Integrating Housing Design and Case-Based Reasoning

Ji-Hyun Lee

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

School of Architecture and
Institute for Complex Engineered Systems (ICES)
Carnegie Mellon University
Fall 2002

Thesis Committee:

Professor Ulrich Flemming (Chair)
School of Architecture and
Institute for Complex Engineered Systems (ICES)
Carnegie Mellon University

Professor James H. Garrett, Jr.
Department of Civil and Environmental Engineering and
Institute for Complex Engineered Systems (ICES)
Carnegie Mellon University

Professor Stephen R. Lee
School of Architecture
Carnegie Mellon University

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

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

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

Ji-Hyun Lee

Table of Contents

List of Figures

List of Tables

Acknowledgement

I was worried about my life—including my studies—before I embarked on a long journey to study in the US. But I have met a lot of good people along the way, and it becomes difficult for me to include everyone who made my life rich. I will try to mention most of them.

First, I would like to express my sincere gratitude to my advisor, Professor Ulrich Flemming, for his confidence and patience in me, his clarity of thinking, and his continuous sources of research. I am also grateful to my thesis committee members: to Professor James Garrett for his precise critiques of the *what*, *why*, and *how* of my research; and to Professor Stephen Lee for his support and key discussions at some critical points, especially in developing housing design scenarios.

I am indebted to my former advisor in Korea, Professor Bokcha Yoon, not only for guiding my research during almost 6 years when I was a graduate student in Yonsei University, but also for introducing her older sister, who lived in Pittsburgh (I feel so sad because she passed away this year), to me to help my settling down and to take care of me while I stayed in Pittsburgh.

I cannot think about CMU without remembering my colleagues. I would like to specially thank Robert Ries, Dan Greenwood, and others who read various drafts of this thesis, pointed out mistakes, and suggested better explanations.

Working within the SEED project has given me a priceless learning experience. Even though many talented students had already graduated when I joined the SEED project, I still had a chance to meet some of them and would like to specially mention Shang-chia Chiou, Zeyno Aygen, and Sheng-Fen (Nik) Chien. I have learned a lot from the SEED project members, Wen-Jaw (Jonah) Tsai and Michael Cumming.

I also have friends who came from the all different places in the world and have given me precious opportunities to learn about their cultures and countries. I am glad that my former roommate and classmate Jayada Boonyakiat is able to graduate with me almost at the same time. I am also glad to have met my other classmates, Hesham Eissa and Halil Erhan (and his family). I thank all my office mates and neighbors—we had lunch and dinner together and encouraged each other: Mustafa-Emre Ilal, Nabeel Koshak, Ye Zhang, Seongju Chang, Kuhn Park, Kristie Mertz, Prechaya Mahattanatawe, Zhengchun Mo, Heakyung Yoon, Min Oh, and Sang-Hoon Lee.

I was surrounded by many great people and friends outside of the campus. I most likely have met them in the Korean United Presbyterian Church of Pittsburgh. I would especially like to say thanks to Jeesook Lee for her generous offer to stay in her house twice during trips back to and forth from the US. I also want to thank Pastor Stephen Kim and Hana Yoon in New Jersey for their endless concerns and support, especially when I was sick during the last year.

I want to acknowledge the Korean Government and the Institute for Complex Engineered Systems (ICES) for supporting me up to the end so that I was able to finish this degree in a very conducive environment.

From the bottom of my heart, I thank my parents—in memory of my father Keun-Sup Lee, my mother Daija Shin, as well as my brother Sang-Hun Lee for their unconditional support, love, and faith in many phases of my life. Without their support, I would not have been able to complete the whole course.

Finally, thanks God! I can finish my dissertation and be ready to enter a new phase of my life. Guide me in your truth and teach me.

“Then you will know the truth, and the truth will set you free.”

Abstract

Expert designers typically refer to and re-use past solutions for recurring design problems. Case-based design (CBD) attempts to transfer this natural design reasoning process to computer-aided design using artificial intelligence (AI) methods and databases. The housing design domain is particularly suited for applying the CBD approach because the traditional method of home design already makes extensive use of precedents and solutions are highly standardized in that industry, at least in the U.S. A generally accessible and continuously updated database of case could also alleviate some of the structural problems that have plagued this industry and stood in the way of innovation.

This thesis focuses on a general framework and computational environment that supports the schematic design phase for housing through a CBD capability. It describes formally typical activities during early housing design in the U.S.—by both for-profit and non-profit developers—in the form of scenarios (based on the housing literature and interviews with various stakeholders in the industry). The framework links crucial activity blocks to typical phases in CBD. The thesis furthermore introduces classificatory types of housing precedents that provide a basis for a structured knowledge representation that supports case retrieval.

The prototype has been implemented using various components of SEED (Software Environment to Support Early Building Design). It adds to these components an efficient classification and indexing mechanism derived from the classificatory types that combines form- and component-based features and remains flexible (i.e. can be modified and customized by users); a case base on top of SEED's object-oriented database; and a retrieval mechanism that uses the indexing mechanism. For the generation and adaptation of cases, the prototype relies on functionalities provided by SEED-Layout. Prototype development uses selected methods and concepts of use case-driven software development.

CHAPTER 1

Introduction

Barry held one of the punched cards up to the light. ‘See those holes?’ he asked the manager. ‘Those holes are the only part of the software that actually goes into the plane.’
- *A Fortran Programmer working on the avionics software*

‘A building is, in principle, four walls with windows for light and air,’ and he replies that, ‘on the contrary, a building may just as well be four windows with walls for privacy and shade.’
- Geoffrey Broadbent, *Design in Architecture*

1.1 Motivation

It seems that there are commonalities between layout design in architecture and software development using punched cards. In both, emptiness is the main purpose for creation. That is, empty spaces surrounded by architectural material are a product of architectural design, as empty holes in punched cards are a product of Fortran programs. Another similarity between architectural design and software development is that they are highly goal-oriented activities seeking solutions for given problems.

In the context of the present thesis, however, one of the most important similarities between these two types of tasks is that expert designers in both domains typically refer to and re-use past solutions. For a standard situation, architects generally use a collection of standard designs, which have evolved over many decades of experience [Jackson 1995, p.189]. In most cases of software design, expert designers also avoid solving every problem from first principles. Instead, when they find a good solution, they try to use it repeatedly. Gamma et al.(1995) called the experience accumulated in designing software **design patterns**. “Design patterns help a designer get a design ‘right’ faster” [Gamma et al. 1995, p.2]. In the next two sections, I will discuss *what* case-based reasoning in design is and *why* the case-based design paradigm is promising.

1.1.1 Case-based Reasoning in Design

Case-based reasoning (CBR) is a paradigm for re-using past experience. As a part of the broader field known as artificial intelligence (AI), it is a form of analogical[1] reasoning, a central inference method in human cognition [Carbonell 1983]. People like lawyers, doctors, mechanics, and managers usually remember similar past experiences when they face a new problem and apply this experience to the new problem. CBR is an approach to transfer this natural human reasoning process to the computer using AI methods and database technology. In CBR, a reasoner remembers previous situations, called *cases*, similar to the current one and uses them to help solve the current problem. A case is defined as "a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner" [Kolodner 1993, p. 13]. The specific knowledge of previously encountered problem situations is organized into a computerized case-base.

Aamodt and Plaza (1994) describe the main phases of the CBR as the *CBR-cycle*. A general CBR-cycle includes four processes: retrieve, reuse, revise, and retain. "A new problem is solved by *retrieving* one or more previously experienced cases, *reusing* the case in one way or another, *revising* the solution based on reusing a previous case, and *retaining* the new experience by incorporating it into the existing knowledge-base (case-base)" [Aamodt and Plaza 1994, p.8].

CBR is, at its core, as a problem-solving process, but what a problem and a solution *is* varies from application to application. In *design,* the problem is generally a functional specification that includes goals and a set of requirements to be satisfied. The solution in design is a description of an artifact to solve a design problem. In architectural design, precedents from the past are often used to deal with similar current problems: "Typologies, generic solutions, and prototypes are used to help clarify the nature of problems during the intelligence phase, as a basis for generating solutions during the design phase, and as a yardstick for comparison during the choice phase of praxis" [Lang 1987, p. 62]. In the fields of environmental design, building, landscape and urban design, typologies are classifications of built structures according to the similarity of their purposes and/or their formal structure" [Lang 1987, p. 61, 62]; Chapter 3 will return to this topic. The use of typologies is an example of a *direct* analogy. A designer who encounters a new design situation recalls analogous previous design problems that can help with the new situation [Maher et al. 1995, p.3].

1. Analogies suggest an equivalence or likeness of relationship between something in one medium and something in another medium [Lang 1987] p. 63.

Computer-aided design research has a number of goals including the improvement of graphical representations, a better understanding of design processes, and the development of tools to assist designers. Most commercial CAD (computer-aided drafting or computer-aided design) systems allow designers to create, fix, re-use, and reproduce 2D drawings or 3D models with the help of the computer. CAD systems can replace manual editing because of their precise and effective modification tools. Animation extends 3D modeling by adding time to space and allows clients to see how a building will look before it is complete. Commercial CAD tools are successful as visual design media and as tools for automating low-level manual processes, like drafting and model-building.

However, intelligent CAD systems are still not available to help designers *generate* solutions. Case-based design (CBD), an application of CBR to design, promises an efficient way of finding complex design solutions by minimal search, provided that problems presented to the system have strong similarities to known cases for which solutions exist. A design *case* describes a past design experience. The content and knowledge structure of design cases as well as the organizational structure of case memory is an important aspect of a CBR system because it influences subsequent retrieval and adaptation of design cases [Maher et al. 1995].

To date, CBR has been widely used in a great variety of application domains such as mechanical engineering, medicine, and business administration. Nevertheless, its use is not common in architectural design, let alone housing design, despite the fact that CBR appears particularly appropriate for this domain because the traditional method of home design already makes extensive use of precedents, and solutions are highly standardized in the industry, at least in the U.S. The wide-spread use of CBR in that industry would have to rely on a conceptual and methodological framework that integrates all of the required functionalities and data and is accessible to a broad portion of that industry through a robust implementation. If such a CBD system is to be integrated into the daily working environment of a firm involved in housing design, it must also be able to share data with a commercial CAD, modeling and visualization software.

1.1.2 Why CBD for Housing?

1.1.2.1 Expert Systems in Design

Most AI problem-solving theories of design have concentrated on *routine design*[2]. Expert systems using rule-based and model-based reasoning techniques have been used to build design automation and design decision support systems.

Expert or rule-based systems are “computerized systems that use knowledge about some domain to arrive at a solution to a problem from that domain. This solution is essentially the same as that concluded by a person knowledgeable about the domain of the problem when confronted with the same problem” [Gonzalez and Dankel 1993, p. 21].

Although such systems have met with some success in selected design domains, difficulties have been encountered in terms of formalizing generalized design experiences as rules, logic, and domain models [Maher and Pu 1997, p.1]. Moreover, those classical AI methods of design are not applicable when design characteristics are ill-defined, and the system do not possess the flexibility that practiced designers use in the real world. In this situation, it is more efficient to create software to *assist* a human designer interactively rather than to fully automate the design task.

The following is a summary of some basic shortcomings of the rule-based system paradigm as described by Slade (1991) when it comes to higher-level design assistance:

- **The task of knowledge acquisition**[3] **is often difficult**. Most experts cannot easily describe or communicate their expertise or the way they make decisions. Furthermore, experts frequently perform their tasks without being fully aware of the processes and heuristics they use and of how they make use of their experience. They may not realize all the different aspects of a situation that they actually consider when they made a decision [Prerau 1990, p.14]. Because significant effort and time are required to transfer knowledge from the experts to the system, knowledge acquisition is considered the bottleneck in the development of expert systems [Rich and Knight 1991, p.515].

- **Rule-based systems do not have a memory**. Rule-based systems do not remember problems that they have already solved. For example, if a medical diagnosis program is presented with two different patients with the same set of symptoms, the program evaluates the same set of rules for each: it does not remember the first patient when it evaluates the second. Moreover, a program without a memory cannot remember its mistakes and will repeat them. Thus, accuracy and efficiency pose problems for rule-based systems.

2. Detailed descriptions and distinctions among routine, innovative, and creative design can be found in [Gero 1990] p. 34-35.
3. The process of extracting domain knowledge from experts during expert system development is called *knowledge acquisition*. Knowledge acquisition is usually accomplished by meetings between so-called “knowledge engineers” and domain experts, where the knowledge engineers attempt to elicit design knowledge from the experts [Prerau 1990] p.12.

- **Rule-based systems are not robust**. If the system is presented with a problem that cannot be solved by its rules, the program cannot respond because the system's knowledge base is limited to its rules: if no rule applies, the system has no way to respond, and its performance degenerates rapidly when it encounters unexpected situations.

As an alternative to rule-based systems, a case-based system has several advantages in terms of knowledge acquisition, memory and performance, and ease in constructing solutions. First, the unit of knowledge is the case, not the rule. It is easier to articulate, examine, and evaluate cases than rules. In fact, it may be possible to construct a case base without the help of knowledge engineers. Second, a case-based system can learn from its past performance in the sense that bad cases can be weeded out and new and better ones added over time. This may involve human administrators, but is generally easier because cases, as the unit of knowledge, are easier to understand by humans than rules, which often interact with each other in unanticipated ways. Third, a case-based system does not need exact matches to come up with promising solutions by reasons of analogy; even it confronted with a rather novel situation, it may still retrieve past solutions that have some relevance to the current problem and be modifiable to solve it; that is, the system degenerates more gracefully in unexpected situations [Slade 1991, p. 49]. Table 1 shows a comparison of the main characteristics between rule-based reasoning and case-based reasoning.

TABLE 1. The comparison between rule-based reasoning and case-based reasoning

	Rule-Based Reasoning	Case-Based Reasoning
Knowledge Acquisition	* The unit of knowledge: rule * Knowledge acquisition: extracting rules from experts	* The unit of knowledge: case * Knowledge acquisition: collection of the cases
Memory and Performance	Repeating the same mistake	Remember as cases, and avoid repeating prior mistakes
Ease in Constructing Solutions	* Simple chain of rules. * Find exact-match rules -> hard to construct solutions	By reasoning from analogy, ease to construct solutions to new problems.

1.1.2.2 The Housing Market

The CBD approach is particularly promising for the housing industry because of the wide-spread use of standardized plans that reflect commonly accepted plan or house types in that industry. In the US, the housing industry's design process is

largely clientless. Houses are generally developer-built products, sold on the open market just like cars or shoes [Rouda 1999]. In the absence of individual clients demanding a custom design, homebuilders usually base the plans they use on drawings they find in magazines or journals or rely on stock plans, which can be purchased through magazine advertisements or catalogues [Gutman 1985, p.1-2]. Designers and developers anticipate the reaction of the housing market based on their past experience and select the designs accordingly. They may also look at other projects that are locally under way; if this work has promise for their own business, they will attempt to learn from it and apply it with just a slight variation.

However, these traditional methods used by the housing industry have several shortcomings.

First, there are some disadvantages when plans are collected in books. The major problem with this is that the data is arranged sequentially [Flemming and Aygen 2001]. For example, authors tend to choose one major building type classification and group the designs in their books using that classification. When readers want to select data under different categories, they have to select the proper data manually unless they somehow rearrange the book for that purpose. A second disadvantage of using books is their static nature. Once they are printed, data cannot be inserted or updated only by manual annotations. It is also impossible to combine or merge data from heterogeneous sources.

Second, purchasing a home is usually an individual's largest lifetime investment. Residential designs are evolving constantly in response to changing societal values and needs. Like in the automobile industry, buyers looking for specialized products have a growing influence in the housing industry; instead of cookie-cutter houses, they want room configurations that reflect and accommodate their particular lifestyle [Wentling 1995, p.3]. However, when searching for a house in the real world, people mostly rely on model home advertisements and site visits to decide if the model meets their needs and expectations.

Third, home building in the US is a very conservative industry. There are several obstacles to innovation in the housing industry [Goldberg 1989]. The customers themselves cause some of them, but others are caused by the structure of the industry or the nature of the market. That is, the housing delivery process is a fragmented system that involves architects/designers, manufacturers, builders, regulators, financiers, and homeowners/clients. Vertical fragmentation among home builders, designers, financiers and homeowners obstructs communication between each group. Horizontal fragmentation between subcontractors further impedes the free flow of information among the participants in the design and construction process. Fragmentation in both directions becomes an obstacle to

innovation in the housing industry. The small size of the firms in each category, which may be fiercely competing with each other, also obstructs communication and the spread of innovation. (Chapter 2 returns to these aspects of the housing industry)

CBD seems to answer many of the raised questions. That is, strategic application of computer technology based on CBD can help individual designers, overcome limitations of paper-based static media, the difficulties of exploring alternatives and adapting houses easily and instantly, and the vertical and horizontal fragmentation that interferes with intra-industry communication.

An organized case-base as mentioned in Section 1.1.1 becomes more powerful over time as designers solve each new problem and add the solution to the case-base to be re-used when needed. Given such a case-base and a flexible indexing and retrieval mechanism, designers can explore various alternatives with ease, including ones they have not encountered before. If the cases can be imported into a CAD system that facilitates adaptation, customization can also happen in an efficient manner, which, in turn, may add new cases to the case base. This can also be helpful to the homebuyer looking for a more customized solution. In this case, tools to visualize a case or its adaptation in 3D would be especially helpful. The case base, in short, can overcome many limitations of traditional paper-based and static media.

Provided such a case base is widely accessible within the industry, it can also help in overcoming some limitations imposed on the industry by vertical and horizontal fragmentation. First, each group can access this case-base and make use of the newest data more easily and quickly. That is, hindrance of communications among each vertically fragmented group can be overcomed by this industry-wide accessible case-base. Second, a common case-base would allow each group to share the data that it created with others. Communication among each horizontally fragmented group or segment can be improved by the rich collection of precedents in the case base, which may spread information and new knowledge more rapidly throughout a segment. The case base, per se, does not address vertical fragmentation directly. But if it is set up so that comments or annotations can be added to a case as it passes through the different development phases, information about advantages or disadvantages associated with a specific case may become available up- or down-stream in a more timely manner.

In short, CBD offers a promising application for the housing domain because it can overcome some limitations of rule-based systems and of traditional methods of information storage and exchange in the housing industry.

1.2 Research Objective and Approach

The main objective for this dissertation is to investigate the application of CBR to housing design, especially in its initial stage. In particular, the research focuses on a general framework and computational environment that supports the schematic design phase through a case-based reasoning capability. Using this framework, the research validates the approach by implementing a prototype of CBD for housing design based on the **S**oftware **E**nvironment to support **E**arly building **D**esign (SEED) project[4] at Carnegie Mellon University.

This research attempts to satisfy these objectives through the following tasks:

1. **Investigate promising typologies and classifications that can be used for an efficient and flexible indexing and retrieval mechanism**
2. **Model the design process for single-family houses in the early phases**
3. **Define a general framework to support this housing design process through a CBD capability**
4. **Define the functional requirements of the system**
5. **Design, implement, and test a research prototype of CBD for housing.**

1.3 Overview

This dissertation is organized as follows. In Chapter 2, I provide background that covers the housing market in the US, which includes industry characteristics and supply and demand for housing. In Chapter 3, I deal with types and typologies of housing precedents, which include form-based and component-based classifications. In Chapter 4, I develop design scenarios for single-family houses and formalize the design scenarios with customized building blocks. In Chapter 5, I present a framework for integrating the housing design process and CBR. In addition, I introduce SEED as a suitable development platform for a CBD prototype of the proposed system. In Chapter 6, I develop use cases and present the user interface of a CBD research prototype for housing. These use cases specify the functional requirements of the prototype from the users' perspective. In Chapter 7, I illustrate how the CBD prototype works by concrete examples of case creation, retrieval, and adaptation. Finally, in Chapter 8, I summarize the research contributions and discuss possible future research areas.

4. Detailed descriptions of the SEED can be found in [Flemming and Woodbury 1995] p. 147-152.

CHAPTER 2

Background: The Housing Market in the US

When an architect says that he or she has a new idea, the response will always be "Great, show me some market research to prove it will work." Of course, whatever innovation an architect might come up with is more likely to take hold if it can be explained very quickly to the home-shopper during a short conversation with the sales agent. The shopper must also respond to that new idea by choosing the home above all others, as one might choose one cereal box over another from a crowded supermarket shelf.
- Mitchell Rouda, *Houses as Products*

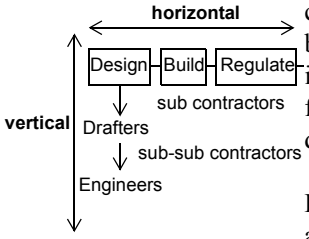
2.1 Industry Characteristics

The residential housing industry is one of the largest and most important sectors of the U.S. economy. The productivity and competitiveness of the industry and the affordability and quality of its products and processes can be increased by accelerating the rate of adoption and application of technological innovation [Goldberg 1989]. Factors that influence the development and diffusion of innovation in the housing industry are the industry's structure, the nature of the market, and the characteristics of firms. Firms in the housing industry are segmented into primary producers, manufacturers, suppliers, and home builders. Firms are classified according to the stage of production in which they operate. The following is a summary of industry characteristics that influence the generation and adoption of innovation as described by Goldberg (1989).

2.1.1 The Small Average Size of Firms

In the home building segment of the housing industry, the small average size of firms is a restriction for doing formal research and development (R&D). Generally, single-family home builders are smaller than multifamily home builders and too small to command the considerable additional resources in new management, technical personnel, and facilities that formal research and development require.

2.1.2 Vertical and Horizontal Fragmentation of the Housing Industry



Vertical fragmentation among home builders, craftsmen, architects, and sales people hinders innovation in several ways. First, complicated hierarchical communication results in poor feedback on specific improvements that need to be investigated. Second, it is difficult and costly to educate companies about an innovation when they did not participate in developing it. Finally, vertical fragmentation including the network of subcontracting makes implementation difficult and increases the costs of adoption.

Horizontal fragmentation of the various subcontractors and trade specializations also make the generation and adoption of innovations difficult. Since the contractors are only interested in their respective responsibilities, they often do not talk to one another, tend to resist any innovations that might change their work allocations, and may not be interested in research on larger systems that combine a number of different products or materials. Consequently, the scope and benefit of the innovations that can be adopted are limited.

2.1.3 Reaction to Cyclical Characteristics

The home building industry is cyclical. In such circumstances, small firms that are hardly able to cope with fluctuations usually cannot fund research during a down-cycle. The few firms that undertake research may initiate research during an upsurge in demand, but may be stopped again in the down-cycle. They tend to waste valuable time in reassembling and training a new team during the next up-cycle. As a result, the outputs of such research are either wasted or the payback is delayed, affecting the future benefit stream.

2.1.4 Sparse Management

Home building firms usually tend to emphasize research that will help them solve short-term problems and, as a result, forego the opportunity to achieve the major savings inherent in longer-term research involving systemic improvements. The relative lack of management resources tends to make research more expensive and increases the probability of eventual failure. For the same reasons, management tends to be more pessimistic when calculating and evaluating the future benefits of such research. Some managers are reluctant to engage in incremental innovation because it may threaten the schedule of current operations.

The lean management of home building firms, preoccupied with daily problems, is likely to be less aware of new technological developments and be disposed

against adopting innovations unless they provide an immediate solution to current problems. Innovations can be disruptive to building operations. Management will probably not have the time or ability to plan or incorporate them into the construction schedule. As a result, the minimum cost and the risk of adopting innovations are increased.

2.2 Supply and Demand in the Housing Market

In American society, the market for housing can be defined as "a set of institutions and procedures for bringing together housing supply and demand—buyers and sellers, renters and landlords, builders and consumers—for purposes of exchanging resources" [Bourne 1981, p. 72].

2.2.1 Making Housing Choices

Housing choices require values and a series of decisions based on needs, personal priority, and life situations. However, the real-world housing choices people make do not always reflect their housing preferences or attitudes. **Preferences** help us understand tastes that exist *independently* of constraints. As such, preferences are conceptually distinct from **choice**, which is the outcome of the interaction of preferences and constraints [Maclennan 1982]. Therefore, constraints force people to make trade-offs between preferences based on their personal priorities and the resources available when choosing their homes. That is, the real choice of housing in the market is much more complicated and dynamic, and decision-making by a housing consumer includes some uncertainty. In order to account for profits or utility and to reflect uncertainty of consumers' decision behavior, probability models are suggested by Luce and Suppes (1965), and developed by McFadden (1973, 1978). These models focus on understanding different housing choices according to housing characteristics and socio-economics characteristics of households when housing decisions are made.

2.2.2 Households: The Demand Side of the Market

According to Nissen et al.(1994, p.170), anthropologist Paul Bohannan defines the *household* as "a group of people who live together and form a functioning domestic unit. They may or may not constitute a family, and if they do, it may or may not be a simple nuclear family". Clearly, this definition encompasses a great variety of living arrangements.

Households are the units of demand in the housing market. Three aspects contribute to the housing choices made by households: lifestyle, social stratification and income, and family life-cycle.

2.2.2.1 Lifestyle of the household

"A lifestyle is a living pattern or way of life" [Lewis 1994, p. 20]. A household's lifestyle is affected by the composition of its members, their values, and social status.

Wentling (1995, p.1-2) summarizes changing lifestyle trends in American households as follows:

- Since the size of the average American household is shrinking to 2.3 persons per household, less space should be required in a house.

- Household compositions are also changing. The traditional nuclear family is shrinking both in size *and* as a percentage of total U.S. households—with only 25 % of all households now considered "traditional." The percentage of "nontraditional" households—married adults without children, single parents, persons living alone, and unrelated people sharing housing—has increased.

- Employment trends have changed. In 50 % of all married couples, both adults work. But corporate downsizing and innovations in technology force or allow people to work part-time, with flexible hours, or to work from home. Therefore, *a portion of the home is becoming an extension of the workplace*.

- Personal values are also changing. Although more households include two working spouses, education, leisure, and cultural pursuits are often rated higher in priority than career advancement. The relative importance of family and individual relationships is increasing. This means that people *want to spend more time in the comfort and security of a well-designed home environment.*

People's lifestyle influences their choice of house and the way they use their house. Households vary in their design preferences and in their use of space according to the closeness of the family, the extent to which it regards its home as the center of activity, and expectations and values conditioned by ethnic and cultural backgrounds.

2.2.2.2 Social stratification and income of the household

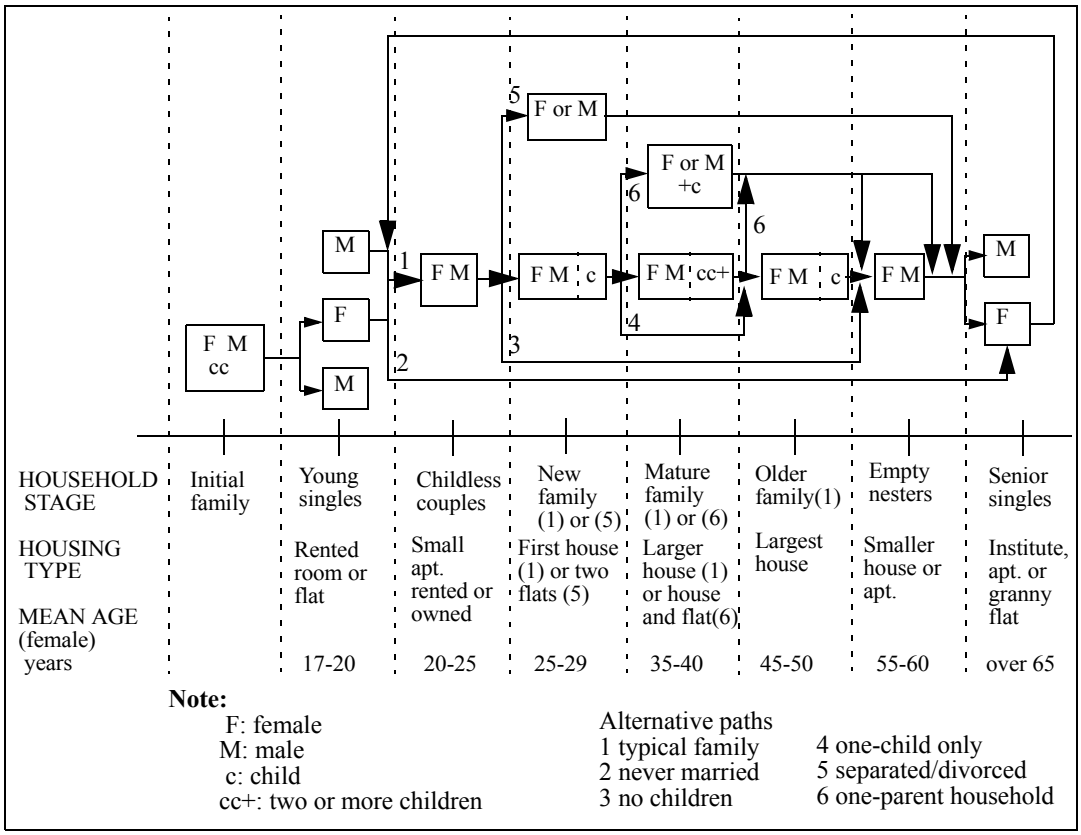
Social stratification refers to "the arrangement of any social group or society into a hierarchy of positions that are unequal with regard to power, property, social evaluation, and/or psychic gratification" [Tumin 1967, p. 12]. The consequences of stratification can determine people's *life-chances* and *lifestyles*. Income determines social stratification to a large degree. In the housing market, the housing supply and demand mechanism is based on price. Therefore, "income is usually taken as an overall index of demand and purchasing power, while dwelling price is taken as an index of the type of housing supply available" [Bourne 1981, p. 76].

2.2.2.3 Family life-cycle

As time goes by, households pass through different life-cycle stages. Since housing values, norms, needs, and preferences change as people move from one stage of the life-cycle to another, different housing decisions are made at each stage. Generally, the life-cycle of a *traditional* family has five stages: the beginning stage, the expanding stage, the developing stage, the launching stage, and the aging stage.

"The *beginning stage* is the time during which the married couple is without children. The husband and wife make adjustments to married life and to each other. The *expanding stage* is the time when the family is growing. It includes the childbearing periods and the years of caring for young children. The *developing stage* is the time when children are in school. This stage includes the years of caring for school-age children and teenagers. The *launching stage* is the time when the children become adults and leave their parents' home. They may leave to go to college, to begin a career, or to get married. When all the children have left home, the couple is again on its own. The *aging stage* is the time after retirement. At some point in this stage, either the husband or the wife live alone after the death of his or her spouse. As people live longer, this stage increases in the length of time" [Lewis 1994, p. 18, 19].

Bourne (1981, p.136) describes the relationships between demographic changes and housing needs and demand. Figure 1 illustrates a life-cycle progression starting with a typical family unit (parents [F, M] and two children [c]). It follows one of the children (in this illustration, I follow the daughter) through as many as seven stages until the children have left home and the parents have retired. This model is broadly applicable because each stage contains almost all possible alternatives for a specific family to take the next step.



Source: Reconstruction from the original source by Bourne (1981) p.136

FIGURE 1. Idealized household life-cycle

2.2.3 Housing Units: The Supply Side of the Market

The housing market has several unique characteristics that make it different from most other markets. First, products cannot by physically moved, with the exception of mobile homes. Second, some properties can be severely restricted through zoning regulations and planned neighborhoods, and can be withdrawn by public fiat such as compulsory purchase or expropriation. Third, there is no single geographic marketplace for housing. Instead, buyers move to the goods [Bourne 1981, p.72]. Because of these reasons, housing encompasses not only the dwelling itself but all that is within and near it.

Housing changes physically and formally according to environmental, cultural, societal, economic, technological, and governmental influences.

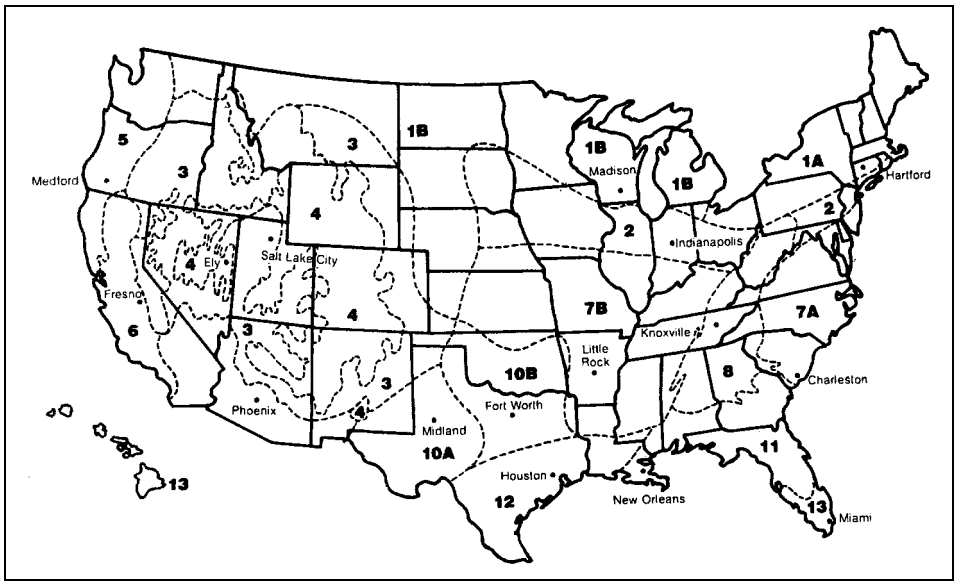
2.2.3.1 Environmental Influences

The environment comprises the conditions, objects, places, and people around us. People adapt to their environment in the housing they design and build [Lewis 1994, p.36].

Climate is the combination of weather conditions in a region over a period of years as shown by temperature, wind velocity, and precipitation [Lewis 1994, p.36]. Since the United States covers wide areas, it can be divided into several climate zones. Figure 2 presents the 13 zones[5] of climate in US[6].

According to Amos Rapoport[7], climate is one of the factors affecting housing form—the plan, roof shape, materials and insulation, and color for the interior and exterior. In the U.S., climate is one of the factors affecting housing form for vernacular (see Figure 3) and possibly custom-built houses. But this is much less true for tract houses, where one can find the same housing layouts and materials in very different climatic zones.

5. Each climate zone has a reference city: 1A- Hartford, Connecticut, 1B- Madison, Wisconsin, 2- Indianapolis, Indiana, 3- Salt Lake City, Utah, 4- Ely, Nevada, 5- Medford, Oregon, 6- Fresco, California, 7A- Charleston, South Carolina, 7B- Little Rock, Arkansas, 8- Knoxville, Tennessee, 9- Phoenix, Arizona, 10A- Midland, Texas, 10B- Fort Worth, Texas, 11- New Orleans, Louisiana, 12- Houston, Texas, 13- Miami, Florida
6. The regions have been based on heating and cooling needs, solar usefulness in a 50 to 65F range, wind usefulness in a 75 to 85F range, diurnal temperature impact, and low humidity impact for natural heating and cooling of homes [The AIA Research Corporation 1978, p. 12].
7. Amos Rapoport (1969). House Form and Culture, p.83



Source: The AIA Research Corporation (1978)

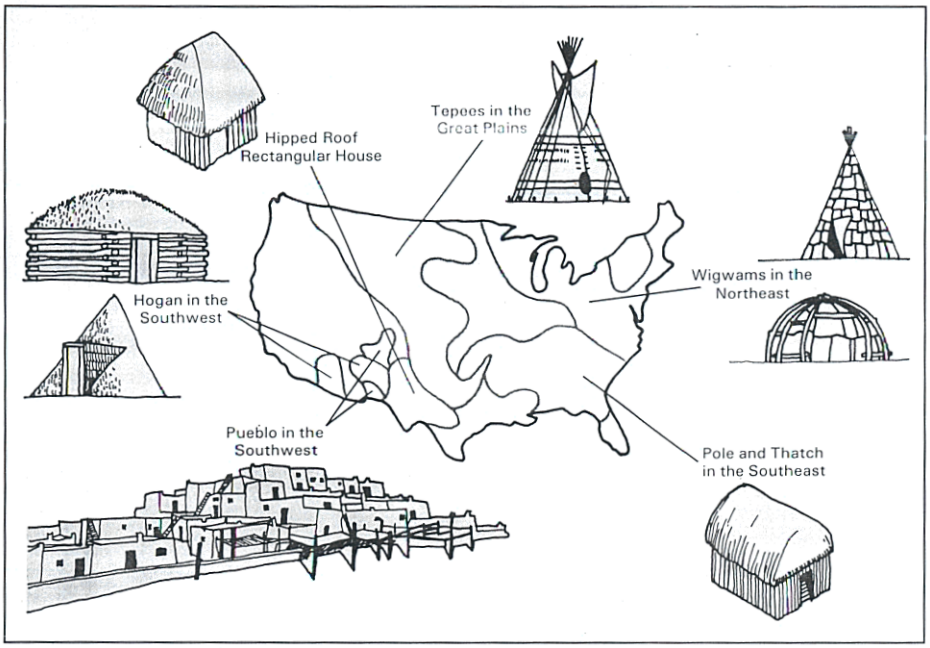
FIGURE 2. The 13 general climate regions in US

Natural constraints such as the topography, soil conditions, water supply, orientation to the sun, the wind and the scenery also influence the location and design of dwellings.

2.2.3.2 Cultural Influences

The beliefs, social customs, and traits of a group of people form their *culture*. The culture of a group of people influences its housing. *Hogans*, the housing of the Navajo, a Native American tribe of the early North American Southwest, had religious significance in the placement of the door. The tradition exists today, even though the type of housing for many has changed [Lewis 1994, p.32]. It is an example illustrated how culture influences housing.

Cultures that came from Europe also contributed their styles of housing to the American vernacular: Spanish mission, Swedish log cabin, Dutch colonial, Pennsylvania Dutch (German) Colonial, French Normandy, and Italianate housing styles are all examples of European influences on American housing.



Source: Lewis (1994) p. 28

FIGURE 3. **Early Native Americans lived in an assortment of housing. Housing designs varied, according to the region in which they were located**

2.2.3.3 Societal Influences

The growth of the cities, the movement of people to new jobs and locations, and the changing tends in American lifestyles, mentioned in section 2.2.2.1, show the signs of societal change. Choosing a certain location as one's social environment is influenced by several decisions such as employment opportunities; closeness to family members or friends; density and composition of population according to age, religion, income, ethnic group, and occupation; community facilities such as shopping, environmental protection, transportation, banking, religious organizations, and educational, recreational, and medical facilities. Cost of living can also be considered when one chooses a community.

While dual-income families, which result from changing roles in modern society, may have more income, they may have less time for household chores and

therefore may desire more convenient housing and time-saving devices [Lewis 1994, p.34].

Available leisure time also affects housing decisions. People may choose housing with low maintenance requirements so that they can spend more time on leisure activities. On the other hand, people may choose housing that provides opportunities for leisure activities such as a swimming pool, golf course, or tennis court—with the accompanying maintenace requirements [Lewis 1994, p.35].

2.2.3.4 Economic Influences

The economic influences on housing include the production and consumption of goods and services related to housing [Lewis 1994, p.38]. Houses are expensive, and price is a crucial factor in housing choices for almost everyone.

Since the housing industry employs developers, builders, material suppliers, financiers, buyers, and sellers, employment goes up and down related to the condition of the housing industry, mentioned in section 2.1.3.

Housing is traditionally the first major sector of the economy to rebound after a slump. Growth in the housing industry has a positive impact on the **Gross Domestic Product** (GDP), which is the value of all goods and services produced within a country during a given time period. Mortgage interest rates and tax regulations affect growth in the housing industry [Lewis 1994, p.39].

2.2.3.5 Technological Influences

The Industrial Revolution had a large technological impact on housing. Goods started to be mass-produced. Prefabricated houses became popular because they could be shipped in sections. Many parts of houses, such as doors and windows, come from the factory ready to install [Lewis 1994, p.40].

Many electrical devices including climate control units have been developed. Advanced technologies let people program outlets and light fixtures to save energy and control the amount of light. By touching a button, window material can change from clear to translucent. Computer technology can also be found throughout housing [Lewis 1994, p.40, 41].

2.2.3.6 Governmental Influences

Legal constraints are imposed by federal, state, or local laws. State laws are required to conform to federal legislation and local ordinances must conform to both federal and state laws. Most local housing legislation includes the following categories: standards for quality construction, control of land and density, funding for housing, housing for people in need, health, and environmental protection including health and safety standards [Lewis 1994, p.41].

Housing standards include *building codes*, which establish minimum standards for materials and construction methods. *Zoning regulations* control land use and density in certain areas. An area may be zoned for residential, commercial, or industrial use. Sometimes within a residential area, only one type of dwelling may be built. Manufactured housing and multifamily dwellings are usually restricted to specific areas. Sometimes the minimum size of dwellings to be built are also specified [Lewis 1994, p.42].

Funding is another example of government involvement in housing. The government assures some loans, which means it stands behind the lender if homeowners do not meet their obligations. Financial organizations such as Government National Mortgage Association (GNMA), Federal National Mortgage Association (FNMA), Federal Housing Administration (FHA), etc. are a part of the Department of Housing and Urban Development (HUD) [Lewis 1994, p.42].

CHAPTER 3

Housing Types and Classification Systems

There is a general distinction concerning thinking: that between *categories* and *individuals*, or *classes* and *instances*. (Two other terms sometimes used are "types" and "tokens"). It might seem at first sight that a given symbol would inherently be either a symbol for a class or a symbol for an instance -- but that is an oversimplification. Actually, most symbols may play either role, depending on the context of their activation.
- Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*

3.1 Type and Typology in Architecture

An essential aspect of cognition is the ability to categorize: to judge that a particular thing is or is not an instance of a particular category [Jackendoff 1994, p.135]. Types, in the more generic sense, are categories of thought that can be organized in generalization hierarchies [Aygen 1998]. From the eighteenth century on, type is used as a classifying tool, as in Linnaeus' famous plant classification system. The notion of type entered the architectural discourse based on this meaning [Leupen 1997, p.133]. However, the architectural notion of type depends on the context in which it is used.

Moneo (1978) defines a type as follows: "It can most simply be defined as a concept which describes a group of objects characterized by the same **formal structure**". Moneo explains that this formal structure is connected with reality, which covers a vast hierarchy of concerns running from social activity to building construction as well as to abstract geometry. Leupen (1997, p.132) draws a distinction between analytical typology and generative typology. The analytical typology is confined to naming various architectural elements and describing how these elements fit together in a composition. The generative typology, on the other hand, provides the designer with solutions, where type is *the bearer of design experiences pertaining to a similar issue* [Aygen 1998]. For the purpose of this dissertation, I consider only typologies of houses able to

support a classification of housing precedents useful for CBD. I use the term *classification* as the name or label given to a type.

3.2 Types in Housing Design

Designers have made extensive re-use of precedents through analogical reasoning for a long time. Especially in the housing domain, workbooks (Schneider, 1997; Sherwood, 1994) are organized systematically, comparing and evaluating housing precedents. Typology in housing is used to extract common characteristics and compositional principles from housing precedents and to classify them through the comparative analysis based on these characteristics and principles.

In order to allow the re-use of precedents for a computational system, a structured knowledge representation is needed. The notion of type in housing may provide a basis for arriving at such a structured knowledge representation. Based on a survey of the literature, I define two main classes of concepts in housing design: *form-based* classifications and *component-based* classifications.

3.2.1 Form-based Classifications

A form-based classification addresses higher levels of spatial organization with focus on the outline of the house plan and its context. This classification reflects site information; access method; the shape, orientation, and size of the floor plan as well as elevations and section; and style.

The shape of the house plan overall leads to some basic types: the horizontal and vertical rectangle type, the square type, the linear type, the L type, the T type, the U type, and the courtyard type.

Exterior design or articulation determines the housing *style*. It is common for home buyers that they have already decided what their favorite housing style is when they choose to purchase a house. Exterior style may determine the mood that a house conveys and the basic layout and design of the interior of the house [Kicklighter and Kicklighter 1998, p.375]. **Traditional** house styles[8] in the United States include Native American, Spanish, Swedish, Dutch, German, French, English, English/Colonial, Salt Box, Garrison, Cape Cod, Georgian, Federal, Greek Revival, Southern Colonial, Italianate, and Victorian[9]. Housing

8. They are designs created in the past that remain attractive [Lewis 1994] p. 106.

styles that have been developed in the recent past are called **modern** [Lewis 1994, p.111]. Most modern housing styles are variations of one of two basic designs: the ranch and the split-level [Kicklighter and Kicklighter 1998, p.383].

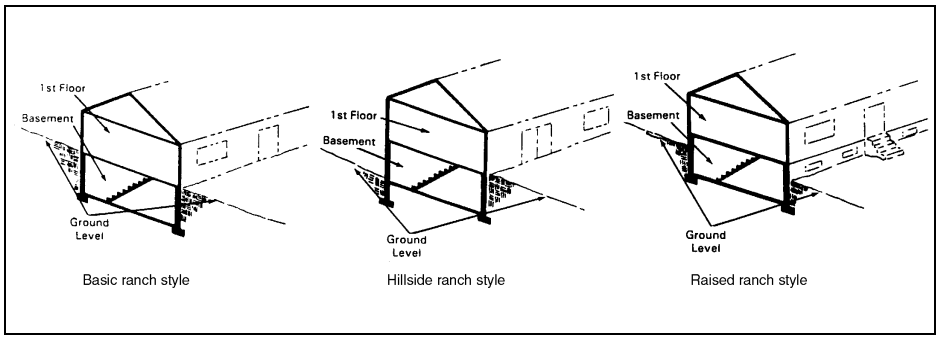
The **ranch** style, inspired by ranchers' homes in the southwest, was ideal for that region because of the informal lifestyle, open land areas, and warm climate. Now it has become popular throughout the country. Basic features of the ranch include a one-story design with no stairs and a low-pitched, gable or hipped roof with a wide overhang. The structure underneath may be rectangular or have an irregular shape, such as L, T, U, or H. Ranch houses also tend to have large window areas and sliding-glass patio doors. These houses are easy to maintain for outside tasks such as painting, cleaning gutters, or replacing window screens. They are also easily expanded and pose fewer problems of accessibility because they have no stairs. However, they cover large areas and are less energy-efficient than other housing styles because of their long, rambling configuration [Kicklighter and Kicklighter 1998, p.383, Lewis 1994, p.112-114]. Extensive foundations and roofs cause an increase in construction costs compared to multi-story houses.

Variations of the ranch include the hillside ranch and the raised ranch (Figure 4). The **hillside ranch** is built on a hill so that part of the basement is exposed. Depending on the layout of the lot, the exposed part may be anything from a living area to a garage. The **raised ranch**, also called the split-entry ranch, has the top part of the basement and garage above ground. This allows light to enter the basement through windows so that the living area in the basement, like a den, can be pleasant if it is well-insulated and waterproof. The main living quarters occupy the floor above the basement, hence the term "raised ranch." The split-level label refers to the fact that one enters the house a half level above the basement and below the main floor so that a short flight of stairs can take one up or down.

The **split-level house** is designed for a sloping or hilly site. It has either three or four different levels that are vertically offset from adjacent levels by half a floor. The general arrangement places the social, private, and service areas of the house on different levels, for which many variations exist. The three main variations of the split-level design are the side-to-side, the front-to-back, and the back-to-front arrangement (Figure 5). Advantages of split-level houses are that they provide separation of functions within the house and that they are easily adapted to all kinds of sloping sites. On the other hand, they are often more expensive to build than two-story or ranch homes because of the complicated section. Heating may

9. Detailed descriptions and illustrations of these housing styles can be found in [Kicklighter and Kicklighter 1998] p. 375-384.

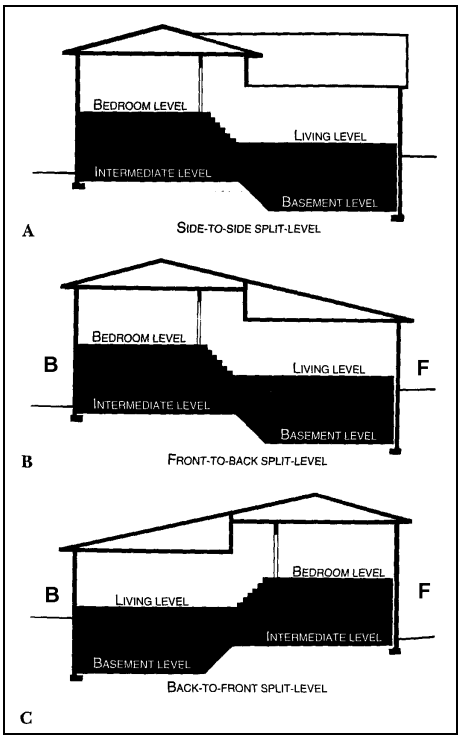
also be difficult because of the different levels [Kicklighter and Kicklighter 1998, p.384, Lewis 1994, p.114].



Source: Lewis (1994) p. 113

FIGURE 4. The three ranch styles

Since a single-family house is often adapted to the site on which it is built, the shape and orientation of the site affect the size, shape, and orientation of the house plan as well as the number of stories and the means of access to the house. For example, if a developer wants to build a house on a sloping site, the developer can consider a hillside ranch or a split-level house and take advantage of the natural slope of the site to make efficient use of space. Depending on the shape of the lot, some variations may exist within the chosen housing style. The following is an example of how housing style, access for cars and people, and number of stories can reflect the natural constraints of the site. Sites sloping sideways are best suited for the side-to-side design. Sites that are high in front and low in back are best suited for the front-to-back style. This type of house appears as a ranch from the front and as a two-story house from the back. A lot that is low in front and high in back requires a back-to-front design. In this style, the living area is typically at the rear of the house, giving it direct access to the outdoor area [Kicklighter and Kicklighter 1998, p. 384].



Source: Kicklighter and Kicklighter (1998) p. 384

FIGURE 5. **The three main split-level designs**

3.2.2 Component-based Classifications

Some geometries have been discovered that deal not with surfaces of uniform curvature, but with surfaces which are bent, twisted, magnified, shrunk or otherwise distorted. The study of such shapes falls under the general heading of *topology*, the mathematics of position (*geometria situs*) and of distortion, which deals not with the bending, twisting, and so on themselves, but with the properties of objects which are so fundamental that no amount of such distortions alters them [Broadbent 1973, p.224]. Among these, *connectivity* is especially important in a design context.

Collections of connected objects are often represented by graphs. A *graph* comprises a set of points, called *nodes*, that are connected by *edges*. In a topological plan analysis, each habitable space is typically represented by a node and the possibility of movement between spaces or a direct connection is represented by an edge. The graph can represent all interior spaces of a building and also the surrounding external spaces. A graph-based analysis of house plans can lead to a topological interpretation of the organization of rooms in diverse buildings that may uncover common connectivity features despite widely varying shapes.

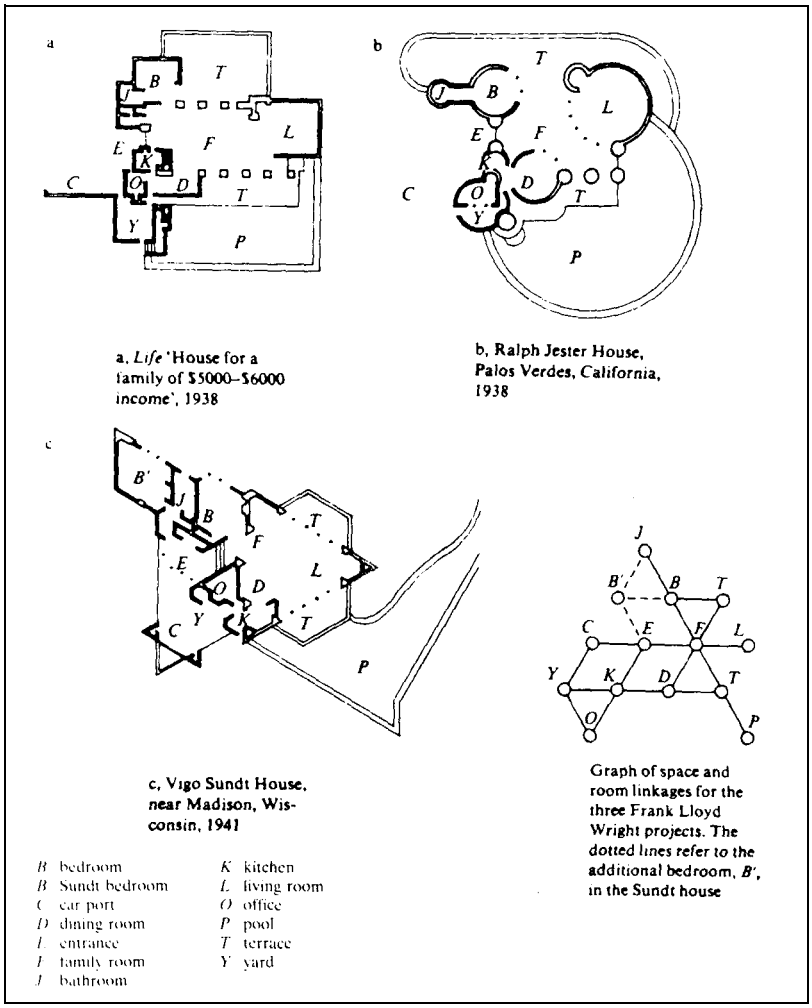
This method can show that a number of buildings which appear to have very different configurations share nevertheless an underlying structural pattern. March and Steadman (1971) demonstrated this in their analysis of three houses designed by Frank Lloyd Wright (Figure 6). This example serves to illustrate that a topological analysis of building configurations is not merely a means of visual representation, but a method of capturing spatial organization for comparative analysis [Lawrence 1987, p.51-52].

The component-based classification in the planning process for housing is based on some sort of topological analysis that classifies various ways of organizing internal spaces through the circulation system of the house plan and the connectivity among spaces. People's lifestyle, needs, and wants affect preferred adjacencies and zoning within the house. There are three primary zones in a house: social, private, and support areas. The social zone in any home encompasses the areas where members of the household gather and where friends are entertained, such as living room and dining room. Private spaces include sleeping and dressing, and hygiene areas like bedroom, bathroom, and dressing areas. The kitchen is usually the center for the support zone [Nissen et al. 1994, p. 206-280].

The spatial arrangements of the house plan can be classified based on the topological relationships between the zoning and the circulation system such as halls, corridors, and stairways. According to Schneider (1997), and Chun and Yoon (1989), some significant types of spatial arrangements are the following (see Figure 7):

- The corridor type: the floor plan is organized according to a circulation axis and the rooms are lined up on one or both sides.

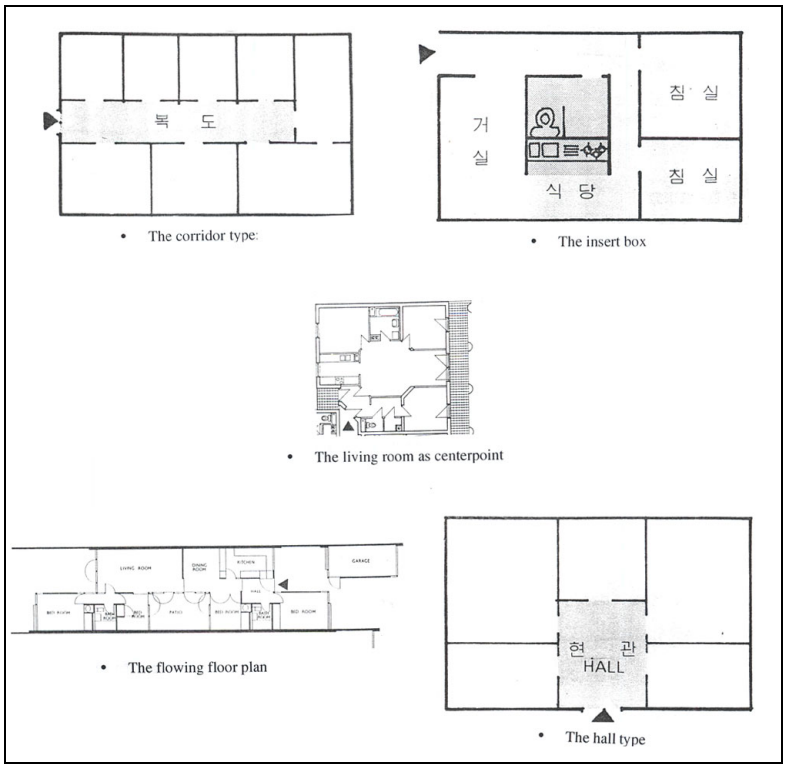
- The insert box: the floor plan is visually interpreted as a large, open space with an inserted cube (or inserted walls). This is also called the core type.



Source: Lawrence (1987) p. 52

FIGURE 6. **March and Steadman (1971) show how three Frank Lloyd Wright houses, designed for different sites, share underlying spatial arrangements of rooms**

- The living room as centerpoint: the floor plan develops around the living room. There is a variation where the central living room is combined with the corridor or hall.
- The flowing floor plan: The rooms are rarely separated from the circulation area and only slightly separated from each other.
- The hall type: the floor plan develops around a hall that is directly connected to the entry. There is a variation of combining a central hall with the corridor.



Source: Schneider (1997) p. 26, 28 and Chun and Yoon (1989) p. 45, 46

FIGURE 7. Some types of spatial arrangements

Once the arrangement of blocks of spaces is completed, designers consider the relationships between individual rooms. The relations between living room, dining room and kitchen are among the most important connections in the house. The requirements for each room or area depend on the number of inhabitants and the residential profile. The number of bedrooms, bathrooms and garages is a major consideration and often directly reflected in the classification of a home, like in "3-bedroom split-level.". Needs for special rooms such as a powder room are also important because they make certain house plans more popular than others for people with this preference.

Note that certain configurational features can be derived automatically from a structured, graph-based representation of cases; i.e. they do not have to be captured explicitly through classification labels.

3.3 Classification of Housing Precedents in CBD

Form-based and component-based classifications can be used to classify housing precedents. Figure 8 shows parts of a possible type hierarchy for housing precedents. Since a single precedent can combine features represented by several types, it can be grouped under different types. This makes the classification of housing precedents complex and requires multiple classification taxonomies.

The classificatory types are able to serve as an indexing scheme for the retrieval of housing precedents with desired characteristics or features in a CBD system. I show in Chapter 5 how housing precedents can be multiply indexed based on form-based and component-based classifications.

Housing Types				Example
Generic	Regional Location			NW, SW, Central, NE, etc.
	Cost			$50,000-$100,000, $100,000-150,000, etc.
Form	Site	Shape		
		Orientation		North, East, West, South, etc.
	Access			Front, Side, Back, etc.
	Plan	Shape		
		Orientation		North, East, West, South, etc.
		Size		under 2000 Sq.ft, 2000-4000 Sq.ft, etc.
	Elevation	Exterior Style		Georgian, Victorian, Italianate, etc.
	Section			
Component	Spatial Arrangement			Corridor, Insert Box, Hall type, etc.
	Configuration	Zoning		Social, Private, Support, etc.
		Number of Units	Bed	One, Two, Three, etc.
			Bath	One, Two, Three, etc.
			Garage	One, Two, Three, etc.

FIGURE 8. **An example of type hierarchy for the classification of housing precedents**

CHAPTER 4

Development of Design Scenarios for Single-Family Houses

The word *design* has always intrigued me because a sign is a label for something, and if you have a label you must be familiar with it. The process of "*de*-signing," of taking the label away, suddenly frees that something from being what it was to become something new. A great design process is one in which something that you're familiar with becomes something new because you've transformed it. That process of transforming it takes it from being a static object to becoming something you can really use.
- Edwin Schlossberg, in C. Thomas Mitchell, *New Thinking in Design*

The housing delivery process is a collaborative activity, involving architects/ designers, manufacturers, builders, regulators, financiers, and clients. One of the aims of this thesis is to provide a basis of design support applications for those design practitioners. This requires an understanding of the design process in the housing industry.

4.1 Interviews with Housing Design Experts

In my study of the housing delivery process, I used three methods to gather information: I reviewed the housing literature, interviewed housing design experts, and visited real estate agencies to observe tasks performed there. All these methods provide different points of views of the application domain, but interviews with housing design experts were—in the end—the main source for developing the design scenarios below.

In order to develop design scenarios for housing development, five interviews with housing domain experts were conducted. Subjects A, B, and C are practicing architects. Subject A is also a professor in the School of Architecture at Carnegie Mellon University. Subject C is an instructor in the same department. Subject D is an executive director of the Housing Authority of the City of Pittsburgh (HACP). Finally, subject E is a developer with experience in both for-

profit and non-profit housing development. The interviews mainly focused on single-family detached housing, since that is the main focus of my thesis.

4.2 Housing Development Types in the US

Single-family housing includes tract houses, custom-built houses, and manufactured houses.

The most individualistic house is *custom-designed* and *custom-built*. A designer considers the needs, personal priorities, and lifestyle of a household and then designs a house to fit these conditions [Lewis 1994, p.71]. This is a good way to build a house for most of people. However, this type of house costs more per square foot than other types [Kicklighter and Kicklighter 1998, p. 19] and takes a longer time to plan and build [Lewis 1994, p. 71].

In contrast to the custom-built house, tract houses are designed for *potential* buyers. To build tract houses, a developer subdivides a larger piece of land into lots, and builds houses based on a limited number of different designs to reduce the cost of each house through repetition and economy of scale. A model house of each design is generally completed and opened to the public to entice prospective buyers. There are several advantages for tract houses. A tract house usually costs less than a custom-built house. The buyer can see several model homes to choose from, and the subdivision has been planned as a whole including facilities. But tract houses also have several disadvantages. First, they may look very similar to each other and lack individuality. Second, the sites often look unfinished until trees and shrubs grow. Third, the lots are generally sized to maximize profit for the builder. Finally, initial buyers are faced with the uncertainty of whether the development will eventually be successful or not [Kicklighter and Kicklighter 1998, p.18-19].

If the prospective homeowners want a custom house, but cannot afford to hire an architect, they may purchase a stock plan that has been well-designed by professionals from a magazine or other source, consult with a builder, and modify it to fit their needs [Kicklighter and Kicklighter 1998, p.19].

According to Bourne (1981), there are two basic mechanisms for housing allocation.

> ..... One is the traditional private “market” which allocates households to housing on a competitive basis in terms of the values people attach to housing and their ability to pay. A second is that of public sector allocation in which governments, housing officials or some other community group distribute housing

> according to individual and collective needs and the objectives of the agency involved (Bourne 1981, p. 69).

The former can be called *for-profit*, while the letter can be called *non-profit*. Those development types pursue different objectives and use different criteria for housing allocation as shown in Table 2.

TABLE 2. Private vs. public housing allocation

	Private market allocation	Public sector allocation
Principal objective	Profit	Social equity
Criteria of efficiency	Minimizing aggregate housing prices and rents Maximizing output and profits Maintaining rates of return	Maximizing use of existing stock Minimizing administrative costs Maintaining adequate stock
Criteria of equity	No one can move without making others worse off Price restricts over-consumption	Assuring adequate housing for all Treating all equally according to their needs
Process of allocation	Competition (ability to pay)	Needs and social priorities
Countervailing process	Collusion, cooperation	Competition (among agencies and tenants)

Source: Redrawn from Bourne (1981) p. 71

There are some commonalities between for-profit and non-profit housing development. In both, the feasibility of the development must be established and its basic architecture must be refined by designers.

On the other hand, their goals are different. The definition of success for for-profit developers is to make money. In contrast, the goal of non-profit developers is to improve the community or neighborhood environments and to provide affordable housing such as co-ops, public housing, low-income family housing, housing for the elderly, or housing for people with special needs.

Another major difference between for-profit and non-profit development are the participants involved in the respective design processes. A for-profit

development is more likely to concist of one-to-one interactions between the developer and the designer or other contributors, while non-profit development is done by an organization that has a mission. The organization forms a design committee, which includes several groups such as the board of directors of a community development corporation (CDC), potential residents of the housing project, the residents of surrounding neighborhoods, local community leaders, property managers, etc. The committee collects the opinions of these various groups to incorporate their ideas into the housing development. A non-profit organization thus develops its design based on feedback from the design committee and the characteristics of the neighborhood and architecture. To revitalize neighborhoods, non-profit housing projects in the US tend to occur in mixed-income neighborhoods with a mix of public housing, subsidized rental, and market-rate for-sale housing. Note that there is no architectural distinction among those types of homes.

Another difference between for-profit and non-profit developments are the funding sources. Non-profit development can secure more varied sources of funds from federal government agencies, urban redevelopment authorities or housing authorities, corporate contributions, as well as private foundations or lenders. Public policy agencies such as Housing and Urban Development (HUD) and State and local governments have coordinated their programs and resources to promote worthwhile projects. But it has to be kept in mind that buyers need funds if they want to purchase a home in for-profit and non-profit developments.

4.3 Overview of Design Scenarios

Based on the interviews with housing design experts, I developed design scenarios, each of which describes a specific development phase. These design scenarios can be a starting point for software developers to define requirements of a system meant to support the process. Note that the term *design scenario* is not related to the term *scenario*[10] in UML.

Let's imagine that a client wants to obtain housing. There are many decisions to make: where to live; whether to buy or to rent; whether to build or buy pre-owned; how to pay, etc. The decisions will depend on the client's lifestyle, social stratification and income, and family life-cycle as described in section 2.2.2. The client may go to a real estate agency that is connected with housing practitioners such as builders, regulators, financiers, and manufacturers, and work with the

10. In UML, the definition of a scenario is "a narrative description of what people do and experience as they try to make use of computer systems and applications" [Bruegge and Dutoit 2000].

agent to acquire a house. Such a situation is illustrated by the following (hypothetical) scenario.

> A client decides to build a house and visits a real estate agency to meet a sales person. The client works with the sales person to find available lots in the client's preferred area. Once a lot is chosen, the sales person introduces the client to a builder to discuss which plan(s) are available. The client chooses one of them and discusses the house in detail with the builder. If the client is not completely happy with the plan, the client and the builder adapt the plan to fit the client's needs and situation. This process will be iterated until all problems are resolved.

The scenario above describes a design process for a client building a house on a chosen lot. The following question has to be asked about this scenario: Does this scenario explain the design activities that *ought to* occur or the design activities that *actually* occur?

There are two models used in design research: One is a *descriptive* model, which is based on observations of real world design processes and explains observed behavior in order to form a scientific theory of the design process. The second model is *prescriptive* and normative because it intends to tell designers how to structure their design activities [Cumming 1999].

All models in the present section are descriptive. The main goal is to arrive at a detailed understanding of typical phases in housing design as it occurs today and to identify steps or tasks that may benefit from support by a CBD system and case base.

4.4 Formalized Representation of Design Scenarios

As illustrated in section 4.3, a design scenario description is typically written in standard prose. The *building blocks* concept used by Cumming (1999) is helpful in formalizing a design scenario description. He borrows some aspects of process representation from Colored Petri Nets (CPNs), which can be used as general-purpose process representations and adapted to capture *design* processes in various domains.

In most design processes, the practitioners have meetings and perform individual tasks, which may be sequential or iterative. Cumming develops ten types of building blocks for design scenarios: performing a task, sequential task, branch in, branch out, conditional tasks, iterating processes, composing processes using recursion, holding meetings, building consensuses, and list of design participants.

These customized building blocks provide a structured format for describing design scenarios. For a detailed description of the representation and building blocks for the design scenario, refer to Appendix A.

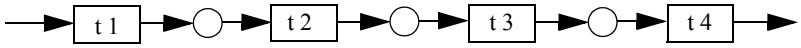


FIGURE 9. **An example of building blocks: 'sequential task'**

Each housing development can be captured by a sufficient number of design scenarios depending on the type of participants performing them. A design scenario, in turn, comprises several building blocks, each of which has a set of associated main tasks that the respective process should accomplish. For instance, in the design scenario illustrated in section 4.3, the overall design scenario can be described in terms of four sequential task building blocks: Meeting 1 and Iterative processes (Figure 10).

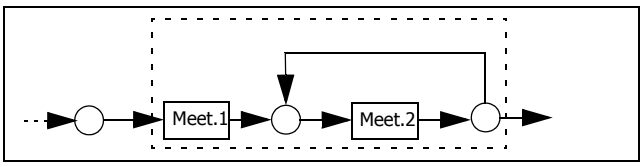


FIGURE 10. **Body of a design scenario combining linear and iterate design tasks with meetings**

```
Sequential task Building a house on a chosen lot =>
    Ordered task list
      >>• Meeting 1
      >>• Iterate building block (including Meeting 2)
```

Each of these building blocks can be expanded into a more detailed task description. For example, the body of the 'Meeting 1' building block my expand this task as follows:

Hold meeting Meeting 1 =>

Meeting participants

- Client
- Sales Person

Agenda tasks

>>• Choose a lot for the client

>>• Find a house plan based on the client's specification

Design conflicts anticipated

- The suggested house plan may not fit the chosen lot.

Consensus anticipated

- Some alternative for the suggested house plan will be acceptable.

Tasks may be either non-terminal or terminal. Non-terminal processes are those which are further decomposed into lower level processes (shown with the '>>' prefix), while terminal processes (shown with the '[LEAF]' prefix) are at their final level of decomposition. Since the agenda tasks of Meeting 1 are non-terminal processes, they can be decomposed into lower-level processes. This is illustrated by the 'Choose a lot for the client' task:

Sequential task Choose a lot for the client =>

Ordered task list

- [LEAF] The client gets some information about neighborhoods and decides where to live: choose a neighborhood
- [LEAF] The sales person searches for available data
- [LEAF] The sales person suggests available lots in the client's preference area
- [LEAF] The client chooses one of the alternative lots

4.5 Design Scenarios

4.5.1 Scenario 1: Developer-Designer Interaction—Establishing Feasibility

4.5.1.1 Overview

A designer has been hired by a developer who is interested in developing a for-profit project at a specific site. The developer and the designer meet. After the meeting, each pursues a separate feasibility study. After their studies are completed, they have a second meeting to discuss both studies and make some decisions.

4.5.1.2 Actors and Goals

Actors

- Developer
- Designer

Goals

- Establishing feasibility for a for-profit development project at a specific site

4.5.1.3 Body of Design Scenario

```
Sequential task Establishing Feasibility =>
     Ordered task list
      >>• Meeting 1
      >>• The developer and designer go off separately
      >>• The developer and designer meet again
      >>• Meeting 2
```

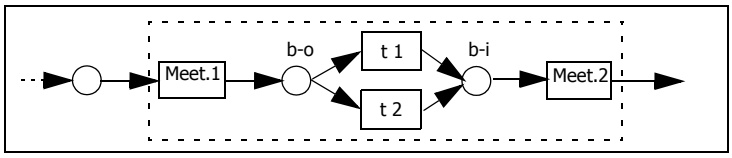


FIGURE 11. Design Scenario 1: design task consisting of parallel activities, with meetings at the beginning and end.

a. Meeting 1

The agenda of this meeting is to start the process of establishing feasibility of the development. The developer describes the project goals and provides information related to potential market, total project budget and the site to the designer. The developer and the designer divide responsibilities for the feasibility study.

Body of Meeting 1

Hold meeting Meeting 1 =>

Meeting participants

- Developer
- Designer

Agenda tasks

- [LEAF] Developer describes the project goals and provides information related to potential market, total project budget, and the site to the designer
- [LEAF] Designer and developer divide responsibilities for feasibility study

Design conflicts anticipated

- None

b. The developer and designer go off separately

Once Meeting 1 is completed, the developer and the designer go off separately and complete their respective feasibility studies in parallel.

Branch-out The developer and designer go off separately =>

Input tasks

- [LEAF] Meeting 1

Output tasks

>>- Feasibility study by Developer

>>- Feasibility study by Designer

Subtask b1. Feasibility study by Developer

The Developer estimates soft costs.

Body of feasibility study by developer

Do task Feasibility study by developer =>

Collect data

- Marketing information: Housing value[11] from tax information and demographic features of the neighborhood for the given site
- Construction financing (bank)
- Tax: county and municipal tax
- Insurance
- Utility (impact fees, tap-in, temporary power, heat, water, and sewage, and permanent utility cost for power, gas, water, sewage, telephone, cable TV, and Internet)
- Sales cost (real estate fees)

Subtask b2. Feasibility study by designer

The designer starts with the feasibility study.

Body of feasibility study by designer

Sequential task Feasibility study by designer =>

Ordered task list

- [LEAF] Study the neighborhoods and architecture around the site

>> • Prepare a preliminary site plan

11. Housing value can be acquired from tax office. The calculation of the tax varies with the community. Example:
    - Housing value: Market value = Assessed value * 4 + various municipality
    - The county and municipal tax is some percentage of housing value.

>> • Refer to precedent(s)

>> • Lay out prototype unit(s)

• [LEAF] Calculate all hard cost

Subtask b2-a. Prepare a preliminary site plan

The designer studies the zoning ordinances for the site under consideration to determine the existing context and zoning designation, which includes R1, R2, R3, B, and I. The designer develops a preliminary site plan based on the zoning ordinance.

Body of Prepare a preliminary site plan

Sequential task Prepare a preliminary site plan =>

Ordered task list

- [LEAF] Check the zoning ordinances and set of rules and regulations implied by the context and zoning designation.
- [LEAF] Develop a preliminary site plan indicating infrastructure, individual lots, and the placement of housing units.

Subtask b2-b. Refer to precedent(s)

The designer is looking for housing precedents from different sources.

Body of Refer to precedent(s)

Do task Refer to precedent(s) =>

Assemble resources

- In-house plans
- On-line
- Memory or databases

Subtask b2-c. Lay out prototype unit(s)

Based on the above studies, the designer comes up with an idea about the type, size, and cost of the units. The designer takes the configuration(s) from the preliminary site plan and designs a prototype unit. Then, the designer develops variations according to site situations.

Body of Lay out prototype unit(s)

Sequential task Lay out prototype unit(s) =>

Ordered task list

- [LEAF] Design a prototype unit considering housing style, massing, type of construction, exterior materials, and finishing packages.
- [LEAF] Vary the prototype to adapt the housing configuration to the topography of each site.

c. The developer and designer meet again

Once the designer and developer complete their feasibility study, they meet again.

Branch-in The developer and designer meet again =>

Input tasks

>>• Feasibility study by Designer

>>• Feasibility study by Developer

Output tasks

>>• Meeting 2

d. Meeting 2

The agenda of this meeting is to determine if the developer should go ahead with the project. In this meeting, the developer and the designer discuss the project based on feasibility studies done by each in order to determine if it is profitable or not.

Body of Meeting 2

Hold meeting Meeting 2 =>

Meeting participants

- Developer
- Designer

Agenda tasks

>>• Determine to go ahead with the project

Design conflicts anticipated

- Housing configuration is not appropriate for target market segment
- Housing volume is too large or small
- Exterior materials are too expensive or cheap
- Finish packages are too expensive or cheap
- Amenity packages are too expensive or cheap

Consensus anticipated

- Adjust all conflicts listed above

Subtask d. Determine to go ahead with the project

The developer checks total cost for the development based on feasibility studies and looks at the total anticipated income from the development. If the rate of return meets the developers expectations, the developer will develop the project. Otherwise, the developer and the designer will re-cycle through the process until the rate of return is acceptable or abandon the project.

Body of Determine to go ahead with the project

Conditional task Determine to take the project =>

Test

- 'Current total cost'

Input tasks

- Finish feasibility study by designer
- Finish feasibility study by developer

Conditional outputs

- **If** 'Current total cost is higher than income'

  >> • **Do Task** Reduce project cost

- **If** 'Current total cost is less than income'
  - [LEAF] Continue to the project

Subtask d-a. Reduce project cost

If the current total cost is higher than the income, the developer and the designer try to reduce project cost.

Body of Reduce project cost

Iterate Reduce Project Cost =>

Test

• **If** 'Current cost is higher than income'

- [LEAF] Analyze and reduce hard cost
- [LEAF] Analyze and reduce soft cost

Repeat test

• **Else**

- Go to the next step

4.5.2 Scenario 2: Designer Working Independently—Refining Basic Architecture

4.5.2.1 Overview

After the preliminary feasibility has been established, the designer starts to refine the design of the prototype. The designer and developer then meet to examine and discuss the schematic layout. This process iterates several times in a similar fashion until the schematic layout is completed. Then, the designer goes into detailed design and meets again with the developer to discuss the detailed design. This process also iterates several times until the detailed design is completed.

4.5.2.2 Actors and Goals

Actors

- Designer
- Developer

Goals

- Developing schematic layout and detailed design for the project

4.5.2.3 Body of Design Scenario

```
Sequential task Refining basic Architecture =>
     Ordered task list
      >>• Iterate building block 1
      >>• Iterate building block 2
```

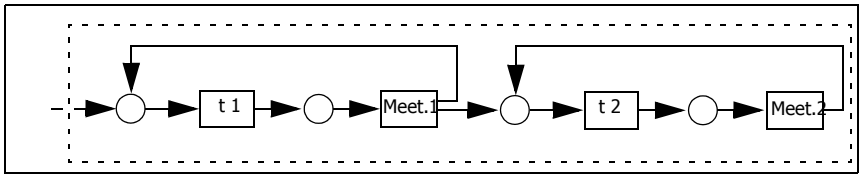


FIGURE 12. **Design Scenario 2: Combining linear and iterate design task**

a. Iterate building block 1

These tasks are done iteratively until the task is completed. They involve a design task and a meeting in the iterative loop.

Iterate Iterate building block 1 =>

Test

• **If** 'Design conflicts still remain'

>>• Refine the design of the prototype

>>• Meeting 1

Repeat test

• **Else**

• Archive the layouts and go to the next step

Subtask a1. Refine the design of the prototype

The designer develops the schematic layout design by arranging the interior spaces. For example, the designer may want to put the public space in the back of the first floor, kitchen in the front of the first floor, and the private space in the second floor. Then, the designer configures each room. For example, the designer may select the "great room" concept, which is a combination of a family room, a dining room, and a kitchen without any separation, places the master bath to serve the master bedroom, and adds one full and another half bathroom for other family members to share, and so on.

Body of Refine the design of the prototype

Sequential task Refine the design of the prototype =>

Ordered task list

• [LEAF] Arrange the interior spaces

• [LEAF] Configure each room

Subtask a2. Meeting 1

In this meeting, the developer examines the designer's schematic layout and provides comments or ideas to improve the plan. During the meeting, design

conflicts are identified and some are resolved. When both the designer and the developer are satisfied with the plan, a consensus has been reached.

Body of Meeting 1

Hold meeting Meeting 1 =>

Meeting participants

- Developer
- Designer

Agenda tasks

- [LEAF] Refine the schematic layout

Design conflicts anticipated

- Zoning plans of some houses need to be changed because of the orientations of the site
- Housing configuration is not appropriate for target market segment
- The developer wants to change some room configurations

Consensus anticipated

- Try to adjust all conflicts listed above

---

b. Iterate building block 2

These tasks are done iteratively until they are completed. They involve a design task and a meeting in the iterative loop.

Iterate Iterate building block 2 =>

Test

- **If** 'Design conflicts still remain'

>>- Do detailed designs

>>- Meeting 2

Repeat test

- **Else**
  - Archive the drawings and go to the next step

Subtask b1. `Do detailed designs`

Once the designer has finished the schematic design, he/she prepares the working drawings.

Body of Do working drawings

`Do task` `Do detailed designs =>`

`Assemble resources`

- `Working drawings (plans, elevations, sections, perspectives, details, etc.)`
- `Specifications`
- `Create schedules for paint, trim, wall coverings, doors, or windows`

Subtask b2. `Meeting 2`

This meeting is very similar to Meeting 1, but the developer examines the designer's construction drawings.

Body of Meeting 2

`Hold meeting` `Meeting 2 =>`

`Meeting participants`

- `Developer`
- `Designer`

`Agenda tasks`

- `[LEAF] View the construction drawings and discuss them`

`Design conflicts anticipated`

- `quite possible from every detail`

`Consensus anticipated`

- `Try to adjust all conflicts`

4.5.3 Scenario 3: Sales Agent-Client and Builder-Client Interaction—Building a House for a Client on a Chosen Lot

4.5.3.1 Overview

A client decides to build a house and visits a real estate agency to meet a sales person. The client works with the sales person to find available lots in the client's preferred area. Once a lot is chosen, they find house plans based on the client's specification. The client chooses one of them and meets with a builder to discuss the house in detail. If the client is not completely happy with the plan, the client and the builder adapt the plan to fit the client's needs and situation. This process will be iterated until all problems are resolved.

4.5.3.2 Actors and Goals

Actors

- Client
- Sales Person
- Builder

Goals

- Building a house on a chosen lot for a specific client

4.5.3.3 Body of Design Scenario

```
Sequential task Building a house on a chosen lot
     Ordered task list
      >>• Meeting 1
      >>• Iterate building block (include Meeting 2)
```

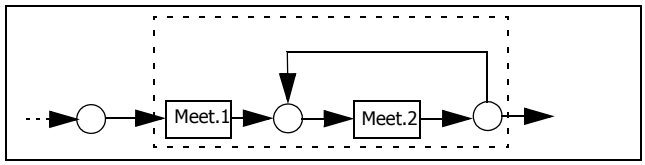


FIGURE 13. **Design Scenario 3: combining linear and iterate design task with meetings**

a. Meeting 1

The client and the sales person meet. The agenda is to choose a lot in the client's preferred area and to find a house plan to build on it. Once a lot is chosen, the sales person tries to find housing precedents that are similar to this situation. The client chooses one of them.

Body of Meeting 1

Hold meeting Meeting 1 =>

Meeting participants

- Client
- Sales Person

Agenda tasks

>>• Choose a lot for the client

>>• Find a house plan based on the client's specification

Design conflicts anticipated

- The suggested house plan may not fit the chosen lot

Consensus anticipated

- Try to find alternative possibilities for the suggested house plan

Subtask a1. Choose a lot for the client

The client obtains some information about communities and neighborhoods from the sales person and selects a neighborhood to live in. The sales person then searches for available data and finds available lots in the client's preference area. The client chooses one of the alternative lots.

Body of Choose a lot for the client

Sequential task Choose a lot for the client =>

Ordered task list

- [LEAF] The client gets some information about neighborhoods and decides in which neighborhood he/she wants to live
- [LEAF] The sales person searches for available data
- [LEAF] The sales person identifies available lots in the client's preference area
- [LEAF] The client chooses one of the alternative lots

Subtask a2. Find a house plan based on the client's specification

Once a lot is chosen, the sales person searches through the collection of house plans and recommends one or more precedents which are similar to the client's specification. The client chooses one of the plans.

Body of Find a house plan based on the client's specification

Sequential task Find a house plan based on the client's specification =>

Ordered task list

- [LEAF] The sales person searches through the collection of house plans
- [LEAF] Suggest one or more precedents similar to the client's specification
- [LEAF] The client chooses one of the alternative house plans

b. Iterate building block

These tasks are done iteratively and involve a meeting in the iterative loop.

Iterate Iterate building block =>

Test

• **If** 'Design conflicts still remain'

>> • Meeting 2

Repeat test

• **Else**

• Finalize a house plan to be built

Subtask b. Meeting 2

The agenda is to discuss the house plan to be built in detail.

Body of Meeting 2

Hold meeting Meeting 2 =>

Meeting participants

• Client

• Builder

Agenda tasks

>> • Adapt a house plan to be built

Design conflicts anticipated

• The client doesn't like aspects of the house plan

Consensus anticipated

• The client and the builder agree to modify the house plan

Subtask b-a. `Adapt a house plan to be built`

One of the alternative house plans is chosen, but the client is still not satisfied with the plan because it does not completely fit the client's situation. The client and the builder try to adapt the house plan to fit the client's need and site situation.

Body of Adapting a house plan to be built

`Sequential task` `Adapt a house plan to be built =>`

`Ordered task list`

- `[LEAF] The client points out a part of the plan want to be modified`
- `[LEAF] Adapt the house plan to fit the Client's need and site situation (site, house, and space levels)`
- `[LEAF] Archive this (new) house plan`

4.5.4 Scenario 4: Non-profit Housing Development—A Neighborhood Planning Process

A community development corporation (CDC) approaches the local housing authority to develop an in-fill non-profit housing project.

4.5.4.1 Overview

A CDC contacts the housing authority. They get pre-development financing from the housing authority and hire designers to develop a neighborhood master plan. Based on that, representatives of the housing authority, urban redevelopment authority, and CDC meet to establish the financial feasibility of the whole neighborhood development.

4.5.4.2 A Neighborhood Planning Process: Actors and Goals

Actors

- Housing authority (director and staff)
- Community development corporation (CDC)
- Urban redevelopment authority
- Market consultant
- Architect

Goals

- Developing neighborhood plan for the chosen neighborhood

4.5.4.3 Body of Design Scenario

```
Sequential task A Neighborhood Planning Process =>
     Ordered task list
      >>• Meeting 1
      >>• Preparing to develop a neighborhood master
          plan
      >>• Developing the neighborhood master plan
      >>• Meeting 2
```

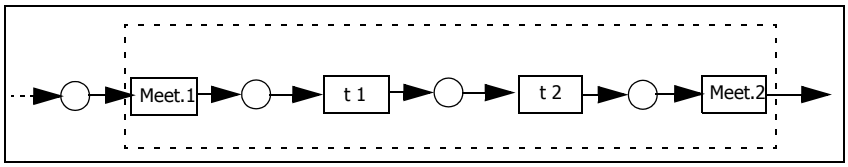


FIGURE 14. **Design Scenario 4: linear design task with meetings at the beginning and the end.**

a. Meeting 1

The agenda of this meeting is twofold: The housing authority provides pre-development financing to the CDC. After that, the CDC hires a market consultant and architect to investigate the neighborhood context.

Body of Meeting 1

Hold meeting Meeting 1 =>

Meeting participants

- Housing authority
- CDC
- Market consultant
- Architect

Agenda tasks

- [LEAF] The Housing Authority provides pre-development financing to the CDC
- [LEAF] The CDC hires a market consultant and architect to investigate the physical neighborhood (i.e. ownership, tenure, value of property, building massing, condition of building, etc.)

Design conflicts anticipated

- None

b. Preparing to develop a neighborhood master plan

The market consultant and architect meet and prepare to develop a master plan of the neighborhood. After conducting several design "charrettes" and reviews, they publicly present the plan to get some feedback.

Body of Preparing to develop a neighborhood master plan

Sequential task Preparing to develop a neighborhood master plan =>

Ordered task list

- [LEAF] Obtain marketing information from the market consultant
- [LEAF] Make an analysis in the area
- [LEAF] Conduct design charrettes in the area to develop plan alternatives
- [LEAF] Present the analysis publicly to obtain feedback
- [LEAF] Meet with traffic experts, real estate analysists, and government officials to discuss alternatives
- [LEAF] Present analysis and plan alternatives publicly to obtain comments and feedback

C. Developing the neighborhood master plan

Based on the studies above, the market consultant and architect develop a master plan of the neighborhood.

Body of Developing the neighborhood master plan

Sequential task Developing the neighborhood masterplan =>

Ordered task list

- [LEAF] Develop a master plan - perspectives of the neighborhood and preliminary unit designs
- [LEAF] Perform a preliminary pricing and cost analysis

d. Meeting 2

The agenda of this meeting is to establish the financial feasibility of the master plan. Representatives of the housing authority, urban redevelopment authority, mayor's office, and CDC meet to establish the financial feasibility of the whole neighborhood development.

Body of Meeting 2

Hold meeting Meeting 2 =>

Meeting participants

- Housing authority
- Urban redevelopment authority
- Mayor's office
- CDC

Agenda tasks

- [LEAF] Establish the financial feasibility of the master plan

CHAPTER 5

A Framework for Integrating Housing Design and CBD

The artificial world is centered precisely in this interface between the inner and outer environments; it is concerned with attaining goals by adapting the former to the latter. The proper study of those who are concerned with the artificial is the way in which that adaptation of means to environments is brought about -- and central to that is the process of design itself.
- Herbert A. Simon, *The Sciences of the Artificial*

I have developed, in Chapter 4, design scenarios for single-family houses. The design scenarios can be used to create a framework for augmenting the housing design process with CBD. The main issue I address in this chapter is how to make CBD an integral part of this process.

I first describe the main steps of CBR in design. I then identify those building blocks in the scenarios presented in the preceding chapter where typical CBD steps can assist the housing design process. I conclude by introducing a suitable platform for a first prototype implementation of the proposed system.

5.1 The Main Steps in CBD

The main problems addressed by CBD can be grouped into three areas: creating design cases, retrieving design cases, and adapting design cases [Maher et al. 1995].

5.1.1 Creating Design Cases

Any CBR system creates cases in a particular *representation.* In general, a case has three major parts: a problem/situation description; a solution; and an outcome. The **problem/situation description** may contain the goals to be achieved, the constraints on those goals, and the features of the problem context.

It must have sufficient detail for a retriever to be able to determine whether a past design case is applicable to a new design problem. The **solution** may contain the set of reasoning steps used to solve the problem, the set of justifications for decisions, alternative solutions, and expectations about outcome. The set of reasoning steps can be helpful to reuse in later design problem solving. The set of justifications is also useful to provide guidelines during evaluation of a solution and index selection. The **outcome** of a case "specifies what happened as a result of carrying out the solution or how the solution performed. Outcome includes both feedback from the world and interpretations of that feedback" [Kolodner 1993, p. 158].

Most design problems are large and complex [Maher et al. 1995]. To deal with such problems, designers often break them into smaller sub-problems. Solutions may be available for (sub)problems at any level forming a solution *hierarchy* [Maher et al. 1995]. Therefore, a hierarchically structured[12] case memory is desirable. The implication for a CBD system is that it has to provide designers with the capability to generate cases in an appropriate hierarchical representation.

5.1.2 Indexing and Retrieving Design Cases

An index is "a pointer (or indicator) to which a keyword (or label) is assigned and which leads to information about a specific and related topic in a large collection of data" [Rivard 1997, p. 112]. The indexing process is described by Kolodner (1993) as "assigning labels to cases at the time that they are entered into the case library to ensure that they can be retrieved at appropriate times."

Retrieving the most relevant design cases in a CBR system depends on an indexing mechanism with an efficient memory organization. Typical indexing problems have two parts: one part is the *indexing vocabulary* problem, which is how to decide what descriptors should be used for some classes of cases. This problem needs a careful domain analysis to find an appropriate terminology. The second part is the *indexing assignment* problem, which is the process of choosing identifying descriptors for a particular case [Kolodner 1993]. The classification scheme presented in Chapter 3 shows us how to solve the vocabulary problem in the present context. The assignment problem has to be handled by the prototype system developed in this thesis.

Retrieval consists of selecting the most relevant case to a current design situation. Based on appropriately constructed indices, the retrieval process searches case

12. A hierarchical structure is a formalized structure formed by decomposing a complicated configuration into sub-parts and basic components.

memory to find the relevant design cases [Maher et al. 1995]. To find the most relevant ones, the case selection process may use various strategies and algorithms for search and similarity measurement.

5.1.3 Adapting Design Cases

A retrieved case can offer a solution to the new situation. However, it often needs to be modified to fit the new problem. Adapting a selected design case to a set of new design requirements is typically a conflict resolution process that requires additional design knowledge [Lee et al. 1995]. Lee et al. (1995) suggested several design adaptation strategies including *dimensional* adjustment, *configurational* adjustment, and *topological* adjustment. In housing design, an example of dimensional adjustment is changing the length or width of a room. An example of a configurational adjustment is the addition or removal of a room in a plan. A topological adjustment may consist of changing the spatial relations between areas or switching the location of areas, for example, by switching the location of a kitchen and dining area.

Note that all these changes typically trigger adjustments in the dimensions of adjacent rooms and may have to be propagated through an entire layout.

5.2 Design Scenarios Meet CBD: An Integrated Approach

In Figure 15, I show all design scenarios in parallel and identify the building blocks that can be assisted by the main steps of CBD. The left column of this figure shows the building blocks of design scenarios S1 - S4 developed in Chapter 4. The right column shows where the main CBD steps are able to assist tasks in housing design.

5.2.1 Retrieval

Task 2 of design scenario 1 (thick outlines), '`Feasibility study by designer`', includes the task '`Refer to precedent(s)`'. Meeting 1 of design scenario 3 includes the task '`Find a house plan based on the client's specification`', which, in turn, contains the sub-task '`The sales person searches through the collection of house plan`'. All of these tasks can be supported by the retrieval step in CBD.

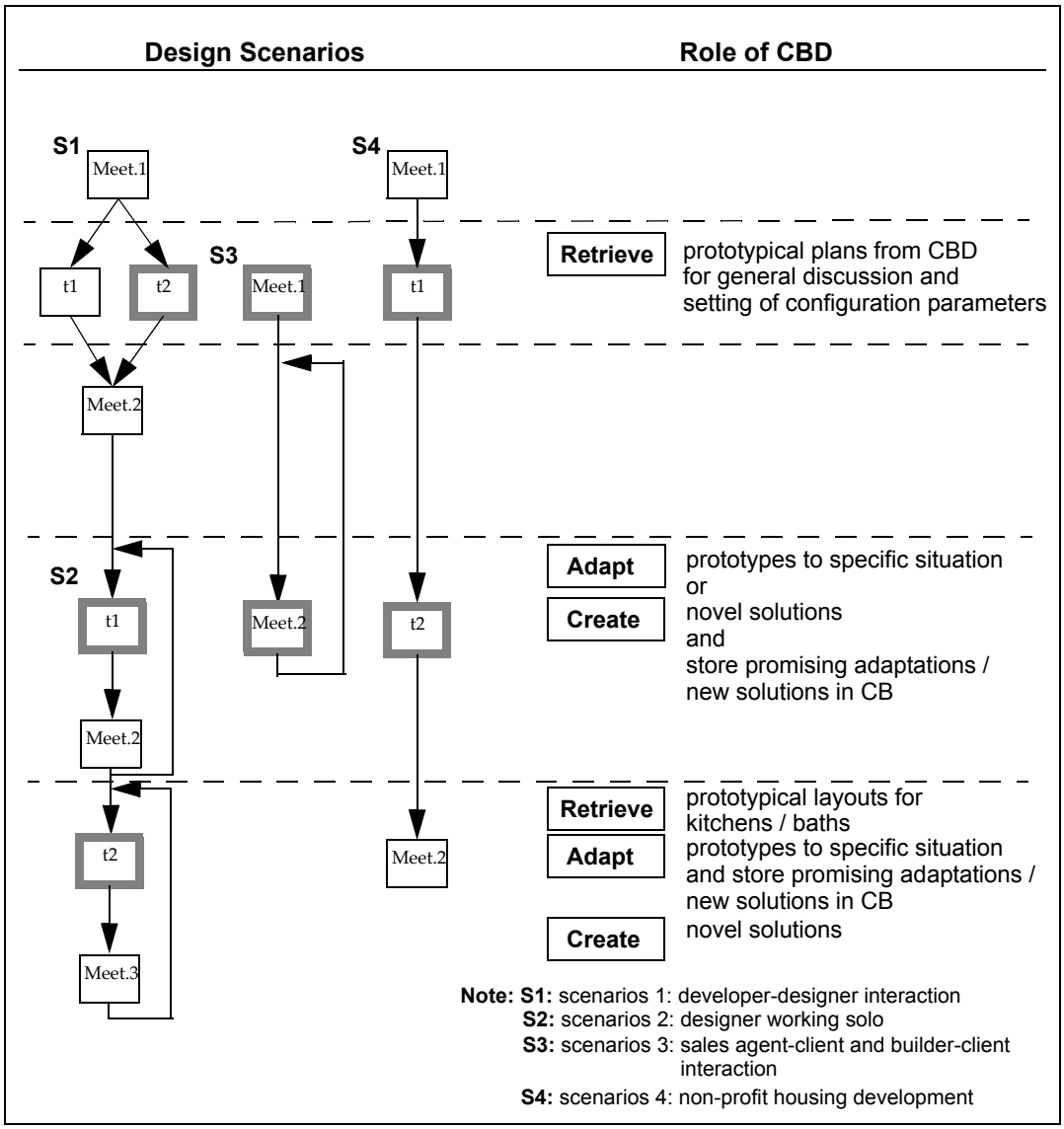


FIGURE 15. Design Scenarios vs. CBD

As I mentioned in section 5.2.2, classificatory types can provide a promising indexing scheme for the retrieval of housing precedents. The classification of housing precedents can incorporate orthogonal taxonomies so that a memory organization that supports *multiple classifications* is needed in order to implement such taxonomies. According to the housing types I defined in section 3.3, the same house can be retrieved as a raised ranch, a two-story house, or a 3-bedroom unit depending on which types designers or clients are interested in for a specific design problem. Note also that it is possible to *combine classifications* for retrieval, for example, search for a 3-bedroom raised ranch. Thus, in design scenario 3, the sales person should be able to combine classifications into a search index based on the client's specification.

Supporting *flexible classifications* is also important. Each party involved in housing design may have its own perspectives and interests when looking at a design case and may not want to use classifications already made by others. Therefore, it will be useful to have an extensible or adjustable indexing scheme to support different indexing vocabularies for different parties. That is, individuals can add new classification instances to the system, and also can modify existing instances. In the scenarios above, both the designer and the sales person may want to use the retrieval function in CBD. However, the sales person's interest may differ from the designer's. The sales person may want to retrieve best-selling house plans and to show them to a specific client quickly. On the other hand, the designer may be interested in retrieving house plans based on cost constraints.

5.2.2 Creation/Adaptation

Task 1 of design scenario 2 (thick outlines), '`Do schematic layout`', and Meeting 2 of design scenario 3 correspond to the creation or adaptation steps of CBD. A case-based system supporting these types of tasks for housing designers must be integrated with a CAD system that is able to represent, generate, and adapt the geometry and hierarchy of housing cases.

The system must first be able to represent and generate rapidly the geometry and hierarchical structure of *known* standard solutions. As I mentioned in section 1.1, standardized, well-known house plans from diverse sources can be put into the initial case base to 'seed' it. However, if the system requires considerable time and effort to collect and transfer the case knowledge to the system, this knowledge acquisition may become a bottleneck. Therefore, a CAD system that makes it possible to recreate housing layouts with ease for the explicit purpose of populating a case base is required.

The system should also be able to support designers when they search for novel solutions to novel problems. Generally, during the design process in the early phases, designers explore layout alternatives and variants of those alternatives. In housing, this becomes particularly important when designers confront new demands that cannot be satisfied by known configurations or by cases already in the case base. A CAD system integrated with a generative design capability has an advantage when the system has the capability to aid in the rapid generation of design representations. Utilizing a computer's inherent speed, designers may be able to generate, or generate faster, design solutions which they have never seen before and which may be innovative or novel.

Case adaptation is one of the most important issues in CBD. A retrieved case from the case base may not completely fit the given specific context and may need modifications. For example, the task '`Adapt the house plan to be built`' may require not only dimensional, but also configurational or topological adaptations as introduced in section 5.1.3. In order to facilitate this process, the CAD system associated with the CBD system should be able to propagate automatically changes triggered by these adaptations through an entire configuration. Note that no commercially available CAD system provides such a capability at present.

5.2.3 Summary

I have shown how all typical CBD steps can find their place and role in the housing design scenarios. The following list summarizes the issues raised in this section:

- Efficient classification and indexing mechanism into a case memory derived from classificatory types as described in section 3.3
- A case-base with special capabilities of classification and retrieval to store persistently the case information generated by a CBD system
- Effective methods for representing geometries of housing components and their relationships, for generating cases to be entered into the case base, including novel solutions for novel demands, and for adapting cases retrieved by the CBD system

5.3 Platform for a First Prototype Implementation

5.3.1 Database

The core of a CBR system is a case base or case library, essentially a database that is searchable under appropriate indices. The schema underlying the database must be able to capture cases in an appropriate representation.

Houses are volumetrically decomposed into components such as floors, zones, rooms as mentioned in section 3.2.2, which can be viewed as forming a hierarchical spatial containment structure. A housing case can be decomposed into sub-cases that are the hierarchical components of the house. For example, the first floor of a split-level house can be a complete case in its own right if a designer is interested in finding a precedent for just this part of the whole design. Therefore, the sub-cases must be independent entities and be retrievable as complete sub-solutions.

These hierarchical representations must be supported by a database that stores persistently the objects generated by the system. Relational database technology was developed for conventional business data-processing applications like inventory control, payroll, and accounts. This technology cannot, however, deal with a wide variety of other types of application such as CAD, CAE, CASE and CAM systems; knowledge-based systems; multimedia systems which deal with images, voice, and textual documents; and programming language systems [Kim 1990].

There are currently two proposed approaches for transitioning from relational database technology to the next-generation technology: *object-oriented databases* and *extended relational databases*. An object-oriented schema includes the object-oriented concepts of encapsulation, inheritance, and polymorphism. An object-oriented programming language may be extended into a unified programming and database language. By contrast, the extended relational approach starts with the relational model of data and a relational query language, and extends them in various ways to allow the modeling and manipulation of additional semantic relationships and database facilities. The case for the extended relational approach is based largely on its use of the familiar current-generation database technology. There is also a mathematical foundation for the query language and even an industry-wide standard for the database language [Kim 1990]. Since both the object-oriented and some of the extended relational databases can support the geometry and hierarchical representation for housing cases as envisioned here [Snyder et al. 1994], we can choose either of them as a database for housing.

5.3.2 The SEED Environment

The generic requirements derived in section 5.2, are satisfied to a large degree by the SEED development environment introduced in Section 1.2. The SEED-Database and SEED-Layout[13], both of which are implemented based on object-oriented concepts, are especially useful in this context. They have recently been integrated in a configuration called SL_Comm[14].

Below is a list of the features of SL_Comm that are particularly relevant with respect to the generic requirements.

- The SEED-Database supports multiple and flexible classification as mentioned in section 5.3.
- UniSQL, the object-relational database system used to implement the SEED-Database, can support the required geometry and hierarchical representation.
- SEED-Layout is a generative design system that is able to represent, and - under the control of the user - to generate and adapt design cases efficiently and propagate changes as described in section 5.2.

Figure 16 shows the architecture of SL_Comm. I describe below the components of SL_Comm in greater detail.

5.3.2.1 SEED-Layout

SEED-Layout is a generative design system for the creation of building layouts. When we have a standard solution for a recurring housing type, it is quite easy to reconstruct the Layout of the solution in the SEED-Layout format. This process allocates the components of the housing solution in the form of (rectangular) *Design Units*, each of which satisfies the requirements of an associated *Functional Unit* (Figure 17). A collection of Functional Units is called a *Layout Problem* or problem specification in SEED-Layout, which also provides designers with the capability to generate Layout Problems from scratch and to save or retrieve them from the SEED-Database.

13. a module of SEED that supports the generation of schematic Layouts of the Functional Units specified in an architectural program [Flemming and Chien 1995] p. 162.
14. Ongoing PhD work by Wen-Taw (Jonah) Tsai.

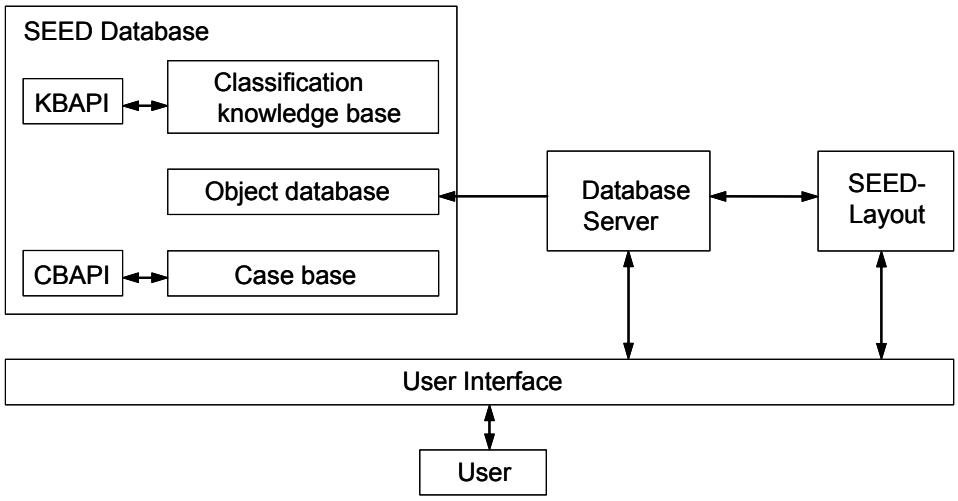


FIGURE 16. System architecture of SL_Comm

Given a Layout Problem, designers are able to place interactively the corresponding Design Units in a given context. An experienced SEED-Layout user often can do this in minutes. Once the generation is done, designers can store the Layout in the object database, from where it can be retrieved when needed.

SEED-Layout provides various mechanisms to generate and search for alternatives. Given a Layout Problem for which no good solutions are known, SEED-Layout assists designers trying to create a solution satisfying the problem systematically and quickly because it has access to explicitly stated requirements (in Functional Units) and is able to use them, for example, to size or resize Design Units. In addition, SEED-Layout allows designers to create alternatives in parallel by allocating Design Units associated with the same Functional Units in different places. For example, when the location of a living room is to be decided, the designer can develop alternative Layouts in parallel, for example, placing the living room in the front or rear and the remaining rooms in the remaining areas. SEED-Layout can also enumerate novel alternatives for new specifications exhaustively. It thus allows designers to investigate new and innovative housing design solutions quickly and easily.

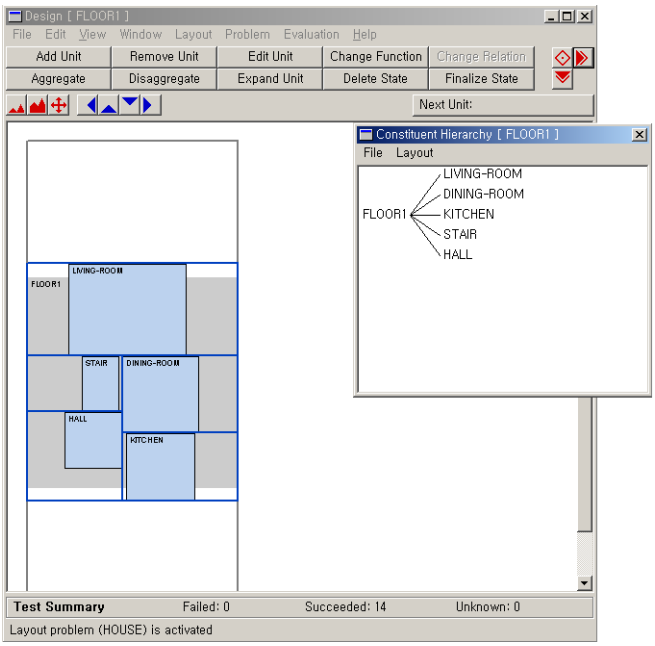


FIGURE 17. **A solution allocating a number of Functional Units in SEED-Layout**

SEED-Layout also provides the capability for case adaptation, one of the most difficult problems in CBD. In most of the current CBD systems, the retrieved case can be shown only as digitalized passive pictures, an essentially unstructured pixel-based representation that makes computer-supported adaptation very difficult, if not impossible, and adaptation is impossible. However, this problem can be solved by including generative capabilities like those of SEED-Layout in a CBD system, which supports interactive editing and expanding of Layout Problems and Layouts. SEED-Layout is able to do this because it can handle Functional Units that capture design requirements in a computable form. This feature enables SEED-Layout, for example, to automatically propagate changes caused by some modifications through an entire layout. Users can furthermore add, remove, and edit the requirements of Functional Units and ask SEED-Layout automatically to reshape the corresponding Design Units. The system thus offers designers instantly support for case adaptation.

5.3.2.2 Classification Knowledge Base

In order to deal with multiple and flexible classification described in Section 5.3, SEED contains a classification knowledge base (CKB) that is independent of the object database [Aygen 1998]. This “hybrid” approach separates precedent instances stored in the object database from the concepts they represent, which are expressed in the CKB.

The classification labels in the CKB may belong to multiple classification taxonomies that imply subsumption relations. That is, the SEED-CKB engine can infer subsumption relations from the classificatory taxonomies. The separation between CKB and object database also makes the SEED-CKB very flexible. Each user can create his/her own classifications and keep the respective knowledge bases separate from those of others according to his/her needs and interests.

Figure 18 illustrates the components of SEED-CKB[15]. The terminology used in CKB is based on the CLASSIC knowledge representation, after which the CKB has been modeled [Borgida et al. 1992]. A classification label is called a *primitive* in CLASSIC. It corresponds to the name of a classificatory type described in Chapter 3. Primitives can be arranged in taxonomies where higher level primitives subsume lower level ones. For example, the primitive ‘split-level’ can be defined so that it is subsumed by the primitive ‘residential’. A classification engine modeled after CLASSIC will retrieve any object labeled ‘split-level’ when it is asked to find any objects labeled ‘residential’.

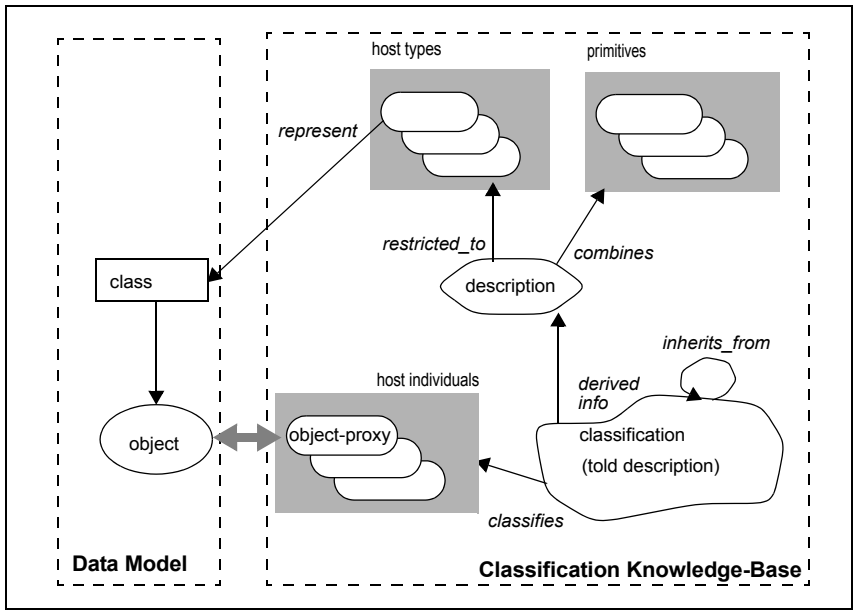
Primitives can be combined into *descriptions*, which may comprise primitives from different taxonomies. For example, we may construct a description combining the primitives ‘split-level’ and ‘sloped’, where the first belongs to a section-based taxonomy and the second to a topography-based taxonomy. Furthermore, descriptions can be restricted to selected classes of objects, called *host types* in CLASSIC. For example, we may restrict the above description to Layout Problems or Layouts, which prevents a user from erroneously attaching it to a Functional Unit.

CLASSIC and the CKB engine based on it are able to take a description and *normalize* it in the sense that it is augmented by all primitives that subsume the ones in the original description without redundancies. Such a construct is called a *classification*. Classifications can be modified by means of adding or retracting primitives [Aygen 1998]. A classification can be attached to an object in the object database. This happens by proxy; that is, the CKB uses the unique object

15. For detailed descriptions of SEED-CKB, refer to Aygen (1998) p. 45-52.

identifiers in the SEED-Database as references for the objects to which a classification is attached. In this way, different CKBs can attach different classifications to the same object. This is the basis for the flexibility with which SEED's classification component can be used by different designers.

The CKB is ideal to implement form-based classifications in the prototype, which will use it to attach such classifications to three classes of objects: Functional Units, Layout Problems, and Layouts. These classifications can be used for direct database retrieval or to construct automatically more complex indices or queries.



Source: Aygen (1998) p. 46

FIGURE 18. SEED-CKB

5.3.2.3 Object Database

The SEED object database is able to support the persistent storage of the housing precedents themselves, including their geometry and spatial hierarchy of the

components as described in section 5.2. An object may belong to a type or class hierarchy through which it inherits attributes and values from other objects in an object-oriented representation. It is important to keep in mind that this class hierarchy has nothing to do with the classifications and taxonomies in a CKB.

SEED-Layout provides a class hierarchy of Functional Units that can be used to represent the spatial hierarchy of a building. The spatial Functional Units are specialized into Building, MassingElement, Floor, Horizontal Zone, Vertical Zone, and Room (Figure 19).

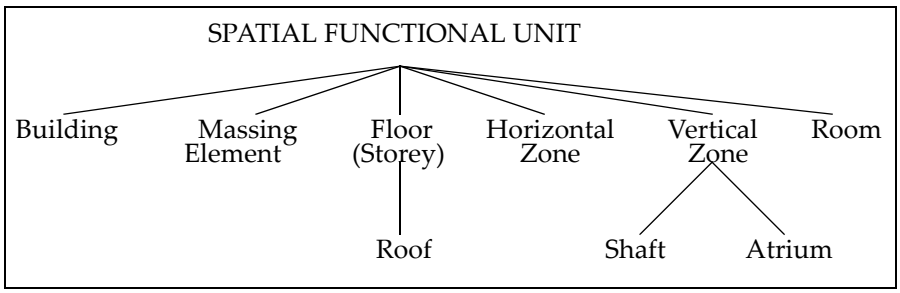


FIGURE 19. A class hierarchy of Functional Unit in SEED-Layout

In SEED-Layout, a Layout Problem is defined by a collection of Functional Units arranged in a spatial hierarchy, called a *constituent hierarchy*. SEED-Layout interprets constituent relationships strictly as spatial containment relations, which is a special form of a 'part-of' relationship. Figure 20 shows an example spatial hierarchy of a residential house. A spatial containment hierarchy is recursive in SEED-Layout, Therefore, any level and parts in a spatial hierarchy are retrievable directly from the database through querying.

I mentioned already in section 5.3.2.1 that a Layout in SEED-Layout is a collection of Design Units[16]. Each Design Unit is associated with a Functional Unit in a constituent hierarchy of a Layout Problem. If this Functional Unit contains Functional Unit constituents, the Design Unit contains a sub-Layout whose Design Units are associated with the constituents. In this way, the Layout/ sub-Layout relationships mirror the constituent hierarchy of the associated Functional Units. For example, in Figure 20, the 'public zone' Functional Unit

16. a continuous spatial area of a building with a specific geometry and location [Flemming and Aygen 2000]

has a ‘Living Room’ and ‘Dining Room’ Functional Unit. A corresponding Layout would contain at one level a Design Unit associated with the public zone, which contains a sub-Layout with Design Units associated with living room and dining room.

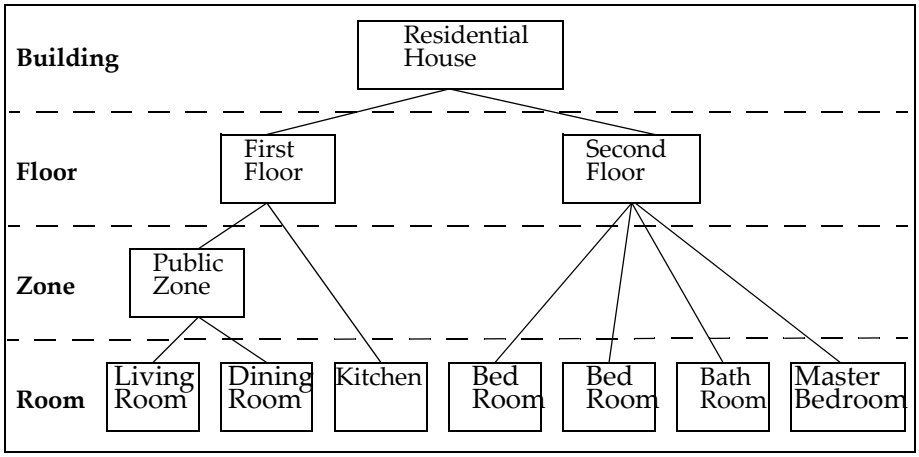


FIGURE 20. An example spatial hierarchy in SEED-Layout

The Design Units have a geometry, that is, location and dimensions. In this way, they capture the geometry of a Layout, whereas the Layout/sub-Layout relations reflect the spatial hierarchy as expressed in the constituent hierarchy of the associated Functional Units in the Layout Problem the Layout solves. Like a constituent hierarchy, the Layout/sub-Layout hierarchy is recursive; that is, a Layout at any level is formally complete and retrievable as such. Consequently, the representation used in SEED-Layout for Layout Problems and Layouts has exactly the hierarchical structure we are looking for to represent cases, and the object database captures these structures.

5.3.2.4 Case Base

SEED has a case base with additional capabilities beyond the object database and CKB. The case base schema defines the following concepts: *case*, *target*, and *match operators*.

What constitutes a case is highly domain-dependent, even in an environment like SEED, which may contain applications dealing with architectural programming or structural design in addition to SEED-Layout. Application developers are able to define cases as they please and use the CKB to classify them. But such classifications may not be sufficient to represent all aspects that an application may want to use for retrieval. CB therefore provides a target that can be used by any SEED-based application to attach additional information to a case. The SEED developers also realized that certain retrievals should take configurational aspects into account that may require matching on components independently of any classifications. The CB component therefore allows application developers to add match operators to a particular CB.

Since the generative capabilities available in SL_Comm are those of SEED-Layout and layout design is a crucial part of the initial design phase in housing design, the prototype system will consider cases mainly in the form of Layouts that can be retrieved through form- or component-based classifications attached to them. These classifications identify, at an abstract level, the type of problem the Layout solves. That is, these classifications represent the problem specifications part of a case, while the Layout represents the solution part.

Under this scheme, retrieval can be implemented in a computationally very efficient way because the CKB and CB engines are able to retrieve the object proxies associated with specific classifications or targets very efficiently. Another advantage of this scheme is that query indices can be computed automatically from a given Layout Problem if we allow Layout Problems to have the same classification as the Layouts that solve them: in this situation, the prototype can use the classification attached to the current Layout Problem as search index to find in the case base all Layouts with the same classification.

This degree of efficiency and automation induced me to prefer this scheme over a more elaborate one that seemed more natural at first. Since we have an explicit problem formulation in the form of a Layout Problem, we may attach the entire Layout Problem to a Layout as an index and retrieve cases by comparing the current Layout Problem to the Layout Problems attached to the Layouts in the case base and retrieving those Layouts with the closest matches. I rejected this scheme because Layout Problems consist of a, possibly rather complex, constituent hierarchy, and any matching algorithm would have to solve a computationally hard combinatorial problem that may well prove untractable.

There is no equivalent in this scheme to also include information resembling the outcome portion of the traditional problem/solution/outcome triad used to represent cases (see section 5.1.1). It is true that SEED-Layout evaluates each Layout and Layout modification against the requirements in the current Layout

Problem and makes these evaluations accessible to users through special windows. But the database schema does not capture these evaluations. It would be possible to use the target construct to record some of this information, but I did not pursue this for the current prototype because SEED-Layout generates this information anyway when a Layout is retrieved.

I do use the target to capture component-based classifications. In the current prototype, they are restricted to simple attributes like the number of bedrooms or bathrooms in a Layout, which can be automatically computed if the respective Functional Units in a Layout Problem are appropriately classified.

Note that the case base engine can maintain multiple case-bases in which different designers or organizations define and populate their cases, targets, and match operators and register the related objects from the object database as proxies.

5.3.3 System Architecture

SL_Comm and the CKB and CB in their present implementations are not sufficient to provide all the functionalities needed by the CBD system for housing as envisaged above. The capabilities of the CKB and CB are accessible through application programming interfaces (APIs) which, so far, have not been utilized by any application. There is also no graphical user interface (GUI) available that would allow designers to create classifications cases and attach them to objects created with SEED-Layout and stored in the object database. Finally, no mechanism exists to retrieve object identifiers from the object database so that they can be used as proxies by the CKB or CB.

Figure 21 shows the components that have to be added to SL_Comm in order to provide a complete CBD prototype:

- A GUI that makes those functions in the CKB and CB APIs that deal with the construction of classifications and cases and their association with objects generated by SEED-Layout directly accessible to users.

- A CKB manager and CB manager that control the information exchange between the GUI and the CKB and CB APIs. This intermittent layer separates the domain functionality proper from the user interface and is based on the model-view-controller separation, which is a hallmark of good object-oriented design. The managers furthermore interact with the database server to obtain object identifiers from the object database or retrieve cases from it.

- The target construct in the CB has to be augmented in order to represent the specific component-based attributes needed for the prototype.

I call this prototype SL_CB.

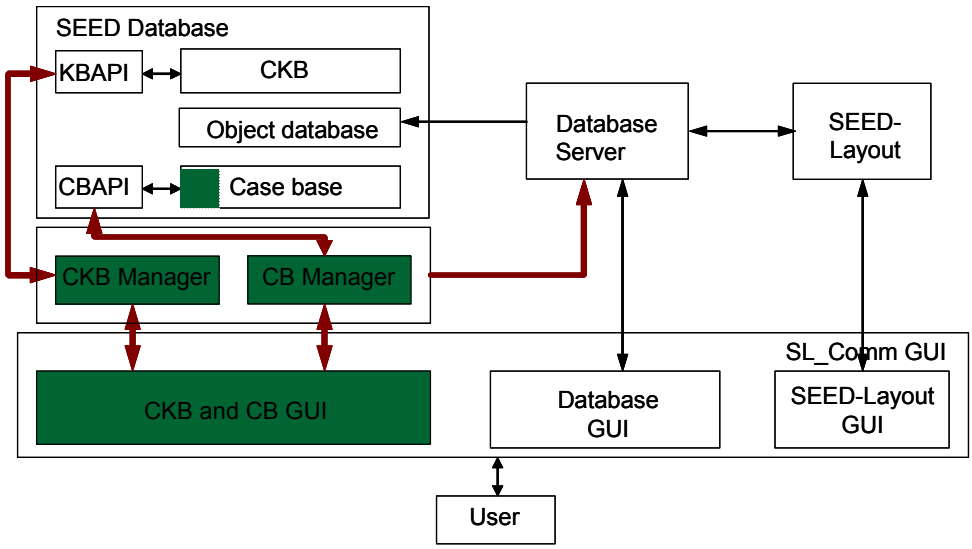


FIGURE 21. System architecture for the first prototype implementation, SL_CB

CHAPTER 6

Functional Specification and User Interface of a Prototype

A *requirement* is a feature that the system must have or a constraint that it must satisfy to be accepted by the client. *Requirements engineering* aims at defining the requirements of the system under construction. Requirements engineering includes two main activities; *requirements elicitation*, which results in the specification of the system that the client understands, and *analysis*, which results in an analysis model that the developers can unambiguously interpret.
- Bernd Bruegge & Allen H. Dutoit, *Object-oriented software engineering*

Based on SL_CB as platform for a prototype implementation, I have developed use cases describing the functionality and designed a graphical user interface (GUI) of SL_CB. In this chapter, I first introduce briefly use case-driven software development. I then introduce in detail the use cases that describe the functionality and GUI specific to SL_CB.

6.1 Use Case-Driven Software Development

Software developers must define at the outset what the software is supposed to accomplish. This is often called “behavior modeling”. The *use case model* introduced by Jacobson et al. (1994) uses **actors** and **use cases** to aid in defining who makes use of the system (actors) and what they can do with the system (use cases).

A use case is a “sequence of actions an actor performs using a system to achieve a particular goal” [Rosenberg 1999, p. 38]. *Actors* represent external entities that interact with the system. An actor can be human or an external system [Bruegge and Dutoit 2000, p. 106]. The following example illustrates these concepts.

Use Case: View classification hierarchy

1. The designer issues the “Classification Hierarchy” command from the command bar in the CB window.
2. SL_CB displays in the CB Window a tree view of the classification hierarchy to which the currently active classification belongs.

In this use case, the designer is the actor and the sequence of actions describes how this actor can accomplish as specific task, in this case, to create a specific display.

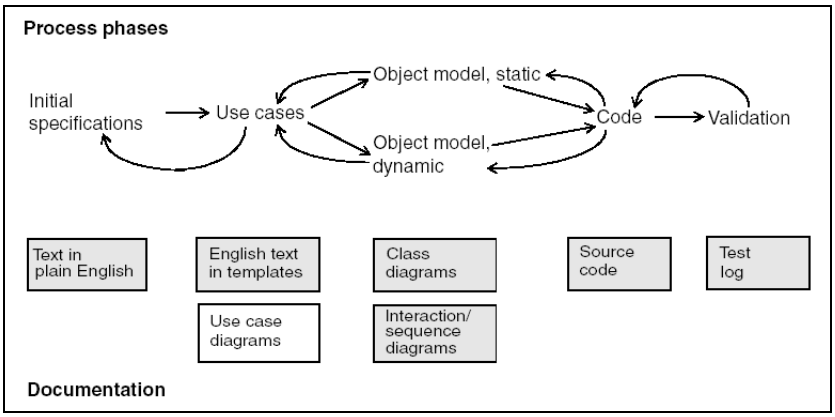
A use case-driven approach to software development is based on the assumption that a system can be described in a number of different views represented by the different actors. Each of those views leads to a set of tasks the actors have to be able to perform with the help of the system. When we consider all of the different actors and associated tasks, we arrive at a set of use cases that describe the entire functionality of the system from the end users’ point of view. From the system developer’s perspective, the use cases specify the functional requirements the system has to satisfy.

The use case-driven software development process specifies a series of ‘models’ within the overall iterative process: requirement, analysis (sometimes called the static object model), design (sometimes called the dynamic object model), implementation, and testing model. Derived from the requirements model described through the use cases, the analysis model identifies the classes and their relations that must be instantiated to capture all data of interest. The design model identifies the object instances that must interact to deliver the functionalities of the individual use cases. The implementation model includes the source code written in a specific programming language. Finally, the testing model consists of documentation of test specifications and test results for each use case. Figure 22 shows the phases of the process and the products each phase creates.

In short, use case specifications govern not only the requirements model, but also work on the analysis, design, implementation, and test models. Currently, it is the approach of choice when it comes to object-oriented software development.

6.2 Overview of Use Cases

This section provides an overview of the use cases that I defined to deliver the functionality outlined in Chapter 5.



Source: Flemming et al. (2001) p.12

FIGURE 22. Phases and products of use case-driven software development

6.2.1 Case Creation

The capabilities of SEED-Layout are used in SL_CB to create cases and all related information: Functional Units, Layout Problems, and Layouts. Flemming et al. (2000) describe the use cases that deliver the functionality to designers. Readers are also referred to Flemming (1999) and Flemming and Chien (1998) for a *SEED-Layout Tutorial* and *SEED-Layout Reference Manual*, respectively. These descriptions are not included in the present document.

6.2.2 Indexing and Retrieving Cases

The use cases covering this aspect of SL_CB fall into following two groups:

- **Primitives and classifications**. This group contains the use cases that describe how designers can create primitives and classifications, attach them to objects created by SEED-Layout and retrieve objects with these classifications from the object database.

- **Cases**. This group contain the use cases dealing with the creation and retrieval of cases, where a case is a Layout, to which form- and component-based classifications have been attached.

These use cases and the respective GUIs are described in detail in the bulk of this chapter. Class and sequence diagrams covering the same use cases can be found in Appendix B and C.

The template used to describe use cases expands the format used in the example above. It starts each use case with a brief statement that summarizes the task. It then states the preconditions that much be met if the use case is to be executed. The main portion lists the individual steps that comprise the use case, possibly by distinguishing a basic course from one or several alternative courses. GUI snapshots augment the descriptions. This format has been borrowed from Flemming et al. (2001).

Since the tasks covered by these use cases are closely related, it seems unnecessary to distinguish between several actors in their execution. I therefore describe use cases for a single, generic actor called ‘user’.

6.2.3 Adapting Cases

SL_CB again relies entirely on SEED-Layout for this capability. The use cases describing these capabilities of SEED-Layout can be found in the literature cited in section 6.2.1.

6.3 Functional Specification and User Interface

6.3.1 Primitives and Classifications

I argued in Chapter 3 and 5 that the classification engine of CKB is capable of capturing form-based features of a case. I have developed twenty-five use cases that make the capabilities of CKB available to designers.

6.3.1.1 Session Handling

The capabilities of CKB are accessible to designers from a general window called CKB Window. All actions that can be performed while a single CKB Window is open constitute a classification *session*.

1. Start a classification session

The user starts a classification session in order to employ the specific functionalities of the classification engine of CKB.

Preconditions:
1. The SL_Comm Window (SCW) is open.

Basic Course:
1. The user selects the “Classification” option from the “Communication” menu in the SCW menu bar.
2. SL_CB opens the “Classification Login” dialog box. It prompts the user to select a database type and to input a database name, username and password.
3. If the three input values are all correct, SL_CB connects to the database and opens the Classification Knowledge Base (CKB) Window; otherwise, it shows an appropriate error message and returns to step 2.
4. The user may exit this use case anytime by hitting the “Cancel” button in the “Classification Login” dialog box.

2. End a classification session

The user ends a classification session and closes the CKB Window.

Preconditions:
1. The CKB Window is open.

Basic Course:
1. The user selects the “Close” option from the “File” menu.
2. If changes have been made after the last save, SL_CB asks the user if the changes should be saved and acts accordingly.
3. SL_CB closes the CKB Window and ends the classification session.

6.3.1.2 CKB Administration

A specific CKB instance contains all primitives, restrictions, classifications, and classified object proxies that a specific designer or group of designers uses to index and retrieve objects from the object database. All CKBs are identified by names.

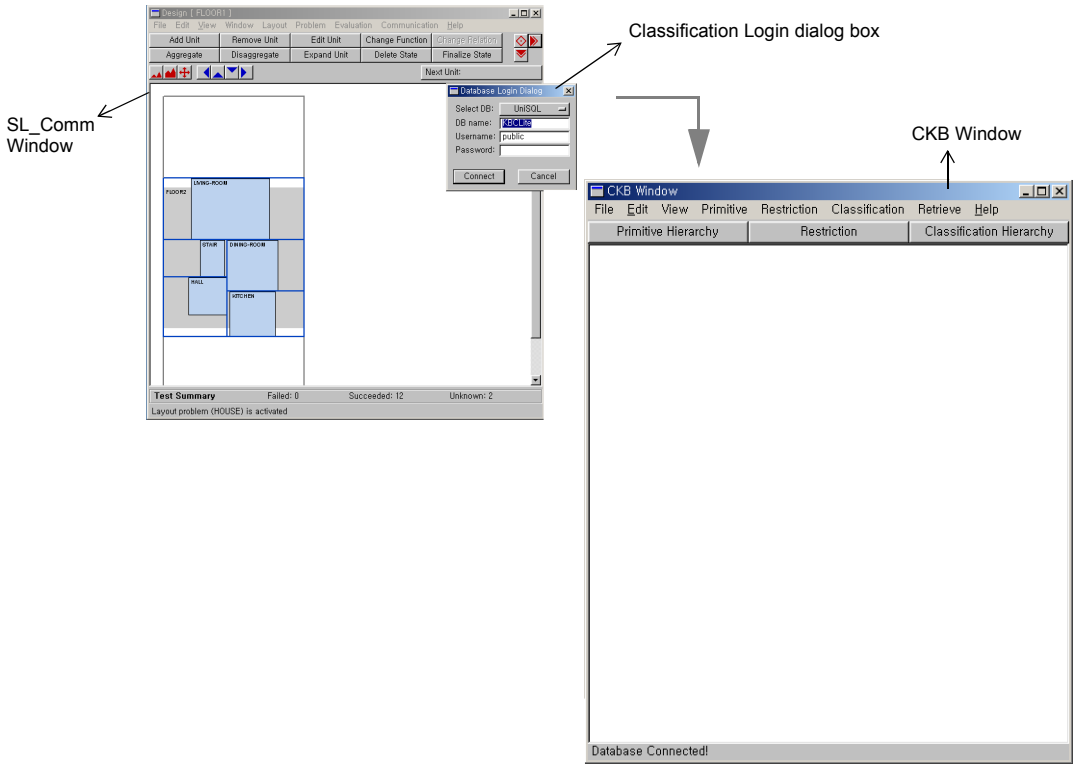


FIGURE 23. **Start a classification session**

3. Create a CKB

The user creates a new CKB instance and gives it a name.

Preconditions:
1. SL_CB is connected to a database.
2. The CKB Window is open.

Basic Course:
1. The user selects the “New CKB” option from the “File” menu.
2. SL_CB opens the “New Knowledge Base” dialog box (see Figure 24).
3. The user enters a name in the “New KB Name:” field and hits the “Commit” button.

4. SL_CB creates a new CKB in working memory, provided a CKB with the same name does not exist in the database; otherwise, SL_CB shows an error message and the use case returns to step 3.
5. The user may exit this use case anytime by hitting the “Cancel” button in the “New Knowledge Base” dialog box.

Comments:
This use case creates a new CKB only in working memory; it does not save it persistently in the database. When the user tries to create a CKB again or load a CKB without saving the first one, SL_CB asks if the user wants to save or dismiss the former one.

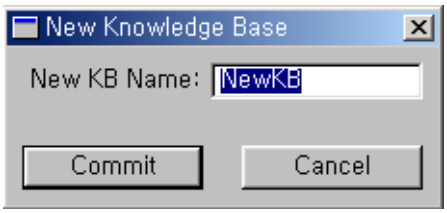


FIGURE 24. **New knowledge base dialog box**

4. Load a CKB

Upon the user’s request, SL_CB loads a previously created CKB, using its name for identification.

Preconditions:
1. SL_CB is connected to a database.
2. The CKB Window is open.

Basic Course:
1. The user selects the “Load CKB” option from the “File” menu.
2. SL_CB opens the “Load Knowledge Base” dialog box showing a list of existing CKB names in the connected database (see Figure 25).
3. The user selects a desired CKB by name and hits the “Commit” button.
4. SL_CB retrieves the respective CKB from the database and makes it the currently active one.
5. The user may exit this use case anytime by hitting the “Cancel” button in the “Load Knowledge Base” dialog box.

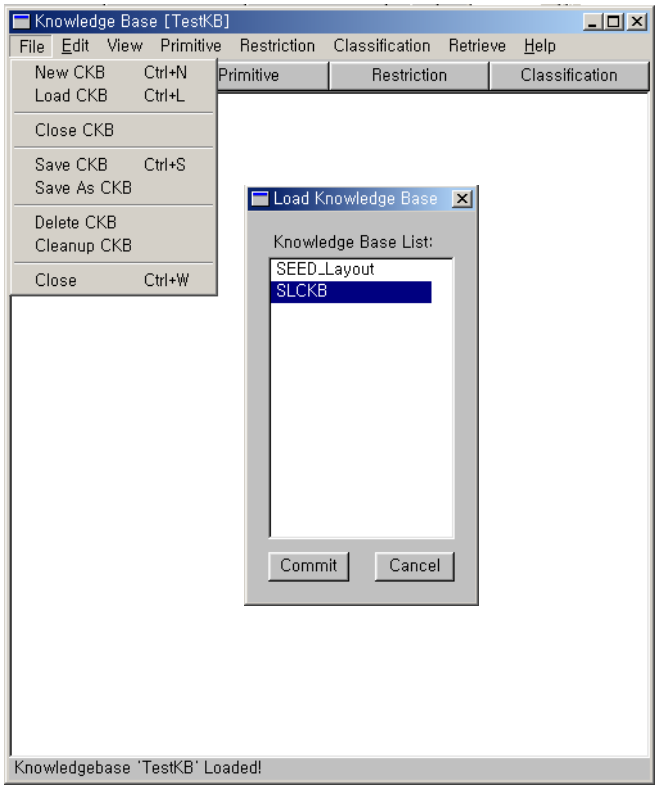


FIGURE 25. **Load a CKB**

5. Close a CKB

Upon the user's request, SL_CB closes the active CKB.

Preconditions:
1. A CKB is active.

Basic Course:
1. The user selects the "Close CKB" option from the "File" menu.
2. If changes have been made after the last save, the SL_CB asks the user if the changes should be saved and acts accordingly.
3. The currently active CKB is closed.

6. Save a CKB

Upon the user's request, SL_CB saves the active CKB in the database.

Preconditions:
1. A CKB is active.

Basic Course:
1. The user selects the "Save CKB" option from the "File" menu.
2. SL_CB saves the currently active CKB in the persistent database.
3. If changes have not been made after the last save, SL_CB lets the user know the fact.

7. Save a CKB as

Upon the user's request, SL_CB saves the active CKB in the database under a different name.

Preconditions:
1. A CKB is active.

Basic Course:
1. The user selects the "Save As CKB" option from the "File" menu.
2. SL_CB opens the "Save As Knowledge Base" dialog box.
3. The user enters a desired name in the "New KB Name:" field and hits the "Commit" button.
4. SL_CB saves the currently active CKB under the new name in the persistent database, provided a CKB with the same name does not exist in the database; otherwise, SL_CB shows an error message and prompts the user again to input another name.
5. The user may exit this use case anytime by hitting the "Cancel" button in the "Save As Knowledge Base" dialog box.

8. Delete a CKB

Upon the user's request, SL_CB deletes a specific CKB in the database.

Preconditions:
1. The CKB Window is open.

Basic Course:
1. The user selects the “Delete CKB” option from the “File” menu.
2. SL_CB opens the “Delete Knowledge Base” dialog box showing a list of existing CKB names in the connected database.
3. The user selects a name to be deleted and hits the “Delete” button.
4. SL_CB prompts the user to confirm the deletion.
5. If the user confirms and the selected CKB is not currently active, SL_CB deletes that CKB from the database along with all primitives, restrictions, DB objects, and classifications associated with it; otherwise, SL_CB dismisses this use case.
6. The user may exit this use case anytime by hitting the “Cancel” button in the “Delete Knowledge Base” dialog box.

9. Cleanup a CKB

Upon the user’s request, SL_CB cleans up the active CKB.

Preconditions:
1. A CKB is active.

Basic Course:
1. The user selects the “Cleanup CKB” option from the “File” menu.
2. SL_CB asks the user to confirm the clean-up.
3. If the user confirms, SL_CB discards the primitives, restrictions, DB objects, and classifications associated with the currently active CKB; otherwise, the use case terminates.

6.3.1.3 Handling of Primitives

10. View primitive hierarchy

Upon the user’s request, SL_CB displays a specific taxonomy as a tree of primitives. The view also indicates all subsumption relations between primitives and whether they are disjunct or disjoined.

Preconditions:
1. A CKB is active.
2. At least one primitive has been created in the CKB.

Basic Course:
1. The user issues the “Primitive Hierarchy” command in the SL_CB commands box.
2. SL_CB displays a tree view of the primitive hierarchy in the currently active CKB (see Figure 26).

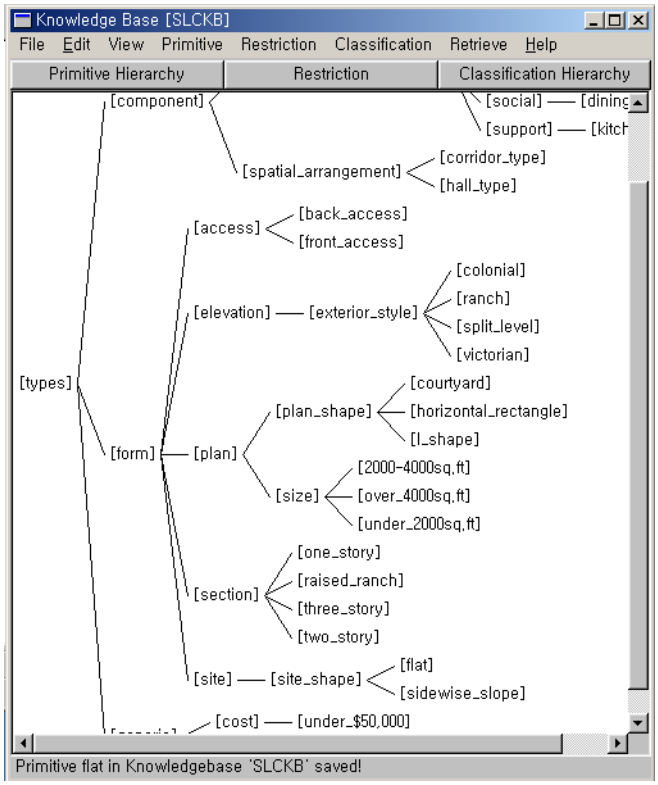


FIGURE 26. **View primitive hierarchy**

11. Create a primitive

The user creates a primitive by determining its name, relations to existing primitives, and its type.

Preconditions:
1. A CKB is active.
2. A tree view of the primitive hierarchy is displayed.

Basic Course:
1. The user selects a primitive from the tree view hierarchy with the left mouse button and clicks the right mouse button to display the pop-up menu.
2. The user selects “Add” option to create a new primitive (see Figure 27).
3. SL_CB opens the “Add Primitive” dialog box with the super primitive - the first selected primitive -.
4. The user enters a name in the “New Primitive Name:” field and chooses its type (simple or disjoint).
5. The user hits the “Commit” button.
6. If a primitive with the same name does not exist in the database, SL_CB creates a new primitive with the given name in the persistent database and displays the new primitive in the correct position in an updated tree view; otherwise, SL_CB shows an error message and prompts the user to enter another name.
7. The user may exit this use case anytime by hitting the “Cancel” button in the “New Primitive” dialog box.

Alternate course:
1. The user selects the “New Primitive” option from the “Primitive” menu.
2. SL_CB opens the “New Primitive” dialog box.
3. The user enters a name in the “New Primitive Name:” field, chooses its super primitive from a list of existing primitives in the current CKB, and chooses its type (simple or disjoint).
4. The use case continues with step 5 of the basic course.

Comments:
The basic course can save time because the user already knows where the primitive is placed in the tree view hierarchy and the super primitive can be decided automatically. This way is more intuitive.

12. Edit a primitive

The user modifies the attributes, relations with other primitives, or type of a specific primitive.

Preconditions:
1. A CKB is active.
2. At least one primitive has been created in this CKB.
3. A tree view of the primitive hierarchy is displayed.

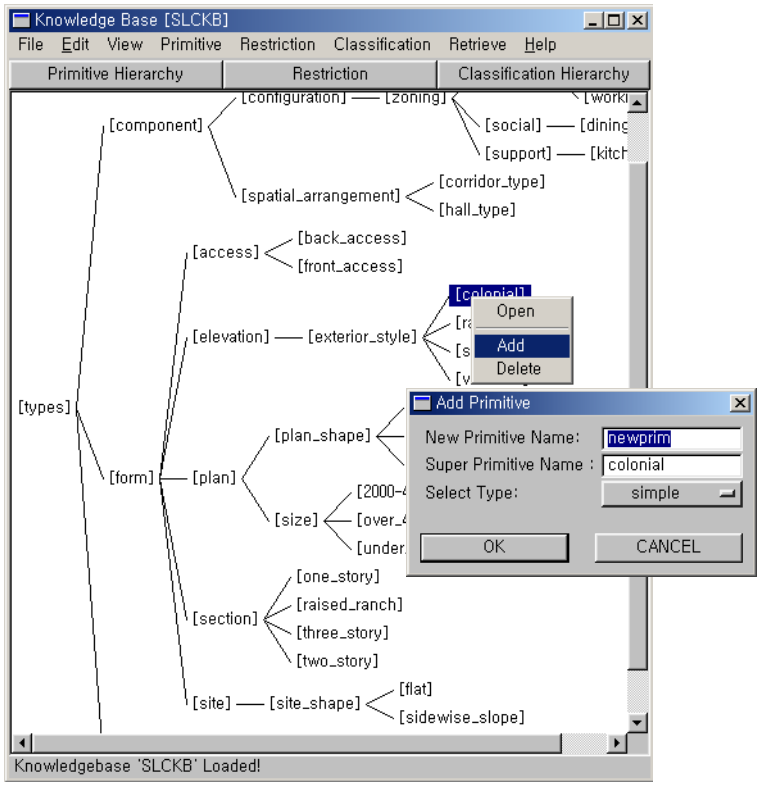


FIGURE 27. **Create a primitive**

Basic Course:

1. The user selects the “Edit Primitive” option from the “Primitive” menu.
2. SL_CB opens the “Edit Primitive” settings box, which displays - in the “Primitive List” field - the names of all primitives currently defined in the active CKB (see Figure 28).
3. The user selects a primitive in the list.
4. SL_CB displays the name of itself in the “Primitive Name” field, the name of its super primitive in the “Select Super Primitive” button, and the primitive type in the “Select Type” button.
5. The user types the new primitive name, or changes the super primitive, or primitive type as desired.
6. The user hits the “Commit” button.
7. SL_CB saves the changed attributes.

8. The user may exit this use case anytime by hitting the “Cancel” button in the “Edit Primitive” settings box.

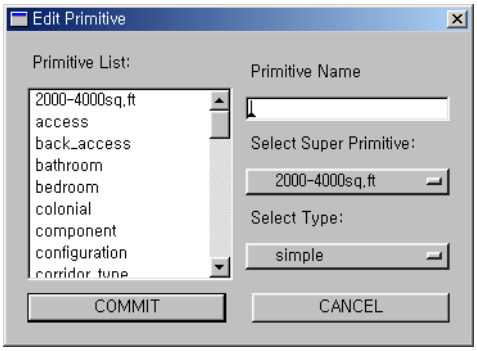


FIGURE 28. **Edit primitive settings box**

13. Delete a primitive

Upon the user’s request, SL_CB deletes a selected primitive in the active CKB.

Preconditions:
1. A CKB is active.
2. At least one primitive has been created in the CKB.
3. A tree view of the primitive hierarchy is displayed.

Basic Course:
1. The user selects a primitive from the tree view hierarchy with the left mouse button and clicks the right mouse button to display the pop-up menu.
2. The user selects “Delete” option.
3. SL_CB displays an Alert box asking the user if the primitive should be deleted.
4. If the user confirms, SL_CB deletes the primitive and updates the display of the primitive hierarchy.

Alternate course:
1. The user selects the “Delete Primitive” option from the “Primitive” menu.
2. SL_CB opens the “Delete Primitive” dialog box, with displays - in the “Primitive List” field - the names of all primitives currently defined in the active CKB.

3. The user selects a primitive name in the list and hits the “Delete” button.
4. The use case continues with step 3 of the basic course.
5. The user may exit this use case anytime by hitting the “Cancel” button in the “Delete Primitive” dialog box.

Comments:
Deleting a primitive is also deleting all its derived sub-primitives because the primitive consists of subsumption hierarchy relationships.

14. Get derived information

The user displays all derived information for a primitive, such as the primitives it subsumes or is subsumed by.

Preconditions:
1. A CKB is active.
2. At least one primitive has been created in the CKB.

Basic Course:
1. The user selects the “Get Primitive Info” option from the “Primitive” menu.
2. SL_CB opens the “Primitive Information” settings box, which shows the names of the primitives currently defined in the “Primitive List” field in the active CKB (see Figure 29).
3. The user selects a primitive in the list.
4. The settings box displays the “super primitive” and “sub primitives” list for the selected primitive.
5. The user inspects the current settings and closes the settings box.

Alternate course:
1. If the primitive hierarchy is currently not displayed in the CKB Window, the user displays it by selecting the “Primitive Hierarchy” command in the SL_CB command bar (use case 10).
2. The user selects a primitive from the tree view hierarchy with the left mouse button and clicks the right mouse button to display the pop-up menu.
3. The user selects “Open” option.
4. SL_CB displays the derived information of the selected primitive.
5. The user inspects the current settings and closes the information box.

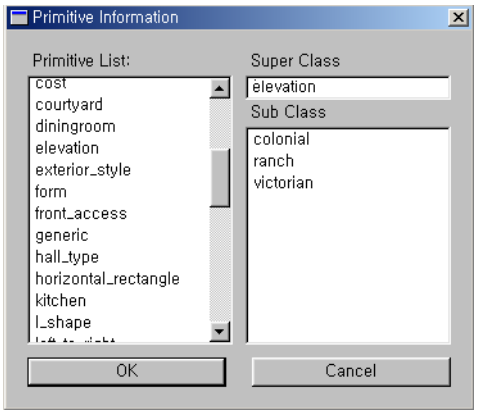


FIGURE 29. **Primitive information settings box**

6.3.1.4 Handling of Classifications

15. Create a classification

The user creates a new classification by giving it a name and selecting the primitives it encompasses and restrictions that apply.

Preconditions:
1. A CKB is active.

Basic Course:
1. The user selects the “New Classification” option from the “Classification” menu in the CKB Window.
2. SL_CB opens the “New Classification” settings box (see Figure 30).
3. The user inputs a classification name in the “Classification Name:” field; chooses its super classification from the “Super Classification List:”; chooses the primitives from the “Primitive List:”; and selects a restriction from the “Restriction List:”.
4. The user hits the “Commit” button.
5. If no classification with the same name exists in the active CKB and the selected super classifications do not have inheritance conflicts each other, SL_CB creates the new classification; otherwise, SL_CB displays an appropriate error message and the use case returns to step 3.

6. The user may exit this use case anytime by hitting the “Cancel” button in the “New Classification” dialog box.

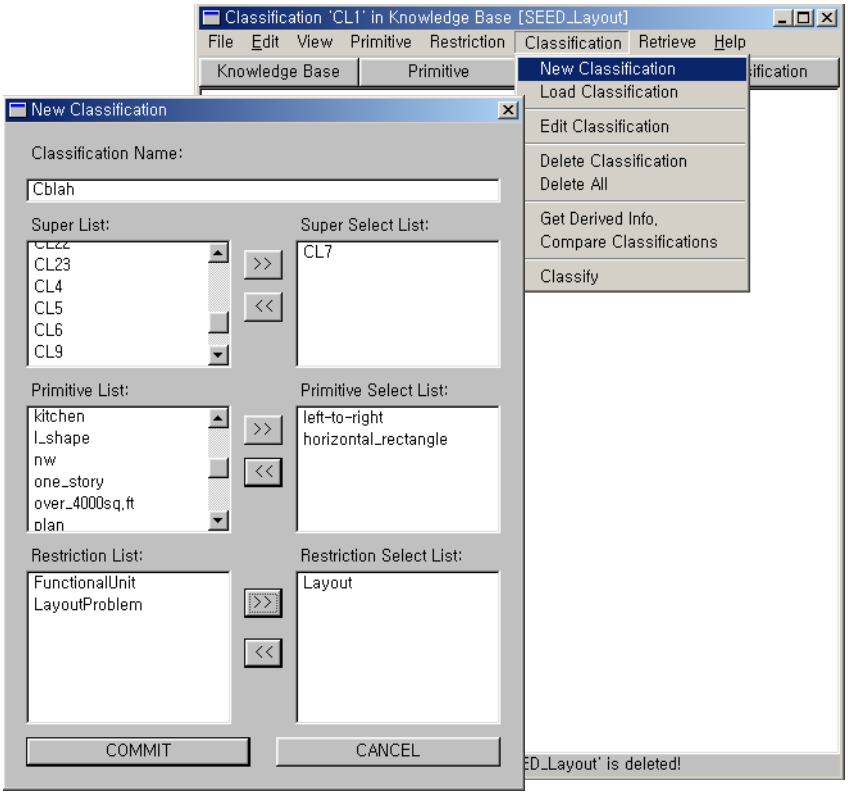


FIGURE 30. Create a classification

16. Load a classification

Upon the user’s request, SL_CB loads a previously created classification, using its name for identification. The settings box also displays all the parents, primitives, and restrictions information.

Preconditions:
1. A CKB is active.
2. At least one classification has been created in the active CKB.

Basic Course:
1. The user selects the “Load Classification” option from the “Classification” menu.
2. SL_CB opens the “Load Classification” settings box which shows in the “Classification List:” field a list of the names of the classifications in the active CKB.
3. The user selects a classification name.
4. SL_CB displays in the settings box the parent, primitives, and restrictions of the selected classification.
5. The user hits the “Commit” button.
6. SL_CB retrieves the classification from the database and makes it the current active one.
7. The user may exit this use case anytime by hitting the “Cancel” button in the “Load Classification” settings box.

17. View classification hierarchy

Upon the user’s request, SL_CB creates a display of the classification hierarchy to which the currently active classification belongs.

Preconditions:
1. A classification is active.

Basic Course:
1. The user issues the “Classification Hierarchy” command in the CKB commands box.
2. SL_CB displays in the CKB Window a tree view of the classification hierarchy to which the currently active classification belongs.

18. Edit a classification

The user edits the currently active classification by adding/removing its parents, primitives, or restrictions. The box also displays its subsumers, subsumees, and synonyms information.

Preconditions:
1. A classification is active.

Basic Course:
1. The user selects the “Edit Classification” option from the “Classification” menu.

2. SL_CB opens the “Edit Classification” settings box, which displays - in the “Classification Name:” field - the active classification name, the parent of the active classification in the “Super List:” field, the primitives it encompasses in the “Primitive List:” field, and the restrictions that apply in the “Restriction List:” field. It also displays its derived information such as its subsumers, subsumees, and synonyms (see Figure 31).
3. The user types the new classification name, or changes the super classifications, primitives, or restrictions as desired.
4. The user hits the “Commit” button.
5. If no classification with the same name exists in the active CKB and the selected super classifications do not have inheritance conflicts each other, SL_CB saves the changed settings; otherwise, SL_CB displays an error message and the use case returns to step 2.
6. The user may exit this use case anytime by hitting the “Cancel” button in the “Edit Classification” settings box.

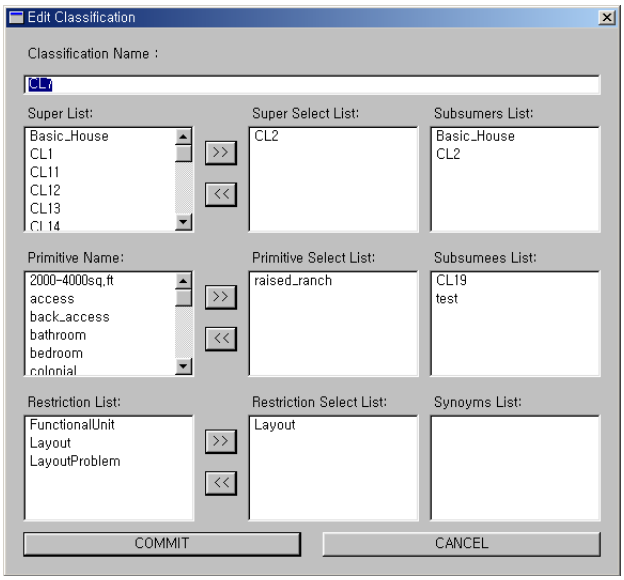


FIGURE 31. Edit classification settings box

19. Delete a classification

Upon the user's request, SL_CB deletes a specific classification and propagates the changes to the effected classifications. It also removes all pairings between that classification and the objects to which it is attached.

Preconditions:
1. A CKB is active.
2. At least one classification has been created in the CKB.

Basic Course:
1. The user selects the "Delete Classification" option from the "Classification" menu.
2. SL_CB opens the "Delete Classification" dialog box, which shows - in the "Classification List:" field - the names of the classifications in the active CKB (see Figure 32).
3. The user selects a classification name and hits the "Delete" button.
4. If the selected classification is not currently active, SL_CB displays an Alert box prompting the user to confirm the request; otherwise, SL_CB displays an error message and the use case returns to step 3.
5. If the user confirms, SL_CB deletes the classification and all classifications it subsumes along with all pairings between these classifications and the objects to which they are attached. SL_CB also updates the tree view of the current classification hierarchy if it is shown in the CKB Window.
6. The user may exit this use case anytime by hitting the "Cancel" button in the "Delete Classification" dialog box.

20. Delete all classifications

Upon the user's request, SL_CB deletes all classifications in the active CKB.

Preconditions:
1. A CKB is active.

Basic Course:
1. The user selects the "Delete All" option from the "Classification" menu.
2. SL_CB opens an Alert box prompting the user to confirm the request.
3. If the user confirms, SL_CB deletes all classifications along with all pairings between these classifications and the objects to which they are attached.

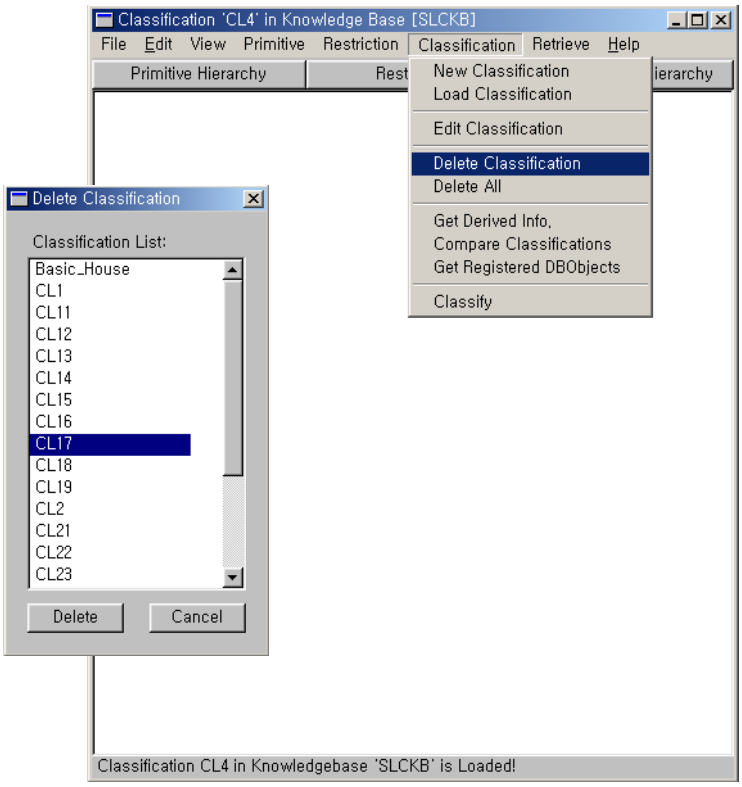


FIGURE 32. **Delete a classification**

21. Get derived information

Upon the user's request, SL_CB displays all derived information for a classification, such as the classifications it subsumes or is subsumed by.

Preconditions:
1. A CKB is active.
2. At least one classification has been created in the CKB.

Basic Course:
1. The user selects the "Get Derived Info" option from the "Classification" menu.
2. SL_CB opens the "Derived Classification" settings box (see Figure 33).

3. The user selects a classification in the list.
4. The settings box displays the “Derived Primitives”, “Derived Restrictions”, “Subsumers”, “Subsumees”, and “Synonyms” list for the selected classification.
5. The user inspects the current settings and closes the settings box.

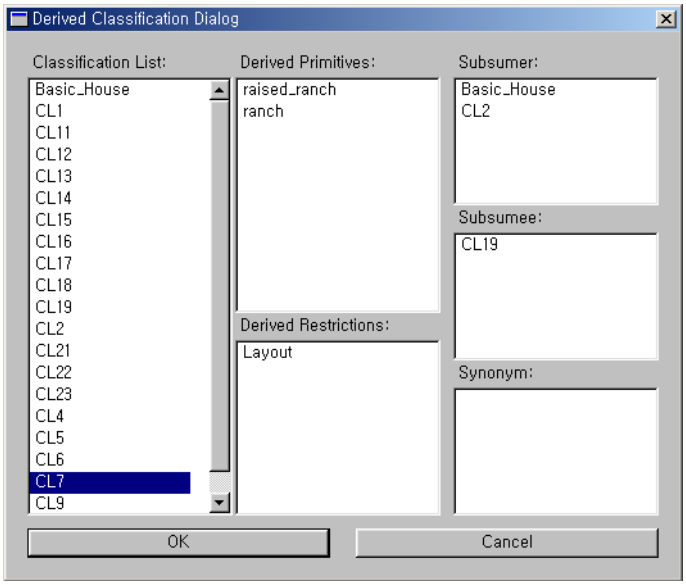


FIGURE 33. **Classification derived information settings box**

22. Compare classifications

Upon the user’s request, SL_CB compares a classification to others by displaying the comparison types.

Preconditions:
1. A CKB is active.
2. At least two classifications have been created in the active CKB.

Basic Course:
1. The user selects the “Compare Classifications” option from the “Classification” menu.
2. SL_CB opens the “Compare Classification” settings box, which shows a list of all classification names in the active CKB (see Figure 34).

3. The user selects two classifications and hits the “Compare” button.
4. SL_CB displays the attributes of the selected classifications such as SUBSUMER, SUBSUMEE, EQUIVALENT, EQUAL, DISJOINED, and DISTINCT.
5. The user may exit this use case anytime by hitting the “Cancel” button in the “Compare Classifications” dialog box.

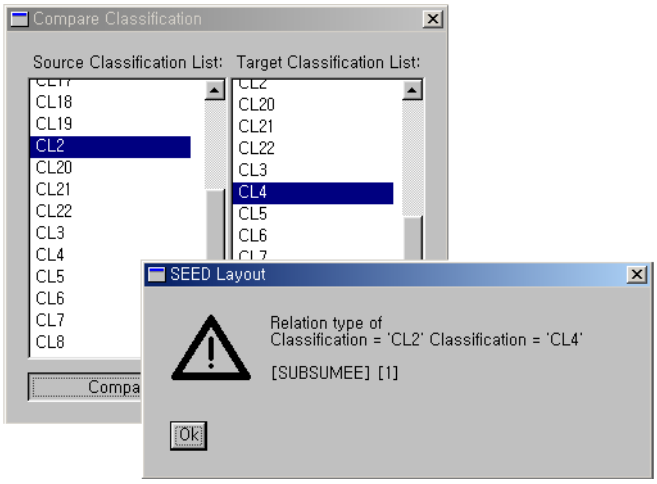


FIGURE 34. **Compare classification settings box and the attribute**

23. Attach a classification

The user attaches a classification to the active Layout or Layout Problem (LP) or to a selected Functional Unit (FU).

Preconditions:
1. SL_CB has loaded a LP from a file or the database. This implies that an active layout exists. if the user wants to classify a Functional Unit, a Unit must have been selected.
2. SL_CB is connected to the object database.
3. A CKB is active.
4. At least one classification has been created in the active CKB.

Basic Course:
1. The user selects the “Classify” option from the “Classification” menu.

2. SL_CB opens the “Classify” settings box, which shows a lists of the names of all classifications in the active CKB (see Figure 35).
3. The user selects a classification.
4. SL_CB displays all of the attributes for the selected classification.
5. The user selects the type (or class) of object to which the selected classification should be attached and hits the “Classify” button.
6. If the user selected “FU” but no FU has been selected, SL_CB displays an error message and aborts the use case. If the selected classification has a restriction that excludes the selected object type, SL_CB displays an error message and the use case returns to step 3 or 5. If the active Layout/LP or selected FU has already an attached classification, SL_CB asks the user in a special Alert box if the existing classification should be replaced by the selected one. If the user does not confirm this, the use case returns to step 3 or 5. In all other cases, the use case continues with step 7.
7. Depending on the user’s selection, SL_CB attaches the selected classification to Layout, LP, or FU.
8. The user may exit this use case anytime by hitting the “Cancel” button in the “Classify” dialog box.

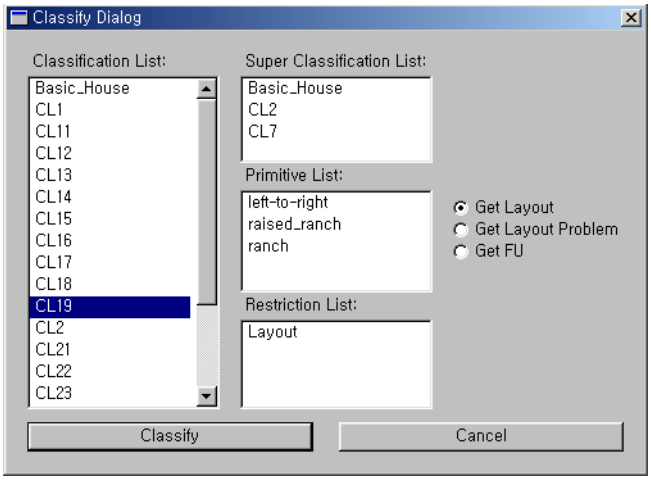


FIGURE 35. Classify settings box

24. Compare

The user creates a temporary description and compares it with existing classifications.

Preconditions:
1. A CKB is active.
2. At least one classification has been created in the CKB.

Basic course:
1. The user selects the “Compare” option from the “Retrieve” menu.
2. SL_CB opens the “Compare” settings box (see Figure 36).
3. The user inputs a temporary description name in the “Temp Description Name:” field and selects the super classifications from the “Super Classification List:” field, primitives from the “Primitive List:” field, and restrictions from the “Restriction List:” field; and selects the classification from the “Existing Classification List:” field.
4. If the user clicks the “Compare” button, the temporary description is compared with the selected classification.
5. The user can decide if the temporary description is saved or not. If the user decides the temporary description is saved, the user clicks the “SaveTemp” button to save the temp description.
6. The user may exit this use case anytime by hitting the “Cancel” button in the “Compare” settings box.

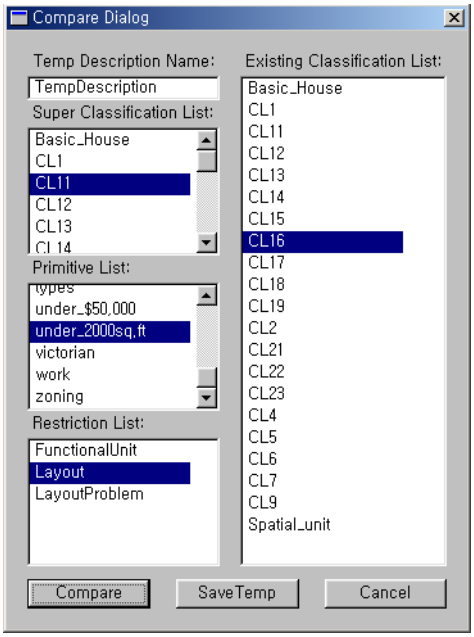


FIGURE 36. **Compare settings box**

25. Retrieve classified objects

Upon the user's request, SL_CB retrieves an object with a specific classification from the database. Recall that users may attach classifications to Functional Units, Layout Problems, and Layouts.

Preconditions:
1. SL_CB is connected to the object database.
2. A CKB is active and contains at least one classification that is attached to an object in the database.

Basic course:
1. The user selects the "Retrieve" option from the "Retrieve" menu.
2. SL_CB opens the "Retrieve" settings box (see Figure 37).
3. The user specifies interactively the retrieval type (SUBSUMER, SUBSUMEE, or EQUIVALENT) and selects one name.
4. The user selects a desired combination of classifications and the class of objects he wants to retrieve (FU, Layout or LayoutProblem), and clicks the "GetDBObjectName" button. During this process, the classification engine checks the disjoined primitive conflicts among selected classifications.
5. The dialog box displays the names of all objects of that class with the specified classification and shows them in a selection field.
6. The user selects one name and clicks the Retrieve button.
7. SL_CB retrieves the selected object and loads it into working memory. If the object is a Functional Unit, it will be added as a constituent to the current Layout Problem. If it is a Layout Problem, SL_CB deletes the current Layout Problem and all Layouts generated from it, and makes the retrieved Layout Problem the current one. If the object is Layout, it adds this Layout to the set of currently available Layout alternatives; the Design Units in the retrieved Layout become associated with Functional Units in the current Layout Problem that have the same name as the Functional Units with which the Design Units were associated when the Layout was saved in the database. If no such Functional Unit can be found, the Functional Unit originally associated with the Design Unit will also be retrieved and added as a constituent to the current Layout Problem.
8. The user may exit this use case anytime by hitting the "Cancel" button in the "Retrieve" settings box.

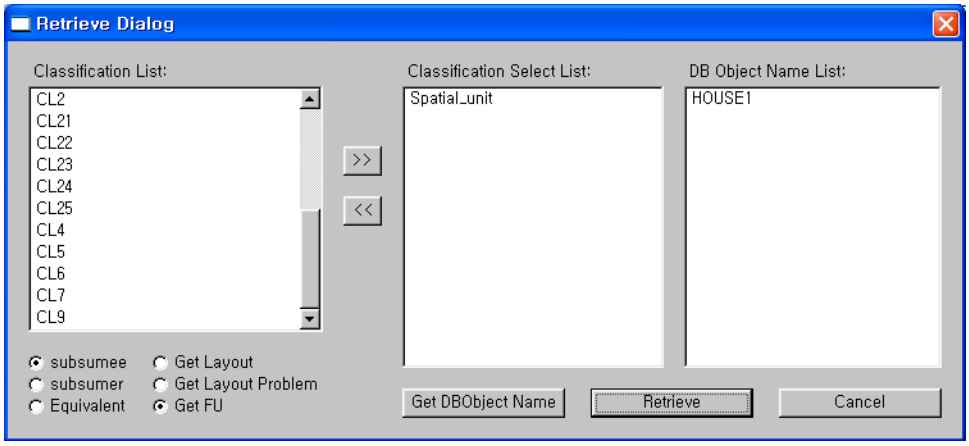


FIGURE 37. Retrieve settings box

6.3.2 Cases

I have argued in Chapter 3 and 5 that cases can be indexed based on their form or the components they contain. I also suggested that the classifications handled by the CKB engine can be used to index cases according to their form ("split-level", "ranch"), features which a CAD system often cannot discover on its own. However, certain component-based features can be automatically recognized. I suggested that the CB engine, especially the target construct, can be used to automatically add component-based features to a case that can be used as indices for retrieval *in addition to* the classifications.

The first prototype implementation is restricted to three basic component features that can be easily implemented with the CB engine: the number of bedrooms, bathrooms, and work rooms in a Layout. Whenever a user saves the active Layout as a case, SL_CB automatically determines how many rooms of these types it contains and adds these numbers as attributes to the case. In other words, a user does *not* have to capture these features explicitly through classifications. Upon retrieval, users are then able to specify not only a classification, but also the number of desired components for which they are looking. For example, they can specify that they are looking for a split-level house with three bedrooms, two bathrooms and one work room.

I have developed twelve use cases that capture the required functionality based on the schema specifications and case-base API provided by the CB engine

mentioned in section 5.3.2.4. In the following, I describe in greater detail the use cases and the GUI that makes them available to users.

6.3.2.1 Session Handling

26. Start a case-base design session

The user starts a case-base session in order to define or retrieve appropriate cases from the case base (CB).

Preconditions:
1. The SL_Comm Window (SCW) is open.

Basic Course:
1. The user selects the “Case base” option from the “Communication” menu in the SCW menu bar.
2. SL_CB opens the “Database Login” dialog box.
3. SL_CB prompts the user to select a database type and to input a database name, username and password.
4. If the three input values are all correct, SL_CB connects to the selected case base and opens the Case-based Design (CBD) Window; otherwise, it shows an appropriate error message and prompts the user again for the required inputs.
5. The user may abort this use case anytime by hitting the “Cancel” button in the “Database Login” dialog box.

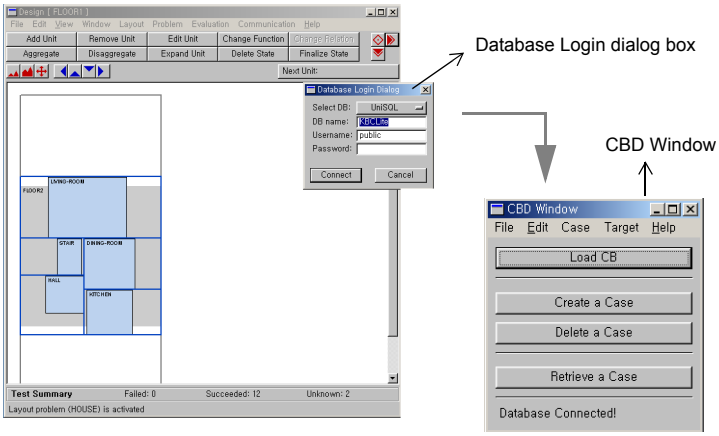


FIGURE 38. Start a case-based design session

27. End a case-based design session

The user ends a case-based design session and closes the CBD Window.

Preconditions:
1. The CBD Window is open.

Basic Course:
1. The user selects the “Close” option from the “File” menu.
2. If changes have been made after the last save, SL_CB asks the user if the changes should be saved and acts accordingly.
3. SL_CB closes the CBD Window and ends the case-based design session.

6.3.2.2 CB Administration

Like with CKBs, the information a designer or group of designers wants to collect to form a specific CB can be recorded and saved in a specific, named CB instance.

28. Create a CB

Upon the user’s request, SL_CB creates a new CB instance and gives it a name.

Preconditions:
1. The CBD Window is open.

Basic Course:
1. The user selects the “New CB” option from the “File” menu.
2. SL_CB opens the “New Case-base” dialog box.
3. The user inputs a CB name in the “New CB Name:” field.
4. The user hits the “Commit” button.
5. If a CB with the same name does not exist in the database, SL_CB creates a new CB in working memory; otherwise, it shows an error message and prompts the user to input another name. The new CB is stored persistently only when the user selects the “Save CB” option from the File menu (use case 32).
6. The user may exit this use case anytime by hitting the “Cancel” button in the “New Case-base” dialog box.

29. Load a CB

Upon the user's request, SL_CB loads a previously created CB.

Preconditions:
1. The CBD Window is open.

Basic Course:
1. The user selects the "Load CB" option from the "File" menu.
2. SL_CB opens the "Load Case-base" dialog box, which shows in the "Load Case Base" field the names of the CBs in the connected database.
3. The user selects a CB name and hits the "Commit" button.
4. SL_CB retrieves the CB from the connected database and makes it the currently active one.
5. The user may abort this use case anytime by hitting the "Cancel" button in the "Load Case-base" dialog box.

Alternate course:
The user issues the "Load CBD" command from the CBD command bar in step 1. All steps are the same from step 2 on.

30. Close a CB

Upon the user's request, SL_CB closes the active CB.

Preconditions:
1. A CB is active.

Basic Course:
1. The user selects the "Close CB" option from the "File" menu in the CBD Window.
2. If changes have been made after the last save, SL_CB asks the user if the changes should be saved and acts accordingly.
3. SL_CB closes the currently active CB.

31. Save a CB

Upon the user's request, SL_CB saves the active CB in the connected database.

Preconditions:
1. A CB is active.

Basic Course:
1. The user selects the “Save CB” option from the “File” menu.
2. SL_CB saves the currently active CB in the connected database.
3. SL_CB informs the user if no changes have been made after the last save.

32. Save a CB as

Upon the user’s request, SL_CB saves the active CB in the connected database under a different name.

Preconditions:
1. A CB is active.

Basic Course:
1. The user selects the “Save CB As” option from “File” menu.
2. SL_CB opens the “Save As Case-base” dialog box (see Figure 39).
3. The user enters a name in the “New CB Name:” field and hits the “Save” button.
4. If a CB with the same name is not in the connected, SL_CB saves the currently active CB under the new name in the connected database; otherwise, it shows an error message and prompts the user to enter another name.
5. The user may abort this use case anytime by hitting the “Cancel” button in the dialog box.

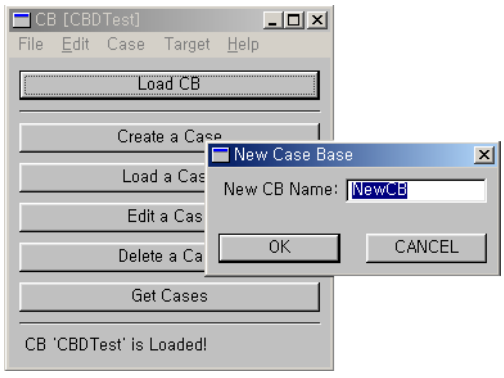


FIGURE 39. **Save as case-base dialog box**

33. Delete a CB

Upon the user's request, SL_CB deletes a CB in the database.

Preconditions:
1. The CBD Window is open.

Basic Course:
1. The user selects the "Delete CB" option from the "File" menu.
2. SL_CB opens the "Delete Case-base" dialog box, which shows a list of the CBs in the connected database.
3. The user selects a CB name and hits the "Delete" button.
4. If the selected CB is not the active one, SL_CB opens an Alert box asking the user to confirm the request; otherwise, it displays an error message and the use case returns to step 3.
5. If the user hit the "OK" button in the Alert box, SL_CB deletes the selected CB in the connected database along with all the associated CB components. This deletion is immediately persistent. If the user hit the "Cancel" button in the Alert box, SL_CB terminates the use case and closes all related dialog boxes.

34. Cleanup a CB

Upon the user's request, SL_CB cleans up the active CB.

Preconditions:
1. A CB is active.

Basic Course:
1. The user selects the "Cleanup CB" option from the "File" menu.
2. SL_CB opens an Alert box asking the user to confirm the request.
3. If the user hit the "OK" button in the Alert box, SL_CB cleans up the active CB. If the user hit the "Cancel" button in the Alert box, SL_CB terminates the use case and closes all related dialog boxes.

6.3.2.3 Case Handling

35. Create a Case

Upon the user's request, SL_CB turns the active Layout into a new case.

Preconditions:
1. A CB is active.
2. A Layout Problem (LP) is active, which implies that there exists an active Layout, and this LP has been read from the object database.
3. Users who wish that SL_CB add component information (number of bedrooms, bathrooms, and work rooms) as case attributes must make sure that the Functional Units representing these rooms in the active LP have been classified accordingly and have been allocated in the active Layout.

Basic Course:
1. The user selects the “Create a Case” option from the “Case” menu.
2. SL_CB opens the “New Case” settings box.
3. The user enters a name in the “Case Name:” field.
4. SL_CB displays the classifications attached to the current Layout Problem and component-based attributes of the currently active Layout derived from the classifications of the Functional Units allocated in it.
5. The user hits the “Save” button.
6. If no case with the new name exists in the active CB, SL_CB creates a new case; otherwise, it shows an error message and the use case returns to step 3. The form- and component-based classifications are attached to the Layout in the background, i.e. do not need any user action.
7. The user may exit this use case anytime by hitting the “Cancel” button in the “New Case” settings box.

Alternate course:
1. The user issues the “Create a Case” command from the CBD command bar in step 1. All steps are the same from step 2 on.

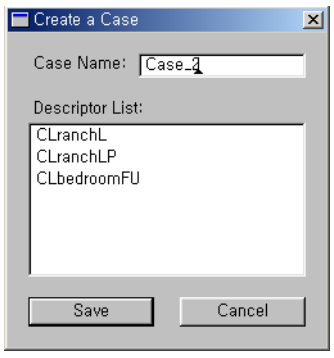


FIGURE 40. **Create a Case settings box**

36. Delete a Case

Upon the user's request, SL_CB deletes a selected case in the active CB.

Preconditions:
1. A CB is active.
2. At least one case has been created in the CB.

Basic Course:
1. The user selects the "Delete a Case" option from the "Case" menu.
2. SL_CB opens the "Delete Case" dialog box, with displays - in the "Case Name List" field - the names of all cases currently defined in the active CB.
3. The user selects a case name in the list and hits the "Delete" button.
4. SL_CB displays an Alert box asking the user if the case should be deleted.
5. If the user confirms, SL_CB deletes the case.
6. The user may exit this use case anytime by hitting the "Cancel" button in the "Delete a Case" dialog box.

Alternate course:
1. The user issues the "Delete a Case" command from the CBD command bar in step 1. All steps are the same from step 2 on.

37. Retrieve a Case

Upon the user's request, SL_CB retrieves a Layout as a case from the active CB. The retrieved case is added to the current design space and becomes the active Layout. Note that a case can be retrieved in two ways: (i) by name or (ii) by index, which is a combination of a classification and components.

Preconditions:
1. A CB is active.
2. A Layout Problem is active and has been read from the object database.

Basic Course:
1. The user selects the "Retrieve Cases" option from the "Case" menu.
2. SL_CB opens the "Retrieve Cases" settings box.
3. The user selects the retrieval method by checking the "By Name" or "By Index" check box.

4. If the user checked the “By Name” box, SL_CB displays the names of all cases in the active CB. If the user checked the “By Index” box, the use case continues with the alternate course.
5. The user selects a case name and hits the “Retrieve” button.
6. If the object representing the case exists in the object database, SL_CB retrieves the case; otherwise, it displays an error message, and the use case returns to step 3. The retrieved case is added to the current design space and becomes the active Layout. The DesignUnits in that Layout are associated with FunctionalUnits in the active LayoutProblem that have the same name as the FunctionalUnits with which these Design Units were associated when the case was created (use case 36); if such Functional Units do not exist, the FunctionalUnits associated with the original layout are added to the LayoutProblem.
7. The user may exit this use case anytime by hitting the “Cancel” button.

Alternate course (Retrieval by index):

4. If the user checked the “By Index” box, SL_CB computes automatically a retrieval index by combining form-based classification attached to the current Layout Problem with the number of Functional Units classified as bedrooms, bathrooms, or workrooms. SL displays this classifications to the user.
5. The user inspects this classification, and may change the number of desired components of some type. The user then hits the “Commit” button.
6. If cases with this combination of indices exist in the active CB, SL_CB displays the names of these cases as in step 4 of the basic course.
7. The use case continues with step 5 of the basic course.

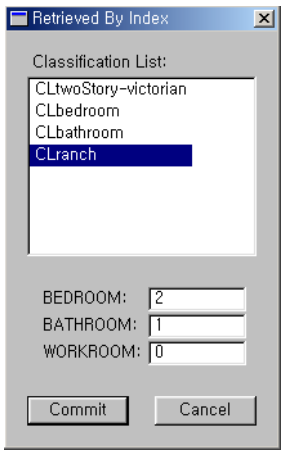
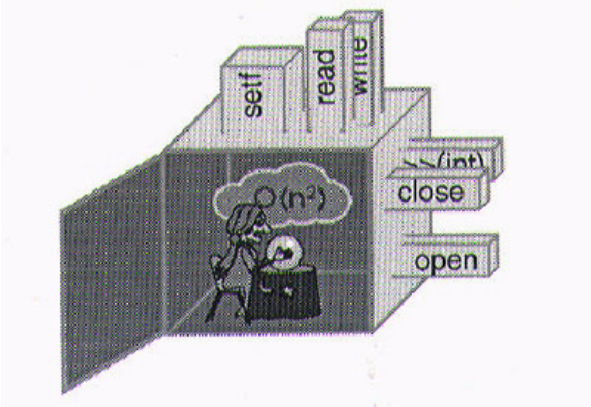


FIGURE 41. **Retrieved by index settings box**

CHAPTER 7

How it All Works -- Case Creation, Retrieval and Adaptation in Action



The present chapter illustrates how the prototype described in the preceding chapters' works by describing concrete examples of case creation, retrieval, and adaptation that make use of SL_CB. These examples tie the use cases back to the scenarios in Chapter 4 and demonstrate concretely how SL_CB supports selected tasks in housing design. The examples also illustrate how the capabilities SL_CB adds to SL_Comm interact with its other components, especially SEED-Layout.

7.1 Case Base: Initial Seeding

I mentioned in section 5.3.2.4 that a case base needs some initial cases as contents or needs to be 'seeded'. For the present context, the initial case base may contain standard housing configurations that are or have been widely used in the industry, from ranches, raised ranches, and split-levels to the currently popular 'MacMansions'.

Designers charged with the task of generating such a case must first use the SEED-Layout GUI to create a Layout Problem and its Functional Unit constituents, and then generate a Layout, possibly over several floors, that allocates the Functional Units. If the case to be created is a traditional split-level residence, the Layout Problem may be formulated as shown in Table 3. Constituent relations are indicated by indentation in the left-most column. Note

how MassingElements have to be used in SEED-Layout to assure proper exterior alignments across floors within a building block and to allow floors in different blocks to have different floor heights (off-set by 1/2 floor—the basic idea underlying the split-level scheme). Figure 42 shows a snapshot of the SEED-Layout GUI that supports the generation of Functional Unit constituents: a constituent hierarchy window showing the evolving spatial hierarchy and the dialog box a designer can use to define various numerical constraints.

TABLE 3. The Functional Units of a split-level residence

FU name	FU class	Dimensional Requirements	Required Adjacency
SplitLevel	BuildingFU	min.width: 24ft., min.area: 1152sq.ft.	
Part1	MassElementFU	min.width: 24ft., min.area: 576sq.ft.*	
Floor0	StoreyFU	min.width: 24ft., min.area: 576sq.ft.	
		elevation: 0 ft.	
Garage	ZoneFU	min.width: 20ft., min.area: 480sq.ft.	
Stair	VertZoneFU	min.width: 6ft.	Garage
Utilities	RoomFU	min.width: 5ft.	Stair
Floor2	StoreyFU	min.width: 24ft., min.area: 576sq.ft.	
		elevation: 9 ft.	
Stair	VertZoneFU	min.width: 6ft.	
Hall	ZoneFU	min.width: 3ft.	Stair
MBedroom	RoomFU	min.width: 12ft., min.area: 176sq.ft.	Hall
Bedroom1	RoomFU	min.width: 10ft., min.area: 125sq.ft.	Hall
Bedroom2	RoomFU	min.width: 10ft., min.area: 125sq.ft.	Hall
Bathroom1	RoomFU	min.width: 5ft., min.area: 25sq.ft.	Hall
Bathroom2	RoomFU	min.width: 5ft., min.area: 25sq.ft.	Hall
Part2	MassElementFU	min.width: 24ft., min.area: 576sq.ft.	
Floor1	StoreyFU	min.width: 24ft., min.area: 576sq.ft.	
		elevation: 4.5 ft.	
Entry	RoomFU	min.width: 3ft.	
Living Room	RoomFU	min.width: 15ft., min.area: 250sq.ft.	Entry
Dining Room	RoomFU	min.width: 12ft., min.area: 176sq.ft	Living Room
Kitchen	RoomFU	min.width: 10ft., min.area: 125sq.ft.	Dining Room

* The area requirements for MassingElements apply to the *footprint* of the element, *not* the sum of floor areas contained in it

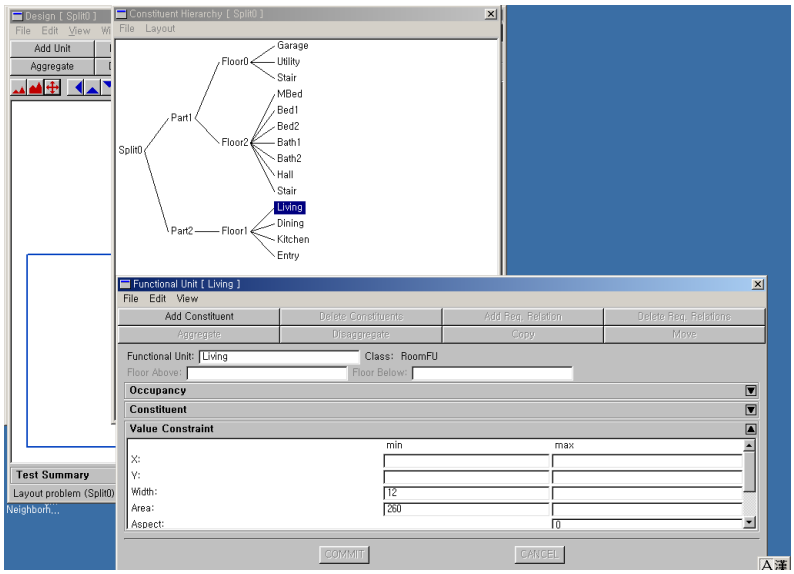


FIGURE 42. **SEED-Layout GUI supporting creation of Functional Units**

Based on this information, designers are now able to create, in a short time, the three floors of the house as shown in Figure 43. If designers have a sketch of the entire configuration (in their head or on paper), they can tell SEED-Layout exactly where to place each unit. SEED-Layout is able to use the requirements in the Layout Problem to size the spaces on each floor automatically and to check if all required relations are satisfied. This process may take only minutes if the designers are experienced SEED-Layout users.

Formulating the Layout Problem may take a little longer because quite a bit of information has to be entered. Nevertheless, the entire example discussed here was completed by an experienced SEED-Layout user in less than 30 minutes.

Before the Layout can be saved as a case, the Layout Problem has to be classified, for example, with the classification '*CLsidewiseSlope-splitLevel*'. In order to create this classification, the designer has to use first use case 11 (Create a primitive) to create the two primitives *sidewise_slope* and *split_level,* respectively; the first specializes the primitive *site_shape* and the second specializes *exterior_style* in the taxonomy (see Figure 26).

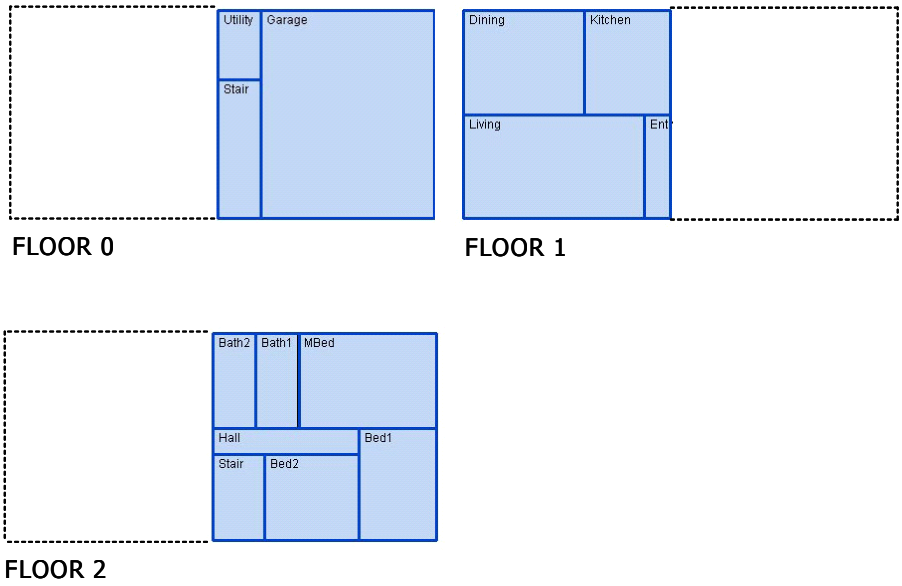


FIGURE 43. Three floors of a split-level residence created with SEED-Layout

Once the primitives have been defined, the designer can create the classification that combines both with use case 15 (Create a classification). Since the classification *CLsidewiseSlope-splitLevel* is a specific kind of split-level, the designer should make it a specialization of the more general classification *CLsplitLevel*. As a result, the classification *CLsidewiseSlope-splitLevel* inherits all derived information from the classification *CLsplitLevel*.

The designer is now able to attach this classification to the Layout Problem with use case 23 (Attach a classification). In order to do this, it is not necessary that the Layout Problem has been saved persistently in the object database. When a classifiable object (Layout Problem, Functional Unit or Layout) is being classified, SL_CB checks if the object is already persistently stored and saves it persistently if this is not so. In the end, the classification is persistently attached to the object (via an object proxy).

In order to assure that component-based classifications can be generated accurately, the designer must also ensure that all Functional Units representing bedrooms or bathrooms are classified correctly with classification *CLbedroom* or

CLbathroom (again with use case 23). This enables SL_CB to derive the number of rooms of each type independently of their names and to save these as an additional case attribute.

At this point, the generated Layout is ready to be saved as a case in the initial case base. This happens with use case 35 (Create a Case). SL_CB computes automatically the index to be used by combining the classification attached to the Layout Problem with the number of bedrooms and bathrooms contained in the Layout (Figure 41 shows the settings box supporting this use case).

The primitives and classifications attached to the Layout Problems and Functional Units are saved in a specific, named CKB, and the case with the added component information is saved in a specific, named CB. Both of these are databases in their own right and independent of the object database storing the Layout Problem and Layout themselves. The CKB and CB must be made available, together with the object database, to any user or organization interested in using this information because otherwise, the case base cannot be accessed. In combination, the three databases allow designers and other users to browse the case base. Once they have developed an idea about its content, they are able to start building their own CKBs and CBs over the same objects in the object database, if they desire to do this.

7.2 The Retrieve-Adapt-Create Cycle

What emerges clearly from Figure 15 is that the CBD steps supporting parts of the scenarios form retrieve-adapt-create patterns that are cyclical because the adapted case that is saved as a new case may, in turn, become the first step in a next cycle (see Figure 44). This section illustrates how SL_CB supports this cycle through three episodes.

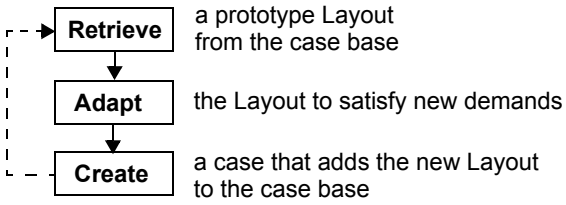


FIGURE 44. Retrieve-adapt-create cycle in the design scenarios

7.2.1 Episode 1

Suppose a potential client discusses with a designer or sales person possible configurations for a three-bedroom/two-bathroom residence (building block xx in scenario 3). Suppose furthermore that the designer has access to SL_CB and the initial case base. As a first step, the designer may retrieve a Layout Problem using a classification as search index that lists three bedrooms and two bathrooms as attributes. Suppose that one of the Layout Problems retrieved in this way is the Layout Problem generated for the split-level case. As the designer and client inspect the Functional Units in the problem, they may find out that all the desired units are there and consequently retrieve the Layout shown in Figure 43.

In order to retrieve the case, the designer opens the *Retrieved By Index* dialog box (use case 37; see Figure 41), which shows a search index combining the classification attached to the Layout Problem and the number of bedrooms and bathrooms in the problem. If the designer commits, the Layout shown in Figure 43 can be retrieved from the case base (together with other cases that have the same classification).

Let's further assume that when the client discusses this option with the designer, they discover that it would be easy to extend Floor 1 to the front so that it can contain an additional home office that is accessible by the client directly from the living room and by outside visitors from the outside. The other part of the residence, with its independent foundation, is unaffected by this modification.

In order to generate this variant, the designer has to add a home office as a RoomFU with the appropriate requirements to the current Layout Problem on Floor 1 and interactively direct SEED-Layout to allocate it on Floor 1 in a desired location. SEED-Layout will size the room automatically based on the requirements specified in the Functional Unit. The corresponding plan is shown in Figure 45. Note that SEED-Layout would also be able to find automatically feasible room locations, but the interactive method is much faster for an experienced SEED-Layout user who knows exactly where the space should be.

The designer finds this adaptation interesting enough and decides to add it to the case base; this happens in the same way in which the initial case was created (see section 7.1). If the added home office was classified as *CLworkroom*, this case can now be retrieved when the index calls for a residence (or split-level residence) with 3 bedrooms, 2 bathrooms and one work room.

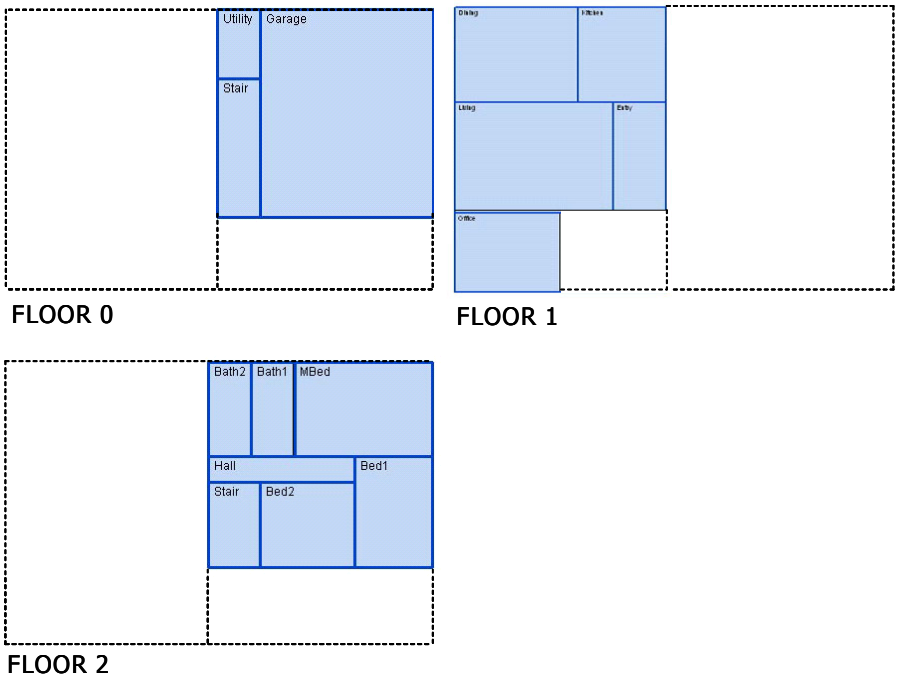


FIGURE 45. **Modified split-level residence**

7.2.2 Episode 2

Suppose now that at a later time, the same or a different designer retrieves this case for a new client interested in an integrated home office from the start. However, this client wants more than a single office; he/she expects frequent visitors and needs an independent entrance and reception area. The designer is again able to modify the Layout Problem and Layout to include the new Functional Unit (see Figure 46). The process is analogous to the one described in episode 1.

The designer saves this new adaptation again as a case. If he has started to built his own CKB, he may create a specific classification that adds the primitive *extended_workarea* to the classification of the Layout Problem. In order to do

this, he either uses use case 15 to create the new classification from scratch or modify the existing classification (use case 18, Edit a classification).

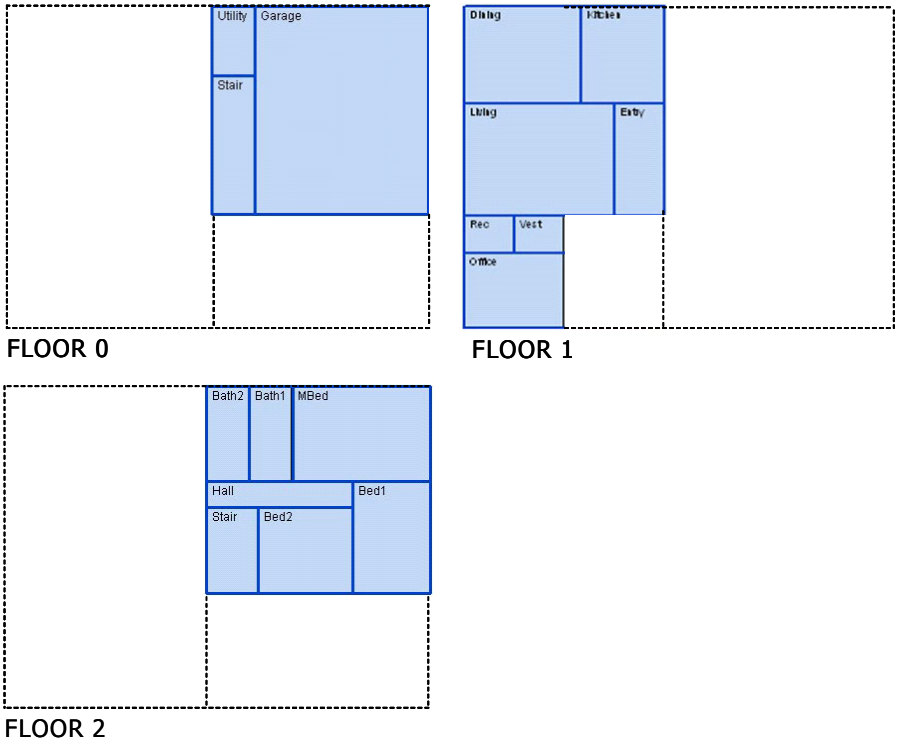


FIGURE 46. Second modification of split-level residence

7.2.3 Episode 3

As a later time, this case may be retrieved for a client interested in an extended home office, but the site does not easily accommodate the pronounced L-shape of this solution. In discussing this, the designer suggests that a more compact shape could be achieved if a floor is added to Part 2 of the residence. This requires a significantly different Layout Problem with a constituent hierarchy as shown in Table 4. A Layout allocating the Functional Units in the Problem is shown in Figure 47.

TABLE 4. **Functional Units in extended split-level residence**

FU name	FU class
SplitLevel	BuilldingFU
Part1	MassElementFU
Floor0	StoreyFU
Garage	ZoneFU
Stair	VertZoneFU
Utilities	RoomFU
Floor3	StoreyFU
Hall	ZoneFU
MBedroom	RoomFU
Bedroom1	RoomFU
Bedroom2	RoomFU
Bathroom1	RoomFU
Bathroom2	RoomFU
Part2	MassElementFU
Floor2	StoreyFU
Stair	VertZoneFU
Living Room	RoomFU
Dining Room	RoomFU
Kitchen	RoomFU
Floor1	StoreyFU
Stair	VertZoneFU
Entry/reception	RoomFU
Home office	RoomFU
1/2 Bathroom	RoomFU

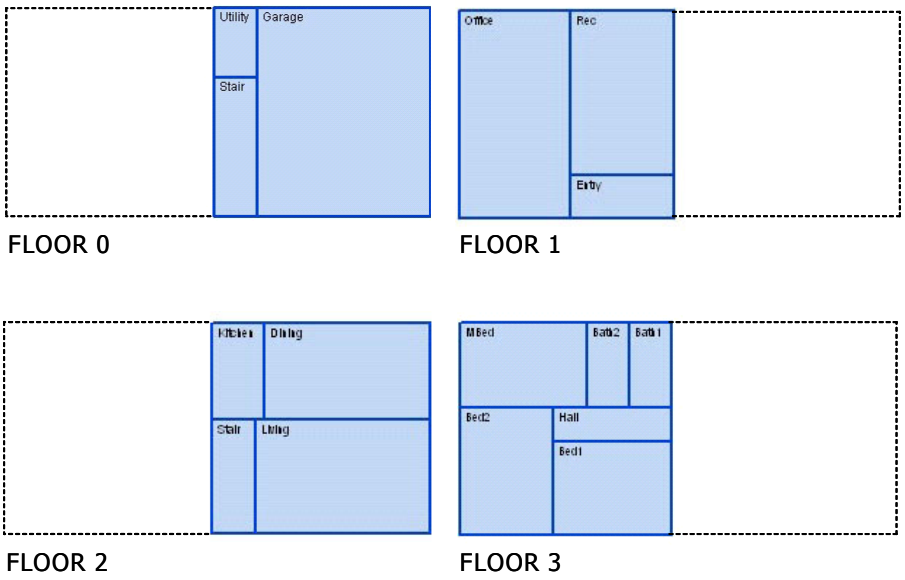


FIGURE 47. **Third modification of split-level residence**

These episodes illustrate how SL_CB supports the retrieve-adapt-create cycle identified in Figure 15. At the same time, they illustrate how the case base can continue to expand as these cycles multiply. They specifically illustrate how adaptations, even novel solutions, once they have been created as cases, become immediately accessible to anyone who has access to SL_CB and the connected databases.

CHAPTER 8

Conclusion

Stat rosa pristina nomine, nomina nuda tenemus.
- Umberto Eco, *The name of the rose*

This chapter presents the contributions of this research. It then makes suggestions for several enhancements and indicates promising directions for future research.

8.1 Contributions

Case-based design offers an efficient way of finding complex design solutions by minimal search, provided that problems presented to the system have strong similarities to known cases for which solutions exist. The housing market is a prime candidate for this approach because both public and for-profit developments rely heavily on standardized plan configurations or established precedents. An additional benefit for this special market arises from the fact that the case base of a CBD system may serve as general mechanism to distribute innovative design solution and thus counteract the problems inherent in that particular industry. This thesis identifies classificatory types of housing precedents to serve as indexing features for the retrieval in a CBD system. This typology is embedded in a general framework that integrates building blocks of the housing design scenarios and the main phases of CBR in design. The building blocks of the housing design scenarios have been formally modeled based on interviews with various housing design experts. The suitability of the framework for its intended purpose has been demonstrated by a prototype of CBD systems

for housing design based on the SEED databases and SEED-Layout environment.

The following contributions result from this work:

- **Description of the housing design process with different parties in the housing market**: Based on a literature survey about the housing industry in the US and interviews with housing experts, this research has developed design scenarios to model the housing design process, which is inherently an ill-structured domain. Each design scenario comprises several building blocks in a formalized structure. This thesis makes only limited use of this formalization, but it allows other researchers to investigate housing design in greater detail.

- **Definition of a general framework for applying CBR to housing design**: The general framework integrates the housing design process and CBR in design. It associates appropriate building blocks of the housing design scenarios with typical CBD phases. The integrated approach provides the context, defines the overall functionality, and raises the issues that have to be addressed in the design and implementation of a CBD system for housing design.

- **Identification of types for the classification of housing precedents**: This research identifies classificatory types of housing precedents that are most useful for a CBD system that retrieves precedents based on form- and component-based classifications. These types serve as an indexing scheme for the retrieval of housing precedents in a CBD system.

- **Functional specification of a CBD system supporting this process**: This research develops a functional specification for a prototype CBD system. The use case approach is used here to specify the capabilities of a CBD prototype for housing. Thirty-nine use cases have been developed so far and used in the design of the prototype implementation. However, they could also guide developments in a different context and environment from the one underlying the present prototype.

- **User interface design**: This research designs and implements an appropriate graphical user interface to deliver the use cases to users. It integrates (and hide from the user) heterogeneous modules like CKB, object database, case base, a database and a communication server, SEED-Layout, and so on, and provides effective user interactions to manage complex information.

8.2 Future Research Directions

In this section, I provide a brief outline of possible enhancements and future research directions to extend the work in this dissertation.

- **Improvement of the graphical user interface**: I have implemented the graphical user interface (GUI) for CKB and CBD. Testing the GUI with real end-users is needed to see if layout/graphics/terms are intuitive and easy to learn. Especially, some of the terms are based on the CLASSIC knowledge representation system and not necessarily familiar to designers or developers.

- **Extension of component-based classifications to define more complex queries**: Component-based queries in the first prototype implemented in this dissertation are restricted to simple checks that ascertain if certain room types exist in a layout independently of where they are in the layout or how they are related to each other or to other spaces. The implementation is further restricted (hard-coded) to consider only three basic room types: bedrooms, bathrooms, and work rooms. This was easy to implement based on the given API. Extended and more complex component-based queries that take an open-ended set of room types, location and spatial relations into account are conceivable, but introduce a significant increase in algorithmic complexity that deserves a careful investigation in its own right.

- **Classification-based matching of rooms during retrieval**: As explained in Section 5.4.3, a Layout in SEED-Layout consists of a collection of rectangular areas called Design Units that are associated with the Functional Units in a Layout Problem. These Functional Units collect the requirements the Design Unit must satisfy and indicate, to the user, the basic function of that Design Unit. The most basic of these is that the Functional Units are used to give the Design Units a name (like ‘living room’). When a Layout is stored persistently, the association between Design Units and Functional Units is maintained. When the same Layout is retrieved as solution to a current Layout Problem, the retrieved Design Units must be associated with Functional Units in the current problem. Right now, this is done by name; that is, whenever SEED-Layout retrieves a Design Unit, it retrieves at the same time the name of the Functional Unit originally associated with it and checks if the current Layout Problem contains a Functional Unit with the same name. If this is the case, it associates that Functional Unit with the retrieved Design Unit. If no such Functional Unit can be found, SEED-Layout retrieves the original Functional Unit and adds it to the current Layout Problem. It is up to the user to inspect this Functional Unit and decide what to do with it. This scheme works properly only if some strict

naming conventions are maintained across Layout Problems, a restriction that appears far too severe for a general-purpose CBD system intended to be used by various designers independently of each other. In this situation, it would be much more convenient to assigns Functional Units to Design Units by classification; for example, Design Units associated with a Functional classified as “bedroom” will be associated with a Functional Unit of the same classification independently of how these Functional Units are named in the original and current Layout Problems. However, this approach is again algorithmically and computationally decidedly non-trivial because of the ambiguities that arise when several units have the same classification; in fact, under subsumption, most units may end up with the same classification: there may be choices for each Functional Unit involved. This creates a hard combinational optimization problem that cannot be addressed in this thesis, but seems to represent an interesting and practically relevant topic for a second dissertation.

- **Acquiring learning capability with accumulated housing precedents**: I introduced in the Chapter 1 that the CBR-cycle includes four main activities: retrieve, reuse, revise, and retain. An important phase in this cycle is the retain phase in which the system may be modified as a result of its usage and success in the past; new cases may be collected within the case base and/or the similarity concept in use may be tuned [Kamp et al. 1998]. CBR, Information Retrieval (IR), database management system, and machine learning are coming closer to provide the intelligent retrieval techniques over complex, structured data. Recently, technologies developed allow for much larger case-base. In order to extract patterns from these large databases, “data mining” algorithms are useful to analyze the data efficiently. In the first prototype implementation, housing solutions are generated, modified, and stored in the CBD system. However, I do not have enough amount of cases to apply data mining algorithms up to now. Acquiring knowledge from databases using data mining techniques will be an interesting topic for a later research.

References

[Aamodt 1995] Aamodt, A. (1995). Knowledge acquisition and learning by experience - the role of case-specific knowledge. In *Machine Learning and Knowledge Acquisition - Integrated Approaches*, eds. G. Tecuci and Y. Kodratoff, 197-245. Academic Press.

[Aamodt and Plaza 1994] Aamodt, A. and E. Plaza (1994). Case-based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. In *AI Communications* 7(1):39-59. IOS Press.

[Aygen 1998] Aygen, Z. (1998). A Hybrid Model for Case Indexing and Retrieval in SEED. Ph.D. Dissertation. School of Architecture, Carnegie Mellon University, Pittsburgh, PA.

[Bergmann and Althoff 1998] Bergmann, R. and K-D. Althoff (1998). Methodology for Buildiing CBR Applications. In *Case-Based Reasoning Technology*, eds. M. Lenz, B. Bartsch-Spörl, H. Burkhard, and S. Wess, 299-326. Heidelberg, Berlin: Springer-Verlag.

[Borgida 1992] Borgida A., R. Brachman, D. McGuiness, and L. Resnick (1992). CLASSIC: A Structural Data Model for Objects, Technical report. AT&T Bell Laboratories. Murray Hill, NJ.

[Börner 1998] Börner, K. (1998). CBR for Design. In *Case-Based Reasoning Technology*, eds. M. Lenz, B. Bartsch-Spörl, H. Burkhard, and S. Wess, 201-233. Heidelberg, Berlin: Springer-Verlag.

[Bourne 1981] Bourne, L. (1981). *The Geography of Housing.* New York: John Wiley & Sons.

[Broadbent 1973] Broadbent, G. (1973). *Design in Architecture: Architecture and the Human Sciences.* NewYork: John Wiley & Sons.

[Bruegge and Dutoit 2000] Bruegge, B. and A. Dutoit (2000). *Object-oriented software engineering: conquering complex and changing systems*. New Jersey: Prentice-Hall.

[Carbonell 1983] Carbonell, J. (1983). Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In *Machine Learning: An Artificial Intelligence Approach*, eds. R.S. Michalski, J.G. Carbonell, and T. M. Mitchell, 137-161. Palo Alto: Tioga Publishing Co.

[Chien 1998] Chien, S. (1998). Supporting Information Navigation in Generative Design Systems. Ph.D. Dissertation. School of Architecture, Carnegie Mellon University, Pittsburgh, PA.

[Chun and Yoon 1989] 전경배 , 윤충렬 (1989). *주택 계획론* . 중판 , 서울 : 산업도서출판공사

[Coyne et al. 1993] Coyne, R., U. Flemming, P. Piela, and R. Woodbury (1993). Behavior Modeling in Design System Development. In *CAAD Futures '93*, eds. U. Flemming and S. V. Wyk, 335-354. Amsterdam, The Netherlands: Elsevier Science Publishers B.V.

[Cumming 1999] Cumming, M. (1999). SEED: Collaborative Design Scenarios, US Army Construction Engineering Research Laboratory, P.O.Box 9005, Champaign IL. *Contact Order No. DAC88-96-D-0004*. March 8, 1999.

[Flemming and Aygen 2001] Flemming, U. and Z. Aygen (2001). A hybrid representation of architectural precedents. In *Automation in Construction* 10: 687-699.

[Flemming et al. 2000] Flemming, U. and SEED-Team (2000). Appendix C: SEED-Layout Use Cases. In *The SEED Experience*, Internal Report. School of Architecture and Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA 15213.

[Flemming 1999] Flemming, U. (1999). SEED-Layout Tutorial. Internal Report. School of Architecture and Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA 15213.

[Flemming and Chien 1998] Flemming, U. and S. Chien (1998). SEED-Layout Reference Manual. Internal Report. School of Architecture and Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA 15213.

[Flemming et al. 1997] Flemming, U., Z. Aygen, R. Coyne, and J. Snyder (1997). Case-based Design in a Software Environment that supports the Early phases in building Design. In *Issues and Applications of Case-Based Reasoning in Design*, eds. M. Maher and P. Pu, 61-85. Lawrence Erlbaum Associates, Publishers.

[Flemming and Chien 1995] Flemming, U. and S. Chien (1995). Schematic Layout Design in SEED Environment. In *Journal of Architectural Engineering* Dec.1995:162-169.

[Flemming and Woodbury 1995] Flemming, U. and R. Woodbury (1995). Software Environment to support Early building Design (SEED): Overview. In *Journal of Architectural Engineering* Dec.1995:147-152.

[Gamma et al.1995] Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns*. Addison Wesley Longman, Inc.

[Gero 1990] Gero, J. (1990). Design Prototypes: A Knowledge Representation Schema for Design. In *AI magazine* 11(4):26-36.

[Goldberg 1989] Goldberg, B. (1989). Diffusion of Innovation in the Housing Industry. *prepared by NAHB National Research Center.*

[Gonzalez and Dankel 1993] Gonzalez, A. and D. Dankel (1993), *The Engineering of Knowledge-Based Systems: Theory and practice*. Prentice-Hall, Inc.

[Gutman 1985] Gutman, R. (1985). *The Design of American Housing: A Reappraisal of the Architect's Role*. the Publishing Center for Cultural Resources.

[Jackendoff 1994] Jackendoff, R. (1994). *Consciousness and the computational mind*, 2nd ed. Cambridge, Massachusetts: The MIT Press.

[Jackson 1995] Jackson, M. (1995). *Software Requirements and Specifications: a lexicon of practice, principles and prejudices*, Addison-Wesley Publishing Company.

[Jacobson et al. 1994] Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard (1994). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, Inc.

[Jones et al. 1998] Jones, T., W. Pettus, and M. Pyatok (1998). *Good Neighbors: Affordable Family Housing*, 2nd ed. North America: McGraw-Hill, a division of The McGraw-Hill Companies, Inc.

[Kamp et al. 1998] Kamp, G., S. lange and C. Globig (1998). Related Areas. In *Case-Based Reasoning Technology*, eds. M. Lenz, B. Bartsch-Spörl, H. Burkhard, and S. Wess, 327-351. Springer-Verlag Berlin Heidelberg.

[Kicklighter and Kicklighter 1998] Kicklighter C. and J. Kicklighter (1998). *Residential Housing & Interiors*. Tinley Park, Illinois: The Goodheart-Willcox Company, Inc.

[Kim 1990] Kim, W. (1990), *Introduction to Object-Oriented Databases*. The MIT Press.

[Kolodner 1993] Kolodner, J. (1993). *Case-Based Reasoning*. San Mateo: Morgan Kaufmann Publishers.

[Kolodner 1991] Kolodner, J. (1991). "Improving human decision making through case-based decision aiding" in *AI magazine* 12(2):52-68.

[Krier 1988] Krier, R. (1988). *Architectural Composition*. New York: Rizzoli.

[Lang 1987] Lang, J. (1987). *Creating Architectural Theory: The role of the Behavioral Sciences in Environmental Design*. New York: Van Nostrand Reinhold.

[Lawrence 1987] Lawrence, R. (1987). *Housing, Dwellings and Homes - Design theory, research, and practice*. New York: John Wiley & Sons.

[Lee et al. 1995] Lee, H., J. Lee, and S. Chang (1995). Design Adaptation for Handling Design Failures. *Proceedings of the CAAD Futures '95*, eds. Milton Tan and Robert Teh, 567-576. Centre for Advanced Studies in Architecture, National University of Singapore.

[Leupen 1997] Leupen, B. (1997). *Design and analysis*. New York: Van Nostrand Reinhold.

[Lewis 1994] Lewis, E. (1994). *Housing Decisions*. Tinley Park, Illinois: The Goodheart-Willcox Company, Inc.

[Lewis et al. 1995] Lewis, T., L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Andert, J. Vlissides, and K. Schmucker (1995). *Object-Oriented Application Frameworks*. Greenwich CT: Manning Publications Co.

[Luce and Suppes 1965] Luce, R. and P. Suppes (1965). Preference, Utility and Subjective Probability. In *Handbook of Mathematical Psychology,* eds. R. Luce, R. Bush and E. Galanter, vol. III. New York: Wiley and Sons.

[Maclennan 1982] Maclennan, D. (1982), *Housing Economics: an applied approach*. London; NewYork: Longman.

[Maher and Pu 1997] Maher, M. and P. Pu (1997). *Issues and Applications of Case-Based Reasoning in Design*. Lawrence Erlbaum Associates, Publishers.

[Maher et al. 1995] Maher, M., M. Balachandran, and D. Zhang (1995). *Case-Based Reasoning in Design.* Lawrence Erlbaum Associates, Publishers.

[McFadden 1978] McFadden, D. (1978). Modelling the Choice of Residential Location. In *Spatial Interaction Theory and Planning Models,* eds. A. Karlqvist, L. lundqvist, F. Snickars, and J. Weibull, 75-96. Amsterdam: North-Holland.

[McFadden 1973] McFadden, D. (1973). Conditional Logit Analysis of Qualitative Choice Behavior. In *Frontiers in Econometrics*, ed. P. Zarembka, 105-142. New York: Academic Press.

[Meyer 1988] Meyer, B. (1988). *Object-oriented software construction.* New York: Prentice-Hall.

[Nissen et al. 1994] Nissen L., R. Faulkner, and S. Faulkner (1994). *Inside Today's Home,* 6th ed. Orlando, Florida: Harcourt Brace College Publishers.

[Prerau 1990] Prerau, D. (1990), *Developing and managing expert system: proven techniques for business and industry,* Addison-Wesley Publishing Company, Inc.

[Resnick et al. 1993] Resnick, L., A. Borgina, R. Brachman, D. McGuinness, P. Patel-Schneider, and K. Zalondek (1993). “CLASSIC Description and Reference Manual For the COMMON LISP Implementation - Version 2.1”.

[Rich and Knight 1991] Rich, E. and K. Knight (1991), *Artificial Intelligence*, 2nd edition, McGraw-Hill, Inc.

[Rivard 1997] Rivard, H. (1997). A Building Design Representation for Conceptual Design and Case-Based Reasoning. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA.

[Rosenberg 1999] Rosenberg, D. (1999), *Use Case Driven Object Modeling with UML - A Practical Approach*, Reading, Massachusetts: Addison Wesley Longman, Inc.

[Rouda 1999] Rouda, M. (1999). Houses as products. In *Architectural Record* ‘99(1): 115-117, 176-177.

[Schneider 1997] Schneider, F. (1997). *Floor Plan Atlas: Housing*, ed. F. Schneider, second, revised, and expanded ed. Basel, Switzerland: Birkhauser.

[Snyder et al. 1994] Snyder, J., U. Flemming, R. Coyne, R. Woodbury, S-C. Chiou, B. Choi, H. Kiliccote, T-W. Chang, and S-F. Chien (1994). SEED Database Requirements. SEED Website, URL: *http://seed.edrc.cmu.edu/SD/dbreq/seedreq.html*.

[Sherwood 1994] Sherwood, R. (1994). *Modern Housing Prototypes*. sixth printing, Cambridge, Massachusetts: Harvard University Press.

[Slade 1991] Slade, S. (1991). Case-Based Reasoning: A Research Paradigm. In *AI magazine* 12(1):42-54.

[Snyder 1998] Snyder, J. (1998). Conceptual Modeling and Application Integration in CAD. Ph.D. Dissertation. School of Architecture, Carnegie Mellon University, Pittsburgh, PA.

[The AIA Research Corporation 1978] The AIA Research Corporation (1978). Regional Guidelines for Building Passive Energy Conserving Homes, *U.S. Department of Housing and Urban Development, US Government Printing Office*, Washington DC.

[Tumin 1967] Tumin, M. (1967). *Social stratification; the forms and functions of inequality.* Englewood Cliffs, NJ: Prentice-Hall.

[Weinand and Gamma 1994] Weinand, A. and E. Gamma (1994). ET++ - a Potable, Homogeneous Class Library and Application Framework, *Proceedings of UBILAB Conference '94*. Konstanz, Switzerland.

[Wentling 1995] Wentling, J. (1995). *Housing by Lifestyle: The component Methods of Residential Design*. 2nd ed. McGraw-Hill, Inc.

Appendix A

Representation and Building Blocks for Design Scenarios

A.1 Design Scenario Representation

A.1.1 Layout of Design Scenarios

- A description of the flow of events written in standard prose. This part will be shown in standard body text.
- A translation of this prose into a more formalized representation using the design process ‘building blocks’. These parts are formatted to indicate keywords and variable names, e.g.

```
Sequential task Prepare FU Hierarchy in SP =>
    Ordered task list
        • Task A
        • Task B
```

A.1.2 Text Formatting Conventions

- Keywords: **`Iterate`**
- Variable name: `Preliminary Design`
- Element defined as: =>

A.1.3 Terminal and Non-terminal Processes

Tasks may be either non-terminal or terminal processes. Non-terminal are those which are further decomposed into lower level processes (shown with ‘>>’ prefix),

while terminals (shown with '[LEAF]' prefix) are at their final level of decomposition and description in the context of these use cases. e.g.

```
Sub-tasks
 >>• Do preliminary design [= non-terminal]
   • [LEAF] Open layout in SL [= terminal]
```

A.2 Building Blocks for Design Scenarios

Each building blocks shows both states (shown as circles or ovals) and activities, or tasks (shown as rectangles).

A.2.1 Performing a Task

Performing a design-related task with the constraint that the required resources to perform the task are assembled prior to performing the task.

A.2.1.1 Flow of events

1. Assemble all resources required [including people]
2. Do task[s]

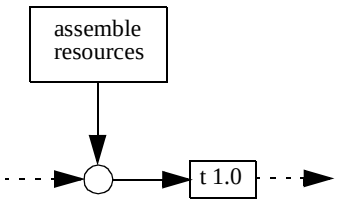


FIGURE 1. 'Performing a task' building block

A.2.1.2 Comments

A.2.1.3 Object required

- Input task[s] [including resource assembly]
- Assembly node
- Output task[s]

A.2.1.4 Textual description example

Do task Massing Design =>

Assemble resources

- Architect
- Client
- SEED Config application

A.2.2 Sequential Task

A task which is decomposed into a lower level set of tasks which are performed one after the other.

A.2.2.1 Flow of events

1. Do each task in the order specified

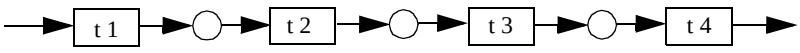


FIGURE 2. **'Sequential task' building block**

A.2.2.2 Comments

A.2.2.3 Object required

- List of tasks

A.2.2.4 Textual description example

Sequential task Standard Building Design Process =>

Ordered task list

- Preliminary Design
- Detailed Design
- Contract Documents
- Bidding and Award
- Construction Review

A.2.3 Branching In

A task which requires that a set of tasks be completed prior to starting it.

A.2.3.1 Flow of events

1. If all input tasks are completed
2. Activate branch-in node
3. Do output task[s]

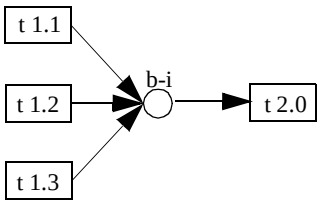


FIGURE 3. 'Branching in' building block

A.2.3.2 Comments

A.2.3.3 Object required

- Input tasks
- Assembly node
- Output task[s]

A.2.3.4 Textual description example

Branch-in Start Detailed Design =>

Input tasks

- Finish structural design
- Finish massing scheme
- Complete site plan

Output task

- Start detailed design

A.2.4 Branching Out

A state which enable the start of a set of tasks running in parallel.

A.2.4.1 Flow of events

1. If branch-out node activated
2. Do all output tasks

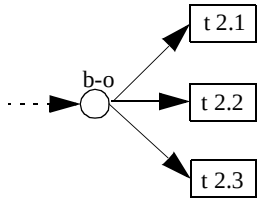


FIGURE 4. **'Branching out' building block**

A.2.4.2 Comments

A.2.4.3 Object required

- Input task[s]
- Assembly node
- Output tasks

A.2.4.4 Textual description example

Branch-out End of Preliminary Design =>

Input task

- Finish preliminary design

Output tasks

- Start detailed structural design
- Start preliminary mechanical design
- Start demolition design

A.2.5 Conditional Tasks

A set of tasks whose execution are dependent on the outcome of a single test.

A.2.5.1 Flow of events

1. Complete all input tasks
2. Activate decision node
3. Do appropriate task depending on outcome

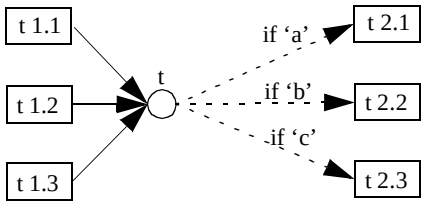


FIGURE 5. ‘Conditional tasks’ building block

A.2.5.2 Comments

A.2.5.3 Object required

- Input task[s]
- Test node
- Conditional task[s]

A.2.5.4 Textual description example

Conditional task Find Current Cost =>

Test

- ‘Cost of current scheme’

Input tasks

- Finish detailed architectural design
- Finish detailed structural design
- Finish preliminary mechanical design

Conditional outputs

- **If** ‘over budget’
  - **Do Task** Reduce scope of project
- **If** ‘under budget’
  - **Do Task** Check for errors
- **If** ‘on budget’
  - **Do Task** Continue to next phase

A.2.6 Iterating Processes

A closed loop structure which iterates according to the results of a single test.

A.2.6.1 Flow of events

1. Activate decision node
2. If ‘true’ do task[s] [may be multiple/parallel tasks]
3. Else end iteration [and proceed to next task]

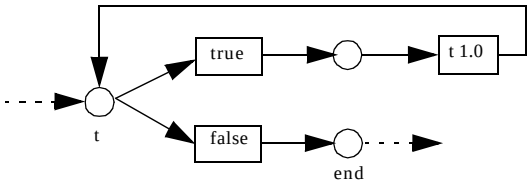


FIGURE 6. ‘Iterating processes’ building block

A.2.6.2 Comments

A.2.6.3 Object required

- Input task[s]
- Iteration test node
- Iteration task[s]
- Post-iteration task[s]

A.2.6.4 Textual description example

Iterate Reduce Project Cost =>

Test

- **If** 'Current scheme is over budget'
  - Eliminate un-requested functions
  - Eliminate extravagant finishes

Repeat test

- **Else**
  - Continue...

A.2.7 Composing Processes Using Recursion

The composition of layered processes into a containment hierarchy

A.2.7.1 Flow of events

[Note: this seems to perform the same function as a 'Sequential task'. One of these constructs will probably be eliminated...Perhaps 'sub-tasks' will not imply a sequential ordering].

1. If task is simple, do task
2. Else (task is composed of lower level tasks) recurs one level down

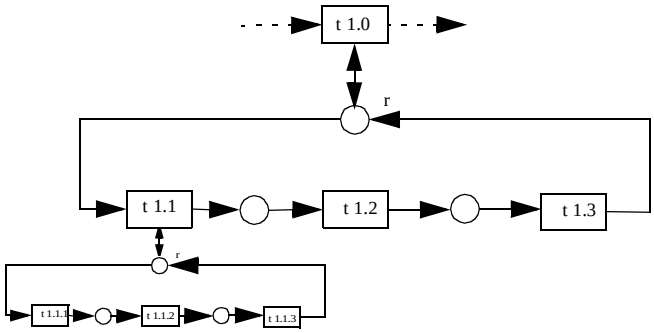


FIGURE 7. **'Composing processes using recursion' building block**

A.2.7.2 Comments

A.2.7.3 Object required

- Simple tasks [non-decomposable ones]
- Complex tasks [tasks composed of other tasks]
- Assembly nodes

A.2.7.4 Textual description example

Composed task Design Building =>

Sub-tasks

- Design massing
- Design urban context
- Define functional relationships

A.2.8 Holding Meetings

A set of tasks which structure the holding of meetings according to the items described in a meeting agenda. As a simplification, meeting participants are assumed to perform only those tasks which are listed on the agenda. The design conflicts and the consensuses reached which are assumed to resolve these conflicts, are also listed.

A.2.8.1 Flow of events

1. Gather agenda items

2. Organize meeting participants

2.1 Choose meeting participant

2.2 Assemble participants

2.3 Hold meeting

2.4 Disband meeting

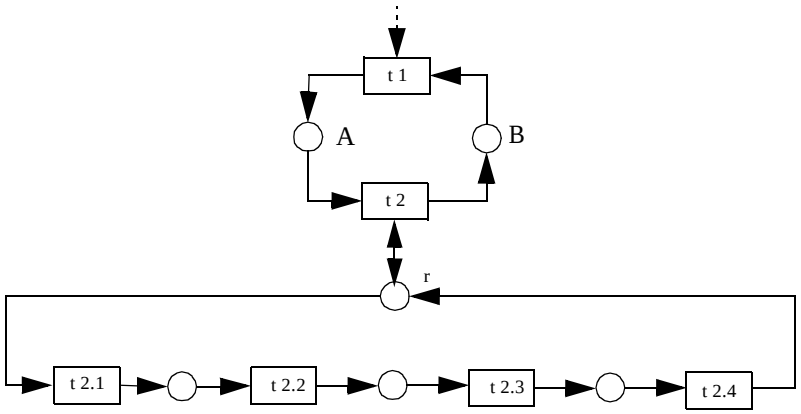


FIGURE 8. **'Holding meetings' building block**

A.2.8.2 Comments

- **A** State: Meeting required [sufficient number/urgency of agenda items]
- **B** State: Waiting for new meeting

A.2.8.3 Object required

- Tasks / sub-tasks
- State nodes
- Assembly nodes

A.2.8.4 Textual description example

Hold meeting Preliminary Massing Design Meeting =>

Meeting participants

- Engineer
- Client
- Architect

Agenda tasks

- Decide on massing scheme
- Meet with client

Design conflicts anticipated

- Client prefers metal cladding, while Architect prefers brick

Consensus anticipated

- Metal cladding to be used, but of a higher than normal quality

A.2.9 Building Consensuses

A set of tasks which structure the building of consensuses according to a list of design conflicts, and the participants required to resolve these conflicts.

A.2.9.1 Flow of events

1. Gather design conflict items
2. Attempt to reach a consensus
   - **2.1** Choose required participants
   - **2.2** Assemble participants
   - **2.3** Hold conflict resolution session
   - **2.4** Disband conflict resolution session

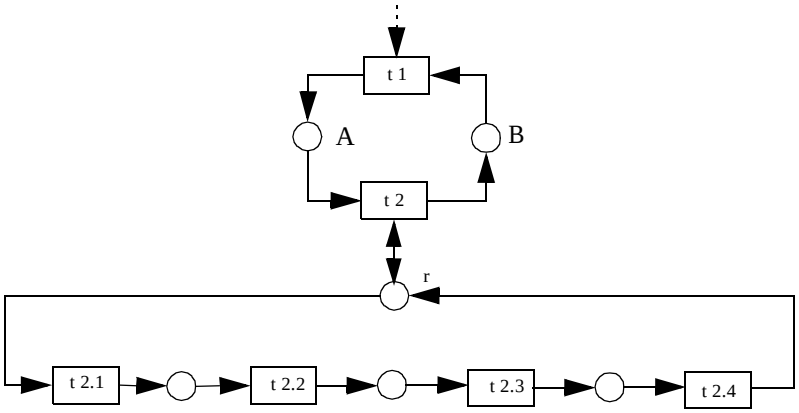


FIGURE 9. 'Building consensuses' building block

A.2.9.2 Comments

- **A** State: Consensus required [sufficient number of design conflict items]
- **B** State: Waiting for new design conflicts

A.2.9.3 Object required

- Tasks / sub-tasks
- State nodes
- Assembly nodes

A.2.9.4 Textual description example

Building consensus Conflict 1.5.3 =>

Required participants

- Structural engineer
- Mechanical engineer
- Architect
- [Client is not required]

Design conflicts anticipated

- building is too expensive
- Structural design ignores mechanical design

Arguments anticipated

- Budget is upwardly flexible (Architect)
- Structural drawings are complete, Mech. drawing have not been started (Str. E.)
- Mechanical design is innovative (Mech. E.)

Resolutions anticipated

- Mechanical engineer agrees to move plumbing pipes to standard location
- Structural engineer agrees to modify the structure slightly

A.2.10 List of Design Participants

A way of identifying a set of participants in a design process, without having to name each one individually. The intention here is that this list can be dynamically generated rather than specified at ‘compile-time’.

A.2.10.1 Flow of events

- N/A

A.2.10.2 Comments

- A simple dynamically generated linked list with no implied order.

A.2.10.3 Object required

- List of participants
- Participants

A.2.10.4 Textual description example

Design participants Preliminary design team 1a =>

- Client
- Architect
- Structural engineer
- Real estate agent
- Advertising designer

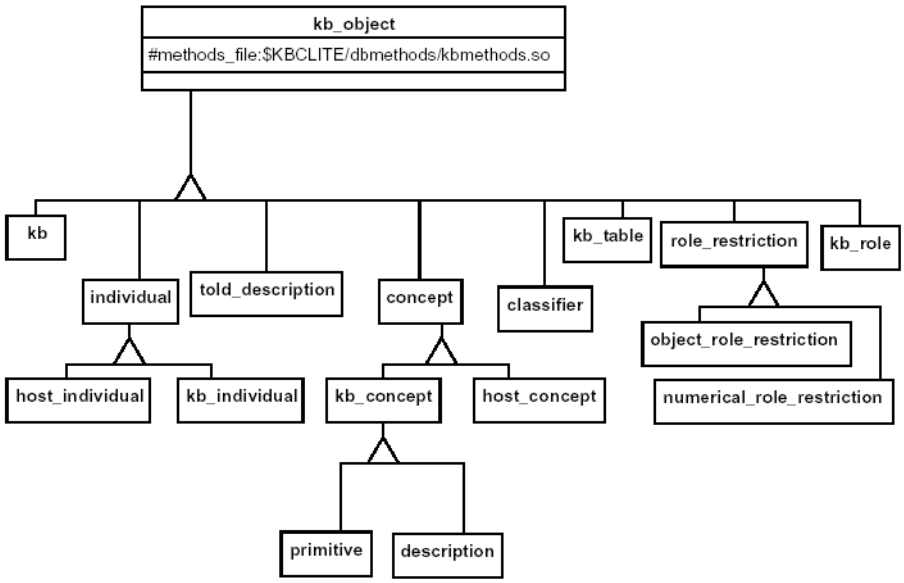
Appendix B

System Object Models

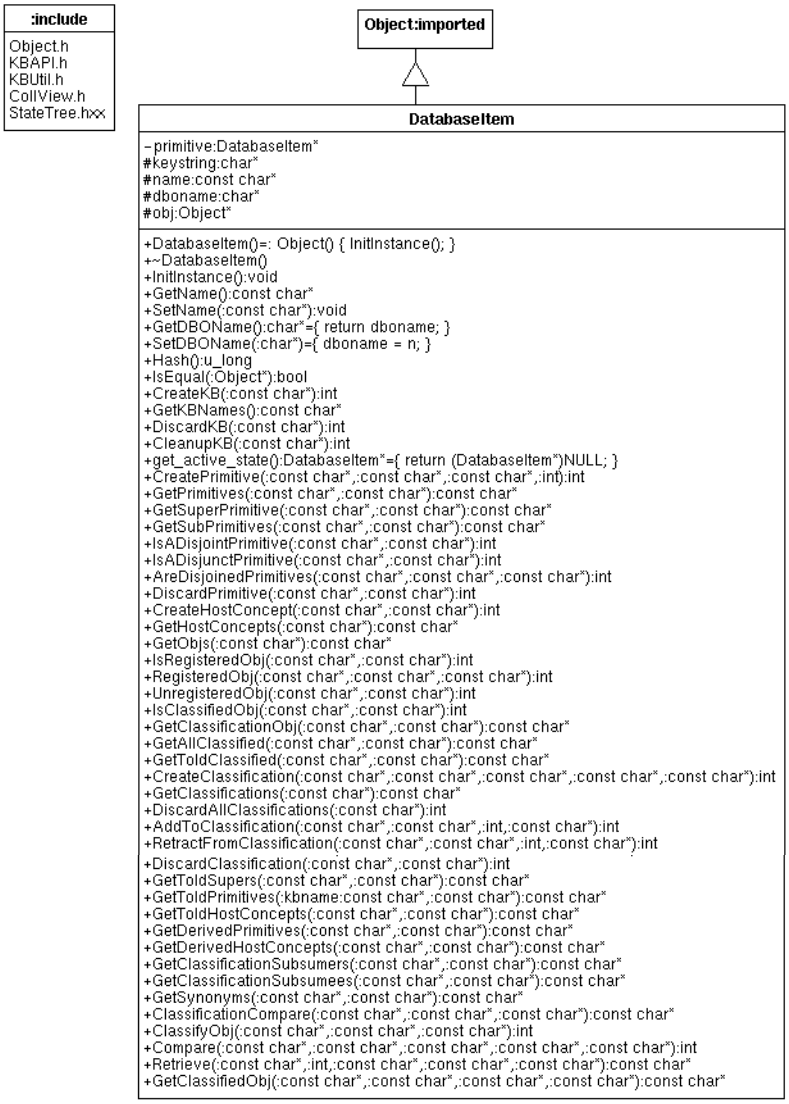
B.1 Domain Object Models

B.1.1 Classification Knowledge Base (CKB) Domain Object Models

B.1.1.1 CKB Schema

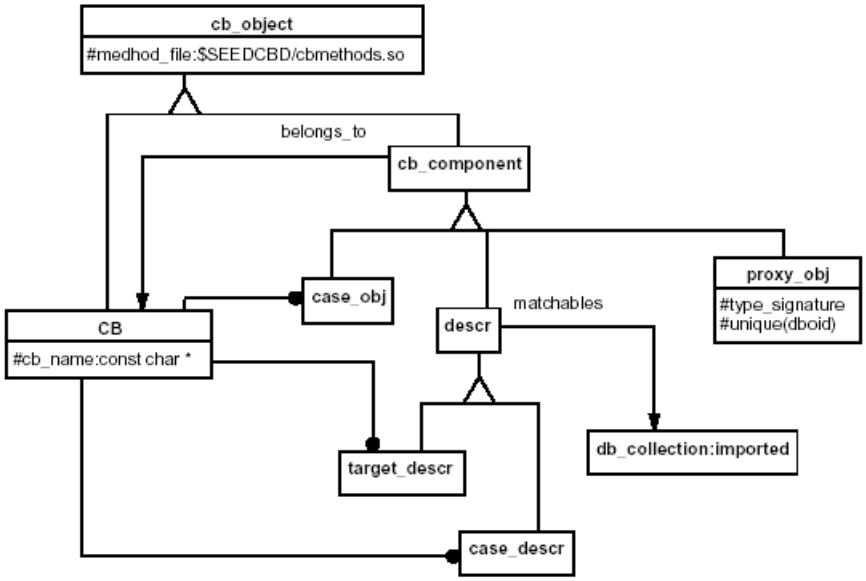


B.1.1.2 CKB Manager for API Implementation

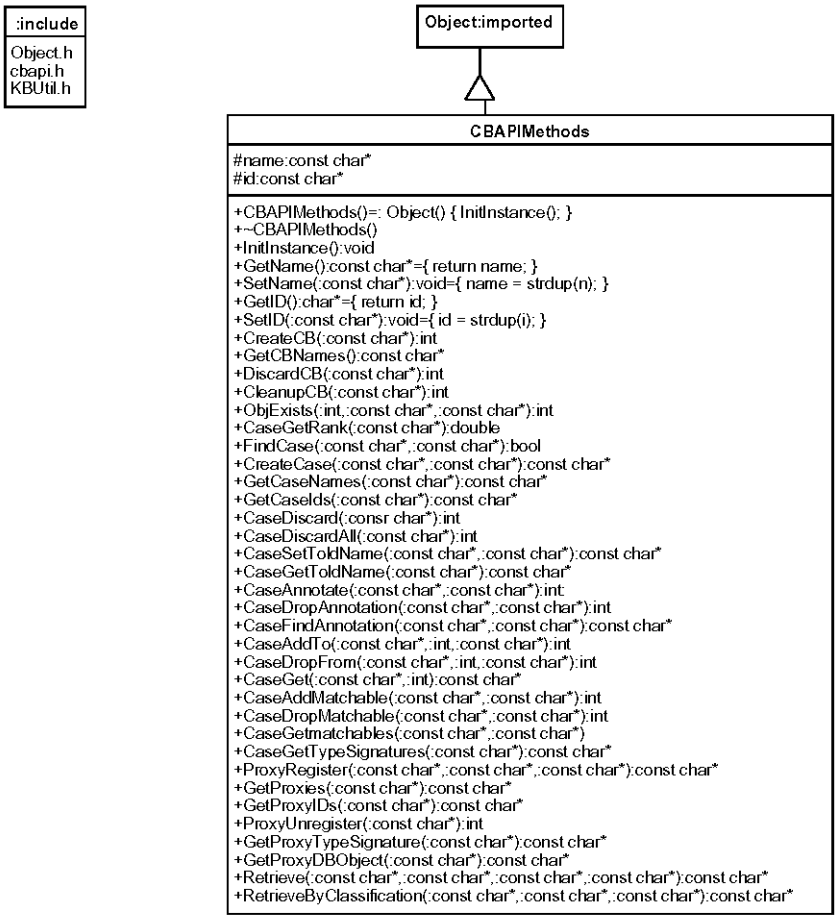


B.1.2 Case-Based Design (CBD) Domain Object Models

B.1.2.1 CBD Schema

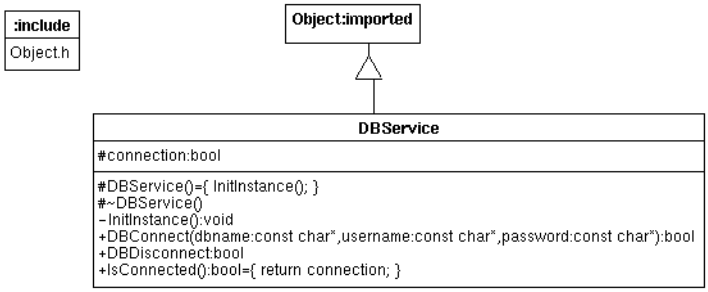


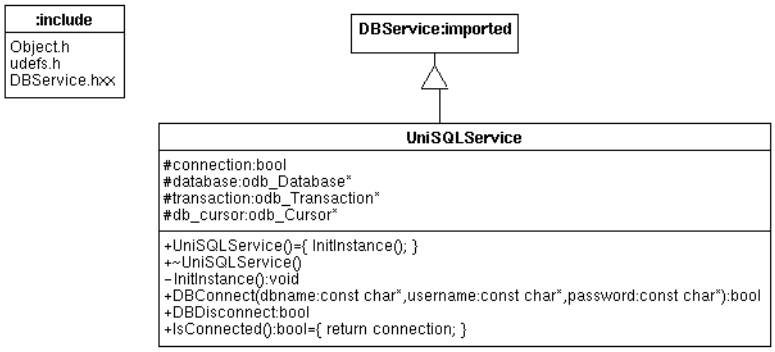
B.1.2.2 CBD Manager for API implementation

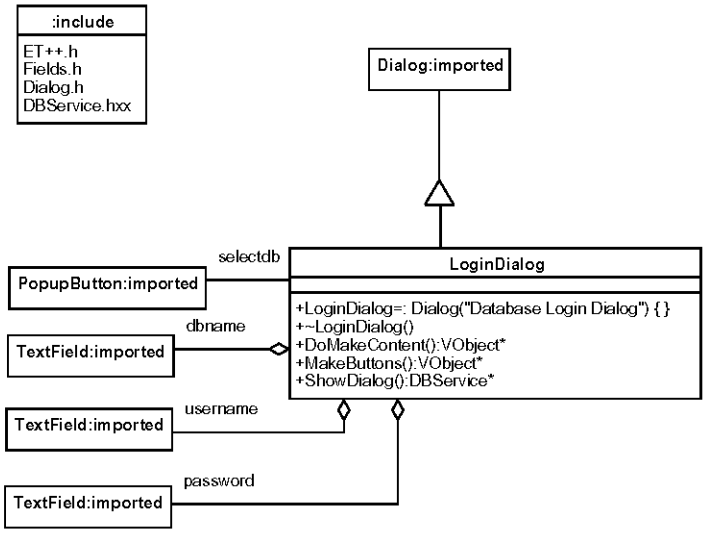


B.2 Interface Object Models

B.2.1 Database Connections

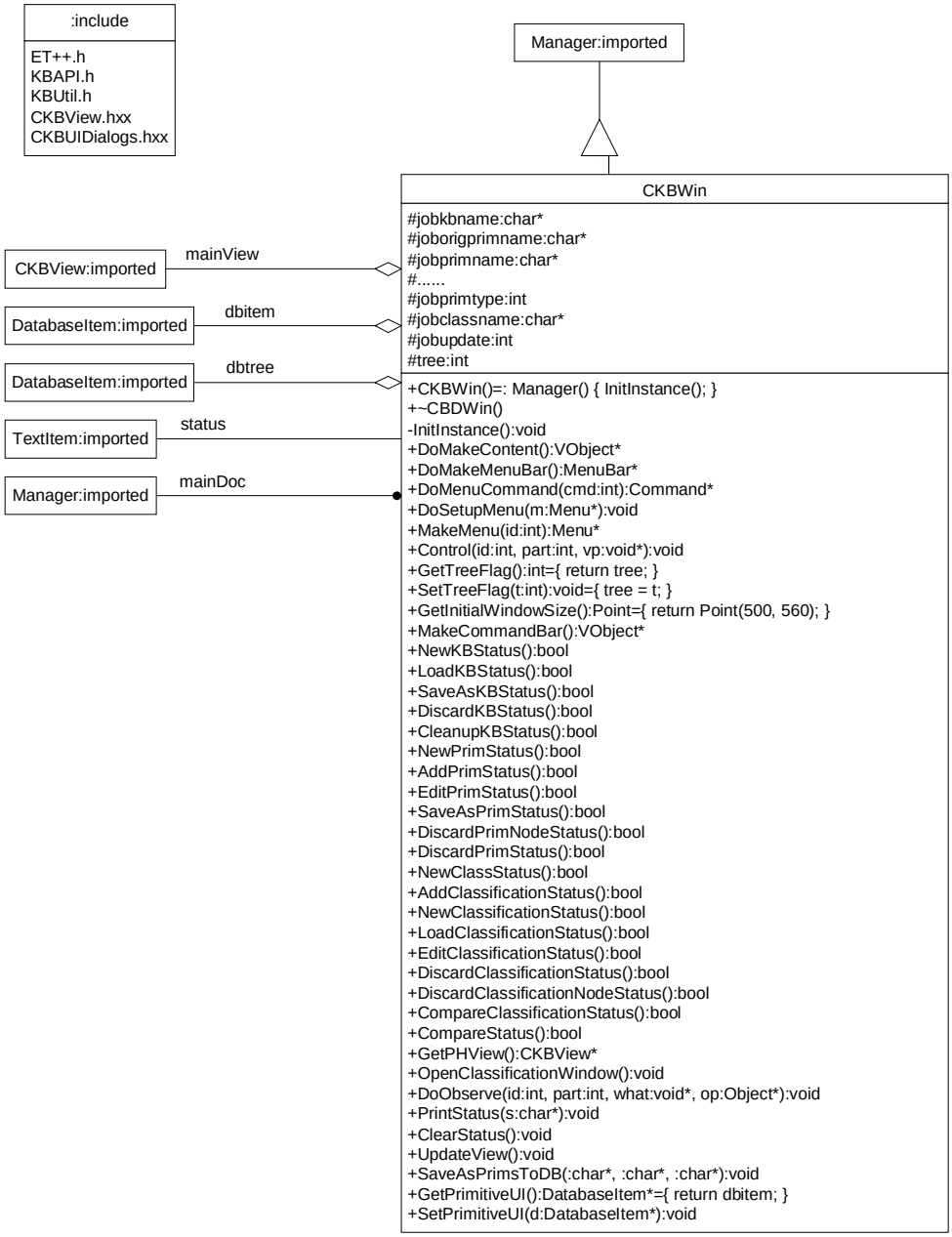




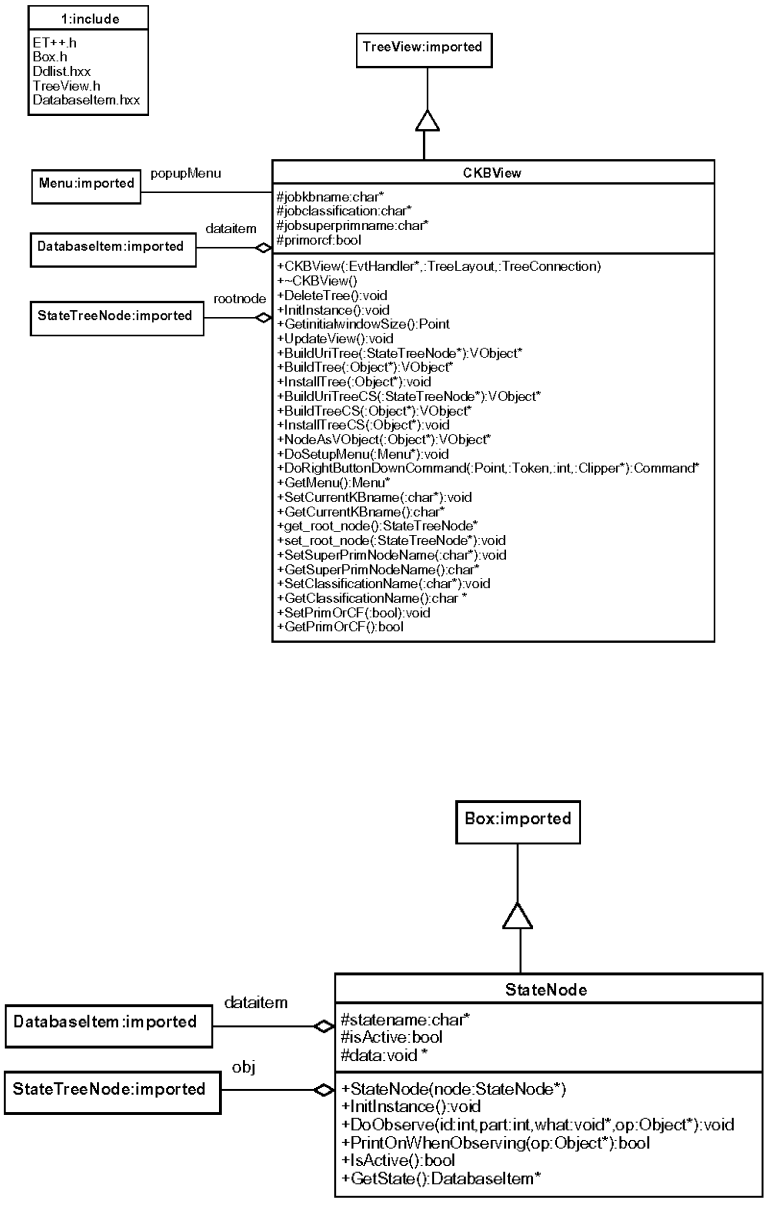

:include
ET++.h
Fields.h
Dialog.h
DBService.hxx
Dialog:imported
selectdb
PopupButton:imported
LoginDialog
+LoginDialog=: Dialog("Database Login Dialog") { }
+~LoginDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+ShowDialog():DBService*
dbname
TextField:imported
username
TextField:imported
password
TextField:imported

B.2.2 Main GUI Windows and Dialogs

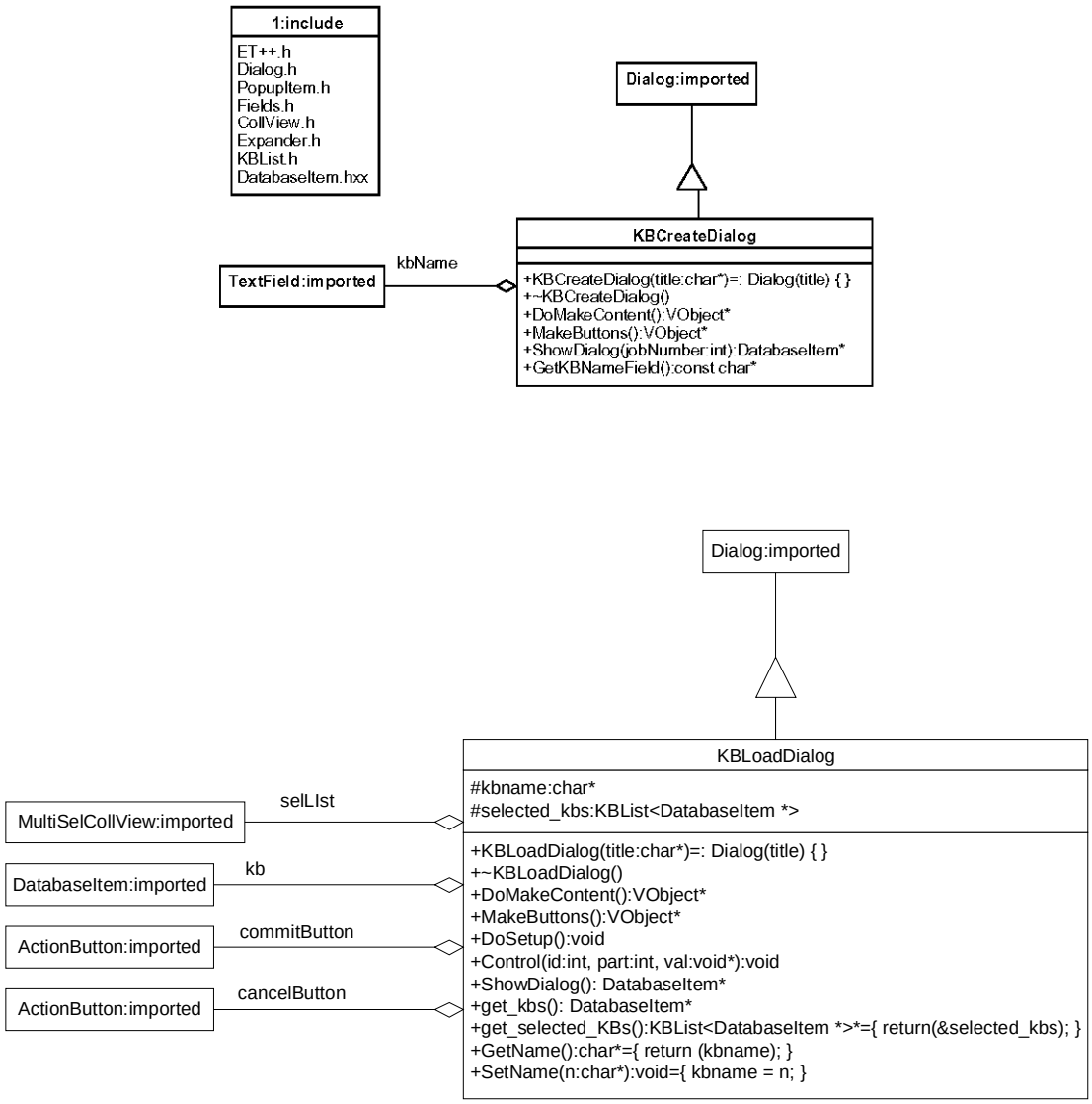
B.2.2.1 The Main CKB Window

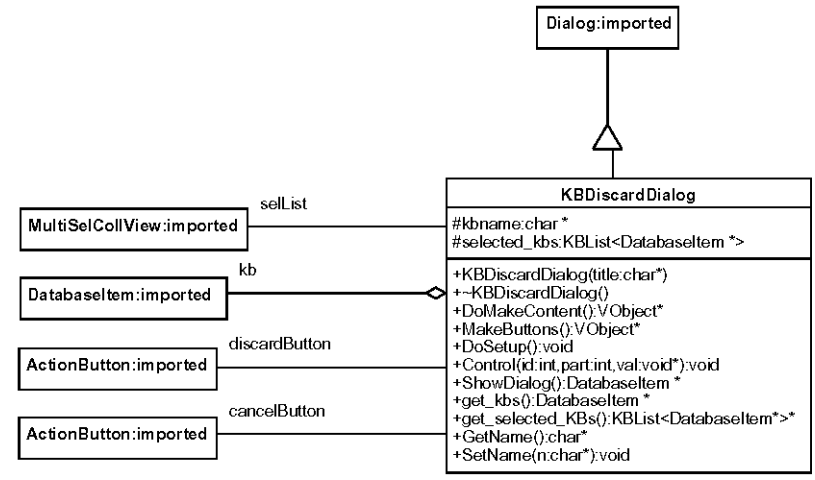


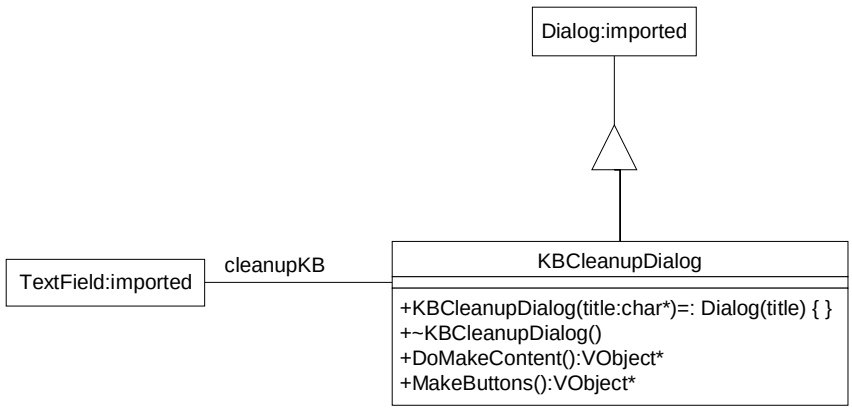
B.2.2.2 The Tree View (Primitive and Classification Hierarchy)

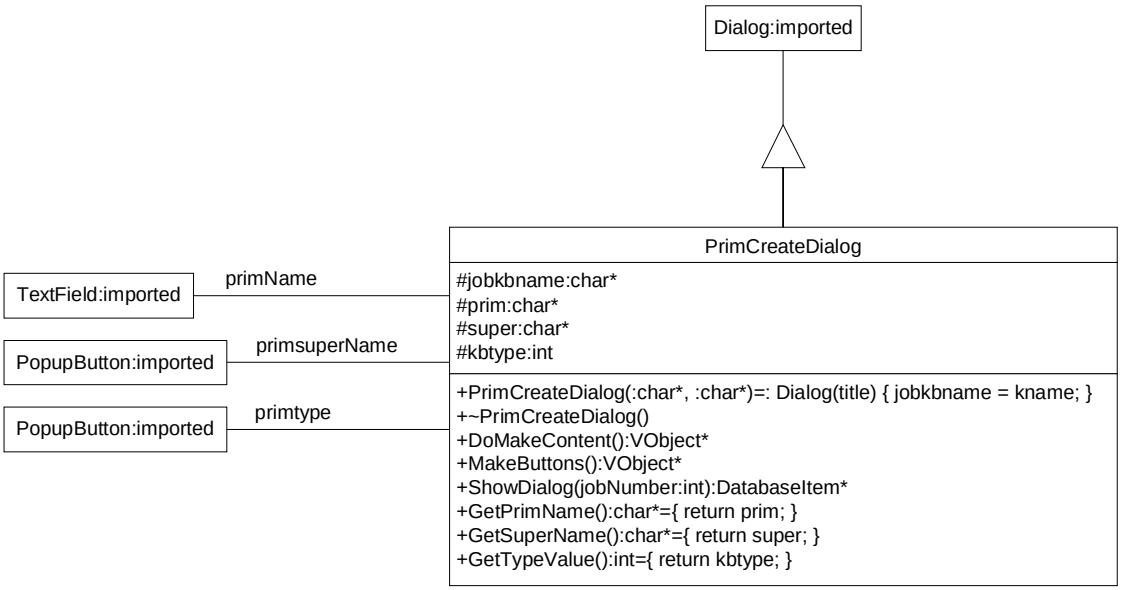


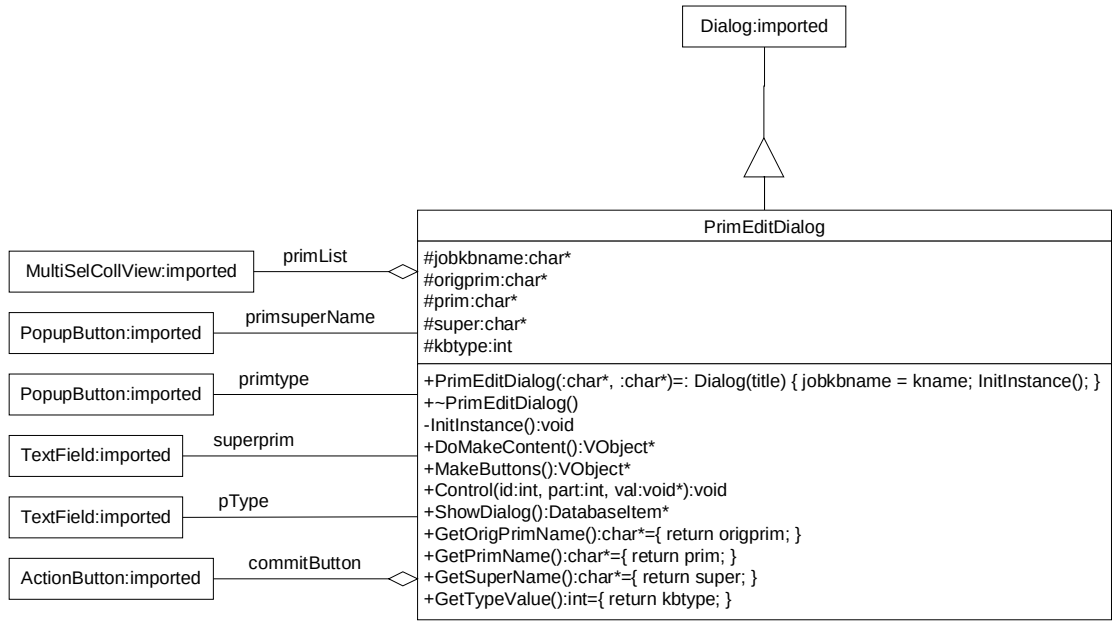
B.2.2.3 CKB UI Dialogs

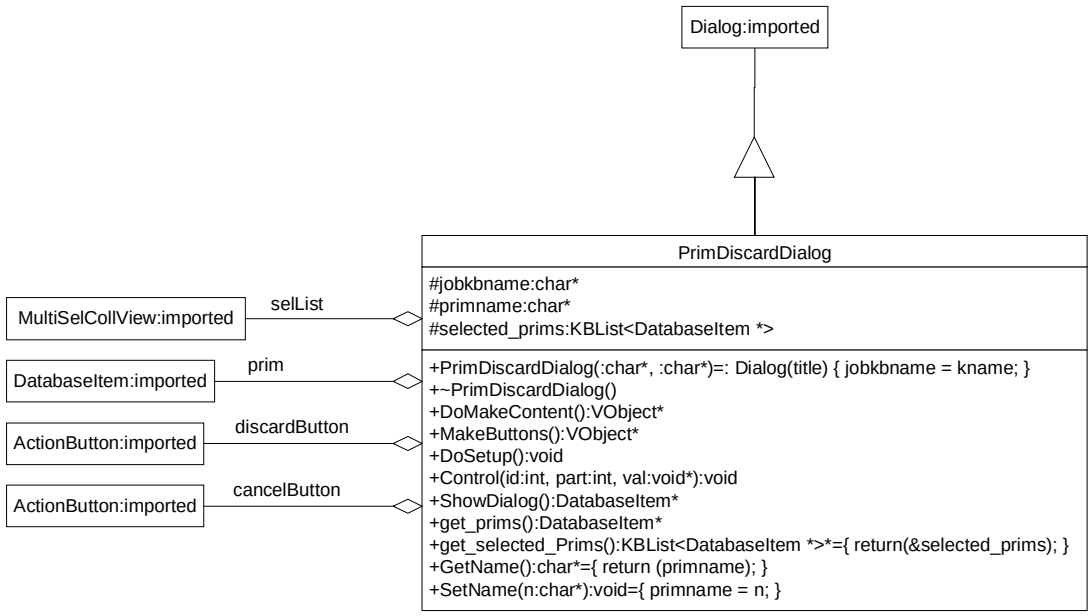


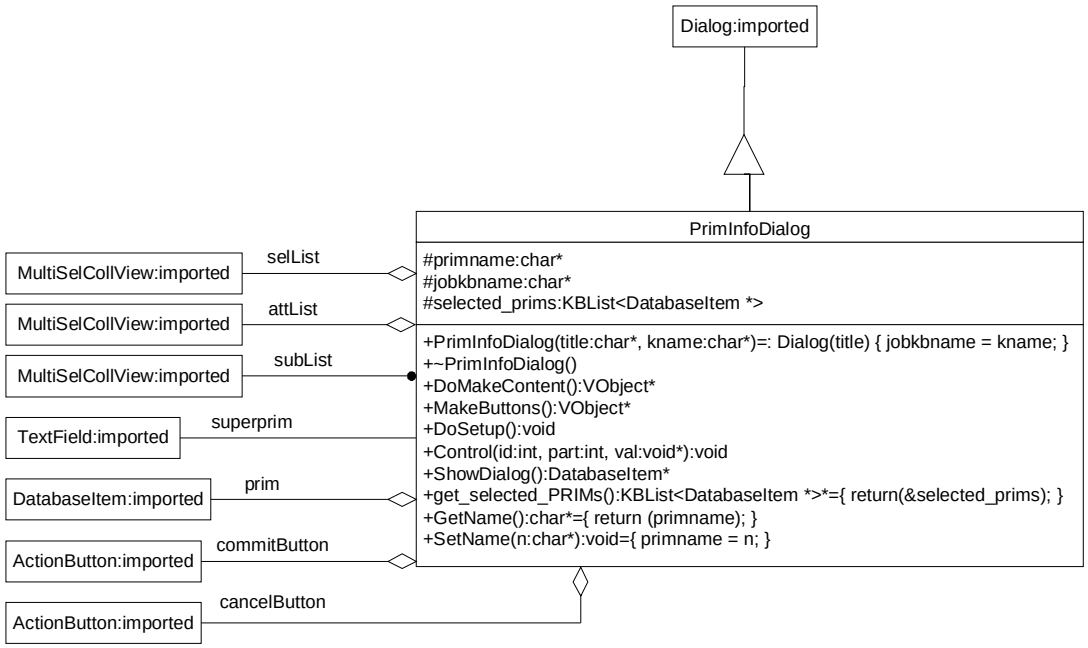

Dialog:imported
KBDiscardDialog
#kbname:char *
#selected_kbs:KBList<DatabaseItem *>
+KBDiscardDialog(title:char*)
+~KBDiscardDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int,part:int,val:void*):void
+ShowDialog():DatabaseItem *
+get_kbs():DatabaseItem *
+get_selected_KBs():KBList<DatabaseItem*>*
+GetName():char*
+SetName(n:char*):void
MultiSelCollView:imported
selList
DatabaseItem:imported
kb
ActionButton:imported
discardButton
ActionButton:imported
cancelButton

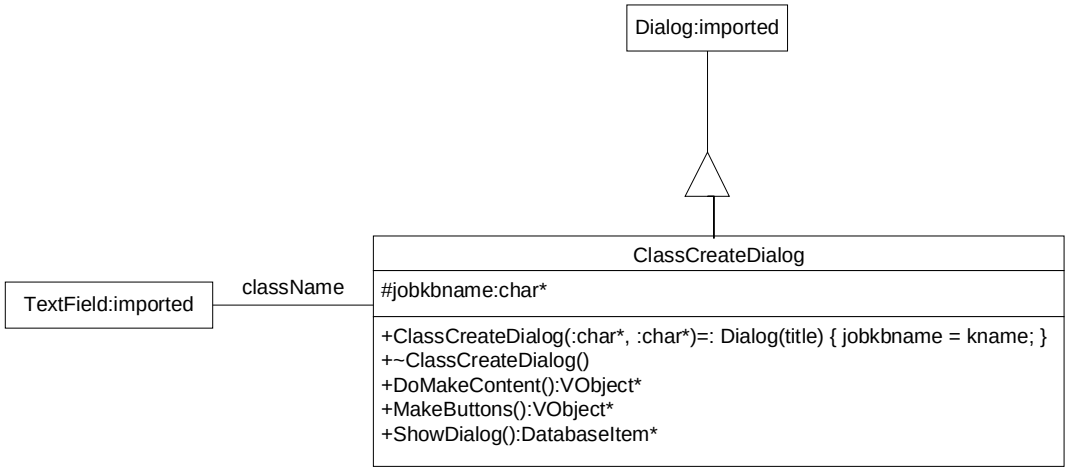

Dialog:imported
KBCleanupDialog
+KBCleanupDialog(title:char*)=: Dialog(title) { }
+~KBCleanupDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
TextField:imported
cleanupKB

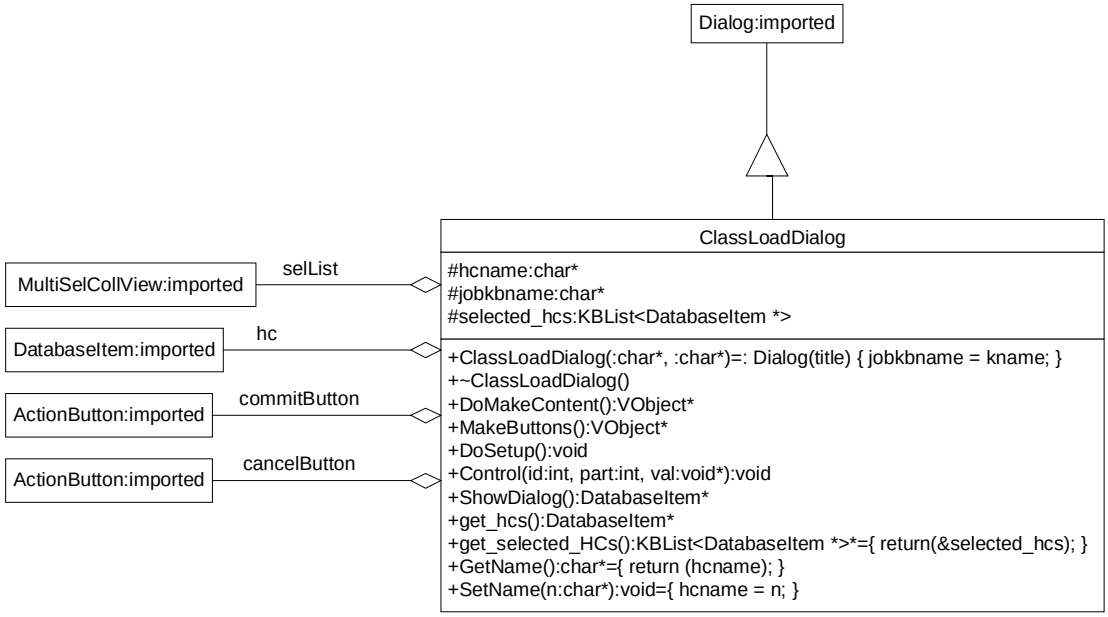

Dialog:imported
PrimCreateDialog
#jobkbname:char*
#prim:char*
#super:char*
#kbtype:int
+PrimCreateDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~PrimCreateDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+ShowDialog(jobNumber:int):DatabaseItem*
+GetPrimName():char*={ return prim; }
+GetSuperName():char*={ return super; }
+GetTypeValue():int={ return kbtype; }
TextField:imported
primName
PopupButton:imported
primsuperName
PopupButton:imported
primtype

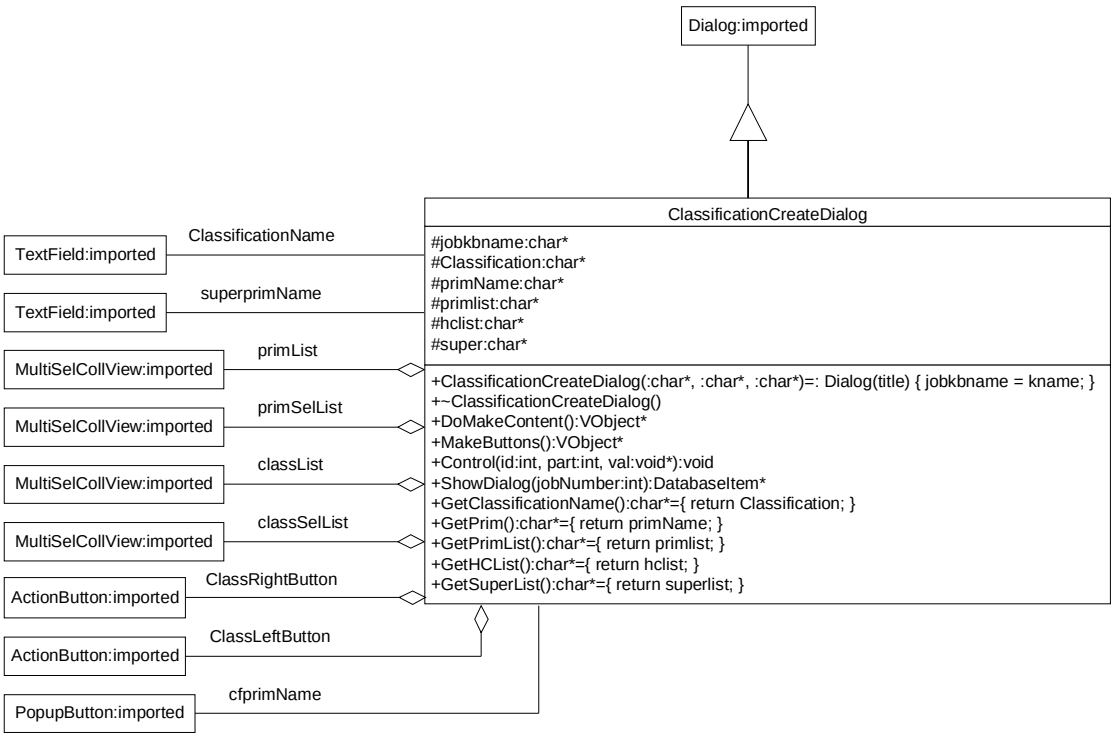

Dialog:imported
PrimEditDialog
#jobkbname:char*
#origprim:char*
#prim:char*
#super:char*
#kbtype:int
+PrimEditDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; InitInstance(); }
+~PrimEditDialog()
-InitInstance():void
+DoMakeContent():VObject*
+MakeButtons():VObject*
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+GetOrigPrimName():char*={ return origprim; }
+GetPrimName():char*={ return prim; }
+GetSuperName():char*={ return super; }
+GetTypeValue():int={ return kbtype; }
MultiSelCollView:imported
primList
PopupButton:imported
primsuperName
PopupButton:imported
primtype
TextField:imported
superprim
TextField:imported
pType
ActionButton:imported
commitButton

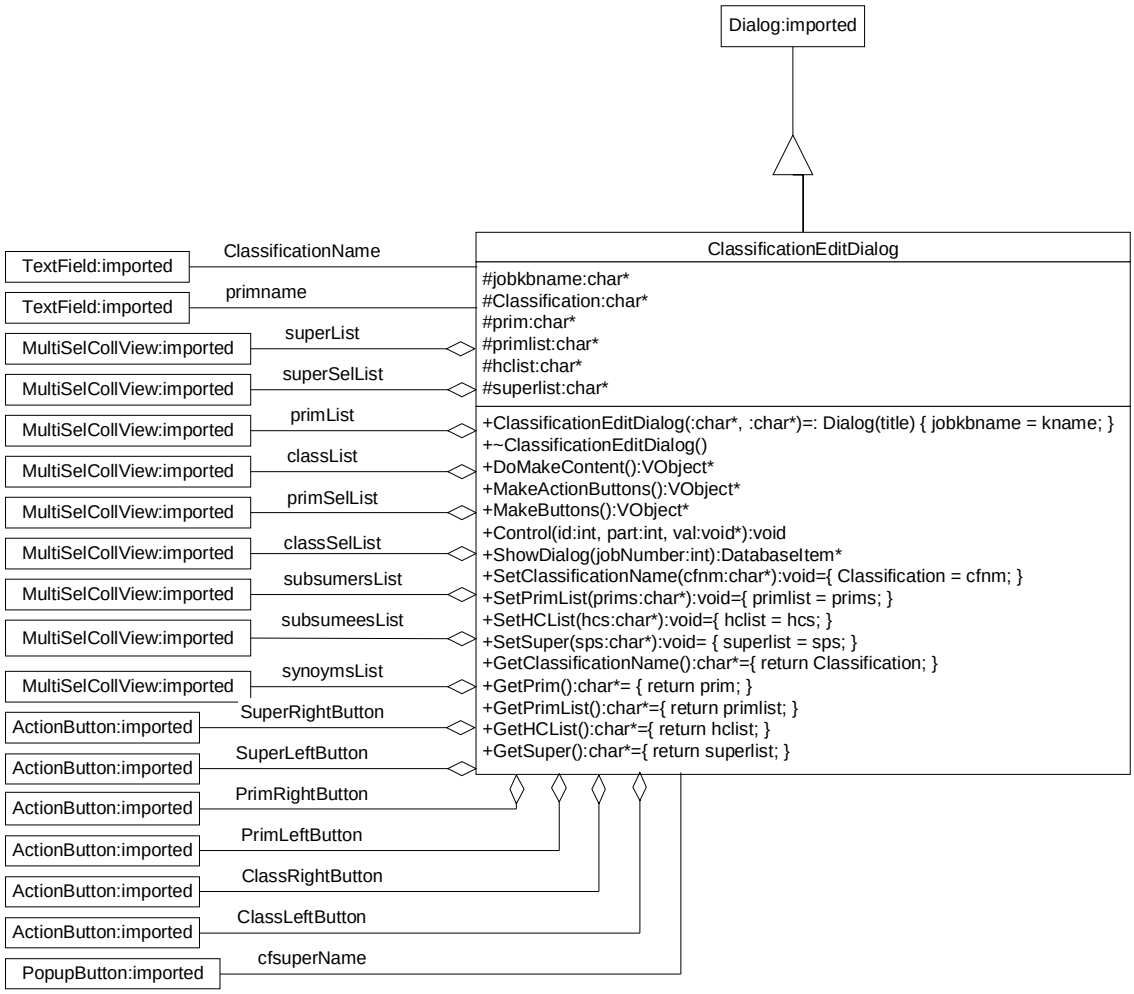

Dialog:imported
PrimDiscardDialog
#jobkbname:char*
#primname:char*
#selected_prims:KBList<DatabaseItem *>
+PrimDiscardDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~PrimDiscardDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+get_prims():DatabaseItem*
+get_selected_Prims():KBList<DatabaseItem *>*={ return(&selected_prims); }
+GetName():char*={ return (primname); }
+SetName(n:char*):void={ primname = n; }
MultiSelCollView:imported
selList
DatabaseItem:imported
prim
ActionButton:imported
discardButton
ActionButton:imported
cancelButton

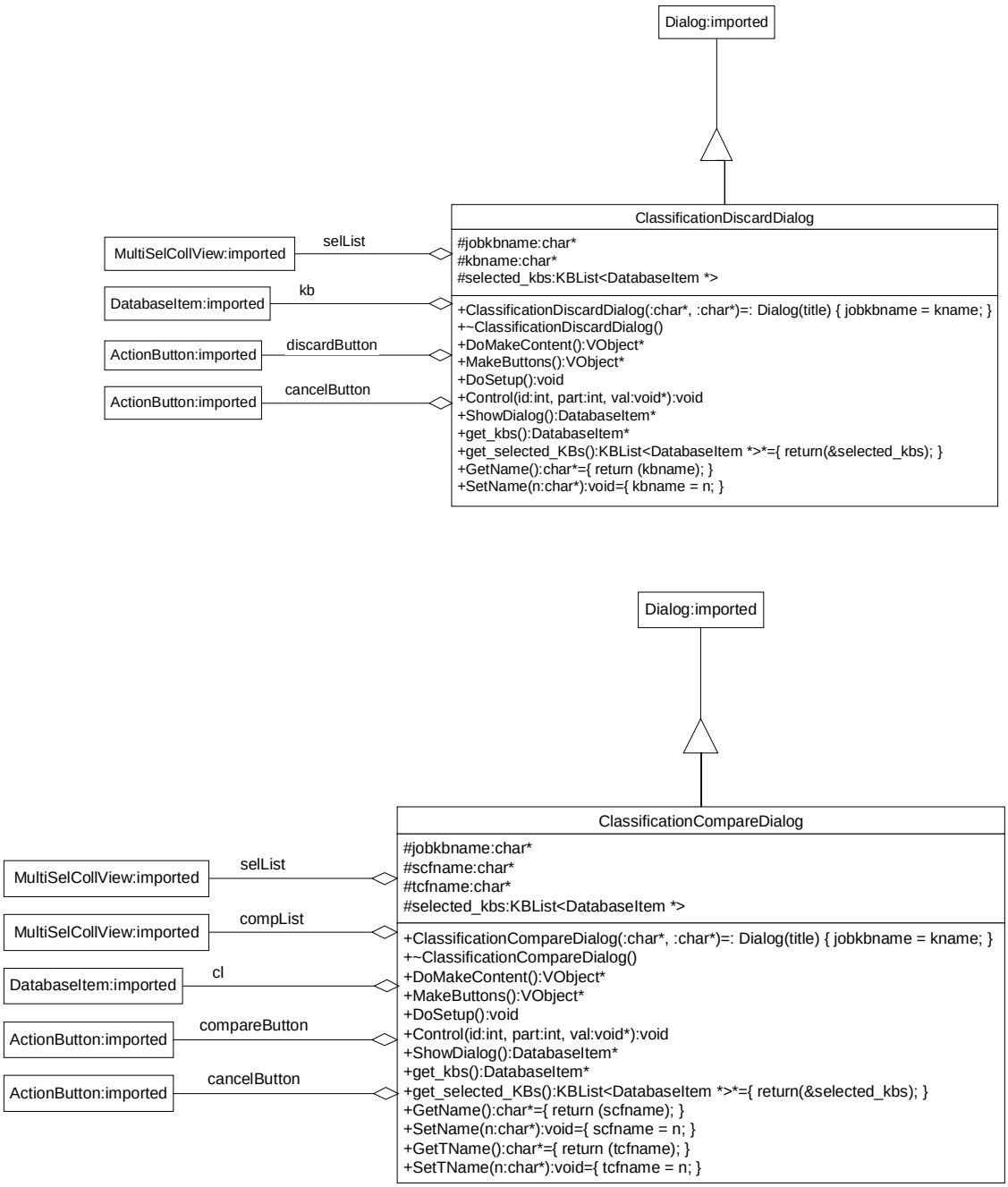

Dialog:imported
PrimInfoDialog
#primname:char*
#jobkbname:char*
#selected_prims:KBList<DatabaseItem *>
+PrimInfoDialog(title:char*, kname:char*)=: Dialog(title) { jobkbname = kname; }
+~PrimInfoDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+get_selected_PRIMs():KBList<DatabaseItem *>*={ return(&selected_prims); }
+GetName():char*={ return (primname); }
+SetName(n:char*):void={ primname = n; }
MultiSelCollView:imported
selList
MultiSelCollView:imported
attList
MultiSelCollView:imported
subList
TextField:imported
superprim
DatabaseItem:imported
prim
ActionButton:imported
commitButton
ActionButton:imported
cancelButton

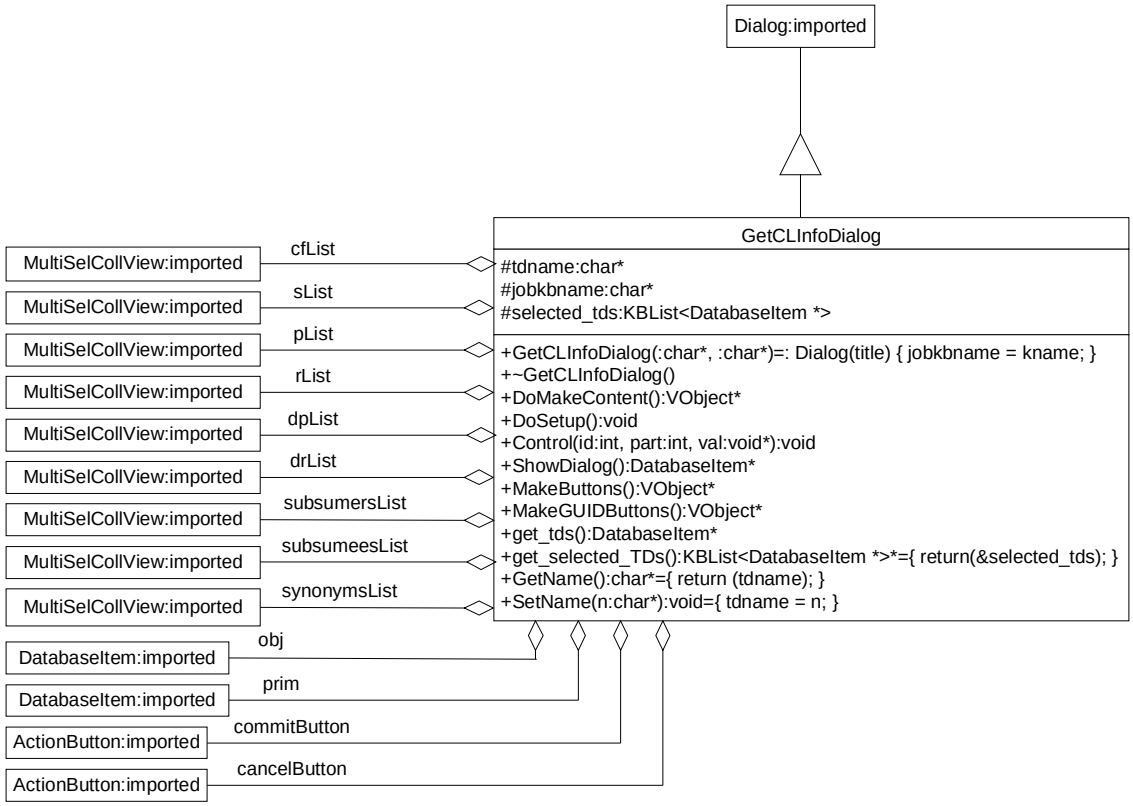

Dialog:imported
ClassCreateDialog
#jobkbname:char*
+ClassCreateDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassCreateDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+ShowDialog():DatabaseItem*
TextField:imported
className

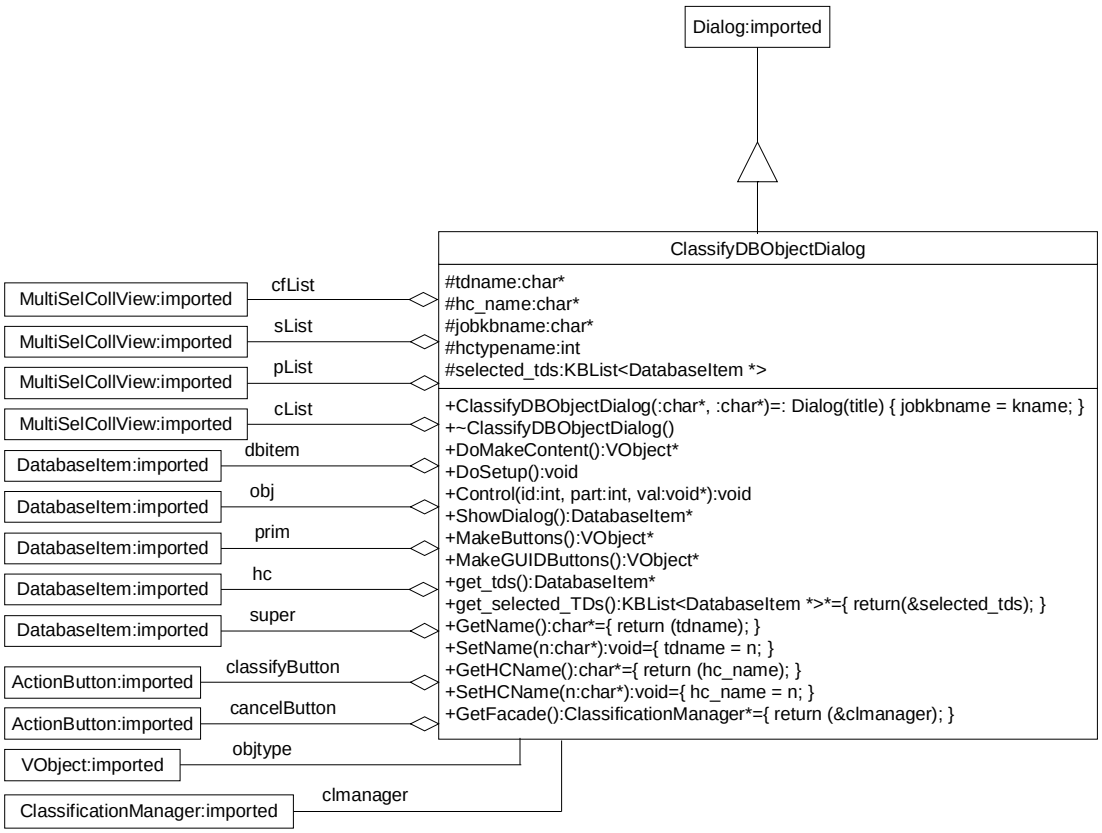

Dialog:imported
ClassLoadDialog
#hcname:char*
#jobkbname:char*
#selected_hcs:KBList<DatabaseItem *>
+ClassLoadDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassLoadDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+get_hcs():DatabaseItem*
+get_selected_HCs():KBList<DatabaseItem *>*={ return(&selected_hcs); }
+GetName():char*={ return (hcname); }
+SetName(n:char*):void={ hcname = n; }
MultiSelCollView:imported
selList
DatabaseItem:imported
hc
ActionButton:imported
commitButton
ActionButton:imported
cancelButton

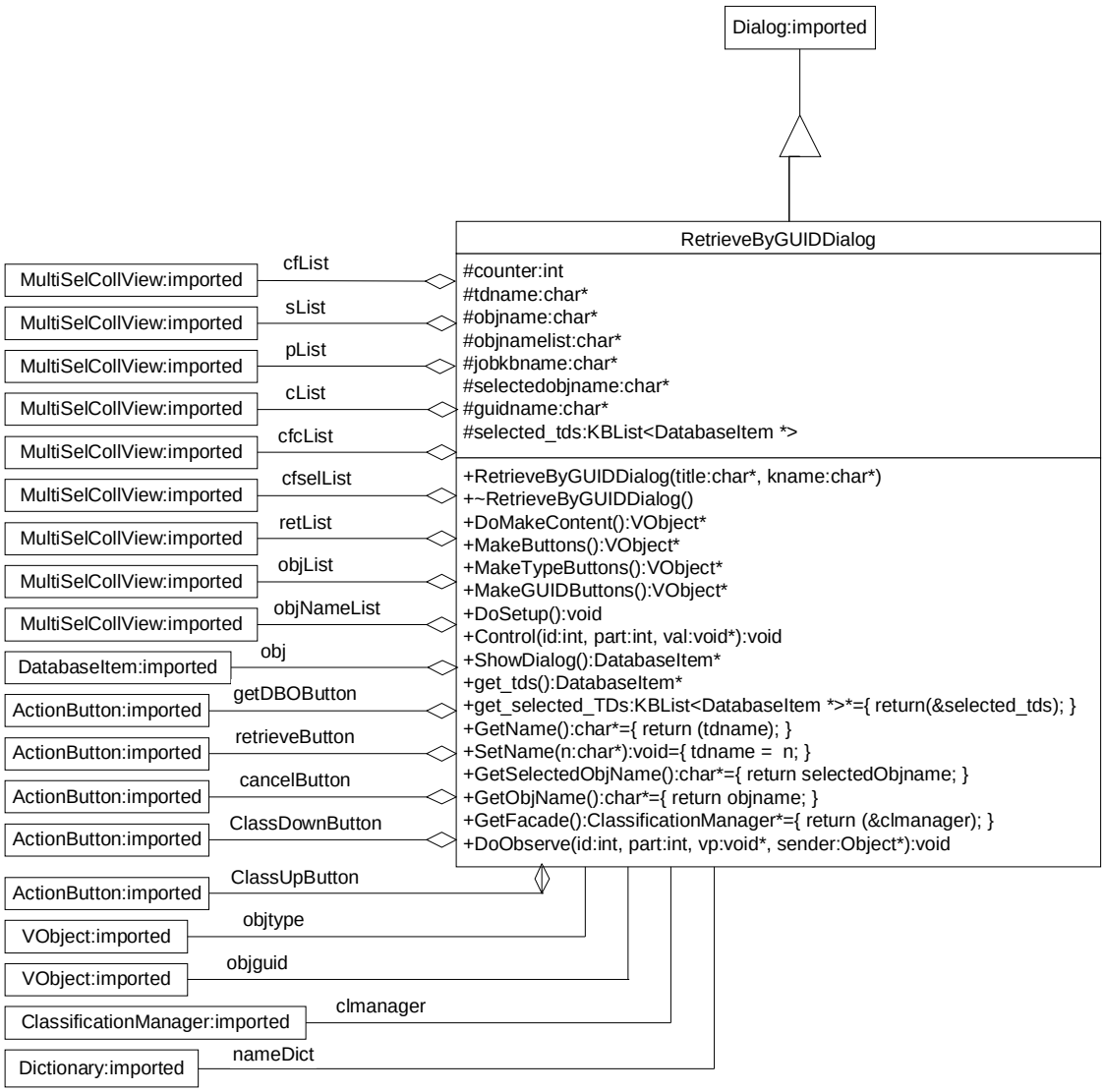

Dialog:imported
ClassificationCreateDialog
#jobkbname:char*
#Classification:char*
#primName:char*
#primlist:char*
#hclist:char*
#super:char*
+ClassificationCreateDialog(:char*, :char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassificationCreateDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+Control(id:int, part:int, val:void*):void
+ShowDialog(jobNumber:int):DatabaseItem*
+GetClassificationName():char*={ return Classification; }
+GetPrim():char*={ return primName; }
+GetPrimList():char*={ return primlist; }
+GetHCList():char*={ return hclist; }
+GetSuperList():char*={ return superlist; }
TextField:imported
ClassificationName
TextField:imported
superprimName
MultiSelCollView:imported
primList
MultiSelCollView:imported
primSelList
MultiSelCollView:imported
classList
MultiSelCollView:imported
classSelList
ActionButton:imported
ClassRightButton
ActionButton:imported
ClassLeftButton
PopupButton:imported
cfprimName


Dialog:imported
ClassificationEditDialog
#jobkbname:char*
#Classification:char*
#prim:char*
#primlist:char*
#hclist:char*
#superlist:char*
+ClassificationEditDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassificationEditDialog()
+DoMakeContent():VObject*
+MakeActionButtons():VObject*
+MakeButtons():VObject*
+Control(id:int, part:int, val:void*):void
+ShowDialog(jobNumber:int):DatabaseItem*
+SetClassificationName(cfnm:char*):void={ Classification = cfnm; }
+SetPrimList(prims:char*):void={ primlist = prims; }
+SetHCList(hcs:char*):void={ hclist = hcs; }
+SetSuper(sps:char*):void= { superlist = sps; }
+GetClassificationName():char*={ return Classification; }
+GetPrim():char*= { return prim; }
+GetPrimList():char*={ return primlist; }
+GetHCList():char*={ return hclist; }
+GetSuper():char*={ return superlist; }
TextField:imported
ClassificationName
TextField:imported
primname
MultiSelCollView:imported
superList
MultiSelCollView:imported
superSelList
MultiSelCollView:imported
primList
MultiSelCollView:imported
classList
MultiSelCollView:imported
primSelList
MultiSelCollView:imported
classSelList
MultiSelCollView:imported
subsumersList
MultiSelCollView:imported
subsumeesList
MultiSelCollView:imported
synoymsList
ActionButton:imported
SuperRightButton
ActionButton:imported
SuperLeftButton
ActionButton:imported
PrimRightButton
ActionButton:imported
PrimLeftButton
ActionButton:imported
ClassRightButton
ActionButton:imported
ClassLeftButton
PopupButton:imported
cfsuperName

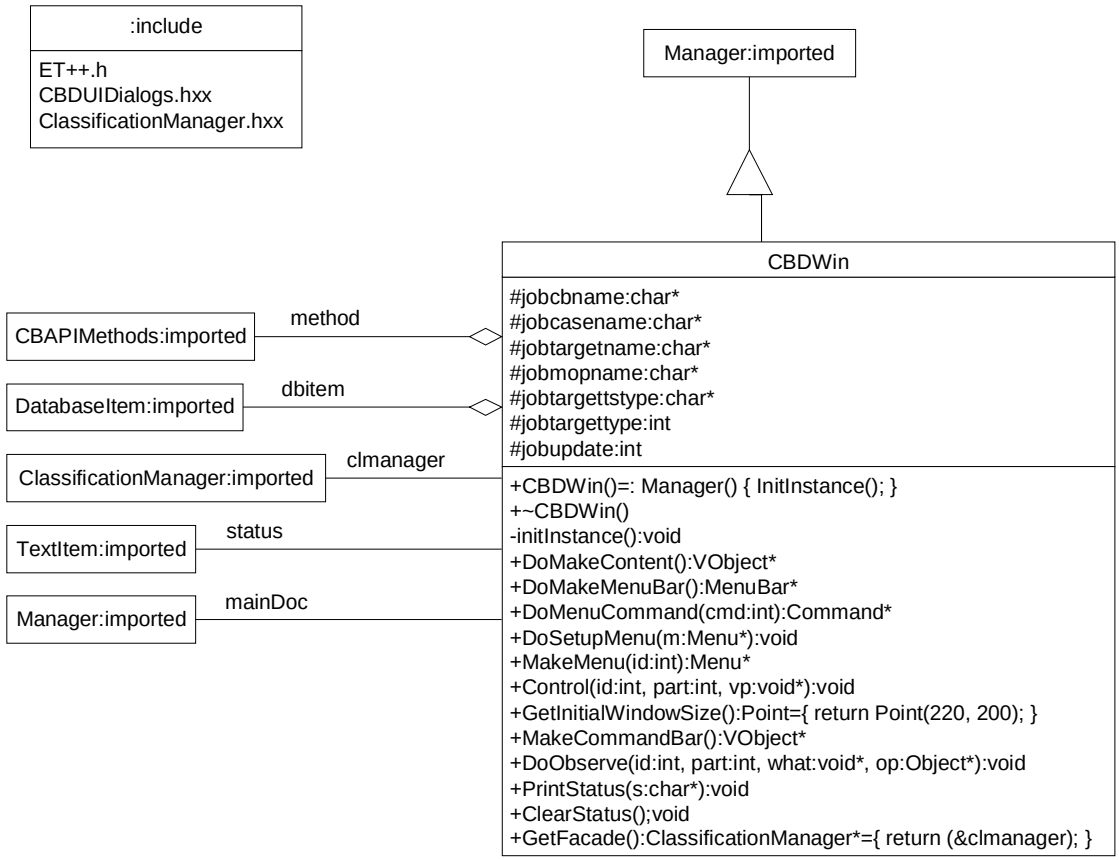

Dialog:imported
ClassificationDiscardDialog
#jobkbname:char*
#kbname:char*
#selected_kbs:KBList<DatabaseItem *>
+ClassificationDiscardDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassificationDiscardDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+get_kbs():DatabaseItem*
+get_selected_KBs():KBList<DatabaseItem *>*={ return(&selected_kbs); }
+GetName():char*={ return (kbname); }
+SetName(n:char*):void={ kbname = n; }
MultiSelCollView:imported
selList
DatabaseItem:imported
kb
ActionButton:imported
discardButton
ActionButton:imported
cancelButton
Dialog:imported
ClassificationCompareDialog
#jobkbname:char*
#scfname:char*
#tcfname:char*
#selected_kbs:KBList<DatabaseItem *>
+ClassificationCompareDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassificationCompareDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+get_kbs():DatabaseItem*
+get_selected_KBs():KBList<DatabaseItem *>*={ return(&selected_kbs); }
+GetName():char*={ return (scfname); }
+SetName(n:char*):void={ scfname = n; }
+GetTName():char*={ return (tcfname); }
+SetTName(n:char*):void={ tcfname = n; }
MultiSelCollView:imported
selList
MultiSelCollView:imported
compList
DatabaseItem:imported
cl
ActionButton:imported
compareButton
ActionButton:imported
cancelButton


Dialog:imported
GetCLInfoDialog
#tdname:char*
#jobkbname:char*
#selected_tds:KBList<DatabaseItem *>
+GetCLInfoDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~GetCLInfoDialog()
+DoMakeContent():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+MakeButtons():VObject*
+MakeGUIDButtons():VObject*
+get_tds():DatabaseItem*
+get_selected_TDs():KBList<DatabaseItem *>*={ return(&selected_tds); }
+GetName():char*={ return (tdname); }
+SetName(n:char*):void={ tdname = n; }
MultiSelCollView:imported
cfList
MultiSelCollView:imported
sList
MultiSelCollView:imported
pList
MultiSelCollView:imported
rList
MultiSelCollView:imported
dpList
MultiSelCollView:imported
drList
MultiSelCollView:imported
subsumersList
MultiSelCollView:imported
subsumeesList
MultiSelCollView:imported
synonymsList
DatabaseItem:imported
obj
DatabaseItem:imported
prim
ActionButton:imported
commitButton
ActionButton:imported
cancelButton

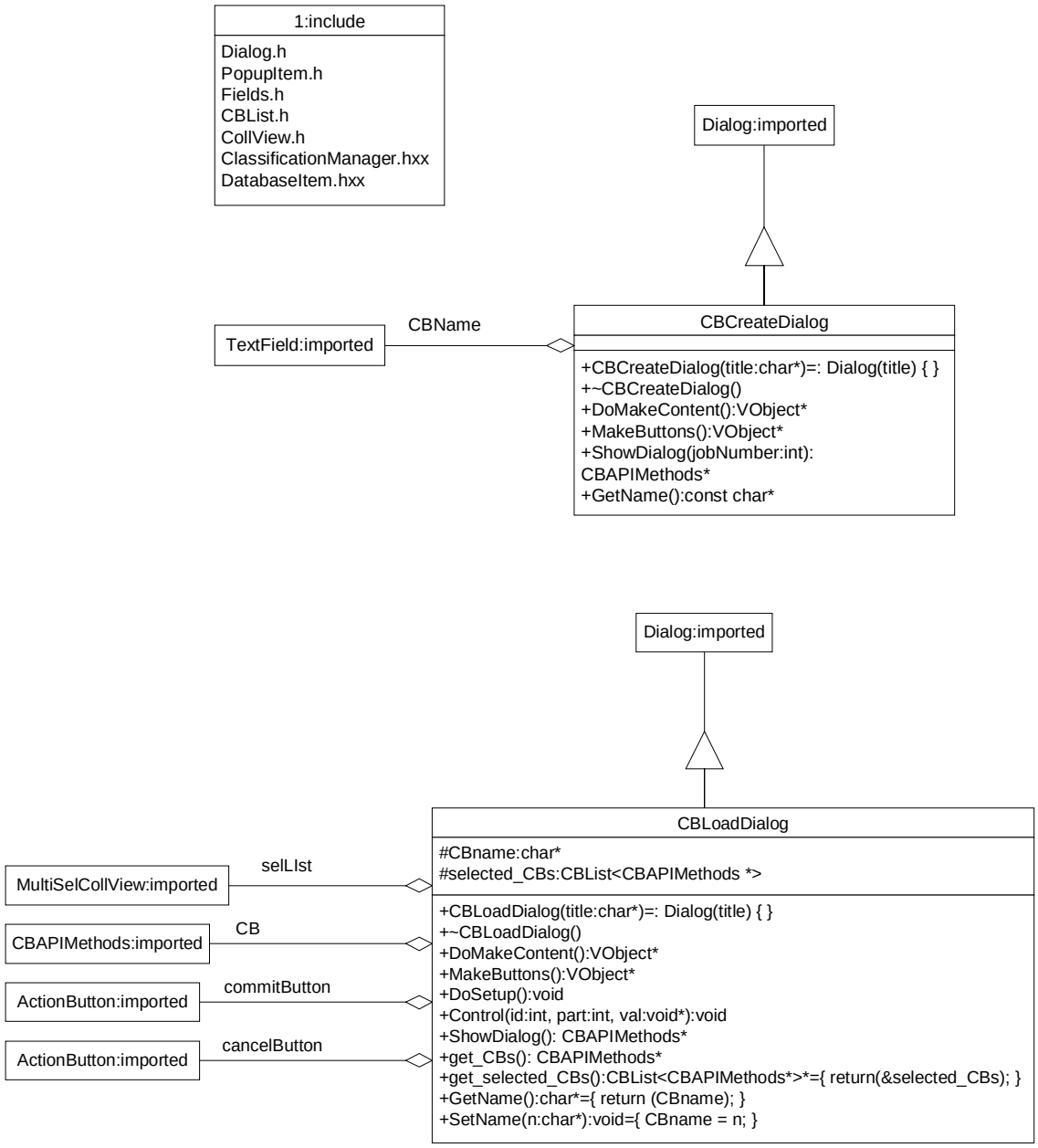

Dialog:imported
ClassifyDBObjectDialog
#tdname:char*
#hc_name:char*
#jobkbname:char*
#hctypename:int
#selected_tds:KBList<DatabaseItem *>
+ClassifyDBObjectDialog(:char*, :char*)=: Dialog(title) { jobkbname = kname; }
+~ClassifyDBObjectDialog()
+DoMakeContent():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+MakeButtons():VObject*
+MakeGUIDButtons():VObject*
+get_tds():DatabaseItem*
+get_selected_TDs():KBList<DatabaseItem *>*={ return(&selected_tds); }
+GetName():char*={ return (tdname); }
+SetName(n:char*):void={ tdname = n; }
+GetHCName():char*={ return (hc_name); }
+SetHCName(n:char*):void={ hc_name = n; }
+GetFacade():ClassificationManager*={ return (&clmanager); }
MultiSelCollView:imported
cfList
MultiSelCollView:imported
sList
MultiSelCollView:imported
pList
MultiSelCollView:imported
cList
DatabaseItem:imported
dbitem
DatabaseItem:imported
obj
DatabaseItem:imported
prim
DatabaseItem:imported
hc
DatabaseItem:imported
super
ActionButton:imported
classifyButton
ActionButton:imported
cancelButton
VObject:imported
objtype
ClassificationManager:imported
clmanager

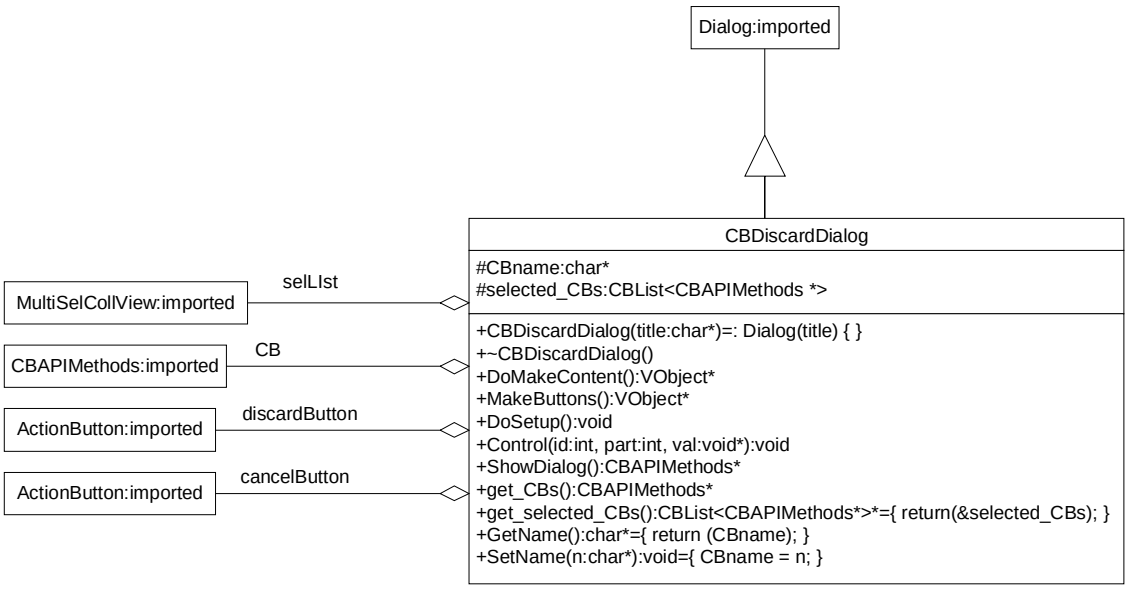

Dialog:imported
RetrieveByGUIDDialog
#counter:int
#tdname:char*
#objname:char*
#objnamelist:char*
#jobkbname:char*
#selectedobjname:char*
#guidname:char*
#selected_tds:KBList<DatabaseItem *>
+RetrieveByGUIDDialog(title:char*, kname:char*)
+~RetrieveByGUIDDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+MakeTypeButtons():VObject*
+MakeGUIDButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():DatabaseItem*
+get_tds():DatabaseItem*
+get_selected_TDs:KBList<DatabaseItem *>*={ return(&selected_tds); }
+GetName():char*={ return (tdname); }
+SetName(n:char*):void={ tdname = n; }
+GetSelectedObjName():char*={ return selectedObjname; }
+GetObjName():char*={ return objname; }
+GetFacade():ClassificationManager*={ return (&clmanager); }
+DoObserve(id:int, part:int, vp:void*, sender:Object*):void
MultiSelCollView:imported
cfList
MultiSelCollView:imported
sList
MultiSelCollView:imported
pList
MultiSelCollView:imported
cList
MultiSelCollView:imported
cfcList
MultiSelCollView:imported
cfselList
MultiSelCollView:imported
retList
MultiSelCollView:imported
objList
MultiSelCollView:imported
objNameList
DatabaseItem:imported
obj
ActionButton:imported
getDBOButton
ActionButton:imported
retrieveButton
ActionButton:imported
cancelButton
ActionButton:imported
ClassDownButton
ActionButton:imported
ClassUpButton
VObject:imported
objtype
VObject:imported
objguid
ClassificationManager:imported
clmanager
Dictionary:imported
nameDict

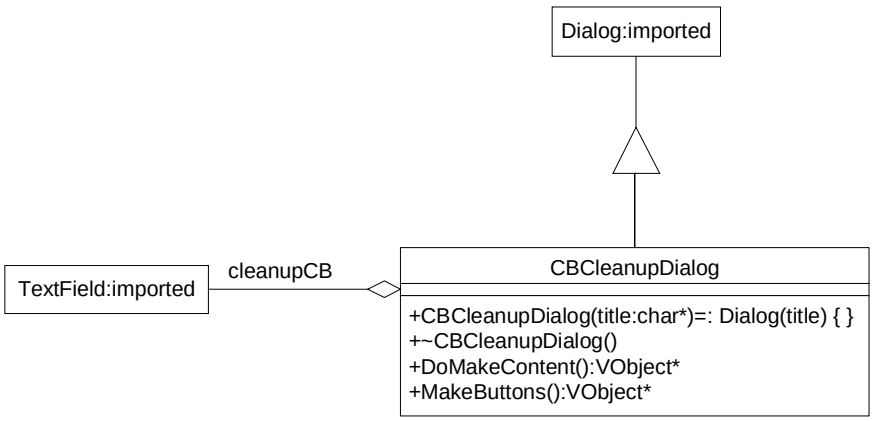
B.2.2.4 The Main CBD Window

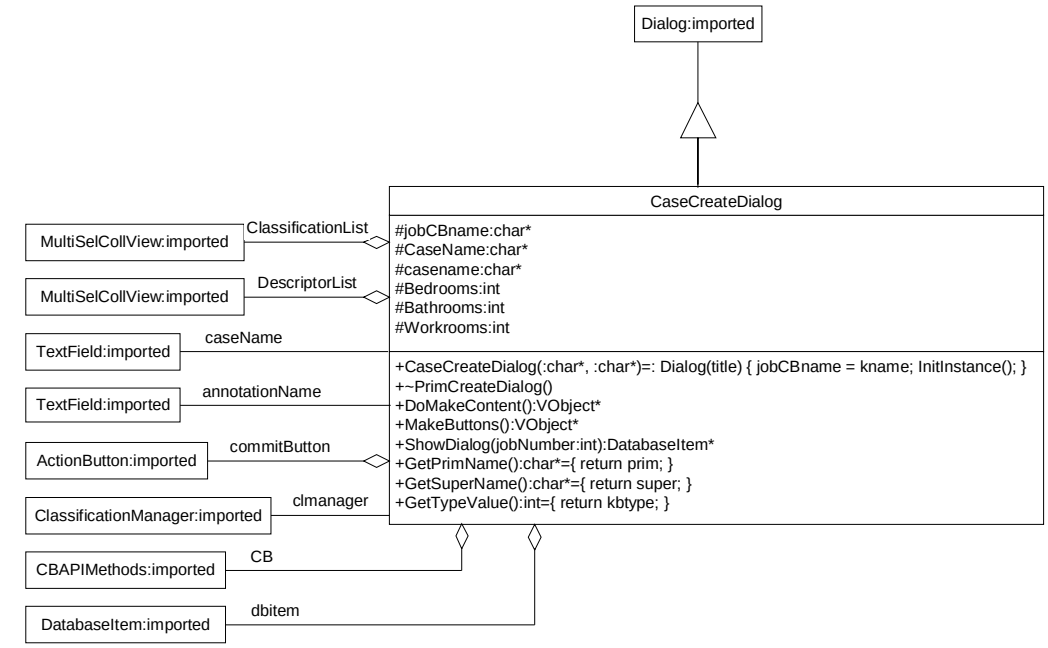


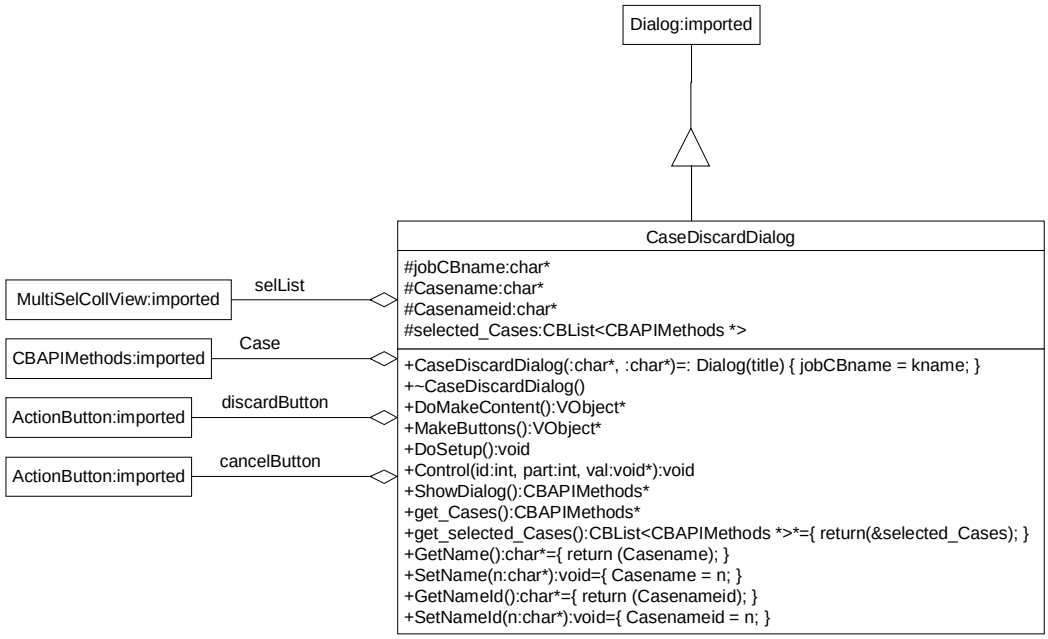
B.2.2.5 CBD UI Dialogs

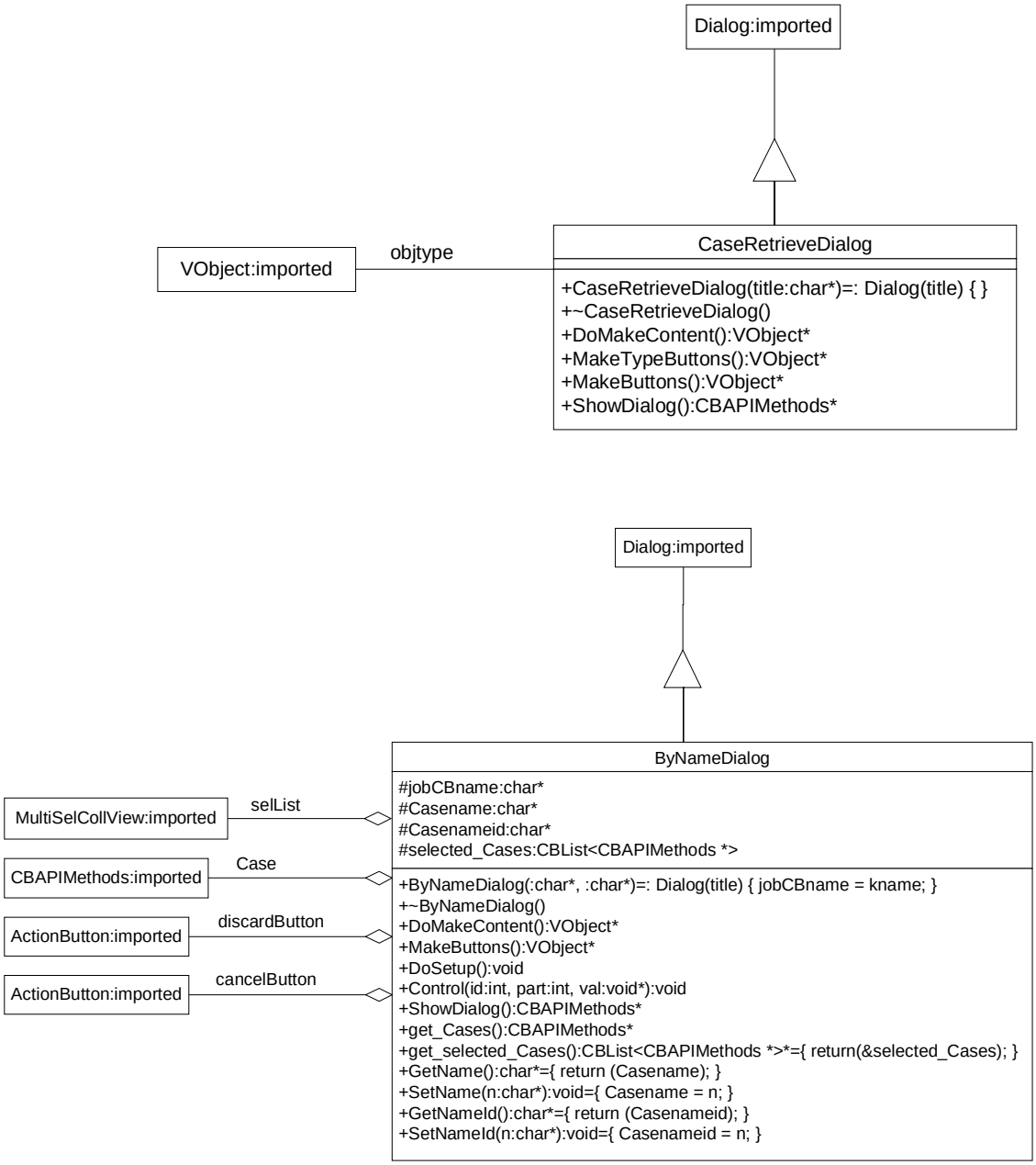


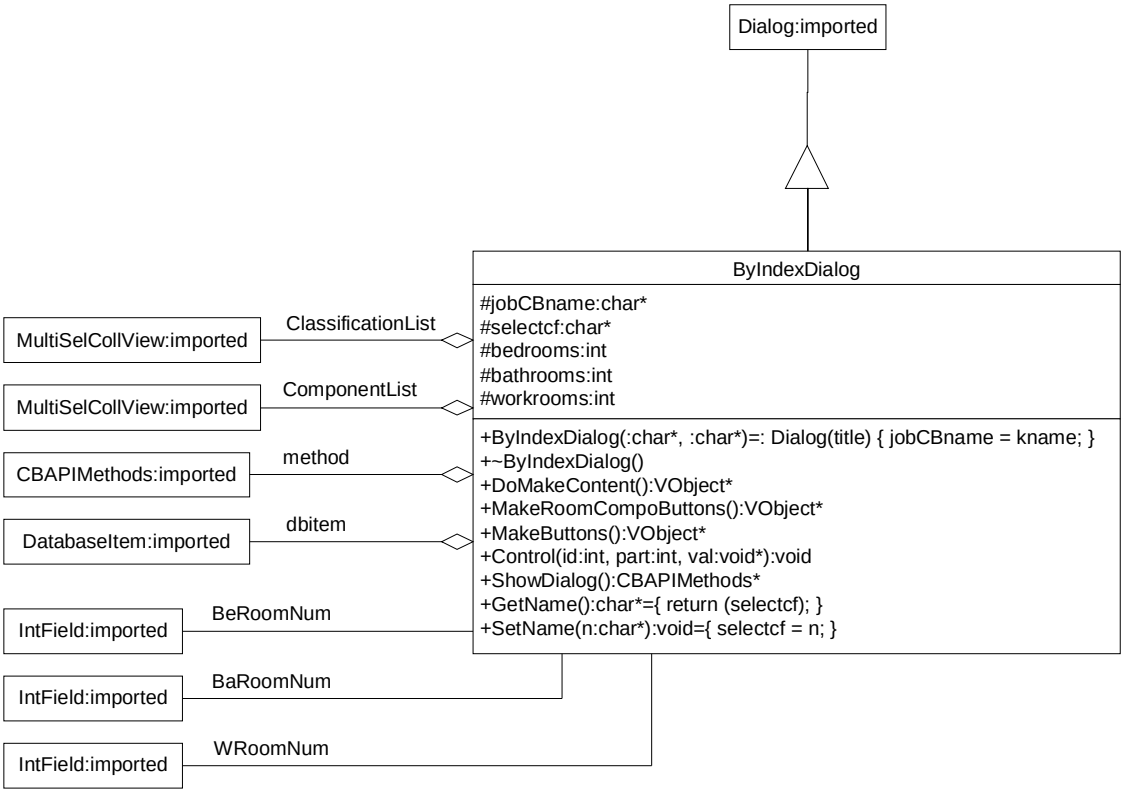

Dialog:imported
CBDiscardDialog
#CBname:char*
#selected_CBs:CBList<CBAPIMethods *>
+CBDiscardDialog(title:char*)=: Dialog(title) { }
+~CBDiscardDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():CBAPIMethods*
+get_CBs():CBAPIMethods*
+get_selected_CBs():CBList<CBAPIMethods*>*={ return(&selected_CBs); }
+GetName():char*={ return (CBname); }
+SetName(n:char*):void={ CBname = n; }
MultiSelCollView:imported
selLIst
CBAPIMethods:imported
CB
ActionButton:imported
discardButton
ActionButton:imported
cancelButton

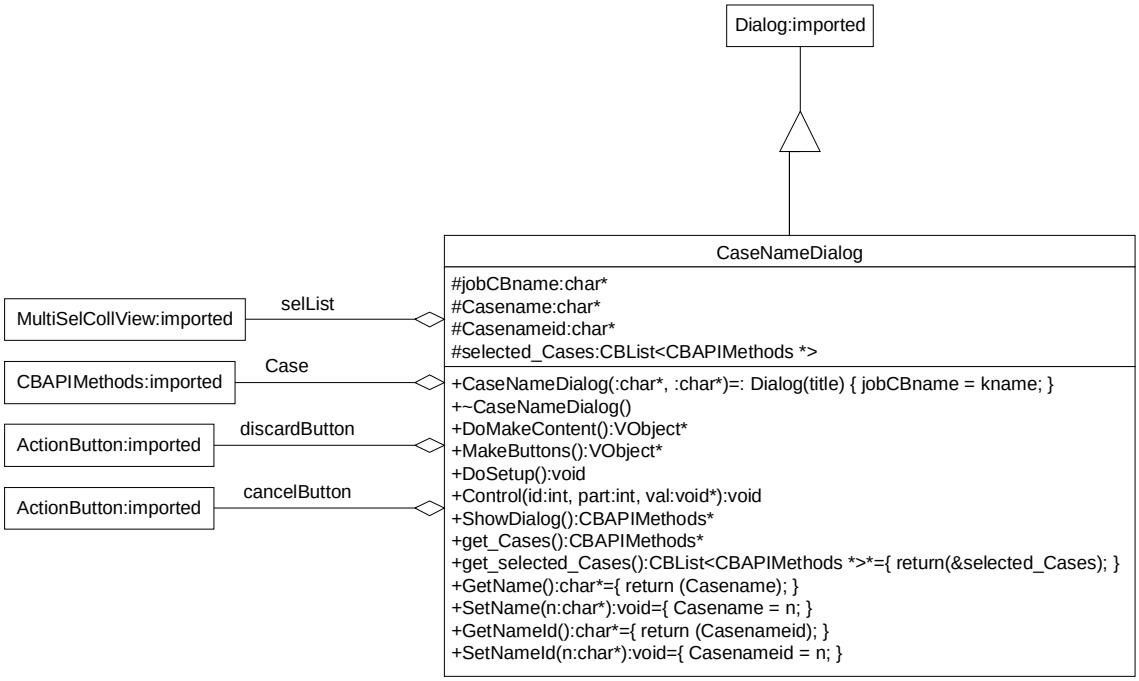

Dialog:imported
CBCleanupDialog
+CBCleanupDialog(title:char*)=: Dialog(title) { }
+~CBCleanupDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
TextField:imported
cleanupCB


Dialog:imported
CaseCreateDialog
#jobCBname:char*
#CaseName:char*
#casename:char*
#Bedrooms:int
#Bathrooms:int
#Workrooms:int
+CaseCreateDialog(:char*, :char*)=: Dialog(title) { jobCBname = kname; InitInstance(); }
+~PrimCreateDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+ShowDialog(jobNumber:int):DatabaseItem*
+GetPrimName():char*={ return prim; }
+GetSuperName():char*={ return super; }
+GetTypeValue():int={ return kbtype; }
MultiSelCollView:imported
ClassificationList
MultiSelCollView:imported
DescriptorList
TextField:imported
caseName
TextField:imported
annotationName
ActionButton:imported
commitButton
ClassificationManager:imported
clmanager
CBAPIMethods:imported
CB
DatabaseItem:imported
dbitem


Dialog:imported
CaseDiscardDialog
#jobCBname:char*
#Casename:char*
#Casenameid:char*
#selected_Cases:CBList<CBAPIMethods *>
+CaseDiscardDialog(:char*, :char*)=: Dialog(title) { jobCBname = kname; }
+~CaseDiscardDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():CBAPIMethods*
+get_Cases():CBAPIMethods*
+get_selected_Cases():CBList<CBAPIMethods *>*={ return(&selected_Cases); }
+GetName():char*={ return (Casename); }
+SetName(n:char*):void={ Casename = n; }
+GetNameId():char*={ return (Casenameid); }
+SetNameId(n:char*):void={ Casenameid = n; }
MultiSelCollView:imported
selList
CBAPIMethods:imported
Case
ActionButton:imported
discardButton
ActionButton:imported
cancelButton


Dialog:imported
CaseRetrieveDialog
+CaseRetrieveDialog(title:char*)=: Dialog(title) { }
+~CaseRetrieveDialog()
+DoMakeContent():VObject*
+MakeTypeButtons():VObject*
+MakeButtons():VObject*
+ShowDialog():CBAPIMethods*
VObject:imported
objtype
Dialog:imported
ByNameDialog
#jobCBname:char*
#Casename:char*
#Casenameid:char*
#selected_Cases:CBList<CBAPIMethods *>
+ByNameDialog(:char*, :char*)=: Dialog(title) { jobCBname = kname; }
+~ByNameDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():CBAPIMethods*
+get_Cases():CBAPIMethods*
+get_selected_Cases():CBList<CBAPIMethods *>*={ return(&selected_Cases); }
+GetName():char*={ return (Casename); }
+SetName(n:char*):void={ Casename = n; }
+GetNameId():char*={ return (Casenameid); }
+SetNameId(n:char*):void={ Casenameid = n; }
MultiSelCollView:imported
selList
CBAPIMethods:imported
Case
ActionButton:imported
discardButton
ActionButton:imported
cancelButton

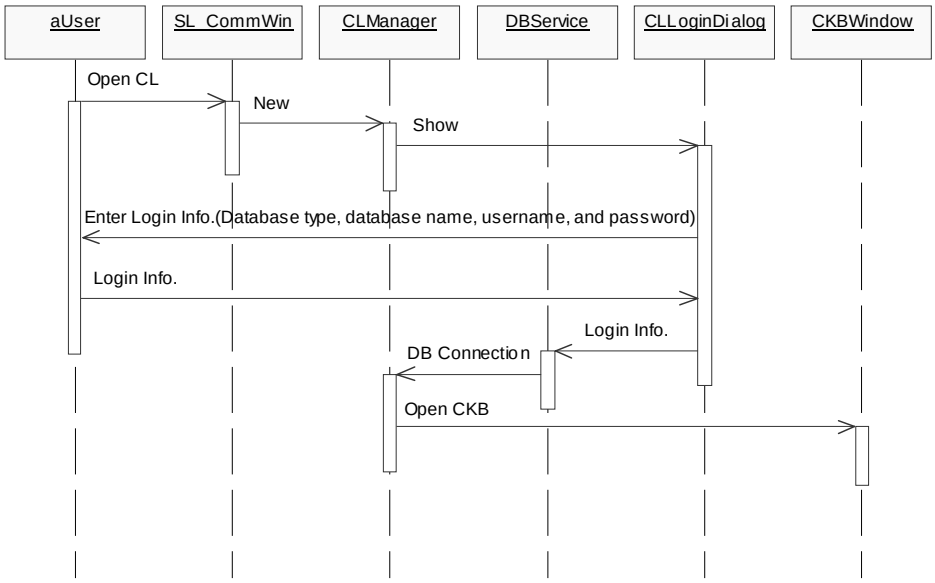

Dialog:imported
ByIndexDialog
#jobCBname:char*
#selectcf:char*
#bedrooms:int
#bathrooms:int
#workrooms:int
+ByIndexDialog(:char*, :char*)=: Dialog(title) { jobCBname = kname; }
+~ByIndexDialog()
+DoMakeContent():VObject*
+MakeRoomCompoButtons():VObject*
+MakeButtons():VObject*
+Control(id:int, part:int, val:void*):void
+ShowDialog():CBAPIMethods*
+GetName():char*={ return (selectcf); }
+SetName(n:char*):void={ selectcf = n; }
MultiSelCollView:imported
ClassificationList
MultiSelCollView:imported
ComponentList
CBAPIMethods:imported
method
DatabaseItem:imported
dbitem
IntField:imported
BeRoomNum
IntField:imported
BaRoomNum
IntField:imported
WRoomNum


Dialog:imported
CaseNameDialog
#jobCBname:char*
#Casename:char*
#Casenameid:char*
#selected_Cases:CBList<CBAPIMethods *>
+CaseNameDialog(:char*, :char*)=: Dialog(title) { jobCBname = kname; }
+~CaseNameDialog()
+DoMakeContent():VObject*
+MakeButtons():VObject*
+DoSetup():void
+Control(id:int, part:int, val:void*):void
+ShowDialog():CBAPIMethods*
+get_Cases():CBAPIMethods*
+get_selected_Cases():CBList<CBAPIMethods *>*={ return(&selected_Cases); }
+GetName():char*={ return (Casename); }
+SetName(n:char*):void={ Casename = n; }
+GetNameId():char*={ return (Casenameid); }
+SetNameId(n:char*):void={ Casenameid = n; }
MultiSelCollView:imported
selList
CBAPIMethods:imported
Case
ActionButton:imported
discardButton
ActionButton:imported
cancelButton

Appendix C

Sequence Diagrams for the Use Cases described in Section 6.2

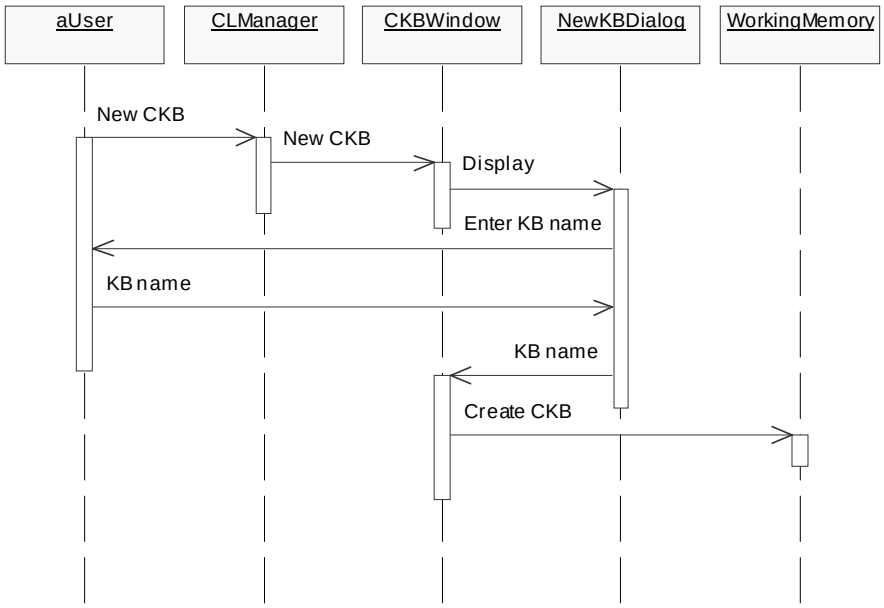
C.1 Start a classification session



Flow of Events:

1. The user selects the “Classification” option from the “Communication” menu in the SCW menu bar.
2. SL_CKB opens the “Classification Login” dialog box. It prompts the user to select a database type and to input a database name, username and password.
3. If the three input values are all correct, SL_CKB connects to the database and opens the Classification Knowledge Base (CKB) window.

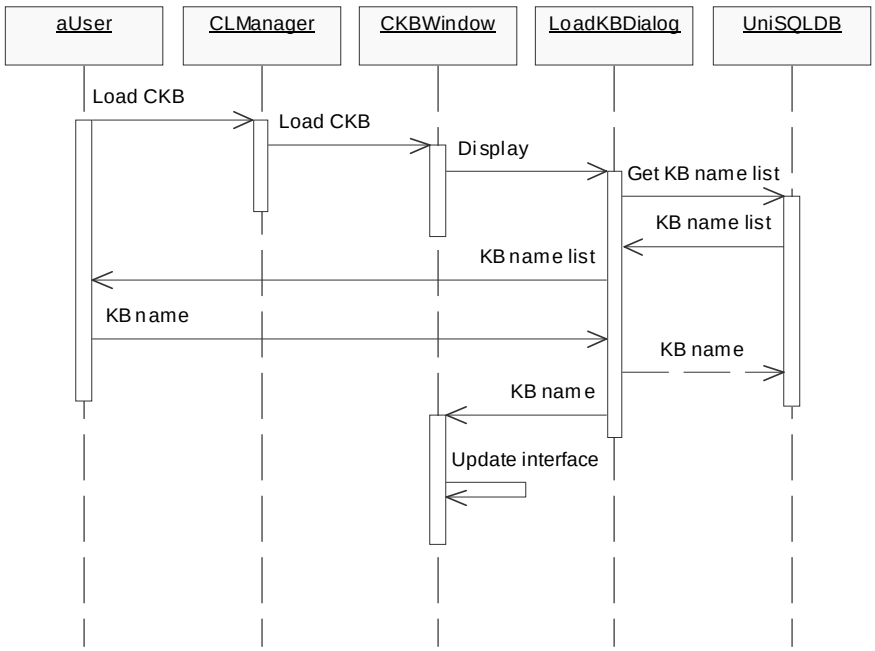
C.2 Create a CKB



Flow of Events:

1. The user selects the “New CKB” option from the “File” menu.
2. SL_CKB opens the “New Knowledge Base” dialog box.
3. The user enters a name in the “New KB Name:” field and hits the “Commit” button.
4. SL_CKB creates a new CKB in working memory, provided a CKB with the same name does not exist in the database.

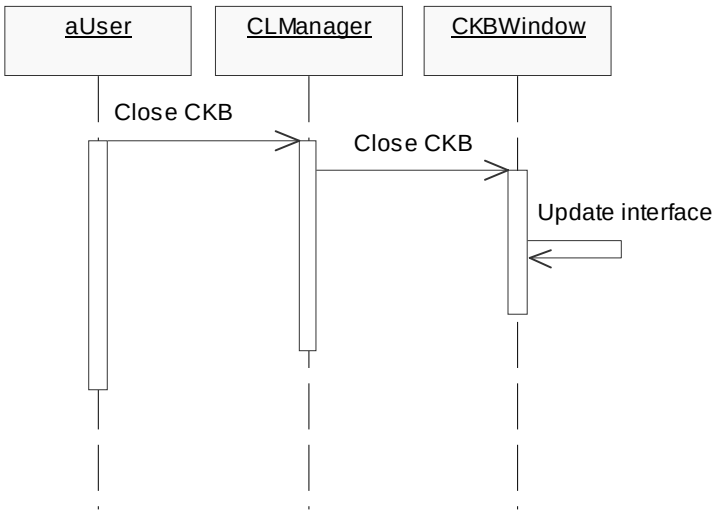
C.3 Load a CKB



Flow of Events:

1. The user selects the “Load CKB” option from the “File” menu.
2. SL_CKB opens the “Load Knowledge Base” dialog box showing a list of existing CKB names in the connected database.
3. The user selects a desired CKB by name and hits the “Commit” button.
4. SL_CKB retrieves the respective CKB from the database and makes it the currently active one.

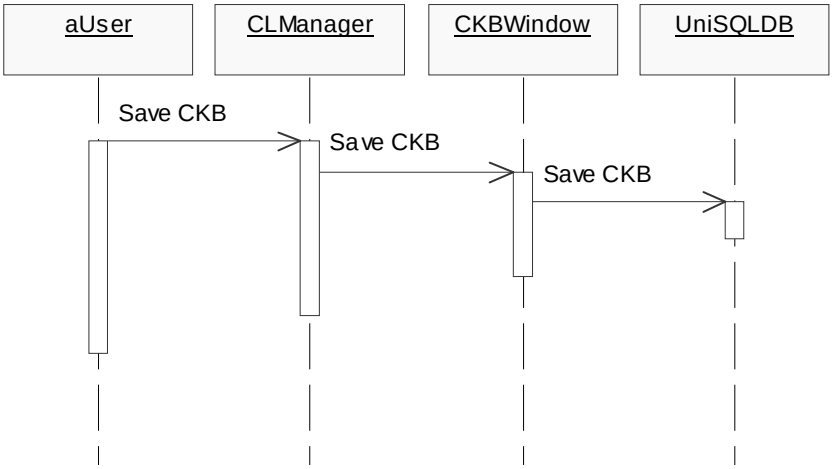
C.4 Close a CKB



Flow of Events:

1. The user selects the “Close CKB” option from the “File” menu.
2. If changes have been made after the last save, the SL_CKB asks the user if the changes should be saved and acts accordingly.
3. The currently active CKB is closed.

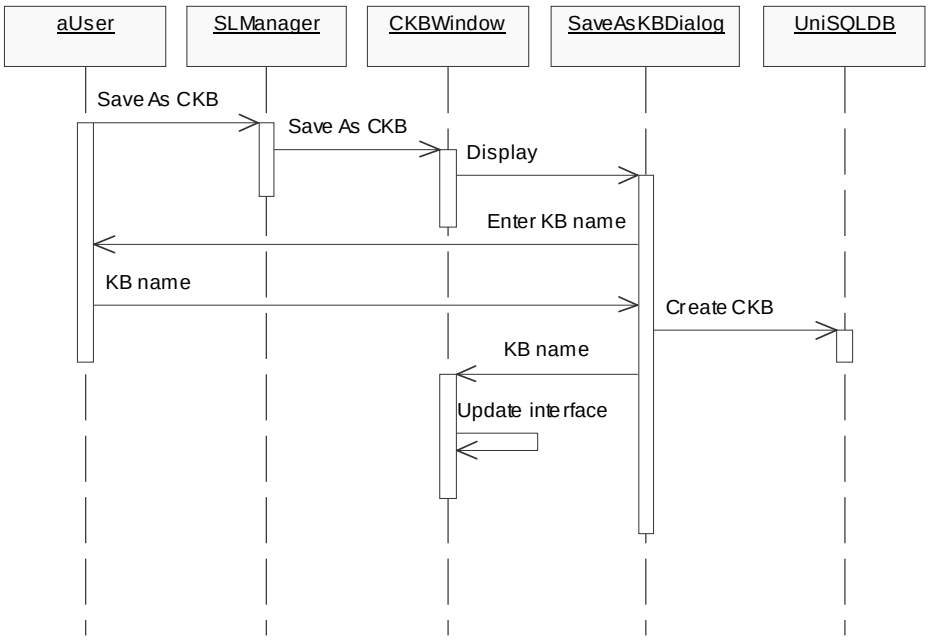
C.5 Save a CKB



Flow of Events:

1. The user selects the “Save CKB” option from the “File” menu.
2. SL_CKB saves the currently active CKB in the persistent database.
3. If changes have not been made after the last save, SL_CKB lets the user know the fact.

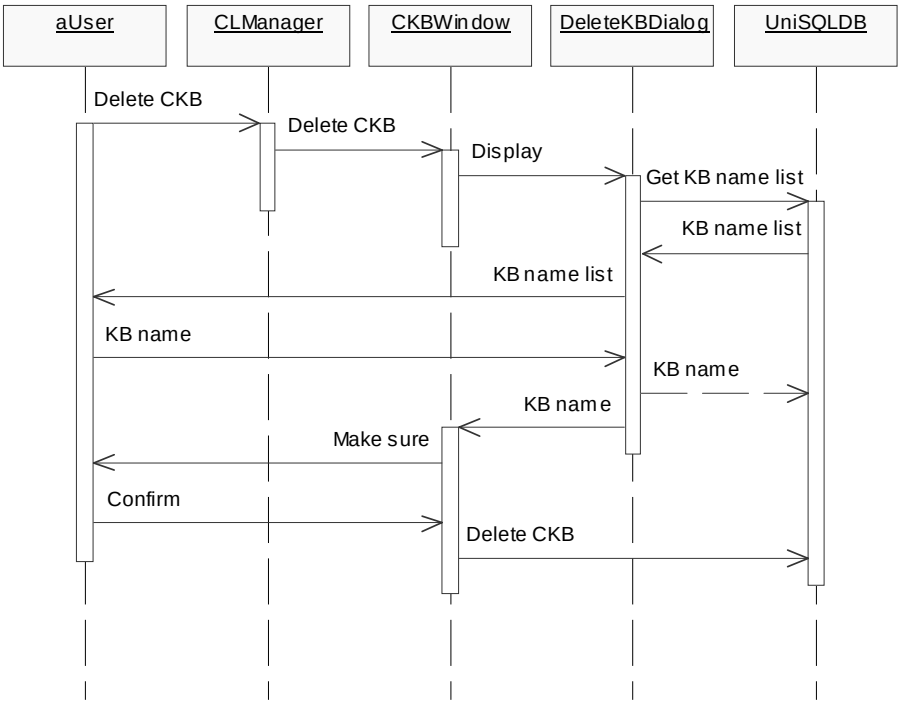
C.6 Save a CKB as



Flow of Events:

1. The user selects the “Save As CKB” option from the “File” menu.
2. SL_CKB opens the “Save As Knowledge Base” dialog box.
3. The user enters a desired name in the “New KB Name:” field and hits the “Commit” button.
4. SL_CKB saves the currently active CKB under the new name in the persistent database, provided a CKB with the same name does not exist in the database.

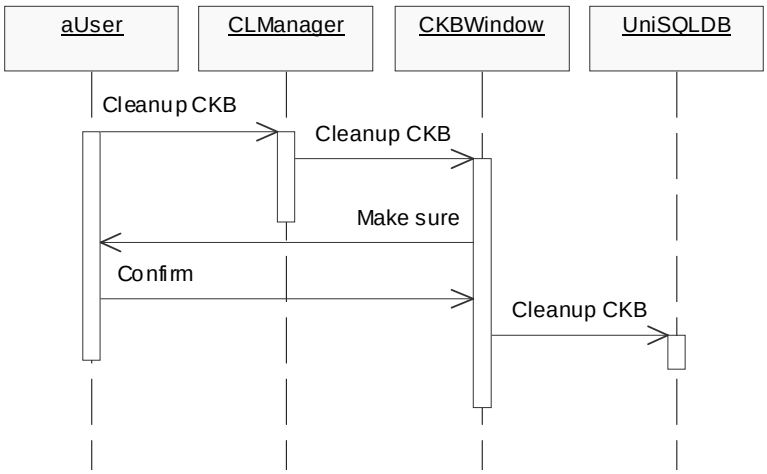
C.7 Delete a CKB



Flow of Events:

1. The user selects the “Delete CKB” option from the “File” menu.
2. SL_CKB opens the “Delete Knowledge Base” dialog box showing a list of existing CKB names in the connected database.
3. The user selects a name to be deleted and hits the “Delete” button.
4. SL_CKB prompts the user to confirm the deletion.
5. If the user confirms and the selected CKB is not currently active, SL_CKB deletes that CKB from the database along with all primitives, restrictions, DB objects, and classifications associated with it.

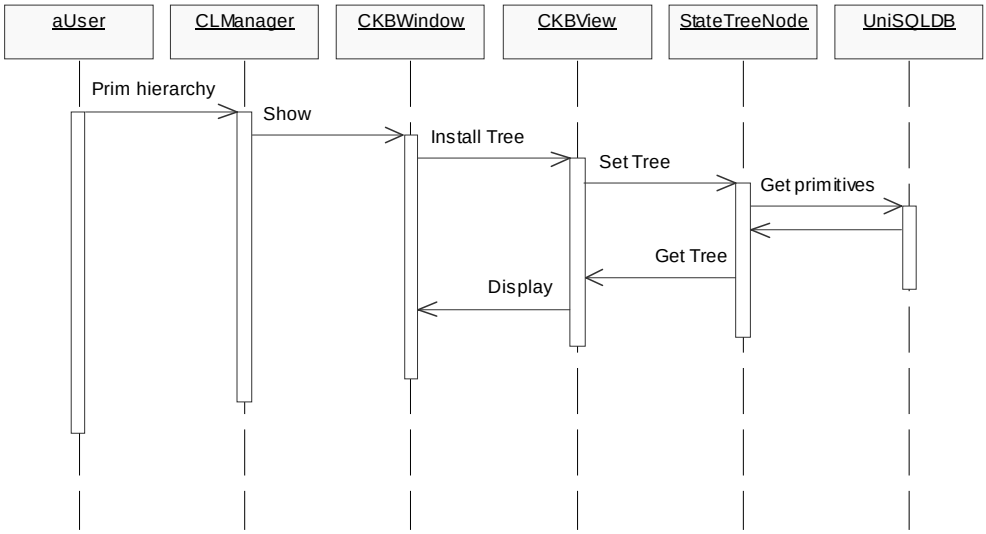
C.8 Cleanup a CKB



Flow of Events:

1. The user selects the “Cleanup CKB” option from the “File” menu.
2. SL_CKB asks the user to confirm the clean-up.
3. If the user confirms, SL_CKB discards the primitives, restrictions, DB objects, and classifications associated with the currently active CKB.

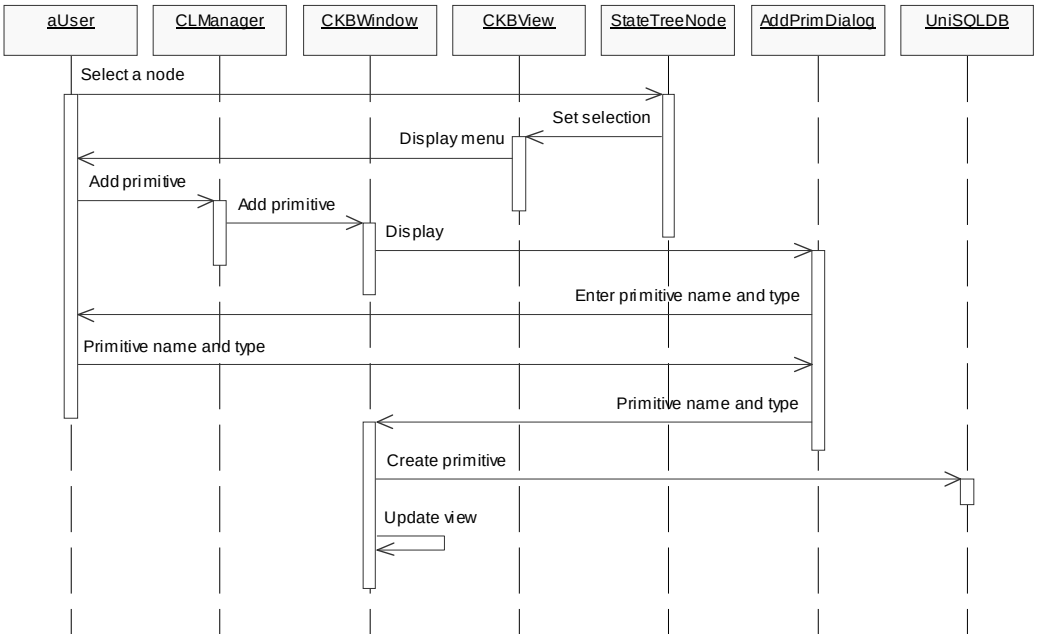
C.9 View primitive hierarchy



Flow of Events:

1. The user issues the “Primitive Hierarchy” command in the SL_CKB commands box.
2. SL_CKB displays a tree view of the primitive hierarchy in the currently active CKB.

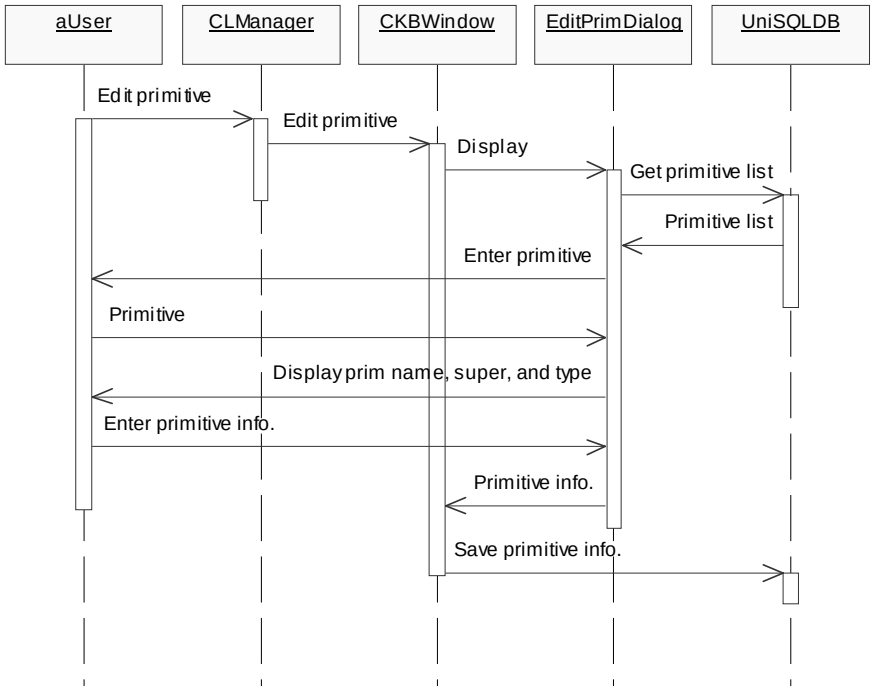
C.10 Create a primitive



Flow of Events:

1. The user selects a primitive from the tree view hierarchy with the left mouse button and clicks the right mouse button to display the pop-up menu.
2. The user selects “Add” option to create a new primitive.
3. SL_CKB opens the “New Primitive” dialog box with its super primitive.
4. The user enters a name in the “New Primitive Name:” field and chooses its type (simple or disjoint).
5. The user hits the “Commit” button.
6. If a primitive with the same name does not exist in the database, SL_CKB creates a new primitive with the given name in the persistent database and displays the new primitive in the correct position in an updated tree view.

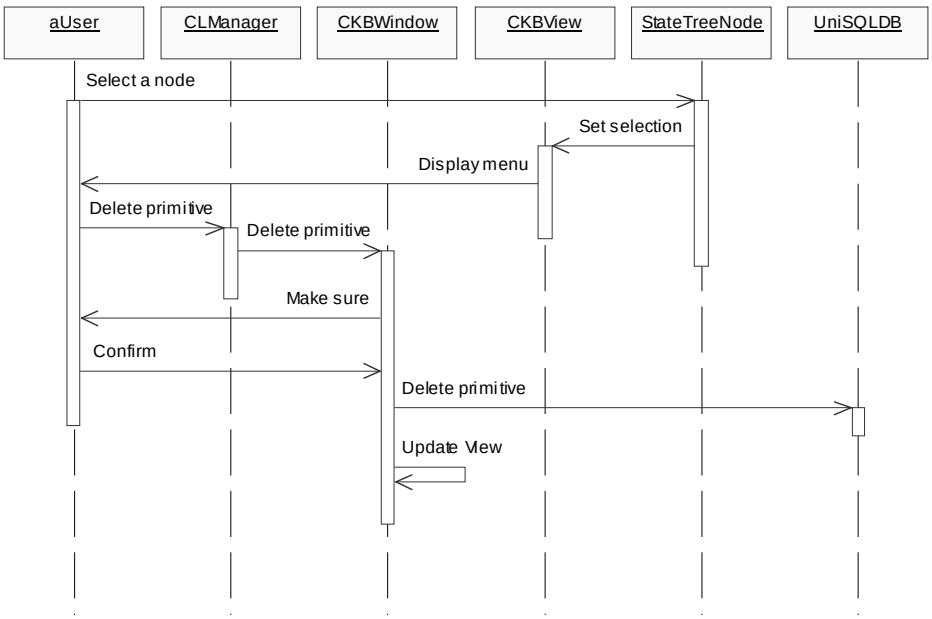
C.11 Edit a primitive



Flow of Events:

1. The user selects the “Edit Primitive” option from the “Primitive” menu.
2. The CKB opens the “Edit Primitive” settings box, which displays - in the “Primitive List” field - the names of all primitives currently defined in the active CKB.
3. The user selects a primitive in the list.
4. SL_CKB displays the name of itself in the “Primitive Name” field, the name of its super primitive in the “Select Super Primitive” button, and the primitive type in the “Select Type” button.
5. The user types the new primitive name, or changes the super primitive, or primitive type as desired.
6. The user hits the “Commit” button.
7. SL_CKB saves the changed attributes.

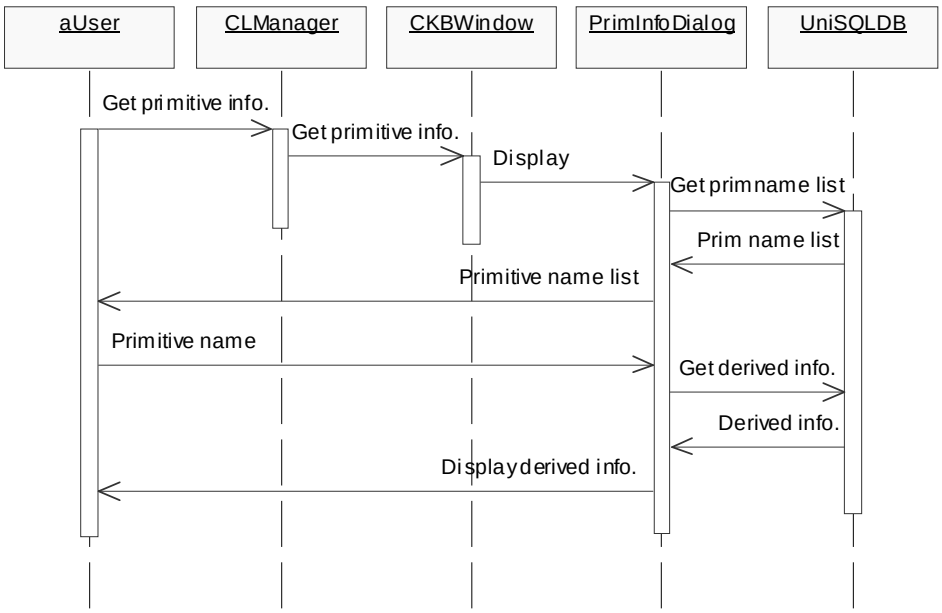
C.12 Delete a primitive



Flow of Events:

1. The user selects a primitive from the tree view hierarchy with the left mouse button and clicks the right mouse button to display the pop-up menu.
2. The user selects “Delete” option.
3. SL_CKB displays an Alert box asking the user if the primitive should be deleted.
4. If the user confirms, SL_CKB deletes the primitive and updates the display of the primitive hierarchy.

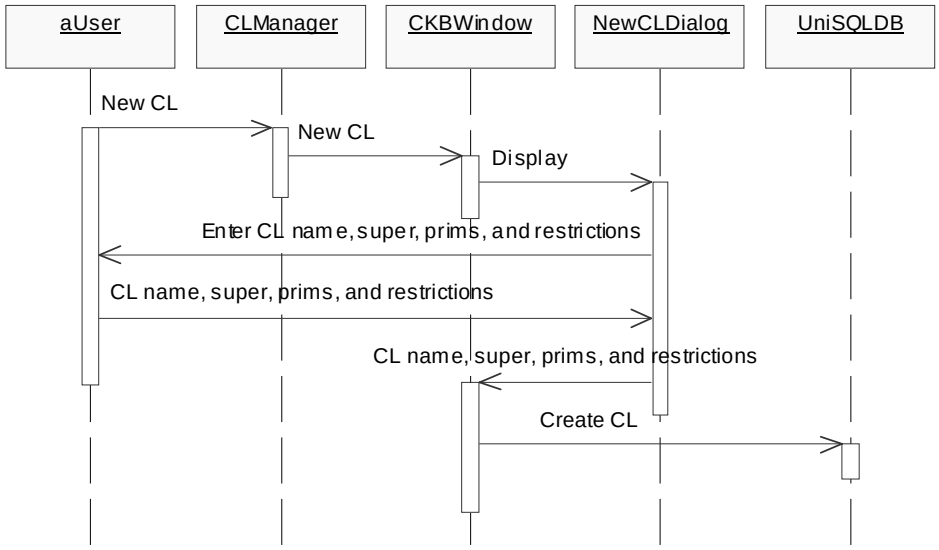
C.13 Get derived information



Flow of Events:

1. The user selects the “Get Primitive Info” option from the “Primitive” menu.
2. SL_CKB opens the “Primitive Information” settings box, which shows the names of the primitives currently defined in the “Primitive List” field in the active CKB.
3. The user selects a primitive in the list.
4. The settings box displays the “super primitive” and “sub primitives” list for the selected primitive.
5. The user inspects the current settings and closes the settings box.

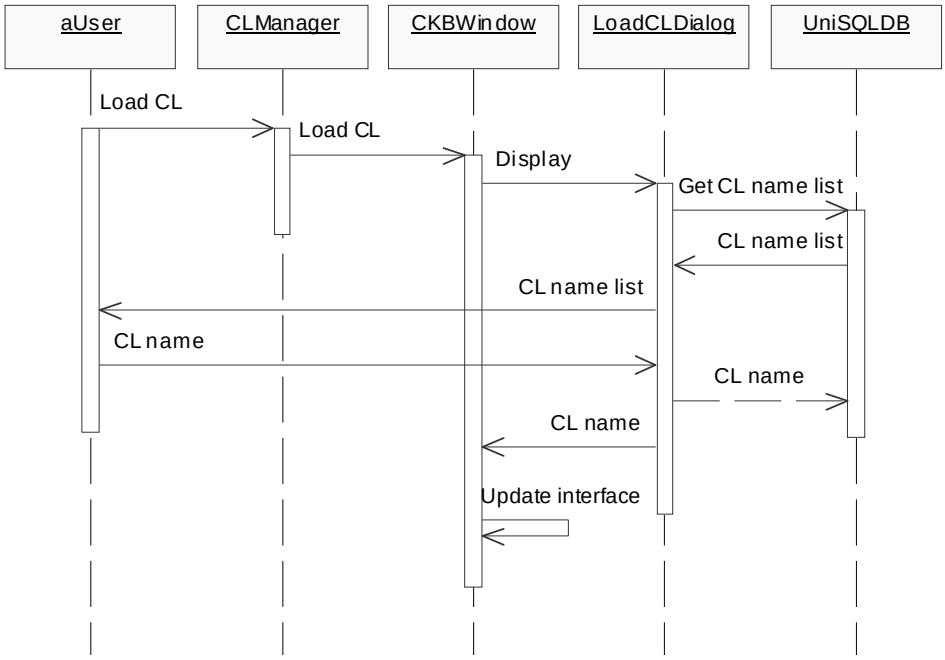
C.14 Create a classification



Flow of Events:

1. The user selects the “New Classification” option from the “Classification” menu in the CKB Window.
2. SL_CKB opens the “New Classification” settings box.
3. The user inputs a classification name in the “Classification Name:” field; chooses its super classification from the “Super Classification List:”; chooses the primitives from the “Primitive List:”; and selects a restriction from the “Restriction List:”.
4. The user hits the “Commit” button.
5. If no classification with the same name exists in the active CKB and the selected super classifications do not have inheritance conflicts each other, SL_CKB creates the new classification.

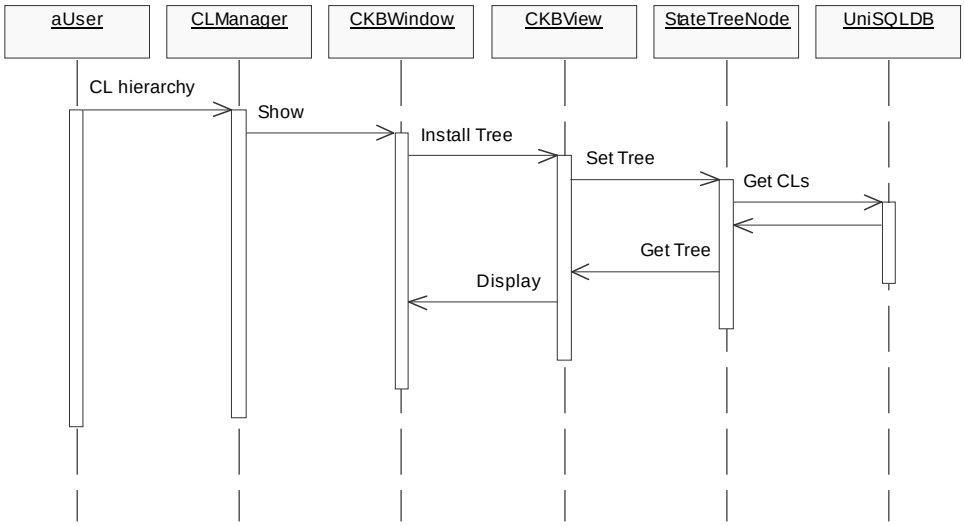
C.15 Load a classification



Flow of Events:

1. The user selects the “Load Classification” option from the “Classification” menu.
2. SL_CKB opens the “Load Classification” settings box which shows in the “Classification List:” field a list of the names of the classifications in the active CKB.
3. The user selects a classification name.
4. SL_CKB displays in the settings box the parent, primitives, and restrictions of the selected classification.
5. The user hits the “Commit” button.
6. SL_CKB retrieves the classification from the database and makes it the current active one.

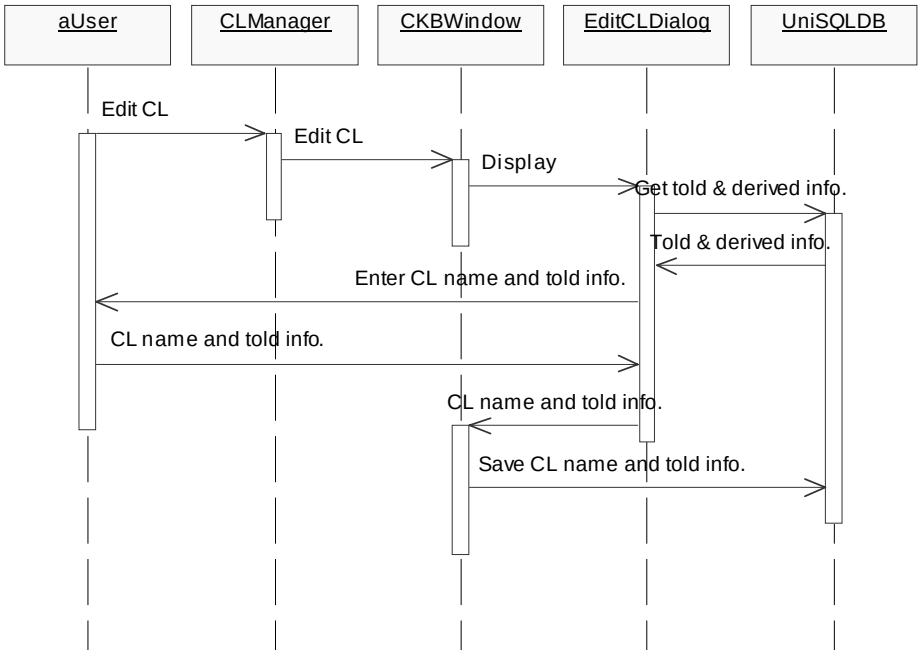
C.16 View classification hierarchy



Flow of Events:

1. The user issues the “Classification Hierarchy” command in the CKB commands box.
2. SL_CKB displays in the CKB Window a tree view of the classification hierarchy to which the currently active classification belongs.

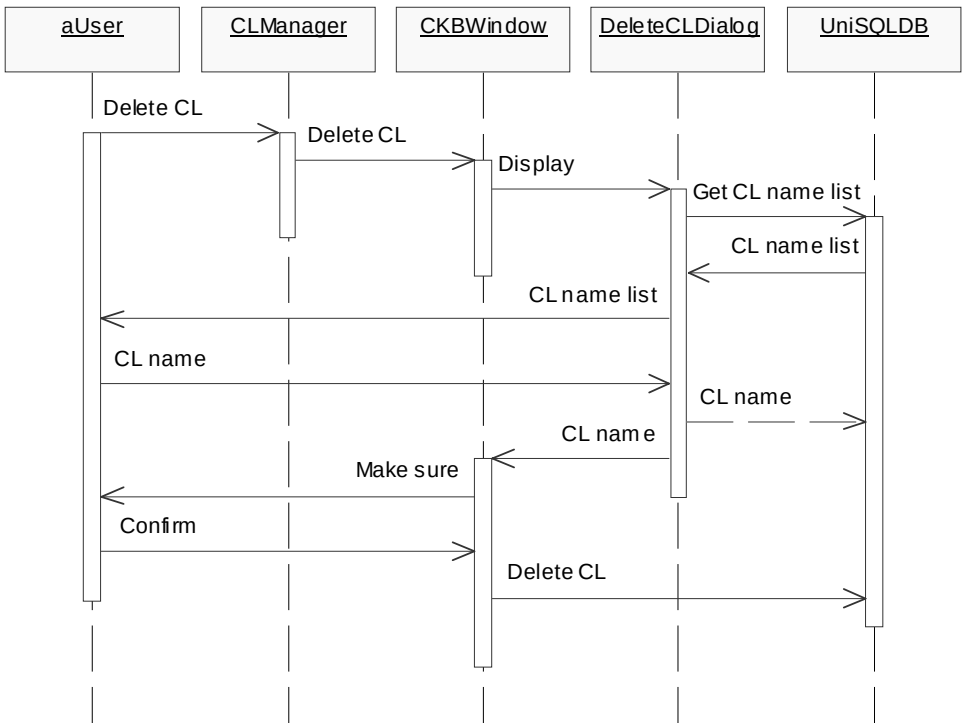
C.17 Edit a classification



Flow of Events:

1. The user selects the “Edit Classification” option from the “Classification” menu.
2. SL_CKB opens the “Edit Classification” settings box, which displays - in the “Classification Name:” field - the active classification name, the parent of the active classification in the “Super List:” field, the primitives it encompasses in the “Primitive List:” field, and the restrictions that apply in the “Restriction List:” field. It also displays its derived information such as its subsumers, subsumees, and synonyms.
3. The user types the new classification name, or changes the super classifications, primitives, or restrictions as desired.
4. The user hits the “Commit” button.
5. If no classification with the same name exists in the active CKB and the selected super classifications do not have inheritance conflicts each other, SL_CKB saves the changed settings.

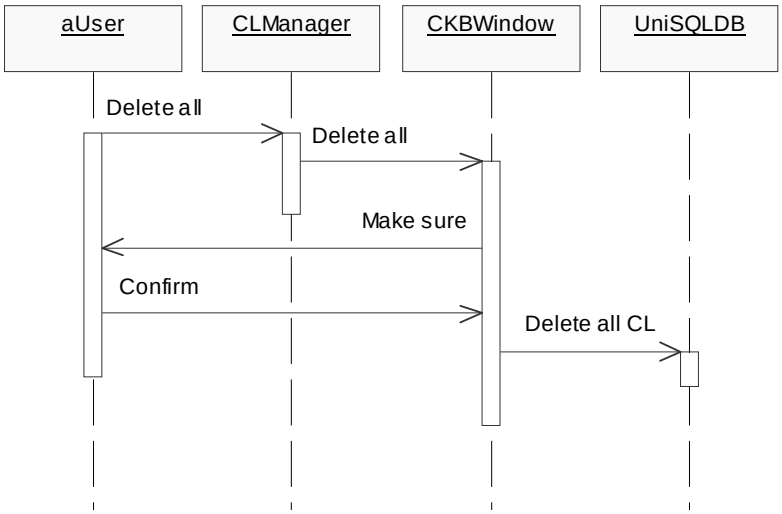
C.18 Delete a classification



Flow of Events:

1. The user selects the “Delete Classification” option from the “Classification” menu.
2. SL_CKB opens the “Delete Classification” dialog box, which shows - in the “Classification List:” field - the names of the classifications in the active CKB.
3. The user selects a classification name and hits the “Delete” button.
4. If the selected classification is not currently active, SL_CKB displays an Alert box prompting the user to confirm the request.
5. If the user confirms, SL_CKB deletes the classification and all classifications it subsumes along with all pairings between these classifications and the objects to which they are attached.

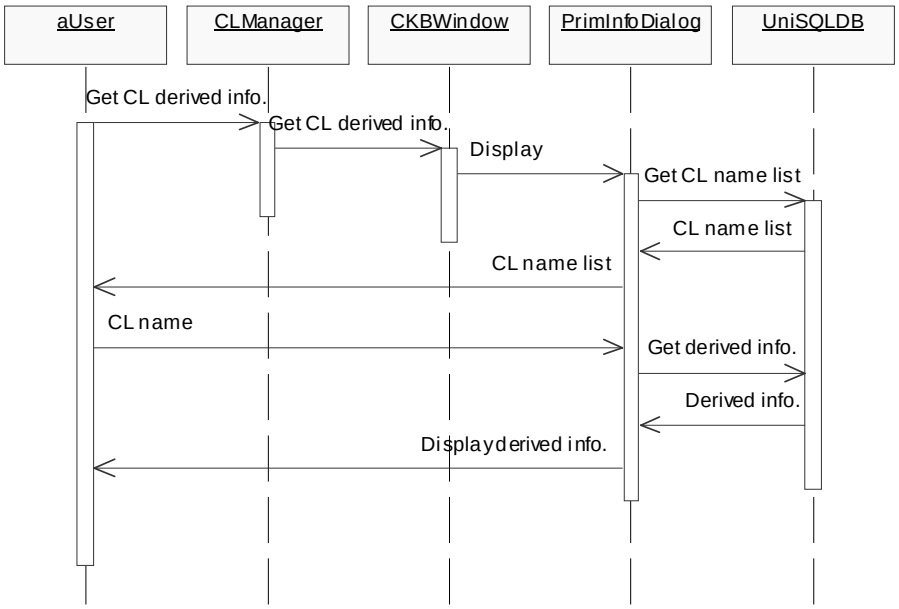
C.19 Delete all classifications



Flow of Events:

1. The user selects the “Delete All” option from the “Classification” menu.
2. SL_CKB opens an Alert box prompting the user to confirm the request.
3. If the user confirms, SL_CKB deletes all classifications along with all pairings between these classifications and the objects to which they are attached.

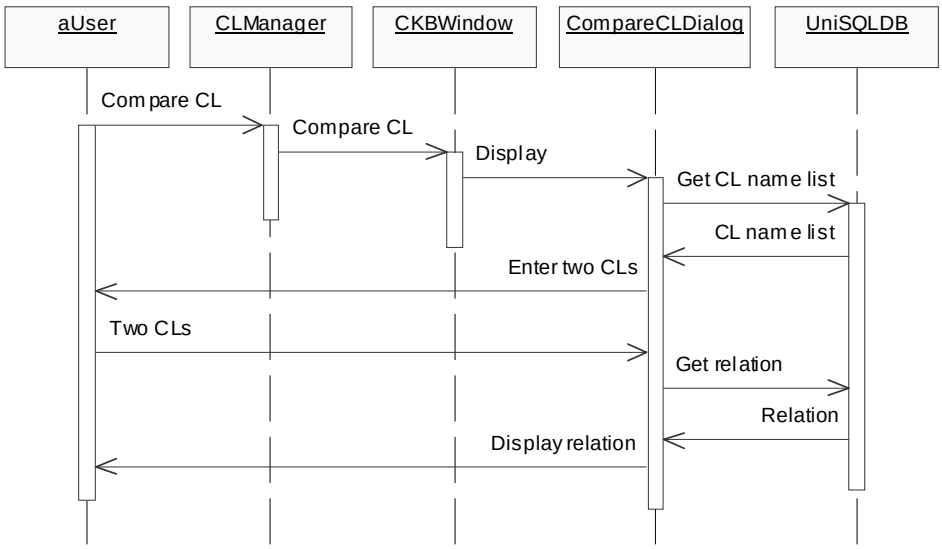
C.20 Get derived information



Flow of Events:

1. The user selects the “Get Derived Info” option from the “Classification” menu.
2. SL_CKB opens the “Derived Classification” settings box.
3. The user selects a classification in the list.
4. The settings box displays the “Derived Primitives”, “Derived Restrictions”, “Subsumers”, “Subsumees”, and “Synonyms” list for the selected classification.
5. The user inspects the current settings and closes the settings box.

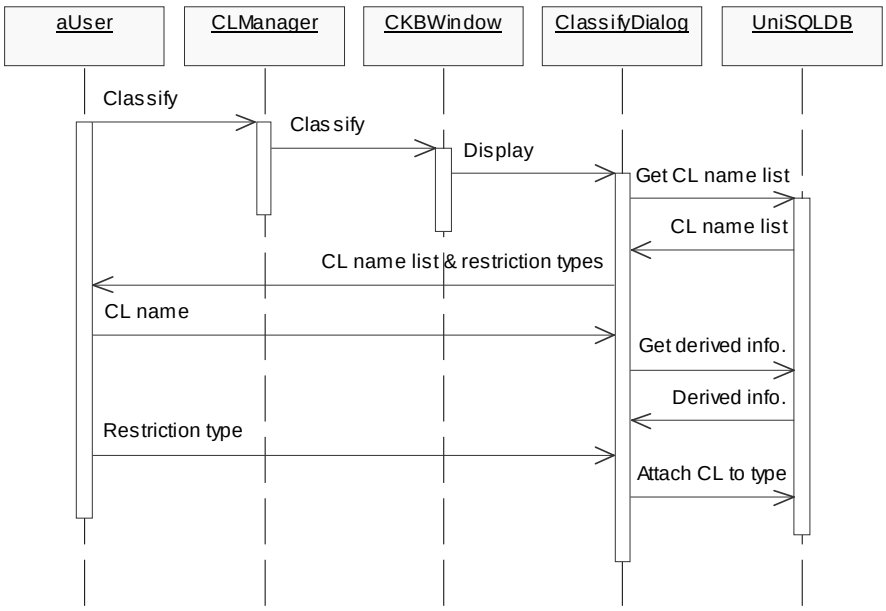
C.21 Compare classifications



Flow of Events:

1. The user selects the “Compare Classifications” option from the “Classification” menu.
2. SL_CKB opens the “Compare Classification” settings box, which shows a list of all classification names in the active CKB.
3. The user selects two classifications and hits the “Compare” button.
4. SL_CKB displays the attributes of the selected classifications such as SUBSUMER, SUBSUMEE, EQUIVALENT, EQUAL, DISJOINED, and DISTINCT.

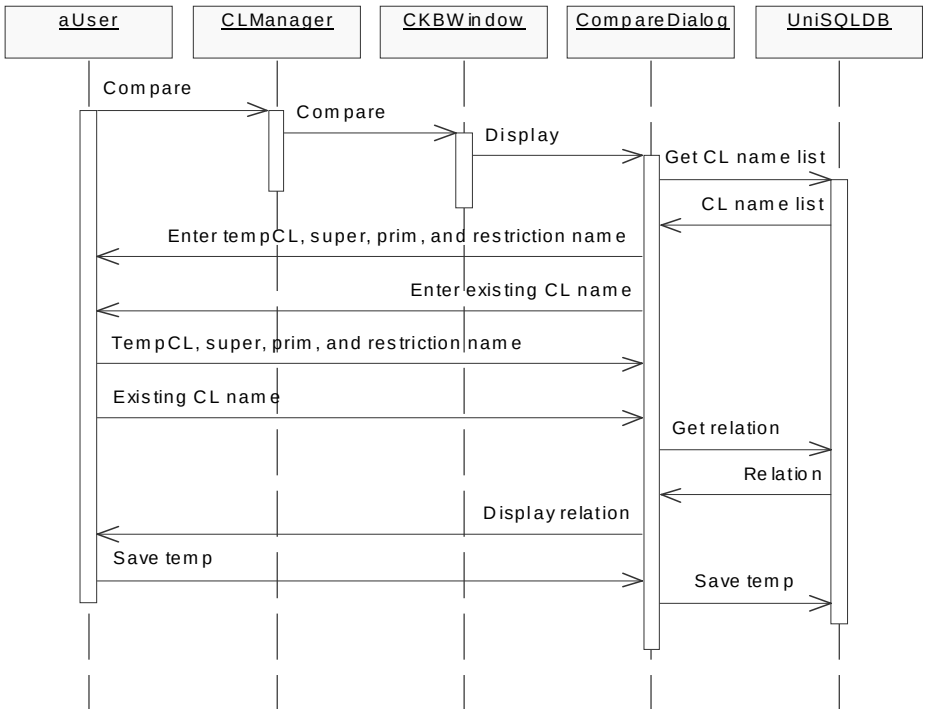
C.22 Attach a classification



Flow of Events:

1. The user selects the “Classify” option from the “Classification” menu.
2. SL_CKB opens the “Classify” settings box, which shows a lists of the names of all classifications in the active CKB.
3. The user selects a classification.
4. SL_CKB displays all of the attributes for the selected classification.
5. The user selects the type (or class) of object to which the selected classification should be attached and hits the “Classify” button.
6. Depending on the user’s selection, SL_CKB attaches the selected classification to Layout, LP, or FU.

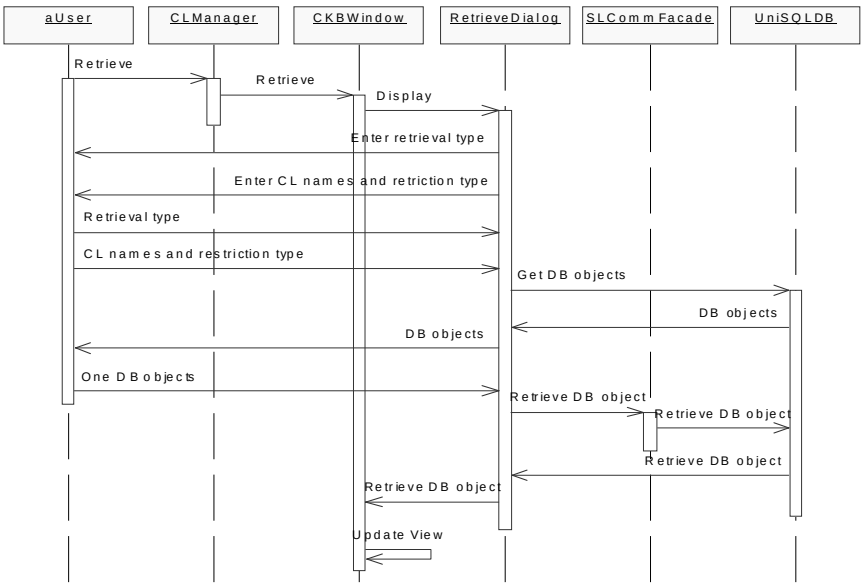
C.23 Compare



Flow of Events:

1. The user selects the “Compare” option from the “Retrieve” menu.
2. SL_CKB opens the “Compare” settings box.
3. The user inputs a temporary description name in the “Temp Description Name:” field and selects the super classifications from the “Super Classification List:” field, primitives from the “Primitive List:” field, and restrictions from the “Restriction List:” field; and selects the classification from the “Existing Classification List:” field.
4. If the user clicks the “Compare” button, the temporary description is compared with the selected classification.
5. The user can decide if the temporary description is saved or not. If the user decides the temporary description is saved, the user clicks the “SaveTemp” button to save the temp description.

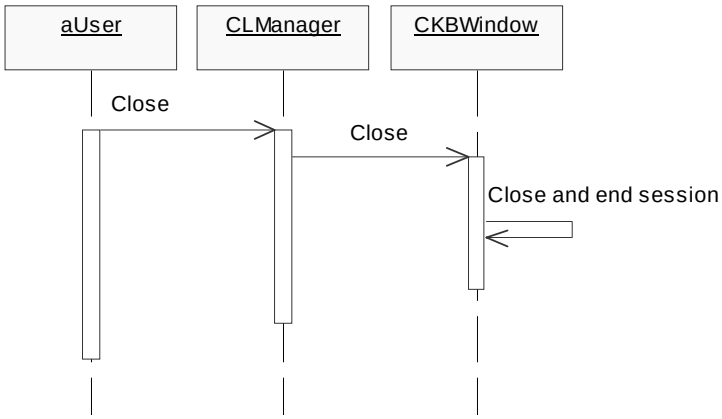
C.24 Get classified DB objects



Flow of Events:

1. The user selects the “Retrieve” option from the “Retrieve” menu.
2. SL_CKB opens the “Retrieve” settings box.
3. The user specifies interactively the retrieval type (SUBSUMER, SUBSUMEE, or EQUIVALENT) and selects one name.
4. The user selects a desired combination of classifications and the class of object he wants to retrieve (FU, Layout or LayoutProblem), and clicks the “GetDBObjectName” button. During this process, the classification engine checks the disjoined primitive conflicts among selected classifications.
5. The dialog box displays the names of all objects of that class with the specified classification and shows them in a selection field.
6. The user selects one name and clicks the Retrieve button.
7. The classification engine asks SLCommFacade to retrieve the selected object and load it into SL.
8. SL_Comm retrieves the object from the object database and loads it into SL; the specifics depend on the class of the object being loaded.

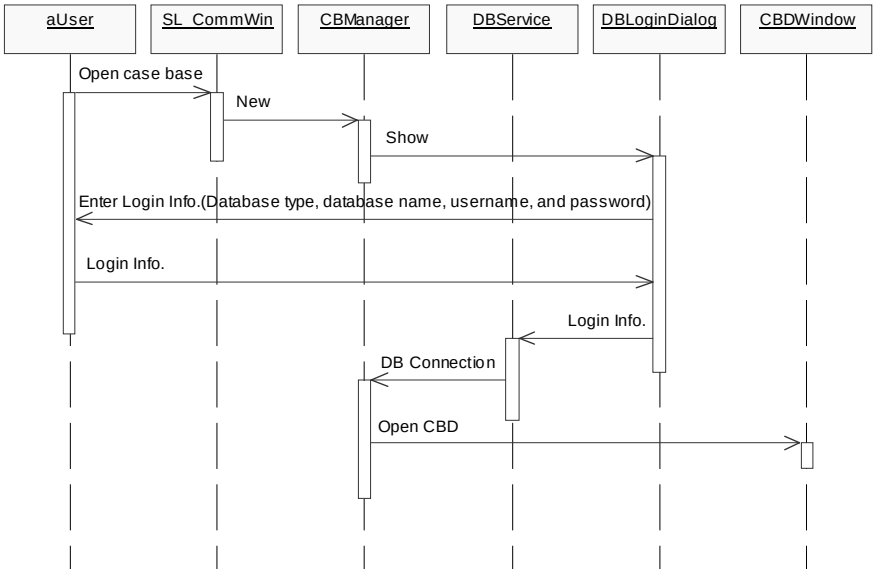
C.25 End a classification session



Flow of Events:

1. The user selects the “Close” option from the “File” menu.
2. If changes have been made after the last save, SL_CKB asks the user if the changes should be saved and acts accordingly.
3. SL_CKB closes the CKB Window and ends the classification session.

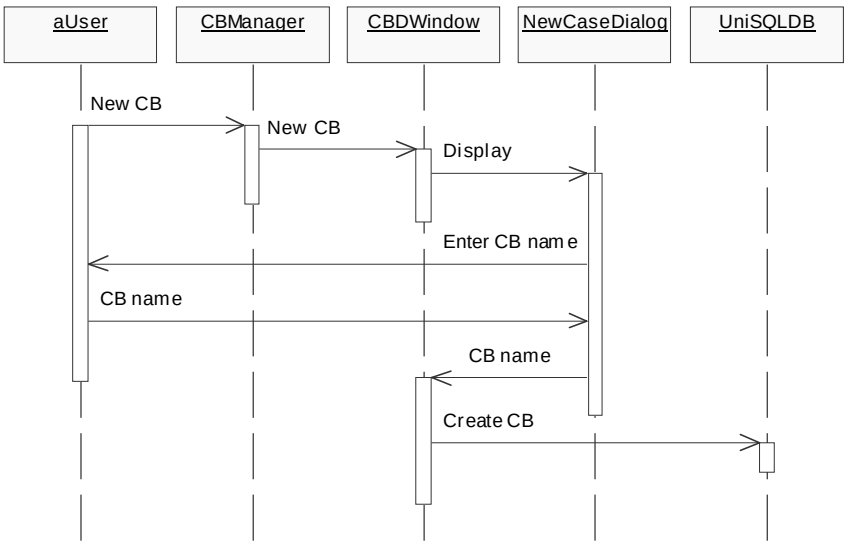
C.26 Start a case-base design session



Flow of Events:

1. The user selects the “Case base” option from the “Communication” menu in the SCW menu bar.
2. SL_CBD opens the “Database Login” dialog box.
3. SL_CBD prompts the user to select a database type and to input a database name, username and password.
4. If the three input values are all correct, SL_CBD connects to the selected case base and opens the Case-based Design (CBD) Window.

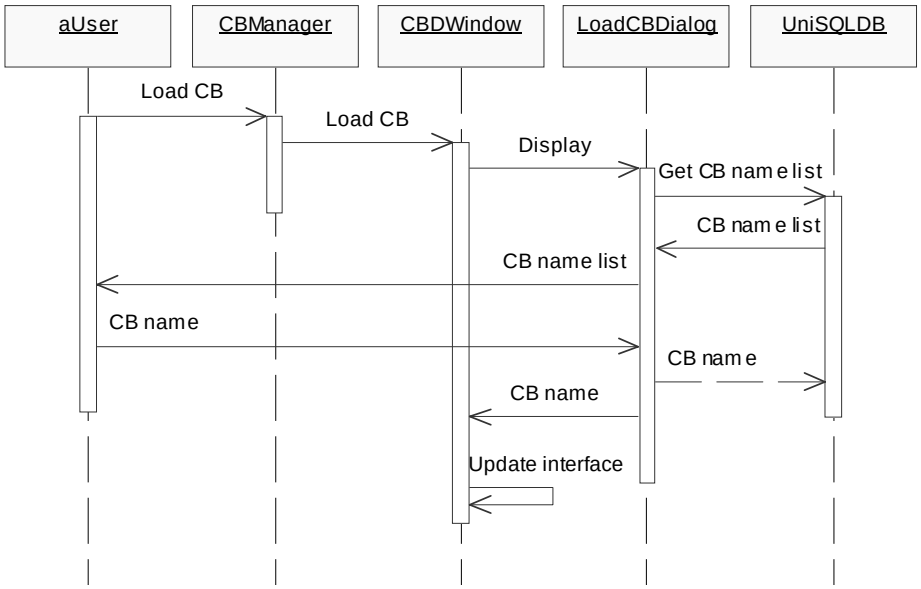
C.27 Create a CB



Flow of Events:

1. The user selects the “New CB” option from the “File” menu.
2. SL_CBD opens the “New Case-base” dialog box.
3. The user inputs a CB name in the “New CB Name:” field.
4. The user hits the “Commit” button.
5. If a CB with the same name does not exist in the database, SL_CBD creates a new CB in working memory.

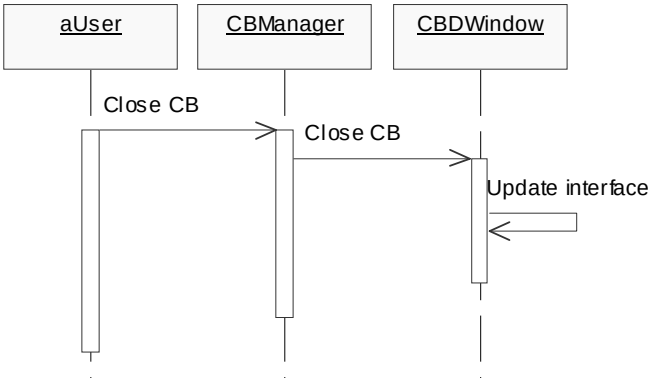
C.28 Load a CB



Flow of Events:

1. The user selects the “Load CB” option from the “File” menu.
2. SL_CBD opens the “Load Case-base” dialog box, which shows in the “Load Case Base” field the names of the CBs in the connected database.
3. The user selects a CB name and hits the “Commit” button.
4. SL_CBD retrieves the CB from the connected database and makes it the currently active one.

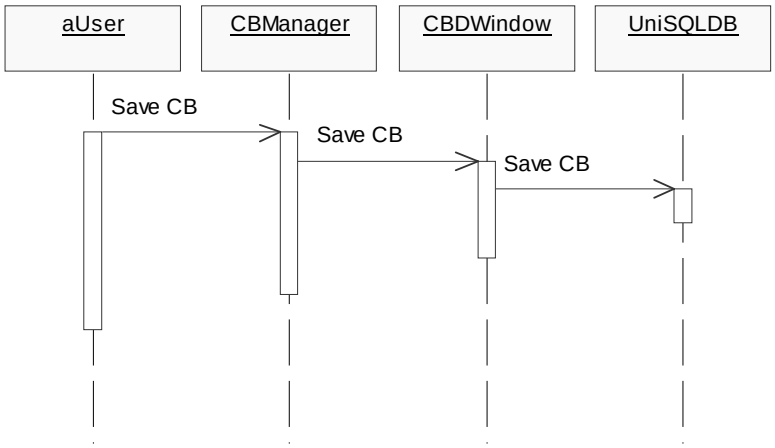
C.29 Close a CB



Flow of Events:

1. The user selects the “Close CB” option from the “File” menu in the CBD Window.
2. If changes have been made after the last save, SL_CBD asks the user if the changes should be saved and acts accordingly.
3. SL_CBD closes the currently active CB.

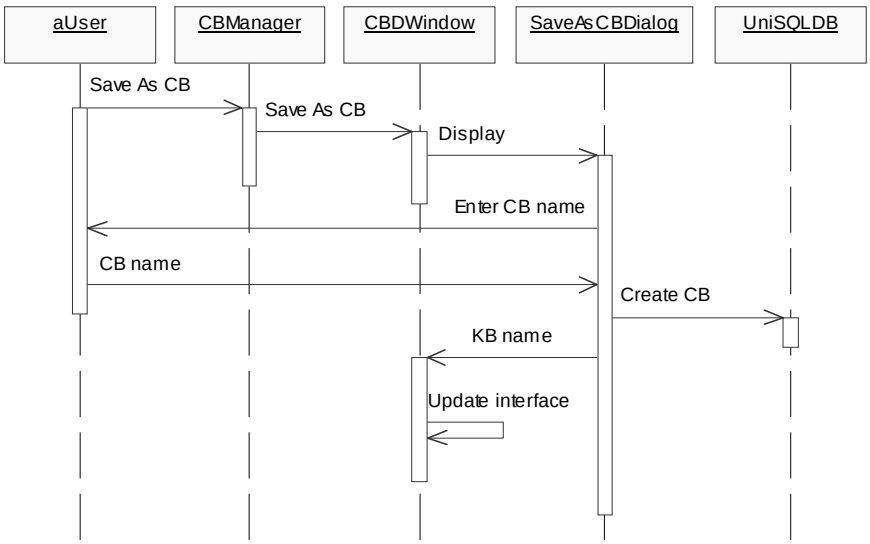
C.30 Save a CB



Flow of Events:

1. The user selects the “Save CB” option from the “File” menu.
2. SL_CBD saves the currently active CB in the connected database.
3. SL_CBD informs the user if no changes have been made after the last save.

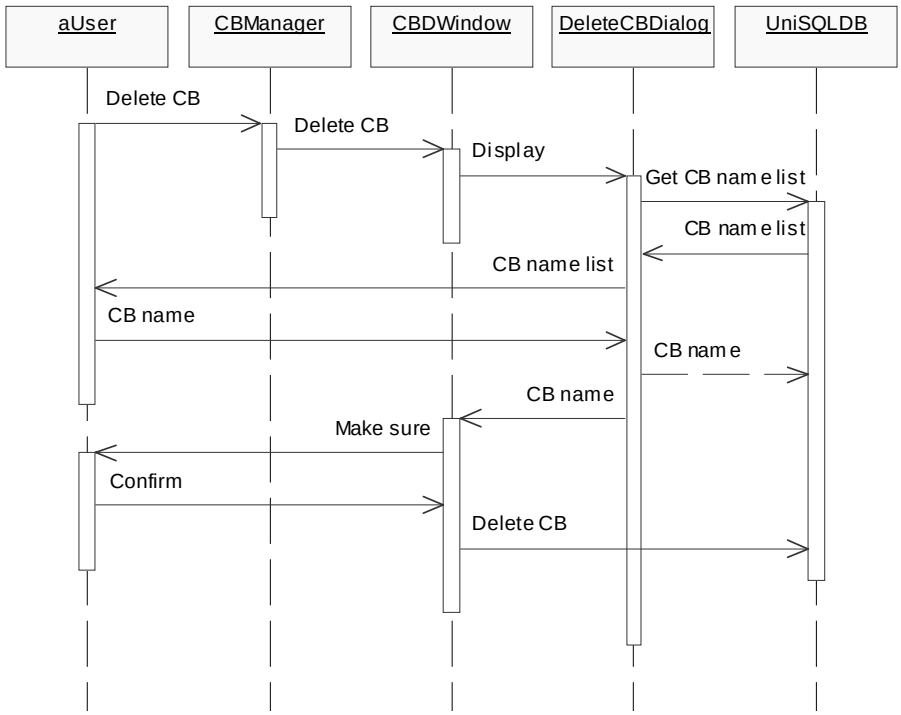
C.31 Save a CB as



Flow of Events:

1. The user selects the “Save CB As” option from “File” menu.
2. SL_CBD opens the “Save As Case-base” dialog box.
3. The user enters a name in the “New CB Name:” field and hits the “Save” button.
4. If a CB with the same name is not in the connected, SL_CBD saves the currently active CB under the new name in the connected database.

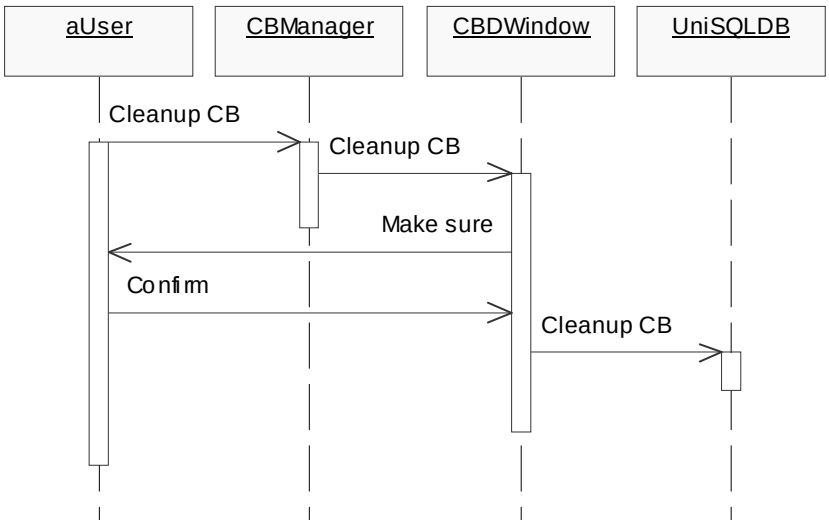
C.32 Delete a CB



Flow of Events:

1. The user selects the “Delete CB” option from the “File” menu.
2. SL_CBD opens the “Delete Case-base” dialog box, which shows a list of the CBs in the connected database.
3. The user selects a CB name and hits the “Delete” button.
4. If the selected CB is not the active one, SL_CBD opens an Alert box asking the user to confirm the request.
5. If the user hit the “OK” button in the Alert box, SL_CBD deletes the selected CB in the connected database along with all the associated CB components. This deletion is immediately persistent.

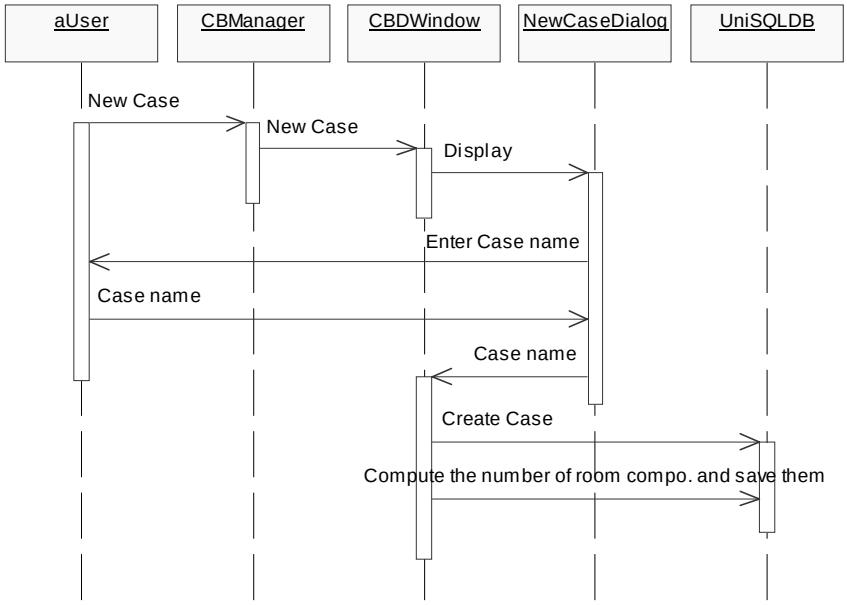
C.33 Cleanup a CB



Flow of Events:

1. The user selects the “Cleanup CB” option from the “File” menu.
2. SL_CBD opens an Alert box asking the user to confirm the request.
3. If the user hit the “OK” button in the Alert box, SL_CBD cleans up the active CB.

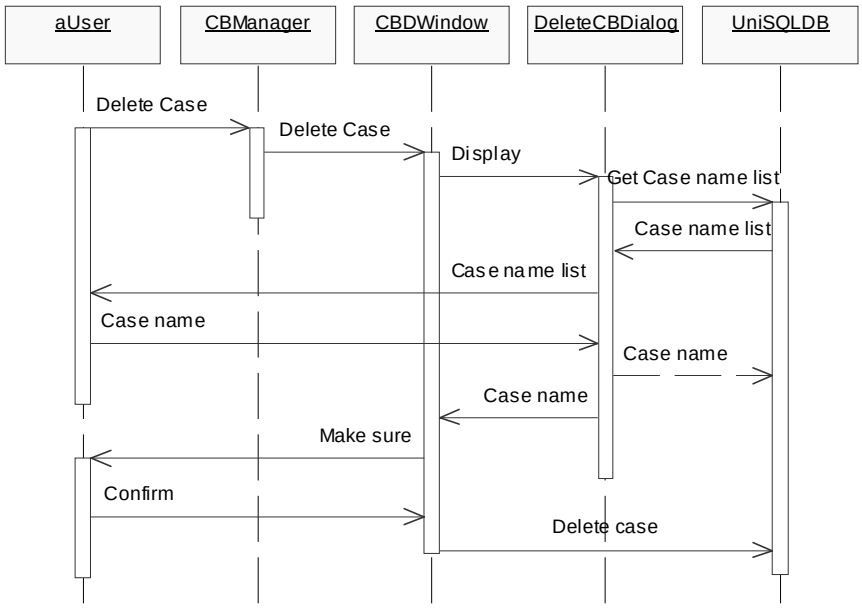
C.34 Create a Case



Flow of Events:

1. The user selects the “Create a Case” option from the “Case” menu.
2. SL_CBD opens the “New Case” settings box.
3. The user enters a name in the “Case Name:” field.
4. SL_CBD displays the descriptors attached to the active Layout.
5. The user hits the “Save” button.
6. If no case with the new name exists in the active CB, SL_CBD creates a new case. When SL_CBD saves a case, it computes in the background the number of significant room components it contains and saves them as additional case attributes.

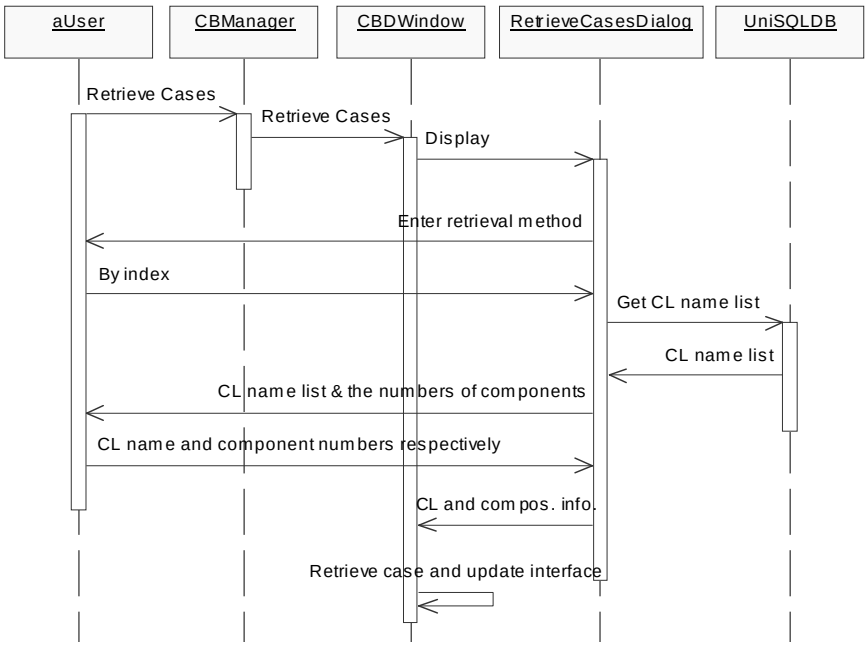
C.35 Delete a Case



Flow of Events:

1. The user selects the “Delete a Case” option from the “Case” menu.
2. SL_CBD opens the “Delete Case” dialog box, with displays - in the “Case Name List” field - the names of all cases currently defined in the active CB.
3. The user selects a case name in the list and hits the “Delete” button.
4. SL_CBD displays an Alert box asking the user if the case should be deleted.
5. If the user confirms, SL_CBD deletes the case.

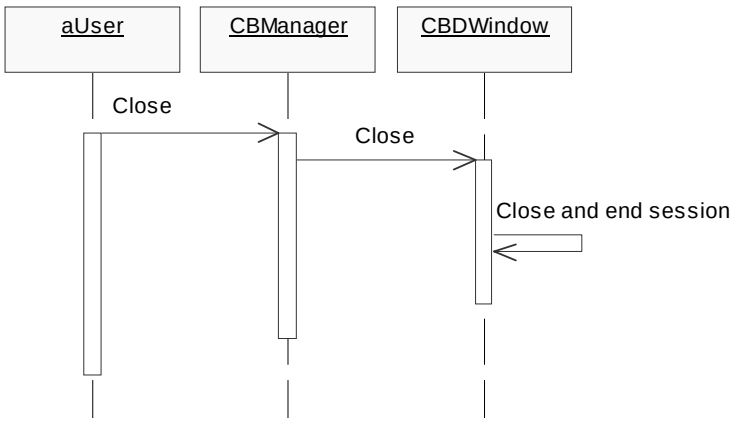
C.36 Retrieve a Case



Flow of Events:

1. The user selects the “Retrieve Cases” option from the “Case” menu.
2. SL_CBD opens the “Retrieve Cases” settings box.
3. The user selects the retrieval method by checking the “By Name” or “By Index” check box.
4. If the user checked the “By Index” box, SL_CBD displays the names of all classifications associated with cases in the active CB and the component attributes.
5. The user selects a classification, enters numbers in selected component attribute fields and hits the “Retrieve” button.
6. If cases with this combination of indices exist in the active CB, SL_CBD displays the names of these cases.
7. The user selects a case name and hits the “Retrieve” button.
8. If the object representing the case exists in the object database, SL_CBD retrieves the case. The retrieved case is added to the current design space and becomes the active Layout.

C.37 End a case-based design session



Flow of Events:

1. The user selects the “Close” option from the “File” menu.
2. If changes have been made after the last save, SL_CBD asks the user if the changes should be saved and acts accordingly.
3. SL_CBD closes the CBD Window and ends the case-based design session.