Computational Modeling of Design Requirements for Buildings

Magd A. Donia

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

Advisory Committee

Ömer Akin [Chair]
Professor
School of Architecture
Carnegie Mellon University

Steven Fenves
Professor
Department of Civil and Environmental Engineering and
Institute for Complex Engineered Systems (ICES)
Carnegie Mellon University

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

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

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

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

_____

Magd Donia

Abstract

During the past few years, several computational design support and simulation tools for building design have emerged, from research as well as industrial institutions. Many of these tools provide design generation and evaluation mechanisms which assist building designers to rapidly create and evaluate design alternatives. These tools normally require a relatively large input of design requirements information, from which design representations are created and evaluated. The usability of design support and evaluation systems has been adversely affected by the lack of computable representations of design requirements. Such representations can provide a repository from which the input information needed by design support and simulation systems can be generated.

This thesis provides a description for a computable model of building design requirements that supports:

- the diverse nature of information typically associated with the architectural programming stage, and
- deriving specialized representations needed by other design support and performance simulation systems.

The approach used to achieve these research goals is a framework that contains organizing concepts and software building blocks from which representations of design requirements can be built.The research builds on and augments the work accomplished in SEED-Pro (the architectural programming module of SEED) by re-engineering its information model. It provides contributions in the areas of architectural programming and design requirements modeling as well.

Contents

Figures

Tables

Acknowledgements

First, I would like to express my deepest love and gratitude for my parents. Their support, guidance and advice have been the principal factors in getting me thus far. They are what I treasure and care about the most in this life.

I've been very privileged to work with all three members of my advisory committee. It was a very enlightening, and humbling experience at the same time. Their comprehensive knowledge, intelligence, dedication and personal integrity helped me set a higher standard for myself on academic, professional and personal levels. Prof. Akin provided me with all the time and effort I needed. He often presented me with his extremely insightful observations and challenges that helped me during all stages of my research. Prof. Fenves has provided a lot of valuable advice which have shaped my research objectives. I've been very fortunate to work with him. Prof. Flemming's uncompromising, and relentless drive for excellence provided me with an example to follow in pursuing my professional and personal objectives. I've enjoyed his company watching and discussing World Cup and European championship soccer games. I've also enjoyed playing for his intramural tennis team at CMU, despite me being the reason for losing a couple of close matches which could have won us the tournament!

I've been very fortunate to be with a very talented group of students during my stay at CMU. Jim Snyder was my colleague and first roommate when I moved to Pittsburgh. He is now one of my dearest friends who have helped me in more ways than I could ever repay him. I would also like to thank my other roommates and dear friends Safwan Ali, Georg Suter and Hesham Eissa. They made my stay in Pittsburgh and CMU as pleasant as I had ever hoped. I'll miss the grueling tennis and squash games with Georg, Safwan and Robert Ries, and the interesting and funny experiences I had with all my CMU friends (two words: *whitewater rafting!*).

Working in the SEED project has been a tremendous learning experience for me. I've learned so much from all project members, and would like to specially mention Rana Sen (my "SEED-Bro"), Sheng-Fen (Nik) Chien (one of the brightest people I've ever met), Weng-Jaw (Jonah) Tsai (resourceful computing dude), Zeyno Aygen (the Classifier!), Michael Cumming and Ye Zhang.

Throughout my life, I've been blessed with many great friends. My college friend Adham El-Sharqawi has been a true brother for me during the past 10 years. The tremendous moral support he has been providing for my parents during my long journey away from home was just what I expected from him. I'm truly honored to have him as a friend.

CHAPTER 1

Background and Motivation

This chapter provides background information in the history and evolution of the field of architectural programming and the motivation behind the research described in this document.

1.1 The Evolution of Architectural Programming

1.1.1 A Brief History

Throughout the course of history, it has been the case that buildings were conceived only after some need had been formulated, which was then followed by instructions given, in some form, to the architect and builder. These instructions were usually brief and mainly functional [Kumlin 95], and it was left to the architect's or the builder's discretion to add the necessary amount of creativity and detail that satisfied the other non-functional types of requirements, as well as add new requirements and explicate implicit ones. This process used to work sufficiently, as building types were relatively simple, until the industrial revolution at the beginning of the nineteenth century brought in a proliferation of specialized building types. This created the need for elaborate and specific architectural programs. Such programs became common towards the end of the nineteenth century. Some architectural design competitions featured these types of programs and resulted in notable and successful buildings, such as the Paris Opera House by Garnier (1898-74) and the Amsterdam Stock Exchange by H. P. Berlage (1898-1903). However, these programs were created and provided by the owners with very little or no input from the architect.

This remained the case, in the United States at least, until the early 1960s when the American Institute of Architects published a booklet titled *Emerging Techniques of Architectural Practices* [AIA 66] which was followed by a sequel in 1969 called *Emerging Techniques 2: Architectural Programming*; by the book *Problem Seeking: An Architectural Programming Primer* [Pena et al. 87]; and by *Methods of Architectural Programming* [Sanoff 74]. These publications emphasized the need for architects to assume the responsibility

of developing architectural programs for the buildings they were commissioned to design.

1.1.2 Evolution as a Separate Discipline

However, these publications differed with respect to the amount of programming the architect needs to handle. The AIA booklet stated that the client was to provide the program from which "the architect then can develop his [sic!] program from which the designs are produced". On the other hand, *Problem Seeking* provided an approach to programming that formed a complete system. This approach developed into a successful and widely emulated methodology which became the flagship of some notable AEC practices. An example is HOK[1], which purchased the professional service business of CRSS, the authors of *Problem Seeking*.

The architectural programming method described in that book consists of five main steps:

1. Establishing project goals which are mainly created by the client organization to ensure its involvement in the project development.
2. Collecting, organizing and analyzing information that influences the design of a project. This includes specific project requirements, such as budgets and space requirements. Information pertaining to site considerations, utility services and existing codes and standards that have to be met is also collected and analyzed at this stage. Goals established in the previous step help determine the type of information needed.
3. Experimenting with different concepts and ideas that can realize project goals. During this stage, concepts are created, then tested against requirements and goals. Many iterations of redefining requirements and verifying concepts can take place at this stage. Concepts developed here are not meant to provide the designer with physical solutions as input. Instead they should be programmatic concepts expressing material flow, functional relationships, flexibility, operations, and levels of safety that constitute the conceptual segment of the program.
4. Determining needs in terms of quantifiable terms, such as building area, quality of construction, equipment that will be used in the building, and construction cost and time.
5. Developing the problem statement jointly by the programming and the design teams. This statement serves as the premise for design and can be used to evaluate the solution. It also constitutes an interface between programming and design.

These studies and methodologies caused architectural programming (AP) to evolve into the process by which the design requirements for buildings are compiled and formulated. There are many definitions of architectural programming. Duerk defines it as "the process of managing information so that the right kind of information is available at the

1. Hellmut, Obata and Kassabaum, St. Louis Missouri.

right stage of the design process and the best possible decisions can be made in shaping the outcome of the building design." [Duerk 93].

1.2 Problem Definition and Motivation

Architectural programming is generally concerned with two main categories of information [Duerk 93]. The first is information describing the existing state within which the design is to be embedded and includes such things as site analysis, client profiles, codes, climate and all types of existing constraints. The second is the information describing what the future state should be, which is the set of criteria, stated at different levels of granularity, that the project should meet in order to be successful.

A study of current AP practices and supporting computational tools [Chapter 3] indicates that architectural programming is perceived as an important step for formulating design requirements that should be achieved in the finished design. However, several problems limit the usability of architectural programs in practice and, consequently, affect the effort invested in the programs. These problems mainly result from the format in which these architectural programs are developed: a text document that contains a description of the goals that the project should achieve; the way the project is structured into existing and future elements (mainly spatial); and the specifications associated with these elements. The descriptions of the project components and specifications are often entered into a spreadsheet to provide a more structured view of the program and to assist in its quantitative aspects, such as area and material calculations.

Such a format requires that the connection between the goals, expressed as text, and the spatial elements and their specifications in the spreadsheet that realize these goals, be maintained manually. It also provides no support for any type of non-numerical reasoning, reuse of existing architectural programs (either as a whole or parts of them) or information transfer to other design systems because there is no underlying conceptual representation to enable such functionalities. Such a representation cannot be practically created for every architectural programming effort due to the complexities involved in doing so. However, a general framework for creating and developing architectural programs offer the functionalities mentioned earlier, and if provided with a suitable interface, could make the AP task easier and more manageable.

In addition, such a framework can serve as a front-end for design systems, supplying them with the information they need to accomplish their tasks. It is well known that the most time-consuming and difficult task in using most design and evaluation system, such as simulation tools, is creating the building model for each of these systems—a task that usually has to be repeated for each design and simulation tool used. This shortcoming can be overcome by extracting the information these systems need from a general building specifications framework and "packaging" it in the format required by these systems. For example, an HVAC design software can receive its input of air temperatures, levels of humidity, and types of occupancy from the set of climatic requirements defined in such a framework.

The problem addressed in this thesis, then is to create a representation of architectural programming information which supports modeling of design requirements as well as generating customized requirements for design systems that collaborate on the process of producing and evaluating a building design.

1.3 Research Objective

The main objective behind this research is to provide the means to achieve an AP development environment that is capable of representing the wide range of information associated with an architectural program. This environment is envisioned to be computationally-based to make use of existing technologies in information modeling and management and to provide high-fidelity communication and data translation to other design systems. This environment can be generally characterized as follows:

1. Providing means of storing and handling *all* aspects of the AP information including site characteristics, codes, client preferences, and different performance criteria and requirements.
2. Ensuring the use of criteria established during the programming phase as a basis of design.
3. Integrating architectural programming and architectural design as a seamless process.
4. Maintaining a database of reasons employed in making programming decisions, and improving the computability of AP information by allowing non-numerical types of reasoning not currently supported by the existing medium of representation (text and spreadsheets).
5. Achieving a flexible way of interaction that does not tie the designer to a particular model of AP.
6. Enabling the use of past programs and programmatic experience in future projects.

Providing means of storing and handling all aspects of the AP information will transform the architectural program from a text document with a short life cycle and limited use to a growing reservoir of information about the project, which reduces information loss and increases architectural program usability. It is important to allow for all types of information to be handled by the computer-aided architectural programming (CAAP) tool, even those that are, presently, in a non-computable form, such as esthetics, corporate missions, and management styles. Such information can still be stored as plain text, just as it would be in a program text document. Its inclusion in the CAAP environment would make that environment a self-sufficient source for addressing these issues in later stages of design, eliminating the need for a companion text document, as is the case with existing tools.

Ensuring that the criteria established during the programming phase are used as a basis of design will enable the designer to find design decisions that do not meet criteria specified in the architectural program, as they will be flagged by the system. It will also make the criteria specified in the program more reliable and realistic because designers

will have to modify them if they cannot be met, rather than just violating these criteria without making that violation evident to the programmer / designer. In that sense it will make the program a more reliable source of project information.

The integration of architectural programming and architectural design is needed to realize the previous characteristic. By integrating the CAAP tool with other CAAD systems capable of supporting the other phases of the design process, AP would become an integral part of that process, and could help support its iterative nature in a seamless and usable manner.

Maintaining a database of reasons employed in making programming decisions provides the basis for employing functional reasoning in design. As the CAAP tool is being used, a database of logic-decision pairs can be formed. There might even be a way for organizing that database in a computable form that permits applying inferencing mechanisms to deduce possible decisions based on the reasoning used. The practical implication of this is still not clear due to reliability concerns of decisions obtained in that manner. However, such a database of reasons is still useful in determining the reasoning behind decisions made during the course of developing a single project, even if it is not reliably transferable to other projects.

Achieving a flexible way of interaction would make CAAP accessible to a wide variety of practitioners who regard the AP process in different ways. It will also make the tool useful for design support activities, such as feasibility studies, and to non-architects, such as real estate developers and financiers.

Finally, the use of past programs in future projects, which capitalizes on some of the previous characteristics, would overcome a huge hurdle that prevents the use of past programs in future projects. This can drastically reduce the time and effort spent in developing programs; it would also improve their quality as it would tend to accumulate knowledge, progressively refining past experiences.

1.4 Research Objective Software Requirements

To achieve these characteristics, certain software functionalities are needed. They can be generally summarized as follows:

Representation of programing information: This calls for a software framework to represent the range of wide and diverse information associated with architectural programming. The Specification Unit domain [Akin et. al., 95], is an example of such a framework. The ongoing design of that framework has been a difficult, challenging but also an exciting task, due to complexity of the information involved as well as the need to accommodate other design tools to provide the integration goal stated in the previous

section[2]. To achieve such a framework the following two software requirements must be fulfilled.

Design flexibility using OOP techniques: Flexibility is a key to a successful design of a framework that supports the representation of architectural programming information. The reason is that this framework is expected to be in a state of flux for some time before it reaches a state of relative stability. Until it reaches that stage, it should still be usable. It also should be able to accommodate the needs of existing, as well as future, CAAD tools that support the rest of the design process. OOP techniques provide useful mechanisms to simplify system design and maintenance, such as *inheritance* and *aggregation*. Recently, two system design mechanisms that increase the system flexibility -at the expense of simplicity- were identified. These mechanism are generally referred to as *Composition and Delegation (*[Gamma et al. 95] & [Rumbaugh et al. 91]*),* and *Classification* ([Brachman et al. 91] & [Woods 91]). To achieve a good system design, a balance of these three important mechanisms is needed. Using each one with discretion is necessary to achieve a rich, powerful, and flexible system design.

Information exchange: Communicating with existing and future CAAD systems is a key to the success of any CAAP tool. The communication feature is needed to support the transfer of information between different design systems. This information transfer carries its share of complexities. It is quite realistic to assume that each design system would have its own representation of information that is significantly different from that of other design systems[3]. The CAAP tool will need to support the mapping of parts of its representation to and from representations used by other tools, which can be accomplished by using pair-wise translation or other more sophisticated techniques such as language binding [Snyder et. al., 95].

Databases: These are needed for two main purposes: ①to accommodate the storage of such a complex framework of information in a reliable and comprehensive manner that increases its usability and provides useful functionalities in design data management such as versioning; and ②to enable the reuse of past architectural programs, at any level of detail, in future projects through sophisticated queries and case-based design.

Support for multiple platforms: The CAAP tool should support multiple platforms, especially since it is quite difficult to speculate on what platforms would still exist in the future. To achieve this, the choice of a development tool can be very important. Graphical user interfaces are the least portable portions of computer program code, as they are usually written for a specific interface library and window system. ET++ and similar

2. The field of conceptual modeling provides useful mechanisms to achieve these types of representations and will be discussed further in Chapter 2.
3. Efforts to create a common computable design representation for the building industry, faced enormous difficulties. These were due to the inability of the various disciplines involved in the building design and construction process to agree on a single building representation stored in a shared database.

application development frameworks provide a convenient environment for developing tools that are portable across platforms.

1.5 Scope

The environment described in the previous section involves a number of core research areas, such as conceptual modeling [Loucopoulos et al. 92], case-based design [Domeshek & Kolodner 92] & [Flemming et al. 94], and functional reasoning [Freeman & Newell 71], as well as some support research areas, such as human-computer interaction (HCI) [Hix et al. 93], databases [Ullman 88], and the processing of standards and codes [Garret et. al., 95].

This research focuses on the conceptual modeling of AP information with the aim of achieving the characteristics stated in the previous section, with special emphasis on modeling building design-related information. Building such a model is the first step in achieving the AP support environment described earlier.

Chapter 2 provides an overview of conceptual modeling and devises the conceptual modeling process used as the methodology used to create the conceptual schema capable of supporting the tasks associated with AP. Chapter 3 presents a study of the AP application domain (its information concepts and the ways in which it is utilized in practice). It also contains an overview of existing computational systems that support the application domain. Chapter 4 describes the design requirements and concepts needed to support the AP application domain. It also defines the scope of the research in terms of the functionalities provided in the prototype design. The system design is described in detail in Chapter 5 using metapatterns [Pree 95] to explain the recurring design patterns employed and their relations. Chapter 6 includes a description of how the resulting system prototype can be used to create and manipulate models of design requirements. Finally, Chapter 7 provides a summary and outline of the contributions and future research areas that can be explored, based on this research.

CHAPTER 2

An Overview of Conceptual Modeling

This chapter provides a brief history of and introduction to conceptual modeling in information systems. Conceptual modeling is presented in this chapter as the means to create a computable model capable of supporting architectural programming practices. It concludes with a description of the conceptual modeling process employed to create the model.

2.1 An Introduction to Information Systems Development and Conceptual Modeling

A large number of software systems can be called *information systems*. These are systems which are data-intensive, transaction-oriented, with a substantial element of human computer interaction [Loucopoulos 92]. Information systems are becoming, increasingly, an integral part of our everyday life. They are essential in the effective running of government, industry and commercial institutions. Their usage has evolved from the automation of structured processes to applications that introduce change into fundamental procedures in many disciplines, such as engineering design and inventory management. As hardware and software technology have advanced, so has the demand for the use of information systems in a wider spectrum of applications.

This demand has been coupled with an increase in the complexity and sophistication of the tasks supported by information systems, thus rendering these systems more difficult to develop and maintain. Ad-hoc approaches to the development of complex software systems became increasingly inadequate and problematic. This was documented in several publications; one of the most famous of these is [Brooks 95]. The increasing complexity of information systems development led to the introduction of several high level modeling languages and techniques by which functional application requirements can be modeled at a *conceptual level* within a computational framework. Research in areas such as Artificial Intelligence [Barr 89], Programming Languages [Minsky 75], Databases [Ullman 88], Classification [Woods 91] & [Brachman et al. 91], and Software Engineering [Rumbaugh et al. 91], as well as Linguistics and Congnitive Psychology [Simon 89], has contributed to the development of such modeling languages and techniques, which are often referred to as *conceptual models*.

In addition to modeling languages and techniques, developing information systems requires a constant empirical observation of a certain activity (such as banking or designing an airplane) for the purpose of gaining knowledge and developing theories about the nature of that activity, particularly how it is done, and the entities involved in it (*requirements modeling*, [Figure 1]).

The subset of the world that is of interest to a specific information system is usually referred to as the *Universe of Discourse* (UoD). The knowledge gained from this process is abstracted and represented in a way that makes it possible to reason about it in the manner required by the tasks to be performed (*analysis and abstraction*, [Figure 1]). Such a process is sometimes referred to as *formalization*. These representations are then implemented in a programming language and tested in the setting where the resulting system should perform. In that respect, developing an information system involves creating "a formal description of an abstract model of a piece of reality (the UoD)" [Loucopoulos 92]. C*onceptual modeling* is the process by which the formal description is created [Figure 1]. This formal description is often referred to as the *conceptual schema* [Mylopoulos 92].

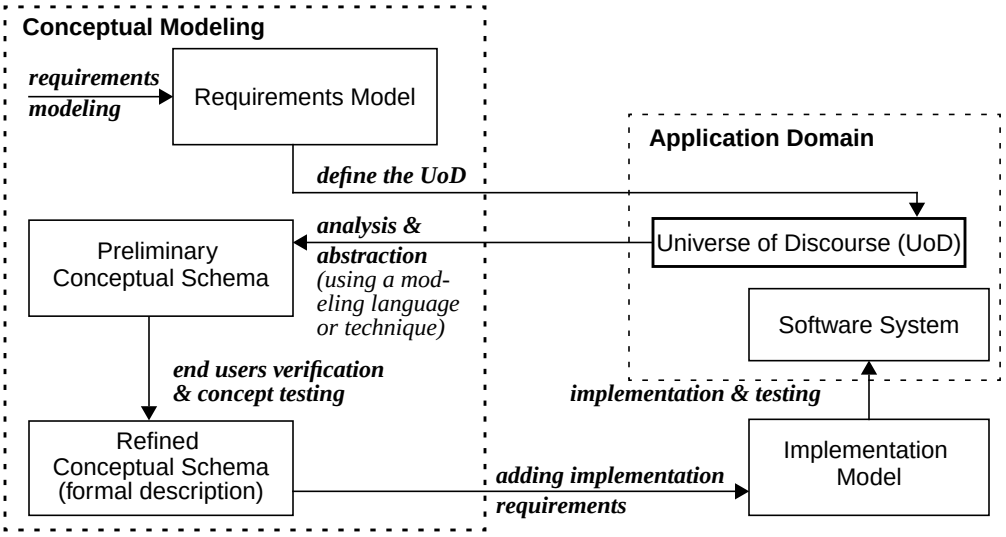


FIGURE 1. The conceptual modeling process in the development of information systems

2.2 Conceptual Schema

A conceptual schema is defined according to a conceptual model[4]. It represents static (structural) as well as dynamic (behavioral) properties and rules of the application domain [Loucopoulos 92]. It is the *conceptual product* of the modeling process and may

be regarded as a general agreement -between the developers and end users of an information system- on the way the UoD is perceived at the time of system development.

2.2.1 A Database Perspective

It is generally argued in the conceptual modeling literature that a conceptual schema should be capable of supporting all applications in the UoD over their lifetime [Loucopoulos 92]. This argument represents a database perspective of conceptual modeling in which a conceptual schema determines the kind of information that is found in the database and shared by a number of different applications. Such a perspective assumes that the applications use the shared schema directly to represent and store information [Figure 2, (a)]. It does not address situations where the information model of some applications needs to be different from the shared schema or when an existing application needs to be integrated with the shared schema.

2.2.2 An Application Perspective

In practice, however, adopting the database approach in the development of information systems has not always been successful, especially in cases where a diverse group of applications collaborate on the development of a single conceptual product, as is the case with the design of buildings. During the past four years, alliances have been established to create a shared conceptual schema for the building design and construction industry, such as the “International Alliance of Interoperability”[5] and AutoDesk’s “Industry Foundation Classes”. Such efforts have previously faced enormous difficulties in their mission, which were due to the inability of the various disciplines involved in the building design and construction process to agree on a single building representation stored in a shared database. Since computer software became available for these disciplines, each has developed its own conceptual building representation that befits the type of design and evaluation tasks it performs. For example, HVAC design software adopted representations of buildings that are drastically different from those adopted by structural design software. Such differences in representations could either be attributed to the way in which members of different disciplines were trained to think about the design problems that exist in their domain, or to the difference in the nature of the problems found in each domain. The latter view is supported by research in artificial intelligence which indicates that the way a problem is represented makes a big difference in problem-solving effectiveness[6] [Rich & Knight 91].

This difference in representations was acknowledge by recent research efforts, such as the “Agent Communication Language” (ACL) project [Khedro 95], [Flemming et al. 92]

4. In keeping with database terminology, the term “conceptual model” will be used throughout this thesis to refer to the modeling language or technique by which the conceptual schema is developed.

5. WWW page for the International Alliance of Interoperability: *http://www.interoperability.com/*, March 97.

in which the main aim was to develop an environment that supports the communication between different design systems working on a single building design. Each of these systems supported a number of design tasks usually associated with a certain discipline, such as schematic layout design, structural design, and HVAC design and energy simulation, and each system represented the building being designed according to its own conceptual schema. The communication between these modules was accomplished through a *facilitator*. The facilitator provided schema translation and propagation of design changes from the system where the changes originate to other systems that expressed interest in being informed of such changes. It was up to each individual system to understand these propagation messages and update its internal representation accordingly to maintain consistency across different representations as there was no notion of a persistent shared schema in this environment [Figure 2, (b)]. The shared schema that resided in the facilitator was developed for data translation between application specific schemata. However, a current design description represented according to the shared schema was not stored persistently.

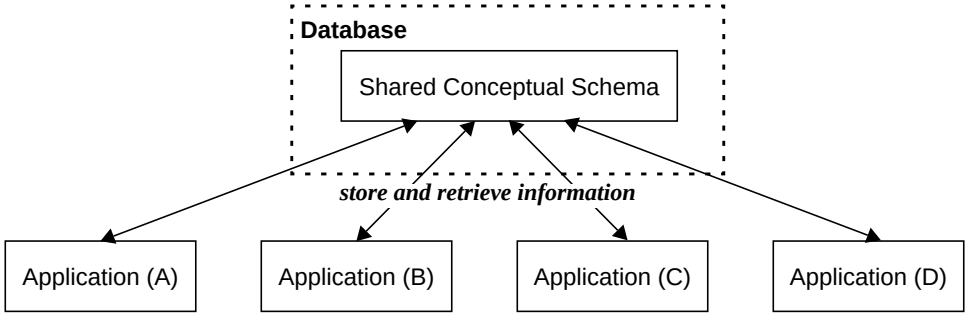
2.2.3 An Application / Communication Perspective

A different perspective of conceptual schema was adopted by the SEED project based, in part, on the experience gained in the ACL project. The main aim of this project is to create a software environment to support the early phases of building design [Flemming et al. 92]. It is composed of different modules. Each supports a practice that takes part in building design, such as architectural programming [Akin et. al., 95], schematic layout design [Flemming & Chien 95], three dimensional building configuration [Woodbury & Chang 95] and structural design [Fenves et al. 95]. Like the ACL project, it acknowledges the need for each module to have its local conceptual schema, but it also includes a shared schema, which is stored persistently in a database, and consists of the information that needs to be communicated to and shared across modules. Modules map information between their local schemata and the shared one when needed [Figure 2, (c)], and maintain an active link between these two schemata through *language binding* techniques [Snyder et. al., 95].

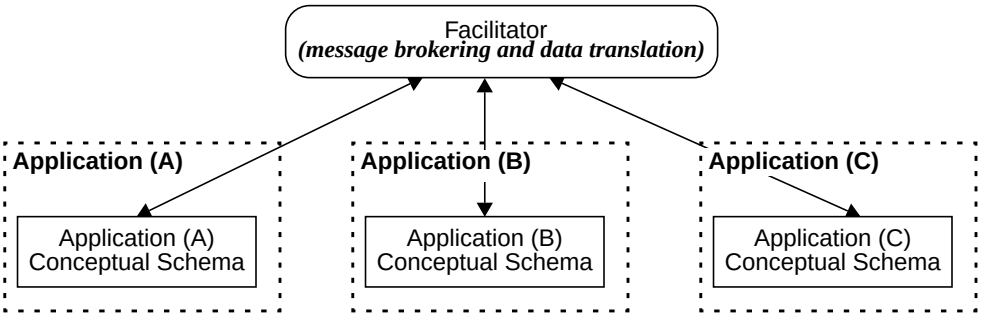
Adopting a particular perspective on the role of conceptual schema in information systems depends on the nature of the information being modeled and the environment within which the system under development will exist. It typically involves a trade-off between consistency and flexibility. A centralized shared conceptual schema ensures the

6. A famous example which is usually cited in that respect is the Mutilated Checkerboard problem in [Rich & Knight 91], pp. 107-108. The problem is to find out if its possible to cover a normal checker board, which is missing two squares from opposite corners, with pieces of dominoes, each of which covers two squares. The dominoes should be placed orthogonally without overlapping. Such a problem can be represented as a drawing of the board showing the missing corners. Such a representation doesn't suggest an answer. Coloring every other square in black will enhance the problem representation as it will show that missing corners are of the same color. Providing a count of the squares of each color suggests the answer immediately: no solution is possible because every piece of domino has to cover two squares of different colors.

a) Conceptual schema from a database perspective



b) Conceptual schema from an application perspective (adopted by the ACL project)



c) Conceptual schema from an application/communication perspective (adopted by the SEED project)

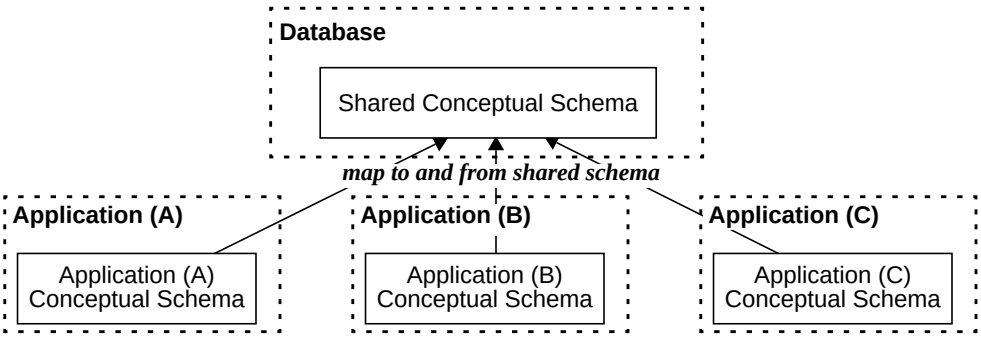


FIGURE 2. Different perspectives on the role of conceptual schema

consistency of information and provides the highest fidelity in communication between the applications that operate on that schema, as no translation is needed. On the other

hand, allowing each application to have its own conceptual schema permits these schemata to be tailored to the needs of individual applications in order to support the types of activities involved. It is important to identify the role of a conceptual schema within an information system in order to address an appropriate set of requirements during the schema development process.

2.3 Conceptual Models[7]

Conceptual models provide the format in which a conceptual schema is defined. They were originally influenced by the database field, [Brodie 86], [Schmid 75] and [Codd 70], which led to the development of the three classical database models: the *hierarchical model*, the *network model* and the *relational model*. These models were mathematical formalisms that consist of two parts: a notation for describing data, and a set of operations used to manipulate that data[8].

Further developments in the field led to the creation of *semantic models*, which support the representation of semantic (structural) aspects of world concepts, and *behavioral models*, which support the behavioral (dynamic) aspects, as well as *object-oriented models*, which support structural, behavioral and system interaction aspects. These three models are discussed in more detail in the following sections.

2.3.1 Semantic Models

Semantic models represent the database approach to conceptual modeling, which emphasizes the static properties of the schema. These properties become a union of the structural aspects extracted from all the programs that would use the schema. Semantic models provide abstraction mechanisms that make it possible to reason about the entities and the relationships between them, before building these entities and relationships into data structures. Semantic models were influenced by research on knowledge representation schemes such as *semantic networks* [Peckham & Maryanski 88], and were based on two main ideas: *data independence*, and *abstraction forms*. These ideas were expressed in three seminal papers by [Chen 76], [Schmid 75] and [Schmidt 77].

Data independence means that the conceptual schema should be free from implementation requirements, such as the physical structure of the database or the restrictions posed by a programming language. In that respect, a change made at the implementation level should not require any changes in the conceptual schema. One of the early conceptual models based on the idea of data independence was the Entity-Relationship Model introduced by Peter Chen [Chen 76].

7. Also known as “Conceptual Modeling Languages and Techniques”.
8. The entity relationship model lacks the notion of operations on data [Ullman 88], and therefore is not mentioned within the context of the three other model despite its importance as a data model.

Abstraction involves the ability to emphasize details essential to the problem and to suppress the irrelevant ones [Brodie 86]. Semantic models provide support for abstraction through a number of fundamental mechanisms known as *abstraction forms*. These mechanisms are generalization, aggregation, association and classification [Rolland et al. 92]. *Generalization* provides a way of organizing entities that represent world concepts into hierarchies based on shared properties. In such hierarchies, similarities among entities could be emphasized by placing the more general ones higher in the hierarchy and specifying specialized properties at the lower levels. This supports the reuse of properties by allowing new entities to be derived from existing ones. *Aggregation* allows entities to exist as parts of other entities, thus allowing entities to be organized into hierarchies based on containment. In that hierarchy, an entity can only exist as a part of another entity. *Association* is used to model all other types of relations that exist between entities and do not form a hierarchy, while *classification* provides a way of grouping entities according to concepts that are neither dependent on the structure of the conceptual schema nor on shared properties between entities.

2.3.2 Behavioral Models

Although the main focus of semantic models is modeling structural relations within a conceptual schema, some of these models have been extended to model the state transitions and dynamic properties of a schema, such as TAXIS [Mylopoulos et al. 80]. The need to model behavioral aspects of a conceptual schema has led to *behavioral models*. These models introduced concepts and techniques for handling the dynamic aspects of systems, and applied techniques developed for semantic modeling, such as generalization and aggregations to actions. The development of behavioral models was largely influenced by programming languages which emphasize system dynamics.

Behavioral models can be classified according to their approach to dynamic modeling into two groups [Olive 86]. The first group follows the operational approach, in which changes in the conceptual schema occur through operations that correspond to world events. When these operations are applied, the system undergoes a state transition; to ensure that only valid transitions take place, conditions are defined which determine when operations can be applied to the conceptual schema. This approach requires the definition of all allowable transitions on the schema. On the other hand, the other group of behavioral models, which follows the declarative approach, emphasize logic over control. This is done by specifying system behavior in the form of rules (which specify "what" the system should do) without taking into account the explicit control over the execution of these rules. Control is managed in an autonomous fashion through the use of facts which contain information about system entities. A rule in the systems will execute if its set of valid facts exists without the user having to consciously cause that execution. Consequently, the execution of a rule can introduce more facts into the system, causing the execution of more rules and so on.

The main distinction between operational and declarative approaches is that the former is *event-centered* while the latter is *entity-centered*. Event-centered means that system behavior is modeled around events that translate into operations being applied to enti-

ties. In entity-centered systems, in contrast, the state of system entities expressed as facts determines the changes to take place.

2.3.3 Object-Oriented Models

Object-oriented languages have existed since the late 1960's and early 1970's with the introduction of SIMULA-67 and SmallTalk. However, it was not until the late 1980's that the *object paradigm* emerged as a conceptual modeling approach that integrates structural as well as behavioral properties of systems. In this approach, objects represent world concepts in three main perspectives [Rolland et al. 92].

The first is the *static* perspective, which represents objects attributes and structural relations between objects expressed as *inheritance*, *aggregation*, *association* and *classification*, in a fashion similar to semantic models. The second is the *behavioral* perspective, which specifies the events that may occur on a certain class of objects during their life-cycle. The third is the *process* perspective, which focuses on the dynamic relationships between objects and maintains the desired overall behavior of the system through the interaction of system objects [Figure 3].

The object-oriented approach to system design provides four main contributions. The first is the refinement of the concept of aggregation into *composition*, and *reference* [Brunet 91]. In a composition, the existence of component objects is dependent on the aggregate object (a room contained inside a building will cease to exist after the building has been destroyed). Reference, in contrast, refers to aggregations where the components can exist independently of the aggregate (furniture in a room can still exist after the room has been destroyed). This distinction allows for more complex object classifications that are based on object compositions. For example, a room in a building can be classified as master bedroom, children's room or living room based on the type of furniture objects it contains. If some of that furniture is removed or new pieces added, then the classification might change accordingly.

The second contribution is an abstraction principle called *localization* [Brodie 86], which refers to the ability to determine the state of every system object on its own, regardless of its relations to other objects in the system. This is achieved by means of the object *structure* and *life cycle* concepts. The object structure consists of a set of attributes whose values can be constrained according to integrity constraints that reside in the object itself. The object life cycle is defined in terms of the states at which the object can exist, the events that cause state changes, and the constraints that restrict the succession of event occurrences [Rolland et al. 92]. This makes the behavior of an object explicit and localized, and provides a description of the overall system behavior in terms of the behavior of individual objects. This is different from classical behavioral models, where the life cycle of a system entity is determined through the operational conditions or rules that exist within the system as a whole.

This behavioral modularity, coupled with a well defined object structure, led to the third contribution, which is referred to as *design patterns* [Gamma et al. 95]. Patterns provide the link between structural properties and behavioral properties of object compositions.

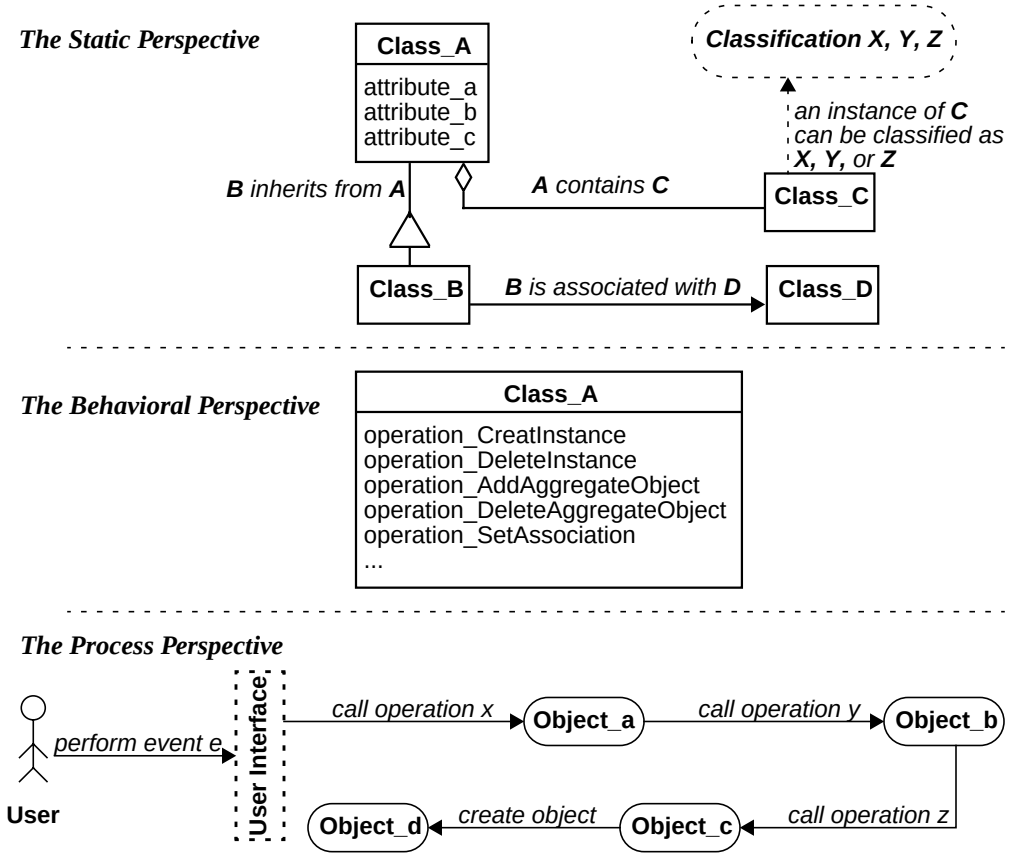


FIGURE 3. An example of the three main perspectives of objects in the object-oriented approach

They do this by providing the object structure that is most capable of solving a given design problem. In addition to that, design patterns makes it possible to reuse system designs at different levels of granularity.

The final contribution is the ability to use objects, at different levels of abstraction, in the entire system development process, starting from the requirements specification stage. This ability was demonstrated in the Use Case approach to modeling requirements [Jacobson et al. 92], which describes the system functionality from the system user perspective. Such description includes references to system objects, especially at the interface level, which later develop into detailed object descriptions.

2.4 The Conceptual Modeling Process Used

Many different methodologies can be conducted with the conceptual modeling process. They have been documented in the Software Engineering literature, such as *Object Mod-*

eling Technique (OMT) [Rumbaugh et al. 91], *Object Oriented Software Engineering* [Jacobson et al. 92], and *Software Engineering with Abstractions* [Berzins & Luqi 91]. Most references emphasize the point that these methodologies should not be considered as "cook books", but rather as general design and modeling guides to be adapted to different modeling situations.

This section describes a conceptual modeling process based on some of these methodologies as well as the author's experience. The process is divided into five main steps: ①studying the application domain, ②defining the information system functionalities (requirements modeling), ③defining the Universe of Discourse (UoD), ④defining the conceptual schema according to a conceptual model, and ⑤verifying and refining the schema. Once the schema is refined, an implementation model can be developed to implement the schema in a given computational environment. These five steps – explained in detail in this section– are presented as the means to achieve the main goal of this research, which is creating a framework for modeling building design specifications.

2.4.1 Studying the Application Domain

Defining the information system functionalities requires a substantial amount of domain knowledge. The modeler therefore has to be familiar with the domain that the system should support. In order to provide a more wholistic perspective of the application domain, this domain knowledge has to be acquired through some knowledge acquisition process, even if the domain expert happens to be the system modeler. Knowledge can be acquired through several methods [Rolland et al. 92].

One of these methods is to conduct a study of the application domain by reviewing its literature and interviewing domain experts as well as observing them while they perform their tasks. Such a study is very likely to provide multiple (possibly different) views of the domain. It is important to realize that these views originate from the personal experiences that the experts have accumulated over the years and can be tied to the ways they prefer to perform their work. This means that the conceptual schema that would be created does not have to map directly to any of these domain views or to one that encompasses all of them. Instead, the domain views provide the modeler with an understanding of the domain which makes it possible to create the conceptual schema that can support the functionalities the information system should provide.

Another way in which knowledge can be acquired is through studying existing information systems of the domain, which can be done in three different ways. The first, and most intuitive, is to study the functionalities that these systems provide and analyze their effectiveness in supporting the domain. The second is by studying the requirements of these systems (usually expressed in a requirements analysis document). This requires specifications that are well documented and revised to reflect the current state of system development. It also requires that the specifications include some description of the functionalities they support; otherwise the modeler will have to infer that link. Studies have been conducted about ways to support the reuse of system specifications, such as providing a repository to store reusable system components [Johnson & Feather

90], and using an "Intelligent Reuse Advisor" to retrieve and customize these components according to cognitive models of specifications reuse and analogous reasoning [Maiden &Sutcliffe 91]. The third way in which existing information systems can be used to develop new ones is by *reverse engineering*, which is defined in [Chikofsky & Cross 90] as a process to analyze a subject system to identify its components and their relationships and to create a representation of the system in another form or at a higher level of abstraction. This leads to recapturing or recreating the system design, then deciphering the requirements that are actually implemented in the system. The design recapturing stage is supported by some commercial CASE (computer-aided software engineering) tools such as Rational Rose by Rational Software Corporation[9]. However, deciphering the requirements from the reconstructed design representation is left to the modeler.

2.4.2 Requirements Modeling: Defining the Information System Functionalities

Once the application domain is sufficiently understood, system requirements can be defined. Although several methods for defining system requirements have been proposed, current software engineering practices and research seem to be converging on the *use case* approach for modeling system requirements [Jacobson et al. 92]. The use case approach defines system requirements in terms of a set of interaction scenarios called *use cases*. Each use case is a natural language description of the way the user performs a certain task when using the system. This description specifies a sequence of events and interactions that take place –during the completion of the task– between the user and the system interface, and between the system components as well. The description ends with the completion of that task. A use case can use other use cases as parts of its events sequence.

The development of use cases start with the main tasks the system should support. Tasks are then decomposed into sub tasks and so on, until an "acceptable" level of granularity is achieved. The level of decomposition depends on two main objectives. The first is that a use case represents a basic task that is clear and simple and does not naturally decompose into sub tasks. The second objective is to maximize the reusability of use cases to simplify requirements and highlight core system functionalities that provide support to other higher level ones. As these two objectives can be contradictory, the acceptable level of granularity is usually achieved after a series of iterations, as is the case with most design activities. Use cases can be grouped by the type of *actor* performing them or by higher level tasks they support. Together, they represent a model of system requirements from the user perspective.

The use case approach has six main advantages:

1. It focuses the system development on supporting the tasks the user needs to accomplish. This helps reduce the size of the system by eliminating unnecessary functionalities and ensures that the tasks to be performed with the system are adequately supported.

9. URL of Rational Software Corporation: *http://www.rational.com*, October 97.

2. The use case document provides the users with a clear description of the system functionalities which they can evaluate. Once the document is approved, it forms an informal contract between the system users and developers.
3. It helps define the system interface needed to accomplish the tasks. This is due to the descriptions of system dynamics included in the use cases, which help the system designer determine the style of interaction needed to support the user actions.
4. Since use cases contain references to world elements that the user manipulates, they help determine the Universe of Discourse of the application domain in terms of the entities needed to accomplish the system tasks, and hence to be modeled.
5. The use case descriptions precede the determination of the objects of interest, which are taken for granted in other object-oriented methods.
6. It structures the entire development process around the requirements; they become a thread that integrates all stages from system modeling to testing.

2.4.3 Defining the Universe of Discourse (UoD)

The third step in the conceptual modeling process is defining the UoD, which is the subset of the world to be modeled. The definition of the UoD starts with the set of domain concepts referenced by use cases. This set is then expanded to include other concepts needed to support the ones included in the original set. The defined UoD can then be used to model the conceptual schema.

2.4.4 Schema Modeling

Choosing a conceptual model: Modeling the conceptual schema entails defining system entities that represent world objects according to a conceptual model. Object-oriented models provide three perspectives for schema modeling which were described earlier: structural, dynamic and process. The conceptual model used in this thesis to support schema modeling combines different techniques:

- Object Modeling Technique (OMT) [Rumbaugh et al. 91] will be used for modeling structural and behavioral properties of the schema through its *object model* and *dynamic model* representations.
- Interaction diagrams [Jacobson et al. 92] & [Gamma et al. 95] will be used to model system processes. They will be mainly used as a tool for verifying and testing the schema, and will be explained in section 2.4.5.

OMT provides a clear and powerful notation for modeling the structural properties of a schema. It supports all abstraction forms mentioned earlier with the exception of classification. Its object modeling notation allows for creating a classification taxonomy, which can be modeled and implemented using a classification framework such as *Classic* [Brachman et al. 91].

While OMT supports the modeling of behavioral properties through its object and dynamic models, its *functional model*, intended to model system processes, is not as useful as its object or dynamic models (which Rumbaugh himself admits). This is the rea-

son for using interaction diagrams to model inter-object interactions; this is true in many of the latest software engineering methodologies such as the Unified Modeling Language. Object and dynamic models, as well as interaction diagrams, are supported by various modeling and CASE tools, such as OMTool by Lockheed Martin, and Rational Rose by Rational Software Corporation.

Modeling the schema: Schema modeling starts by defining the object types and their static properties (attributes) that represent the UoD. The structural properties between objects are inferred from interactions described in the use cases, and modeled as associations and aggregations. Inheritance can then be used to share and reuse common properties (both static and dynamic) that exist among the different types of objects. Classification can be used to create taxonomies of object types based on multiple classification criteria.

Once the structural properties of the schema are defined, its behavioral properties can be modeled in terms of the operations that apply to each type of object. These operations are defined in terms of the information they need to perform their tasks, and the changes they cause to the state of the object. A dynamic model for each type of object in the schema can then be created. It defines the changes in the object state (also known as a description of the object life cycle) as a result of applying every one of its operations.

2.4.5 Verifying and Testing the Schema

After the schema has been created, it can be verified by modeling the interactions between its objects needed to support each one of the use cases, using interaction diagrams. This requires that an interaction diagram be created for every use case to ensure that the defined schema supports all system functionalities. Interaction diagrams also help define the type of information that needs to be exchanged between operations, and verify each operation's inputs and outputs accordingly.

Testing the schema requires designing a set of test cases aimed at using the schema to create representations of building design requirements for actual buildings. This is most suitably done using a prototype of the proposed system- implemented using a programming language, such as C++ [Stroustrup 91], or within the programming environment of a database management system-to facilitate the testing procedure. Although it can still be done manually if a prototype proves too costly to create for the purpose of testing and evaluating the schema. However, it is the author's opinion that testing procedures have to be conducted on a system prototype before a reliable schema design is reached.

CHAPTER 3

A Study of the Application Domain

This chapter provides a study of the application domain, which is the initial step for creating a conceptual schema that supports the generation and manipulation of building design requirements. The study consists of two main parts: a survey of architectural programming practices, and a study of computational tools that attempt to support these practices.

3.1 Architectural Programming in Existing Practices

A study of architectural programming practices was conducted with my participation [Akin et. al., 95]. This involved interviewing a number of architectural firms renowned for their AP practices. It indicated that there are two principal roles of AP in practice. The first is providing a *framework* for architectural design that manages design context, requirements and criteria for the different phases of building design. It regards the architectural program as a reservoir of design information that feeds other stages of design with design data. This role is characterized by the following aspects [Akin et. al., 95]:

- Program documentation results from a process of negotiation that settles disputes regarding the performance requirements desirable in the resulting architectural designs.
- The architectural program is an informal (and in some cases formal) “contract” between the client and the designer that tracks the scope and budget of the project and any subsequent changes to them.
- The program serves as a complete inventory of design requirements and criteria of design evaluation.
- Documenting the program is not a one-time task, but a dynamic one that keeps track of the evolving and changing AP throughout the design process.

The other role of AP in practice regards the program as a *utility* for designers. The program becomes a step in the early stages of design that merely facilitates the transfer of user needs into design entities. Once this is done, the discourse about performance requirements is considered only within the physical design domain, with little or no

need to return to the program. The main assumptions underlying this role are the following:

- The program becomes rapidly outdated, and the design drawings are the most reliable source of information, which includes both design and client requirements.
- The program is just a convenient stage in translating the user needs into design objects.
- Documentation of the program is a routine task and, in most cases, performed in a way that does not consider the reusability of the program, or parts of it, in future projects.
- Documentation of the program is a one-time task; thus, it becomes fixed in time, which is the reason why it becomes outdated.

The study also indicated that practice varies in terms of the *framework* vs. the *utility* models of AP. However, we also found a number of common characteristics. First we came across attempts to use past programs in developing new ones. However, many limitations and difficulties adapting past programs to new situations is fought with, particularly because of the manual methods employed in doing so, and the format in which programs exist. Our study also showed that the program takes into account client's standards and codes, which are as critical to the design as the locally enforced institutional building codes and ordinances. However, the values and resources allocated to programming not only vary from firm to firm, but from project to project within the same firm. It is common that the client and the nature of the project determine the priority of the program. Finally, the study concluded that developing the architectural program is a task of a team with diverse areas of expertise, and that there was no consensus about the use of visual design representation in the program.

Our study also helped us describe the architectural programming process in four steps:

1. *Specifying all design requirements:* This is the most variable and open-ended step of program development. All aspects of the design problem should be clearly defined. These include site, organizational, code ordinance and budgetary aspects.
2. *Deriving from these requirements the functional description of the architectural design problem:* The specifications collected and defined in the previous step are then translated into a more precise format, which includes the functional and area components of the program, as well as the spatial, mechanical and equipment needs associated with these components.
3. *Documentation of the program:* This is done in such a way that design can proceed, based on the requirements and functional descriptions developed in the previous step.
4. *Updating the program to include any changes made during the course of design:* This is a step that varies in practice; it usually takes place in the *framework* model, while the opposite is true in case of the *utility* model.

3.2 A Study of Computer-Aided Architectural Programming Tools

3.2.1 An Overview of Commercial Systems

Currently, a number of computer-based systems marginally support the AP process. Among these systems are Intergraph's *Project Programmer with Project Optimizer* (1991), which provides some space planning capabilities, such as space optimization using stacking and blocking diagrams, and the SARA Facility Development System (1994), which provides cost estimates based on average cost databases in addition to stacking and blocking diagrams. Some firms we interviewed have developed their own programming applications generally based on spreadsheets to assist in the numerical aspects of the program, such as area calculations. There also exist some computer-based tools for facilities management, such as AutoDesks's *AutoFM*, but they provide only building management and maintenance capabilities that include space planning based on stacking and blocking.

Recently, another software system named SABA[10] –for Stacking And Blocking Algorithm– was ported from main frames to the MS Windows environment. This system provides three views for creating and manipulating architectural programs. The first view is a bubble diagram view through which building elements can be entered as spaces with fixed areas. The system allows for expressing adjacency relationships between entities using an adjacency matrix. Adjacencies are the only relationship that can be expressed in this system and are used as a basis for minimizing costs. It therefore uses an objective function to indicate the "preferred" locations for placing spaces into floors in the stacking view, and next to each other within the same floor in the blocking view. When space assignments are done, the system provides reports, such as a break down of spaces by floors or zones. The main algorithm through which space assignments are determined is based on the *quadratic assignment problem*, an optimization problem in which functional units are assigned to locations so that the cost of moving people and equipment between modules is minimized [Koopmans & Beckman 57]; [Liggett 80].

In general, computer applications in the area of architectural programming consist of one, or a combination, of the following elements and techniques:

1. Computerized libraries of building elements and equipment inventories that can be selected and used in the architectural programming process.
2. Cost estimation techniques based on cost databases[11].
3. Space planning using stacking and blocking algorithms.

10. URL for SABA: "*http://www.techexpo.com/WWW/saba*", March 97.
11. Our surveys in the SEED-Pro team indicated that architects rarely use average cost databases in doing cost analysis and estimations. Instead they rely on their past experience given the situation at hand as well as the "fresh" localized information they collect in that respect.

3.2.2 Problems and Shortcomings of Existing Tools

Most existing tools provide only marginal support for selected programming activities, such as stacking and blocking of spaces. They are not integrated with other design systems and therefore, are not widely used in practice. Most of these tools export their information either as text reports or as a simple description of geometry that can be imported into drafting tools such as AutoCAD or MicroStation. They also do not provide any support to handle the non-spatial aspects of AP information, such as performance criteria and preferred material properties.

The main reason for the shortcomings of these tools is their lack of robust representations of architectural programming information. Their representations consist of spaces which can be assigned to floors and zones. These spaces have two attributes: area and adjacency relations to other spaces, while other types of attributes and relations are not part of these representations. It is very likely that only these two attributes have been selected because they are required by the stacking and blocking algorithms. These algorithms are used to assign spaces to floors and design floor layouts by minimizing the travel cost between spaces, which appears to be the functionality these systems were actually developed to provide. However, even that functionality is not adequately supported by these systems. The reason is that traffic flow between spaces is neither the only nor the most important criterion for designing layouts. This is indicated in [Fenves et al. 94], p. 43:

> Layouts must respond to a multitude of concerns, from adjacencies and proximities through visibility and view to more formal aspects of order and organization, and these generally competing concerns may even contradict each other. The quadratic assignment formulation is unattractive for building design because, despite the severe restrictions under which it operates, the computation of an “optimal” assignment remains costly and has led to the employment of more heuristic approaches in practice.

The inadequacy of the representations adopted by these systems has decreased their reliability and limited their use in practice. It has also resulted in the use of mechanisms that do not render adequate support and reliable results for the functionalities they provide. For example, the SARA Facility Development Systems provides estimates on cost using average cost databases, which are known to be unreliable. This makes it unlikely that professional firms will use a system that automates a method they don’t use in the first place. In that respect, I argue that a more sophisticated cost analysis system will have to utilize a rich representation of architectural programming information. Likewise, other early design functions, such as negotiating with clients, task scheduling, feasibility studies, and the like can benefit from comprehensive AP support.

3.2.3 Research Systems: SEED-Pro

Until recently, research on computer-aided architectural programming (CAAP) has not provided any functionalities beyond what the surveyed commercial systems offered. However, in 1993 researches at the Engineering Research Design Center (EDRC) and the

School of Architecture at Carnegie Mellon University started the SEED project, which provides a software environment to support the early phases of building design. The overall goal of SEED is to provide support for the preliminary design of buildings in all aspects that can gain from computational support. This includes using computers for the *generation* of designs to achieve a "*rapid generation of computable design representations* describing conceptual design alternatives and variants of such alternatives with a sufficient level of detail that enables sophisticated evaluation tools to receive all of the needed input data from the representation" [Flemming & Woodbury 95]. SEED intends to encourage the exploratory mode of designing by making it easy for designers to generate and iterate through design versions and to pursue conceptual alternatives in parallel.

SEED features an open-ended modular architecture, where each module provides support for a design activity that takes place in early design stages. Each module consists of five main components: input, specification, generation, evaluation and output. These are supported by a database to store and retrieve information, as well as a user interface to support the interaction with designers [Figure 4].

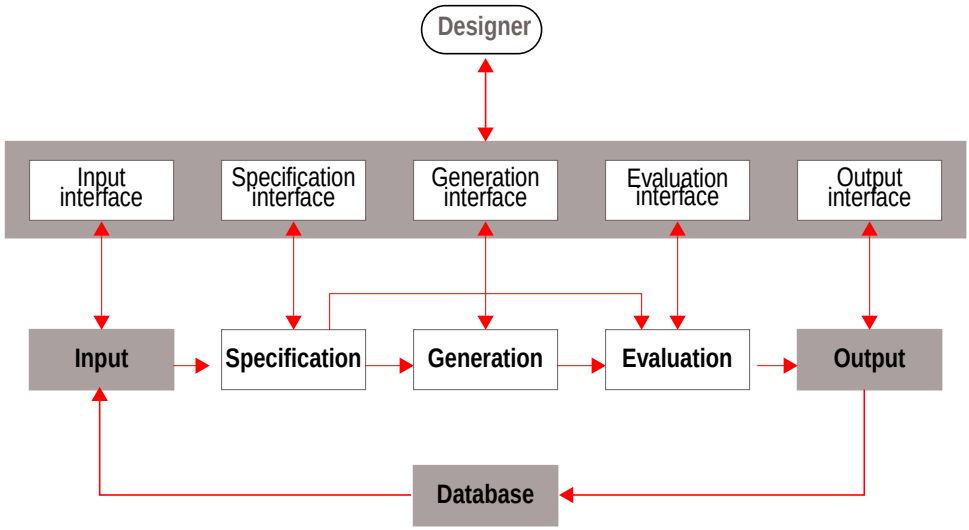


FIGURE 4. Generic Architecture of a SEED module [Flemming & Woodbury 95].

To support design generation, a well-defined set of explicit requirements is needed. The prototypical version of SEED-Pro, the architectural programming module of SEED, is designed with the intention to support the modeling and generation of design requirements in a form that is usable by other modules of SEED. It has the following objectives [Akin et. al., 95]:

1. Provide means of storing and handling all aspects of AP information including site characteristics, codes, client preferences, and different performance criteria and requirements.
2. Ensure the use of criteria established during the programming phase as a basis of design.
3. Enable the integration of architectural programming and architectural design as a seamless process.
4. Maintain a database of reasons employed in making programming decisions, and improving the computability of AP information by allowing non-numerical types of reasoning not currently supported by the existing medium of representation (text for non-numeric and spreadsheets for numeric data).
5. Achieve a flexible way of interaction that does not tie the designer to a particular model of AP.
6. Enable the use of past programs and programmatic experience in future projects.

Through the sharing of domain concepts, SEED-Pro aims to provide a seamless interaction with all of the other modules of SEED and share data across these modules. SEED-Pro positions itself as a good candidate for maintaining a robust record of design requirements, criteria, and constraints to be used persistently in SEED. SEED-Pro is intended to support both the *framework* model of AP by providing all of the features it requires, as well as the *utility* model of AP by allowing users to pick and chose any combination of AP features that they like to have.

By providing the outputs that the other SEED modules require as input and through the shared domain object classes and libraries in SEED, SEED-Pro complements the basic steps of SEED: problem specification (SEED-Pro), spatial layout design (SEED-Layout) and construction specification (SEED-Config). SEED-Pro converts parts of its input into representations used by other modules of SEED, and accepts its input from three main sources:

1. Direct user input through an interactive graphical user interface which allows the user to input and manipulate the AP information represented in SEED-Pro as specification units [Figure 5]. Providing an effective user interface is a major research area in the SEED project.
2. Reading information stored in the SEED database in the form of *Projects*. These can be used to create a new set of project specifications or modify an existing one.
3. Interfacing with the Standards Processor [Garret et. al., 95] to acquire codes and standards to which the design requirements should conform. Interfacing with the Standards Processor provides an external evaluation mechanism, which can be used for compliance checking of the AP against a specific standard, or for compliant generation, which will help the designer to retrieve applicable provisions or constraints, or both, and incorporate them into the generation of design requirements. This will be helpful when an AP for a new building type is being generated and the designer is not familiar with the codes or standards. This process can also be referred to as a kind of guided generation of design requirements because the idea is to generate a program that is compliant with the standards. Other external analysis, simulation and

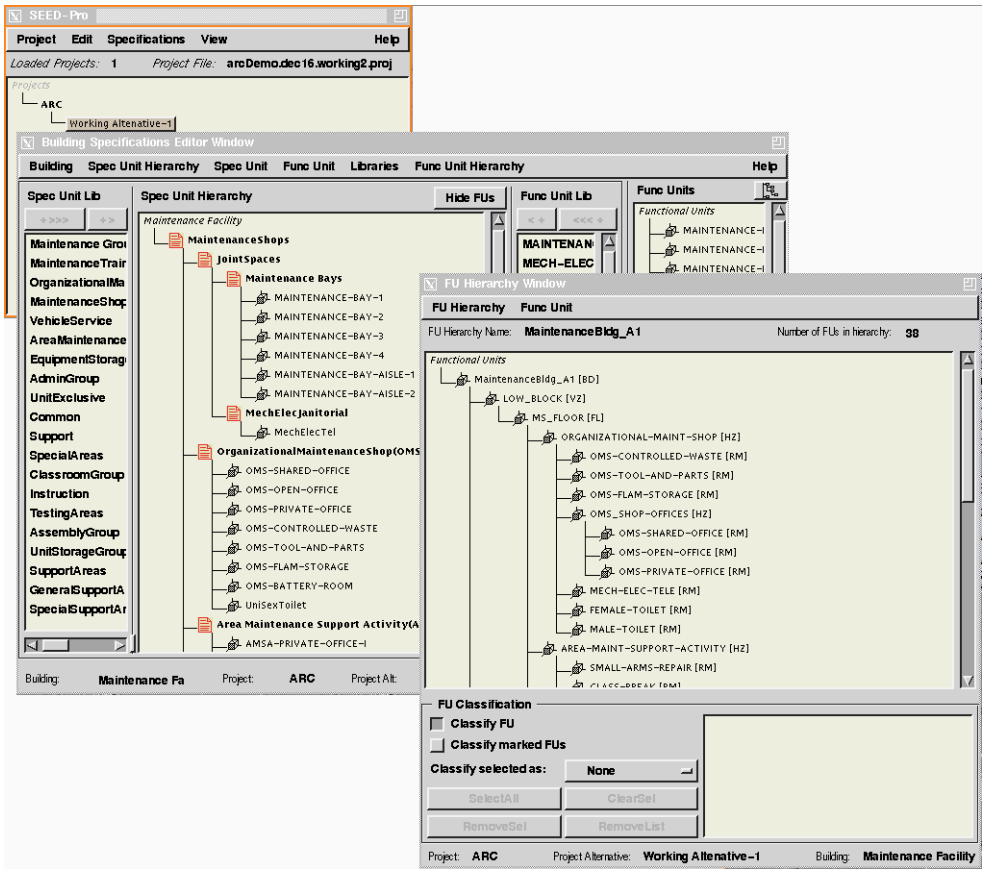


FIGURE 5. User input windows in SEED-Pro

evaluation software can, in principle be integrated as well (e.g., a well established cost estimator or scheduler).

The SEED-Pro conceptual schema:

The *Project* is the main construct in SEED-Pro. It is an object that forms the super container for all the different types of objects in the SEED-Pro object model [Figure 6]. A project consists of a set of *design specifications* which are defined as a collection of design intentions and criteria to be maintained by the architectural program. These design specifications serve as the front end to SEED-Pro and provide it with a consistent and persistent record of specification categories.

Five categories of these specifications have been identified so far: Building, Site, Budget, Implementation Schedule and Client Profile. Each of these is composed of a set of *specification units*. These specifications categories, along with the mechanisms needed to create and manipulate them, constitute the *specification* component of SEED-Pro. Building

specifications provide a description of the design requirements for each building in the project. Site specifications contain a site description which includes existing site elements as well as future ones, such as the building footprint. Implementation Schedule specifications contain a description of the desired implementation schedule. Finally, the Client Profile specifications maintain a database of clients involved with each project. Each client record in the database contains a description of the client profile and preferences as well as contact people within each client organization.

The specification component allows for multiple architectural programming alternatives that satisfy the same set of specifications; it also facilitates case matching and retrieval. It captures the rationale and the intentions of the design criteria, thus providing a basis for functional reasoning by being able to trace a design decision to the specifications that initially triggered it.

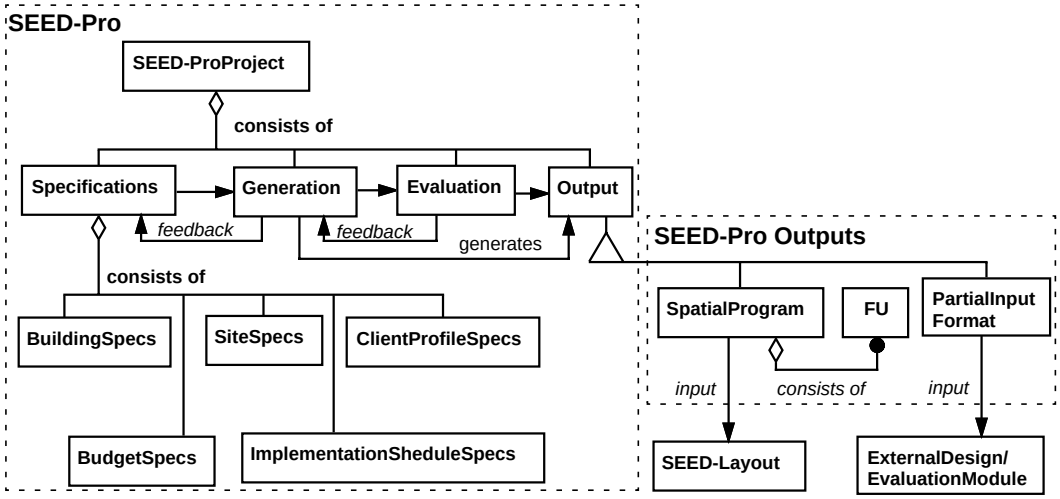


FIGURE 6. The SEED-Pro Project structure using the OMT notation.

A *specification unit* (SU) is the basic building block for representing specifications in SEED-Pro. It embraces an object-oriented representation that consists of the SU and *Component Specifications* objects. A SU object represents an organizational entity in the building, which can correspond either to a physical space or a room, or to an abstract operational concept, such as a department or a section. SUs can be recursively aggregated to form the organizational and functional hierarchies of the building as shown by the link “contains” in [Figure 7]. A SU can be associated with a number of *component specifications* objects, each of which describes desired performance requirements and design criteria of the corresponding SU. The list of components, shown in [Figure 7], is open-ended and allows the system designer to add additional component types as needed. Once the desired attributes of an organizational entity are specified, that entity can serve as input to the generation mechanism within SEED-Pro that develops the AP needed by another design or evaluation module.

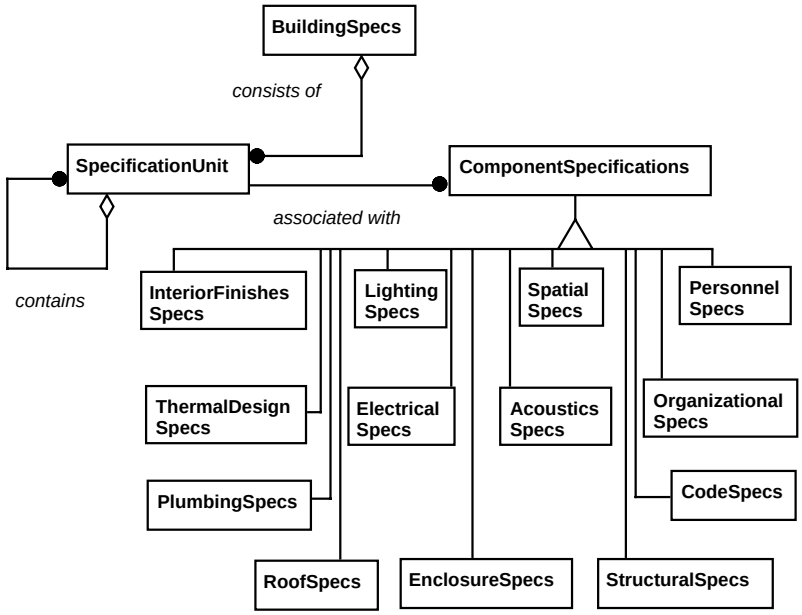


FIGURE 7. The Specifications Object Model using the OMT notation.

One such mechanism is the generator of the spatial requirements in the form of *functional units* (FUs). These are objects that contain the spatial requirements of spaces. They can be structured hierarchically to form a spatial composition that describes the building in terms of floors, zones and rooms etc. Spatial information that resides in SUs is captured and "packaged" into functional unit hierarchies, which constitute the input needed by the layout module of SEED.

Another generation mechanism is the specifier of thermal conditions as partial input to a thermal design module that interfaces with SEED. In addition to these generation mechanisms, the structured representation of SEED-Pro allows for generating an architectural programing document from the specifications when a written document is needed. Such a document is typically required for contract negotiations.

The first prototype of SEED-Pro is now running on Unix, Microsoft Windows 95 and NT platforms. It supports the functionalities needed to:

- create a representation of an architectural program that consists of specification units and the set of component specifications shown in [Figure 7],
- manually create the spatial requirements expressed as functional units,
- edit the spatial configuration of buildings and specify relations among functional units, such as adjacency, minimum distance and accessibility, and

- export the spatial configuration to the layout design module (SEED-Layout) either through the database in the fully networked version, or a common file format in the "Lite" version designed to run on a desktop machine.

3.3 Achieving a Flexible Framework for Modeling Design Requirements

The current SEED-Pro schema provides flexibility in terms of structuring and manipulating AP information, representing existing as well as future building elements and accommodating different building types. However, it provides little support for changing the set of design specifications in terms of adding new types of specifications (such as mechanical systems and equipment) or selecting subsets of specifications with which to work. This requires changes in the user interface, which reflects the model changes to the user, and the generation mechanisms that are affected by these changes. As the schema stands now, the addition of new types of specifications requires some programming effort and cannot be achieved at run-time by the system users.

The complexity of modeling design requirements for buildings arises mainly from the absence of a formal way to define such requirements. Firms specializing in architectural programming, such as HOK, have adopted or created models for defining design requirements. In general, the ways in which these models define design requirements are often grouped into two main *styles* [Kumlin 95]. The first is the *prescriptive style* through which design requirements are specified in terms of properties of materials and building systems based on standards or on the designer's experience. The second in the *performance style*, by which design requirements are specified in terms of performance criteria, such as required air temperature, illuminance, activities to be performed and so on.

Adopting one style over the other is most likely to result in an inadequate model to define design requirements. Different aspects of design requirements are better specified in one style versus the other. It is often the case that both styles are needed for different analysis purposes, and it is sometimes hard to determine whether certain design requirements are prescriptive or performance-based. For example, defining the R-value of a wall can be considered as a performance criterion or alternatively as prescriptive material property.

CHAPTER 4

A Computable Representation for Modeling Building Design Requirements

This chapter provides a description of the criteria used to design the computable representation and an analysis of its features.

4.1 Modeling Principles and Considerations

Given the background and the study presented in chapters 1 and 3, a computable representation for modeling building design requirements needs to fulfil two main goals. The first is to create a repository of design requirements that contains all the information the designer needs to specify. The second goal is to support extracting information from that repository and communicating it to the design systems that collaborate on the process of producing a building design.

To achieve the first goal, the computable representation should support the definition of multiple conceptual schemata, each representing a different way of describing design specifications. In essence, these conceptual schemata are *product models* that provide the categories of information needed to create a set of design requirements for a certain product. An example of a product model is one that represents the AIA specifications, the Army Design Guidelines or any firm-specific way of describing design requirements. Such a model provides the information semantics that can be used to represent design requirements for a given building. The selection of a product model to use depends on the design systems that need to extract information from these requirements. For example, SEED-Pro, the architectural programming research system surveyed in Chapter 3, was required to interface with SEED-Layout, which is a software module within the SEED system that can generate floor layouts from a given set of spatial design requirements. Such a requirement means that these spatial design requirements had to be either directly represented in, or can be inferred from the SEED-Pro product model, in order to create the input needed by SEED-Layout. The SEED-Layout input format is represented within SEED-Pro to allow for structuring it in a way that conforms to the measures of well formedness required by SEED-Layout.

The need to represent multiple product models within the computerized representation calls for having a different perspective on the role of a conceptual schema within an

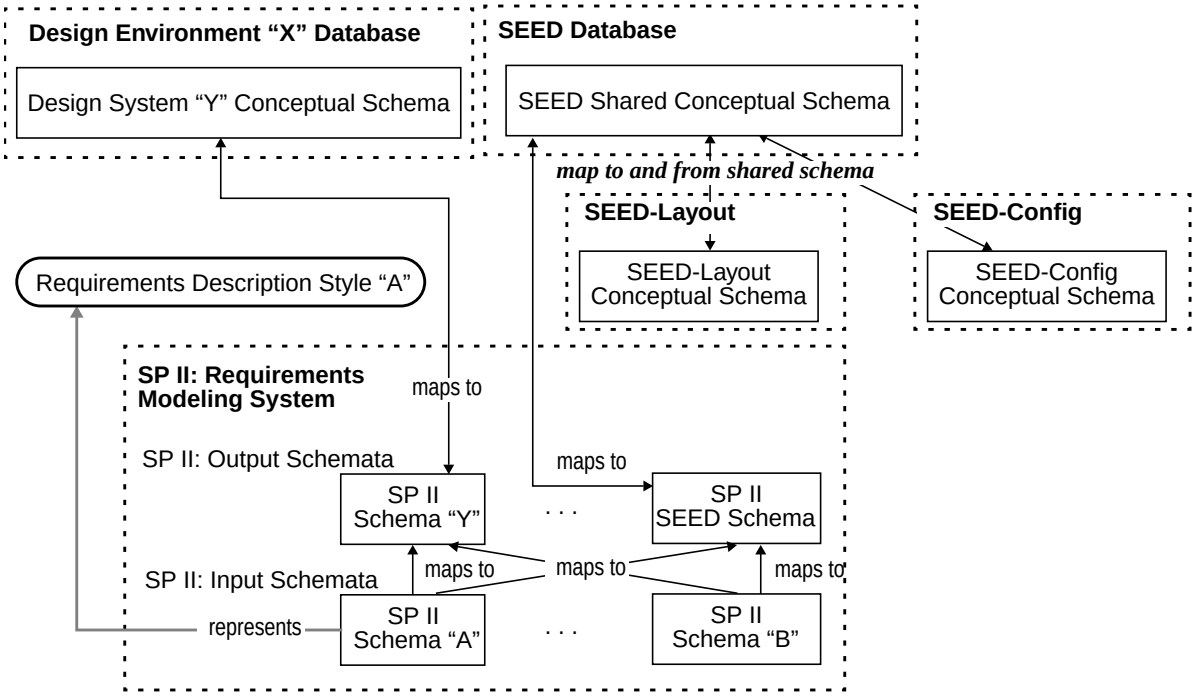


FIGURE 8. Multiple conceptual schemata representing different product models

information system. This perspective differs from all three prespectives discussed in section 2.2, in that it allows a single software module to contain multiple schemata[Figure 8]. Each schema represents a product model that either supports a certain style of describing design requirements, or conforms to the input needed by a design system in a certain environment.

Providing information to other design systems requires a series of generation mechanisms, each specializing in creating the output needed by a specific design system. Once that output is created, it often needs to be structured in a way that is acceptable by the design system for which it was generated. For example, SEED-Layout requires its input to be in the form of a hierarchy of spatial functional units which conforms to certain rules of well-formedness. Such rules concern attribute values within each functional unit (e.g., the minimum area should be at least equal to the square of the minimum width), as well as rules in composing the hierarchy itself (e.g., a functional unit representing a *room* cannot contain one that represents a *floor*). This means that the computable representation should be able to handle the structuring of these various outputs independently from the way the repository of requirements is structured.

Therefore, the principal requirements for a computable representation for modeling building design requirements is the ability to:

- represent, at a minimum, the type of information needed by all design systems interfacing with such a representation,
- support the generation of data needed by each design system, and
- allow the structuring and refinement of the generated data to meet the standards set by each design system.

These principal requirements have been expanded into a detailed requirements analysis using use cases [Jacobson et al. 92], which is included in Appendix 1.

4.2 Approaches to Modeling Design Requirements

Modeling design requirements could be achieved using three different approaches. The first approach is to model design requirements after an industry standard (product model), such as the AIA specifications or the Sweet's catalog classification. In this case, the product model used will be hard-coded into the system, and the design requirements generated using such a system can only use the type of information provided by the product model used. Such an approach is relatively easy to accomplish and is likely to be used by people and firms who adopt the standard on which the model is based. It can be successful when the interfacing design systems are well defined, as it was the case with SEED-Pro. However, this causes the system to be restricted to a single standard and does not achieve the flexibility needed to accommodate the volatile nature of information categories in the domain, or the ability to cater to new design systems that were not envisioned initially when the system was being developed.

The second approach is to provide a product model that encompasses all possible industry standards—a union of all the existing standards and classifications. In this case, the model will not be restricted to a single standard and would cater to a wider group of practitioners than would the model described in the first approach. However, this approach would prove to be impractical due to the possible contradictions that can exist between standards, and the very large number of such conventions used in the field. Each specification writer in the field has his/her own set of categories and conventions. Most standards are based on some criteria that can differ from the ones on which other standards and classifications are based, which increases the possibility of conflicts in the way some design requirements are modeled. In addition, this approach does not address augmenting existing standards and classifications and could prove to be inflexible as well.

The third approach is to provide a flexible framework to model design requirements according to any standard or classification. This framework is intended to be adaptable in a way similar to software frameworks, such as ET++ [Weinand et al. 95]. A software framework is a set of reusable components that can be used to create applications for a certain domain.

Similarly, a framework for modeling and manipulating design requirements provides some overall organizing concepts to model design requirements. This framework, con-

tains adaptable components to accommodate different ways of describing design requirements. This approach should be more flexible than the previous two approaches because flexibility is addressed as a design requirement from the start. It enables the addition of new specifications categories at run-time, and could extend even beyond building design, providing support for modeling design requirements for any engineering artifact that can be described using its overall organizing structure.

I decided to adopt the framework approach for creating a computable model for building design requirements. The reason for that decision is its potential to fulfil the following requirements:

1. The computable model should not be tied to a specific model of defining design requirements.
2. Support an open-ended architecture where new models of specifications can be created to accommodate new design systems that need to interface with the model.
3. Support experimenting with different models and styles of specifications.
4. Support creating mappings between different models of specifications to generate system-specific outputs from the design requirements repository.
5. Creating models of specifications and mapping techniques at run-time using the system interface. This enables domain experts to create these models and mapping techniques without having to reprogram the system.

4.3 Features of a Design Requirements Modeling Framework

The framework created in the course of this research was designed and built to support the functionalities listed in the previous section. Its design features can be grouped into three sets of features [Figure 9]:

- creating building design requirements,
- creating product models, and
- creating and applying generation mechanisms to map information from one product model to another.

Each of these three sets of features is explained in detail in this section.

4.3.1 Building Design Requirements

Creating design requirements entails means to represent and structure the design entities described by these requirements. This framework adopts a hierarchical structure to represent the design entities. One of the reasons for selecting a strict hierarchy is that it is easier to maintain than a lattice structure. A node in a hierarchy will have to maintain just two references: one to its container and another to its list of constituents. On the other hand, a lattice node needs to maintain a reference to every node to which it relates. Updating these references when a node is deleted is more difficult in a lattice structure than a in a hierarchy. The other reason is that the study conducted in Chapter 3 did not

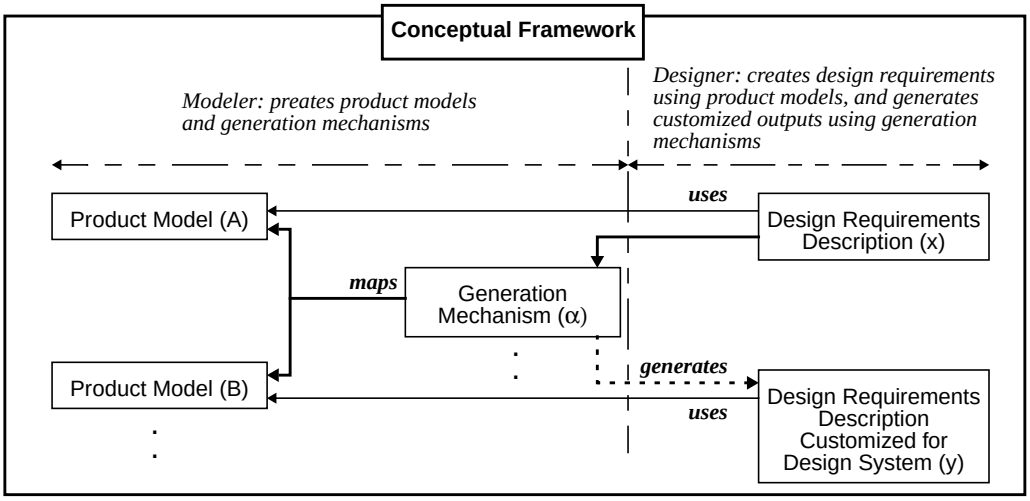


FIGURE 9. A conceptual framework to support modeling design requirements

show cases when a lattice structure is needed. The hierarchy is composed of *specification units* similar to those of SEED-Pro (section 3.2). However, the set of attributes attached to each unit is not fixed, as it is the case with SEED-Pro. Instead, it is defined according to the product model it uses. This enables the use of specification units to create design requirements based on any product model created by the framework.

A specification unit (SU) is the basic building block for representing specifications. A SU object represents an organizational entity in the design which can correspond either to a physical space or a room, or to an abstract operational concept, such as a department or a section. SUs can be recursively aggregated to form the organizational and functional hierarchies of the artifact being designed. A SU can be associated with the categories of information supplied by the product model it uses.

Using specification units, the designer can create models of design requirements and generate outputs needed by systems which interface with the framework. These outputs are also composed of specification units which use a product model designed to match the type of information needed by the interfacing system. This means that the input and output specification units will have an entirely different set of attributes, classifications and relationships, depending on the product model each group of specification units uses.

4.3.2 Product Models

Product models are used in the context of this framework to provide the information semantics used to model design requirements. Creating models of specifications requires the ability to represent the different types of information that exist in such models. The study presented in Chapter 3 indicates that these models of specifications can be represented using three categories of information. These categories are *static attributes*, *classifications* and *relationships*.

Static attributes are attributes for representing information which is specific to each SU. They can be grouped into sets where each set contains a collection of logically related attributes, such as a set of spatial attributes for specifying area, length, orientation and width requirements of a particular SU. The type of information represented by these attributes can be *numeric*, *textual*, *enumerated* or *boolean*. Numeric attributes contain information that can be expressed as numbers (integers or floating point numbers). An example is the number of employees associated with a certain SU[12], its spatial area or the desired sound level. Textual attributes handle information that needs to be expressed in plain text, such as a description of activities related to a SU, or any additional comments the designer needs to include in the design requirements. Enumerated attributes can represent types of information where only a predefined discrete set of values is allowed. An example is *orientation* which can only have values such as *east, south-east, south,* and so on[13]. Finally, boolean attributes provide a special case of enumerated attributes when the enumerated values are either "true" (yes) or "false" (no).

The framework allows for the creation of any number of these types of attributes as part of a product model. However, it does not provide means to express dependencies which we know do exist between them. This is an issue which is worth investigating, but needs to be studied in the context of possible applications and can be a subject of a future research effort.

Classifications are entities which can be used to attach typological information to a SU according to certain criteria. For example, under a *space usage* criterion, a SU can be classified as an *office*, a *meeting room* or a *classroom*. The framework allows for the creation of any number of these classification categories (or groups), each containing a set of *classifiers* which the user defined as well. A SU can have only one classification from each category, but can have multiple classifications as long as each belong to a different category. This feature implies that classifier categories have to be mutually exclusive. Assigning a classifier from a certain category to a SU should not prevent assigning another classifier belonging to another category to the same SU or invalidate any pervious assignments. The issue of classification dependencies can be a future research direction as well.

These classifications are often used by the systems which interface with the framework. SEED-Layout, for example, requires an input of spatial functional units (section 3.2) defined as rooms, zones, floors and so on. However, classifications have two purposes within the framework context. The first is to provide rules of composition to maintain the integrity of the SU hierarchies being created. These rules of composition specify the permissible classifications for the container and constituents of a SU classified in a certain way. For example, a SU classified as an *office* is not allowed to contain another of the same classification but can contain one classified as a *private bathroom*. The framework allows for specifying these constraints for every classifier being created. These con-

12. Representing an organizational entity.

13. If *orientation* needs to be represented as a continuous scale from 0 to 360 degrees, a numeric attribute should be used instead.

straints do not have to be bound by the classification criteria or group in which a classifier is defined; they can refer to classifiers belonging to other groups as well. The second purpose for classifications within the framework context is to provide specialized generation mechanisms, which only apply to specifications units that belong to a certain classification. This property is explained in detail in section 4.3.3.

Relations bind two specification units with a qualified relationship. The meaning of the relationship is determined by the *type* to which it belongs. The framework allows for the definition of any number of *relationship types* within a product model. These relationship types determine the kinds of relationships which can be specified between specification units that use a certain product model. An example of such a relationship type is adjacency, which defines a relationship that requires the spatial representation of two specification units to share a certain edge overlap.

4.3.3 Generation (mapping) Mechanisms

Generation mechanisms are needed to create the outputs of hierarchically organized data needed by systems which interface with the framework. Generation mechanisms are represented in the framework as a set of rules (created by system users) which specify the basis on which the attribute values, classifications and relationships of the output units are set, and how many output units to create from a single input unit[14]. A generation mechanism stores these rules and the two product models on which they apply. One of these product models is considered as the *input* model of design requirements and the other as the *output* model needed by a certain design system. Once a SU hierarchy is defined according to the input model, the appropriate generation mechanism can be applied to create an output of the desired type. A generation mechanism generates an unstructured list of output units from a given input SU hierarchy[15]. The user can then structure this list into a hierarchy if needed, or change the attribute values, classifications, or relationships generated by the mechanism used.

The framework allows for the definition of two sets of generation rules or mappings: a *default* set which applies to all specification units on the input side, and a *classification-based* set which applies only to units with a certain classification. For example, an input unit classified as a *bathroom* uses the bathroom classification mapping rules to create output units. In doing that, the number of the output units are determined by a formula which divides the estimated maximum number of people (during peak usage) by the maximum number of stalls allowed per bathroom unit, multiplied by a certain factor. On the other hand, an input unit classified as a *private office* uses the default mapping rules which determine the number of output units as the number of people multiplied

14. A form of generation where several input units collaborate on creating one or more output units may be needed. This is an issue that has not been addressed within the context of this research, but could be studied in a future research effort.

15. Translating the structure of the information from one product model to another is being addressed in a parallel research effort by another student in School of Architecture at Carnegie Mellon University.

by the area per person, unless a different mapping for the *private office* classification exists. To support such a scenario, the framework provides mappings for three categories of information: attributes, classifications, and relationships.

Attribute mappings provide means to set an attribute value for a newly generated SU based on the value of one or more attributes of the input SU. Such a mapping can be either *direct* or *indirect*. Direct mapping is the case when the value of an input unit attribute is transferred directly to an output unit attribute of the same type. Such is the case when the name of the input unit is copied directly to the name attribute of the output unit. Indirect mapping allows for using several attributes of the input SU to calculate the value of an output SU attribute. This type of mapping works only for numeric attributes (integer or floating point attributes), which means that the attributes of the input SU taking part in that mapping and the attribute of the output SU being set, all have to be numeric.

Indirect mapping is done using a formula which the user creates. The elements of the formula are attribute names, numbers and the mathematical operators "+" (for addition), "-" for subtraction, "*" for multiplication, and " / " for division. A formula can contain parenthesis to provide scoping of calculations. In addition to these elements, a formula can contain a special symbol which refers to the number of the output units being generated from an input SU. The reason for that symbol is that some calculations, such as area, can be based on the number of generated units. The designer might want to have the input unit acting as a specifier of an activity which requires its overall area to be divided by the number of output units generated. When the special symbol is included in a formula, the system retrieves the number of output units and uses that in the calculation specified by the user.

Classification mapping performs two functions. The first is to set the classifications of the output units according to that of the input unit from which they were generated. An example is creating output units classified as *rooms* from input units classified as *private offices*. The second function is to provide specialized attribute mappings for input units of a certain classification. The framework uses these specialized attribute mappings whenever the classification mapping within which they are contained is applicable. If a classification mapping does not contain any specialized mappings, it performs the type mapping (e. g., *private office* to *room*) then uses the default mappings for mapping attributes. The specialized mappings are regular attribute mappings that are only accessible through a classification mapping mechanism. In addition, classification mapping provides means to specify the number of output units to be generated using the same type of formula described earlier.

Relation mapping creates relationships between output units when relationships exist between their corresponding input units. Relationships are mapped after output units are generated. The system finds whether the input unit has relationships with other units in the input model. If it finds relationships, it checks if the units that form the other part of the relationship have output units generated, and creates relationships of the type specified by the mapping which connects the units generated by input units to

each other. Mapping relationships are optional; the user can specify whether relationships should be mapped or not.

CHAPTER 5

A Framework for Modeling and Manipulating Design Requirements

In the previous chapter, I presented a model for modeling and manipulating design requirements. It is based on the idea of a conceptual framework which consists of a number of concepts that can be used to structure design requirements based on different product models, and to generate specialized inputs for various design systems. This chapter describes the implementation of a software framework which realizes that model for the purpose of creating design requirements.

The software framework is designed and implemented as an object-oriented system. Its design and implementation concepts are described in this chapter using metapatterns [Pree 95] whenever suitable. Design metapatterns provide a way to capture and describe the design of a software framework on an abstraction level higher than the underlying programming language. I end the chapter by expressing the features of the framework that support the needed flexibility, which has been already been described in Chapter 4.

5.1 Framework Design

The design requirements modeling framework is a collection of software components that provide the infrastructure and mechanisms needed to create models of design requirements. It implements and realizes the idea of the framework, described in Chapter 4 as a running system with a graphical user interface. There are two main design objectives in developing the framework. The first is to provide the degree of flexibility needed to create and manipulate design requirements, define product models and generation mechanisms, and structure generated outputs independently from the design requirements description. The other objective is to be able to perform all these tasks without the need to re-program and re-compile the system. These objectives are accomplished by the implementation described in this chapter.

5.1.1 Overview

The framework design consists of three interconnected object models. An object model to support the ①definition of product models, one for ②creating generation mechanisms and another to ③create and manipulate design requirements. In addition to these three models, which all support domain operations, the framework contains another object model which ④provides a graphical user interface for using the framework. The design of the framework is modeled according to the model-view-controller (MVC) concept [Krasner et al. 88], where the domain and the interface are two separate entities. Interface elements maintain direct connections to the domain objects they represent to allow user manipulation. On the other hand, domain objects maintain indirect relationships to their corresponding interface objects to provide them with "change" notifications and updates [Figure 10]. This indirect relationship is provided using the *observer* pattern. This pattern "defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [Gamma et al. 95, pp. 293]. The observer pattern is used to maintain the independence of the domain objects from their interface counterparts, so that a change in the interface design would not require the domain objects to change as well.

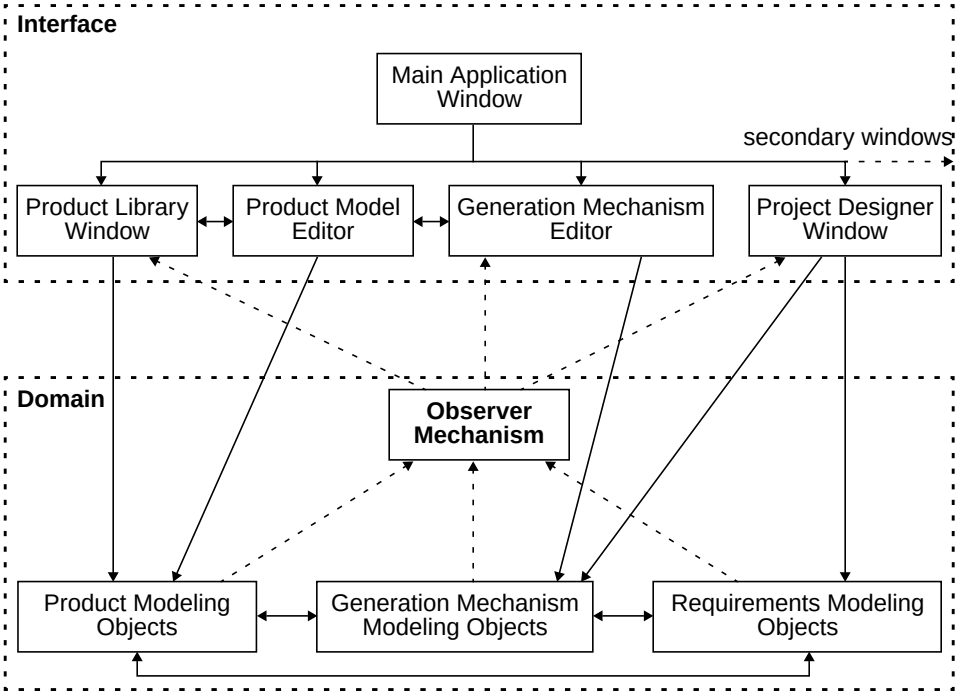


FIGURE 10. The overall system architecture of the framework

5.1.2 Representing Product Models

The framework represents the concept of a product model (explained in section 4.3.2) as the group of classes shown in [Figure 11] . The ProductModel[16] class represents the prod-

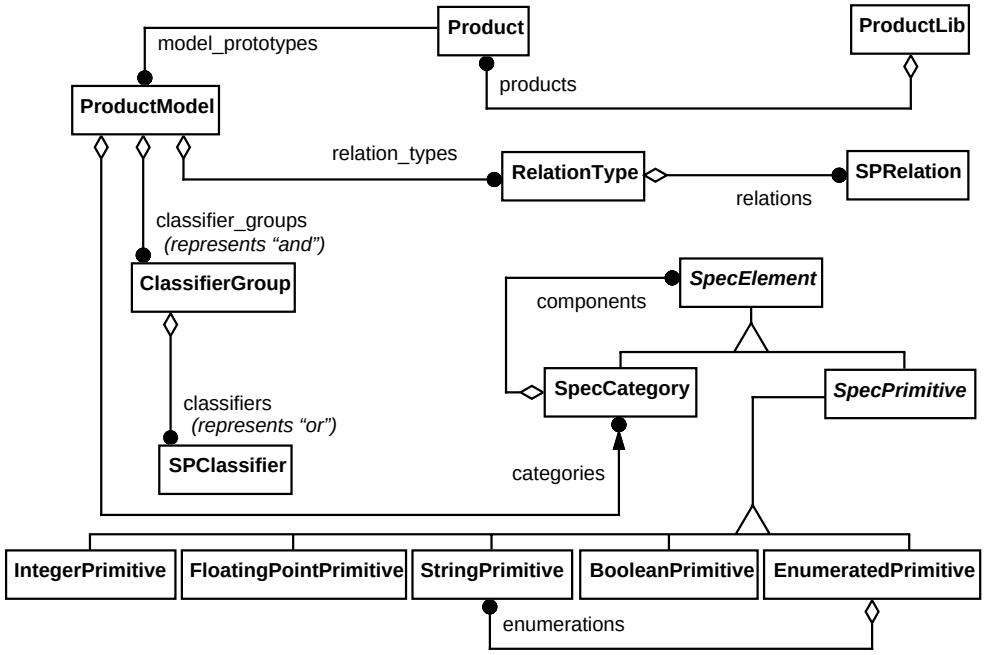


FIGURE 11. The product modeling object model using the OMT notation

uct model concept and contains the classes which represent its components. These are ClassifierGroup and SPClassifer (for classification), RelationType and SPRelation (for relationships) and SpecElement and its subclasses (for specification categories and static attributes). The ProductModel class maintains one-to-many relationships with RelationType, ClassifierGroup, and SpecCategory (SpecCategory represents the container in the 1:N Recursive Connection metapattern described later [Figure 12]). These relationships allow an instance of ProductModel to contain any number of instances of these classes. These instances become, effectively, a set of dynamic attributes for the ProductModel object in which they are contained. Such a feature allows the definition of product models at runtime using compositions of relationship types, classifier groups and specification categories.

Two more classes are defined in this object model. The Product class (for product) is bound to ProductModel with a one-to-many relationship. This allows a Product instance to contain multiple ProductModel instances as alternative or complementary specification sets that can be used to describe a product. The other class is ProductLib (for product

16. The type face Helvetica Narrow is used to denote class names, class attributes and methods.

library), which allows instances of Product to be stored in libraries of related products, such as a library for products related to the construction industry.

A RelationType instance can contain instances of SPRelation, which defines a relationship between two specification units. However, SPRelation objects cannot be created directly in a product model definition because specification units are not defined within its context[17]. An instance of SPRelation can represent any type of relationship between two specification units. It consists of references to two SpecUnit (for specification unit) objects, and a numeric value. The value can be used to qualify a relationship as in the case with *adjacency* relationships where the value can indicate the minimum overlap between two units in a layout. The meaning of the relationship is determined by the RelationType object in which it is contained.

The classification concept, introduced in section 4.3.2, is represented by two object classes: ClassifierGroup and SPClassifier. The ClassifierGroup class provides a grouping of SPClassifier objects, where only one SPClassifier is valid at a time. A SpecUnit instance can have more than one SPClassifier instance attached to it as long as they belong to different ClassifierGroup instances. The SPClassifier class provides means to specify rules of composition for a specification unit hierarchy. This is achieved by storing, in a SPClassifier instance, the names of permissible classifications of the container and constituents of the specification unit to which it is attached.

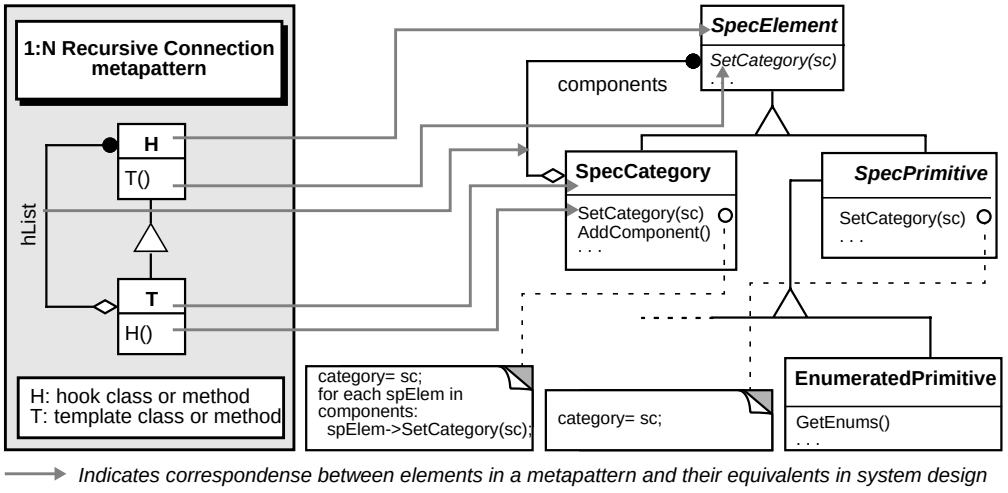


FIGURE 12. The use of 1:N recursive connection metapattern in modeling specification elements

The objects that form the static attributes of a specification unit are represented as instances of the subclasses of SpecPrimitive (these are: IntergerPrimitive, FloatingPointPrimitive, StringPrimitive, BooleanPrimitive and EnumeratedPrimitive). Any number of such instances can be

17. Creation of SPRelation objects is explained later in this section.

grouped into a category represented by the SpecCategory class. SpecPrimitive and SpecCategory are subclasses of SpecElement [Figure 11]. The structure of the these three classes conform to the 1:N recursive connection metapattern described in [Pree 95, pp. 162]. The use of this pattern allows the recursive composition of SpecElement instances as components of SpecCategory. It also provides *template* methods identified in SpecElement that have different implementations (known as *hooks*) in its subclasses according to their roles in the composition. An example of such methods is SetCategory(), which is defined as a template method in SpecElement, while its implementation is defined as hook methods in the subclasses [Figure 12].

5.1.3 Representing Generation Mechanisms

A generation mechanism provides rules for creating customized outputs from a design requirements description. These rules generate the specification units that constitute the output, then it sets their properties based on their input counterparts. The framework supports the creation of generation mechanisms through the object model shown in [Figure 13]. The GenMechanism class (for generation mechanism) provides the overall

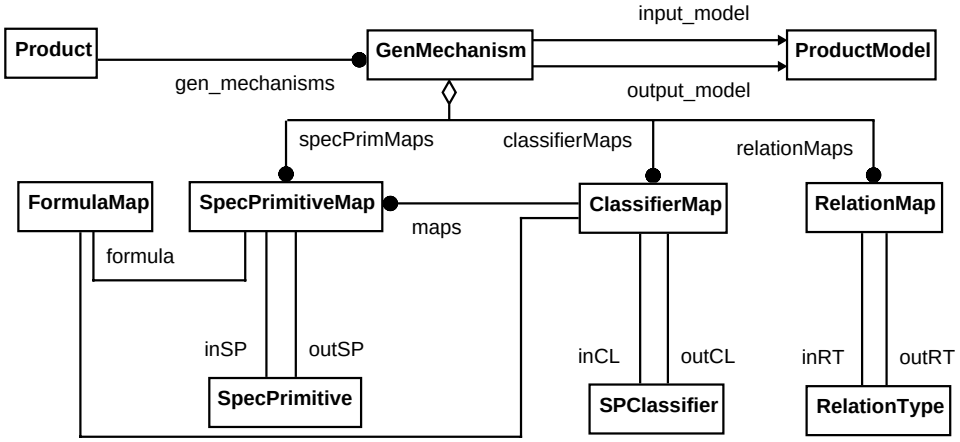


FIGURE 13. The generation mechanism object model using the OMT notation

structure where rules can be created and stored. An instance of GenMechanism relates to two ProductModel instances, where one acts as the *input* model and the other as the *output* model. Instances of GenMechanism are contained inside the Product instance which contains the input and the output models, and possibly other models as well. The framework supports a different mapping technique for each of the three categories of information associated with a product model: specification categories and primitives, classifications and relationships. Each mapping technique is encapsulated in a class that provides associations to the relevant type of information.

Attribute mapping: Mapping attributes (described in section 4.3.3) is implemented using the SpecPrimitiveMap class [Figure 13]. A SpecPrimitiveMap instance maintains a reference (inSP) to a SpecPrimitive instance from the input ProductModel and another reference

(outSP) to a SpecPrimitive instance from the output ProductModel. In the case of direct mapping [section 4.3.3], the value of the input SU attribute matching the primitive referenced by inSP (of a SpecPrimitiveMap object) would be mapped to the output SU attribute matching the primitive referenced by outSP. To perform indirect mapping [section 4.3.3], a SpecPrimitiveMap instance uses a FormulaMap instance which stores a formula that can perform the desired indirect mapping. If a FormulaMap instance is specified for a SpecPrimitiveMap instance, the inSP reference loses significance, because a formula uses its own references to its relevant primitives, which can include the one referenced by inSP as well as other SpecPrimitive objects that exist in the input ProductModel.

A FormulaMap object stores a formula in the form of string of text conforming to the description provided in section 4.3.3. The FormulaMap class includes a mechanism for parsing the formula string, which I have Implemented, based on the recursive descent parsing technique [Aho & Ullman 72] to allow scoping of calculations using parenthesis. The parser substitutes references to primitives in the input ProductModel with their actual values, and stores them and the operators that exist in the formula in two separate arrays. Whenever the parser encounters a nested formula (delineated by *open* "(" and *close* ")" parenthesis), it recursively parses it and replaces it with its computed value. The structure and behavior of the relationship between SpecPrimitiveMap and FormulaMap classes conform to the 1:1 connection metapattern described in [Pree 95, pp. 161].

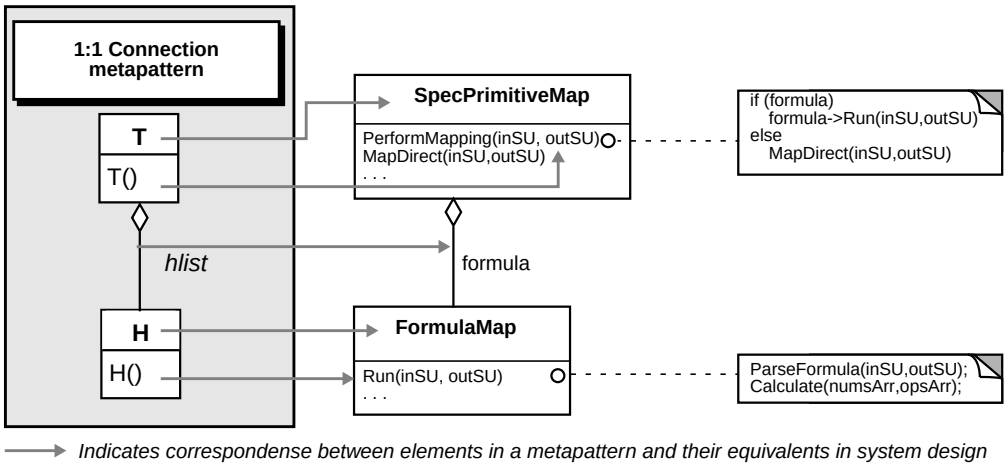


FIGURE 14. The use of 1:1 connection metapattern in modeling attribute mappings

As shown in [Figure 14], if a FormulaMap object is attached to a SpecPrimitiveMap object, the mapping is delegated to the FormulaMap object by calling its Run method. Run parses the formula creating an array of numbers and another of mathematical operators, calculates the formula and returns a numeric value. However, if no formula is attached to the SpecPrimitiveMap object at hand, a direct mapping is performed. First, the attributes matching the types referenced by inSP and outSP are retrieved from the input and output specification units. Then the value of the input SU attribute is copied to that of the output SU attribute.

Classification mapping: The ClassifierMap class enables mapping of classifiers (represented as SPClassifier objects) from a product model to another. A ClassifierMap instance maintains a reference (inCL) to a SPClassifier instance from the input ProductModel and another reference (outCL) to a SPClassifier instance from the output ProductModel. It also maintains a reference to a FormulaMap object which contains a formula for calculating the number of specification units to be created from an input SU. This formula is similar to the one used to map attribute values, and can contain references to attribute names, numbers and operators as well. The only difference is that its value is interpreted as the number of output specification units to be created.

Specialized mappings (described in section 4.3.3) are implemented in the form of a collection of SpecPrimitiveMap instances connected to the ClassifierMap instance at hand [Figure 13]. Once such a collection is defined for a ClassifierMap instance, it is used for attribute mapping for any SU with a classification matching that referenced by inCL. On the other hand if that collection is not defined, the collection of SpecPrimitiveMap instances (specPrimMaps) located directly under the GenMechanism object is used instead. The overall structure and behavior of the generation mechanism object model conforms to a series of 1:N connection metapatterns described in [Pree 95, pp. 161]. As shown in the

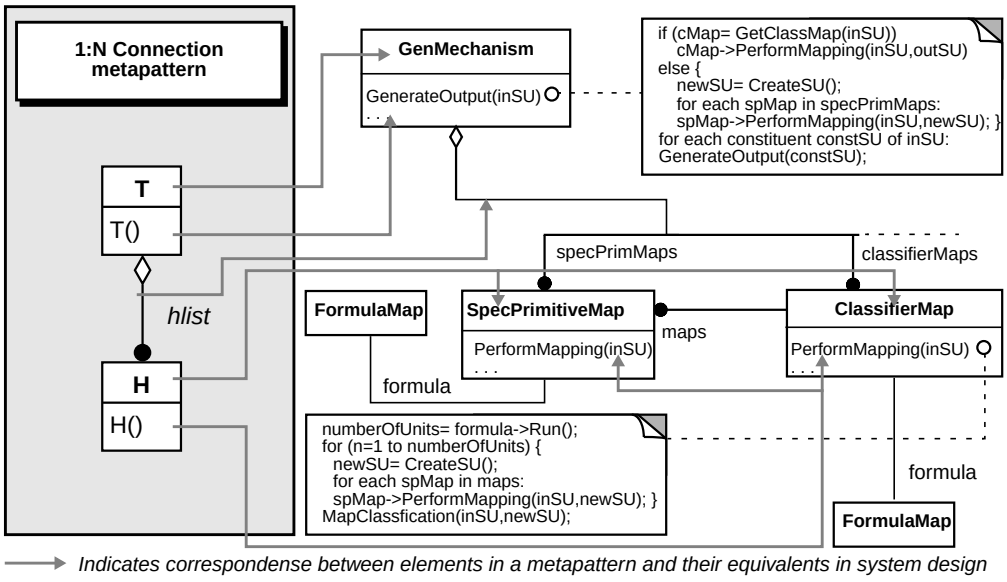


FIGURE 15. The use of 1:N connection metapattern in modeling generation mechanisms

instance diagram [Figure 15], when a GenMechanism object receives a request to generate an output from an input SU, it tries to find a corresponding ClassifierMap object in its classifierMaps list. If it finds one, it delegates the process of creating the output entirely to the ClassifierMap object. The ClassifierMap object calculates the number of output units to be created using its attached formula (if a formula is not attached it assumes the number to be one). It then creates the appropriate number of units, mapping their values using its attached list of SpecPrimitiveMap instances (maps) in a manner identical to the one shown

in [Figure 14]. It then sets the classification of each generated unit according to the mapping rule specified in the ClassifierMap object.

On the other hand, if the GenMechanism object doesn't find an appropriate ClassifierMap object for the input SU, it creates one output unit and performs the mapping as a 1:1 connection metapattern [Figure 14] using its list of SpecPrimitiveMap objects. In essence, each of the two 1:N connection metapatterns (**One**[GenMechanism]->**Many**[SpecPrimitiveMap] and **One**[GenMechanism]->**Many**[ClassifierMap] is reduced to a series of 1:1 connection metapatterns of **One**[SpecPrimitiveMap]->**One**[FormulaMap]. When all output units of a given input SU are generated and their attribute values and classifications mapped, the same operation is applied recursively to the constituents of the input SU.

Relation mapping: The RelationMap class is used to map relationships from the input SU hierarchy to the output SU hierarchy being created. A RelationMap object has a reference (inRT) to a RelationType object which exists in the input ProductModel and another (outRT) to a RelationType object from the output ProductModel. It uses these two references to create SPRelation objects for the output SU hierarchy, of the type specified by the outRT reference. Relationships are mapped according to the following scenario:
If:

- two specification units, (SU1 &SU2) from the input SU hierarchy have output units generated from them in an output SU hierarchy, and
- they are bound by a relationship of a certain type, and
- a RelationMap object exists whose inRT reference matches that type of relationship,

Then:

- create SPRelation objects, of the type specified by the inRT reference of the RelationMap object, that binds each unit generated from SU1 to these generated from SU2.

5.1.4 Representing Design Requirements

The framework represents design requirements as a hierarchy of SpecUnit (for specification unit) objects. A SpecUnit contains other units as constituents (using the one-to-many constituents relationship) to enable the creation of the hierarchy [Figure 16]. The static attributes of a SpecUnit object are created dynamically as set of SpecCategory objects. Each SpecCategory instance contains a group of SpecElement objects as explained earlier [Figure 12]. A SpecUnit object can reference a number of SPClassifier objects that provide classification information. This information is used to constrain the classification of the container and the constituents of the SpecUnit object when forming the SU hierarchy as stated earlier in section 5.1.2. The type of SPClassifier and SpecCategory objects a SpecUnit can reference and contain is determined by the ProductModel object to which it is related. The relationship between a SpecUnit and its ProductModel is handled through the ProductConstruct object, which handles a number of functions. First, it contains the top-level SpecUnit objects in a hierarchy. Second, it maintains the connection to the active ProductModel, which determines the SpecCategory and SPClassifier objects associated with a SpecUnit. Using that connection it handles the creation of SpecUnit objects; ensuring the

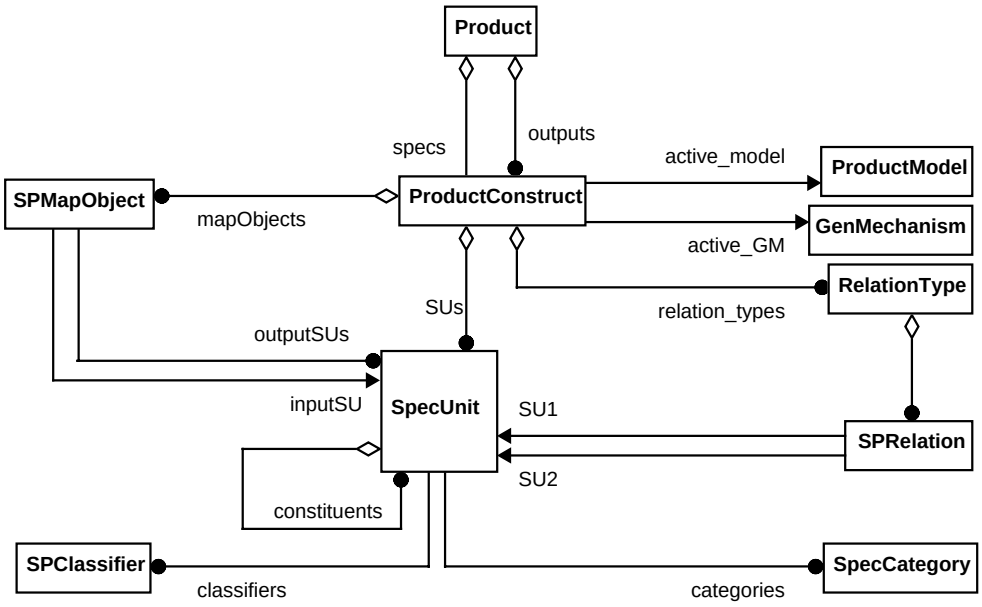


FIGURE 16. Design requirements object model using the OMT notation.

each SpecUnit created has the correct set of SpecCategory objects, and the access to the appropriate set of SPClassifier objects. Third, the ProductConstruct provides a reference to the GenMechanism object which determines how to create output SU hierarchies from the one contained inside the ProductConstruct. Fourth, it maintains the SPRelation objects related to its SU hierarchy. Finally, it manages the collection of SPMapObject instances. Each SPMapObject provides a connection between a SpecUnit object that exists in the input hierarchy and the ones in an output SU hierarchy that were generated from it. The collection of SPMapObject instances can be used by generation mechanisms to map relationships between hierarchies, and provide updates to output units when changes occur to their input counterparts[18].

A ProductConstruct object also provides a way to relate an input SU hierarchy to the multiple output hierarchies generated from it using different generation mechanisms. Each SU hierarchy is contained inside a ProductConstruct, which is contained inside a Product object [Figure 16]. Here, the Product class plays an additional role to the ones it played in the creation of product models and generation mechanisms. It relates an input ProductConstruct object containing the main design requirements description of a product in the form of a SU hierarchy to other ProductConstruct objects, each containing a set of requirements generated from the input set, and customized for a specific design system.

18. Updating the value of output units was not been implemented in this prototype. However, the infrastructure needed to support it is implemented and used to map relationships as shown in the next chapter.

5.2 Achieving Flexibility in the Framework Design

The framework design, presented in the previous section, provides the degree of flexibility needed to create and manipulate design requirements, define product models and generation mechanisms, and structure generated outputs independently from the design requirements description. Such flexibility allowed all these tasks to be performed without the need to re-program the system. To achieve the needed flexibility, certain software development strategies were employed in designing the framework. These strategies are highlighted in this section.

5.2.1 Object Compositions vs. Static Attributes

One of the main design objectives was to allow a specification unit to have varying attributes, classifications, and relationship types according to the product model it uses. To enable such a feature, these properties were not defined as regular static class members for the SpecUnit class, which represents specification units. Instead, they were defined as a collection of objects that gets attached to a SpecUnit instance at run-time. Similar object compositions were used throughout the design of the framework for object properties that need to be defined dynamically. In addition, object compositions allow the delegation of behavior which was used extensively in representing product models (section 5.1.2) and generation mechanisms (section 5.1.3).

5.2.2 Prototype-Based Object Creation

Attributes, classifications and relationship types used by a SpecUnit instance are defined according to the product model it uses. Normally, these attributes, classifications, and relationship types would be classes of objects which can be instantiated, as is the case with SEED-Pro [Figure 7 on page 31]. Adding a new property to specification units in SEED-Pro involves defining a new class or adding attributes and methods to one of its ComponentSpecs classes. This clearly requires some programming effort, and has to be done by people who have programming, as well as domain experience. Furthermore, some programming languages, such as C++, do not allow introducing new classes or modifying existing ones at run-time. This requires the appropriate programming environment to be available for compiling the system every time a new property is introduced.

To avoid such complexities, the framework employs prototype-based object creation when instantiating attributes, classifications, and relationship types for a SpecUnit instance. Using the Prototype pattern, described in [Gamma et al. 95, pp. 117], a ProductModel object specifies the kinds of objects to create using prototypical instances, and creates new objects by copying these prototypes to each SpecUnit object being created. These prototypical instances are created initially, using object composition, as the set of SpecElement objects during the process of defining a product model (described in section 5.1.2). However, for the prototype-based object creation to work, these prototypical instances are required to have unique names.

5.2.3 Separating Generation Mechanisms from Product Models

Creating an output from a set of design requirements is the result of an operation or a set of operations applied to the design requirements object structure. The framework

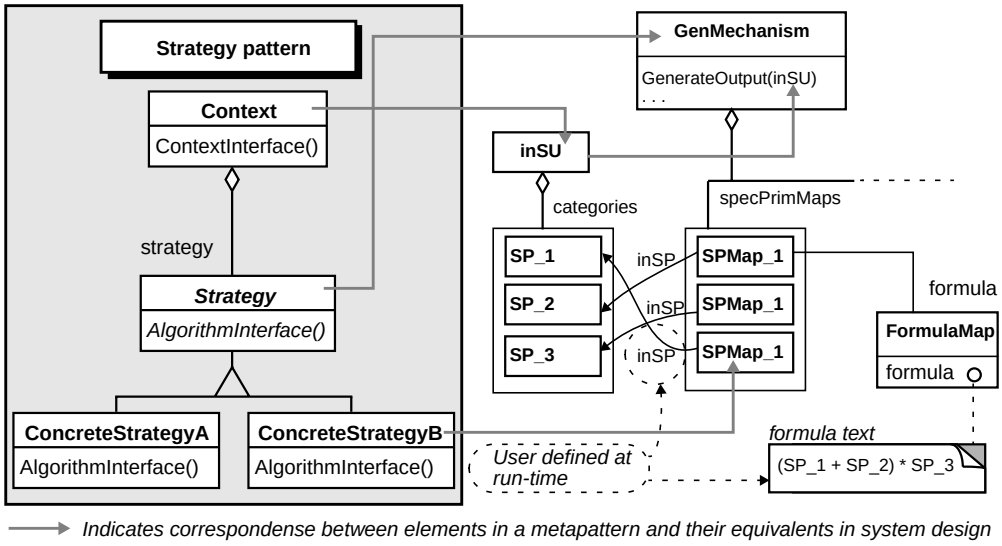


FIGURE 17. Using a variation of the Strategy pattern to create flexible generation mechanisms

employs a variation of the Strategy pattern described in [Gamma et al. 95, pp. 315] to achieve a flexible way of defining these operations. This pattern defines a family of encapsulated algorithms, so that they can vary independently from the objects that use the pattern. The pattern used in the framework differs from the Strategy pattern in the sense that is based on composition instead of inheritance [Figure 17]. It also augments the Strategy pattern by allowing the algorithms to be created and modified at run-time. As explained earlier (section 5.1.3), the operations needed to perform mappings are encapsulated in the SpecPrimitiveMap, ClassifierMap and RelationMap classes. The user can control the type of information an instance of these classes use, by setting its input reference (such as, inSP for SpecPrimitiveMap). The user can also control *how* the mapping takes place for SpecPrimitiveMap and ClassifierMap instances by defining the mapping formulae they use at run-time. Defining such a formula (as explained in section 5.1.3) effectively controls the way the mapping algorithm works. At its present state, the framework allows such formulae to be defined for mappings that involve numbers only. Investigating similar techniques for other types of mappings can be the subject of a future research effort.

5.2.4 User Interface

The framework graphical user interface was designed to accommodate and express the flexibility provided by the domain representation. Interface elements were selected according to the type of information they represent and the function they provide. Object compositions are represented using a tree metaphor to allow the viewing of

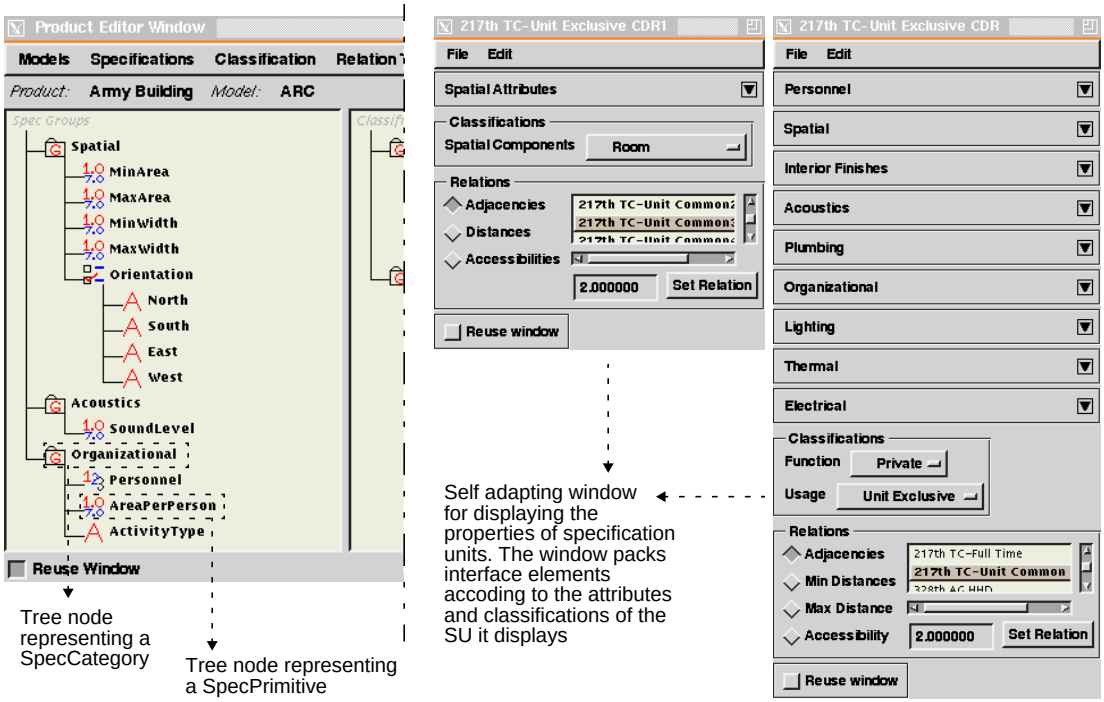


FIGURE 18. Examples of the framework GUI showing its adaptability

entire compositions of objects represented as tree nodes [Figure 18]. They also support the actions the user needs to perform in editing these compositions. An example of such actions is nesting nodes by dragging one that represents a constituent (such as an IntergerPrimitive instance) and dropping it onto another representing a container (such as a SpecCategory object).

Other interface elements, such as the specification unit editor window, creates interface elements according to properties of specification unit it displays. The type of interface element to use is selected to support the interaction style needed to manipulate the information it displays. For example, the window on the far left, in [Figure 18], uses simple text editing fields to display and edit numeric and textual information. At the same time, it uses pop-up buttons for cases when a selection set is available and only one selection is valid at a time, such as classifications.

5.2.5 Expanding the Framework

Some consideration was given the capabilities that allow expanding the framework to enhance its functionality and scope, especially in places where a modification seems probable. An example where such a consideration plays a role, is the way in which the SpecElement class and its sub-classes are structured as a 1:N recursive connection meta-pattern [Figure 12]. In case a new SpecPrimitive sub-class is needed, it can simply be added without any changes in the framework since all of its connections are defined at

its super-class level. The system developer will only need to specialize the methods whose implementation has to be customized for the new class.

CHAPTER 6

Using the Design Requirements Modeling Framework

The framework can be used in two modes according to the task being performed: *modeling* and *designing*. The *modeling* mode allows the user (or *modeler* in this case) to create product models and generation mechanisms and store them in libraries to be used to create design specifications. The designing mode allows the *designer* to use these product models and generation mechanisms to create design requirements for buildings (and potentially, other engineering artifacts) and generate outputs customized for other design systems. The framework is implemented as a running system with a graphical user interface called SP_II. It provides two sets of interface elements for use with each mode. This chapter illustrates using the framework for each mode through examples.

6.1 Modeling Mode

Using the framework in the modeling mode involves creating descriptions of product models that can be used in developing models of design requirements. It also involves modeling generation mechanisms, which are used to create customized outputs for other design systems. It is therefore envisioned that this task is performed by a domain expert with rigor and precision. This domain expert will be called a “modeler” throughout this section.

6.1.1 Creating a Product Model

Product models are created as alternative or complementary ways to describe a product. Products are stored in libraries which can be customized to include products that relate to each other according to some criteria which the modeler defines. The examples, presented in this section illustrate the creation of a new product library and a product in that library with two different product models. One of these product models supports the generation of design requirements for Army Reserve centers. The other supports the creation of a hierarchy of spatial building components, which contains the set of design requirements needed by SEED-Layout to generate floor layouts of buildings.

When the modeler starts the system, the *Main* window appears [Figure 19]. This win-

FIGURE 19. The Main window

dow provides access to the main system functionality for both user modes. It contains five buttons for, respectively, 1)creating a new product library, 2)opening an existing one, 3)creating a new project, 4)displaying general information about the system, and 5)exiting the system.

To create a new library, the modeler clicks on the button labeled “New Library”, which displays the *Product Libraries* window. Once the window is displayed, it asks the modeler to enter a name for the new product library to create. The modeler enters the name “Army Buildings” and the new library is displayed in the tree view which shows the loaded libraries [Figure 20, (a)].

The Product Libraries window contains the *Modeler* menu which provides the functionalities needed during the modeling mode. The modeler selects the library node, then

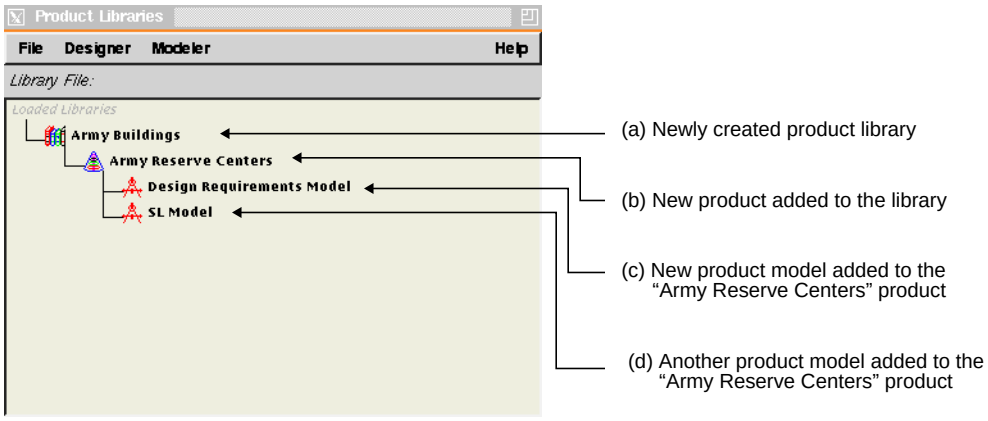


FIGURE 20. Using the Product Library window

selects the “New Product” command from the Modeler menu, which prompts the user to specify a name for the new product. After the user types in the name “Army Reserve Centers”, the system adds the new product to the selected product library and displays it as a child node for the node representing the library in the window tree view [Figure 20, (b)]. To create a product model, the modeler selects the “Army Reserve Centers” product, then selects the “New Model” command from the Modeler menu. Again the system prompts for a name. The modeler types “Design Requirements Model” and confirms. This product model will be used to create the design requirements for an

army research center building later in this chapter. The system adds the new product model to the selected product and displays the complete library-product-product model composition in the tree view [Figure 20, (c)]. Similarly, the modeler adds another product model, named “SL Model”, to the same product. This product model will be used to model the spatial output which will be generated later from the requirements. The modeler is now ready to build a description for each of the newly created product models.

As mentioned earlier (section 4.3.2), a product model description consists of specification attributes, classifications and relationship types. To create such a description, the modeler selects the “Edit Model” command from the Modeler menu of the Product Libraries window. The system displays the *Product Editor* window which contains three panels. The first panel displays the specification primitives the modeler creates, the second displays classifications, while the third displays relationship types. Naturally, all three panels will be empty when the window is opened for the first time. The modeler starts by creating specification categories which will contain the specification primitives. To create a specification category, the modeler selects the “New Group” command from the *Specifications* menu. The system prompts the modeler for a name and confirms. The system then creates the new category and displays it as a node in the specification categories window panel [Figure 21]. The modeler then selects a category, and instructs

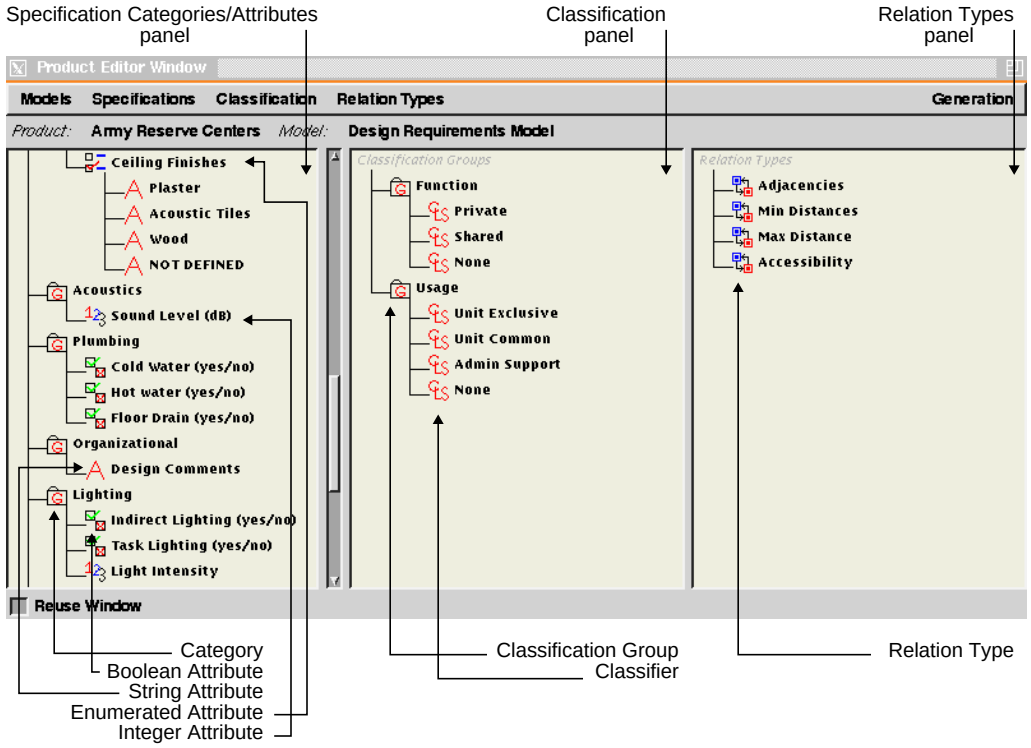


FIGURE 21. The “Design Requirements Model” product model description in the Product Editor window

the system to create a new attribute by selecting the type of the attribute to be created from the Specifications menu. The type of the attribute can be “integer” (to store integers), “floating point” (to store floating point numbers), “string” (to store text), “boolean” (to store *yes* or *no* values), and “enumerated” (to store a list of choices). The system asks the modeler to type in a name for the new attribute and displays it as a sub-node of the selected specification category node. Different types of attributes have different icons displayed next to them to indicate their type [Figure 21]. The modeler creates the specification categories and attributes needed to represent the type of information associated with the design requirements of army reserve center buildings. These include spatial, thermal, acoustic, interior finishes and other types of attributes, as well as the classifications and relationship types shown in [Figure 21].

Likewise, the modeler develops a description for the other product model (SL Model) which contains primarily spatial attributes and the classifications and relationship types shown in [Figure 22]. The modeler is now ready to define a generation mechanism

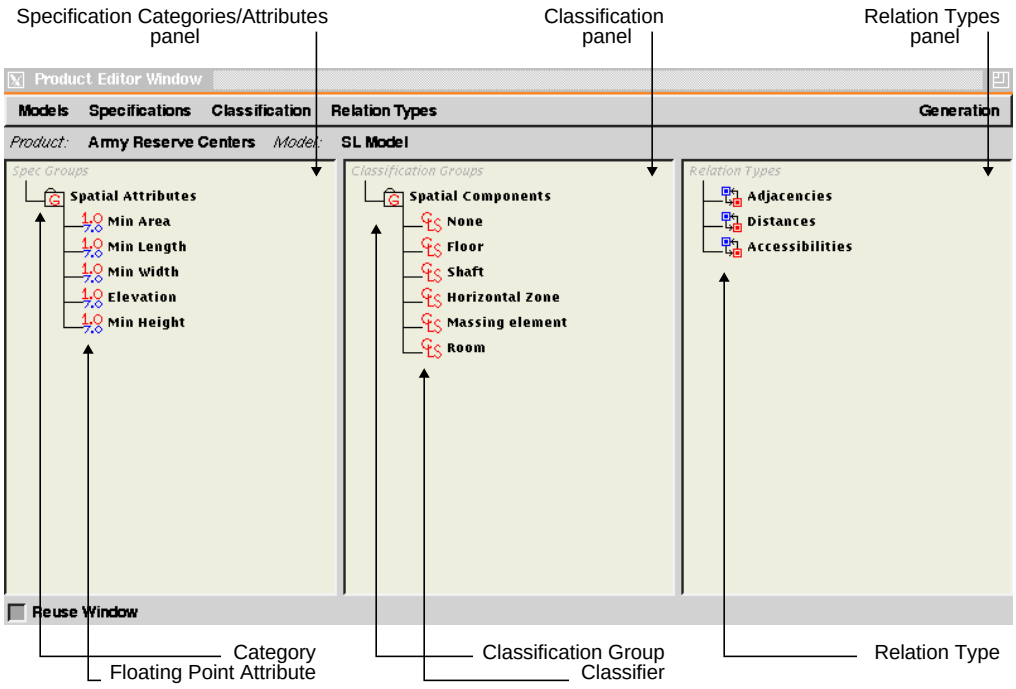


FIGURE 22. The “SL-Model” product model description in the Product Editor window.

which can create an output SU hierarchy that uses *SL-Model* as its product model, from an input hierarchy created using the *Design Requirements Model*.

6.1.2 Creating a Generation Mechanism

A generation mechanism consists of a group of mappings as described in section 4.3.3. It is created by selecting the “Create New Mechanism” command from the *Generation*

menu of the Product Editor window. The system prompts for a name then displays the Generation Manager window. Using this window, the modeler can create the mappings that form a generation mechanism. The Generation Manager window contains three main regions [Figure 23]. The first region contains three tree views for displaying the

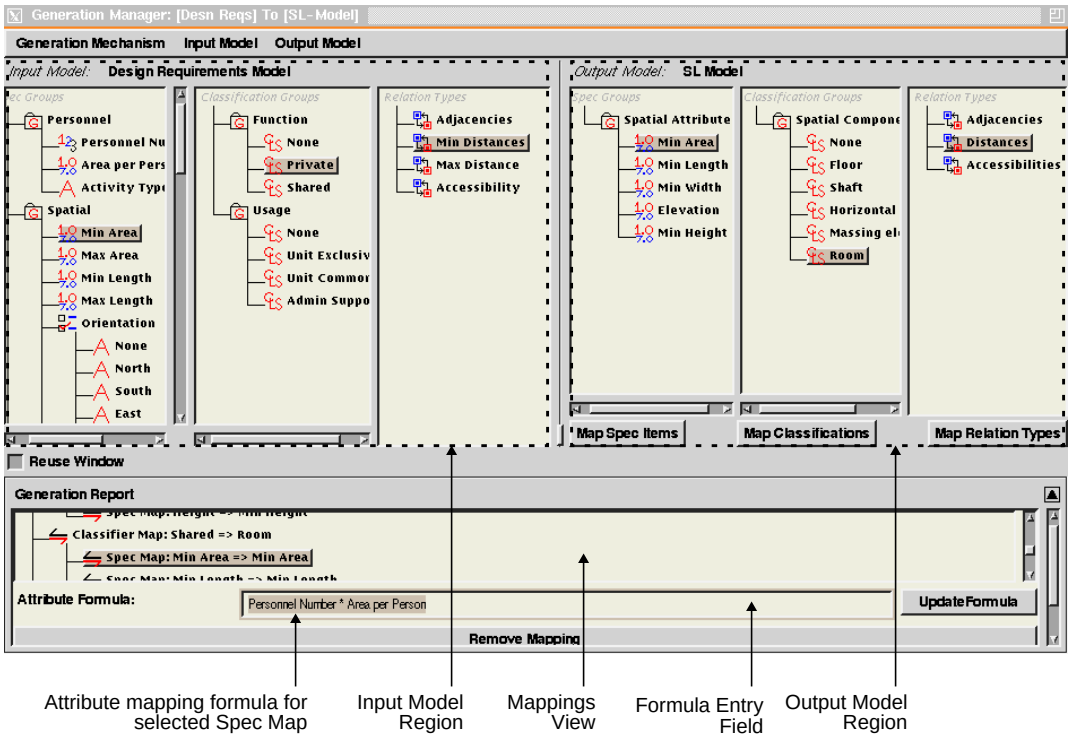


FIGURE 23. Creating a generation mechanism for mapping two product models

information categories of the input product model, while he second region displays the description of the output product model. Below the second region, are three buttons, each can be used to create a different type of mapping. The bottom region of the window contains a tree view for displaying the mappings the modeler creates, as well as a text entry field for inputting arithmetic expressions used to perform the indirect mappings explained in section 4.3.3.

The modeler starts by setting the input and output product models. This can be done by selecting the models from the “Input Model” and “Output Model” menus respectively. Once a product model is selected from the menu, its description is displayed in its corresponding region. To create an attribute (spec) mapping, the modeler selects an attribute from the input model and another from the output model. When these selections are made, and the selected attributes are of matching types, the “Map Spec Items” button, located under the attributes panel of the output model, becomes active. The modeler clicks the button, causing the system to create a mapping description that binds the two

attributes to each other. The newly created mapping is displayed in the bottom region of the window.

If it is left at that, such a mapping will be interpreted by the system as a direct mapping. This means that, when creating a new output SU, the value of its attribute which corresponds to the one referenced by the mapping object, will be set by directly copying the value of the corresponding input SU attribute. In order to perform indirect mapping, the modeler has to define a mathematical expression (formula) in the text field located at the bottom part of the Generation Manager window. An example would be to define the expression "Personnel Number * Area per Person" for the "Spec Map: Min Area=>MinArea" mapping [Figure 23]. Such an expression will cause the value of the Min Area attribute of the output unit created to be equal to the product of the Personnel Number and the Area per Person attributes of the input SU used to generate the output unit at hand. The modeler creates other mappings in a similar fashion, which will all be displayed at the bottom region of the window. The mappings created in this way become the default mappings of the current generation mechanism. They will be used to map attribute values for all input specification units that do not have specialized mapping defined according to their classifications[19].

Creating a classification mapping is similar to creating an attribute mapping. The modeler selects a classifier from the input model and another from the output model. This activates the "Map Classifications" button, located under the classification panel of the output model. The modeler clicks the button, causing the system to create a mapping description that binds the two classifiers to each other. The newly created classifier mapping is also displayed at the bottom region of the window along with the other mappings. The classifier map "Classifier Map: Shared=>Room" [Figure 24] instructs the system to

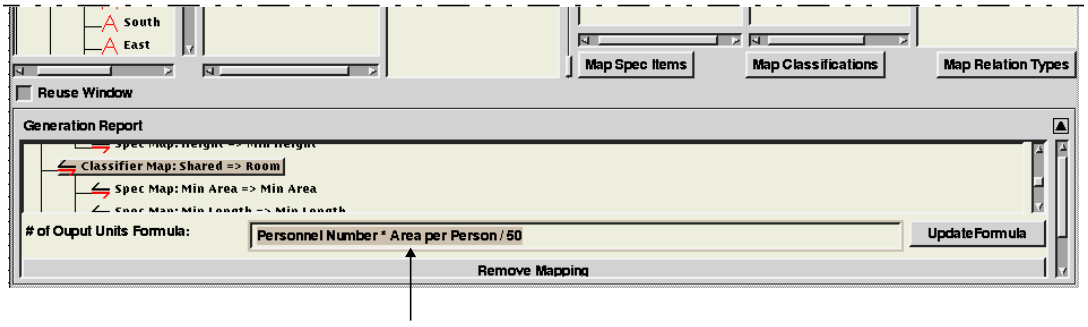


FIGURE 24. A sample expression for specifying the number of output units according to classification mapping.

create SUs classified as "Room" from input SUs classified as "Shared". A classifier map can include a formula to calculate the number of output units to be created from an SU

19. Creating specialized mapping is explained later in this section.

with a certain classification. An example is creating output units for input specification units classified as “Shared” according to the formula: “Personnel Number * Area per Person / 50” [Figure 24]. This instructs the system to multiply the number of people using an SU classified as “Shared”, by the area needed for each person. The result is then divided by 50, which represents the maximum area allowed for each space of such a classification. The final result specifies the number of output units to be created during generation.

To create specialized mappings for specification units classified as “Shared”, the modeler selects the “Classifier Map: Shared=>Room” mapping, then creates attribute mappings in the way explained earlier. However, in this case, the newly created attribute mappings will be displayed as sub-nodes of the selected classifier map node; indicating that they apply only when their container node applies. If a classifier map does not have any specialized attribute mappings, the default ones will be used. However, if it contains even one specialized mapping, none of the default mappings will be used in this case.

Mapping relationships is done in a similar way. The modeler selects a relationship type from the input model and another from the output model. This activates the “Map Relationship Types” button, located under the relationship types panel of the output model. The modeler clicks the button, causing the system to create a mapping description that binds the two relationship types to each other. The newly created relationship type mapping is also displayed at the bottom region of the window (prefixed with “Relation Map”) along with the other mappings. Relationship mapping works according to the way described in section 5.1.3.

The system now has enough information to enable the creation of a design requirements description for an army reserve center building using the “Design Requirements” product model. This description will be used to create another description based on the “SL-Model” product model, using the generation mechanism that maps the two models.

6.2 Using the Framework in the Designing Mode

Using the framework in the designing mode involves creating a description of design requirements for a certain product, and using it to generate another requirements description customized for a certain design system. It is therefore envisioned that this task is performed by an architectural programmer or a decision maker, who will be called a “designer” throughout this section.

6.2.1 Creating a Project

A *project* is an entity which can contain one or more product descriptions. The designer creates a new project by selecting the “Create New Project” command from the Designer menu of the Product Libraries window. The system prompts for a name to the new project, creates a new project and displays it in the Projects window. This window is used for displaying and manipulating the information needed during the designing mode. It contains a tree view that shows the project structure (the opened projects and the product descriptions they contain). It also contains two more views: one for display-

ing and manipulating the specification units that compose the main design requirements description of the active product, and another for displaying the outputs generated from that main description.

6.2.2 Creating a Design Requirements Description

After a project is created, the designer needs to specify the type of product to use. This is done by selecting a product from the Product Libraries window, copying and pasting it to the newly created project in the Projects window. The designer then activates the product by selecting it in the Projects window [Figure 25]. Activating a product means

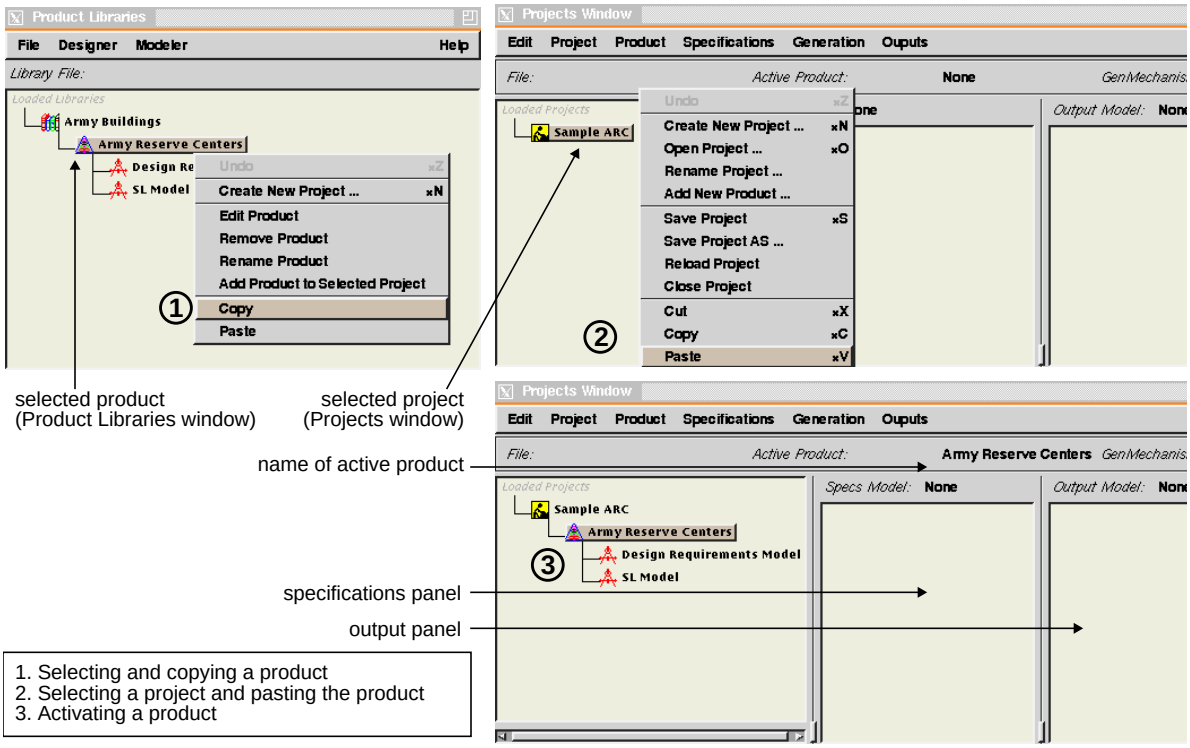


FIGURE 25. Specifying a product type for a project.

that its product models and generation mechanism become available for use, and choices in the Generation and Specification menus of the Projects window will be set accordingly. Given that a product contains several product model (in this case two), the designer has to specify which product model to be used for creating design requirements. This is done by selecting the Design Requirements Model, created earlier, then selecting the “Set Product Model” command from the Specifications menu.

The designer can now start creating the design requirements description in the form of a specification unit hierarchy. To create a specification unit, the designer selects the “Create Spec Unit” command from the specifications menu. The system prompts for a name

then displays the new unit in the specification panel of the Projects window. To add a specification unit as a constituent of another, the designer first selects a unit then creates a new one. The new unit is created and placed as a constituent of the selected unit. The designer can also drag a unit with the mouse and drop it into another unit to achieve the same result. However, the compositional constraints, imposed by the classifications of

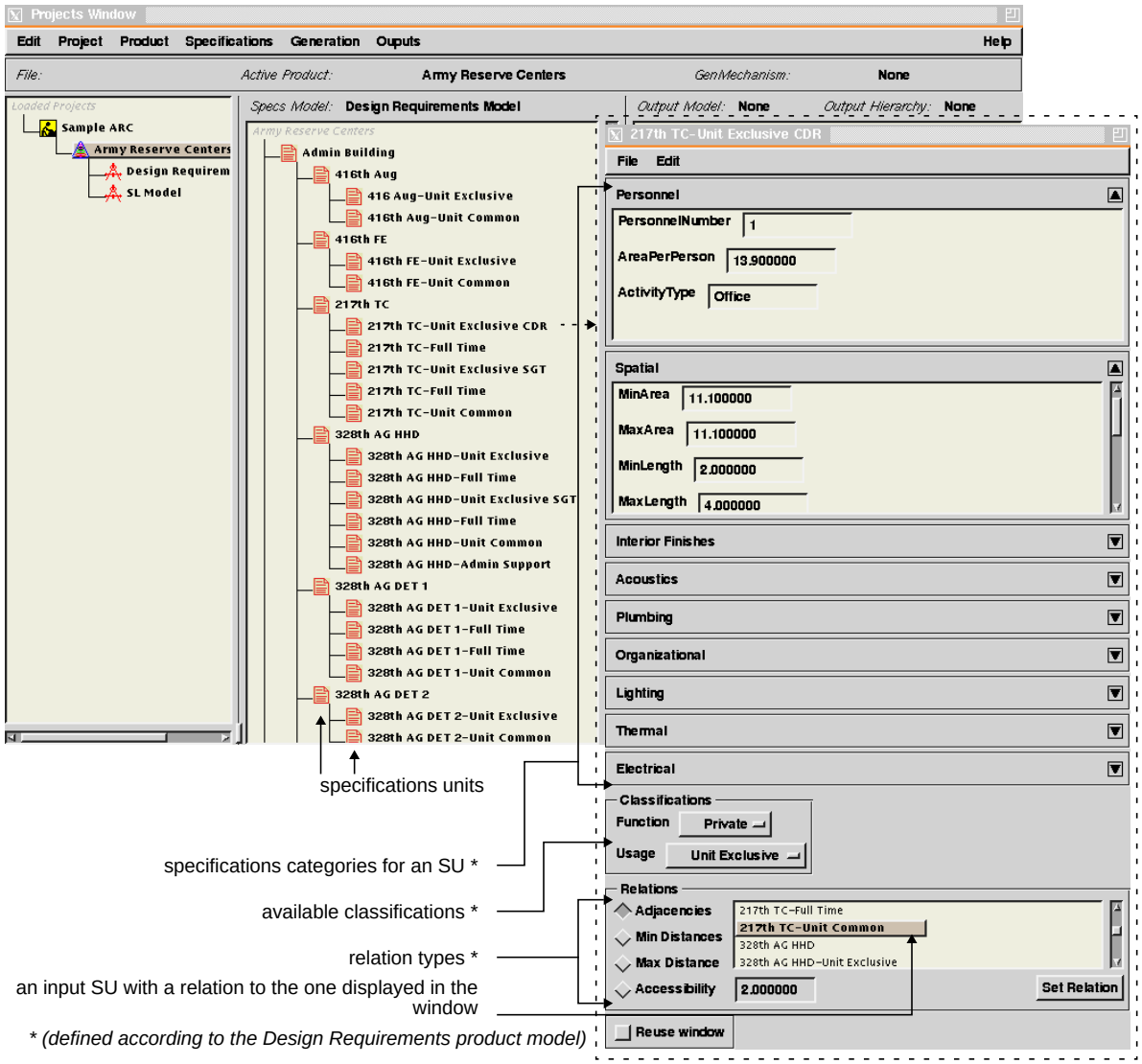


FIGURE 26. A sample design requirements description in the form of specification units.

the two units have to be satisfied in order for the drop process to be allowed[20]. The designer creates as many specification units as needed to represent the design requirements of the active product. A sample SU hierarchy for an army reserve center administrative building is shown in [Figure 26]. Each SU in the hierarchy contains a number of

attributes. These are defined according to the specification categories and attributes created earlier for the Design Requirements Model product model [Figure 21]. To edit the values of these attributes for a give SU, the designer selects the unit, and the “Edit Spec Unit” command from the Specifications menu. The system displays the Specification Unit Editor window, which allows the designer to enter attribute values, set classifications, and specify relationships between the current SU and others using the relationship types available in the product model used [Figure 26].

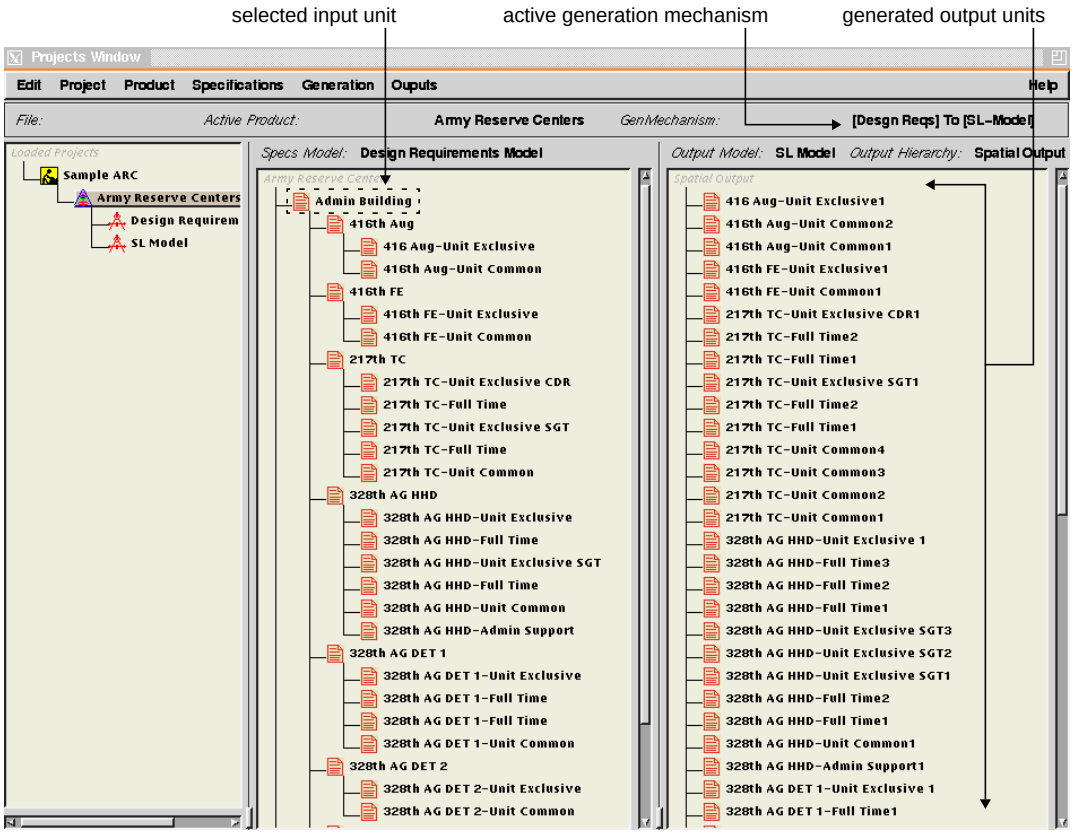


FIGURE 27. Generating a customized output from an input SU hierarchy.

6.2.3 Generating a Customized Output

When the designer finishes creating the design description, customized requirements descriptions can be created using the generation mechanisms available for the active product model. The designer starts by selecting the generation mechanism from the Generation menu of the Projects window. This will set the selected mechanism as active

20. This does not apply when new unit are created directly as constituents of others, because the new unit has no classification specified at the time of its creation.

and will display its name in the window [Figure 27]. The designer then selects a specification unit to be assigned as an input unit, then selects the “Generate Output” command from the Generation menu. The system asks the designer to enter a name for the new output to be generated, then it generates output units for the selected SU according to the mapping rules of the selected generation mechanism. The generated output units are displayed in the output panel of the Projects window[21] [Figure 27]. Each output unit generated, maintains a relationship to its corresponding input unit. This relationship can be used to map relationships as well as update the values of the output units when their input counterparts are changed [22].

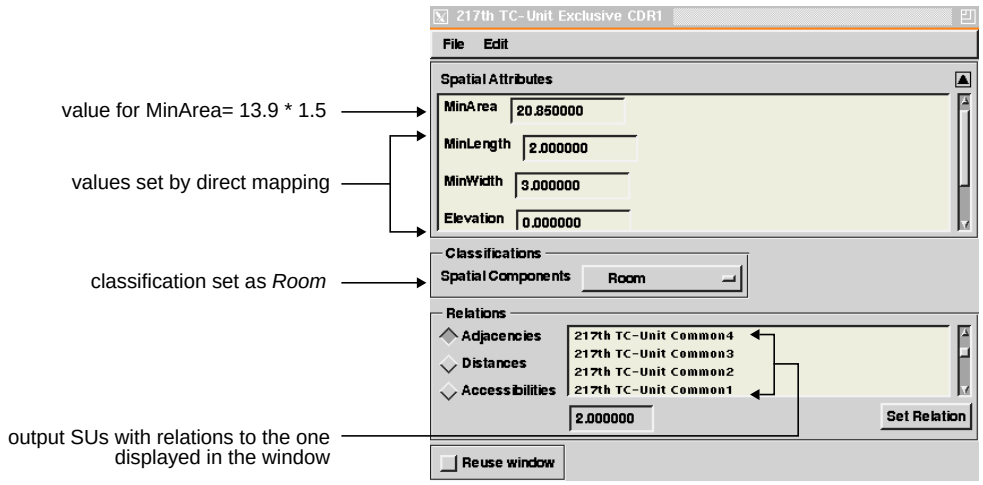


FIGURE 28. Properties of a generated output SU.

The number properties of the generated output units are determined according to their input counterparts using the mapping rules. For example, the *217th TC Unit Exclusive CDR* specification unit, in the input hierarchy, has the attribute values shown in [Figure 26]. It is classified as *Private* and *Unit Exclusive*, and has a relationship of type *Adjacencies* to the *217th TC-Unit Common* specification unit. The generation mechanism found that the mapping “Classifier Map: Private=>Room” applies since the SU is classified as *Private*. This mapping indicates the number of output units to be generated is equal to value in the PersonnelNumber attribute of the input SU (meaning that one person occupies each private unit). That caused the generation mechanism to create one output unit with the same name suffixed with the number “1” (*217th TC Unit Exclusive CDR1)*[23]. The out-

21. In this example, the top node in the input hierarchy was selected for generation. This caused the generation mechanism to be applied recursively to all of its nodes; causing the creation of units shown in the output panel in [Figure 27].
22. Updating the value of output units was not been implemented in this prototype. However, the infrastructure needed to support it is implemented and used to map relationships.
23. If more the one output unit is created the suffix is incremented for each additional unit.

put unit MinArea attribute is set according to the specialized mapping for the MinArea attribute which uses the formula “AreaPerPerson * 1.5”. This formula multiplies the value of the AreaPerPerson attribute in the input SU by 1.5, and places the result in the MinArea attribute of the output unit [Figure 28]. Other attributes are mapped via direct mapping where values are copied directly from the corresponding input attribute according to the specialized mappings defined under the “Classifier Map: Private=>Room” mapping. This mapping also caused the classification of the output unit to be set as *Room*. Finally, the generation mechanism used the mapping “Relation Map: Adjacencies=>Adjacencies”, to create relationships of type *Adjacencies* that binds the output unit generated from *217th TC Unit Exclusive CDR* to the ones generated from *217th TC-Unit Common*[24] [Figure 28].

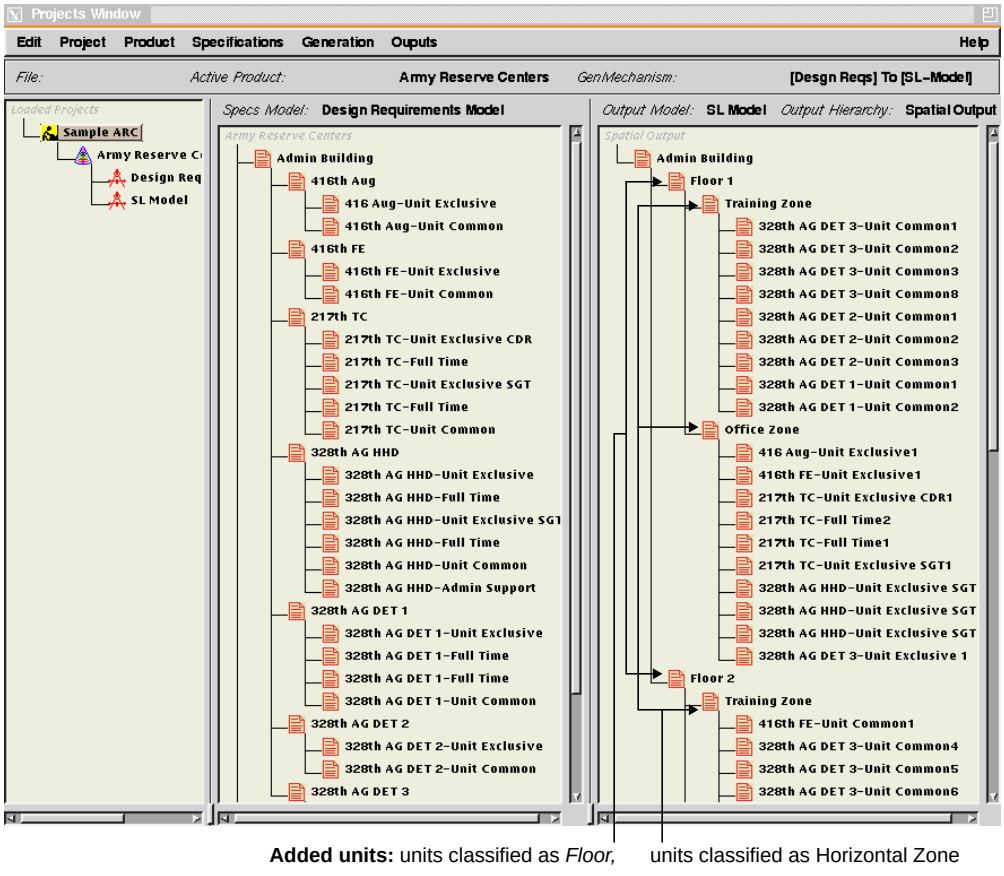


FIGURE 29. structuring an output hierarchy independently from the input one.

24. Multiple units were generated from the *217th TC-Unit Common* SU because it is classified as *Shared*. This caused a different mapping to apply which specifies the number of output units to generate as: PersonnelNumber * AreaPerPerson / 50.

The designer can now structure the generated output into a hierarchy by adding specification units to represent floors and zones and adding the generated units as their constituents [Figure 29][25].

25. An ongoing research effort is being conducted at the School of Architecture, Carnegie Mellon University to provide mechanisms for structuring requirements based on multiple criteria. See [Akin et. al., 95].

CHAPTER 7

Conclusions

This chapter contains the contributions of this research and some possible research directions that can build on what has been accomplished so far.

7.1 Contributions

During the past few years, several computational design support and simulation tools for building design have emerged, from research as well as industrial institutions. Many of these tools provide design generation and evaluation mechanisms which assist building designers to rapidly create and evaluate design alternatives. Such tools normally require a relatively large input of design requirements information, from which design representations are created and evaluated. The usability of design support and evaluation systems has been adversely affected by the lack of computable representations for creating and manipulating design requirements. Such representations can provide a repository from which the input information needed by design support and simulation systems can be generated.

The research, described in this thesis, describes a computable model of building design requirements that supports:

- the diverse nature of information typically associated with the architectural programming stage, and
- the creation of specialized representations, needed by other design support and performance simulation systems.

The approach used to achieve these research goals is a framework that contains organizing concepts and software building blocks from which representations of design requirements can be built. It builds on and augments the work accomplished in SEED-Pro (the architectural programming module of SEED) by re-engineering its information model.

This research constitutes the first effort to attempt the modeling of building design requirements using software engineering principles[26]. It makes contributions in the following areas:

7.1.1 Architectural Programming

The framework described in this document provides means to structure the unstructured domain of architectural programming using software engineering techniques. Such a structure enables the creation of architectural programs in a computable form. This provides mechanisms to experiment with different configurations of architectural programs and manage alternative compositions. It will also make architectural programs more flexible and adaptable to different situations through the reuse of past programs, either partially of fully. It will be possible to perform consistency checking and other types of evaluation mechanisms, as well as feasibility studies, on these representations. This can be done by either extending the framework to support the creation of evaluation mechanisms or by creating customized outputs for design evaluation systems using the current framework functionalities. These evaluation capabilities can provide more support for decision making at the architectural programming stage.

In addition, information captured in computable form will allow the automatic transfer of information to other design representations (used by other design systems), or can simply be printed in reports needed for feasibility studies or legal contracts and agreements. The ability to transfer information, in this manner, will allow architectural programming to be better integrated with other stages of the design process. Changes will be easily propagated back and forth between design requirements and other design representations that use them.

7.1.2 Design Requirements Modeling

The proposed framework allows for the creation of models of design requirements which can be customized according to users' preferences. This is of special importance because many design firms have developed their own models of design requirements over the years. Potentially, design firms will be able to choose between converting their models into a computable form using the proposed framework, or buying a commercial "off-the-shelf" model developed according to a certain standard.

The framework makes it possible to change or augment such models of design requirements during the process of creating architectural programs and other types of design requirements. This is enabled through a graphical user interface that enables users to dynamically create new categories of specification attributes, classifications, and relationship types which can be used to describe design requirements. These functionalities could be used to define design requirements for domains other than buildings as well. The current prototype design was based on the type of requirements associated with the design of buildings. However, the implementation is general enough to potentially support modeling design requirement for any type of product that can be described using the information categories supported by the framework.

26. To date, the core functionality of SEED-Pro is the first prototype we know where building design requirements are being modeled according to software engineering techniques.

7.1.3 SEED

The proposed research builds on, and augments, the work we have accomplished so far in SEED-Pro, the architectural programming module of the SEED project [section 3.2.3]. It provides SEED with an architectural programming module that supports the exploratory nature of the design process. Specifically, it provides the ability to experiment with different ways of modeling design requirements for buildings, as well as different ways to generate specialized requirements from specifications. This will make it possible to conduct studies which compare alternative models of design requirements and generation strategies and evaluate them in different design settings.

	Creating Product Models	Creating Generation Strategies	Creating Requirements Descriptions	Generating Customized Outputs	Structuring Outputs	Exporting to Other Modules
SEED-Pro	No Support. Needs to be programmed into the system	No Support. Needs to be programmed into the system	Supported	Supported if the output product model is already programmed	Supported in a manual mode	Supported using OML
SP_II Framework	Supported	Supported	Supported	Supported	Supported in a manual mode	Supported using OML

TABLE 1. Phases of the design requirements modeling process supported by SEED-Pro and the SP_II framework.

In that regard, the framework provides a clear enhancement over the way design requirements are modeled in SEED-Pro [Table 1]. In addition to the phases the current implementation of SEED-Pro supports, SP_II provides support for two additional phases in the process. It allows for the dynamic creation of product model descriptions at run-time, which can be used to model design requirements. It also enables the dynamic creation of generation strategies and mechanisms that provide rules for generating customized outputs from specifications. Creating a new product model or generation strategy in the current implementation of SEED-Pro requires programming these representations in C++ and incorporating them into the SEED-Pro prototype. This greatly inhibits exploration of these two phases and prevents ordinary users and architectural programming experts who lack the ability to program in C++ from contributing to the creation of product models and generation strategies.

7.2 Enhancements and Future Research Directions

The research described in this document can benefit from several enhancements and suggests directions for future research. This section provides a brief outline of these issues.

7.2.1 Creating a SRPOUT Language Generator that Supports Multiple Shared Schemata

The main reason for creating customized models of design requirements from specifications is to export these models to the design systems for which they were created. The process of exporting these models is currently accomplished in the SEED system using the Object Modeling Language (OML). OML supports the definition of a *shared schema* through which different modules can communicate. Each module then has to have its own set of *language bindings*, which translate information between the module internal representation and the shared schema.

The process of creating these language bindings is fairly straight forward in the case of SEED-Pro as there is only one output model within SEED-Pro for which language bindings must be created. On the other hand, the SP_II framework can have multiple output models, which are also created at run-time. This complicates the process of creating language bindings for these representations, as a different set of language bindings has to be created for every output product model. Furthermore, OML requires these language bindings to be compiled into the system; it cannot interpret them at run-time, which complicates the process even more.

SPROUT [Snyder et. al., 95] is a successor to OML which promises more flexibility than its predecessor. However, the language bindings still have to be created and compiled with the system. A way to go around this problem is for the framework to have a SPROUT language binding generator. This generator takes as an input a shared schema and a product model description. It then generates their language bindings and compiles a copy of SP_II, which can export that product model into the shared schema representation. Creating a separate copy of SP_II for every shared schema is still required since SPROUT does not support the definition of multiple shared schemata within the same communication protocol.

7.2.2 Modeling Evaluation and Integrity Checking Mechanisms

During the course of developing SP_II, the need for capturing dependencies between specification primitives, which constitute the static attributes of a specification unit, became clear. Such dependencies are needed to provide value constraints on some attribute values that are dependent on other attributes. An example is a constraint indicating that the value of the minimum area attribute of a specification unit should be at least equal to the square of its minimum width attribute value. One way to achieve such a model of dependencies is have a spreadsheet representation for a product model that manages these dependencies.

An evaluation mechanism that is currently implemented in SP_II provides rules of composition for a hierarchy using classifications [section 4.3.2]. A detailed study of evaluation techniques needed in the architectural programming process can provide some insights into the kind of evaluation mechanisms the framework needs to support.

7.2.3 Investigating the Dependencies Between Classifications

SP_II allows for the definition of multiple classifications of a specification unit. However, it assumes that these classifications are mutually exclusive. Assigning a classifier from a certain group to a SU does not prevent assigning another classifier belonging to another group to the same SU or invalidate any pervious assignments. This requires the modeler to exercise great care in creating classifications to make sure that they are mutually exclusive. The way classification is implemented in SP_II does not provide for the creation of dependencies between classifications. Such an elaborate classification scheme can be implemented using a classification system such as CLASSIC [Woods 91]. Another alternative is to create a multi-faceted classification taxonomy similar to the one described in [Rivard 97].

7.2.4 Providing Classification-Based Attributes

A specification unit gets its attributes from the specification primitives defined in the product model it uses. Some of these attributes might not be applicable to certain classification of specification units, but have to be there to support other classifications. An example is a SU classified as a *bedroom* that has attributes for specifying plumbing requirements. Such attributes are needed for a SU classified as a *bathroom* or *kitchen*. A set of classification-based attributes can be defined to solve this problem. These attributes will be added to a SU or removed according to its classification. Having classification-based attributes requires a mechanism for managing the addition and removal of these attributes in a way that ensures the integrity of the representation and can be a future research issue.

7.2.5 Investigating Additional Uses for Indirect Mapping

Indirect mapping is currently used to set the value a numeric attribute of an output SU based on the values of a group of numeric attributes in the input SU, that are linked by a mathematical expression. These expressions (or formulae) currently accept numbers, basic arithmetic operators and references to numeric attributes only. If they are expanded to handle logical operators, they can be used to set non-numeric attribute values through a generation strategy based on logical expressions. An example, can be:
IF (Classification == “Bedroom”) THEN (Orientation = “South East”), which set the value of the Orientation attribute of the output SU to *South East*, if the input unit is classified as *Bedroom*. Adding these features requires augmenting the recursive descent parser used to parse formulae.

7.2.6 Usability Analysis: Investigating Interface Metaphors for Representing and Manipulating Design Requirements

Systems that model and manipulate design requirements are a software novelty. Their interfaces primarily provides simple metaphors that are closely tied to their internal information model. For example, SP_II uses a lot of tree displays as a metaphor for representing compositions that exist in its domain object model. Whereas, these metaphors might appear adequate to manipulate the model, empirical studies, involving different

categories of potential users and evaluators, are needed to verify the validity of such metaphors. These studies can be the subject of future research.

CHAPTER 8

Bibliography

[Aho & Ullman 72] Aho, A. & J. Ullman. (1972). *The Theory of Parsing, Translation, and Compiling*. Englewood Cliffs, N.J: Prentice-Hall

[AIA 66] American Institute of Architects. (1966). *Emerging Techniques of Architectural Practice*. Washington, DC: The AIA Press.

An early publication that signaled the emergence of architectural programming as a separate field.

[Akin et. al., 95] Akin, Ö., R. Sen, M. Donia, & Y. Zhang. (1995). SEED-Pro: Computer Assisted Architectural Programming in SEED. *In Journal of Architectural Engineering*, Vol 1 No, 4, December 1995.

A paper that describes the functionalities and design of SEED-Pro.

[Augenbroe 93] Augenbroe G. L. M., S. C. Chase, W. Rombouts, and H. R. Schipper. (1993). Information Models in Building Design. In M. R. Beheshti et al. (Ed.) *Advanced Technologies: architecture - planning - civil engineering*. NY: Elsevier.

This papers deals with the assessment of approaches in information models in buildings from different perspectives. Focuses on the shortcomings of EDM.

[Bachman 69] Bachman, C.W. (1969). Data Structure Diagrams. *In Data Base 1 (2)*, pp. 4-10.

A paper that contains a description of some data structure diagrams, particularly the entity- relationship model.

[Barr 89] Barr, A., Cohen, P. R., & Feigenbaum, E. A. (Eds.) (1989). *The Handbook of Artificial Intelligence* (Vol IV). Reading: Addison Wesley, Inc.

A companion volume to the previous volume which summarizes more recent developments in Artificial Intelligence including: Blackboard Systems, Expert Systems, Computer Visions, Distributed AI, and Simulation.

[Berzins & Luqi 91] Berzins, V., & Luqi. (1991). *Software Engineering with Abstractions.* Reading: Addison Wesley, Inc.

A book on software engineering that describes a design methodology based on abstract data types (ADTs).

[Brachman et al. 91] Brachman, R. J., McGuiness, P.F. Patel-Schneider, L. Borgida. (1991). Living with Classic: When and How to Use a KL-ONE-like Language. In J. Sowa. (Ed.) *Principles of Semantic Networks Explorations in the Representation of knowledge.* San Mateo, CA: Morgan Kaufmann Publishers, Inc.

A paper that shows how to use classification- based knowledge representation systems to their fullest potential.

[Brodie 86] Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (1986). *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages.* Harrisonburgh, VA: Springer-Verlag.

A paper that contains an overview of some conceptual model along with some evaluations.

[Brooks 95] Brooks, F. (1995). *The Mythical Man-Month: Essays on Software Engineering. Reading,* MA: Addison-Wesley Publishing. Co.

The anniversary edition of the famous book which provides a strong argument against ad-hoc approaches to complex systems development through the authors experience with the development of the IBM system 360. It was first published in 1975.

[Brunet 91] Brunet, J. (1991). Modeling the World with Semantic Objects. In *Proceedings of the IFIP TC8/WG8.1 Working Conference on the Object-Oriented Approach to Information Systems*, Van Assche, F., Moulin, B., and Rolland, C., (eds.). Quebec, Canada: North-Holland.

A paper that advocates the refinement of the aggregation relationship in object-oriented system design, into the more general “composition” relationship.

[Chen 76] Chen, P.P.S. (1976). The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol1 No 1, March 1976.

A paper that describes a conceptual model called the Entity-Relationship model which is based on data independence. It adopts the view that the real world consists of entities and relationships among them. Entities and relationships are characterized by properties and the three of them are classified into types.

[Chikofsky & Cross 90] Chikofsky, E., & Cross, J. (1990). Reverse Engineering and Advanced Recovery: A Taxonomy. *IEEE Software*, January 1990.

A paper that describes a conceptual model called the Entity-Relationship model which is based on data independence. It adopts the view that the real world consists of entities and relationships among them. Entities and relationships are characterized by properties and the three of them are classified into types.

[Codd 70] Codd, E. F. (1970). A Relational Model for large shared data banks. *Communications of ACM*, Vol 13 No 6, pp.377-387.

A paper that introduced the relational model for database management systems.

[Domeshek & Kolodner 92] Domeshek, E. A. and J. L. Kolodner. (1992) "A Case-Based Design Aid for Architecture". *Artificial Intelligence in Design `92*, J. Gero (ed.), Boston: Kluwer Academic Publishers, pp. 497-516.

[Duerk 93] Duerk, D. P. (1993). *Architectural Programming: Information Management for Design*. NY: Van Nostrand Reinhold.

An architectural programming book that views the process as the first part of the design process. It contains models and reviews of previous work done on the subject.

[Fenves et al. 94] Fenves, S., U. Flemming, C. Hendrickson, M. L. Maher, R. Quadrel, M. Terk, & R. Woodbury. (1994). *Concurrent Computer Integrated Building Design*. Engelwood Cliffs, NJ: Prentice Hall.

A book that describes the concept of integrated building design with a focus on IBDE. Contains brief descriptions of systems with similar approaches, such as, ICADS and DICE.

[Fenves et al. 95] Fenves, S., Rivard, H., Gomez, N., & Chiou, S. C. (1995). "Conceptual Structural Design in SEED" *Journal of Architectural Engineering*, 1(4), New York: American Society of Civil Engineers. pp. 179-186.

[Flemming & Chien 95] Flemming, U. and Chien, S. F. (1995). "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering*, 1(4), New York: American Society of Civil Engineers. pp. 162-169.

[Flemming & Woodbury 95] Flemming, U. and Woodbury, R. (1995). "Software Environment to Support the Early Phases of Building Design (SEED): Overview" *Journal of Architectural Engineering*, 1(4), New York: American Society of Civil Engineers. pp. 162-169.

A paper that provides an overview of the SEED system.

[Flemming et al. 96] Flemming, U., Aygen, Z., Snyder, J., and Tsai, J. (1996). "A2: An Architectural Agent in a Collaborative Engineering Environment". *Report EDRC 48-38-96, Engineering Design Research Center.* Carnegie Mellon University, Pittsburgh, PA.

[Flemming et al. 94] Flemming, U, Coyne, R., Snyder, J. (1994). "Case-Based Design in the SEED System". *Proc. 1st Congress on Computing in Civil Engineering*, K. Khozeimeh, ed. New York: American Society of Civil Engineers, pp.446-453.

[Flemming et al. 92] Flemming, U., R. Coyne, R. Woodbury, S. Bhavnani, S. Chiou, B. Chio, R. Stouffs, T. Chang, S. Han, C. Jo, H. Kiliccote, J. Shaw, & K. Suwa (1992). *SEED-LOOS Requirements Analysis*. Engineering Design Research Center, Carnegie Mellon University. Unpublished working paper.

This document is a requirements analysis developed using the OOSE software engineering methodology.

[Freeman & Newell 71] Freeman, P. and A. Newell. (1971). *A Model for Functional Reasoning in Design*. Proceedings of the 2nd International Conference on Artificial Intelligence. London: British Computer Society, pp. 621-640.

This document is a requirements analysis developed using the OOSE software engineering methodology.

[Gamma et al. 95] Gamma, E., R. Helm, R. Johnson, & J. Vlissides. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. NY: Addison-Wesley.

A very successful book that gained a wide attention in the field of software engineering. It describes the idea of designing software around behavioral patterns that map to certain object relations. It includes a catalogue of design patterns organized in three main categories: Creational, Structural, and Behavioral patterns.

[Garret et. al., 95] Garret, J. H., Kiliccote, H., & Choi, B. (1995). Providing Formal Support for Standards Usage within SEED. *In Journal of Architectural Engineering*, Vol 1 No, 4, December 1995.

A paper that contains a description of the standards support environment which contains a standards modeling language.

[Hix et al. 93] Hix, D., & H. R. Hartson. (1993). *Developing User Interfaces: Ensuring Usability Through Product & Process*. NY: John Wiley and Sons, Inc.

A book about the design of graphical user interfaces. It contains an overview of the evolution of the user interfaces as well as some suggested design guidelines.

[Jacobson et al. 92] Jacobson, I., M. Christerson, P. Jonsson, & G. Övergaard. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. NY: Addison-Wesley.

A widely referenced book and notion methodology that can be used to completely specify the behavior of a system designed around objects starting from use cases. Extensive coverage of the Object Oriented Software Engineering Method (OOSE) is provided.

[Johnson & Feather 90] Johnson, W.I. & Feather, M. (1990). Building an Evolution Transformation Library. *Proceedings of the 12th International Conference on Software Engineering*. Nice, March 90.

[Khedro 95] Khedro, T., Case, M. P., Flemming, U., Genesereth, M. R., Logcher,R., Pedersen, C., Snyder, J.,Sriram, R. D. and P. M. Teicholz. (1995). "Development of a Multi-Institutional Test bed for Collaborative Facility Engineering Infrastructure" in J. P. Mohsen (Ed.) *Computing in Civil Engineering*: *volume 2: Proceedings of the Second Congress held in conjunction with the A/E/C Systems `95*, Atlanta, GA, June 5-8, New York: American Society of Civil Engineers. pp. 1308-1315.

A paper that contains a description of the ACL project which was a joint project between Carnegie Mellon University, MIT, Stanford Uni-

versity, and the University of Illinois at Urbana, to create a distributed building design environment by combining multiple design systems located in each of the contributing institutions.

[Koopmans & Beckman 57] Koopmans, J. C. & Beckman, M. J. (1957). Assignment Problems and the Location of Econometric Activities. *Econometrica 25: p. 53-76.*

Describes a system to translate architectural drawings (plans) into topological concepts such as circulation elements, adjacencies and direct access.

[Koutamanis & Mitossi 93] Koutamanis, A. & V. Mitossi. (1993). On the representation of Dynamic Aspects of Architectural Design in Machine Environment. In M. R. Beheshti et al. (Ed.) *Advanced Technologies: architecture - planning - civil engineering.* NY: Elsevier.

Describes a system to translate architectural drawings (plans) into topological concepts such as circulation elements, adjacencies and direct access.

[Krasner et al. 88] Krasner, G. E., & S.T. Pope. (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. In *Journal of Object-Oriented Programming, August/September 1988.*

A text about the use of the Model-View-Controller paradigm in Smalltalk.

[Kumlin 95] Kumlin, Robert. (1995). *Architectural Programming: Creative Techniques for Design Professional*s. New York: McGraw-Hill.

A book that contains some techniques for conducting architectural programming as well as an overview of architectural programming evolution and practices.

[Liggett 80] Liggett, R. S. (1992). The Quadratic Assignment Problem: An Analysis of Applications and Solution Strategies. *Environment and Planning B: Planning and Design 7,* pp. 141-162.

A paper that contains a description of creating layout using stacking and blocking diagrams based on the quadratic assignment problem.

[Loucopoulos et al. 92] Loucopoulos, P., & Zicari, R. (1992). *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*. New York: John Wiley and Sons, Inc.

A collection of recent papers that present the collective concept of information modeling as an approach to large information system development.

[Loucopoulos 92] Loucopoulos, P. (1992). *Conceptual Modeling.* In Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development. New York: John Wiley and Sons, Inc.

A paper that provides an introduction to information systems and conceptual modeling.

[Maiden &Sutcliffe 91] Maiden, N.A.M. & Sutcliffe, A.G. (1991). Specification Reuse by Analogy. *Software Engineering Journal 6(1), pp. 3-15.*

A paper that illustrates a mechanism for reusing specifications in the design of information systems.

[Minsky 75] Minsky, M. (1975). A Framework for Representing Knowledge. In *The Psychology of Computer Vision,* Winston, P., ed. New York, NY: McGraw-Hill.

A famous knowledge representation paper in which the notion of *frames* was first introduced.

[Myer 93] Myer, B. A. (1993). State of the Art in User Interface Software Tools. In H. R. Hartson & D. Hix (Ed.), *Advances in Human-Computer Interaction,* Vol 2. Norwood, NJ: Ablex.

An in-depth survey on the available UI software tools.

[Mylopoulos et al. 80] Mylopoulos, J., Bernstein, P.A., & Wong, H.K.T. (1980). Language Facility for Designing Database Intensive Applications. *ACM Transactions on Database Systems,* Vol 15, No 2.

[Mylopoulos 92] Mylopoulos, J. (1992). *Conceptual Modeling and Telos.* In Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development. New York: John Wiley and Sons, Inc.

A paper that provides an introduction to information systems and conceptual modeling as well as introducing a modeling language based on the notion of propositions.

[Olive 86] Olive, A. (1986). A Comparative of the Operational and Deductive approaches to Conceptual Information Systems Modeling. *Information Processing 86.* Nordweighthout, Netherlands: Elsavier Science, North-Holland.

A paper that contains an overview of some conceptual modeling approaches.

[Peckham & Maryanski 88] Peckham, J., & Maryanski, F. (1988). Semantic Data Models. *ACM Computing Surveys*, Vol 20, No. 3, September 1988.

A paper that contains a description of semantic data models.

[Pena et al. 87] Pena, W., S. Parshall & K. Kelly. (1987). *Problem Seeking: An Architectural Programming Primer.* Washington: AIA Press.

A famous book that contains a description of an architectural programming methodology that became adapted by several firms.

[Pree 95] Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. NY: Addison-Wesley.

A book that describes the idea of building and using application frameworks using design patterns. It contains a detailed description of the concepts used in the ET++ application framework.

[Preiser 93] Preiser, W. (1993). *Professional Practice in Facility Programming.* NY: Van Nostrand Reinhold.

A book on facility programming to develop design parameters and specifications over a broad range of project types. It addresses the most recent developments, and the newest applications and methodological advances in the field.

[Rich & Knight 91] Rich, E., & K. Knight. (1991). *Artificial Intelligence, second edition*. NY: McGraw-Hill, Inc.

Provides an overview for artificial intelligence in general.

[Rivard 97] Rivard, H. (1997). *A Building Design Representation for Conceptual Design and Case-Based Reasoning*. Pittsburgh, PA: Carnegie Mellon University, Doctoral Thesis, Department of Civil and Environmental Engineering.

[Rolland et al. 92] Rolland, C., & Cauvet, C. (1992). *Trends and Perspectives in Conceptual Modeling.* In Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development. New York: John Wiley and Sons, Inc.

A paper that provides an introduction to information systems development and conceptual modeling with an emphasis on the influence of object orientation.

[Rumbaugh et al. 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice-Hall.

A widely referenced book and notion methodology that can be used to completely specify the behavior of a system designed around objects. Extensive coverage of the Object Modeling Technique (OMT) is provided.

[Sanoff 74] Sanoff, H. (1974). *Methods of Architectural Programming*. Strousburg, PA: Dowden, Mutchinson & Ross.

A book that contains a description of an architectural programming methodology.

[Schmid 75] Schmid, H.A., Swenson, J.R. (1975). On the Semantics of the Relational Model. *In Proceedings of the ACM SIGMOD Conference, pp 211-223*.

A paper that contains a description of the relational model for database management systems.

[Schmidt 77] Schmidt, J.M., Schmidt, D.C.P. (1977). Database Abstractions: Aggregation and Generalization. *ACM Transaction on Database Systems,* Vol 2, No 2, pp 105-133.

A paper that contains a description of some abstraction forms that became widely used in information modeling.

[Simon 89] Simon, H. A. (1989). Models of Thought, volume I. New Haven, CT: Yale University Press.

A collection of papers, organized into chapters, by the author and others that contains a wide range of studies and theories of cognitive and information processing psychology.

[Snyder et. al., 95] Snyder, J., Aygen, Z., Flemming, U., & Tsai, J. (1995). SPROUT - A Modeling Language for SEED. *In Journal of Architectural Engineering,* Vol 1 No, 4, December 1995.

A paper that contains a description of a modeling language to be used as means of information storage and transfer between the different modules of the SEED system.

[Stroustrup 91] Stroustrup, B. (1991). *The C++ Programming Language (2nd ed.)*. Englewood Cliffs: Prentice-Hall.

This reference was written by the author of the C++ language and describes the language as well as intended uses via examples.

[Ullman 88] Ullman, J. (1988). *Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems*. Rockville, Maryland: Computer Science Press.

A sequel to the widely used reference: Principles of Database Systems, by the same author, that incorporates the emerging idea of the knowledge-base systems and object oriented databases.

[Verrijin-Stuart 87] Verrijin-Stuart, A. (1987). *Themes and Trends in Information Systems*. The Computer Journal, 30.

[Weinand et al. 95] Weinand, A. & E. Gamma. (1995). ET++ - a Portable, Homogenous Class Library and Application Framework, research report, Cupertino CA: Taligent, Inc.

A paper that describes the ET++ application framework for developing object-oriented software application. It contains examples of commercial applications built using the framework.

[Woodbury & Chang 95] Woodbury, R. & Chang, T. W. (1995). "Massing and Enclosure Design with SEED-Config" *Journal of Architectural Engineering*, 1(4), New York: American Society of Civil Engineers. pp. 170-178.

[Woods 91] Woods W. A. (1991). Understanding Subsumtion and Taxonomy: A Framework for Progress. In J. Sowa. (Ed.) Principles of *Semantic Networks Explorarions in the Representation of knowledge*. San Mateo, CA: Morgan Kaufmann Pubishers, Inc.

A paper that analyses the concept of subsumtion and taxonomy and synthesizes a framework that integrates and clarifies many of previous approaches and goes beyond then to provide and account of abstract and partially defined concepts.

APPENDIX 1

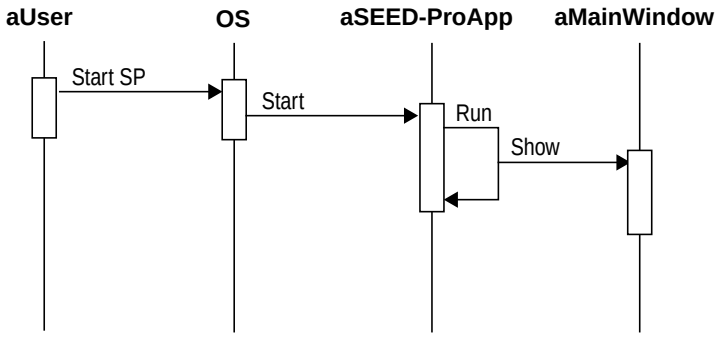
Requirements Modeling Using Use Cases

A 1.1 Common Use Cases

This set of use cases represents the functionality of the system accessible to both actors. These use cases are:

1. Close Product Library [page 90]
2. Open Product Library [page 89]
3. Rename Product Library [page 91]
4. Save Product Library [page 92]
5. Start SEED-Pro [page 88]

A 1.1.1 Start SEED-Pro



Interaction Diagram 30. Starting SEED-Pro

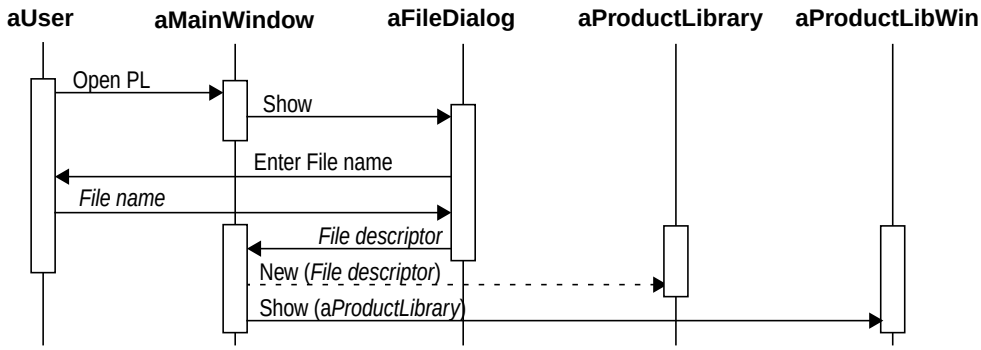
Flow of Events:

1. The user instructs the operating system to start SEED-Pro.
2. The operating system starts the SEED-Pro application.
3. The SEED-Pro application issues a “Run” command which displays the Main window.

Preconditions: A SEED-Pro executable exists in the current file system.

Interface Design:

A 1.1.2 Open Product Library



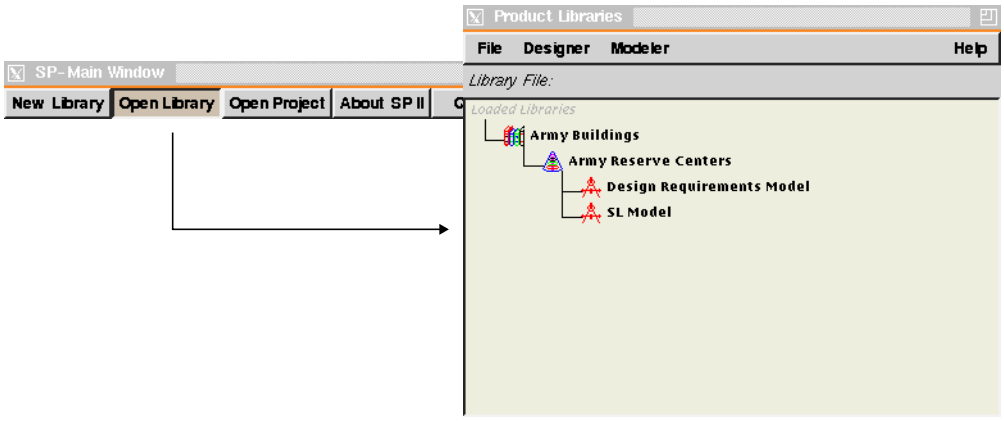
Interaction Diagram 31. Opening Product Library

Flow of Events:

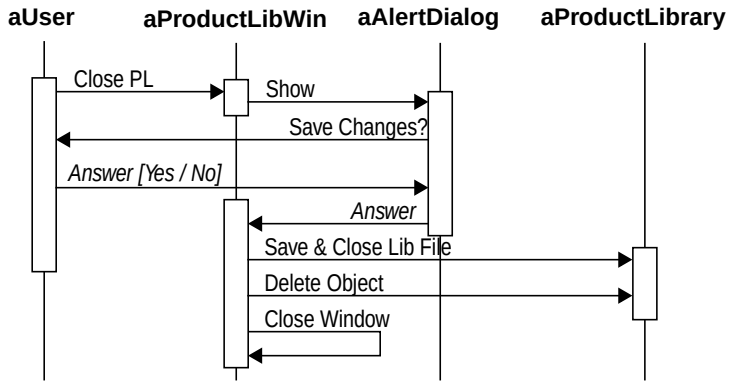
1. The user selects the “Open Product Library” command from the Main window.
2. The system displays the File dialog prompting the user for the Library file name.
3. The user types in the name or selects the file by navigating throught the file system and confirms.
4. The system creates the Library from the file and displays the Product Library window showing the selected Library.

Preconditions: Main window is displayed, and a Library file exists on the file system.

Interface Design:



A 1.1.3 Close Product Library



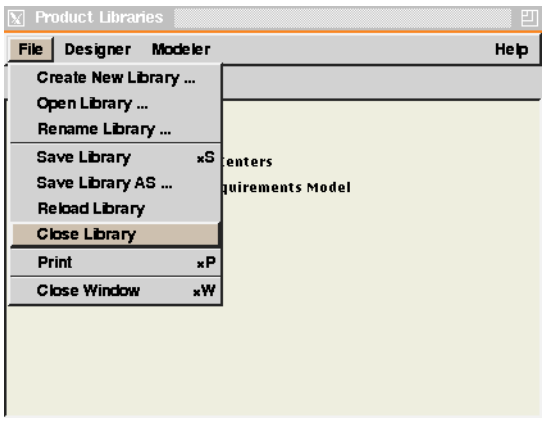
Interaction Diagram 32. Closing Product Library

Flow of Events:

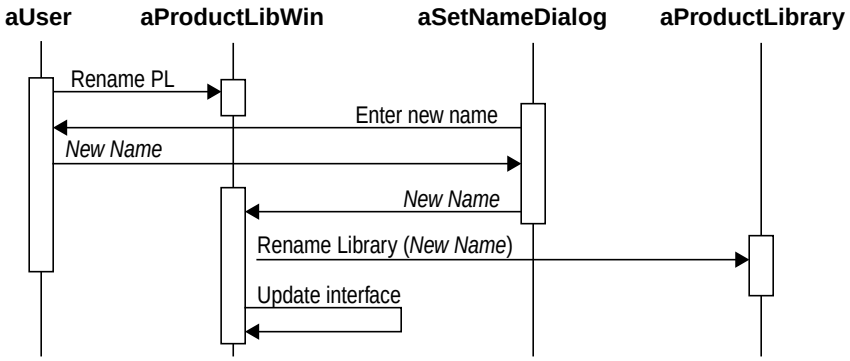
1. The user selects a library in the Product Library window then selects the “Close Product Library” command.
2. The system displays an alert dialog asking the user if changes to the Library should be saved.
3. The user makes a selection, and the system closes the selected library file and deletes the ibrary object.

Preconditions: A library is selected in the Product Library window.

Interface Design:



A 1.1.4 Rename Product Library



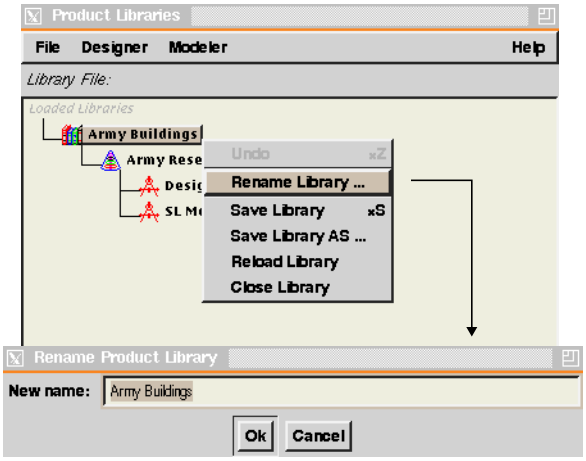
Interaction Diagram 33. Renaming Product Library

Flow of Events:

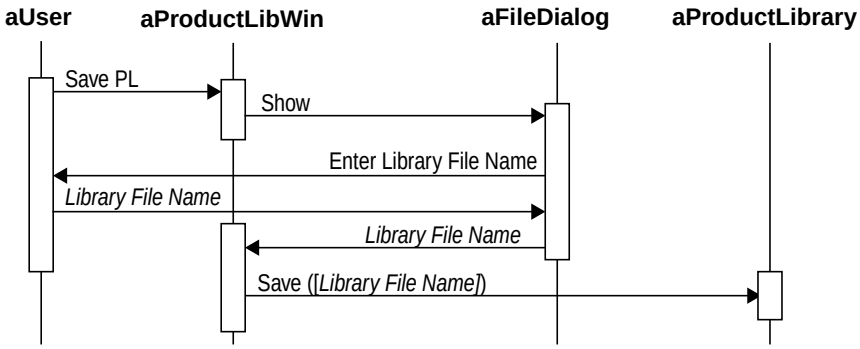
1. The user selects the “Rename Product Library” command from the Product Library window.
2. The system displays a Set Name dialog asking the user for the new name.
3. The user types in the name and confirms.
4. The systems sends a “Rename Library” command to the Product Library object, and updates the interface accordingly.

Preconditions: A library is selected in the Product Library window.

Interface Design:



A 1.1.5 Save Product Library



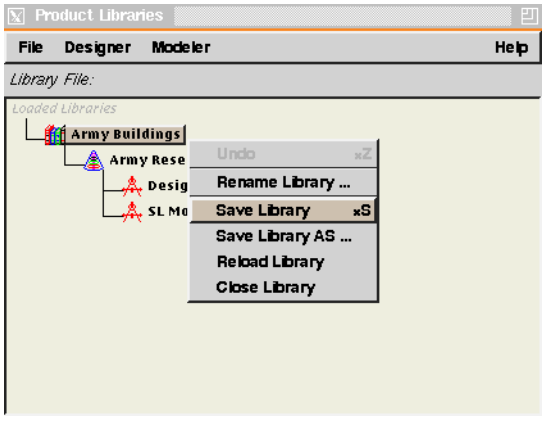
Interaction Diagram 34. Saving Product Library

Flow of Events:

1. The user selects the “Save Product Library” command from the Product Library window.
2. If the Library has been save before the system moves to step (5)
3. The system displays the File dialog prompting the user for Library file name.
4. The user types in the name and confirms.
5. The system send a “Save” command to the Product Library object.

Preconditions: A library is selected in the Product Library window.

Interface Design:



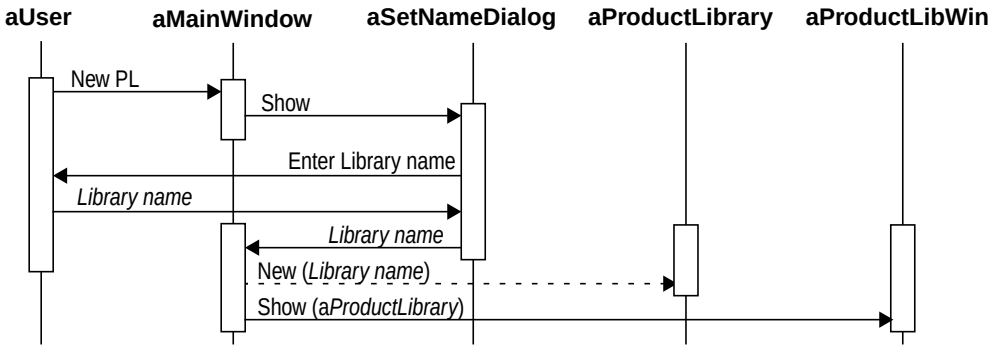
A 1.2 Use Cases for Modeler

This set of use cases represents the functionality of the system accessible to the specification modeler. These use cases are:

1. Copy Product / Product Model [page 103]
2. Copy Product Modeling Element [page 115]
3. Create Classification Group [page 118]
4. Create Classifier [page 121]
5. Create Generation Mechanism [page 130]
6. Create New Product [page 96]
7. Create Product Library [page 95]
8. Create Product Model [page 100]
9. Create Relation Type [page 127]
10. Create Specialized Mapping [page 144]
11. Create Specification Category [page 107]
12. Create Specification Primitive [page 110]
13. Cut (Remove) Product [page 105]
14. Cut (Remove) Product Model [page 106]
15. Cut Product Modeling Element [page 113]
16. Direct-Map Specification Primitives [page 136]
17. Edit Classifier [page 123]
18. Edit Product [page 98]
19. Edit Product Model [page 101]
20. Formula-Map Specification Primitive [page 138]
21. Map Classifiers [page 140]
22. Map Relation Type [page 142]
23. Paste Classifier [page 125]
24. Paste Product Model [page 104]
25. Paste Specification Primitive [page 116]
26. Remove Product [page 97]
27. Rename Classification Group [page 120]
28. Rename Product [page 99]
29. Rename Product Model [page 102]
30. Rename Relation Type [page 129]

31. Rename Specification Category [page 109]
32. Rename Specification Primitive [page 112]
33. Setup Generation Mechanism [page 132]
34. Show Generation Report [page 134]

A 1.2.1 Create Product Library



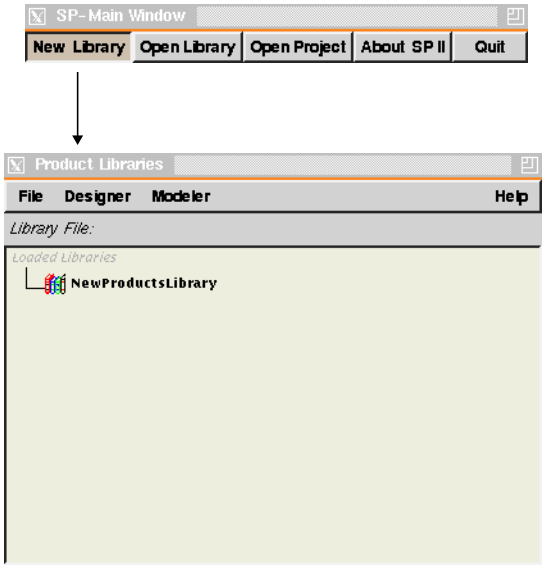
Interaction Diagram 35. Creating Product Library

Flow of Events:

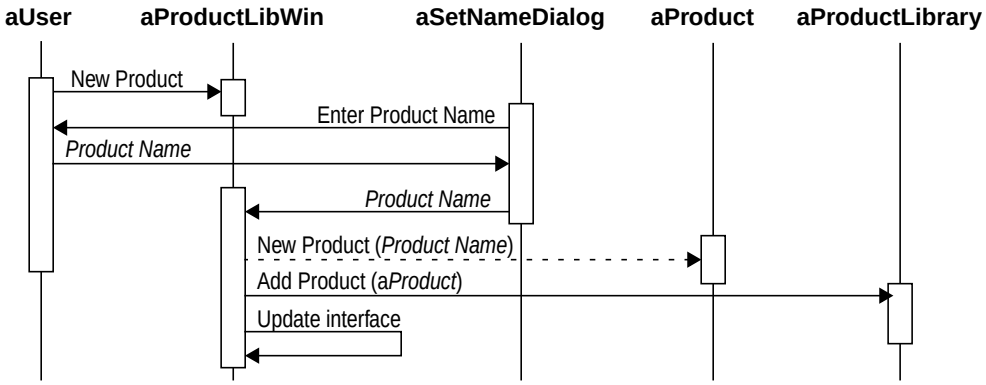
1. The user selects the "New Product Library" command from the Main window.
2. The system displays the Set Name dialog prompting the user for Library name.
3. The user types in the name and confirms.
4. The system creates a new Product Library and displays the Product Library window showing the new Library.

Preconditions: Main window is displayed.

Interface Design



A 1.2.2 Create New Product



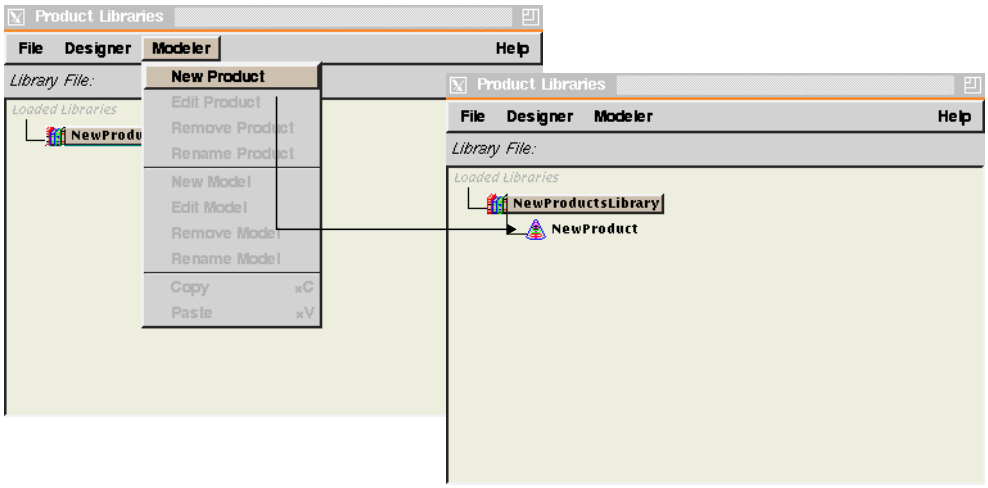
Interaction Diagram 36. Adding a New Product to Library

Flow of Events:

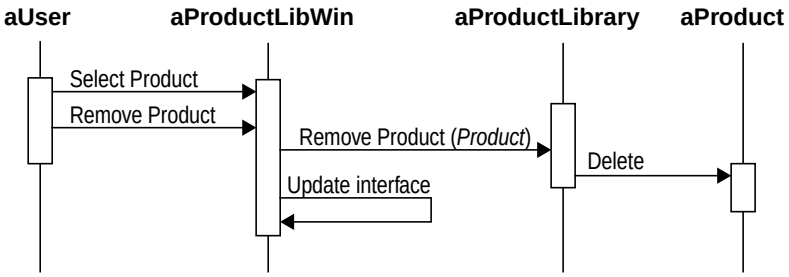
1. The user selects the "Create New Product" command from the Product Library window.
2. The system displays a Set Name dialog asking the user for the new name.
3. The user types in the name and confirms.
4. The systems creates a new Product object, adds it to the Product Library, and updates the interface accordingly.

Preconditions: A library is selected in the Product Library window.

Interface Design:



A 1.2.3 Remove Product



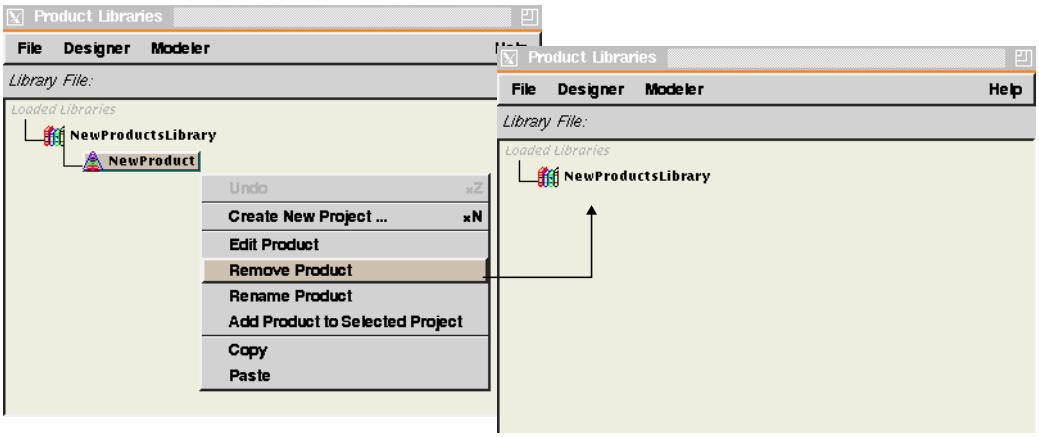
Interaction Diagram 37. Removing a Product

Flow of Events:

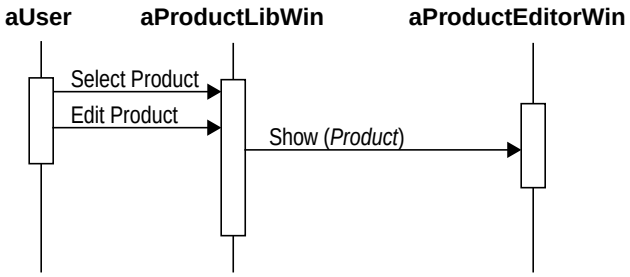
1. The user selects a Product from the Product Library window.
2. The user selects the “Remove Product” command from the Product Library window.
3. The system removes the selected Product from the Product Library, deletes the Product object and updates the interface accordingly.

Preconditions: A Product is selected in the Product Library window.

Interface Design:



A 1.2.4 Edit Product



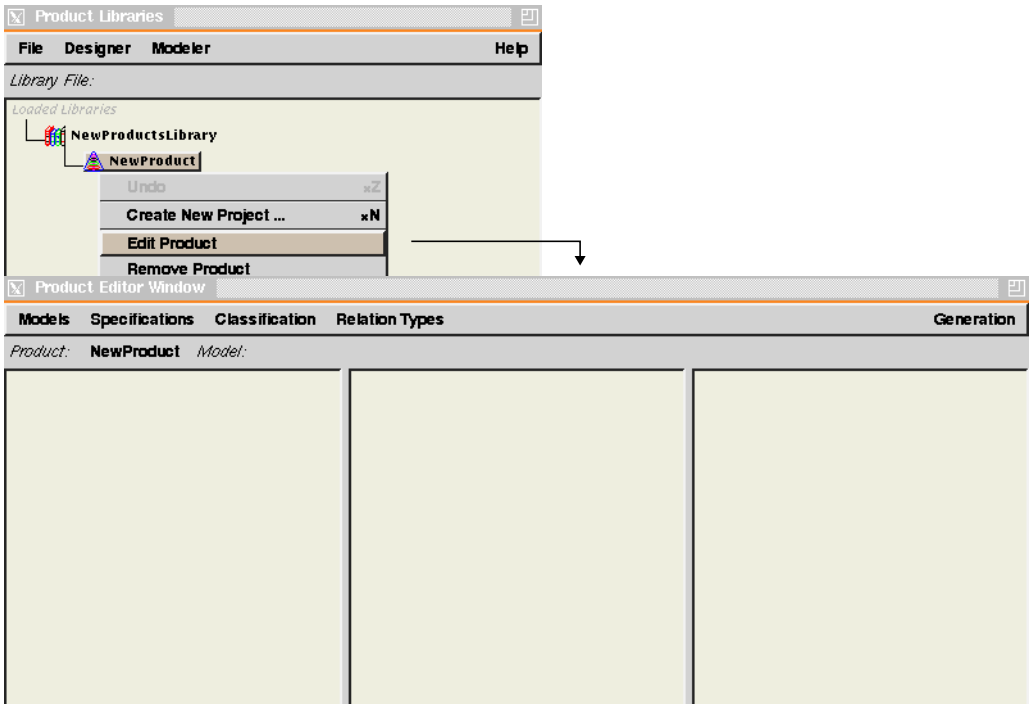
Interaction Diagram 38. Editing (modeling) a Product

Flow of Events:

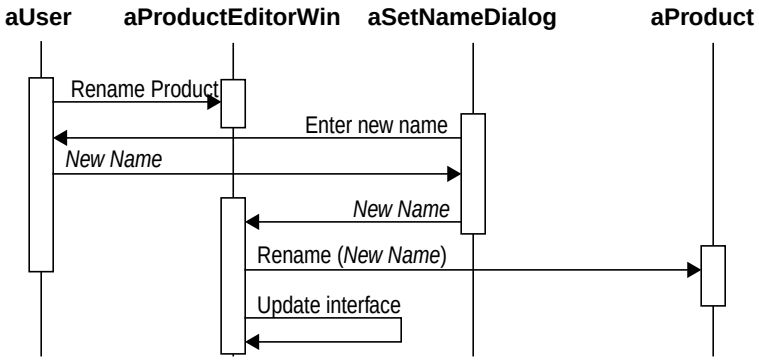
1. The user selects a Product from the Product Library window.
2. The user selects the "Edit Product" command from the Product Library window.
3. The system displays the Product Editor window with the selected Product ready for editing.

Preconditions: A Product is selected in the Product Library window.

Interface Design:



A 1.2.5 Rename Product



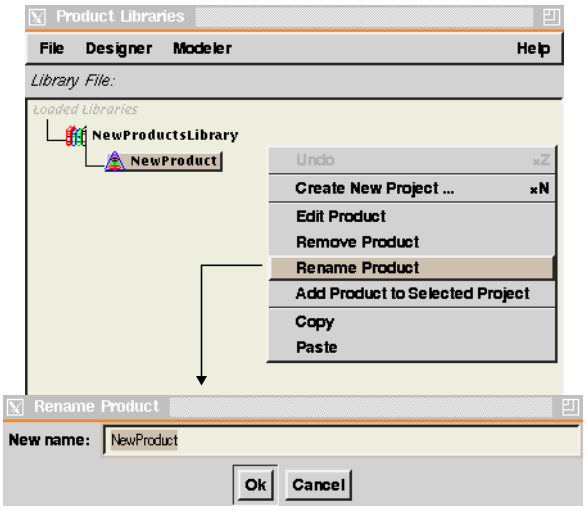
Interaction Diagram 39. Renaming a Product

Flow of Events:

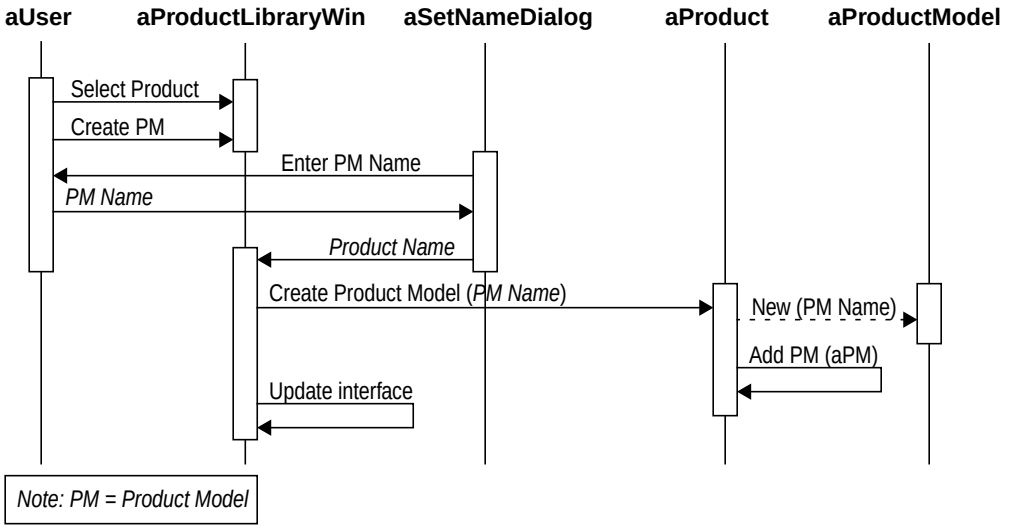
1. The user selects the “Rename Product” command from the Product Editor window.
2. The system displays the Set Name dialog prompting the user for a new name.
3. The user types in the name and confirms.
4. The system sends a “Rename” command to the Product object and updates the interface accordingly.

Preconditions: A Product is selected in the Product Editor window.

Interface Design:



A 1.2.6 Create Product Model



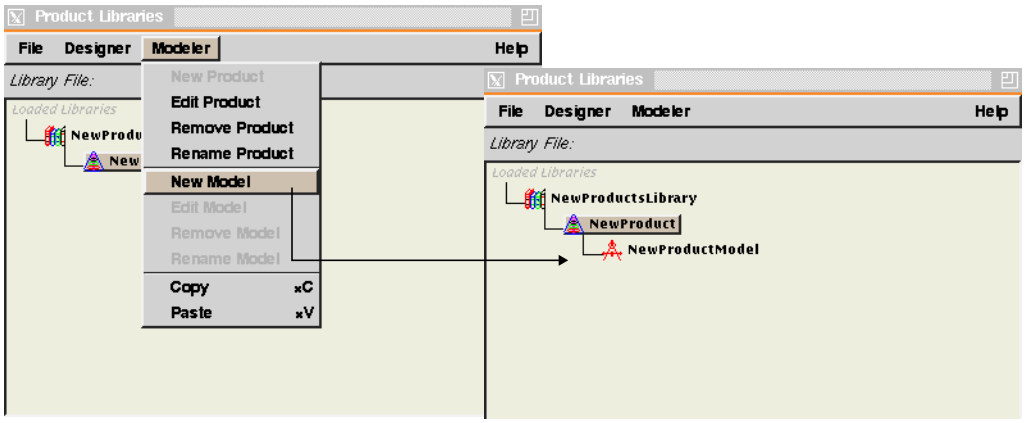
Interaction Diagram 40. Creating a Product Model

Flow of Events:

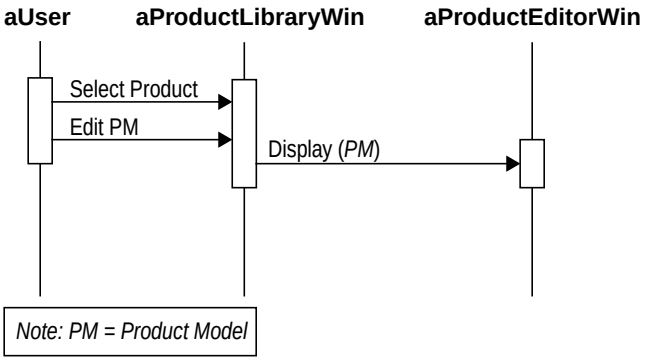
1. The user selects the "Create Product Model" command from the Product Library window.
2. The system displays the Set Name dialog prompting the user for the new Product Model name.
3. The user enters the name and confirms.
4. The system creates a new Product Model and adds it to the selected Model and updates the interface accordingly.

Preconditions: A Product is selected in the Product Library window.

Interface Design:



A 1.2.7 Edit Product Model



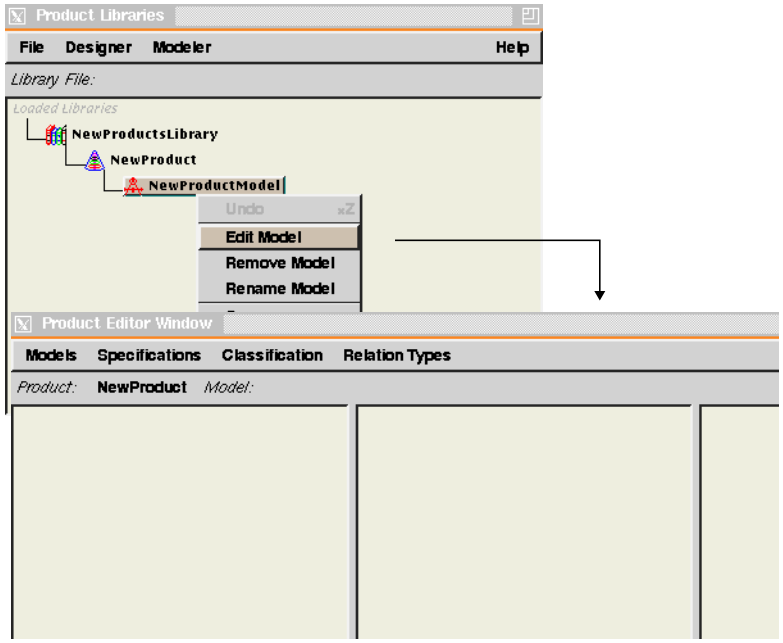
Interaction Diagram 41. Editing a Product Model

Flow of Events:

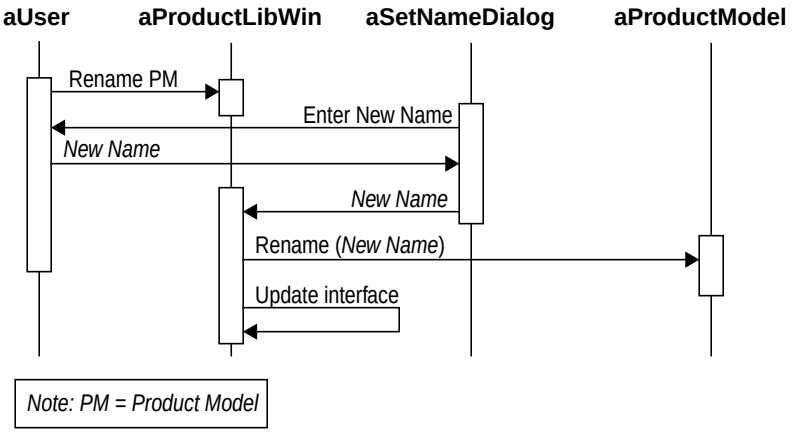
1. The user select a Product Model from the Product Library window.
2. The user selects the “Edit Product Model” command from the Product Library window.
3. The system displays the selected Product Model in the Product Model Editor window.

Preconditions: A Product Model is selected in the Product Library window.

Interface Design:



A 1.2.8 Rename Product Model



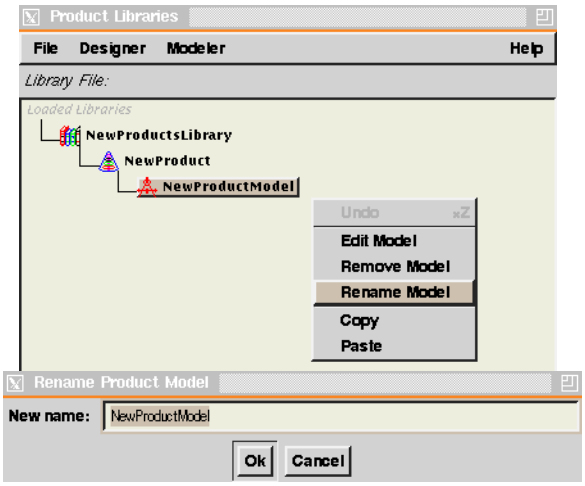
Interaction Diagram 42. Renaming Product Model

Flow of Events:

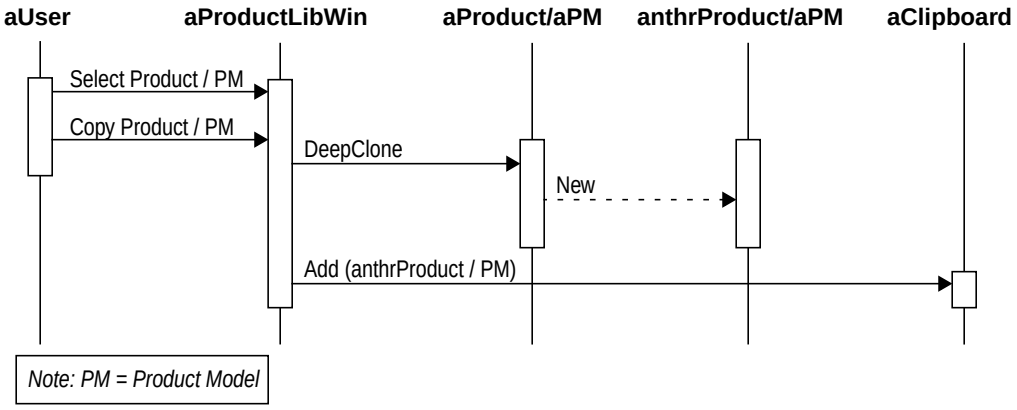
1. The user selects the “Rename Product Model” command from the Product Library window.
2. The system displays a Set Name dialog asking the user for the new name.
3. The user types in the name and confirms.
4. The systems sends a “Rename” command to the Product Model object, and updates the interface accordingly.

Preconditions: A Product Model is selected in the Product Library window.

Interface Design:



A 1.2.9 Copy Product / Product Model



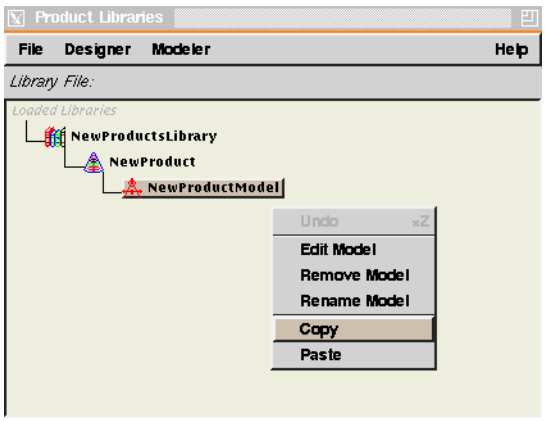
Interaction Diagram 43. Copying a Product or a Product Model

Flow of Events:

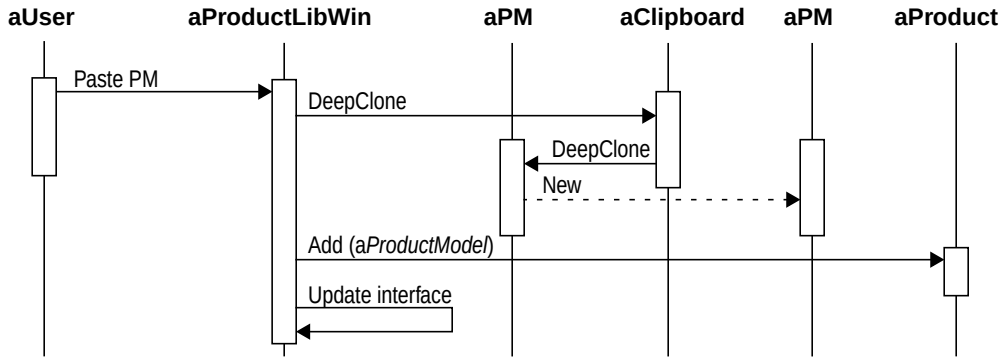
1. The user selects a Product or a Product Model from the Product Library window.
2. The user selects the "Copy" command from the Product Library window.
3. The system sends a "Deep Clone" command to the selected object and saves it in the system clipboard.

Preconditions: A Product or a Product Model is selected in the Product Library window.

Interface Design:



A 1.2.10 Paste Product Model



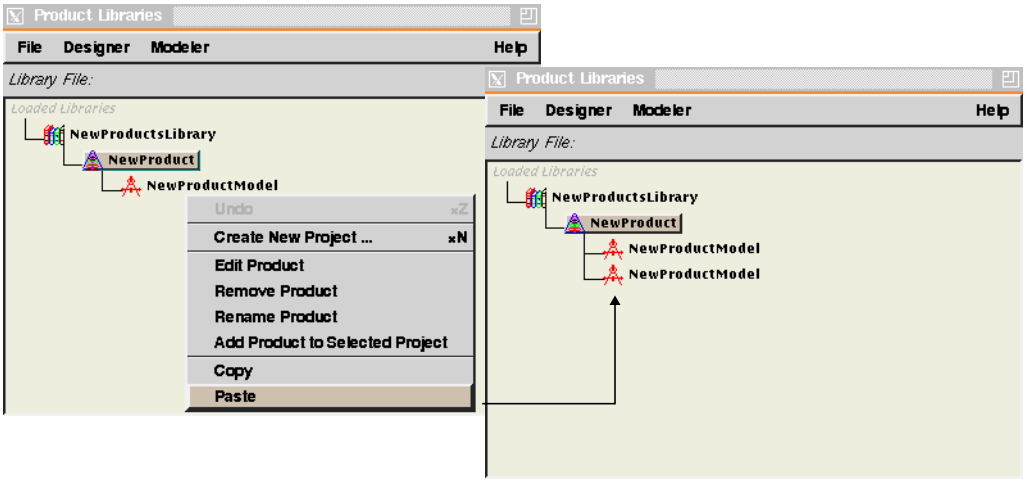
Interaction Diagram 44. Pasting a Product

Flow of Events:

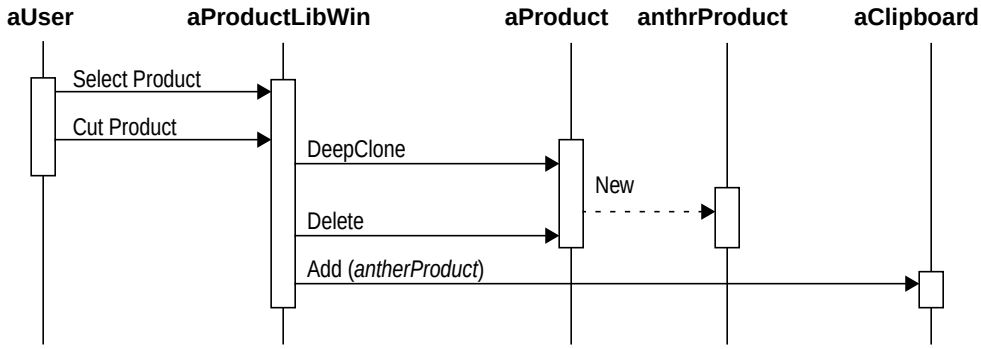
1. The user selects a Product then selects the "Paste" command from the Product Library window.
2. If the object in the system clipboard is a Product Model, the system sends it a "Deep Clone" command, adds the resulting clone to the selected Product and updates the interface.

Preconditions: A Product is selected in the Product Library window and a Product Model is already copied to the system clipboard.

Interface Design:



A 1.2.11 Cut (Remove) Product



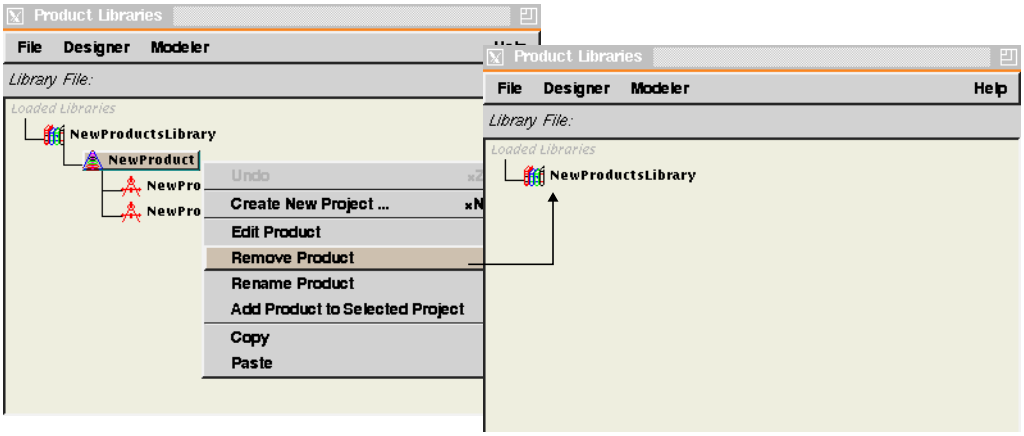
Interaction Diagram 45. Removing a Product

Flow of Events:

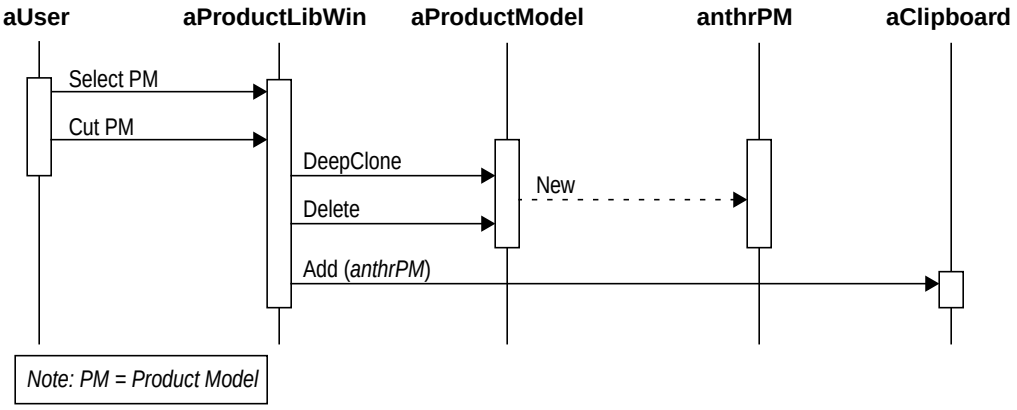
1. The user selects a Product from the Product Library window.
2. The user selects the “Cut” command from the Product Library window.
3. The system sends a “Deep Clone” command to the selected object, save the resulting clone in the system clipboard, and deletes the selected object.

Preconditions: A Product is selected in the Product Library window.

Interface Design:



A 1.2.12 Cut (Remove) Product Model



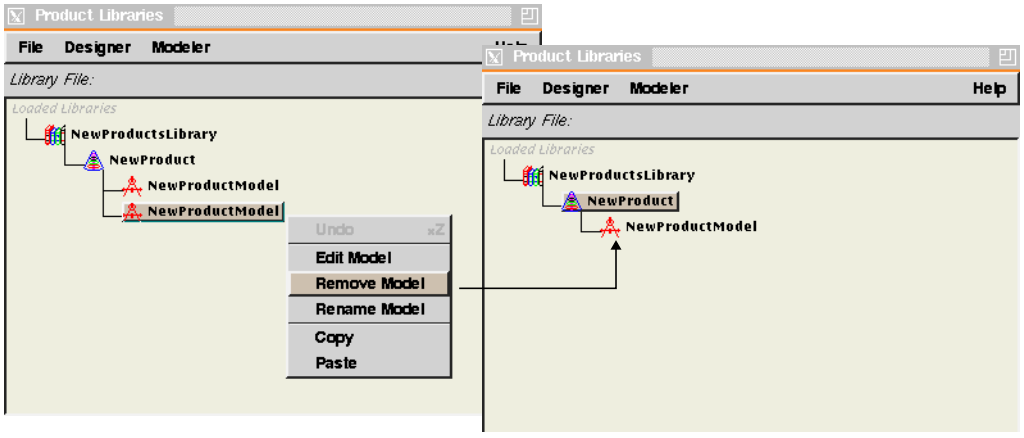
Interaction Diagram 46. Removing a Product Model

Flow of Events:

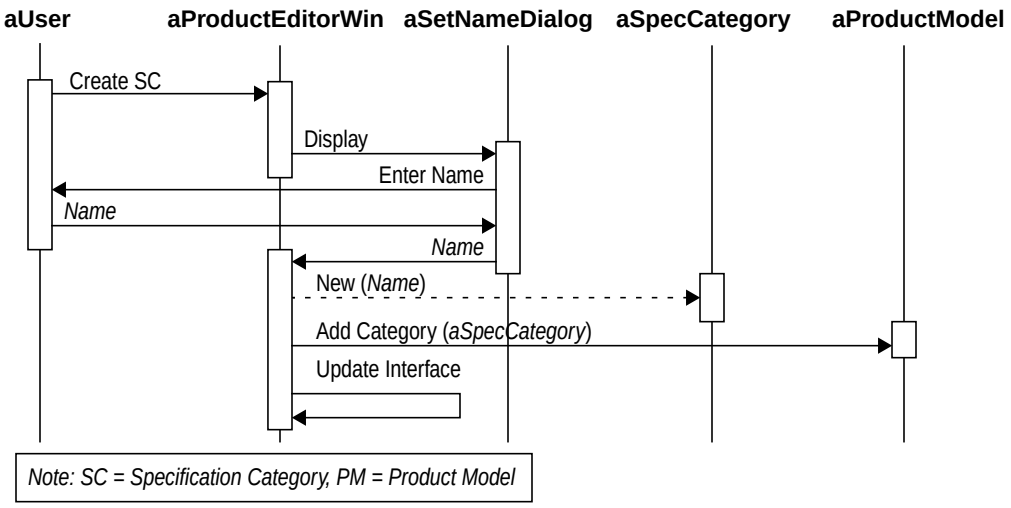
1. The user selects a Product Model from the Product Library window.
2. The user selects the "Cut" command from the Product Library window.
3. The system sends a "Deep Clone" command to the selected object, saves it in the system clipboard and deletes the selected object.

Preconditions: A Product Model is selected in the Product Library window.

Interface Design:



A 1.2.13 Create Specification Category



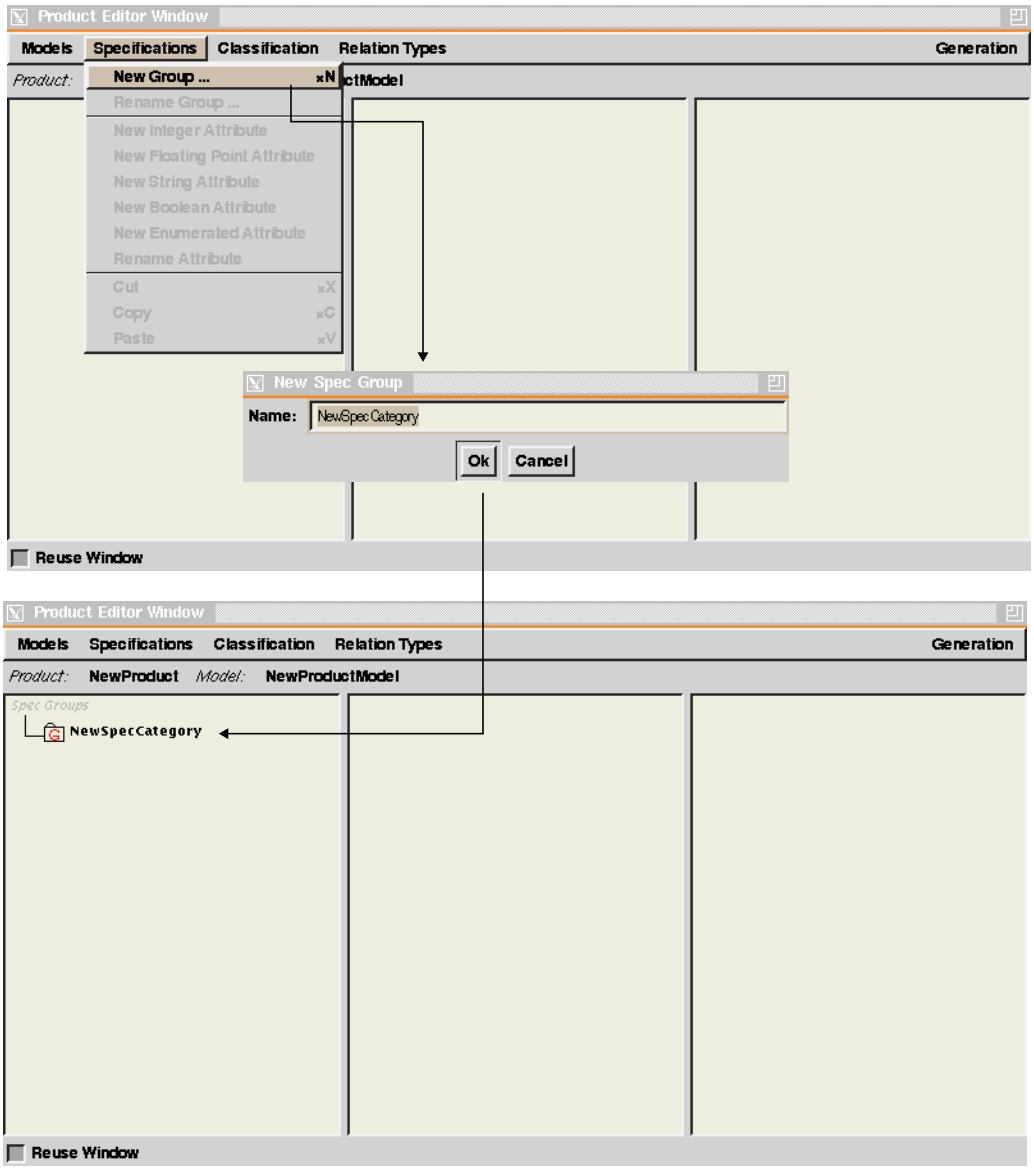
Interaction Diagram 47. Creating a Specification Category

Flow of Events:

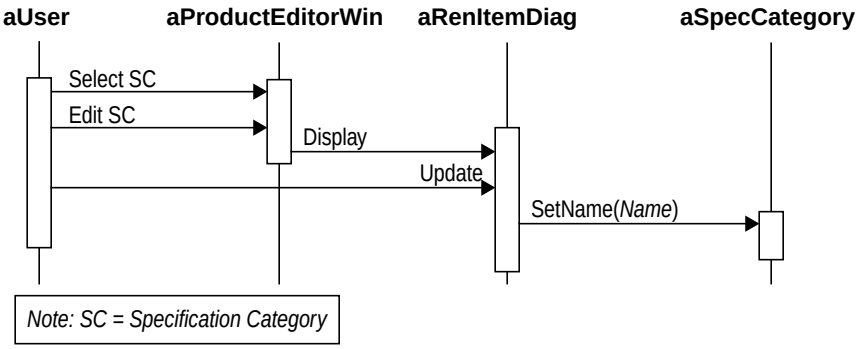
1. The user selects the "Create Specification Category" command from the Product Editor window.
2. The system displays the Set Name dialog prompting the user for a name.
3. The user types in the name and confirms.
4. The system creates a new Specification Category with the given name, adds it to the Product Model and updates the interface accordingly.

Preconditions: A Product Editor window is displayed.

Interface Design:



A 1.2.14 Rename Specification Category



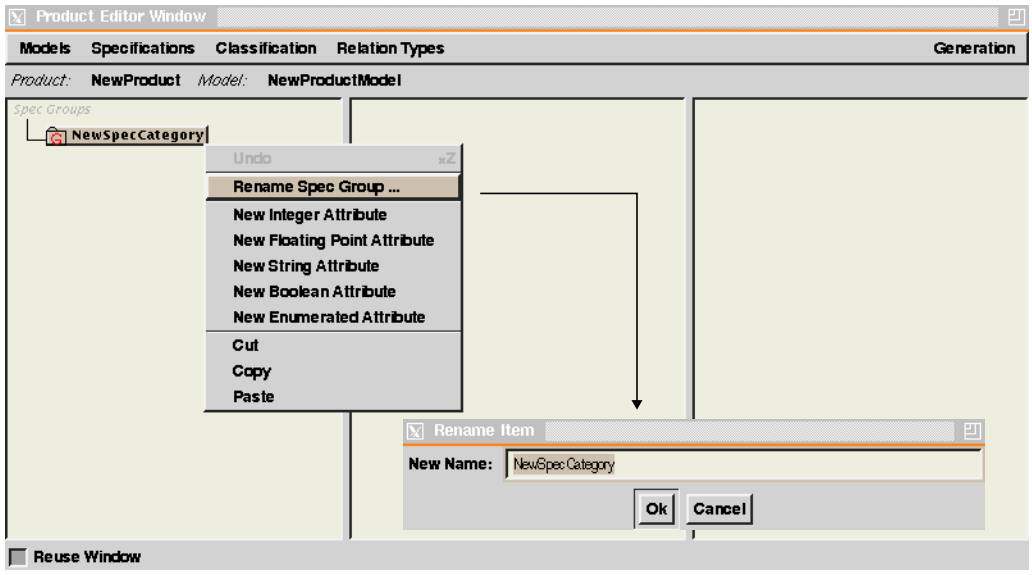
Interaction Diagram 48. Renaming a Specification Category

Flow of Events:

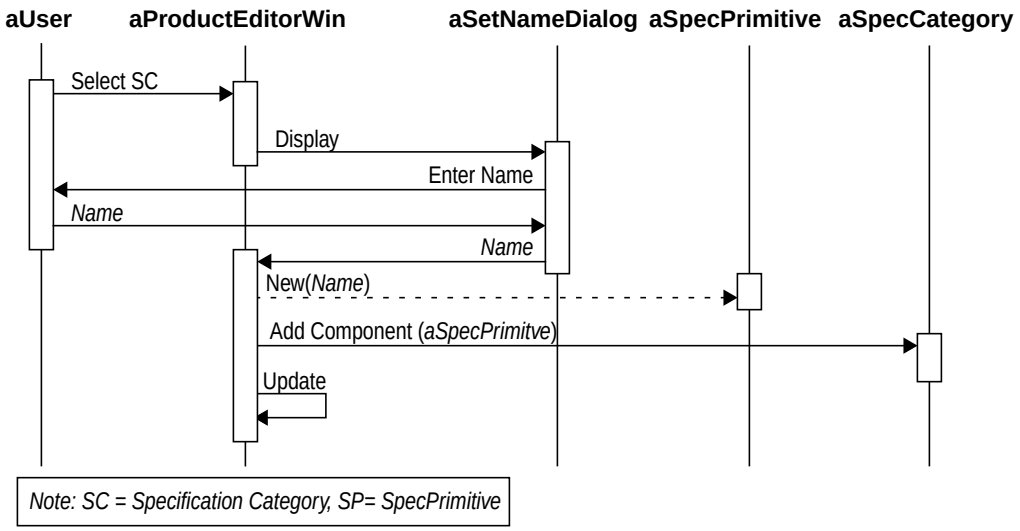
1. The user selects the “Rename Specification Category” command from the Product Editor window.
2. The system displays the Rename Item dialog box.
3. The user makes the changes in the window and confirms.
4. The system updates its corresponding Specification Category object, and updates the interface accordingly.

Preconditions: A Specification Category is selected in the Product Editor window.

Interface Design:



A 1.2.15 Create Specification Primitive



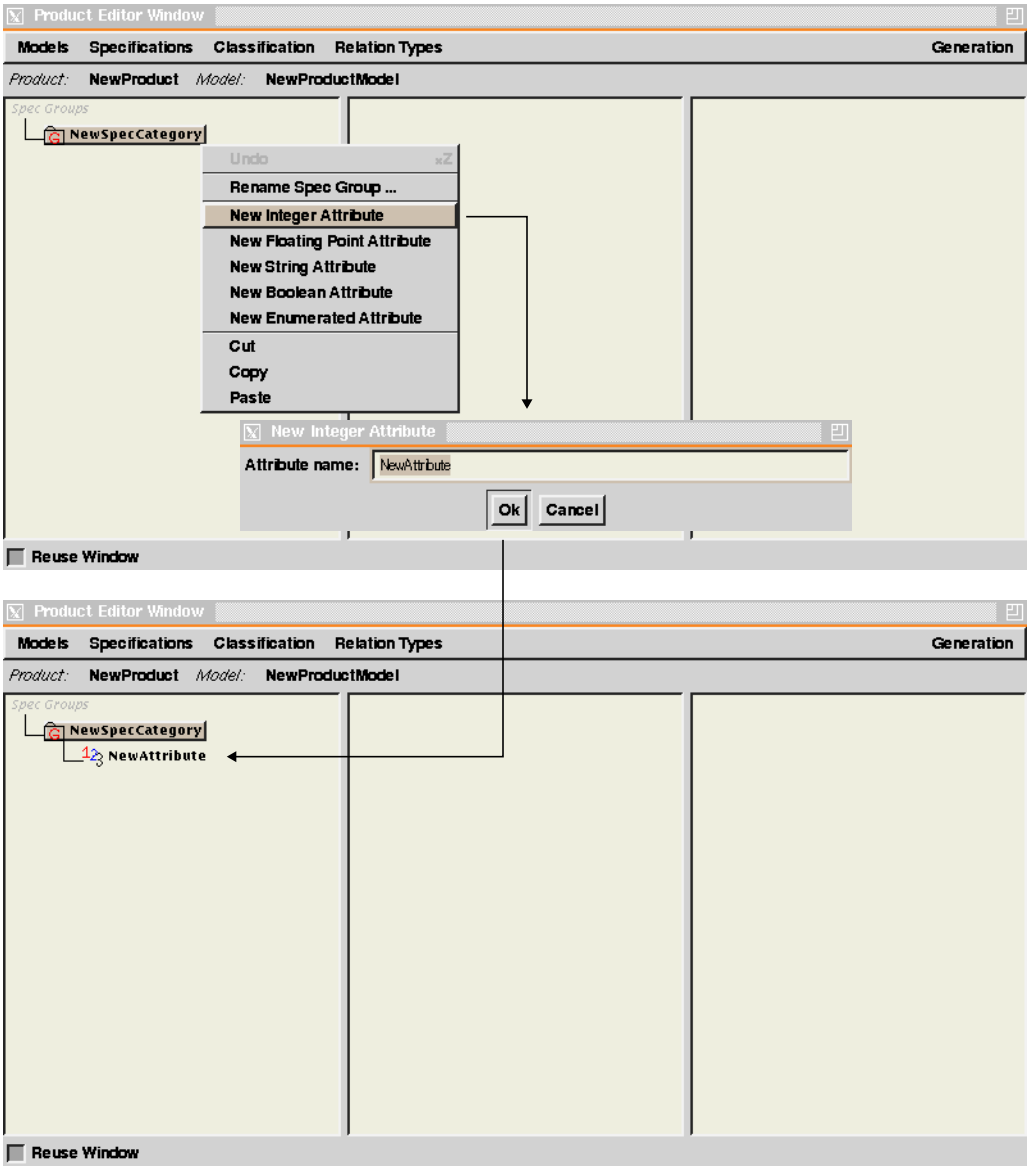
Interaction Diagram 49. Creating a Specification Primitive

Flow of Events:

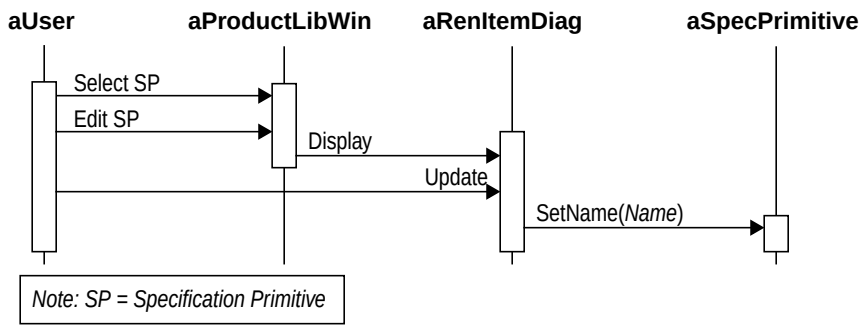
1. The user selects a “Create New Attribute” command from the Product Editor window. There are five types of attributes that user can choose from: integer, floating point, string, boolean, and enumerated.
2. The system displays the Set Name dialog prompting the user for a name.
3. The user enters a name and confirms.
4. The system creates a new Specification Primitive, adds it to the selected specification category, and updates the interface accordingly.

Preconditions: A Specification Category is selected in the Product Editor window.

Interface Design:



A 1.2.16 Rename Specification Primitive



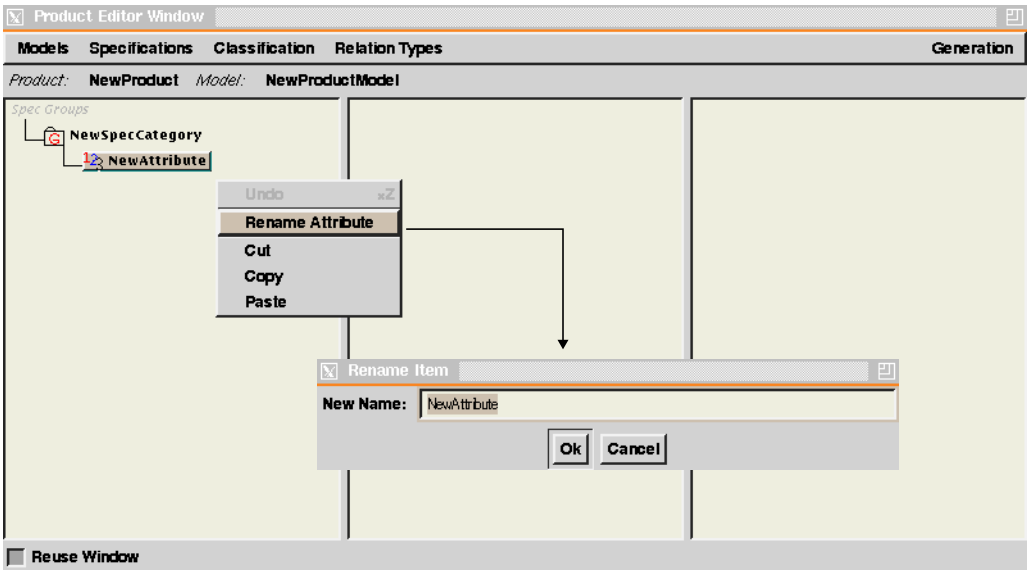
Interaction Diagram 50. Renaming a Specification Primitive

Flow of Events:

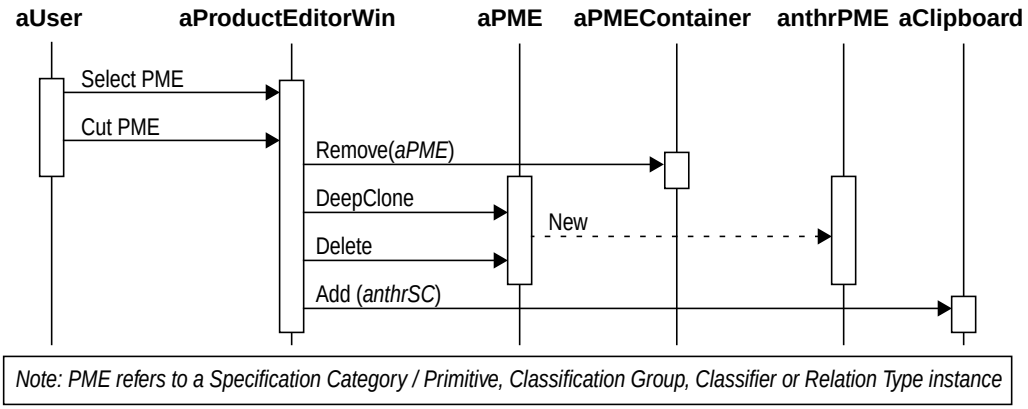
1. The user selects the "Rename Specification Primitive" command from the Product Editor window.
2. The system displays the Rename Item dialog box.
3. The user makes the changes in the dialog box and confirms.
4. The system updates its corresponding Specification Primitive object, and updates the interface accordingly.

Preconditions: A Specification Primitive is selected in the Product Editor window.

Interface Design:



A 1.2.17 Cut Product Modeling Element



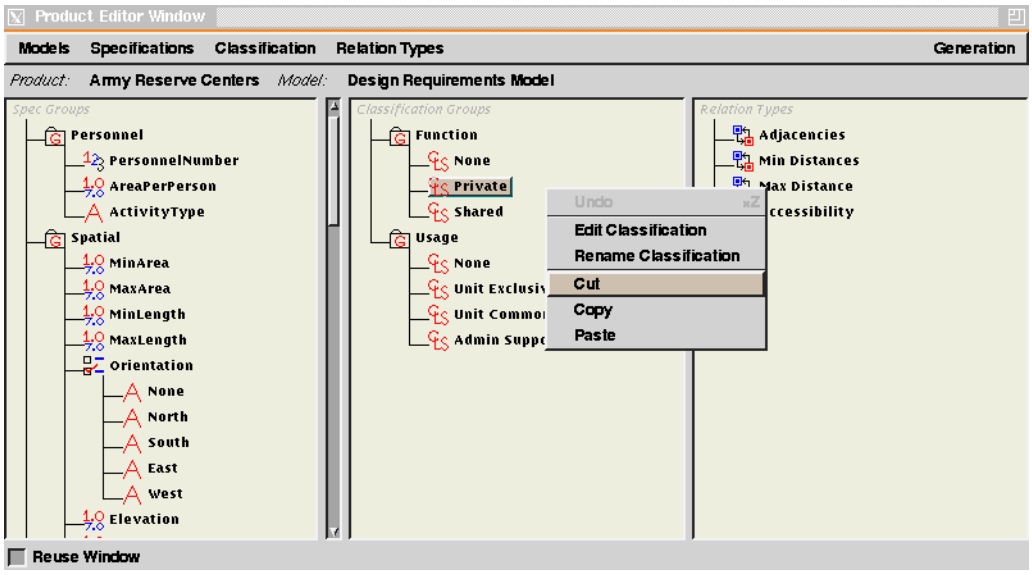
Interaction Diagram 51. Removing a Product Modeling Element

Flow of Events:

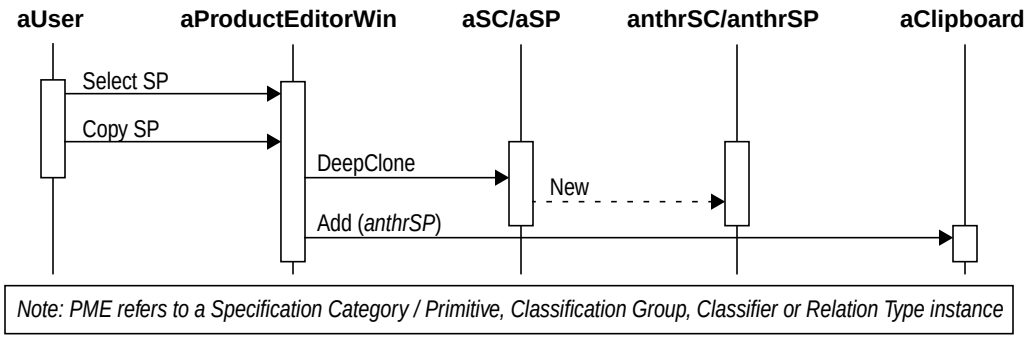
1. A Product Modeling Element refers any one of the the entities that constitute a Product Model. These are Specification Categories and Primitives, Classification Groups and Classifiers, and Relation Types. The user selects a Product Modeling Element from the Product Editor window.
2. The user selects the “Cut” command from the Product Editor window.
3. The system removes the selected object from the Product Model, and clones it.
4. The system then deletes the selected object and saves the clone in the system clipboard.

Preconditions: A Product Modeling Element is selected in the Product Editor window.

Interface Design:



A 1.2.18 Copy Product Modeling Element



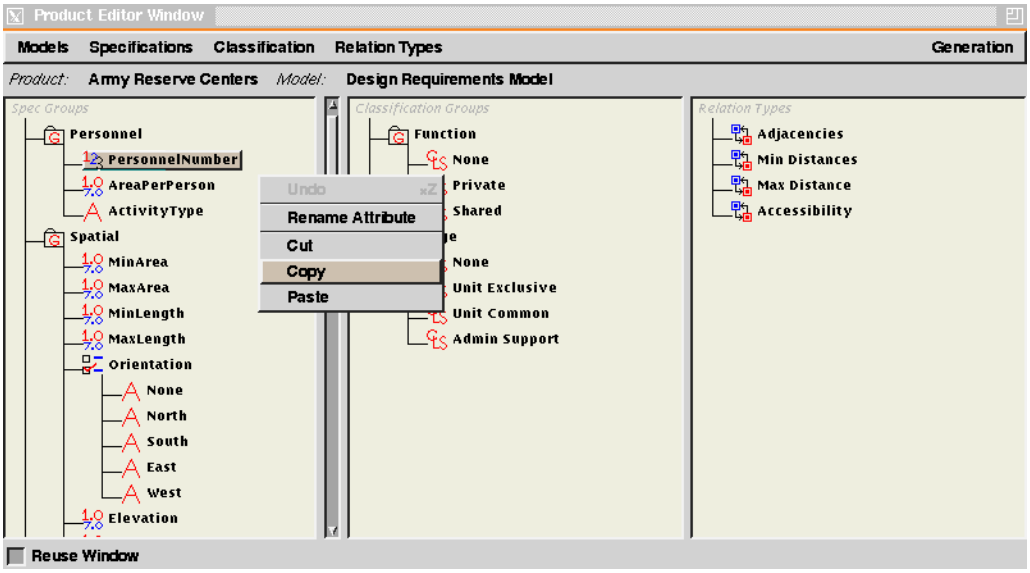
Interaction Diagram 52. Copying a Product Modeling Element

Flow of Events:

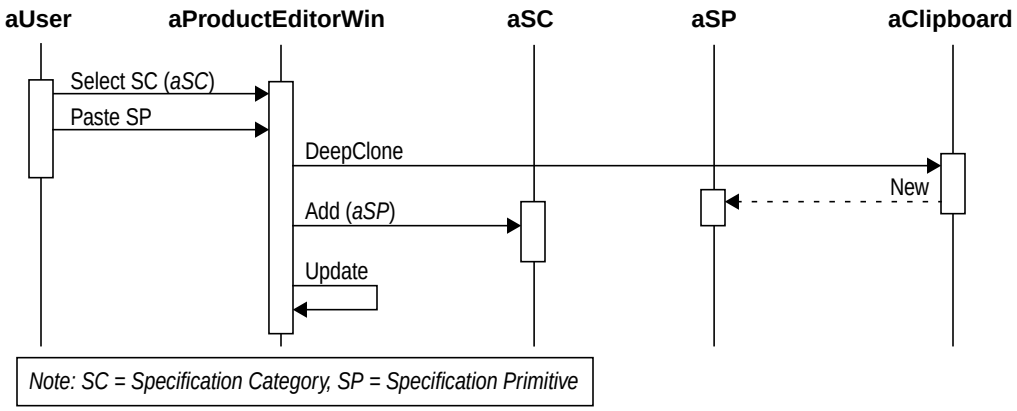
1. A Product Modeling Element refers any one of the the entities that constitute a Product Model. These are Specification Categories and Primitives, Classification Groups and Classifiers, and Relation Types. The user selects a Product Modeling Element from the Product Editor window.
2. The user selects the “Copy” command from the Product Editor window.
3. The system clones the selected object and saves the clone in the system clipboard.

Preconditions: A Product Modeling Element is selected in the Product Editor window.

Interface Design:



A 1.2.19 Paste Specification Primitive



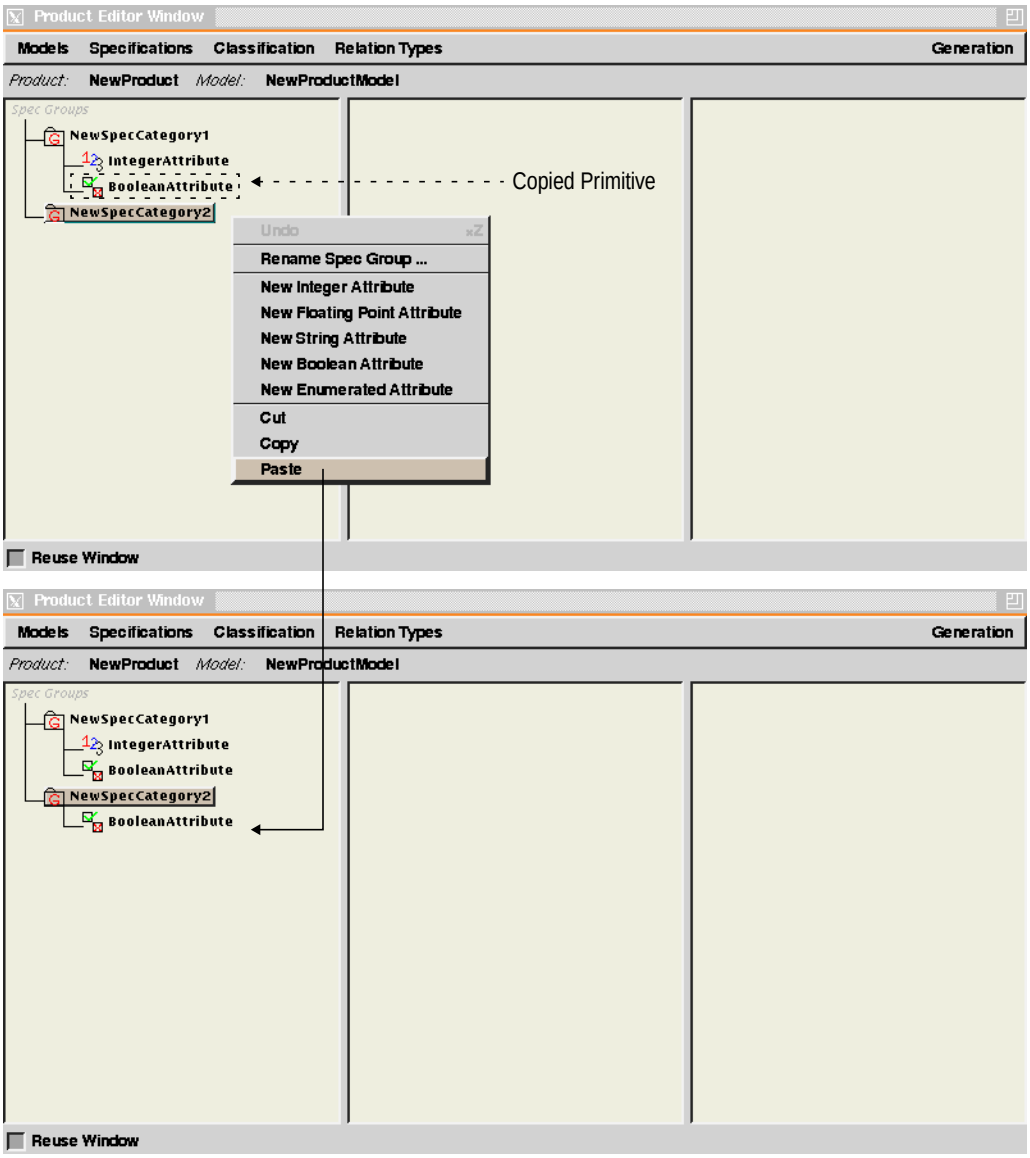
Interaction Diagram 53. Pasting a Specification Primitive

Flow of Events:

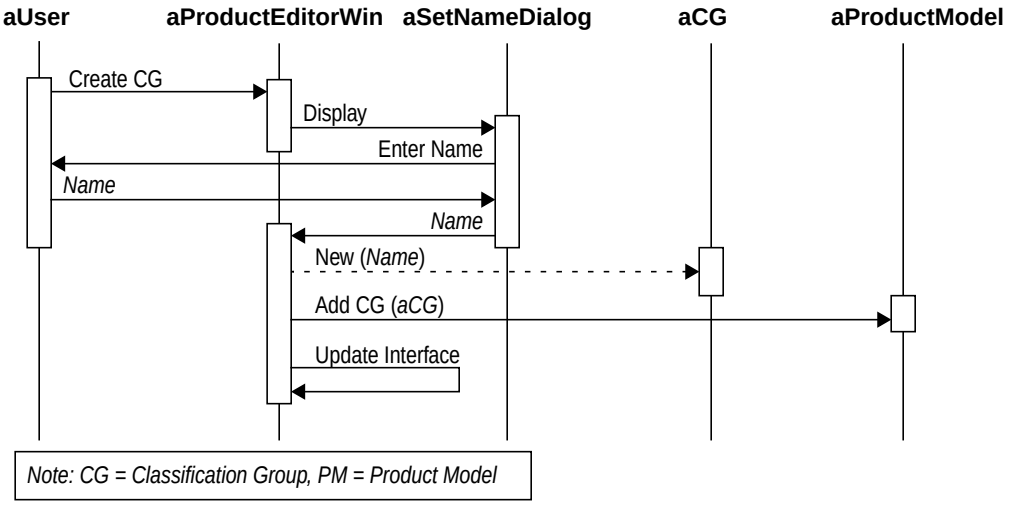
1. The user selects a Specification Category from the Product Editor window.
2. The user selects the “Paste” command from the Product Editor window.
3. If the clipboard object is a Specification Primitive, the system clones it and adds the clone to the selected Specification Category, and updates the display accordingly.

Preconditions: A Specification Category is selected in the Product Editor window and a Specification Primitive object exists in the clipboard.

Interface Design:



A 1.2.20 Create Classification Group



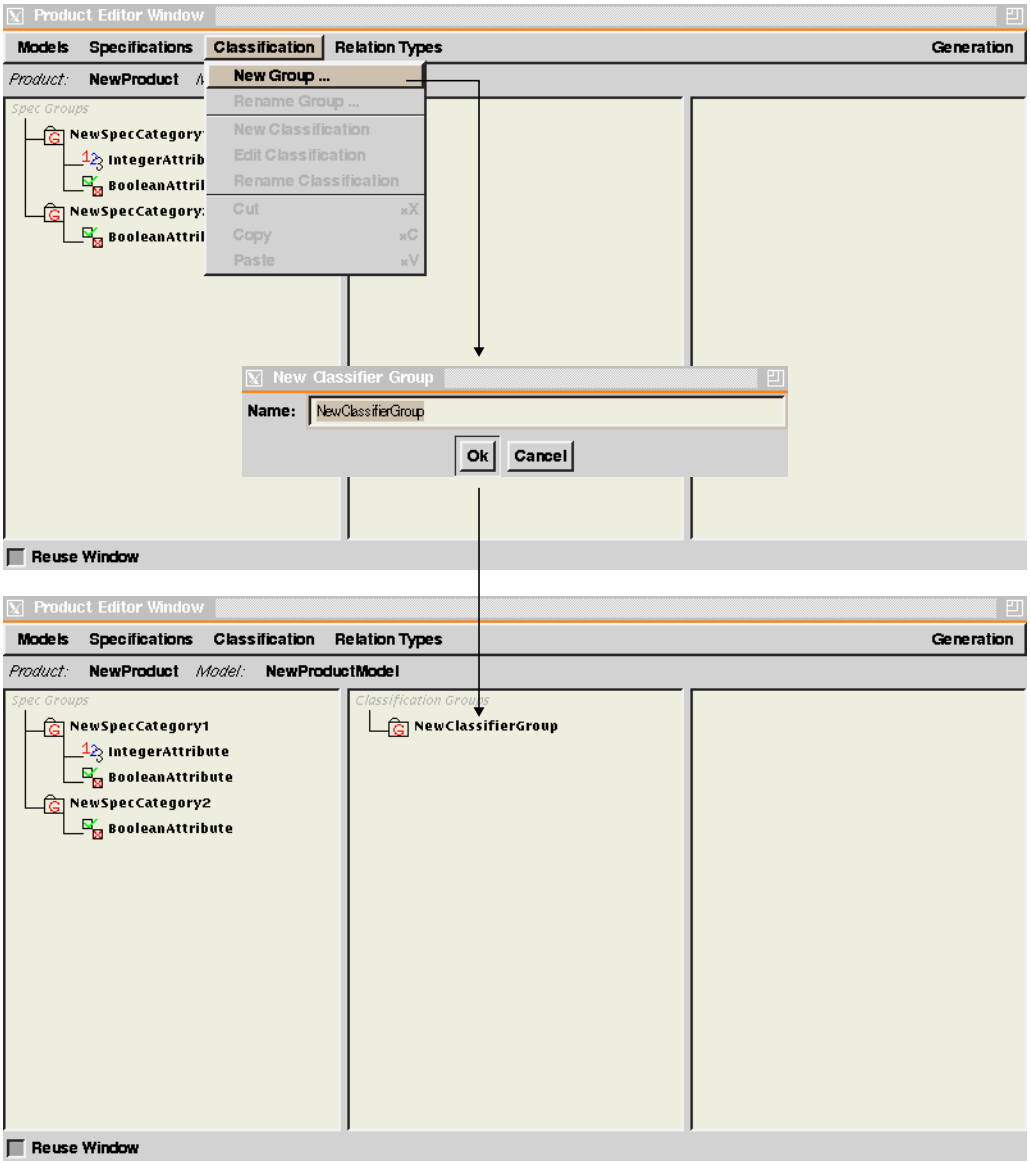
Interaction Diagram 54. Creating a Classification Group

Flow of Events:

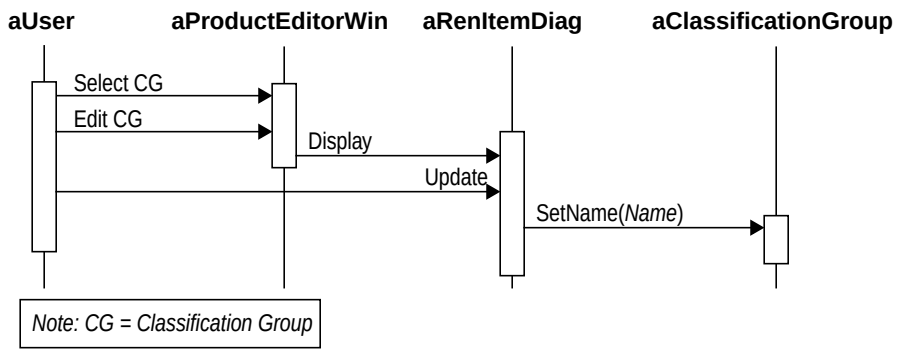
1. The user selects the "Create Classification Group" command from the Product Editor window.
2. The system displays the Set Name dialog prompting the user for a name.
3. The user types in the name and confirms.
4. The system creates a new classification group with the given name, adds it to the Product Model and updates the interface accordingly.

Preconditions: A Product Editor window is displayed.

Interface Design:



A 1.2.21 Rename Classification Group



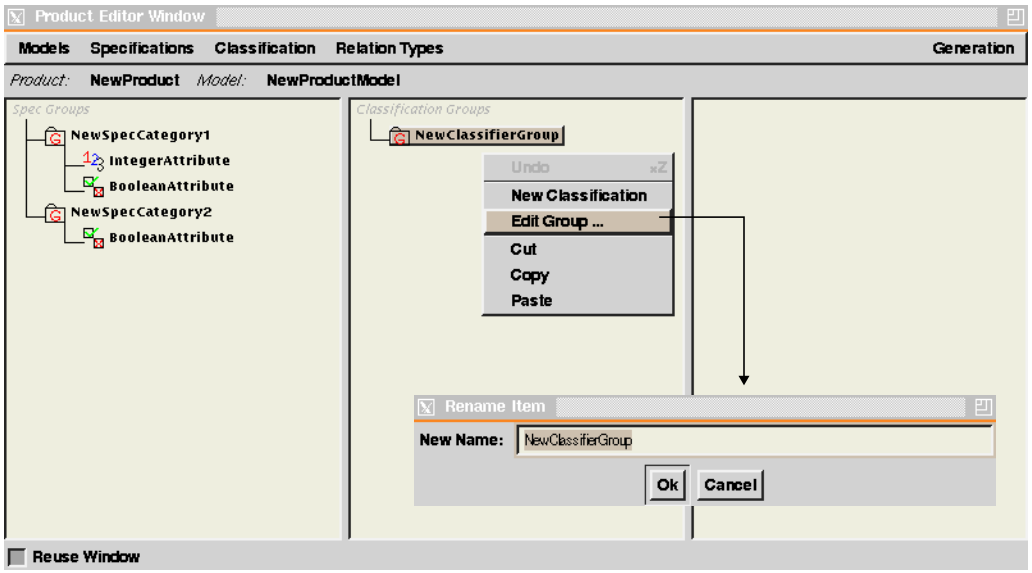
Interaction Diagram 55. Editing a Classification Group

Flow of Events:

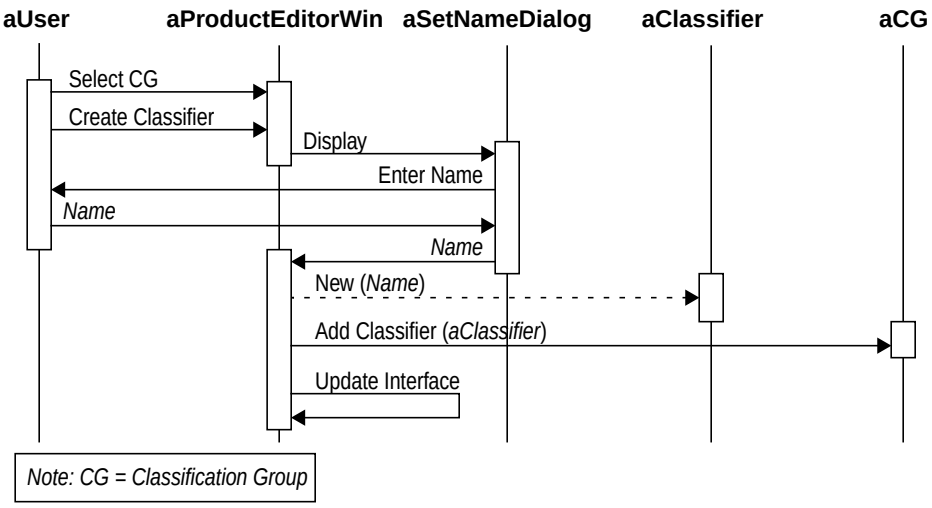
1. The user selects the "Edit Classification Group" command from the Product Editor window.
2. The system displays the Classification Group Editor window.
3. The user makes the changes in the window and confirms.
4. The system updates its corresponding classification group object, and updates the interface accordingly.

Preconditions: A classification group is selected in the Product Editor window.

Interface Design:



A 1.2.22 Create Classifier



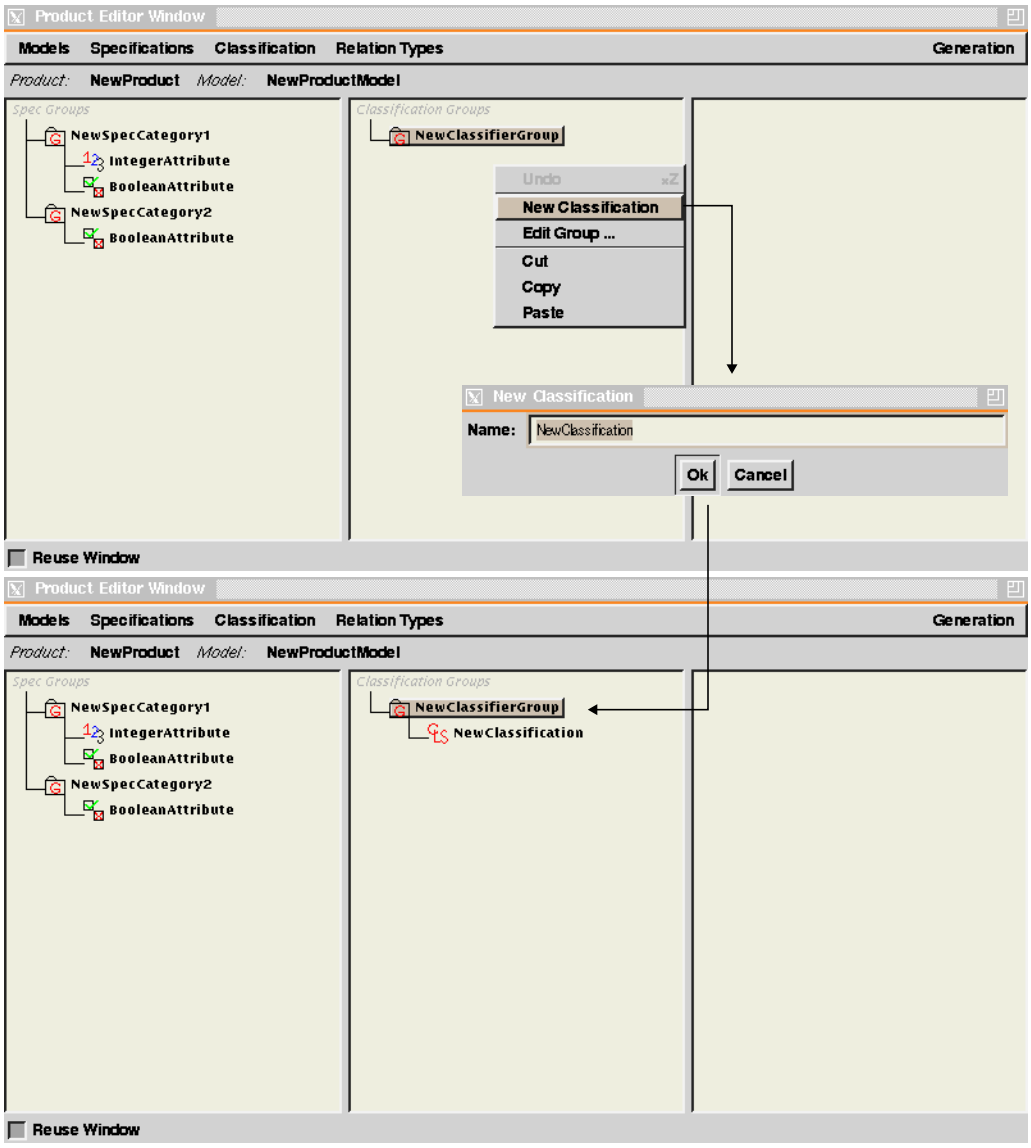
Interaction Diagram 56. Creating a Classifier

Flow of Events:

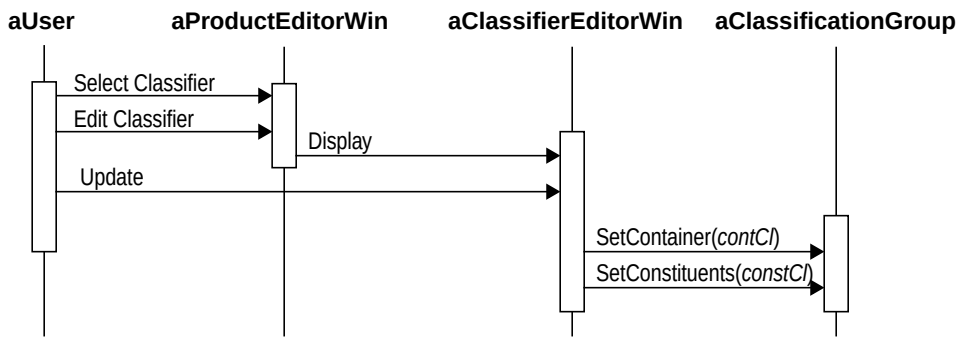
1. The user selects a classification group from the Product Editor window.
2. The user selects the “Create Classifier” command from the Product Editor window.
3. The system displays the Set Name dialog prompting the user for a name.
4. The user types in the name and confirms.
5. The system creates a new classifier with the given name, adds it to the selected classification group and updates the interface accordingly.

Preconditions: A classification group is selected in the Product Editor window.

Interface Design:



A 1.2.23 Edit Classifier



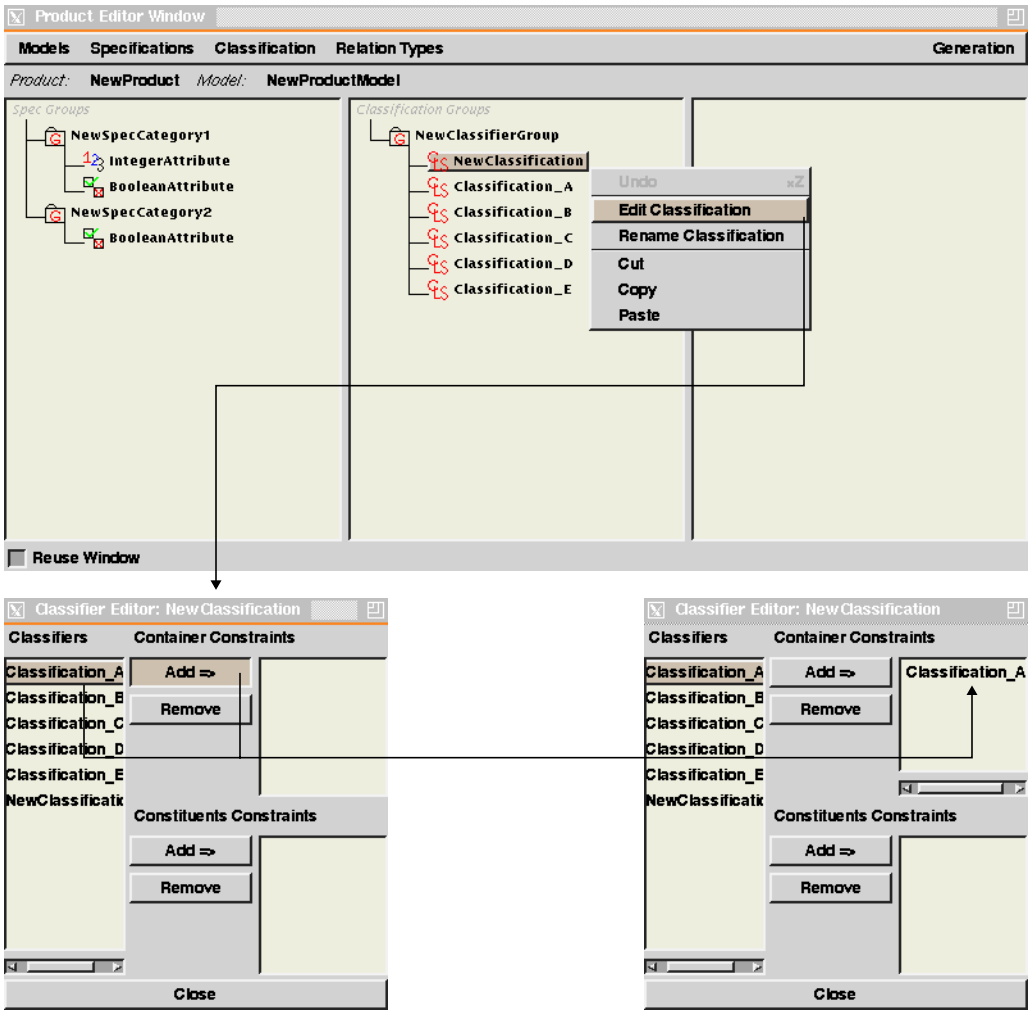
Interaction Diagram 57. Editing a Classifier

Flow of Events:

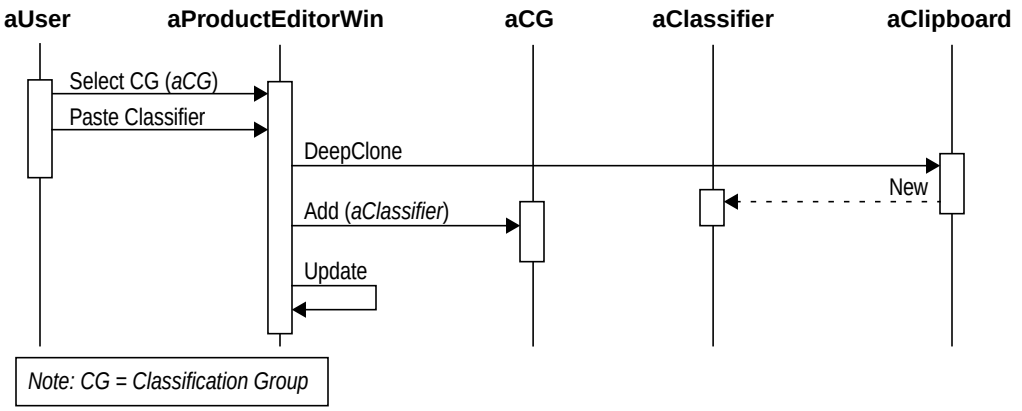
1. The user selects a classifier from the Product Editor window.
2. The user selects the “Edit Classifier” command from the Product Editor window.
3. The system displays the Classifier Editor window.
4. The user makes the changes in the window are:
   - unallowable classifications of the corresponding specification unit parent, and
   - unallowable classifications of the corresponding specification unit constituents
5. The user then confirms the changes and closes the Classifier Editor window.
6. The system updates its corresponding classifier object, and updates the interface accordingly.

Preconditions: A classifier is selected in the Product Editor window.

Interface Design:



A 1.2.24 Paste Classifier



Interaction Diagram 58. Pasting a Specification Primitive

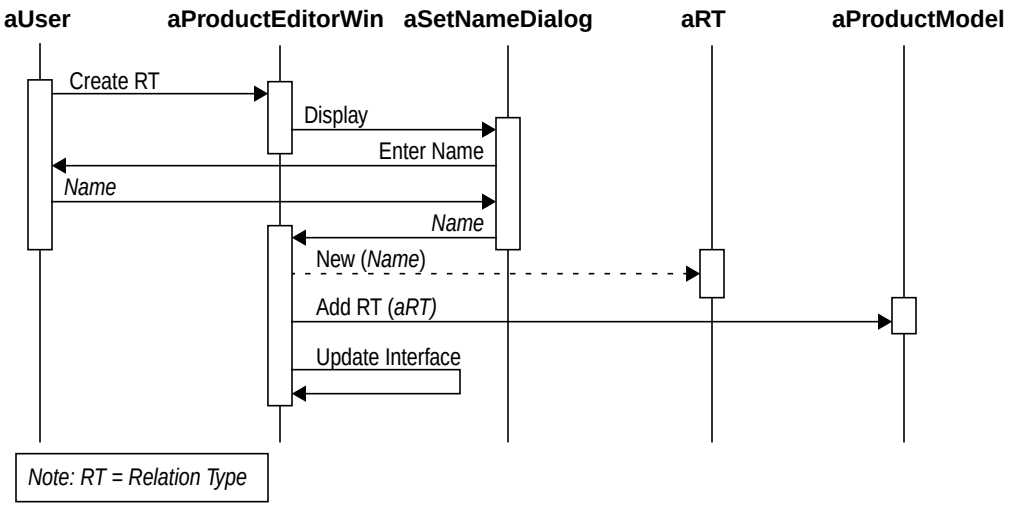
Flow of Events:

1. The user selects a classification group from the Product Editor window.
2. The user selects the “Paste Classifier” command from the Product Editor window.
3. If the clipboard object is a classifier, the system clones it and adds the clone to the selected classifier group, and updates the display accordingly.

Preconditions: A classification group is selected in the Product Editor window and a classifier object exists in the clipboard.

Interface Design:

A 1.2.25 Create Relation Type



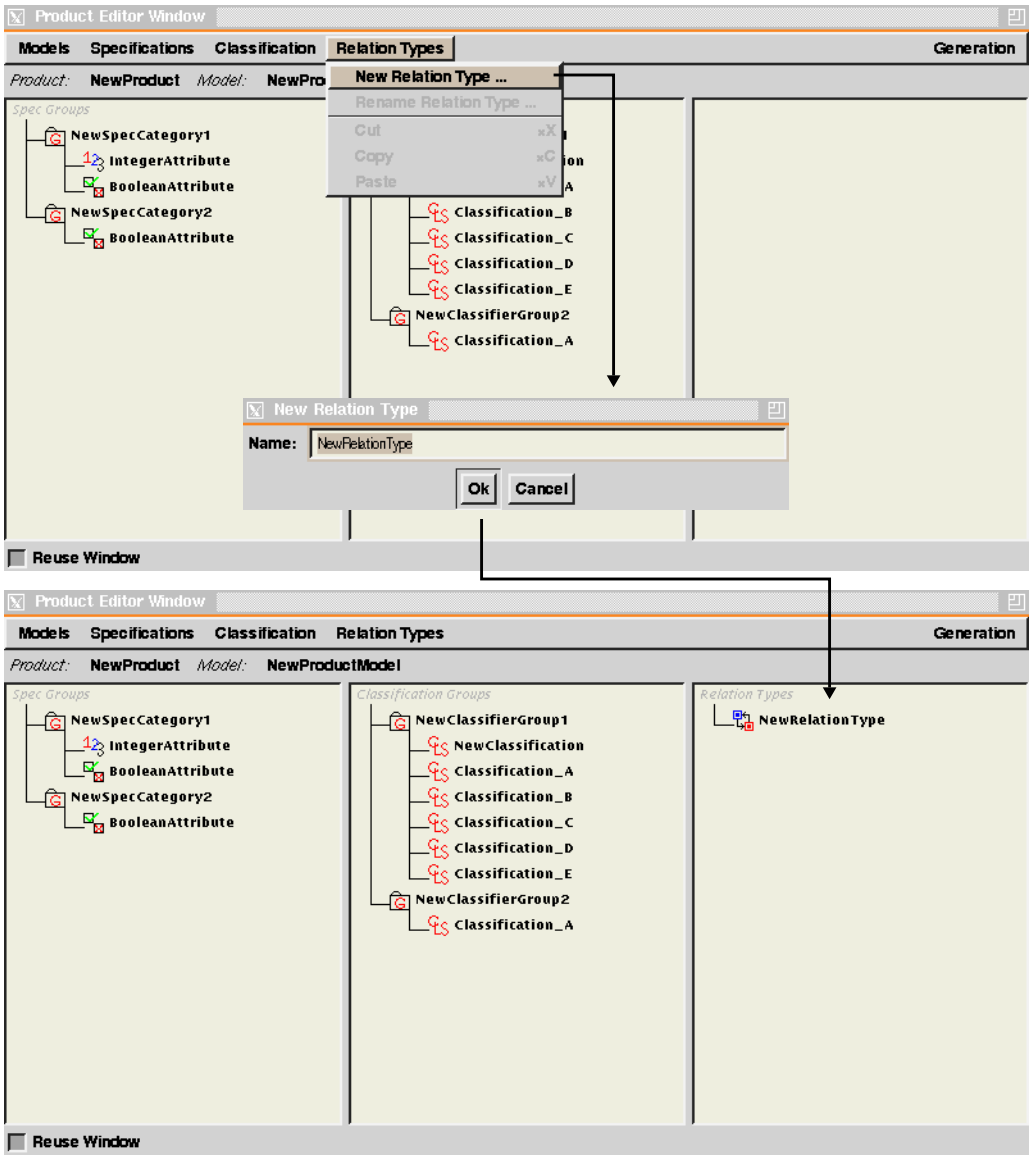
Interaction Diagram 59. Creating a Relation Type

Flow of Events:

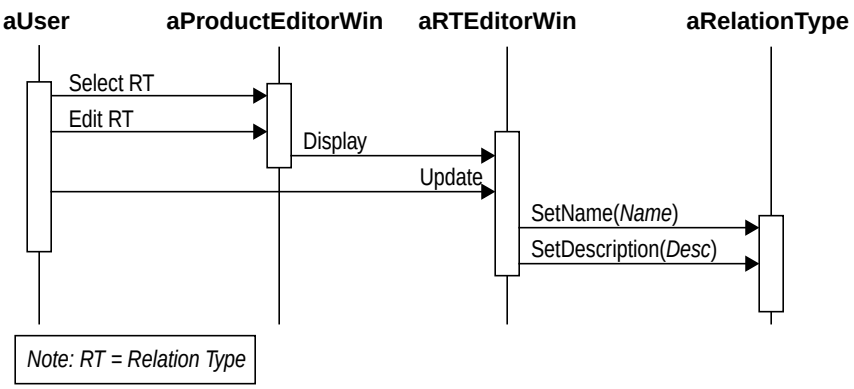
1. The user selects the “Create Relation Type” command from the Product Editor window.
2. The system displays the Set Name dialog prompting the user for a name.
3. The user types in the name and confirms.
4. The system creates a new Relation Type with the given name, adds it to the Product Model and updates the interface accordingly.

Preconditions: A Product Editor window is displayed.

Interface Design:



A 1.2.26 Rename Relation Type



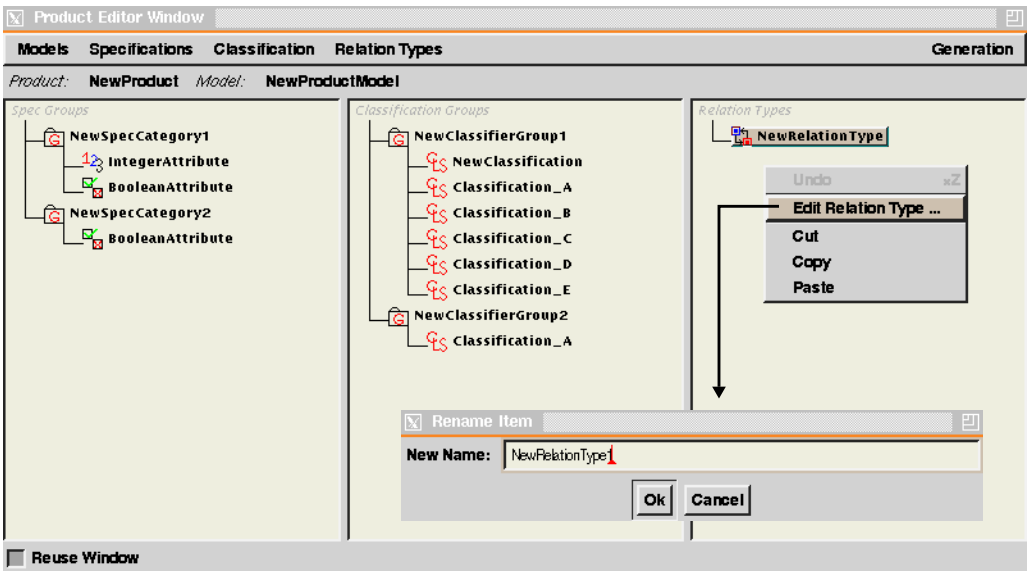
Interaction Diagram 60. Rename a Relation Type

Flow of Events:

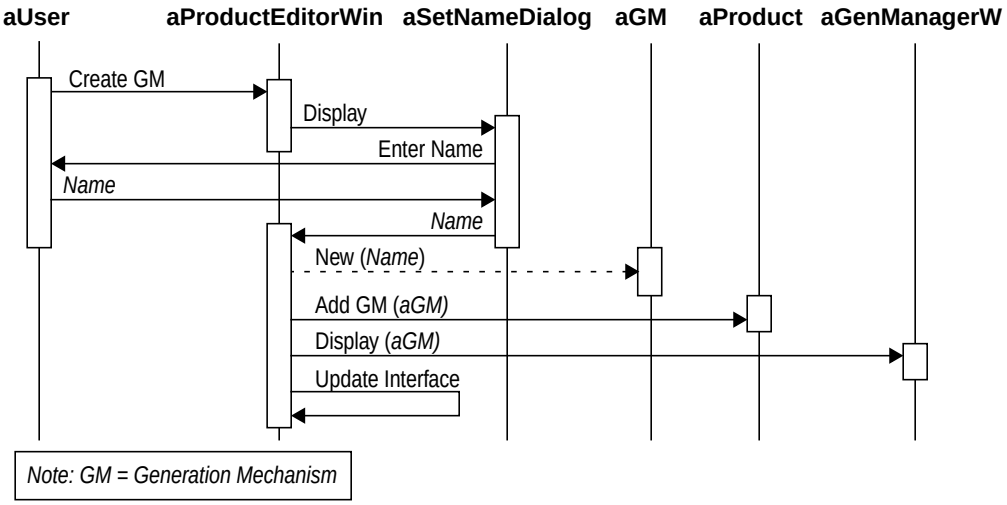
1. The user selects the “Rename Relation Type” command from the Product Editor window.
2. The system displays the Rename Item dialog box.
3. The user makes the changes in the dialog box and confirms.
4. The system updates its corresponding Relation Type object, and updates the interface accordingly.

Preconditions: A Relation Type is selected in the Product Editor window.

Interface Design:



A 1.2.27 Create Generation Mechanism



Interaction Diagram 61. Creating a Generation Mechanism

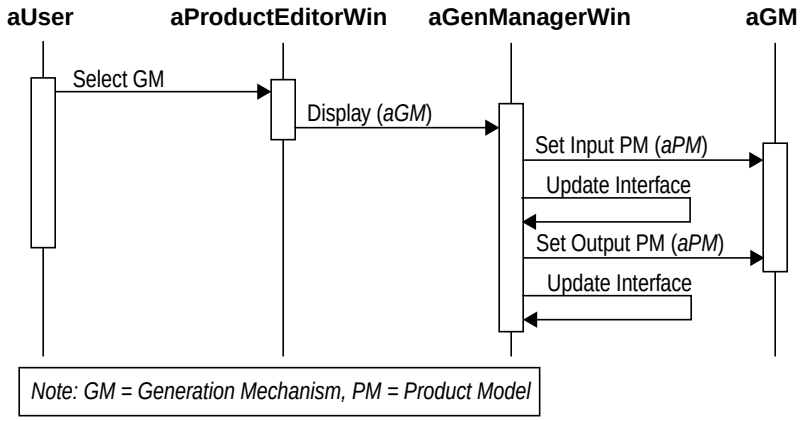
Flow of Events:

1. The user selects the "Create Generation Mechanism" command from the Product Editor window.
2. The system displays the Set Name dialog prompting the user for a name.
3. The user types in the name and confirms.
4. The system creates a new generation mechanism with the given name, adds it to the Product and displays the Generation Manager window showing the newly created mechanism.

Preconditions: A Product Editor window is displayed.

Interface Design:

A 1.2.28 Setup Generation Mechanism



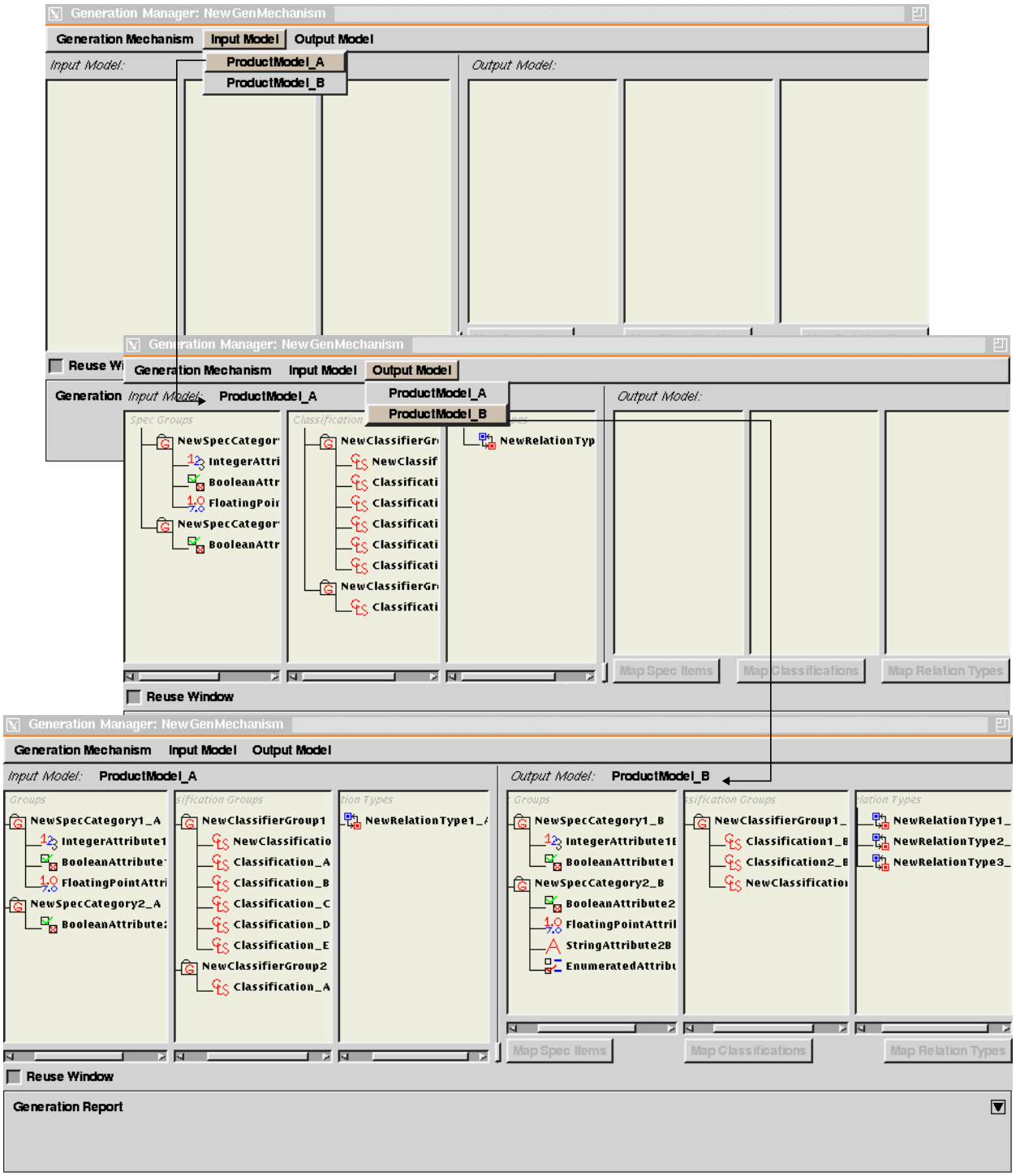
Interaction Diagram 62. Setting up a Generation Mechanism

Flow of Events:

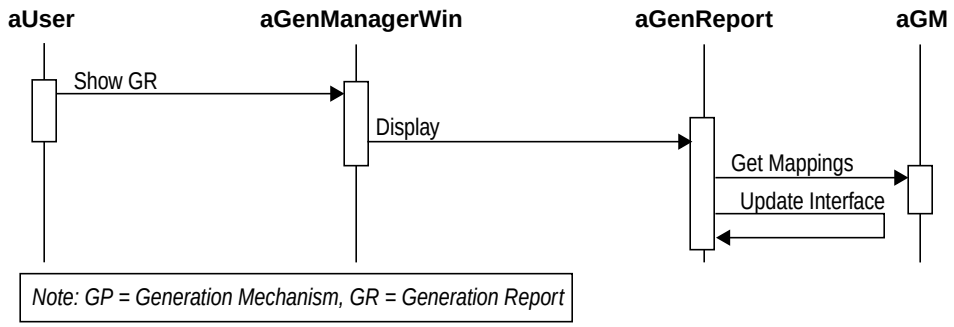
1. The user selects an existing Generation Mechanism from the Generation Menu of the Product Editor window, or creates a new one using the Create Generation Mechanism use case.
2. The system displays the Generation Manager window.
3. The user selects the input Product Model from the Input menu, then selects the output model from the output menu.
4. As changes are made, the corresponding Generation Mechanism object gets updated and the Generation Mechanism window updates its interface to reflect the changes made.

Preconditions: A Generation Mechanism exists in the current Product.

Interface Design:



A 1.2.29 Show Generation Report



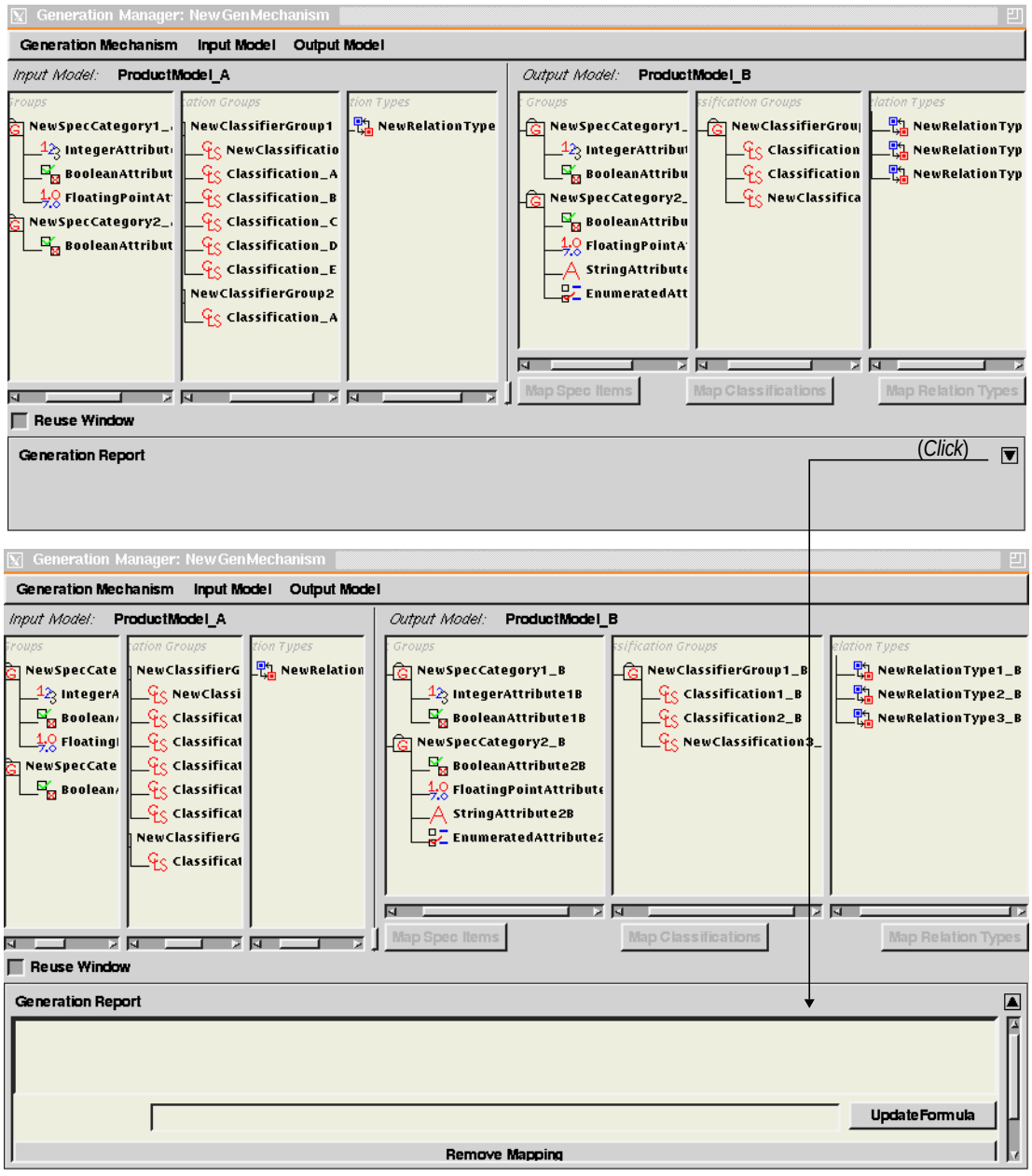
Interaction Diagram 63. Displaying Generation Report

Flow of Events:

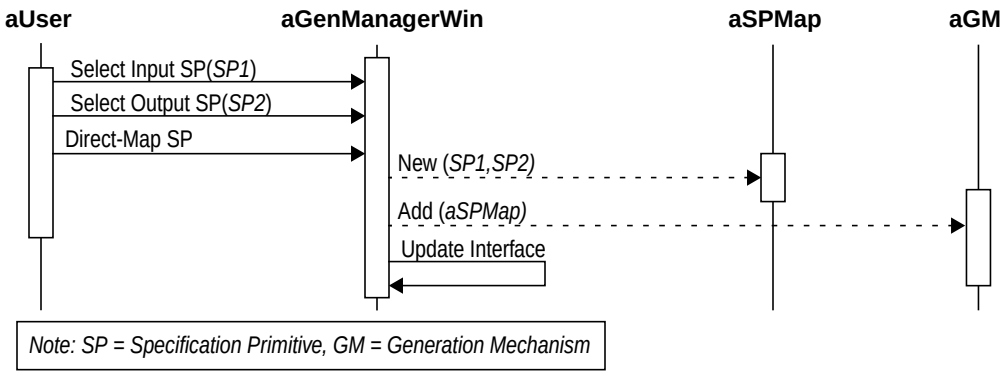
1. In the Generation Manager window, the user selects the "Show Generation Report" command.
2. The system displays the Generation Report window updated with the latest values from the current Generation Mechanism.

Preconditions: The Generation Manager window is displayed.

Interface Design:



A 1.2.30 Direct-Map Specification Primitives



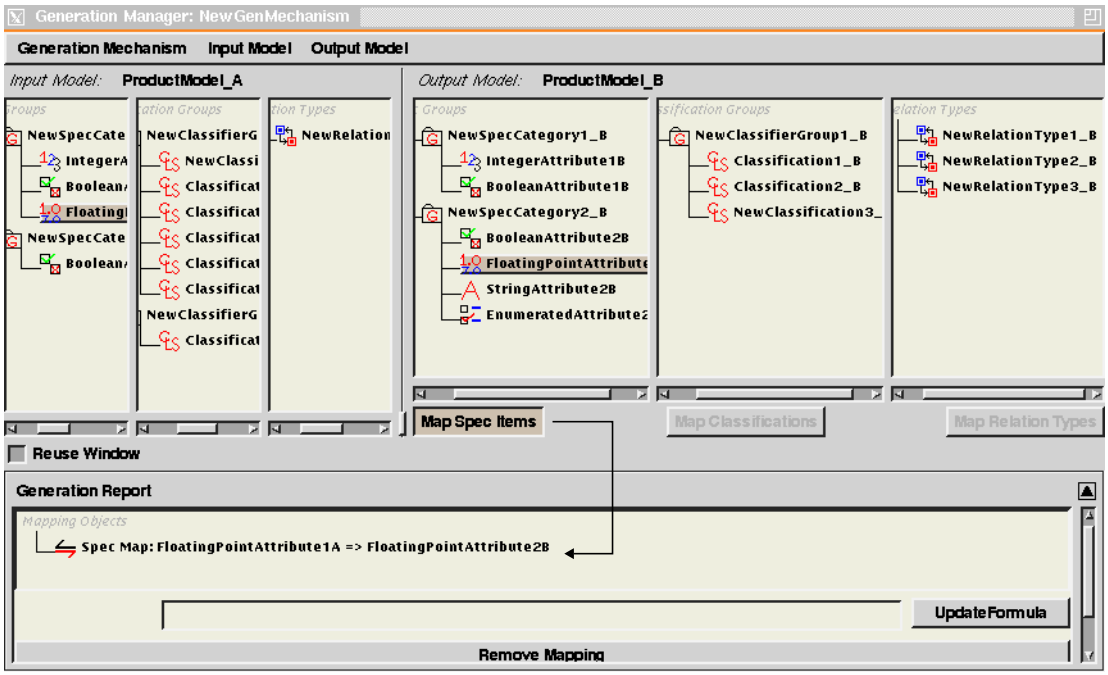
Interaction Diagram 64. Direct-Mapping Specification Primitives

Flow of Events:

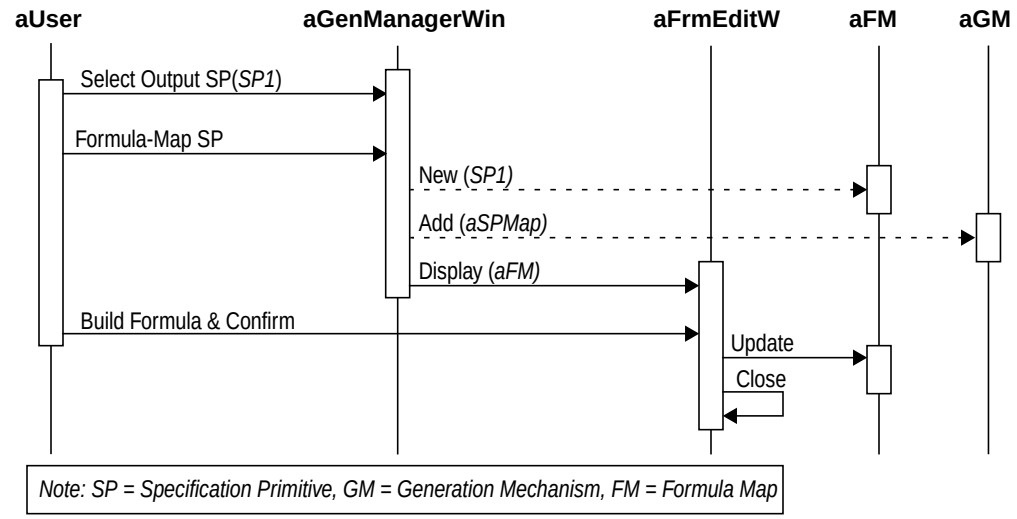
1. In the Generation Manager window, the user selects a Specification Primitive from the input Product Model, and another from the output Product Model, and selects the "Direct-Map Primitives" command.
2. The system creates a new Specification Primitive Map object and adds it to the current Generation Mechanism and updates the Generation Report accordingly.

Preconditions: A Specification Primitives of compatible types are selected, one from the input Product Model and another from the output model.

Interface Design:



A 1.2.31 Formula-Map Specification Primitive



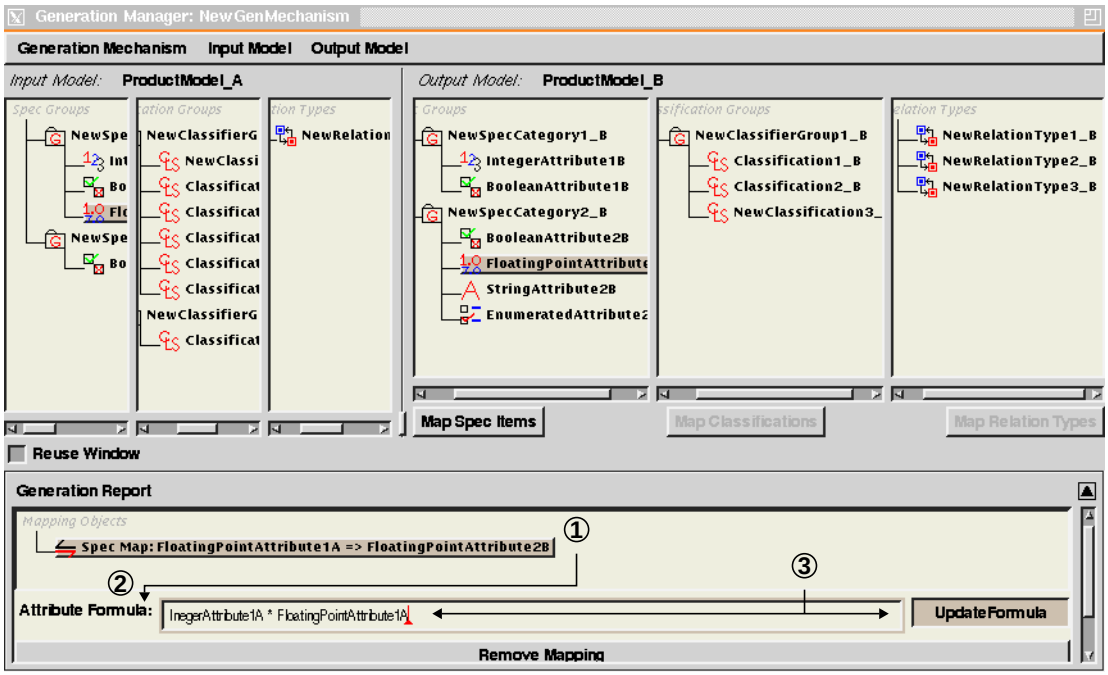
Interaction Diagram 65. Formula-Mapping Specification Primitives

Flow of Events:

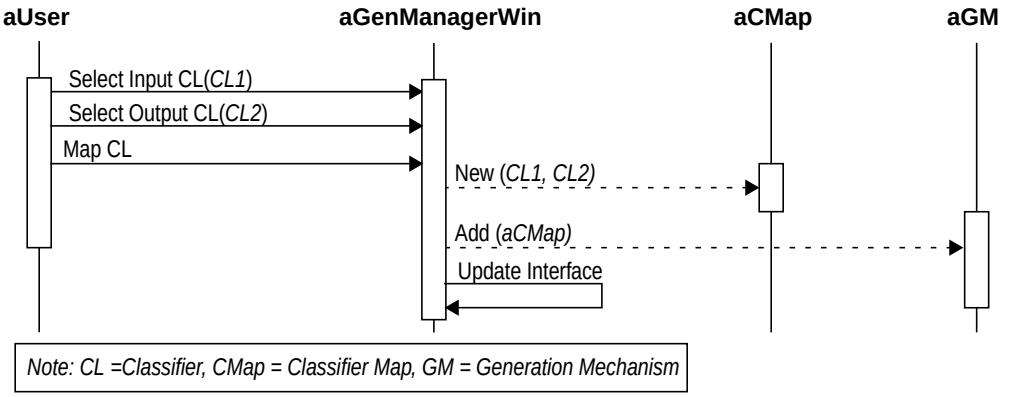
1. In the Generation Report section of the Generation Manager window, the user selects a Specification Primitive Map.
2. The system activates the Attribute Formula field where the Modeler can enter the mathematical expression to be used by the selected mapping.
3. The user builds the formula in that window by combining Specification Primitives from the input model with the mathematical operators (*, /, -, +) and confirms.
4. The system updates the corresponding Formula Map object.

Preconditions: A Specification Primitive Map exist in the Generation Report.

Interface Design:



A 1.2.32 Map Classifiers



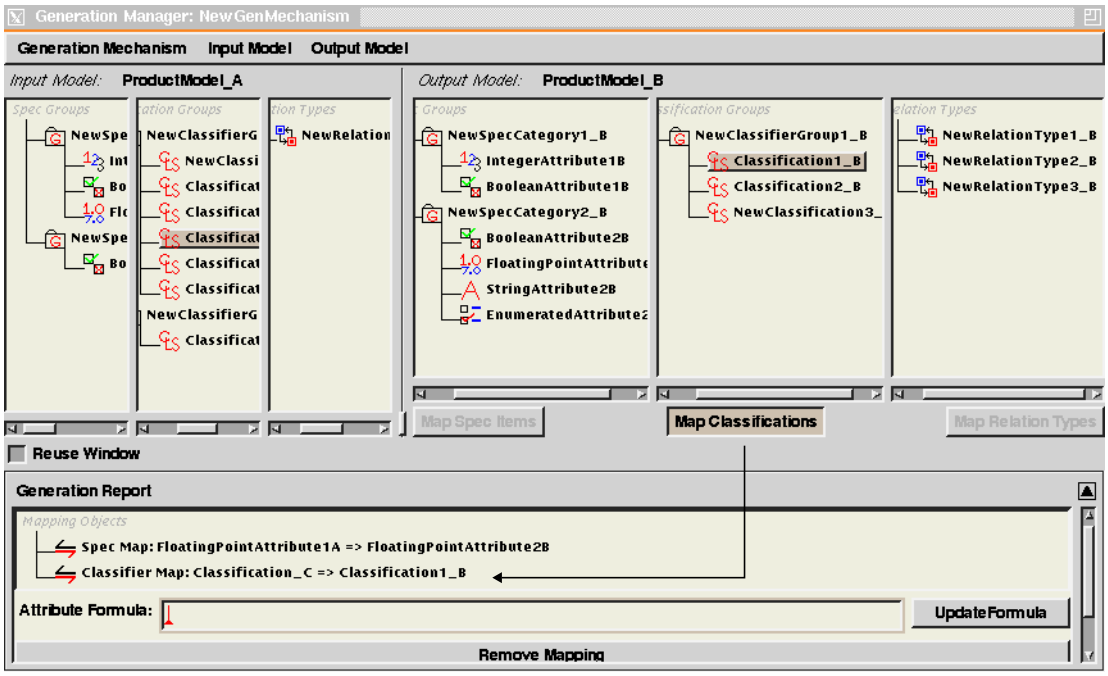
Interaction Diagram 66. Mapping Classifiers

Flow of Events:

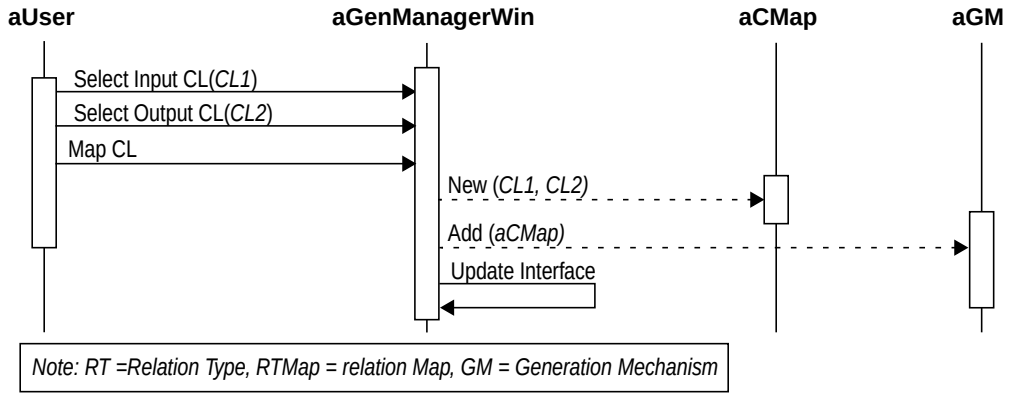
1. In the Generation Manager window, the user selects a Classifier from the input Product Model, and another from the output Product Model, and selects the "Map Classifier" command.
2. The system creates a new Classifier Map object and adds it to the current Generation Mechanism and updates the interface accordingly.

Preconditions: Classifiers exist in the output and input Product Models.

Interface Design:



A 1.2.33 Map Relation Type



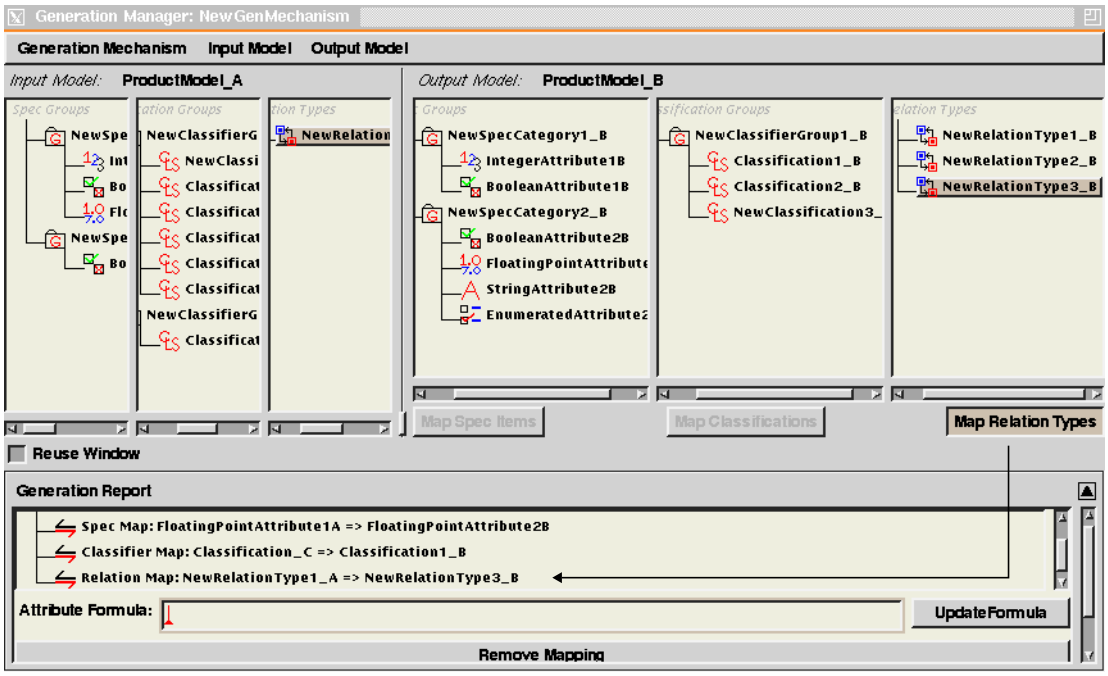
Interaction Diagram 67. Mapping Relation Types

Flow of Events:

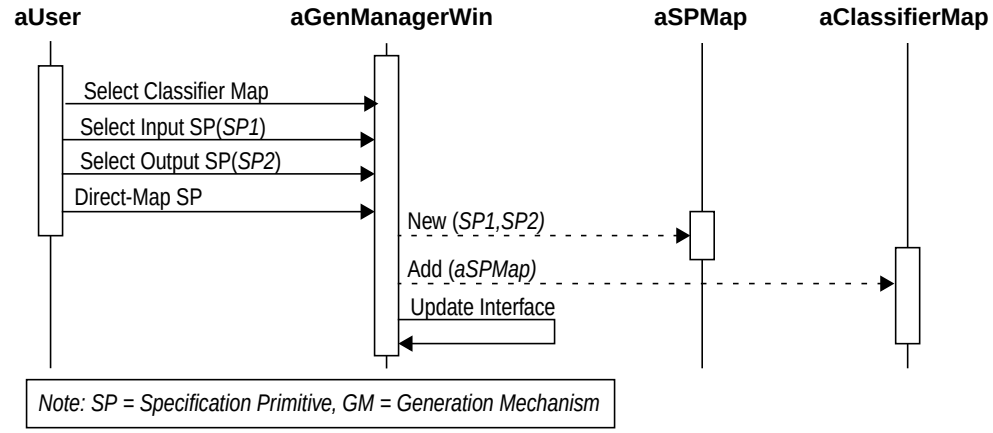
1. In the Generation Manager window, the user selects a Relation Type from the input Product Model, and another from the output Product Model, and selects the "Map Relation Type" command.
2. The system creates a new Relation Map object and adds it to the current Generation Mechanism and updates the interface accordingly.

Preconditions: Relation Types exist in the output and input Product Models.

Interface Design:



A 1.2.34 Create Specialized Mapping



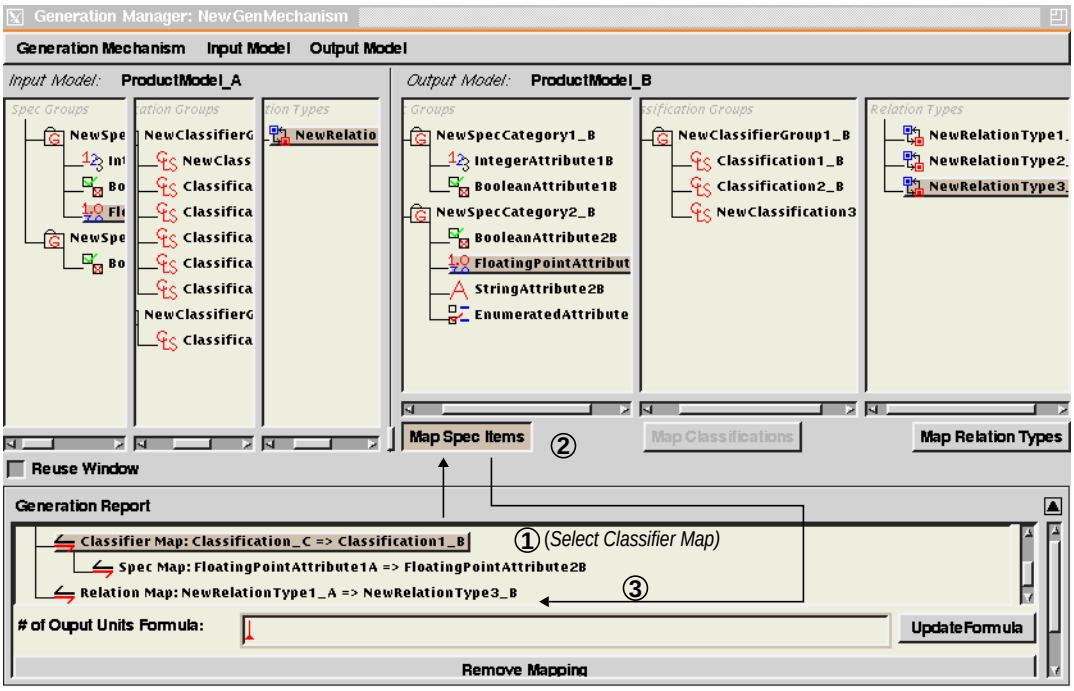
Interaction Diagram 68. Setting Generation Parameters

Flow of Events:

1. In the Generation Manager window, the user selects a Classifier Map from the Generation Report. The user then applies the Direct-Map Specification Primitives use case to create Specification Primitive Maps.
2. Since a Classifier Map is selected, the newly created maps are added as sub-nodes of the selected Classifier Map, indicating that they are applied only when their Classifier Map is used.

Preconditions: A Classifier Map exists in the Generation Report.

Interface Design:

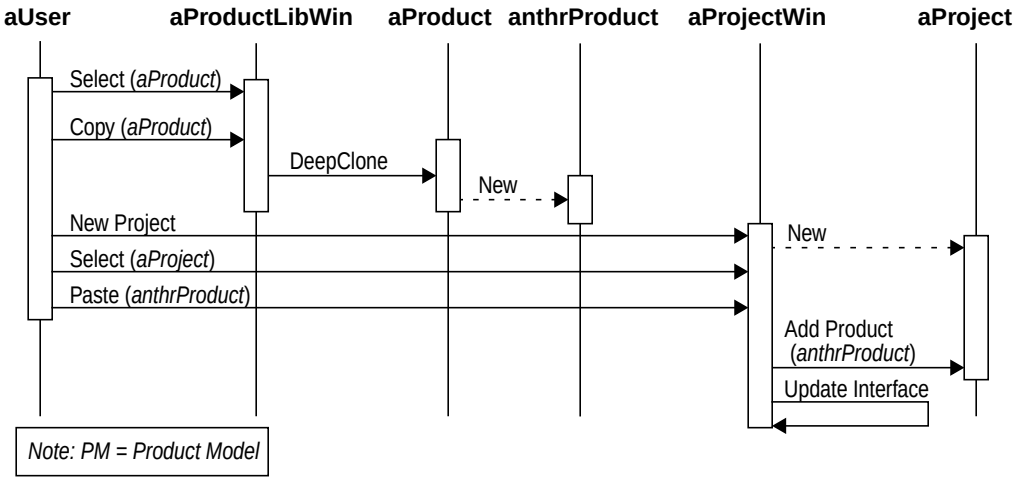


A 1.3 Use Cases for Designer

The use case in this section describe the functionalities pertaining to the creation of products using the prototypical Products and Product Models stored in the system library. It allows for the manipulation of these specifications as well as generating outputs according to the generation mechanisms associated with the product. The use cases are:

1. Copy Product [page 156]
2. Copy Specification Unit [page 162]
3. Create Project [page 147]
4. Create Specification Unit [page 159]
5. Cut Product [page 157]
6. Cut Specification Unit [page 161]
7. Edit Specification Unit [page 165]
8. Generate Output [page 167]
9. Open Project [page 149]
10. Paste Product [page 158]
11. Paste Specification Unit [page 163]
12. Rename Product [page 155]
13. Rename Project [page 153]
14. Save Project [page 151]
15. Select Generation Mechanism [page 166]
16. Set Input Product Model [page 154]

A 1.3.1 Create Project



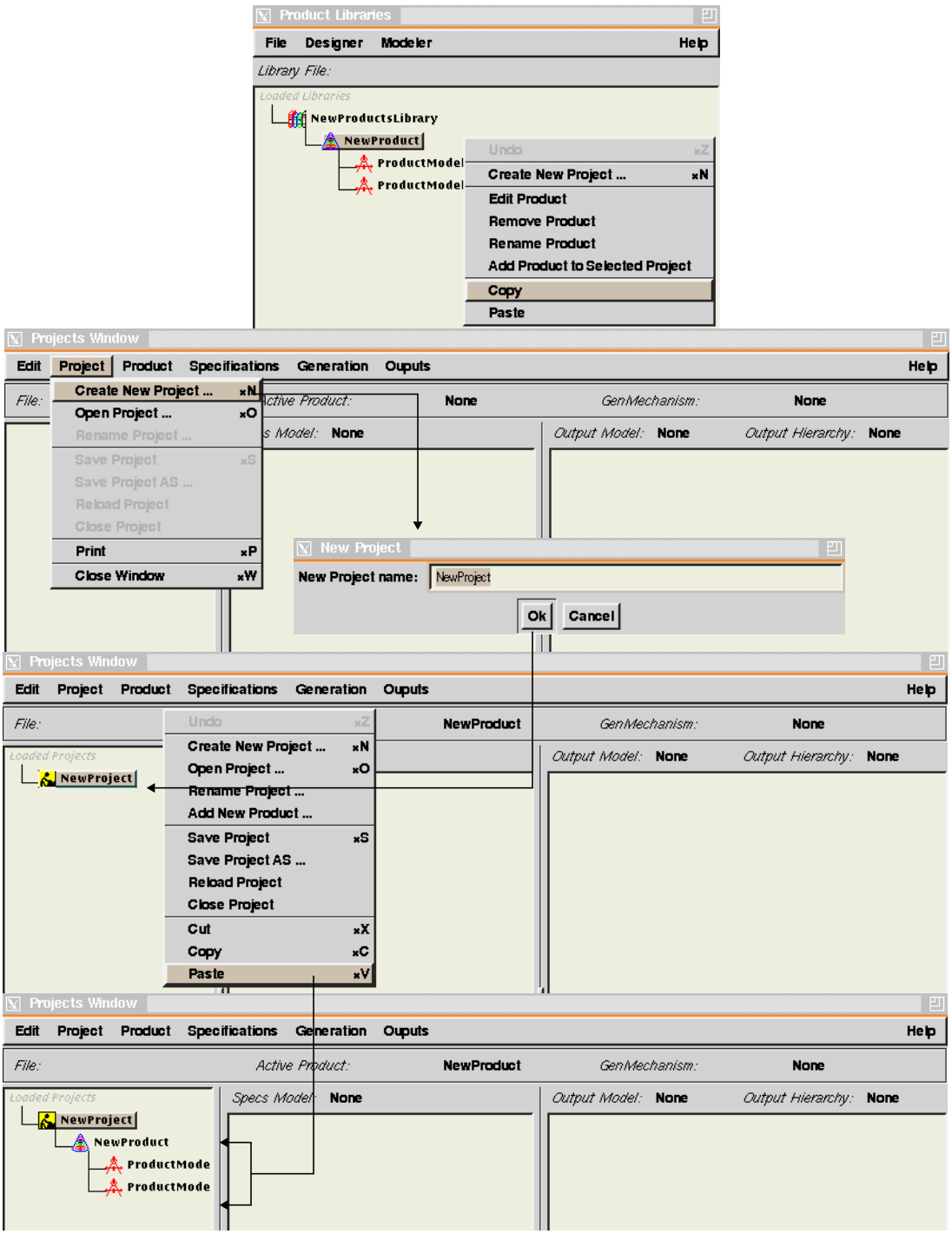
Interaction Diagram 69. Creating a New Project

Flow of Events:

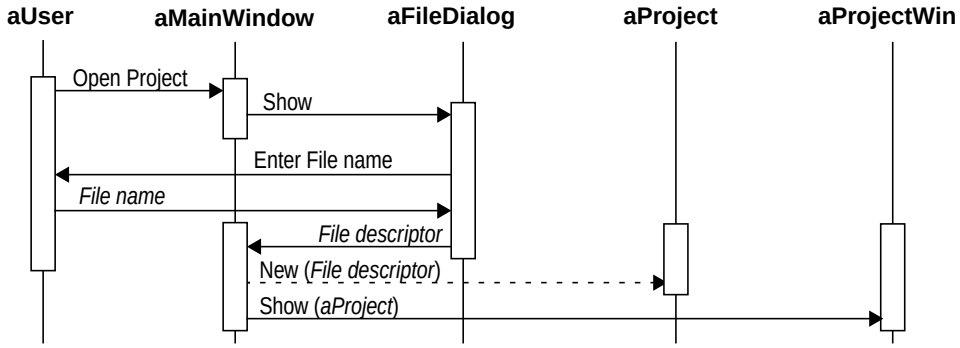
1. The user selects and copies a Product from the Product Library window.
2. The user selects the “Create New Project” command from the Projects window.
3. The system creates a new Project and displays it in the Projects window.
4. The user selects the newly created project, then selects the “Paste” command.
5. The system adds the copied product to the selected project and updates the interface accordingly.

Preconditions: A Product is selected in the Product Library window.

Interface Design:



A 1.3.2 Open Project



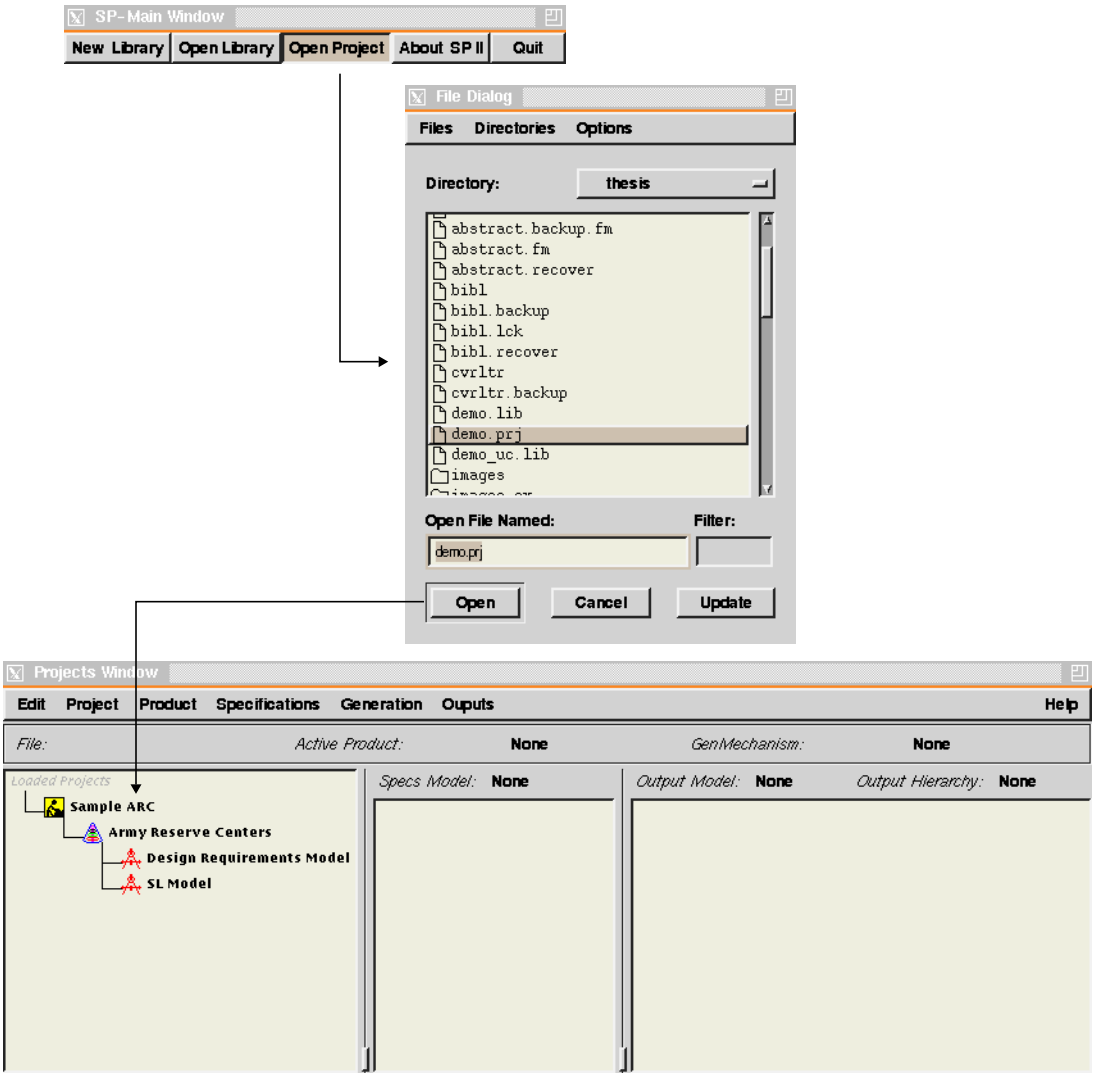
Interaction Diagram 70. Opening an Existing Project

Flow of Events:

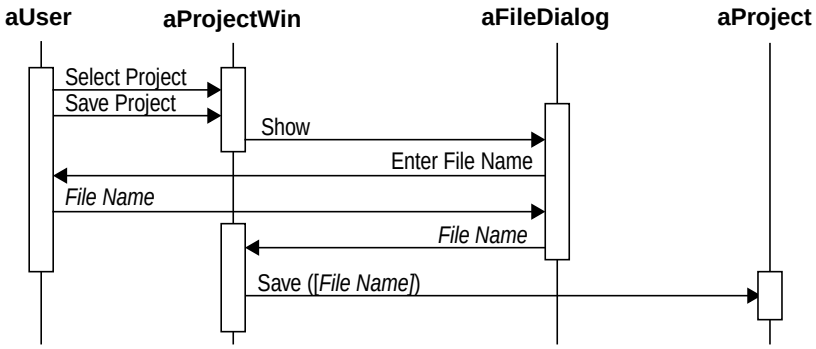
1. The user selects the "Open Project" command from the Main window.
2. The system displays the File dialog prompting the user for a file name.
3. The user types in the name or selects the file by navigating throught the file system and confirms.
4. The system creates a new Project object and loads its information from the file and displays the Project in the Projects window.

Preconditions: Main window is displayed, and a Project file exists on the file system.

Interface Design:



A 1.3.3 Save Project



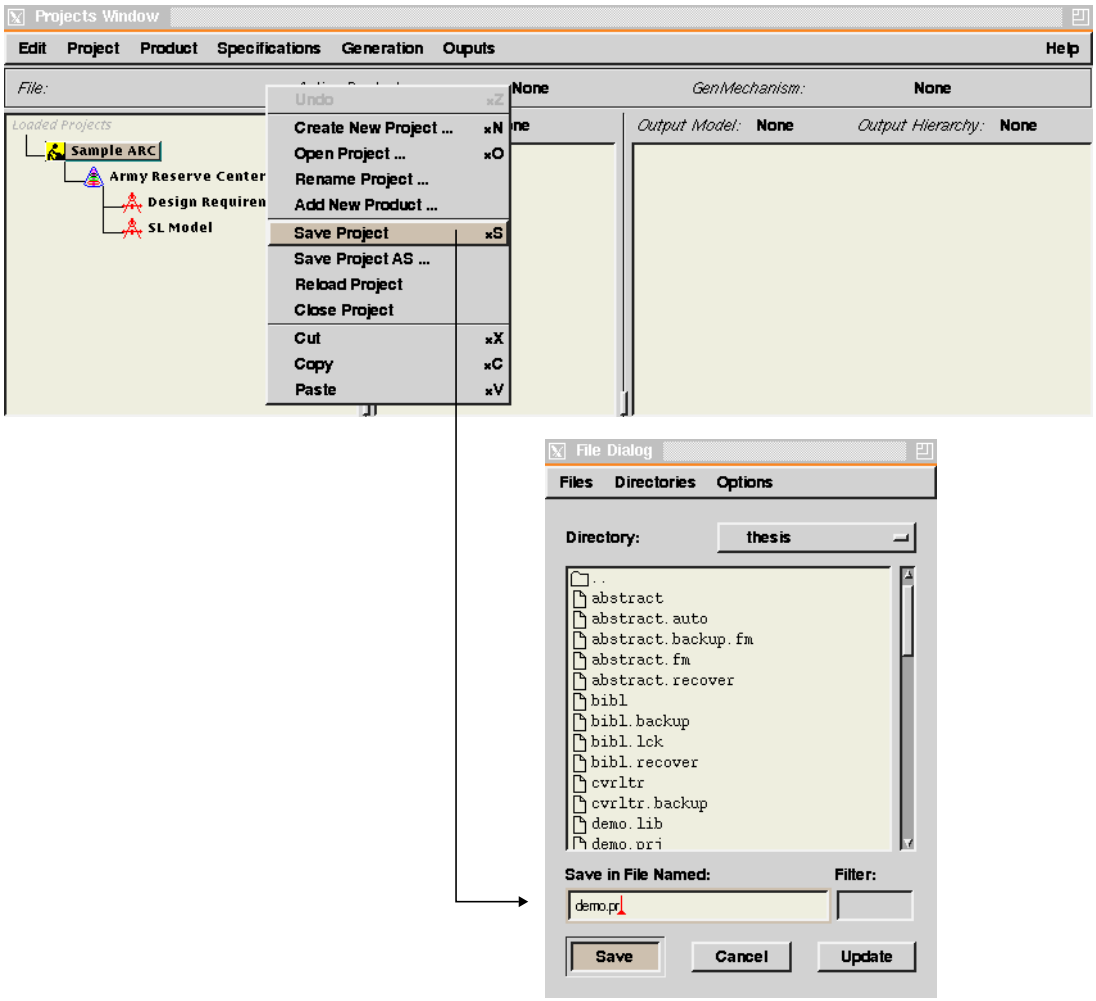
Interaction Diagram 71. Saving a Project

Flow of Events:

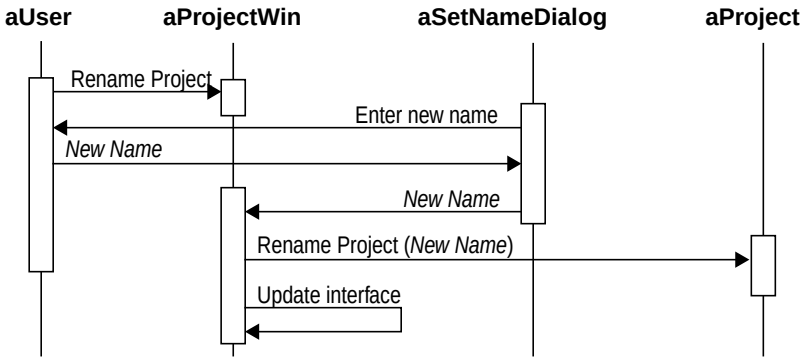
1. the user selects a Project from the Projects window.
2. The user selects the “Save Project” command from the Projects window.
3. If the Project has been save before the system moves to step (6)
4. The system displays the File dialog prompting the user for a file name.
5. The user types in the name and confirms.
6. The system send a “Save” command to the Product Library object.

Preconditions: A Project is selected in the Projects window.

Interface Design:



A 1.3.4 Rename Project



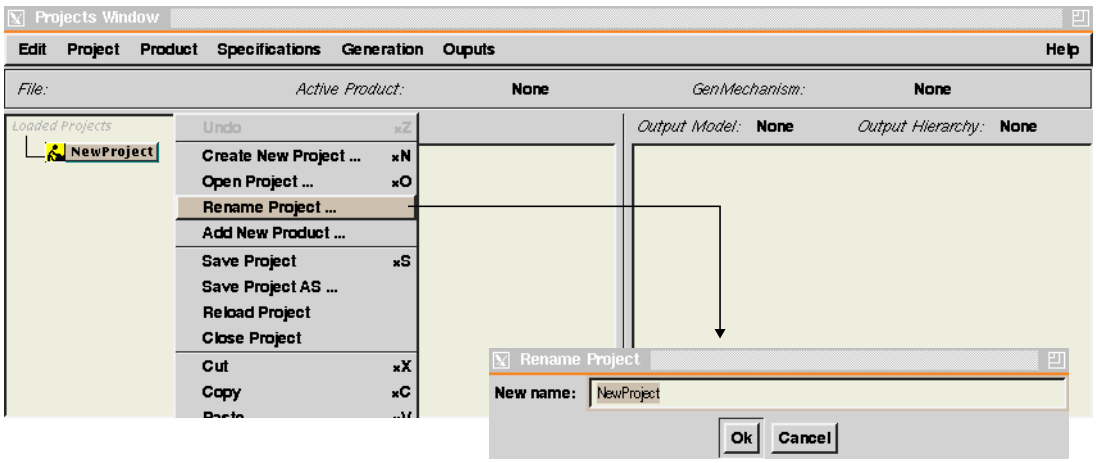
Interaction Diagram 72. Renaming a Project

Flow of Events:

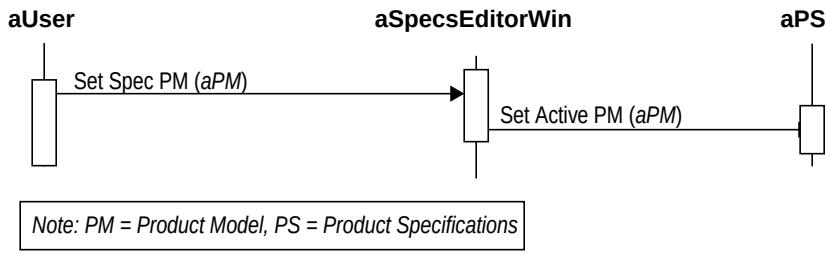
1. The user selects the “Rename Project” command from the Projects window.
2. The system displays a Set Name dialog asking the user for the new name.
3. The user types in the name and confirms.
4. The systems sends a “Rename Project” command to the Project object, and updates the interface accordingly.

Preconditions: A Project is selected in the Projects window.

Interface Design:



A 1.3.5 Set Input Product Model



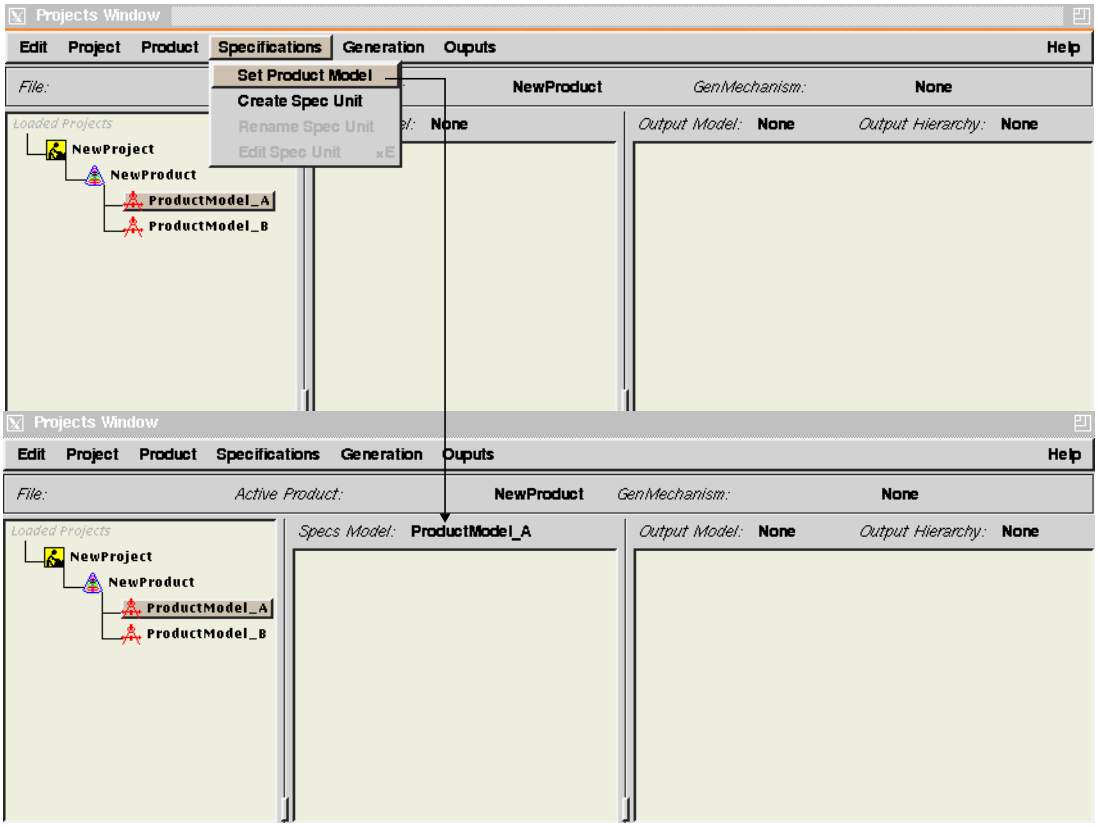
Interaction Diagram 73. Setting the Specification Product Model

Flow of Events:

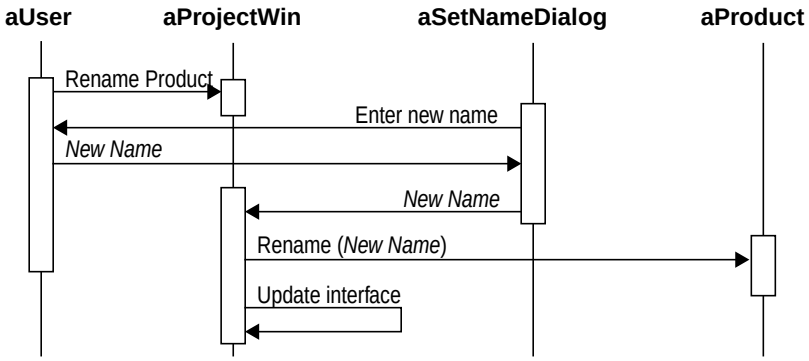
1. The user selects a Product Model for the Specification Model from the Specifications Editor window.
2. The system sets the corresponding active Product Model according to the user selection.

Preconditions: The Specifications Editor window is displayed.

Interface Design:



A 1.3.6 Rename Product



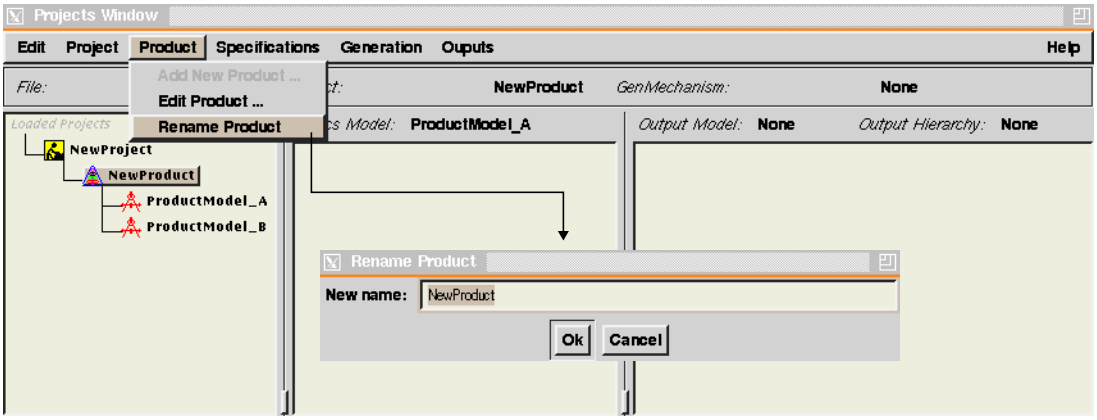
Interaction Diagram 74. Renaming a Product in a Project

Flow of Events:

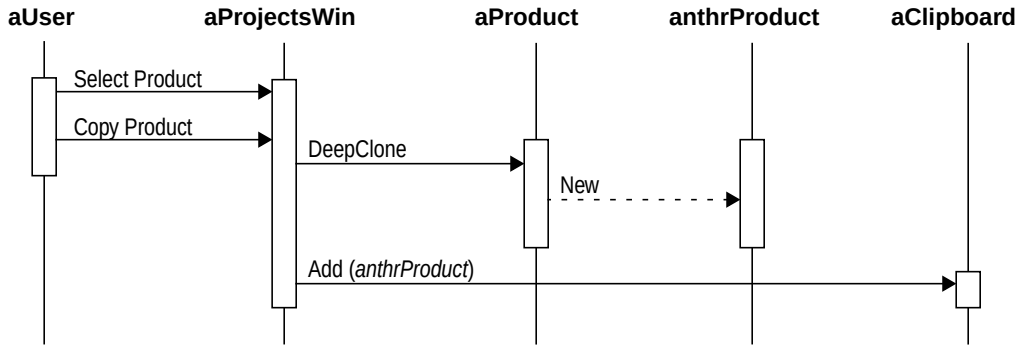
1. The user selects the “Rename Product” command from the Projects window.
2. The system displays a Set Name dialog asking the user for the new name.
3. The user types in the name and confirms.
4. The systems sends a “Rename Product” command to the Product object, and updates the interface accordingly.

Preconditions: A Product is selected in the Projects window.

Interface Design:



A 1.3.7 Copy Product



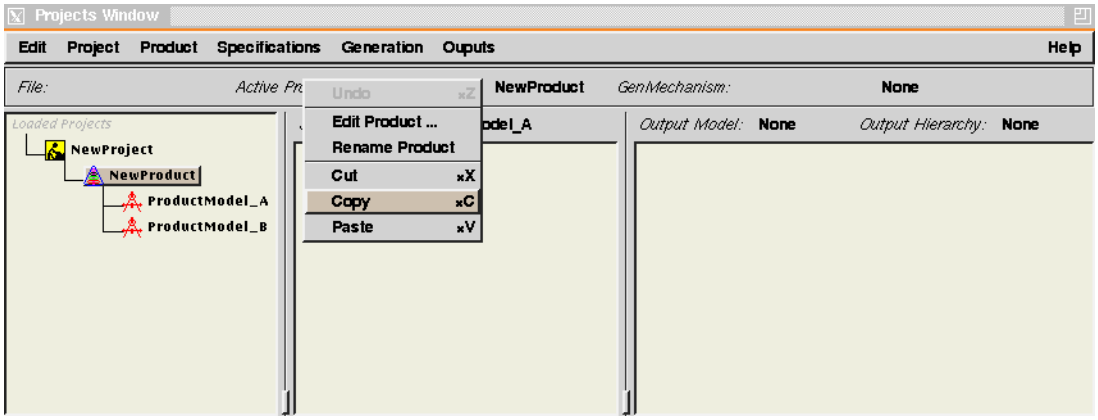
Interaction Diagram 75. Copying a Product

Flow of Events:

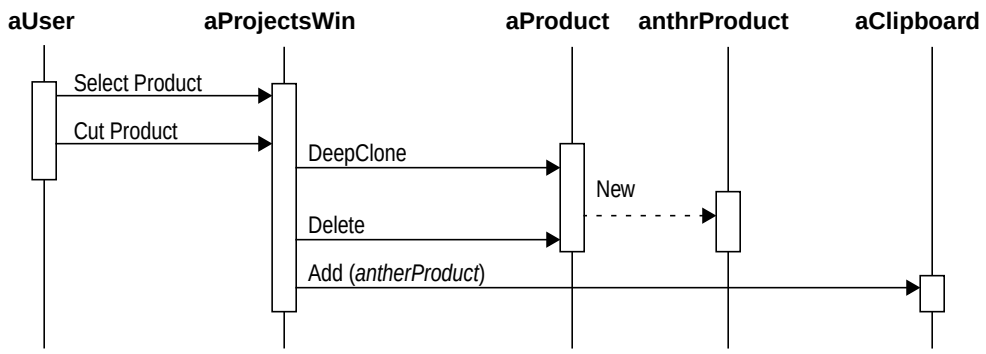
1. The user selects a Product from the Projects window.
2. The user selects the "Copy" command from the Projects window.
3. The system sends a "Deep Clone" command to the selected Project and saves it in the system clipboard.

Preconditions: A Product is selected in the Projects window.

Interface Design:



A 1.3.8 Cut Product



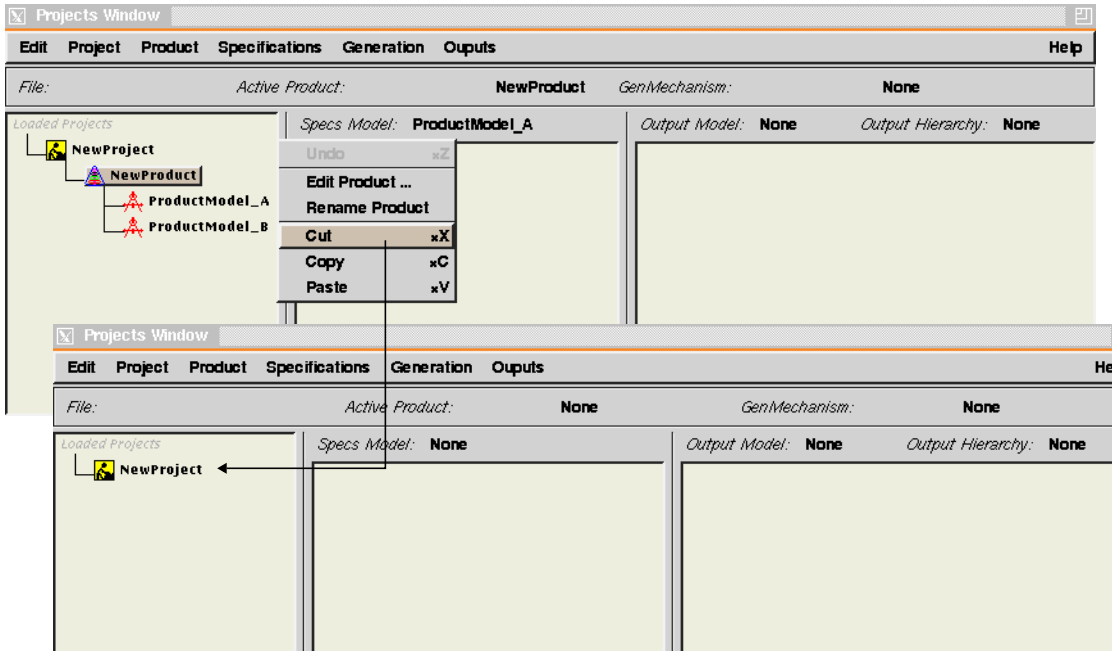
Interaction Diagram 76. Removing a Product from a Project

Flow of Events:

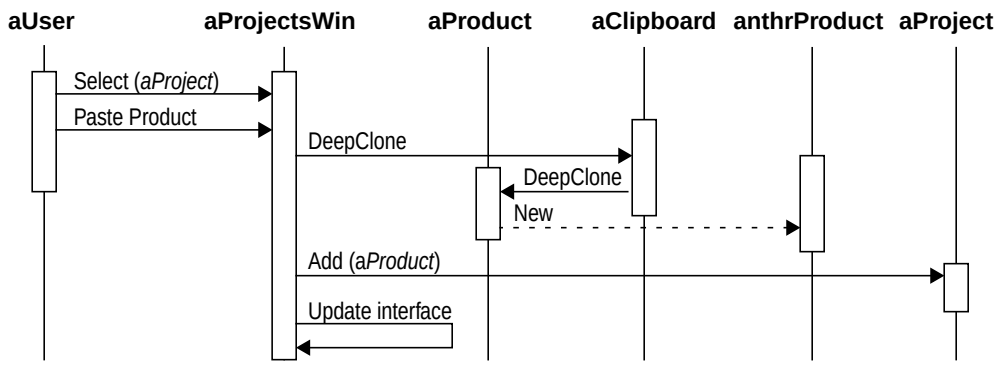
1. The user selects a Product from the Projects window.
2. The user selects the “Cut” command from the Projects window.
3. The system sends a “Deep Clone” command to the selected Project, saves the resulting clone in the system clipboard, and deletes the selected object.

Preconditions: A Product is selected in the Projects window.

Interface Design:



A 1.3.9 Paste Product



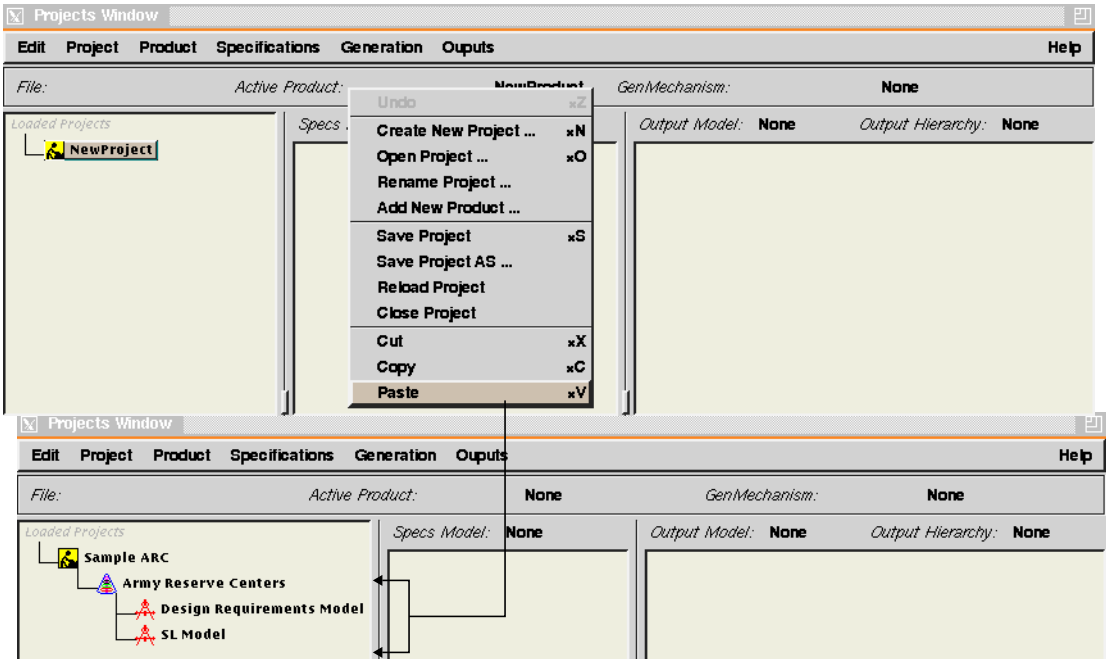
Interaction Diagram 77. Pasting a Product to a Project

Flow of Events:

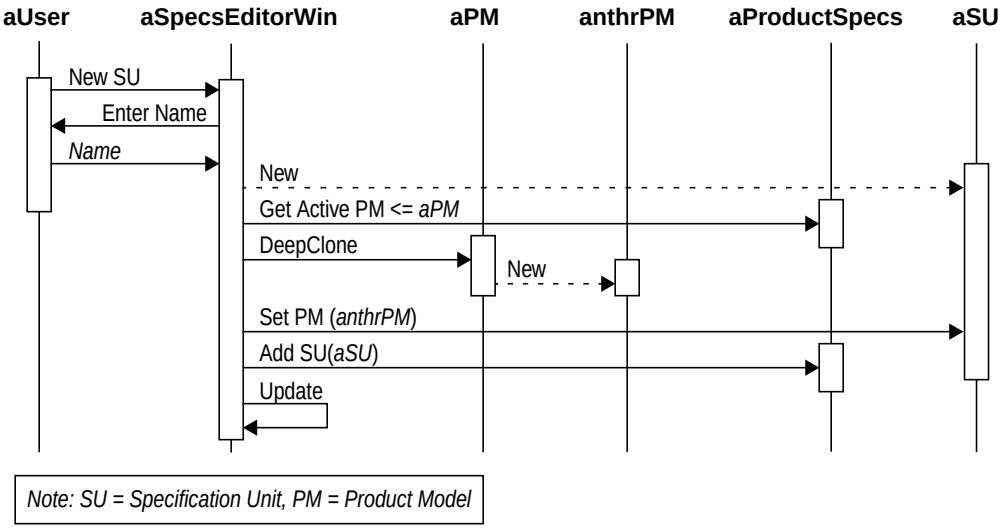
1. The user selects a Project in the Projects window.
2. The user selects the "Paste Product" command from the Projects window.
3. If the object in the system clipboard is a Product, the system sends it a "Deep Clone" command, adds the resulting clone to the selected Project and updates the interface.

Preconditions: A Project is selected in the Projects window and a Product exists in the system clipboard.

Interface Design:



A 1.3.10 Create Specification Unit



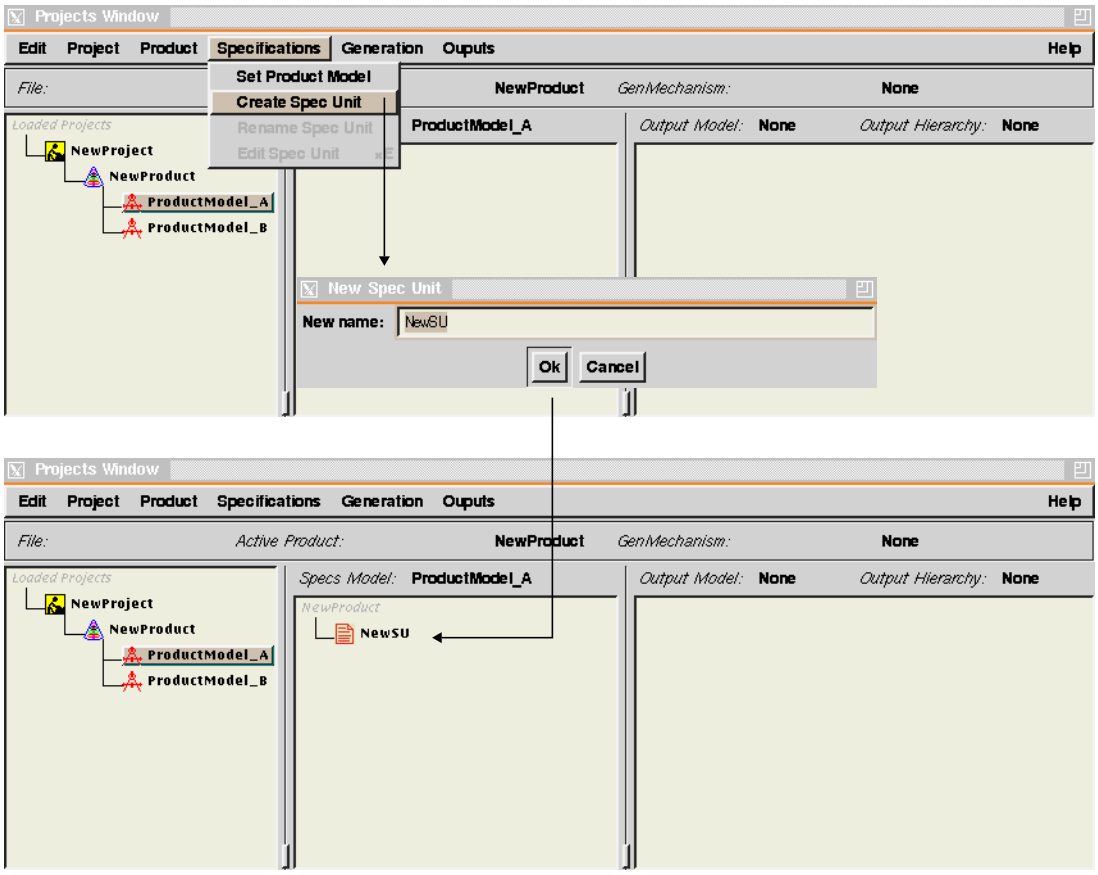
Interaction Diagram 78. Creating a Specification Unit

Flow of Events:

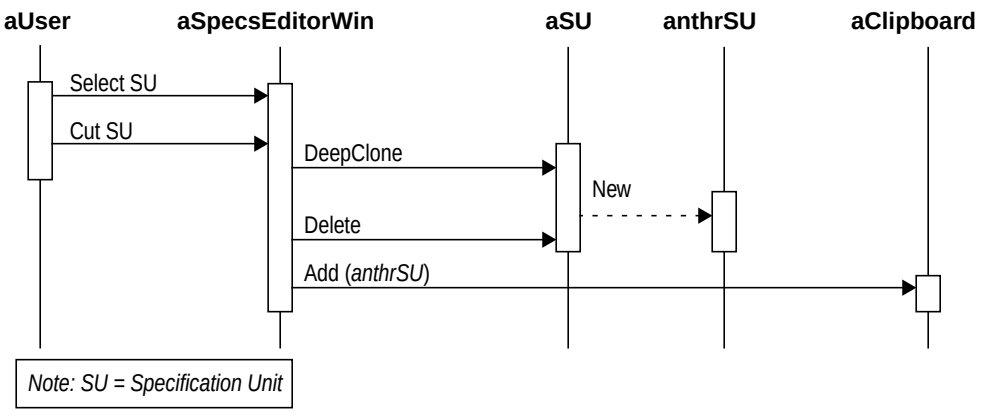
1. The user selects the "Create New Specification Unit" command from the Specifications Editor window.
2. The system prompts the user for a name. The user enters the name and confirms.
3. The system creates a new Specification Unit, clones the active Product Model and adds it to the new Specification Unit.
4. It then adds the new Specification Unit to the current Product Specifications and updates the interface accordingly.
5. If a Specification Unit was selected before the start of this use case, the new Specification Unit will be added as its constituent.

Preconditions: The Specifications Editor window is displayed.

Interface Design:



A 1.3.11 Cut Specification Unit



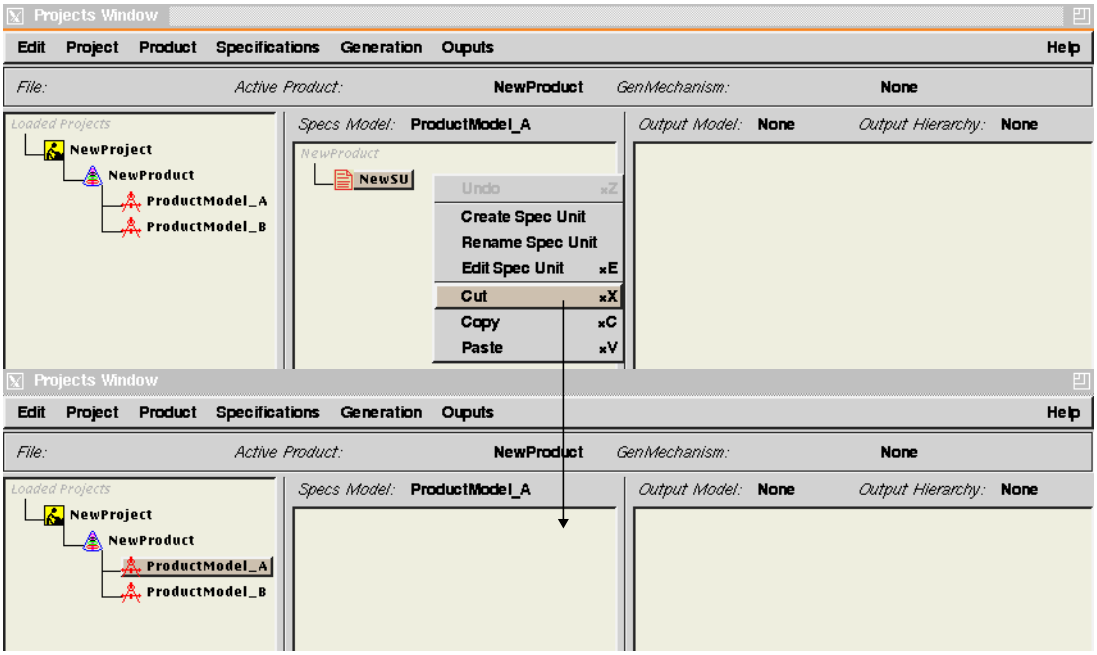
Interaction Diagram 79. Removing a Specification Unit

Flow of Events:

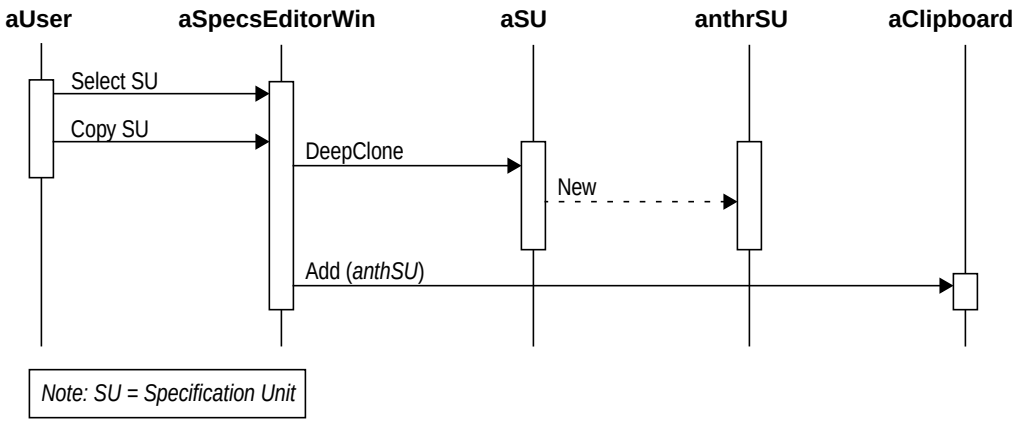
1. The user selects a Specification Unit from the Specifications Editor window.
2. The user selects the "Cut" command from the Specifications Editor window.
3. The system sends a "Deep Clone" command to the selected Specification Unit, saves the resulting clone in the system clipboard, and deletes the selected object.

Preconditions: A Specification Unit is selected in the Specifications Editor window.

Interface Design:



A 1.3.12 Copy Specification Unit



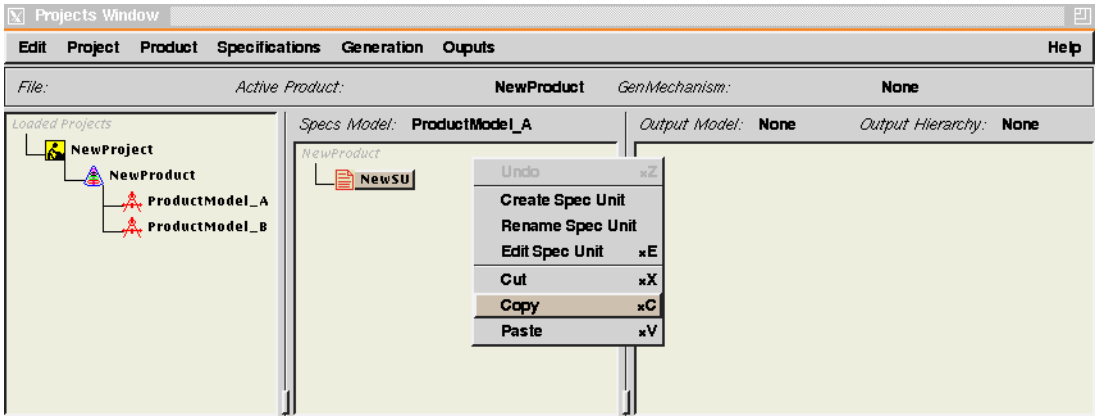
Interaction Diagram 80. Copying a Specification Unit

Flow of Events:

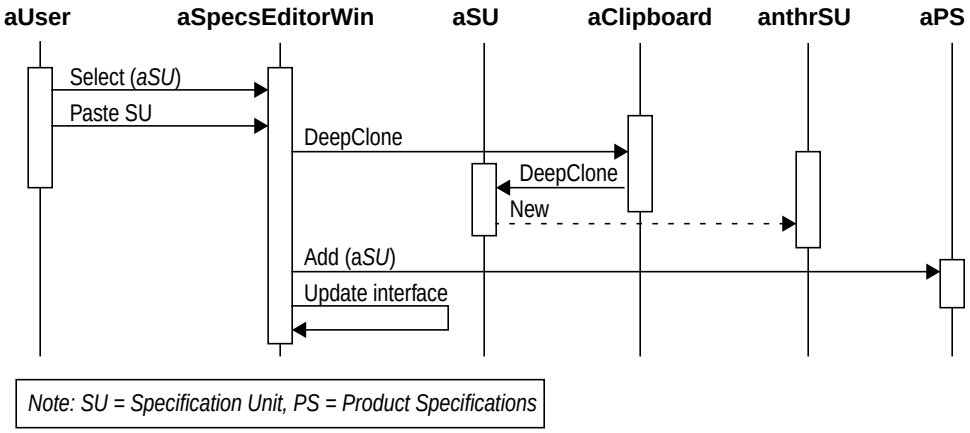
1. The user selects a Specification Unit from the Specifications Editor window.
2. The user selects the "Copy" command from the Specifications Editor window.
3. The system sends a "Deep Clone" command to the selected Specification Unit and saves it in the system clipboard.

Preconditions: A Specification Unit is selected in the Specifications Editor window.

Interface Design:



A 1.3.13 Paste Specification Unit



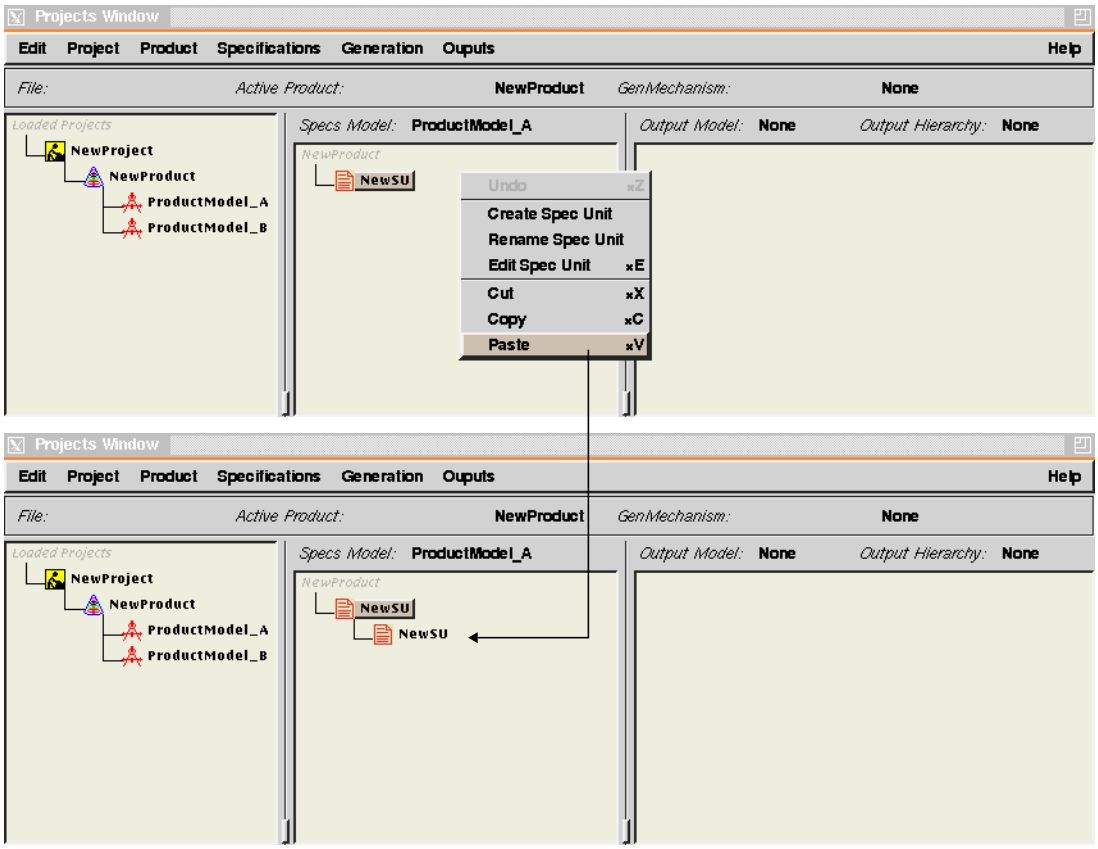
Interaction Diagram 81. Pasting a Specification Unit

Flow of Events:

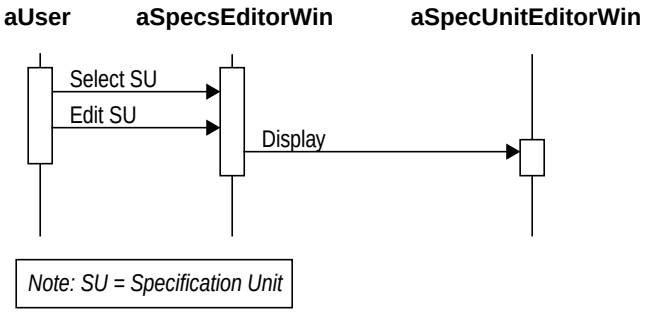
1. The user selects the "Paste Specification Unit" command from the Specifications Editor window.
2. If the object in the system clipboard is a Specification Unit, the system sends it a "Deep Clone" command, adds the resulting clone to the current Product Specifications and updates the interface.
3. If a Specification Unit was selected before the start of this use case, the pasted Specification Unit will be added as its constituent.

Preconditions: The Specifications Editor window is displayed and a Specification Unit exists in the system clipboard.

Interface Design:



A 1.3.14 Edit Specification Unit



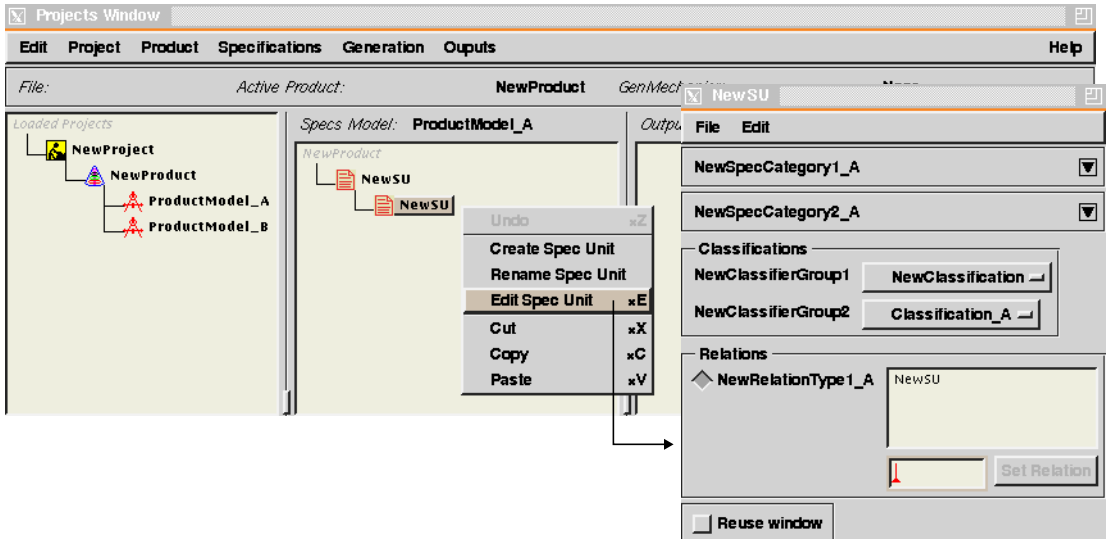
Interaction Diagram 82. Editing Specification Unit

Flow of Events:

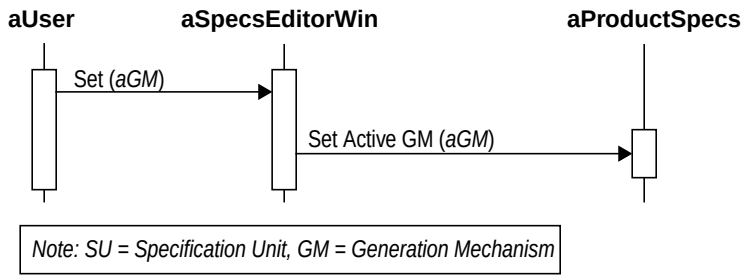
1. The user selects a Specification Unit from the Specifications Editor window then selects the “Edit Specification Unit” command.
2. The system displays the Specification Unit Editor window for the selected Specification Unit.
3. In the Specification Unit Editor window, the user can set attribute values for the selected Specification Unit, set its classifications and relations to other Specification Units and set the generation parameters to override those of the generation mechanism.

Preconditions: A Specification Unit is selected in the Specifications Editor window.

Interface Design:



A 1.3.15 Select Generation Mechanism



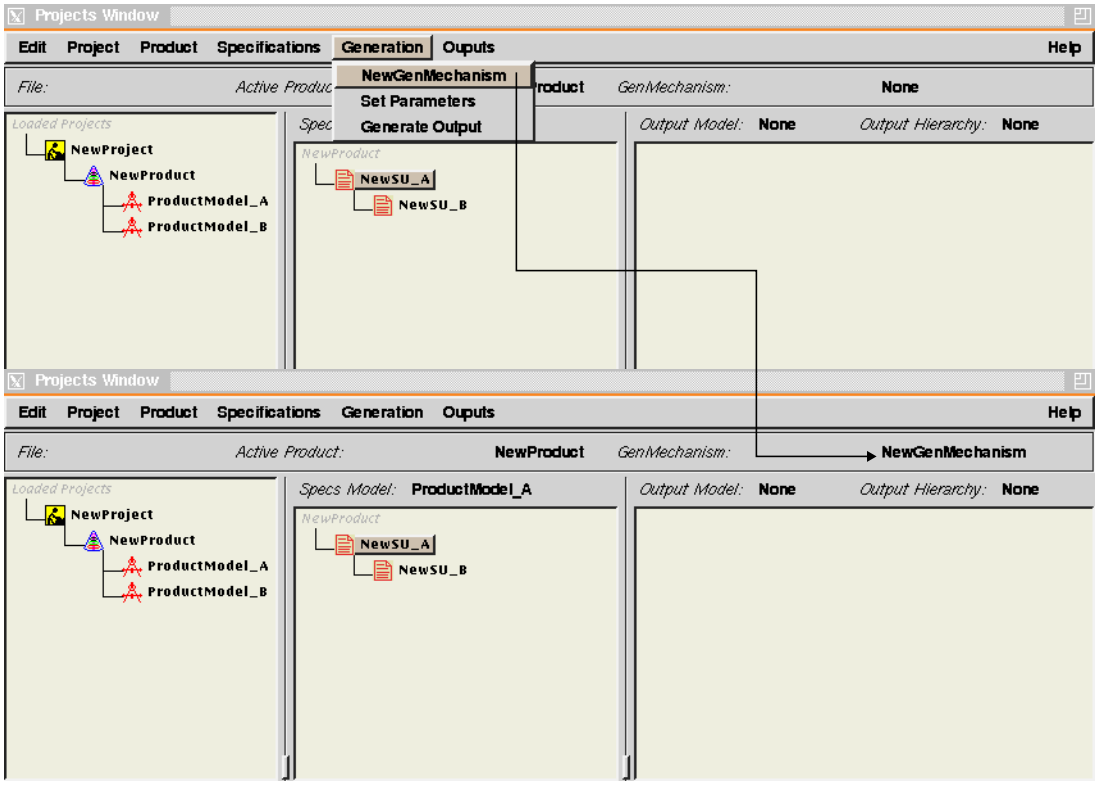
Interaction Diagram 83. Updating Specification Unit Settings from Product Model Defaults

Flow of Events:

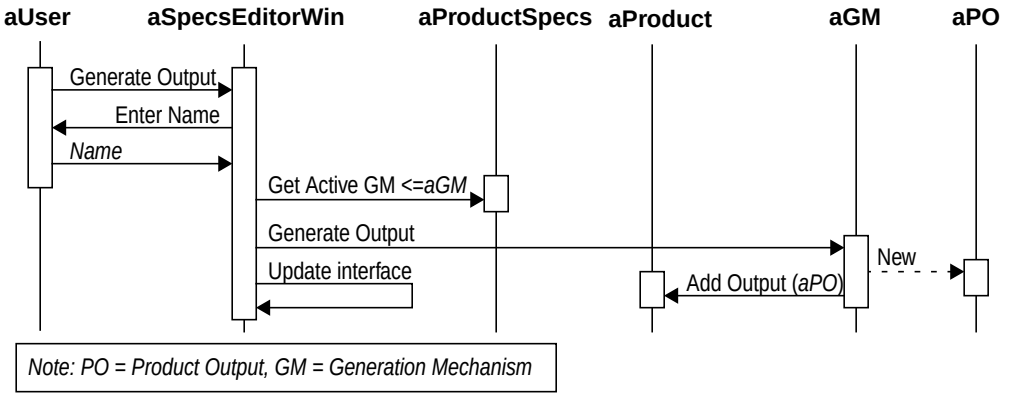
1. The user selects the “Set Generation Mechanism” command from the Specifications Editor window.
2. The system sets the active Generation Mechanism of the current Product Specifications object according to the user selection.

Preconditions: The Specifications Editor window is displayed.

Interface Design:



A 1.3.16 Generate Output



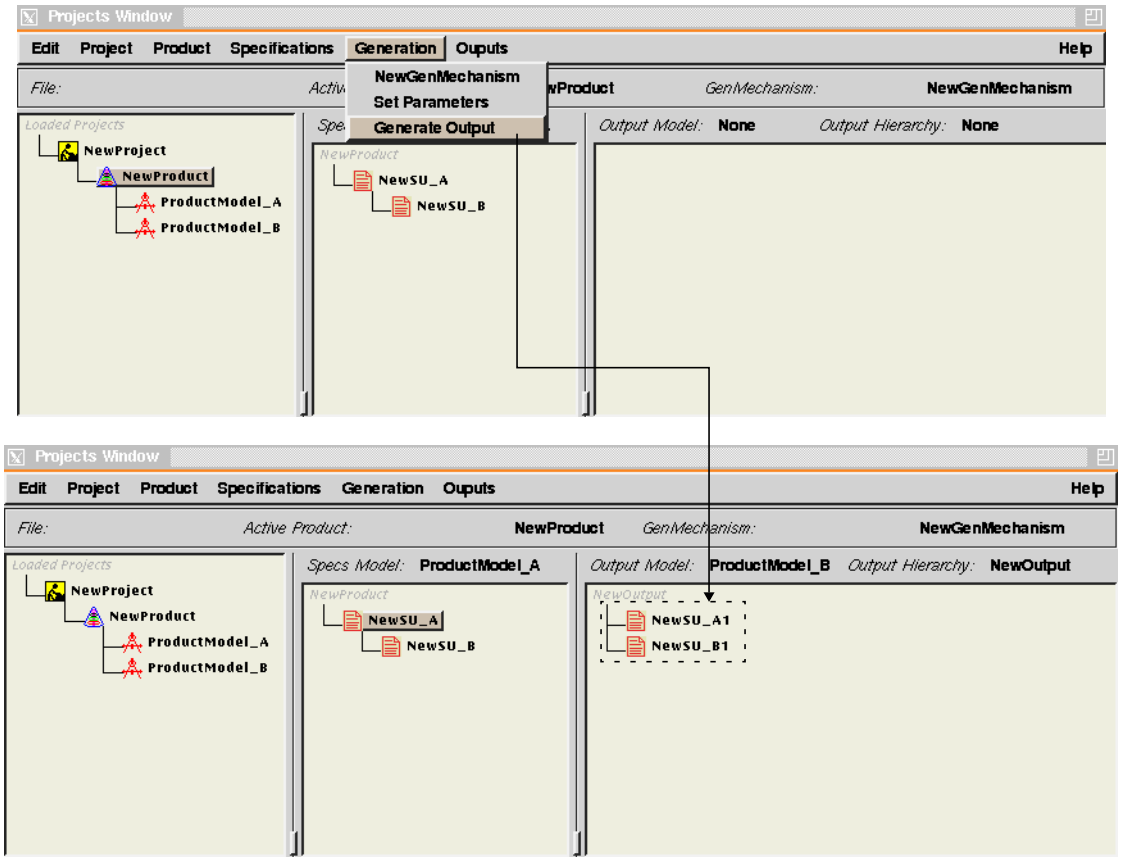
Interaction Diagram 84. Generating an Output from Product Specifications

Flow of Events:

1. The user selects the "Generate Output" command from the Specifications Editor window.
2. The system prompt the user for a name for new output. The user enters the name and confirms.
3. The system uses the active Generation Mechanism of the current Product Specifications object to generate an output from the specifications.
4. The generated output is then added to the current Product then displayed in the output panel of the Specifications Editor window.

Preconditions: Specifications Units exist in the Specifications Editor window, and a Generation Mechanism has been selected for the current Product Specifications.

Interface Design:



APPENDIX 2

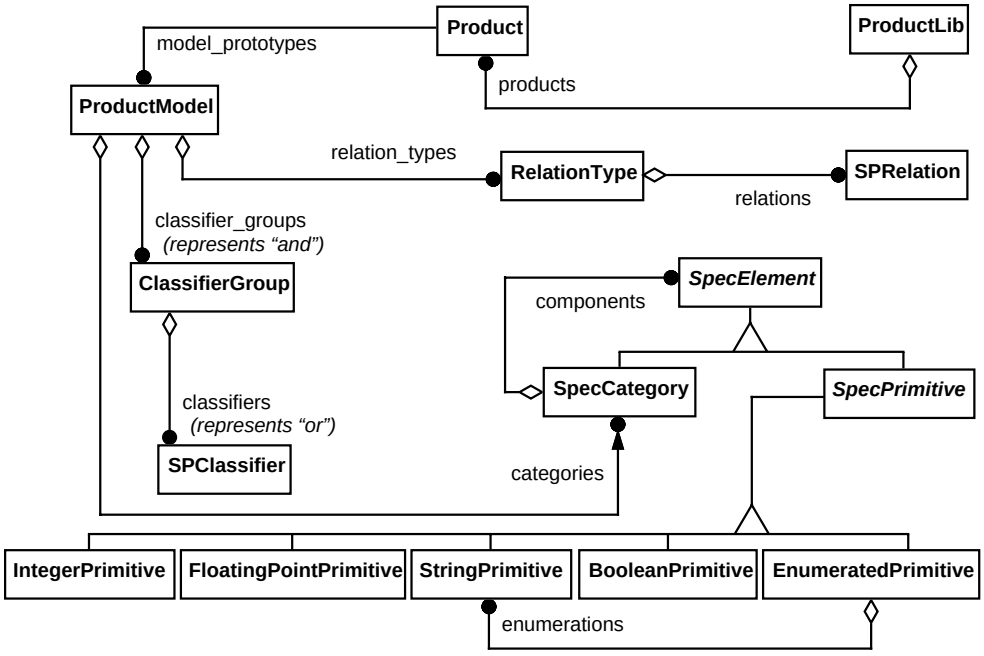
System Object Models

This appendix contains the system object model using the the OMT notation [Rumbaugh et al. 91].

A 2.1 Domain Object Models

A 2.1.1 The Product Modeling Module

Module Objects and Relations



Object Attributes and Methods

ProductLib
#name:char * #libFile:Data *
+ProductLib(nm:char *) +~ProductLib +GetName:char * +SetName(nm:char *):bool +AddProduct(pr:Product *):bool +RemoveProduct(pr:Product *):bool +GetProducts():SeqCollection * +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +GetFile():Data * +SetFile(fl:Data *)

Product
+name:char * #description:char *
+Product(nm:char *) +~Product +%Project:: +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetName(nm:char *):bool +GetName():char * +SetDescription(des:char *):bool +GetDescription():char * +SetProject(pr:Project *):bool +GetProject():Project * +AddModel(md:ProductModel *):bool +RemoveModel(md:ProductModel *):bool +GetModels():SeqCollection * +SetSpecs(sp:ProductConstruct *):bool +GetSpecs():ProductConstruct * +AddOutput(po:ProductConstruct *):bool +RemoveOutput(po:ProductConstruct *):bool +GetOutputs():SeqCollection * +AddGM(gm:GenMechanism *):bool +RemoveGM(gm:GenMechanism *):bool +GetGMs():SeqCollection *

ProductModel
+name:char * #description:char *
+ProductModel(nm:char *,pr:Product *) +~ProductModel +%Product:: +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetName(nm:char *):bool +GetName():char * +AddClGroup(cg:ClassifierGroup *):bool +RemoveClGroup(cg:ClassifierGroup *):bool +GetClGroups():SeqCollection * +AddRelType(rt:RelationType *):bool +RemoveRelType(rt:RelationType *):bool +GetRelTypes():SeqCollection * +AddSpecCategory(sc:SpecCategory *):bool +RemoveSpecCategory(sc:SpecCategory *):bool +GetSpecCategories():SeqCollection * +SetProduct(pr:Product *):bool +GetProduct():Product *

Object Attributes and Methods (contd.)

ClassifierGroup
+name:char * #description:char * +activeClassifier:SPClassifier *
+ClassifierGroup(nm:char *) +~ClassifierGroup +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +AddClassifier(cl:SPClassifier *):bool +RemoveClassifier(cl:SPClassifier *):bool +GetClassifiers():SeqCollection * +SetName(nm:char *):bool +GetName():char * +SetDescription(des:char *):bool +GetDescription():char *

SPClassifier
+name:char * #description:char * +containerConstraints:SeqCollection * +constituentsConstraints:SeqCollection * #suffix:char *
+SPClassifier(nm:char *) +~SPClassifier +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetName(nm:char *):bool +SetSuffix(nm:char *) +GetSuffix():char * +GetName():char * +SetDescription(des:char *):bool +GetDescription():char * +AddContConstraint(co:SPClassifier *):bool +RemoveContConstraint(co:SPNameTag*):bool +GetContConstraints():SeqCollection * +AddConstConstraint(co:SPClassifier *):bool +RemoveConstConstraint(co:SPNameTag *):bool +GetConstConstraints():SeqCollection * +CanClassify(su:SpecUnit *):bool +FindClassifier(col:SeqCollection *,co:SPClassifier *):SPNameTag * +IsContainerConstraint(co:SPClassifier *):bool +IsConstituentConstraint(co:SPClassifier *):bool

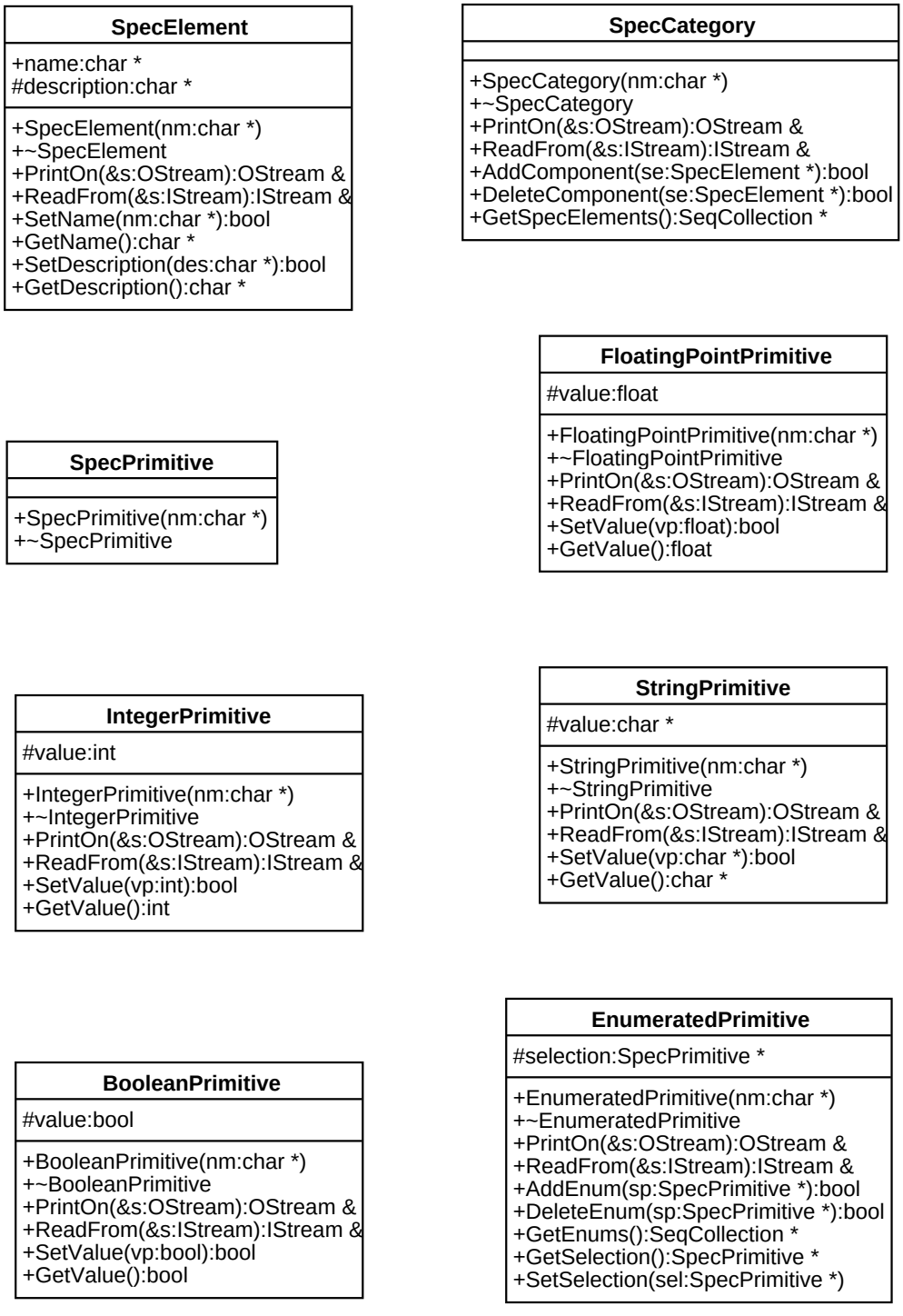
SPNameTag
+name:char *
+SPNameTag(nm:char *) +~SPNameTag +SetName(nm:char *) +GetName():char * +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream &

Object Attributes and Methods (contd.)

RelationType
+name:char * #description:char *
+RelationType(nm:char *) +~RelationType +HasRelation(su1:SpecUnit *,su2:SpecUnit *):SPRelation * +ReadFrom(&s:IStream):IStream & +PrintOn(&s:OStream):OStream & +SetName(nm:char *):bool +GetName():char * +SetDescription(des:char *):bool +GetDescription():char * +AddRelation(rel:SPRelation *):bool +RemoveRelation(rel:SPRelation *):bool +GetRelations():SeqCollection *

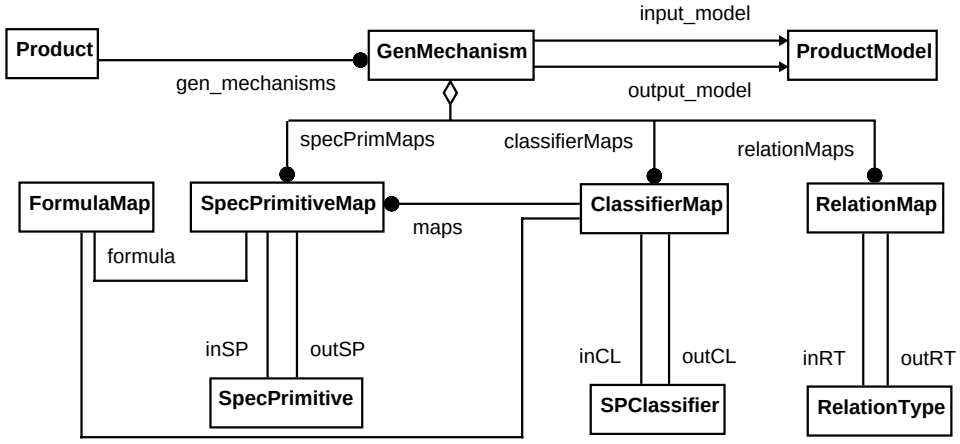
SPRelation
#relValue:float
+SPRelation(val:float,su1:SpecUnit *,su2:SpecUnit *) +~SPRelation +ReadFrom(&s:IStream):IStream & +PrintOn(&s:OStream):OStream & +SetValue(val:float):bool +GetValue():float +SetSU1(su:SpecUnit *):bool +GetSU1():SpecUnit * +SetSU2(su:SpecUnit *):bool +GetSU2():SpecUnit * +SetType(rt:RelationType *):bool +GetType():RelationType *

Object Attributes and Methods (contd.)



A 2.1.2 The Generation Mechanism Module

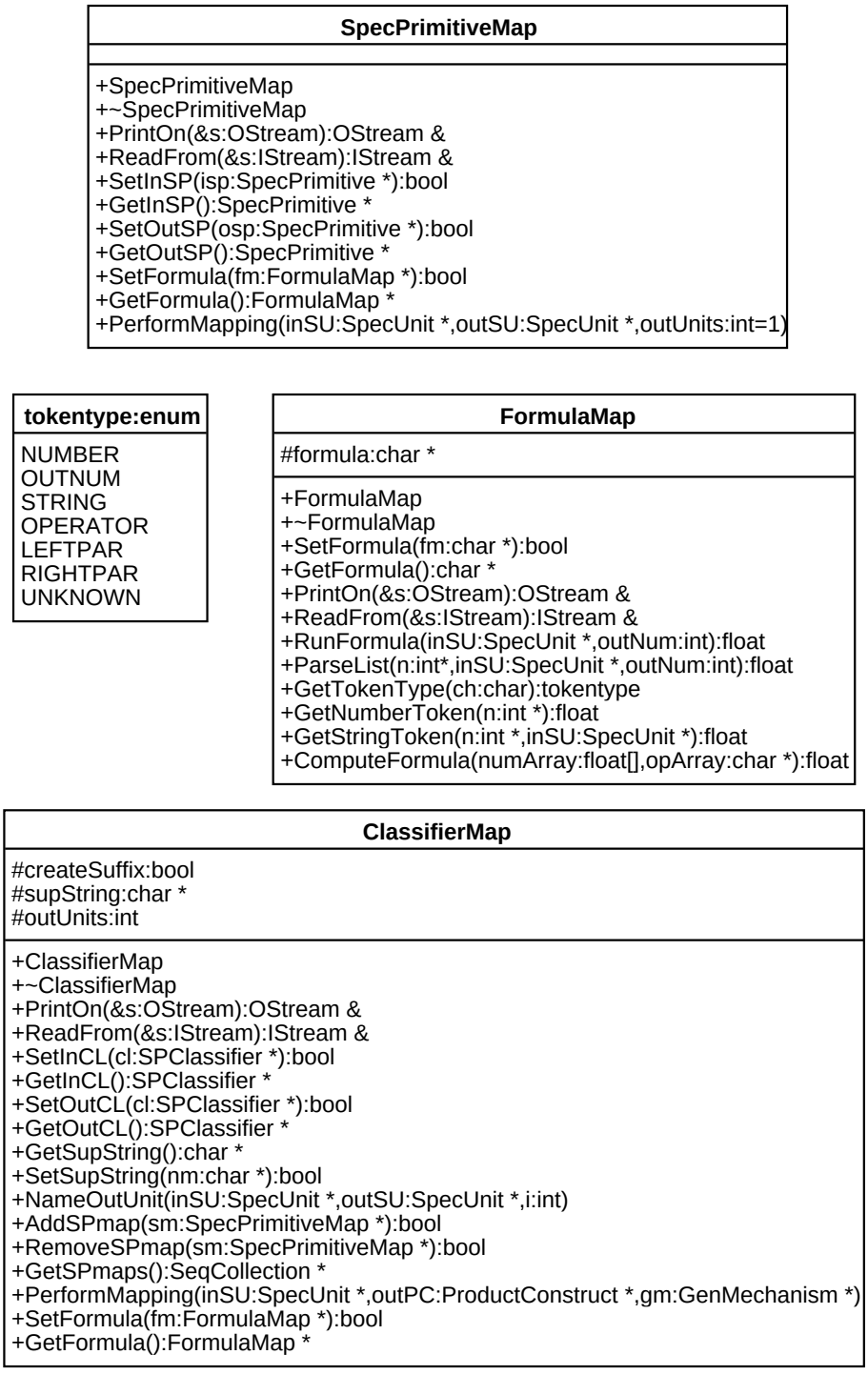
Module objects and Relations



Object Attributes and Methods

GenMechanism
#name:char *
+GenMechanism(nm:char *) +~GenMechanism -%Product:: +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetName(nm:char *):bool +GetName():char * +SetSpecsModel(pm:ProductModel *):bool +GetSpecsModel():ProductModel * +SetOutputModel(om:ProductModel *):bool +GetOutputModel():ProductModel * +SetGenParameters(gp:GenParameters *):bool +GetGenParameters():GenParameters * +AddPrimMap(pm:SpecPrimitiveMap *):bool +RemovePrimMap(pm:SpecPrimitiveMap *):bool +GetPrimMaps():SeqCollection * +AddFormulaMap(fm:FormulaMap *):bool +RemoveFormulaMap(fm:FormulaMap *):bool +GetFormulaMaps():SeqCollection * +AddClassifierMap(cm:ClassifierMap *):bool +RemoveClassifierMap(cm:ClassifierMap *):bool +GetClassifierMaps():SeqCollection * +AddRelMap(rm:RelationMap *):bool +RemoveRelMap(rm:RelationMap *):bool +GetRelMaps():SeqCollection * +GetClassMap(su:SpecUnit *):ClassifierMap * +GenerateOutput(inSU:SpecUnit *,outPC:ProductConstruct *,mapRelations:bool) +MapRelations(inSU:SpecUnit *,outPC:ProductConstruct *) +GetRelMapFor(rel:SPRelation *):RelationMap *

Object Attributes and Methods (contd.)

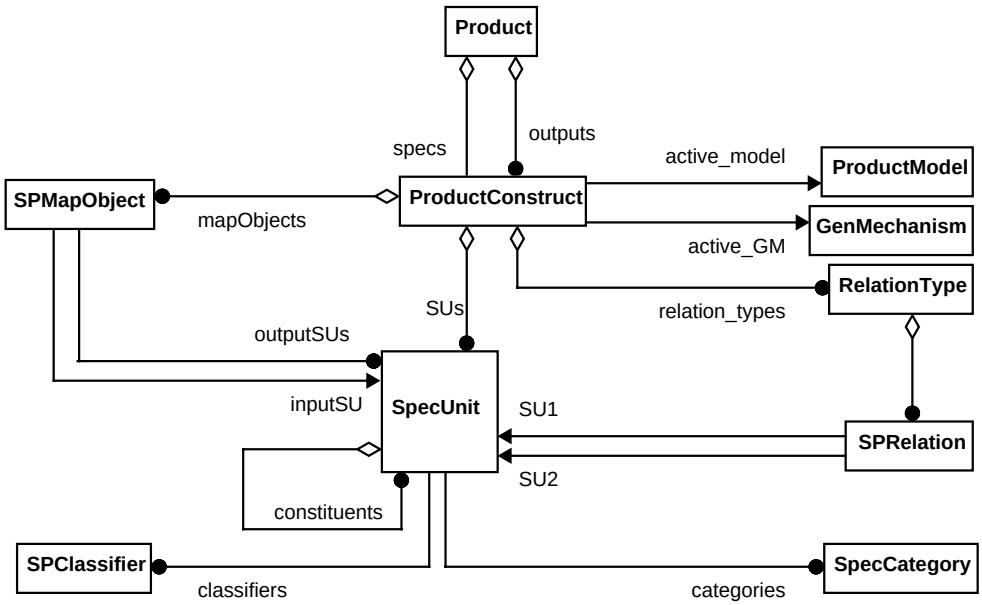


Object Attributes and Methods (contd.)

RelationMap
+RelationMap +~RelationMap +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetInRel(irel:RelationType *):bool +GetInRel():RelationType * +SetOutRel(orel:RelationType *):bool +GetOutRel():RelationType * +PerformMapping(rel:SPRelation*,outPC:ProductConstruct *)

A 2.1.3 The Design Requirements Module

Module objects and Relations



Object Attributes and Methods

Project
#name:char * #description:char * #projFile:Data * #changed:bool=FALSE
+Project(nm:char *) +~Project +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +GetName():char * +SetName(nm:char *):bool +SetProjFile(fd:Data *):bool +GetProjFile:Data * +AddProduct(pr:Product *):bool +RemoveProduct(pr:Product *):bool +GetProducts():SeqCollection *

SPMapObject
+SPMapObject(su:SpecUnit *) +~SPMapObject +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetInSU(su:SpecUnit *):bool +GetInSU():SpecUnit * +AddOutSU(su:SpecUnit *):bool +RemoveOutSU(su:SpecUnit *):Object * +FindOutSU(su:SpecUnit *):SpecUnit * +GetOutSUs():SeqCollection *

Object Attributes and Methods (contd.)

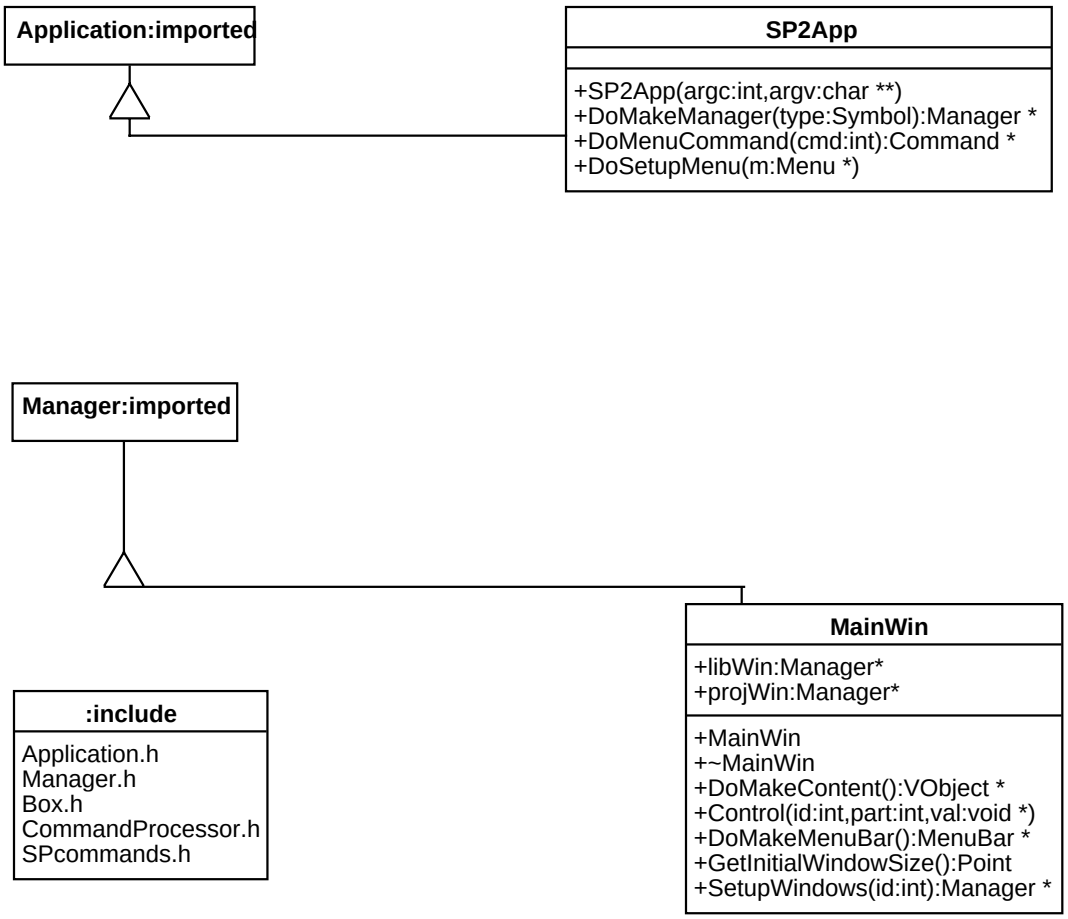
ProductConstruct
+ProductConstruct(pr:Product *) +~ProductConstruct +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetName(nm:char *):bool +GetName():char * +SetActiveModel(pd:ProductModel *):bool +GetActiveModel():ProductModel * +AddRelType(rt:RelationType *):bool +RemoveRelType(rt:RelationType *):bool +GetRelTypes():SeqCollection * +GetRelation(rt:RelationType *,su1:SpecUnit *,su2:SpecUnit *):SPRelation * +RemoveRelationsFor(su:SpecUnit *) +GetRelationsFor(su:SpecUnit *):SeqCollection * +AddSU(su:SpecUnit *):bool +RemoveSU(su:SpecUnit *):bool +GetSUs():SeqCollection * +SetProduct(pr:Product *):bool +GetProduct():Product * +GetActiveGM():GenMechanism * +SetActiveGM(gm:GenMechanism *):bool +CreateSpecUnit(nm:char *):SpecUnit * +AddMapObject(mo:SPMapObject *):bool +RemoveMapObject(mo:SPMapObject *):bool +GetMapObjects():SeqCollection * +CreateMapping(inSU:SpecUnit *,newSU:SpecUnit *) +RemoveSpecsMap(su:SpecUnit *):bool +RemoveOutMap(su:SpecUnit *):bool +GetMapObjectFor(su:SpecUnit *):SPMapObject *

Object Attributes and Methods (contd.)

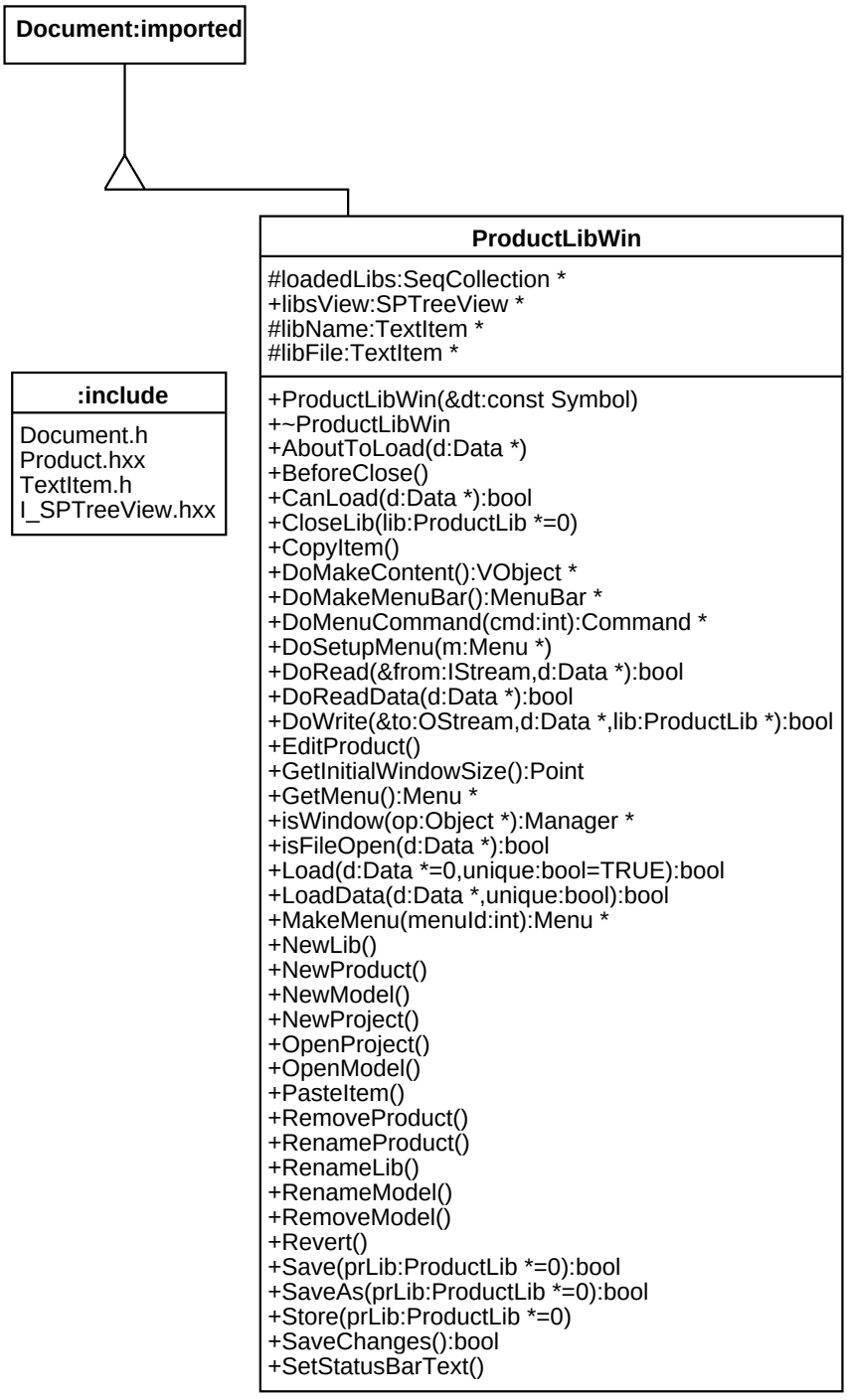
SpecUnit
+name:char * #description:char *
+SpecUnit(nm:char *) +~SpecUnit +%ProductConstruct:: +CheckClassification(su:SpecUnit *):bool +PrintOn(&s:OStream):OStream & +ReadFrom(&s:IStream):IStream & +SetName(nm:char *):bool +GetName():char * +SetDescription(des:char *):bool +GetDescription():char * +IsParent(su:SpecUnit *):bool +AddConst(su:SpecUnit *):bool +RemoveConst(su:SpecUnit *):bool +GetConstList():SeqCollection * +SetContainer(su:SpecUnit *):bool +GetContainer():SpecUnit * +SetProductConst(pc:ProductConstruct *):bool +GetProductConst():ProductConstruct * +SetCategories(sc:SeqCollection *):bool +GetCategories():SeqCollection * +SetClassification(col:SeqCollection *,cl:SPClassifier *):bool +GetClassifications():SeqCollection * +FindClassifier(cl:SPClassifier *):SPClassifier * +FindPrimitive(nm:char *):SpecPrimitive * +FlattenCategories(col:SeqCollection *):SeqCollection * +HasRelation(rt:RelationType *,su:SpecUnit *=0):SPRelation *

A 2.2 Interface Object Models

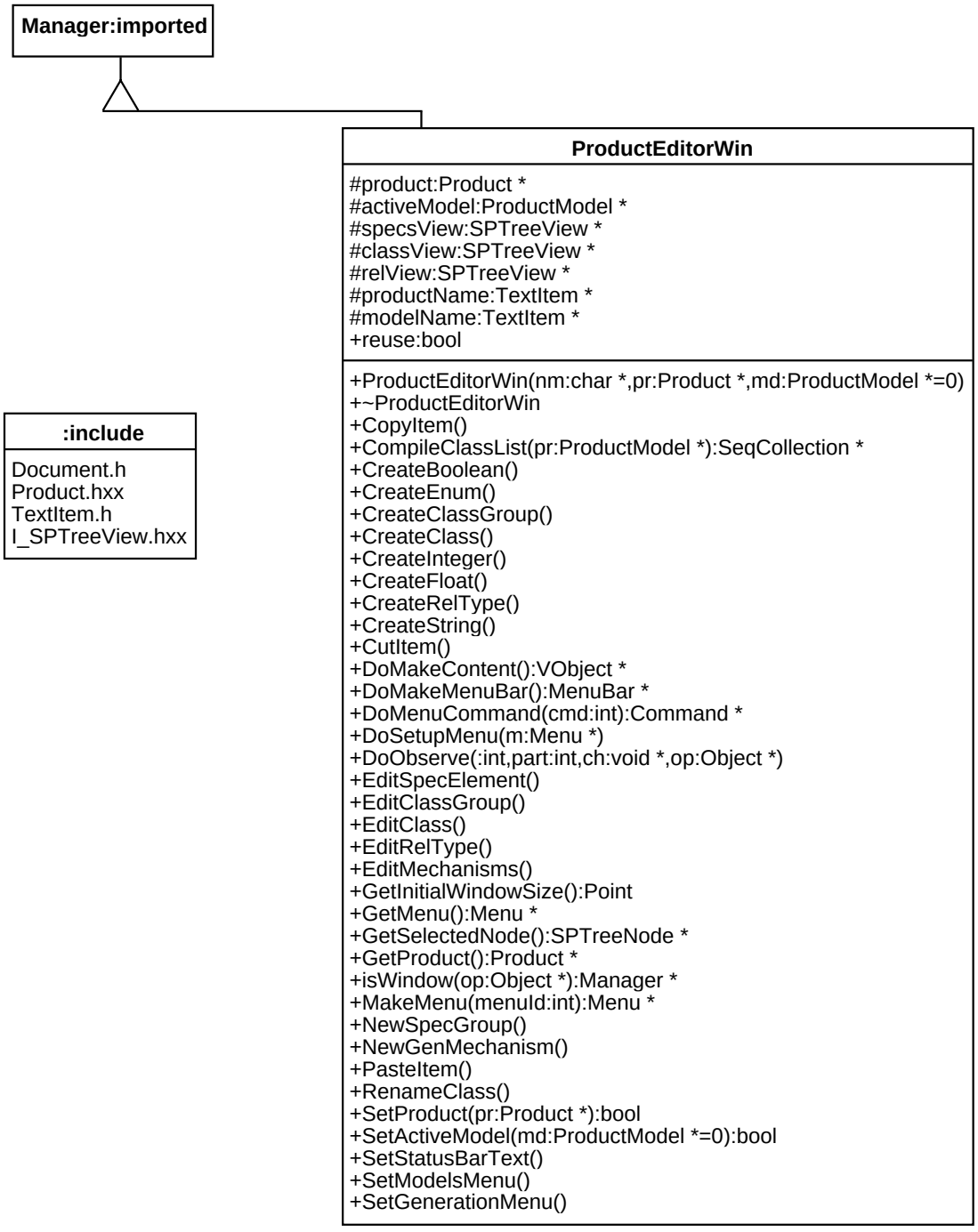
A 2.2.1 The Main Window



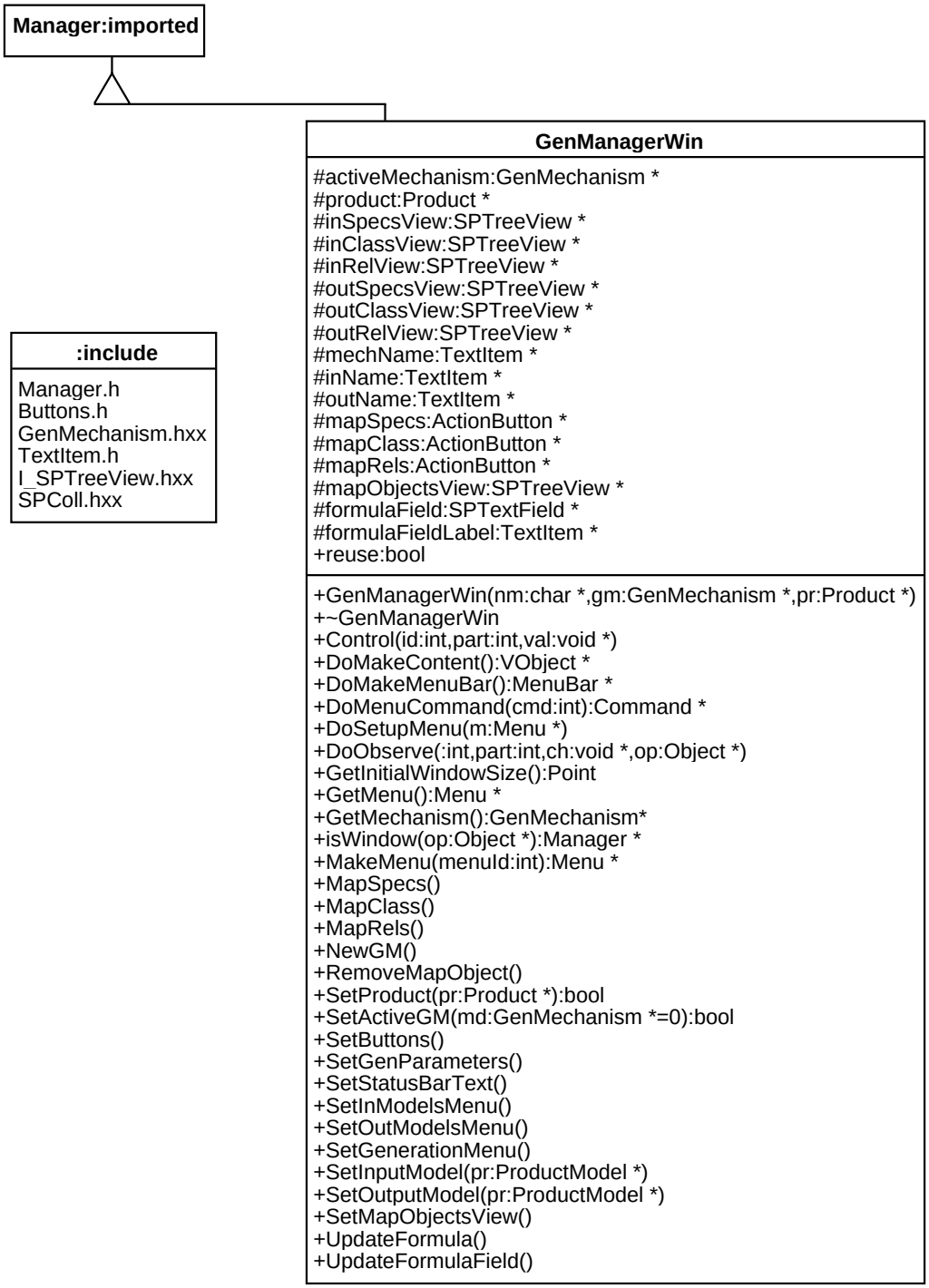
A 2.2.2 The Product Libraries Window



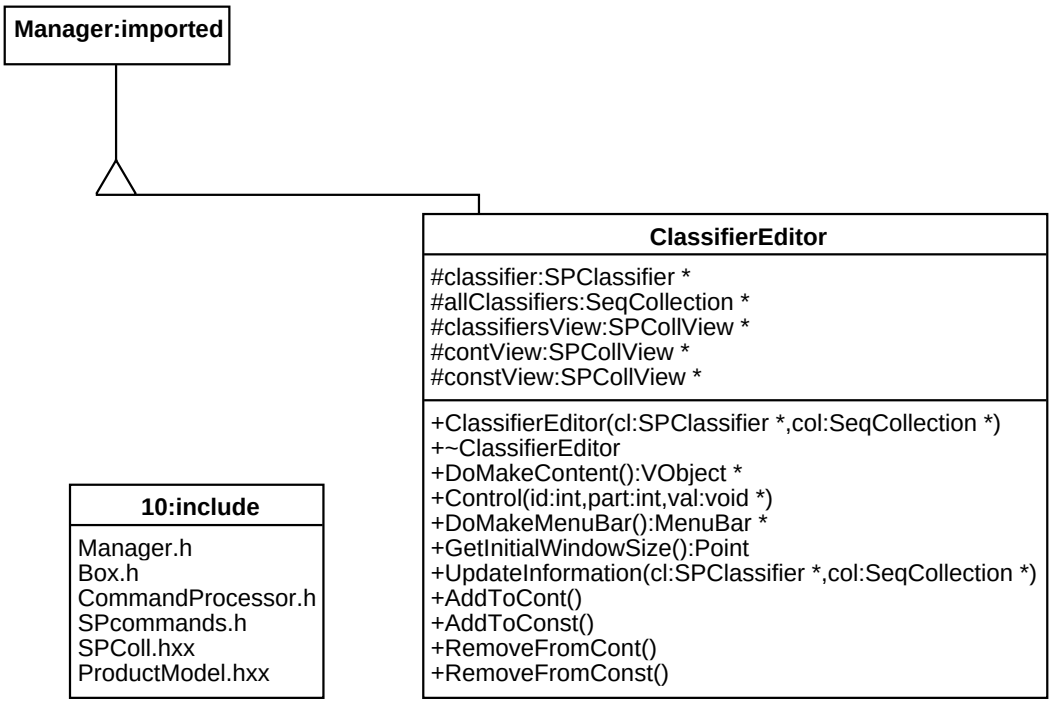
A 2.2.3 The Product Editor Window



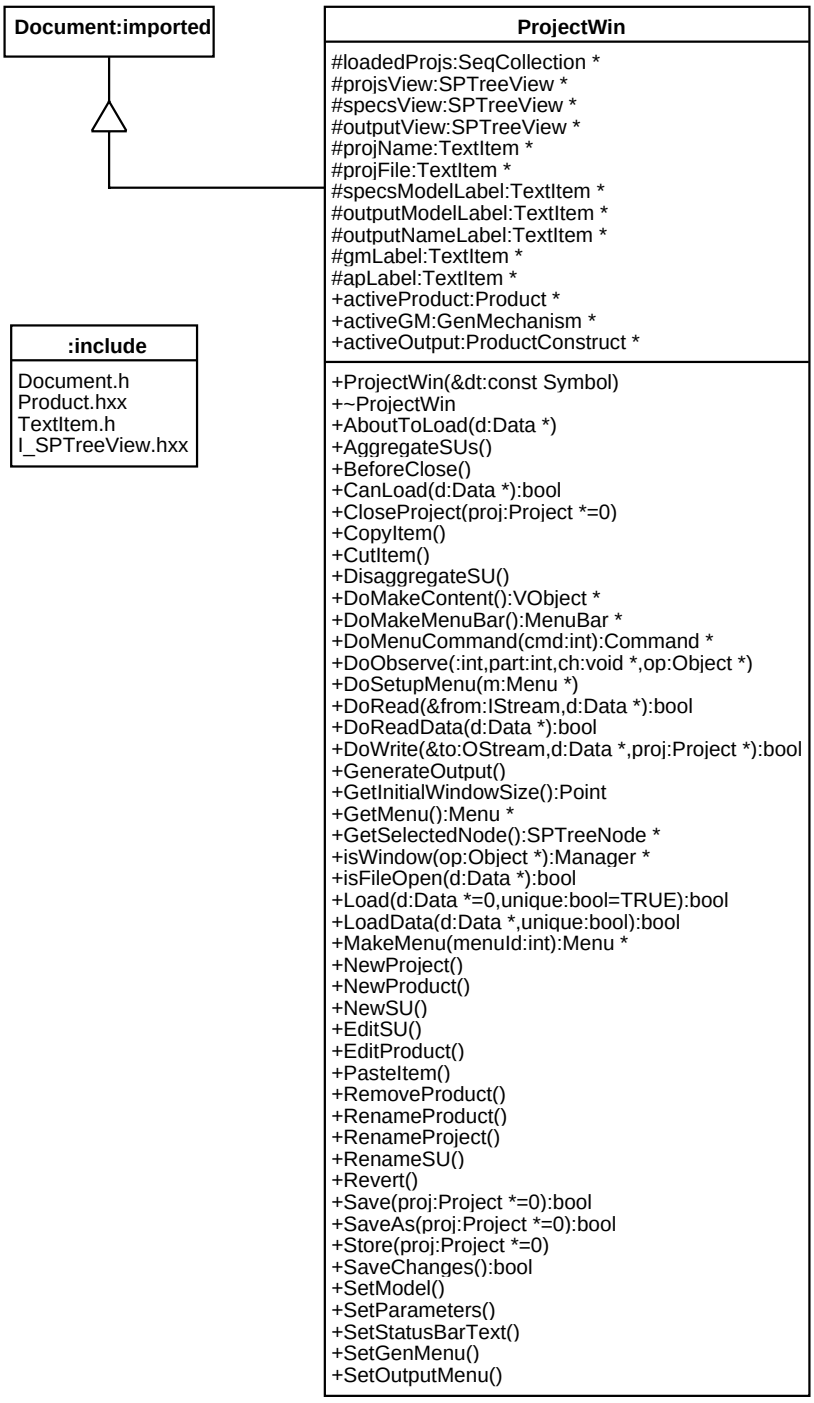
A 2.2.4 The Generation Manager Window



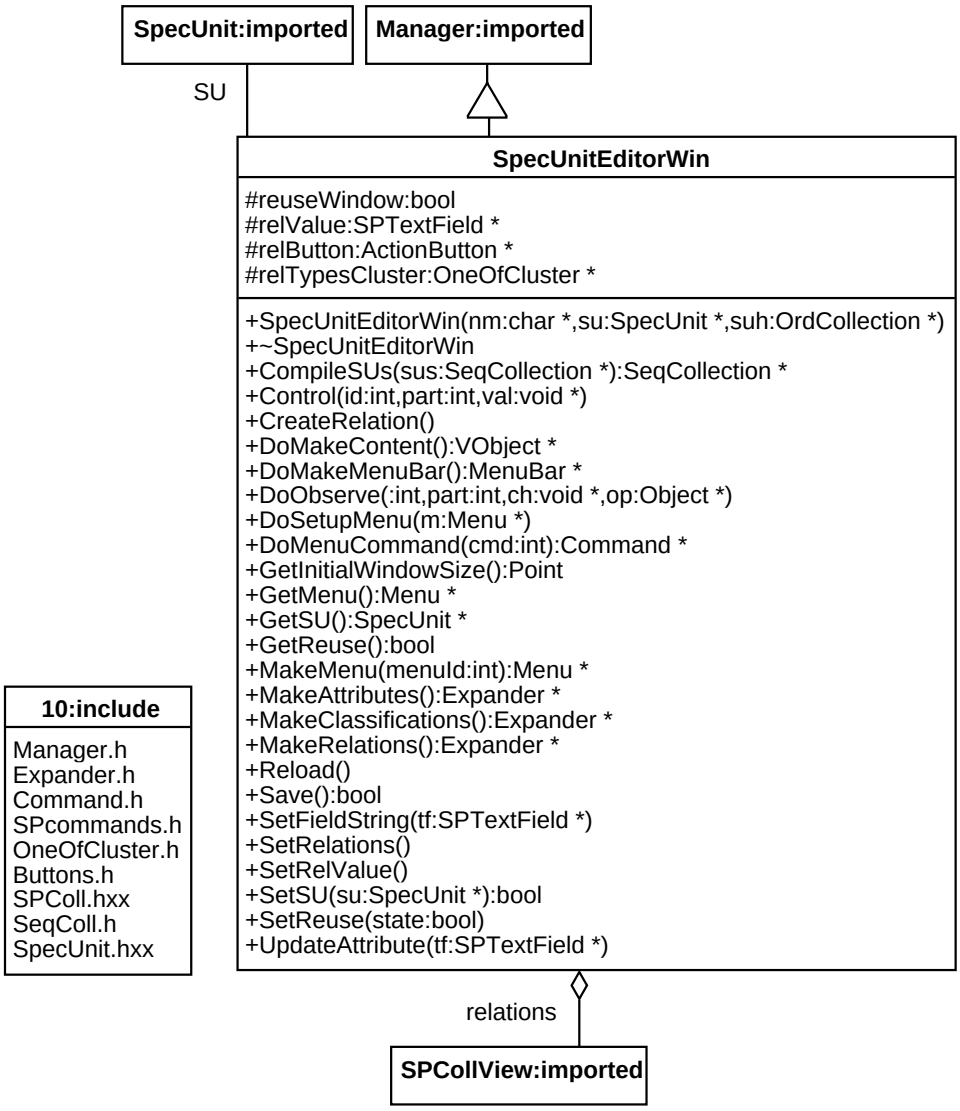
A 2.2.5 The Classifier Editor Window



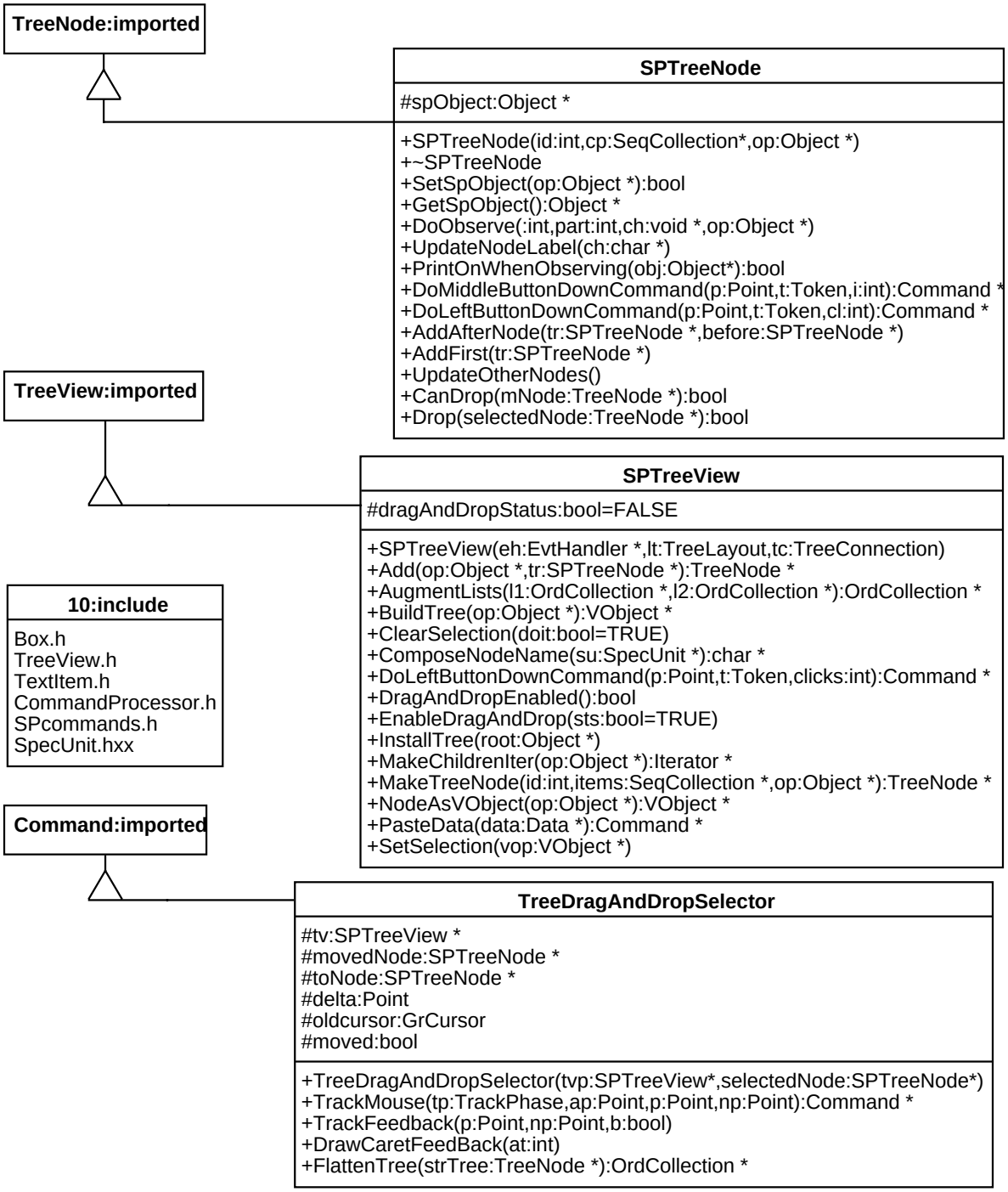
A 2.2.6 The Projects Window



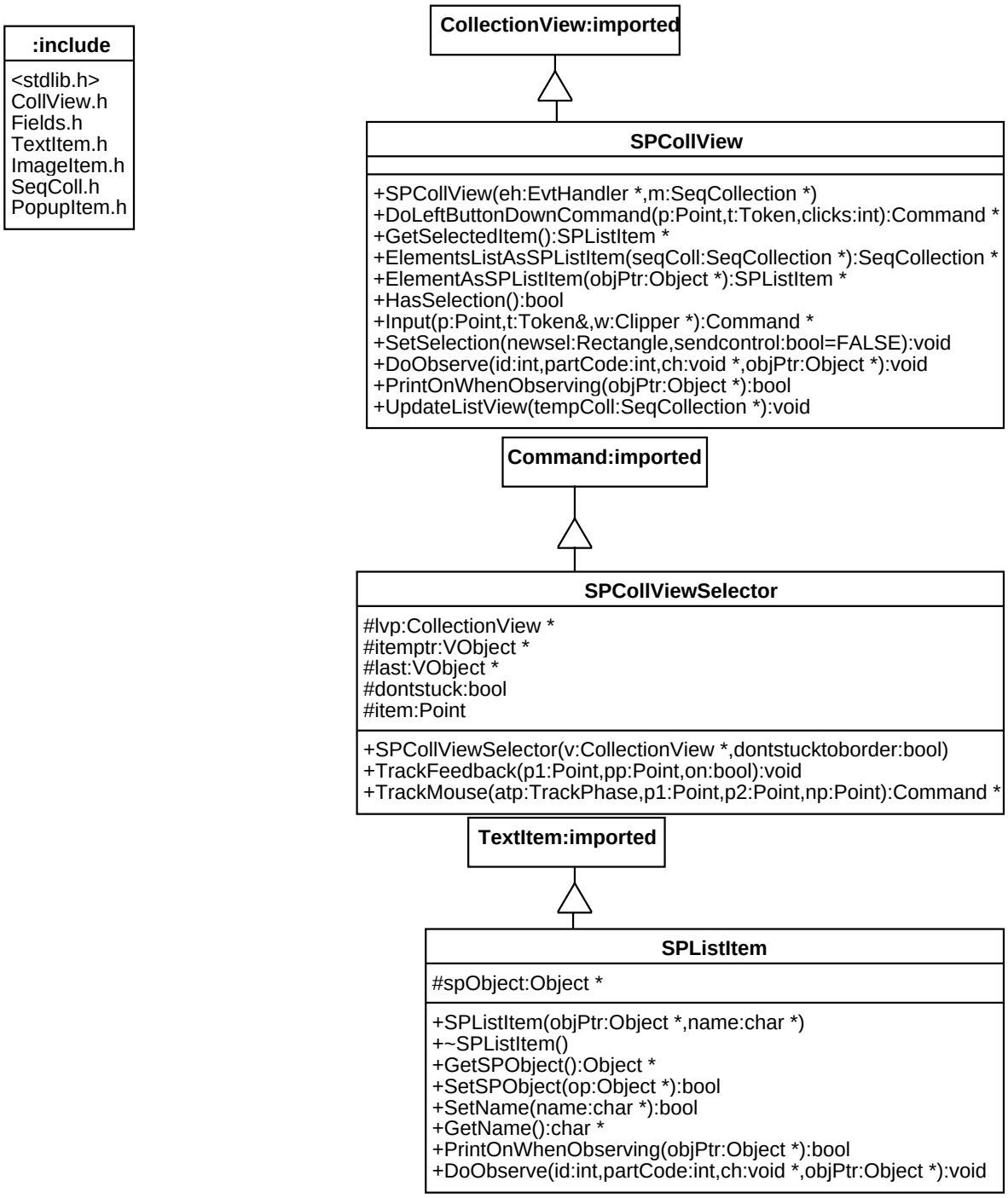
A 2.2.7 The Specification Unit Editor Window



A 2.2.8 The Tree View



A 2.2.9 The Collection View



A 2.2.10 The Text Field and Popup Button View