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-after-style="solid" border-after-width="3pt" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" background-color="#FFFE1" text-align="right" width="30%" padding-before="3pt" padding-after="3pt" display-align="center" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>Address:</fo:block>
   
</fo:table-cell>
- <xsl:for-each select="client">
  - <fo:table-cell border-after-color="#475874" border-after-style="solid" border-after-width="3pt" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" padding-before="3pt" padding-after="3pt" display-align="center" text-align="start" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>
    - <xsl:for-each select="contactAddress">
   
<xsl:apply-templates />
   
</xsl:for-each>  
</fo:block>  
</fo:table-cell>  
</xsl:for-each>

```

        </fo:table-row>
    </fo:table-body>
</fo:table>
- <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
    width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
    <fo:table-column />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell background-color="#3232BE" border-after-
    color="#3232BE" border-after-width="3pt" border-before-
    color="#3232BE" border-before-width="3pt" border-end-
    color="#3232BE" border-end-width="3pt" border-start-
    color="#3232BE" border-start-width="3pt" padding-start="3pt"
    padding-end="3pt" padding-before="3pt" padding-after="3pt"
    display-align="center" text-align="start" border-style="solid"
    border-width="1pt" border-color="#3232BE">
- <fo:block>
    <fo:inline color="#FFFE1" font-size="18" font-
        weight="bold">Version Information</fo:inline>
    </fo:block>
    </fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
    after.optimum="2pt">
    <fo:table-column column-width="proportional-column-width(30)" />
- <xsl:for-each select="versionInformation">
    <fo:table-column />
</xsl:for-each>
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Generation Date:</fo:block>
    </fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="date">
    <xsl:apply-templates />

```

```

        </xsl:for-each>
    </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Version Number:</fo:block>
</fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="versionNumber">
  <xsl:apply-templates />
  </xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-after-style="solid"
  border-after-width="3pt" border-before-color="#475874" border-
  before-style="solid" border-before-width="1pt" border-end-
  color="#475874" border-start-color="#475874" font-size="12"
  padding-end="5pt" padding-start="5pt" background-
  color="#FFFE1" text-align="right" width="30%" padding-
  before="3pt" padding-after="3pt" display-align="center" border-
  style="solid" border-width="1pt" border-color="white">
  <fo:block>Description:</fo:block>
</fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#475874" border-after-
  style="solid" border-after-width="3pt" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="versionDescription">

```

```

        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
</fo:table-body>
</fo:table>
</xsl:template>
- <xsl:template match="category">
- <fo:table color="#BE3232" font-size="18" background-color="#4758A7"
  width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
  <fo:table-column />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell background-color="#3232BE" border-after-
  color="#4758A7" border-after-width="3pt" border-before-
  color="#4758A7" border-before-width="3pt" border-end-
  color="#4758A7" border-end-width="3pt" border-start-
  color="#4758A7" border-start-width="3pt" font-size="12" padding-
  start="3pt" padding-end="3pt" padding-before="3pt" padding-
  after="3pt" display-align="center" text-align="start" border-
  style="solid" border-width="1pt" border-color="#4758A7">
- <fo:block>
  <fo:inline color="#FFFFE1" font-
    size="18">Category: </fo:inline>
  - <xsl:for-each select="categoryName">
    - <fo:inline color="#FFFFE1" font-size="18" font-
      style="normal" font-weight="bold">
      <xsl:apply-templates />
    </fo:inline>
  </xsl:for-each>
  </fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table background-color="#FFFFE1" padding="2" width="100%" space-
  before.optimum="1pt" space-after.optimum="2pt">
  <fo:table-column column-width="proportional-column-width(30)" />
  <fo:table-column column-width="proportional-column-width(70)" />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell border-after-color="#4758A7" border-after-style="solid"
  border-after-width="1pt" border-before-color="#4758A7" border-
  end-color="#4758A7" border-start-color="#4758A7" font-size="12"
  text-align="right" width="30%" padding-start="3pt" padding-
  end="3pt" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="#FFFFE1">
  <fo:block>Description:</fo:block>
</fo:table-cell>
- <fo:table-cell border-after-color="#4758A7" border-after-style="solid"

```

```

<?xml version="1.0" encoding="UTF-8" ?>
- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <xsl:variable name="fo:layout-master-set">
  - <fo:layout-master-set>
    - <fo:simple-page-master master-name="default-page" page-height="11in"
      page-width="8.5in" margin-left="0.6in" margin-right="0.6in">
      <fo:region-body margin-top="0.79in" margin-bottom="0.79in" />
    </fo:simple-page-master>
  </fo:layout-master-set>
</xsl:variable>
- <xsl:template match="/">
  - <fo:root>
    <xsl:copy-of select="$fo:layout-master-set" />
    - <fo:page-sequence master-reference="default-page" initial-page-number="1"
      format="1">
      - <fo:flow flow-name="xsl-region-body">
        - <fo:block>
          - <xsl:for-each select="RabbitProject">
            - <fo:block font-size="18pt" font-weight="bold" space-
              before.optimum="1pt" space-after.optimum="2pt">
              - <fo:block>
                <fo:inline color="#BE3232">RaBBiT Generated
                  Architectural Program View</fo:inline>
              </fo:block>
            </fo:block>
            <xsl:apply-templates select="RabbitProjectInformation" />
          - <fo:block>
            <fo:leader leader-pattern="space" />
          </fo:block>
          - <fo:block break-after="page" color="#BE3232" font-
            size="22">
            <fo:leader leader-pattern="space" />
          </fo:block>
          <fo:inline color="#BE3232" font-size="22">Global
            Parameters</fo:inline>
          <xsl:apply-templates select="constructGlobalAs" />
          - <fo:block break-after="page">
            <fo:leader leader-pattern="space" />
          </fo:block>
          <xsl:apply-templates select="categoryList" />
          - <fo:block break-after="page" color="#BE3232" font-
            size="22">
            <fo:leader leader-pattern="space" />
          </fo:block>
          - <xsl:choose>
            - <xsl:when test="component != """>
              <fo:inline color="#BE3232" font-size="22">Global
                (Uncategoriezed) Requirement
                Information</fo:inline>
            - <fo:block>
              <fo:leader leader-pattern="space" />

```

```

        </fo:block>
        <xsl:apply-templates select="component" />
    </xsl:when>
    <xsl:otherwise />
</xsl:choose>
- <fo:block>
    <xsl:text></xsl:text>
</fo:block>
</xsl:for-each>
</fo:block>
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
- <xsl:template match="RabbitProject">
    <xsl:apply-templates />
</xsl:template>
- <xsl:template match="RabbitProjectInformation">
    <fo:inline color="#BE3232" font-size="22">Project Details</fo:inline>
    - <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
        width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
        <fo:table-column />
        - <fo:table-body>
            - <fo:table-row>
                - <fo:table-cell background-color="#3232BE" border-after-
                    color="#3232BE" border-after-width="3pt" border-before-
                    color="#3232BE" border-before-width="3pt" border-end-
                    color="#3232BE" border-end-width="3pt" border-start-
                    color="#3232BE" border-start-width="3pt" padding-start="3pt"
                    padding-end="3pt" padding-before="3pt" padding-after="3pt"
                    display-align="center" text-align="start" border-style="solid"
                    border-width="1pt" border-color="#3232BE">
                - <fo:block>
                    <fo:inline color="#FFFE1" font-size="18" font-
                        weight="bold">Project Information</fo:inline>
                    </fo:block>
                </fo:table-cell>
            </fo:table-row>
        </fo:table-body>
    </fo:table>
    - <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
        after.optimum="2pt">
        <fo:table-column column-width="proportional-column-width(30)" />
        - <xsl:for-each select="project">
            <fo:table-column />
        </xsl:for-each>
        - <fo:table-body>
            - <fo:table-row>
                - <fo:table-cell border-after-color="#3232BE" border-before-
                    color="#3232BE" border-before-style="solid" border-before-
                    width="3pt" border-end-color="#3232BE" border-start-
                    color="#3232BE" font-size="12" padding-end="5pt" padding-
                    start="5pt" background-color="#FFFE1" text-align="right"

```



```

width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Name:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectName">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Location:</fo:block>
  </fo:table-cell>
  - <xsl:for-each select="project">
    - <fo:table-cell border-after-color="#475874" border-before-
      color="#475874" border-before-style="solid" border-before-
      width="1pt" border-end-color="#475874" border-start-
      color="#475874" font-size="12" padding-end="5pt" padding-
      start="5pt" padding-before="3pt" padding-after="3pt" display-
      align="center" text-align="start" border-style="solid" border-
      width="1pt" border-color="white">
    - <fo:block>
      - <xsl:for-each select="projectLocation">
        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"

```



```

width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>ID:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectID">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-after-style="solid"
    border-after-width="3pt" border-before-color="#475874" border-
    before-style="solid" border-before-width="1pt" border-end-
    color="#475874" border-start-color="#475874" font-size="12"
    padding-end="5pt" padding-start="5pt" background-
    color="#FFFE1" text-align="right" width="30%" padding-
    before="3pt" padding-after="3pt" display-align="center" border-
    style="solid" border-width="1pt" border-color="white">
    <fo:block>Description:</fo:block>
  </fo:table-cell>
  - <xsl:for-each select="project">
    - <fo:table-cell border-after-color="#475874" border-after-
      style="solid" border-after-width="3pt" border-before-
      color="#475874" border-before-style="solid" border-before-
      width="1pt" border-end-color="#475874" border-start-
      color="#475874" font-size="12" padding-end="5pt" padding-
      start="5pt" padding-before="3pt" padding-after="3pt" display-
      align="center" text-align="start" border-style="solid" border-
      width="1pt" border-color="white">
    - <fo:block>
      <xsl:if test="projectDesc = "">None available</xsl:if>
      - <xsl:for-each select="projectDesc">
        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
  width="100%" space-before.optimum="1pt" space-after.optimum="2pt">

```

```

</fo:table-column />
- <fo:table-body>
- <fo:table-row>
  - <fo:table-cell background-color="#3232BE" border-after-
    color="#3232BE" border-after-width="3pt" border-before-
    color="#3232BE" border-before-width="3pt" border-end-
    color="#3232BE" border-end-width="3pt" border-start-
    color="#3232BE" border-start-width="3pt" padding-start="3pt"
    padding-end="3pt" padding-before="3pt" padding-after="3pt"
    display-align="center" text-align="start" border-style="solid"
    border-width="1pt" border-color="#3232BE">
  - <fo:block>
    <fo:inline color="#FFFE1" font-size="18" font-
      weight="bold">Client Information</fo:inline>
    </fo:block>
  </fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
  after.optimum="2pt">
  <fo:table-column column-width="proportional-column-width(30)" />
- <xsl:for-each select="client">
  <fo:table-column />
</xsl:for-each>
- <fo:table-body>
- <fo:table-row>
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Name:</fo:block>
  </fo:table-cell>
- <xsl:for-each select="client">
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="clientName">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>

```

```

- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Contact Name:</fo:block>
</fo:table-cell>
- <xsl:for-each select="client">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="contactName">
  <xsl:apply-templates />
  </xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Phone:</fo:block>
</fo:table-cell>
- <xsl:for-each select="client">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="contactPhone">
  <xsl:apply-templates />
  </xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>

```

- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" background-color="#FFFE1" text-align="right" width="30%" padding-before="3pt" padding-after="3pt" display-align="center" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>**Email:**</fo:block>
   
</fo:table-cell>
  - <xsl:for-each select="client">
    - <fo:table-cell border-after-color="#475874" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" padding-before="3pt" padding-after="3pt" display-align="center" text-align="start" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>
      - <xsl:for-each select="contactEmail">
   
<xsl:apply-templates />
   
</xsl:for-each>  
</fo:block>  
</fo:table-cell>  
</xsl:for-each>
- </fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-after-style="solid" border-after-width="3pt" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" background-color="#FFFE1" text-align="right" width="30%" padding-before="3pt" padding-after="3pt" display-align="center" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>**Address:**</fo:block>
   
</fo:table-cell>
  - <xsl:for-each select="client">
    - <fo:table-cell border-after-color="#475874" border-after-style="solid" border-after-width="3pt" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" padding-before="3pt" padding-after="3pt" display-align="center" text-align="start" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>
      - <xsl:for-each select="contactAddress">
   
<xsl:apply-templates />
   
</xsl:for-each>  
</fo:block>  
</fo:table-cell>  
</xsl:for-each>

```

        </fo:table-row>
    </fo:table-body>
</fo:table>
- <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
    width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
    <fo:table-column />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell background-color="#3232BE" border-after-
    color="#3232BE" border-after-width="3pt" border-before-
    color="#3232BE" border-before-width="3pt" border-end-
    color="#3232BE" border-end-width="3pt" border-start-
    color="#3232BE" border-start-width="3pt" padding-start="3pt"
    padding-end="3pt" padding-before="3pt" padding-after="3pt"
    display-align="center" text-align="start" border-style="solid"
    border-width="1pt" border-color="#3232BE">
- <fo:block>
    <fo:inline color="#FFFE1" font-size="18" font-
        weight="bold">Version Information</fo:inline>
    </fo:block>
    </fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
    after.optimum="2pt">
    <fo:table-column column-width="proportional-column-width(30)" />
- <xsl:for-each select="versionInformation">
    <fo:table-column />
</xsl:for-each>
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Generation Date:</fo:block>
    </fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="date">
    <xsl:apply-templates />

```

```

        </xsl:for-each>
    </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Version Number:</fo:block>
</fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="versionNumber">
  <xsl:apply-templates />
  </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-after-style="solid"
  border-after-width="3pt" border-before-color="#475874" border-
  before-style="solid" border-before-width="1pt" border-end-
  color="#475874" border-start-color="#475874" font-size="12"
  padding-end="5pt" padding-start="5pt" background-
  color="#FFFE1" text-align="right" width="30%" padding-
  before="3pt" padding-after="3pt" display-align="center" border-
  style="solid" border-width="1pt" border-color="white">
  <fo:block>Description:</fo:block>
</fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#475874" border-after-
  style="solid" border-after-width="3pt" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="versionDescription">

```

```

        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
</fo:table-body>
</fo:table>
</xsl:template>
- <xsl:template match="category">
- <fo:table color="#BE3232" font-size="18" background-color="#4758A7"
  width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
  <fo:table-column />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell background-color="#3232BE" border-after-
  color="#4758A7" border-after-width="3pt" border-before-
  color="#4758A7" border-before-width="3pt" border-end-
  color="#4758A7" border-end-width="3pt" border-start-
  color="#4758A7" border-start-width="3pt" font-size="12" padding-
  start="3pt" padding-end="3pt" padding-before="3pt" padding-
  after="3pt" display-align="center" text-align="start" border-
  style="solid" border-width="1pt" border-color="#4758A7">
- <fo:block>
  <fo:inline color="#FFFFE1" font-
    size="18">Category: </fo:inline>
  - <xsl:for-each select="categoryName">
    - <fo:inline color="#FFFFE1" font-size="18" font-
      style="normal" font-weight="bold">
      <xsl:apply-templates />
    </fo:inline>
  </xsl:for-each>
  </fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table background-color="#FFFFE1" padding="2" width="100%" space-
  before.optimum="1pt" space-after.optimum="2pt">
  <fo:table-column column-width="proportional-column-width(30)" />
  <fo:table-column column-width="proportional-column-width(70)" />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell border-after-color="#4758A7" border-after-style="solid"
  border-after-width="1pt" border-before-color="#4758A7" border-
  end-color="#4758A7" border-start-color="#4758A7" font-size="12"
  text-align="right" width="30%" padding-start="3pt" padding-
  end="3pt" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="#FFFFE1">
  <fo:block>Description:</fo:block>
</fo:table-cell>
- <fo:table-cell border-after-color="#4758A7" border-after-style="solid"

```



```

<?xml version="1.0" encoding="UTF-8" ?>
- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <xsl:variable name="fo:layout-master-set">
  - <fo:layout-master-set>
    - <fo:simple-page-master master-name="default-page" page-height="11in"
      page-width="8.5in" margin-left="0.6in" margin-right="0.6in">
      <fo:region-body margin-top="0.79in" margin-bottom="0.79in" />
    </fo:simple-page-master>
  </fo:layout-master-set>
</xsl:variable>
- <xsl:template match="/">
  - <fo:root>
    <xsl:copy-of select="$fo:layout-master-set" />
    - <fo:page-sequence master-reference="default-page" initial-page-number="1"
      format="1">
      - <fo:flow flow-name="xsl-region-body">
        - <fo:block>
          - <xsl:for-each select="RabbitProject">
            - <fo:block font-size="18pt" font-weight="bold" space-
              before.optimum="1pt" space-after.optimum="2pt">
              - <fo:block>
                <fo:inline color="#BE3232">RaBBiT Generated
                  Architectural Program View</fo:inline>
              </fo:block>
            </fo:block>
            <xsl:apply-templates select="RabbitProjectInformation" />
          </fo:block>
          <fo:leader leader-pattern="space" />
        </fo:block>
        - <fo:block break-after="page" color="#BE3232" font-
          size="22">
          <fo:leader leader-pattern="space" />
        </fo:block>
        <fo:inline color="#BE3232" font-size="22">Global
          Parameters</fo:inline>
        <xsl:apply-templates select="constructGlobalAs" />
      </fo:block>
      - <fo:block break-after="page">
        <fo:leader leader-pattern="space" />
      </fo:block>
      <xsl:apply-templates select="categoryList" />
      - <fo:block break-after="page" color="#BE3232" font-
        size="22">
        <fo:leader leader-pattern="space" />
      </fo:block>
    </fo:choose>
    - <xsl:when test="component != """>
      <fo:inline color="#BE3232" font-size="22">Global
        (Uncategoriezed) Requirement
        Information</fo:inline>
    </fo:block>
    <fo:leader leader-pattern="space" />
  </fo:root>
</xsl:template>
</xsl:stylesheet>

```



```

        </fo:block>
        <xsl:apply-templates select="component" />
    </xsl:when>
    <xsl:otherwise />
</xsl:choose>
- <fo:block>
    <xsl:text></xsl:text>
</fo:block>
</xsl:for-each>
</fo:block>
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
- <xsl:template match="RabbitProject">
    <xsl:apply-templates />
</xsl:template>
- <xsl:template match="RabbitProjectInformation">
    <fo:inline color="#BE3232" font-size="22">Project Details</fo:inline>
    - <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
        width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
        <fo:table-column />
        - <fo:table-body>
            - <fo:table-row>
                - <fo:table-cell background-color="#3232BE" border-after-
                    color="#3232BE" border-after-width="3pt" border-before-
                    color="#3232BE" border-before-width="3pt" border-end-
                    color="#3232BE" border-end-width="3pt" border-start-
                    color="#3232BE" border-start-width="3pt" padding-start="3pt"
                    padding-end="3pt" padding-before="3pt" padding-after="3pt"
                    display-align="center" text-align="start" border-style="solid"
                    border-width="1pt" border-color="#3232BE">
                - <fo:block>
                    <fo:inline color="#FFFE1" font-size="18" font-
                        weight="bold">Project Information</fo:inline>
                    </fo:block>
                </fo:table-cell>
            </fo:table-row>
        </fo:table-body>
    </fo:table>
    - <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
        after.optimum="2pt">
        <fo:table-column column-width="proportional-column-width(30)" />
        - <xsl:for-each select="project">
            <fo:table-column />
        </xsl:for-each>
        - <fo:table-body>
            - <fo:table-row>
                - <fo:table-cell border-after-color="#3232BE" border-before-
                    color="#3232BE" border-before-style="solid" border-before-
                    width="3pt" border-end-color="#3232BE" border-start-
                    color="#3232BE" font-size="12" padding-end="5pt" padding-
                    start="5pt" background-color="#FFFE1" text-align="right"

```

```

width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Name:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectName">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Location:</fo:block>
  </fo:table-cell>
  - <xsl:for-each select="project">
    - <fo:table-cell border-after-color="#475874" border-before-
      color="#475874" border-before-style="solid" border-before-
      width="1pt" border-end-color="#475874" border-start-
      color="#475874" font-size="12" padding-end="5pt" padding-
      start="5pt" padding-before="3pt" padding-after="3pt" display-
      align="center" text-align="start" border-style="solid" border-
      width="1pt" border-color="white">
    - <fo:block>
      - <xsl:for-each select="projectLocation">
        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"

```

```

width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>ID:</fo:block>
</fo:table-cell>
- <xsl:for-each select="project">
  - <fo:table-cell border-after-color="#475874" border-before-
    color="#475874" border-before-style="solid" border-before-
    width="1pt" border-end-color="#475874" border-start-
    color="#475874" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="projectID">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-after-style="solid"
    border-after-width="3pt" border-before-color="#475874" border-
    before-style="solid" border-before-width="1pt" border-end-
    color="#475874" border-start-color="#475874" font-size="12"
    padding-end="5pt" padding-start="5pt" background-
    color="#FFFE1" text-align="right" width="30%" padding-
    before="3pt" padding-after="3pt" display-align="center" border-
    style="solid" border-width="1pt" border-color="white">
    <fo:block>Description:</fo:block>
  </fo:table-cell>
  - <xsl:for-each select="project">
    - <fo:table-cell border-after-color="#475874" border-after-
      style="solid" border-after-width="3pt" border-before-
      color="#475874" border-before-style="solid" border-before-
      width="1pt" border-end-color="#475874" border-start-
      color="#475874" font-size="12" padding-end="5pt" padding-
      start="5pt" padding-before="3pt" padding-after="3pt" display-
      align="center" text-align="start" border-style="solid" border-
      width="1pt" border-color="white">
    - <fo:block>
      <xsl:if test="projectDesc = "">None available</xsl:if>
      - <xsl:for-each select="projectDesc">
        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
  width="100%" space-before.optimum="1pt" space-after.optimum="2pt">

```

```

</fo:table-column />
- <fo:table-body>
- <fo:table-row>
  - <fo:table-cell background-color="#3232BE" border-after-
    color="#3232BE" border-after-width="3pt" border-before-
    color="#3232BE" border-before-width="3pt" border-end-
    color="#3232BE" border-end-width="3pt" border-start-
    color="#3232BE" border-start-width="3pt" padding-start="3pt"
    padding-end="3pt" padding-before="3pt" padding-after="3pt"
    display-align="center" text-align="start" border-style="solid"
    border-width="1pt" border-color="#3232BE">
  - <fo:block>
    <fo:inline color="#FFFE1" font-size="18" font-
      weight="bold">Client Information</fo:inline>
    </fo:block>
  </fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
  after.optimum="2pt">
  <fo:table-column column-width="proportional-column-width(30)" />
- <xsl:for-each select="client">
  <fo:table-column />
</xsl:for-each>
- <fo:table-body>
- <fo:table-row>
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Name:</fo:block>
  </fo:table-cell>
- <xsl:for-each select="client">
  - <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
  - <fo:block>
    - <xsl:for-each select="clientName">
      <xsl:apply-templates />
    </xsl:for-each>
  </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>

```

```

- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Contact Name:</fo:block>
</fo:table-cell>
- <xsl:for-each select="client">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="contactName">
  <xsl:apply-templates />
</xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Phone:</fo:block>
</fo:table-cell>
- <xsl:for-each select="client">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="contactPhone">
  <xsl:apply-templates />
</xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>

```

- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" background-color="#FFFE1" text-align="right" width="30%" padding-before="3pt" padding-after="3pt" display-align="center" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>**Email:**</fo:block>
   
</fo:table-cell>
  - <xsl:for-each select="client">
    - <fo:table-cell border-after-color="#475874" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" padding-before="3pt" padding-after="3pt" display-align="center" text-align="start" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>
      - <xsl:for-each select="contactEmail">
   
<xsl:apply-templates />
   
</xsl:for-each>  
</fo:block>  
</fo:table-cell>  
</xsl:for-each>
- </fo:table-row>
- <fo:table-row>
  - <fo:table-cell border-after-color="#475874" border-after-style="solid" border-after-width="3pt" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" background-color="#FFFE1" text-align="right" width="30%" padding-before="3pt" padding-after="3pt" display-align="center" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>**Address:**</fo:block>
   
</fo:table-cell>
  - <xsl:for-each select="client">
    - <fo:table-cell border-after-color="#475874" border-after-style="solid" border-after-width="3pt" border-before-color="#475874" border-before-style="solid" border-before-width="1pt" border-end-color="#475874" border-start-color="#475874" font-size="12" padding-end="5pt" padding-start="5pt" padding-before="3pt" padding-after="3pt" display-align="center" text-align="start" border-style="solid" border-width="1pt" border-color="white">
   
<fo:block>
      - <xsl:for-each select="contactAddress">
   
<xsl:apply-templates />
   
</xsl:for-each>  
</fo:block>  
</fo:table-cell>  
</xsl:for-each>

```

        </fo:table-row>
    </fo:table-body>
</fo:table>
- <fo:table color="#BE3232" font-size="18" background-color="#3232BE"
    width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
    <fo:table-column />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell background-color="#3232BE" border-after-
    color="#3232BE" border-after-width="3pt" border-before-
    color="#3232BE" border-before-width="3pt" border-end-
    color="#3232BE" border-end-width="3pt" border-start-
    color="#3232BE" border-start-width="3pt" padding-start="3pt"
    padding-end="3pt" padding-before="3pt" padding-after="3pt"
    display-align="center" text-align="start" border-style="solid"
    border-width="1pt" border-color="#3232BE">
- <fo:block>
    <fo:inline color="#FFFE1" font-size="18" font-
        weight="bold">Version Information</fo:inline>
    </fo:block>
    </fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table padding="2" width="100%" space-before.optimum="1pt" space-
    after.optimum="2pt">
    <fo:table-column column-width="proportional-column-width(30)" />
- <xsl:for-each select="versionInformation">
    <fo:table-column />
</xsl:for-each>
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" background-color="#FFFE1" text-align="right"
    width="30%" padding-before="3pt" padding-after="3pt" display-
    align="center" border-style="solid" border-width="1pt" border-
    color="white">
    <fo:block>Generation Date:</fo:block>
    </fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#3232BE" border-before-
    color="#3232BE" border-before-style="solid" border-before-
    width="3pt" border-end-color="#3232BE" border-start-
    color="#3232BE" font-size="12" padding-end="5pt" padding-
    start="5pt" padding-before="3pt" padding-after="3pt" display-
    align="center" text-align="start" border-style="solid" border-
    width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="date">
    <xsl:apply-templates />

```



```

        </xsl:for-each>
    </fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" background-color="#FFFE1" text-align="right"
  width="30%" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="white">
  <fo:block>Version Number:</fo:block>
</fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#475874" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="versionNumber">
  <xsl:apply-templates />
  </xsl:for-each>
</fo:block>
</fo:table-cell>
</xsl:for-each>
</fo:table-row>
- <fo:table-row>
- <fo:table-cell border-after-color="#475874" border-after-style="solid"
  border-after-width="3pt" border-before-color="#475874" border-
  before-style="solid" border-before-width="1pt" border-end-
  color="#475874" border-start-color="#475874" font-size="12"
  padding-end="5pt" padding-start="5pt" background-
  color="#FFFE1" text-align="right" width="30%" padding-
  before="3pt" padding-after="3pt" display-align="center" border-
  style="solid" border-width="1pt" border-color="white">
  <fo:block>Description:</fo:block>
</fo:table-cell>
- <xsl:for-each select="versionInformation">
- <fo:table-cell border-after-color="#475874" border-after-
  style="solid" border-after-width="3pt" border-before-
  color="#475874" border-before-style="solid" border-before-
  width="1pt" border-end-color="#475874" border-start-
  color="#475874" font-size="12" padding-end="5pt" padding-
  start="5pt" padding-before="3pt" padding-after="3pt" display-
  align="center" text-align="start" border-style="solid" border-
  width="1pt" border-color="white">
- <fo:block>
- <xsl:for-each select="versionDescription">

```



```

        <xsl:apply-templates />
      </xsl:for-each>
    </fo:block>
  </fo:table-cell>
</xsl:for-each>
</fo:table-row>
</fo:table-body>
</fo:table>
</xsl:template>
- <xsl:template match="category">
- <fo:table color="#BE3232" font-size="18" background-color="#4758A7"
  width="100%" space-before.optimum="1pt" space-after.optimum="2pt">
  <fo:table-column />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell background-color="#3232BE" border-after-
  color="#4758A7" border-after-width="3pt" border-before-
  color="#4758A7" border-before-width="3pt" border-end-
  color="#4758A7" border-end-width="3pt" border-start-
  color="#4758A7" border-start-width="3pt" font-size="12" padding-
  start="3pt" padding-end="3pt" padding-before="3pt" padding-
  after="3pt" display-align="center" text-align="start" border-
  style="solid" border-width="1pt" border-color="#4758A7">
- <fo:block>
  <fo:inline color="#FFFFE1" font-
    size="18">Category: </fo:inline>
  - <xsl:for-each select="categoryName">
    - <fo:inline color="#FFFFE1" font-size="18" font-
      style="normal" font-weight="bold">
      <xsl:apply-templates />
    </fo:inline>
  </xsl:for-each>
  </fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
- <fo:table background-color="#FFFFE1" padding="2" width="100%" space-
  before.optimum="1pt" space-after.optimum="2pt">
  <fo:table-column column-width="proportional-column-width(30)" />
  <fo:table-column column-width="proportional-column-width(70)" />
- <fo:table-body>
- <fo:table-row>
- <fo:table-cell border-after-color="#4758A7" border-after-style="solid"
  border-after-width="1pt" border-before-color="#4758A7" border-
  end-color="#4758A7" border-start-color="#4758A7" font-size="12"
  text-align="right" width="30%" padding-start="3pt" padding-
  end="3pt" padding-before="3pt" padding-after="3pt" display-
  align="center" border-style="solid" border-width="1pt" border-
  color="#FFFFE1">
  <fo:block>Description:</fo:block>
</fo:table-cell>
- <fo:table-cell border-after-color="#4758A7" border-after-style="solid"

```

---

## Appendix C: OO-Model and RaBBiT

---

The content of this appendix is included in the attached CD

1. Class structure in UML (RabbitDomainModel.mdl)
2. RaBBiT API Documentation (.<CD>:\JRaBBiT\_API Documentation\index-all.html)
3. Program Schema ( <CD>:\RaBBiTSchema\RabbitXMLSchemaDoc.html)
4. Program schema file in XML (RabbitXMLSchema.xsd)
5. RaBBiT Executables ( jrabbitw.exe, jrabbit.exe, jrabbit.jar)
6. Sample architectural programming knowledge model for USARCs (usarc.rbt)
7. Sample requirement information category model (Demo\_Categories.cat)
8. Sample XSL style sheet file (RabbitXSLSample.xsl)