11-28-1994

A CASE tool supporting the MOSES development methodology

Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A CASE Tool Supporting the MOSES Development Methodology

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

by

Kevin K. Glupe

1994

To: Arthur Herriott
College of Arts and Sciences

This thesis, written by Kevin K. Glupe, and entitled A CASE Tool Supporting the MOSES Development Methodology, having been approved in respect to style and intellectual content, is referred to you for judgement.

We have read this thesis and recommend that it be approved.

Raimund Ege

Farah Arefi

John Comfort

Date of Defense: November 28, 1994

The thesis of Kevin K. Glupe is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Dr. Richard L. Campbell
Dean of Graduate Studies

Florida International University, 1994

ABSTRACT OF THE THESIS

A CASE TOOL SUPPORTING THE MOSES DEVELOPMENT METHODOLOGY

by

Kevin K. Glupe

Florida International University, 1994

Miami, Florida

Professor Raimund K. Ege, Major Professor

A new breed of CASE tools is taking advantage of object-oriented development. The CASE tool in this thesis will serve as a top-level guidance platform. The user will be able to follow an agenda in which his progress is recorded for each step of the development process. At the same time, the user will still be able to develop documents, diagrams and source code by using provided editors. The specific object-oriented methodology followed in this thesis is MOSES (Methodology for Object-Oriented Software Engineering of Systems). This advocates an iterative approach to development while providing both textual and graphical deliverables to assess the state of the product during it's lifecycle. The tool in this thesis is developed by following the steps and activities outlined by MOSES. To implement this CASE tool, Smalltalk is utilized since it provides full object-oriented support.

ABSTRACT OF THE THESIS

A CASE TOOL SUPPORTING THE MOSES DEVELOPMENT METHODOLOGY

by

Kevin K. Glupe

Florida International University, 1994

Miami, Florida

Professor Raimund K. Ege, Major Professor

A new breed of CASE tools is taking advantage of object-oriented development. The CASE tool in this thesis will serve as a top-level guidance platform. The user will be able to follow an agenda in which his progress is recorded for each step of the development process. At the same time, the user will still be able to develop documents, diagrams and source code by using provided editors. The specific object-oriented methodology followed in this thesis is MOSES (Methodology for Object-Oriented Software Engineering of Systems). This advocates an iterative approach to development while providing both textual and graphical deliverables to assess the state of the product during it's lifecycle. The tool in this thesis is developed by following the steps and activities outlined by MOSES. To implement this CASE tool, Smalltalk is utilized since it provides full object-oriented support.

Acknowledgements

I would like to thank all those who have been instrumental in the preparation of this thesis: my major professor and committee members who have been supportive and have contributed a great deal of time and energy, my family and friends for their understanding, Brian whose assistance in assembling the final product did not go unnoticed and Keith and Tina for their hospitality and warmth.

Kevin K. Glupe

Table of Contents

List of Figures

1 Introduction

CASE tools are essential to the development of industrial strength software. Not only do they support the integration of all software components to create the final product, but they act as a permanent storage facility for integral data structures and functions. Essentially, they provide the developer with a graphically-oriented platform to design, code and test individual program modules. This aids in breaking the edit, compile and debug cycle so prevalent in software development.

Each CASE tool supports a specific development paradigm. Historically, most development has followed the methodologies of structured design and functional decomposition. Naturally, most commercial CASE tools were constructed to support this paradigm. In recent years, though, object-oriented development has gained quite a bit of popularity, not only in educational and research institutions, but in commercial environments as well. As a result, the need has arisen for CASE tools supporting the object-oriented paradigm.

Most CASE tools that support object-oriented development concentrate on aiding the product designers and implementers. They typically provide graphically-oriented tools to document objects and their relationships. In addition, there is usually an option to enter the classes into persistent storage for the creation of a reusable library. To support these notions, low-level compiler and translation constructs are created. This description exemplifies the work done in projects such as Arcadia and Stone. However, tools such as

these do not support the developer throughout the entire product lifecycle from planning to maintenance, nor do they follow one specific object-oriented methodology.

The CASE tool developed as part of this thesis will serve as more than the typical design and code platform. It will be a top-level guidance tool that will allow the developers to document all phases and activities of the software product lifecycle. Specifically, the MOSES (Methodology for Object-Oriented Software Engineering of Systems) methodology [Hen93] and the fountain lifecycle model [HE93], both developed by Henderson-Sellers, will be followed. As a result, this tool will support the output of various textual and graphical deliverables described in this methodology.

The advantage to following a prescribed methodology and not just incorporating general object-oriented concepts is that a framework is provided that will result in the robust, correct and timely production of each software product. This methodology in particular was chosen due to both it's flexibility and rigidity. The developer is allowed to specify the process lifecycle details, yet is required to output certain deliverables to document the software engineering process. To prove that this methodology is indeed viable and useful, this CASE tool will actually be developed by following MOSES. Combining this feature with the fact that support will be provided for the entire development process from planning to maintenance will surely differentiate this tool from other CASE tools and prove it to be useful in commercial and educational environments.

2 Related Work

Since the mid-1980's, CASE tools have been widely used by software developers [You94]. This coincided with the time period in which PC's and workstations had evolved to the point where they could support such sophisticated environments. At first, these tools were based on the popular technology of the time, structured analysis and design. However, since object-oriented development has gained in popularity, CASE tool developers have grudgingly embraced the new technology.

Two types of CASE tools exist in today's computing environment. The first type is the commercial CASE tool. These environments are produced by private companies and cover a wide range of functionality depending on the price. In general, these tools allow the developer to conduct analysis and design activities such as drawing diagrams and identifying classes. The user is also able to implement classes and depending on the tool, access a class library or object-oriented database. The higher-end tools will actually generate source code. Some of the environments follow a specific methodology, such as Booch's, while some only apply general object-oriented concepts. A few examples of these tools are Paradigm Plus, System Architect and TurboCASE [You94].

Since these are produced by private companies, they cannot be discussed or compared to in this thesis unless they are purchased. Instead, the other type of CASE tool can be investigated, the ones developed in a research environment. These tools are typically developed at universities and placed in the public domain. However, most of these tools

are not as comprehensive as some of the privately developed environments. Two of these tools that are fairly thorough in attempting to provide CASE tool-like support are Arcadia and Stone. Arcadia is a higher-end tool providing more overall support while Stone is a lower-end tool focusing on their object-oriented database, OBST.

2.1 Arcadia

Arcadia is a project aimed at producing a development environment that guides a user through the design and implementation of a software product [Kad93]. This project attempts to provide object management, user interface development and management, measurement and evaluation, language processing and analysis and testing. Aside from furnishing these capabilities, the project attempts to prove that such an environment can be extensible, flexible, fast and incrementally improvable. These objectives are certainly conflicting and are hard to satisfy at the same time.

The people behind Arcadia believed that of all their goals, flexibility during the development process is of utmost importance [Kad93]. For this reason, Arcadia supports the development of custom user interfaces via UIDS, the User Interface Development System. In addition, multiple language support is provided via tailorable language processing capabilities.

Since design and coding are at the heart of any project, object management facilities such as persistence, type integrity and constraint maintenance are provided. To assure that these objects are combined to form high quality software, measurement, evaluation, analysis and

testing facilities are furnished. Since there is still no agreement on how to measure the quality of software, Arcadia's system was designed to be flexible and extensible.

2.2 Stone

Another environment that attempts to guide the developer through the creation of a software product is Stone. This is a complete design and code environment that provides the user with a variety of tools. The focus of this environment is OBST, the object management system [CRS92]. Essentially, this is an object-oriented database system that provides persistent storage for objects. OBST presents a strongly typed data model and provides services that can be utilized from the language of interest [CRS92].

To utilize the capabilities of OBST, four categories of tools are available. First, there is a standard library of reusable classes. Second, there are programs to inspect and manipulate class structures and the contents of the database. Third, there are facilities to integrate UNIX applications into the OBST environment. Last, there are utilities for system administration and reorganization of the database.

Since the end storage of objects in Stone are UNIX files, OBST provides the universal structurer and flattener generation. The former of these tools is simply a filter that translates a plain file in to a structured set of objects for direct use by OBST. The latter tool is a filter that translates an object structure into an ASCII representation for a UNIX file.

In addition to these tools, OBST can be inspected and manipulated interactively via the oshell and gsh. The oshell serves as an interactive debugging and inspection facility for

OBST while gsh graphically traverses the structure of the objects stored in OBST.

2.3 Synopsis

CASE tools will undoubtedly assist the developer in creating correct software in a more efficient manner. The area in which the user will benefit the most is in the design and coding of a product. Even the research environments, Arcadia and Stone will help to create a more manageable and controlled environment. By using the tools Arcadia provides, such as UIDS or the object-oriented database Stone provides, the developer will find it easier to progress through each phase of software creation. However, there is still a lack of focus on the overall management aspect of product development as well as a specific methodology. To truly obtain a cohesive environment that results in robust and timely software, a specific object-oriented methodology should be followed such as MOSES.

3 MOSES

The previously discussed environments provide the user with suitable tools to facilitate the development process. The common thread between these environments is that they concentrate on supporting to a small extent the design process and to a large extent the actual implementation. While developers will certainly find this useful for coding a large program, what they really need is support for all development phases. Since the goal of

object-oriented development is to use the same model for analysis, design and implementation, a corresponding CASE tool must also support these three key phases. In addition, the developer needs to follow a set of guidelines for producing diagrams, documents and source code. A useful CASE tool will provide this agenda that the developer may abide by. MOSES is a perfect candidate methodology for warranting such CASE tool support.

MOSES (Methodology for Object-Oriented Software Engineering of Systems) is a lifecycle methodology that presents a seamless transition from the planning stage to product maintenance [Hen93]. It provides a set of activities, phases and deliverables to deliver a robust and timely system developed under the object-oriented paradigm. As of this date, commercial and educational institutions have successfully utilized this methodology to build sound software systems. MOSES is supported by textual and graphical notations throughout the product development lifecycle to provide for deliverables that are created when reaching established milestones.

This seamless transition advertised by MOSES is a direct result of using the object-oriented paradigm. The ability to use one object model throughout the entire development process results in a smooth transition from analysis to design to implementation. This is in contrast to a structured model where the developer must use one model followed by another one in design ending in yet a different model for implementation. If an object-oriented methodology is used, the same language can be utilized for analysis, design and implementation. In fact, if the design is represented by

using an OOPL (object-oriented programming language), then the need for a distinct implementation phase disappears. Although MOSES stresses the benefits of using an OOPL, it is truly a flexible environment in that the implementation language can be procedural if necessary.

3.1 Objectives

In keeping with the need to maintain a flexible development environment, MOSES maintains six objectives [Hen93].

1. Provision of software engineering support for both large and small object-oriented systems development.
2. Provision of a development process that supports a smooth transition from initial modelling through to implementation.
3. Provision of a development approach for flexible and extensible systems.
4. Provision of guidelines for project and product management.
5. Support for development of highly reusable classes, systems and designs.
6. Underpinning of the software development with a quality objective.

To accomplish these goals, two lifecycles are identified, the product lifecycle and the process lifecycle. This is shown in figure 1. In this context, each product lifecycle may contain multiple process lifecycles.

3.2 Product Lifecycle

The product lifecycle ranges from the inception to the retirement of the product. Each enhancement is successively incorporated into this framework. There are two major periods in the product lifecycle; theGrowth Period and the Maturity Period. Whereas the Growth Period encompasses the major thrust of development work, the Maturity Period spans the majority of the product lifecycle. By using object-oriented development, the

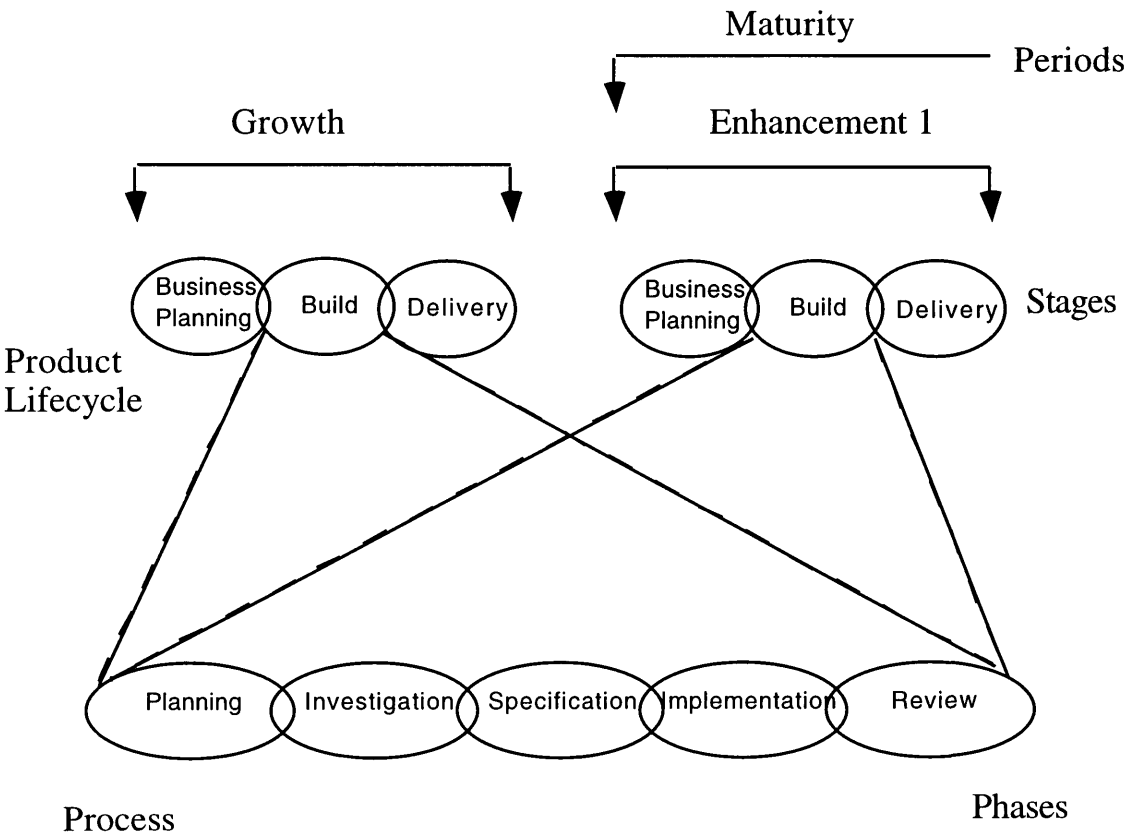


Figure 1: The overall framework of MOSES

Maturity Period is defined as being more than the strict maintenance of a product. Instead, it is classified as successively applying the process lifecycle to each further enhancement. Therefore, for each release of a product, all lifecycle phases are followed rather than strictly modifying the existing program code with no prior planning.

The Growth period, as well as each Enhancement Period, can be divided into three stages; the Business Planning Stage, the Build Stage and the Delivery Stage. The Business Planning Stage consists of feasibility studies, risk assessment and cost benefit analysis. These activities are incorporated into the deliverable for this stage, the business planning report. At this point, the developer will assess the feasibilility of constructing the product or undertaking the proposed enhancement. Should the decision be made to continue, the next stage of the product lifecycle is entered.

In the Build Stage, the process lifecycle is applied. MOSES does not impose any specific lifecycle model, but it is highly suggested that an iterative model be used such as Henderson-Sellers' fountain model. This is due to the iterative nature of object-oriented development. When constructing a product, developers typically need to progress through the phases more than one time. Each iteration builds upon the last until a finished product is achieved. As will be described later, the fountain model allows the developer to move effortlessly from one stage to the next and back again.

Upon completion of the Build Stage, the Delivery Stage is invoked in which the product is installed, tested and reviewed. At this time, the Growth Period is completed and any further changes will be treated as a product enhancement. However, each enhancement

period follows the exact same sequence of events that take place in the Growth period. This is because, the process lifecycle is applied repeatedly until the product is retired.

3.3 Process Lifecycle

As seen by the lack of distinction between the Growth period and a Maturity period, the process lifecycle is viewed as only a part of the life of a product, not as the whole life. MOSES allows for any chosen process lifecycle model to be utilized during the Build Stage. The iterative development method and the waterfall method are the two most popular means. The waterfall model consists of one sequential pass through each step of the process lifecycle. Conversely, an iterative development model allows for multiple iterations over the steps of the model, thereby invoking more of a fountain-like shape. This flexibility proves to be useful for corporate environments who are slowly migrating away from the structured development process. Those who choose an iterative process, however, gain the advantage of not having to specify a complete design before some implementation can be started or some prototypes built. Thus early design flaws can be rectified as user and developer feedback is received. In addition, parallel development of separate subsystems can occur. As one group is designing one subsystem, another group can be designing or implementing a separate subsystem.

Textual Deliverables	Graphical Deliverables
Business Planning Report	Analysis Diagram
Iteration Plan	Design Diagram
Actor Glossary	Inheritance Diagram
Scenario List	Contract Diagram
User Requirements Specification	Class Interface Diagram
Subsystem Requirements Specification	
Source Code	
Review Report	
Test Report	

Figure 2: MOSES Deliverables

No matter which process lifecycle model is chosen, five major structural units known as phases can be identified, Planning, Investigation, Specification, Implementation and

Review. The outcome of each phase is marked by certain textual and/or graphical deliverables displayed in figure 2. If the iteration of the phase is to be the final iteration, then the deliverable produced will be a final document, otherwise it will be an interim management deliverable to be amended or added to.

3.3.1 Planning

The first phase of the lifecycle consists of setting goals for each iteration of the development process. This encompasses estimating resources needed, assigning work schedules and allocating personnel. A Business Planning Report is generated to provide the overall framework of goals to be met while undergoing product development. Since an iterative development process is advocated, an Iteration Plan for each iteration is produced in which specific objectives are outlined such as how much time will be devoted to each individual phase.

3.3.2 Investigation

The next phase involves gaining an understanding of the problem. By interviewing potential users and managers and retrieving relevant information, a URS (User Requirements Specification) written in the language of the user can be created. This will serve as the binding contract between the client and the developer specifying exactly what services the software product will provide. The developer also identifies the types of people, or actors, who will be using the system. Each actor is defined and entered into the Actor Glossary. To further identify the system functionalities, the scenarios that each actor

participates in are documented in the Scenario List.

3.3.3 Specification

The Specification Phase consists of transforming the system requirements into the object model. In traditional models, this is known as OOA (object-oriented analysis) and OOD (object-oriented design). But MOSES, by supporting a seamless transition, combines these two phases into one instead of OOA followed by OOD [Hen93]. Therefore, if so desired, the developer may engage in some design before completing analysis.

One of the analysis activities is to identify subsystems. This serves the purpose of breaking down the problem into manageable-sized pieces. For each subsystem, the appropriate requirements are extracted from the formal URS and included in a Subsystem Requirements Specification. By doing this, development groups can be formed to simultaneously develop each of the subsystems.

Once the problem has been organized in a hierarchical fashion, classes can be identified and placed in Analysis Diagrams. At this point, the boundary between analysis and design becomes blurred as Analysis Diagrams are transformed into Design Diagrams. Essentially, classes are first identified and then iteratively refined until a distinct specification for each class can stated. This specification is documented in a Class Interface Diagram. To further clarify the hierarchy of classes, the inheritance hierarchy is defined in Inheritance Diagrams. Since, classes must interact with one another to produce a working system, the contracts between classes are specified in Contract Diagrams.

3.3.4 Implementation

The Implementation Phase consists of creating source code from the design developed in the Specification Phase. If the programming language is object-oriented then this is a trivial endeavor. Essentially, only a translation from the Class Interface Diagram to the syntax of the language is necessary. In addition, the messages passed between objects have already been specified in the Contract Diagrams. If the target language is not object-oriented then the seamlessness is lost and more effort will be involved. No matter what language is used, the source code must be thoroughly tested to verify that the user's requirements have been satisfied. The results of these tests are stored in the Test Reports.

3.3.5 Review

The last phase, the Review Phase, is where the quality of the software developed is assessed. This includes the quality of the development process, the quality of the deliverables and the testing of the subsystems. Each assessment is recorded in a Review Report. For personal use, the developer might also verify that the goals set forth in the Iteration Plans have been met.

3.4 Application

These five phases can be applied to either the system as a whole or to an individual subsystem. Since most subsystems are physically and semantically separate from each

Activity	Plan	Inv	Spec	Impl	Rvw
Contract Specification			X	X	
Documentation Review	X	X	X	X	X
Event Model Construction			X		
Generalization For Reuse			X	X	
Genericity Specification				X	
Inheritance Identification			X	X	
Inheritance Specification			X	X	
Iteration Plan Development	X				
Library Class Incorporation			X	X	
ObjectChart Construction			X	X	
O/C Identification			X	X	
Optimization			X	X	
Quality Evaluation			X	X	X
Scenario Development		X			
Service Identification			X	X	
Subsystem Co-ordination	X				
Subsystem Identification	X		X		X
Testing			X	X	X
Translation to OOPL				X	
		X			

Figure 3: Activities of MOSES

other, each subsystem can go through it's lifecycle phases in parallel. Thus each subsystem can be in any phase at any given point in time. The system, as a compilation of it's subsystems, can therefore not be characterized as in any one phase while the Build Stage is in progress. Eventually, though, all subsystems will enter the Review Stage, bringing the system into sync.

As each phase progresses, MOSES requires that certain tasks be accomplished to produce the deliverables. These tasks are known as activities. Each activity has a specific purpose and a description of how to accomplish that purpose. Also, activities can occur in one or more phases of the lifecycle. This is shown in figure 3. Their presence in each phase may vary in perspective and extent. However, each activity's conclusion does not produce a deliverable. A deliverable may require the completion of several activities and specifies partial completion of a phase. For example, to produce an Inheritance Diagram, two activities are necessary, inheritance identification and inheritance specification. However, the production of an Inheritance Diagram does not signify the completion of the Specification Phase.

Depending on which lifecycle model is chosen, certain chronological breakpoints where documentation can be delivered need to be identified. These breakpoints are at the end of each phase. Should the waterfall model approach be taken, then each phase must be completed and "signed off" in succession before the next phase is invoked. Each subsystem can still be developed in parallel, but during the development process each subsystem will be in the same stage. This approach, though feasible, is not recommended.

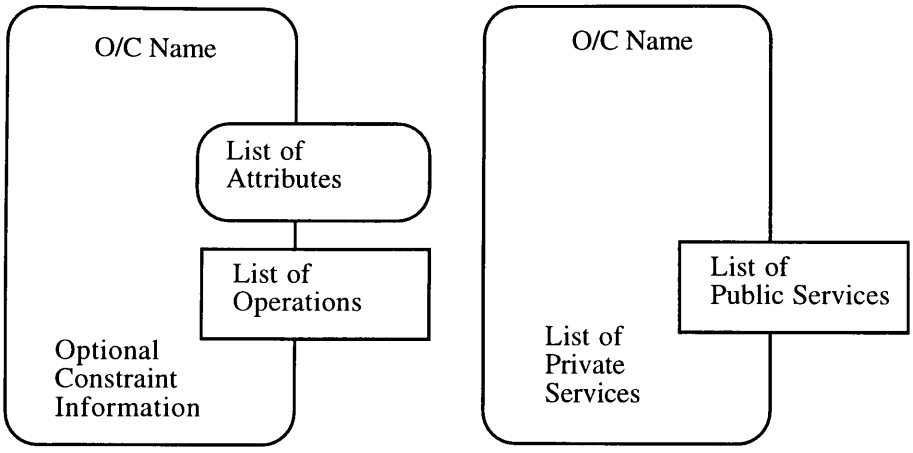
A more preferable approach is an iterative one such as the fountain lifecycle model. This implies that the growth period will consist of numerous iterations of the process lifecycle. Since documents will be undergoing multiple iterations, the developers must decide how to handle multiple versions of each document. The document should not be "signed off" until the last iteration has been completed.

3.5 Notation

Since MOSES provides for five types of graphical deliverables, a standard notation must be developed [EH93]. The object, the core component of analysis, design and implementation, needs to be represented in each of it's many stages. Since analysis objects, design objects and classes can be portrayed similarly, but at different stages of the lifecycle, they will be referred to in the diagrams as O/C's [Hen92]. The notation that is defined must not directly correspond to any language since MOSES is free of any language restrictions.

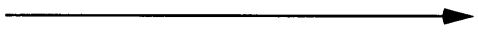
Although analysis and design are intricately intertwined in MOSES, analysis can be denoted as the identification of objects, their attributes and their operations. Therefore, an icon such as the one in figure 4a will be used. However, the focus for design activities centers on uniform reference and information hiding. Thus an icon such as the one in figure 4b is appropriate. This specifies the interface the O/C provides to other O/C's.

Aside from representing the O/C's themselves, their interactions also need a suitable representation. The relationships identified at the analysis level will need to be altered



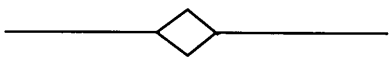
a) b)

Figure 4: O/C notation



a)

b)



c)

Figure 5: Relationship notation

somewhat at the design stage since most object-oriented programming languages do not provide a way to represent both association and aggregation other than a client-server relationship. To represent the client-server relationship, an arrow such as the one shown in figure 5a can be used. To show an inheritance relationship, a thicker arrow such as the one in figure 5b is most appropriate. The other relationship, aggregation, is represented by the symbol in figure 5c.

4 Fountain Lifecycle Model

To take full advantage of MOSES' qualities, Henderson-Sellers developed the fountain lifecycle model [HE93]. As shown in figure 6, this model sharply contrasts with the standard waterfall model. The major difference is the sequencing of steps. In the waterfall model, each step is sequentially followed by the next step, implying no overlap. However, in the fountain model, there is considerable overlapping and merging of steps. This means that it is possible for analysis and design to occur simultaneously. In addition, the possibility exists for iterative cycles across one or more lifecycle steps. Thus this model is classified as an iterative development process.

According to MOSES, the process lifecycle model has five phases, Planning, Investigation, Specification, Implementation and Review. The fountain model appears to have more than five phases, but they can be appropriately mapped to conform to MOSES. The Investigation Phase would correspond to the fountain model's requirements and feasibility study. The Specification Phase would map to the steps of analysis, system

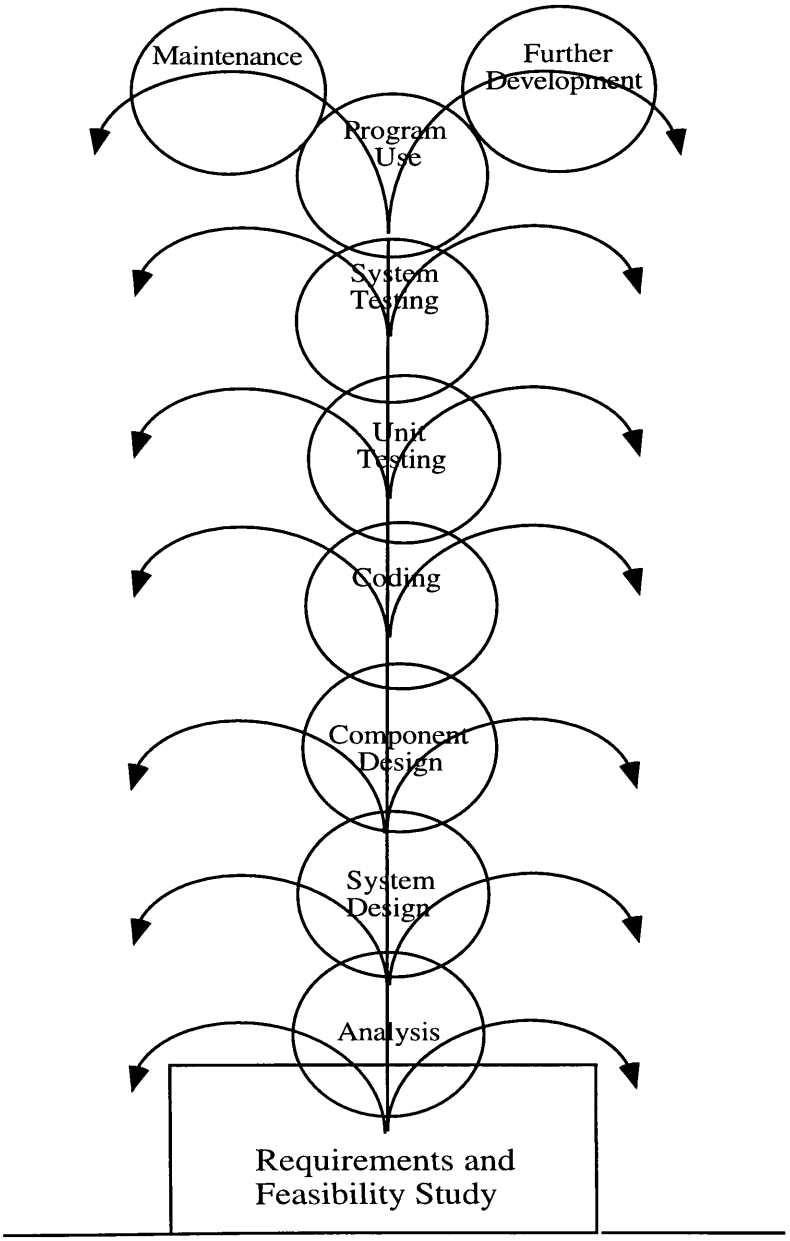


Figure 6: The iterative Fountain Lifecycle Model used by MOSES

design and component design. Last, the Implementation Phase corresponds to coding, unit testing and system testing. Any further development or maintenance as specified in the fountain model would be considered an enhancement in MOSES.

To complete each phase of the lifecycle, MOSES requires that certain activities be accomplished. Although these were outlined in figure 3, there are seven core activities [HE93] specified by the fountain model that need to be explained in more detail. Since this will be in the context of an iterative development process, no distinction will be made between an object and a class. As in the diagram notation, analysis objects, design objects and classes will be referred to as O/C's.

The first core activity is to prepare the systems requirements document. This is prepared in the language of the user but discussed and analyzed in terms of O/C's and their services. The second step is to analyze this document and extract real-world O/C's and their services. These O/C's are labeled with a name, attributes and operations. As these O/C's become evident, the next activity can commence, namely identifying the interactions among O/C's. These interactions can be recognized in terms of association, aggregation and generalization relationships. In addition, a useful exercise at this point is to identify the contracts between client and server O/C's.

Even though there can be considerable overlap between analysis and design, the fourth activity is the unofficial start of the design process. Here, top-level design diagrams are drawn at first, followed by increasingly lower-level diagrams to illustrate the O/C services and relationships. To add to these diagrams, the fifth activity integrates library O/C's. In

addition, the class interfaces are specified in detail. Once these details have been specified, coding can begin.

At this point, design merges into implementation with the specification of inheritance hierarchies and the coding of classes as specified an activity six. As coding and testing progress and are finalized, activity seven indicates the need to generalize for later reuse. To create the reusable class library, candidate classes need to be generalized to become non-project specific. This exercise indicates that the product is ready for delivery.

Although MOSES advocates the performance of more activities than these seven, these activities are essential to the successful completion of any software project and should not be overlooked. Since this model is an iterative one, these activities may be performed more than once depending on the number of iterations. In addition, these same activities will be needed for each product enhancement. Thus the textual and graphical output of these activities will need to be documented according to which iteration is being applied, the version of the document and whether the document corresponds to the development or enhancement of the product. This feature is just one of many that needs to be addressed in the construction of this CASE tool.

5 CASE Tool

To demonstrate how MOSES can be used to successfully develop a software product, the CASE tool in this thesis was constructed by following the phases and activities outlined by

MOSES. As a result, the deliverables previously described had to be created. Some of the deliverables were not necessary and therefore not developed. For example, the Business Planning Report is not aplicable since it outlines the projections and the feasibility of a product in the context of a software development company. The sections of the thesis to follow outline the relevant deliverables that were created.

5.1 Overview

The CASE tool in this thesis supports software development by allowing the user to follow the activities and phases of MOSES. The objective of the environment is to provide the user with top-level guidance and support through all phases of software dvelopment. By enabling the user to obtain a clear picture of how much development has been done and how much more there is to do, it will be easier to produce timely and robust software.

Since MOSES advocates the usage of the fountain lifecycle model, this tool breaks the development process into four major phases, planning, analysis, design and implementation. In each phase, certain deliverables must be produced. The planning phase encompasses the Business Planning Report and Iteration Plans. The analysis phase produces the User Requirements Specification, the Actor Glossary, the Scenario List, Subsystem Requirements Specifications and Analysis Diagrams. The design phase results in Design Diagrams, Inheritance Diagrams, Class Interface Diagrams, Contract Diagrams and ObjectCharts. The last phase, implementation, produces source code and Test Reports. All phases allow Review Reports to be generated.

Each of these deliverables is categorized as being one of three types, single, multiple or hierarchical. The single deliverables can only be produced once for each product. Examples of these are the User Requirements Specification, the Actor Glossary, the Scenario List and the Business Planning Report. The multiple deliverables may have many occurences throughout product development. The Iteration Plan, Subsystem Requirements Specification, Test Report, Review Report, Inheritance Diagram, Contract Diagram, Class Interface Diagram and ObjectChart are all examples of multiple deliverables. The hierarchical diagrams are special cases of multiple deliverables in that each diagram may be assigned to a class in another diagram of the same type. The only deliverables of this type are the Analysis Diagrams and Design Diagrams.

For the user to produce these deliverables he may take full advantage of the five major features offered by this CASE tool. The first feature is the agenda editor. This enables the user to visualize the amount of development accomplished thus far. Specifically, the developer is presented with the number of modifications made to the deliverables for each phase of development. The user is then able to selectively view a phase to see how many modifications have been made to each deliverable. Once this information has been reviewed, the user may mark a phase as being completed or as needing additional work. Essentially, the agenda editor allows the user to gain an overall view of the state of a system as development is in progress.

Another feature of this tool is it's ability to keep an account of modifications made to every deliverable. When a deliverable is created, it is denoted as being the first version.

Each time the user edits the deliverable, the version number is increased and the new modification is saved for later inspection by the developer. As a result, the user is able to recall any version of a deliverable and modify it or view it.

An additional feature is the ability to link deliverables to one another. As the user steps through the development process, he will find it useful to follow an idea from analysis to design to implementation. Therefore, this tool allows the user to select a deliverable such as an Analysis Diagram and link it to any other existing deliverable such as a Design Diagram. By following the link, the user will be placed in an environment in which he may operate on the deliverable. If this deliverable is itself linked to others then the user may follow additional links until no more are present.

One other feature of this environment is the provided graphical and textual editors in which the user is able to prepare all deliverables. The text editor provides the user with functionalities that most standard editors provide. Some of these features include the ability to cut, copy and paste text amongst one or more documents. In addition, the user may undo or repeat the last performed operation. When the user has completed preparing the document he is able to save it or discard his text. The other provided editor enables the user to select predefined components such as a box, line, arrow or aggregation symbol and paste them on to a drawing canvas. These components can then be arranged in any format the user desires. This diagram is then able to be saved or discarded.

The last feature of this tool is the ability to provide a secure development environment. If the security feature is enabled, the user must have an account on the system to perform

any of the operations. Since this tool divides development into four phases, the users are placed into groups which correspond to the phases. Each of the development groups is given certain permissions to operate on the deliverables. When a user logs in, he enters his user name and password followed by a group that he is a member of. He may then perform operations on deliverables according to the permissins granted to the group he is logged in as.

5.2 Actors

Upon successfully logging in to the system, each user assumes the role of one or more actors.

5.2.1 Planner

A planner conducts feasibility studies, risk analysis and cost benefit analysis for a proposed product or a proposed product enhancement. The results are incorporated into the Business Planning Report. For each iteration of the product development, he assigns and schedules workers, estimates resources needed and sets goals to be met in the iteration. This information is incorporated in to the Iteration Plan.

5.2.2 Requirement Gatherer

The requirement gatherer elicits requirements for a software product from the customer via

interviews and discussions. He is responsible for meeting with the customer to establish an agreed upon list of requirements that will be put in a binding legal contract known as the formal User Requirements Specification. He is also responsible for determining which of the user requirements are pertinent to a particular subsystem. These are placed in the Subsystem Requirements Specification.

5.2.3 Analyzer

The analyzer identifies O/C's and subsystems from the formal User Requirements Specification. This information is placed in analysis diagrams. For each O/C, he then identifies attributes, operations and relationships with other O/C's. These are stored in analysis diagrams as well. While preparing this overall analysis, the Analyzer also identifies actors and the scenarios that they participate in. This information is recorded in the Actor Glossary and the Scenario List.

5.2.4 Designer

The designer translates the analysis diagrams into corresponding design diagrams. These diagrams show the O/C's, their services and the client-server relationships that they participate in. For every O/C in a design diagram, he specifies the class features and interface in a class interface diagram. He also identifies generalization hierarchies amongst O/C's and records them in inheritance diagrams. The services that each of these O/C's has to offer are identified by the designer and entered into contract diagrams. He also is

responsible for producing objectcharts and event models.

5.2.5 Programmer

The programmer translates the class interface diagram for each class into actual source code. If specified by the designer, he is responsible for retrieving library classes and incorporating them into the code he is developing. As he implements each class, he also compiles and debugs his source code.

5.2.6 Quality Assurance

The tester reviews all documents, diagrams and source code modules. He develops test cases for classes and performs the tests. He also performs integration and subsystem testing. The result of each test case is incorporated into a Test Report. For all diagrams and documents, the results of his reviews are recorded in Review Reports.

5.2.7 Project Manager

The project manager is in charge of either the entire project or if there is a managerial hierarchy, a portion of the project. His responsibilities include reviewing plans, assuring that a project is on schedule and within budget and maintaining a good working relationship with the customer. Since he is in charge of the project he is also responsible for managing all employees involved in the production of this software project.

5.2.8 Project Secretary

The project secretary is responsible for scheduling customer contact and recording the results. He also provides users accounts and assigns permissions accordingly to ensure that the correct people have access to the portions of the project that they need to access.

5.3 Scenarios

Each of the actors previously defined interacts with the system in one or more scenarios. A scenario corresponds to the user performing a sequence of system functions. Since each user can assume the role of more than one actor, he can potentially participate in a large number of scenarios. For a thorough and detailed explanation of all possible scenarios, see Appendix 1.

5.3.1 Operations Performed on Deliverables

For all scenarios that an actor may participate in, certain common operations need to be defined that may occur when operating on a deliverable. Each time the user elects to operate on a deliverable, the a number of different scenarios may develop. By default, the latest version of the deliverable will be used. Some of the more common scenarios have the user creating, editing, deleting, annotating or viewing a deliverable. A full explanation of these is provided in Appendix 1, section 1.

5.3.2 Scenarios Involving the Planner

The planner typically engages in scenarios in which he is performing operations on one of the planning deliverables such as the Business Planning Report or an Iteration Plan. A more detailed explanation is found in Appendix 1, section 2.

5.3.3 Scenarios involving the Requirement Gatherer

The requirement gatherer's main responsibility is to identify and record system requirements. He is typically involved with scenarios in which he operates on either the User Requirements Specification or a Subsystem Requirements Specification. For a full explanation, see Appendix 1, section 3.

5.3.4 Scenarios involving the Analyzer

The analyzer concerns himself with scenarios involving either subsystem management, including identifying and removing subsystems or operations performed on the Actor Glossary, Scenario List and Analysis Diagrams. A more detailed explanation of the analyzer's scenarios is listed in Appendix 1, section 4.

5.3.5 Scenarios involving the Designer

The scenarios in which the designer is involved with pertain to operating on the design deliverables. These deliverables include Design Diagrams, Inheritance Diagrams, Class Interface Diagrams, Contract Diagrams and ObjectCharts. These scenarios are outlined in Appendix 1, section 5.

5.3.6 Scenarios involving the Programmer

The Programmer participates in scenarios involving product implementation. He typically concerns himself with all source code development. These scenarios are explained in Appendix 1, section 6.

5.3.7 Scenarios involving the Quality Assurance Person

The Quality Assurance Person is responsible for reviewing and testing certain portions of a product. He engages in scenarios involving Review Reports and Test Reports. For a thorough explanation, see Appendix 1, section 7.

5.3.8 Scenarios involving the Project Secretary and Project Manager

The Project Manager and Project Secretary handle all background management functions for the CASE tool. This typically involves product management, user management and assigning permissions to development groups for each of the deliverables. A detailed outline of the relevant scenarios is in Appendix 1, section 8.

5.3.9 Scenarios involving all Users

All users may either view user account information, view product information or invoke the agenda editor. These scenarios are outlined in Appendix 1, section 9.

5.4 Functionality

5.4.1 Security Feature

This CASE tool has a security feature which can be enabled or disabled by a member of the management group. When the security feature is enabled, the user must enter a user name and password to log into the system. Upon successfully logging in, the user is presented with a list of groups that he is a member of. Depending on which group he selects, the user will be able to perform certain operation on the deliverables. If the security feature is disabled then the user will have the main menu displayed without any user verification. He is then able to alter any existing deliverable or create new ones.

5.4.2 Development Groups

There are five groups that users may be a part of. They are planning, analysis, design, implementation and management. Each group gets assigned either read and write, read only or no permission to access each of the deliverables. If the user has read only permissions for a deliverable then he may not alter it's contents or create a new one of that specific type. In addition, he may not enter an annotation for the deliverable, link it to any other deliverable, remove a link the deliverable has to another deliverable or sign off the deliverable. Should the user have read and write access to the deliverable, he may perform any operation on the deliverable including altering it's contents and creating new ones of that type. When the user has no permission for the deliverable, he may not perform any

operations on it.

Development groups exist so that users may access only the portions of the product that are absolutely necessary. Each time a user logs in, he is required to select a group. Essentially, this creates multiple sessions that the user can engage in. For example, logging in as a member of the analysis group allows the user to perform different operations than when he is logged in as a member of the design group. This concept is likened to Unix where a user who has the root password can log in under his own account or the root account. Each account can perform very different functions.

When the user logs in as a particular group member, the permissions that he inherits are valid for every product in the system. Therefore, if a developer is a member of the design group which typically has access to Design Diagrams then the user may access Design Diagrams for each and every product. If a management group member should change the permissions so that the designer may no longer access the Design Diagrams then he is denied access to every Design Diagram in the system. This is so even if he had created Design Diagrams in the past when he did have permission.

To give each group the correct permissions for accessing the deliverables, the management member interacts with the window in figure 7. For the user to arrive at this window, he must follow the sequence of steps outlined in scenario four for the project manager or secretary. This may be found in Appendix 1, section 8. Depending on the group, certain deliverables must be accessible for creation and alteration. For example, a developer in the design group would typically need read and write permission to the

products related to the design phase such as the Design Diagrams, the Inheritance Diagrams, the Class Interface Diagrams, the Contract Diagrams and the Objectcharts. In addition, he would most likely require read only access to analysis products such as the Analysis Diagrams, the User Requirements Specification, the Subsystem Requirements Specifications, the Actor Glossary and the Scenario list. Since his purpose is not to implement, he would not need any access to implementation deliverables such as the source code and Test Reports.

5.4.3 User Accounts

All user accounts are configured by a member of the management group. Figure 8 displays the window in which the manager enters the necessary information. This window is arrived at by following the sequence of steps outlined in scenario 3a for the project secretary or manager. This is located in Appendix 1, section 8. Each user must be assigned a user name and a password. In addition, the user may be assigned to zero or more development groups. If the user is assigned to no groups then he will not be able to log in. The management member may also specify whether the user account should be enabled or disabled. A disabled account denies the user access to logging in to the system although information concerning the user is retained. The only restriction to adding a user is that a duplicate user name must not be entered.

Once in the system, the user may have his account information altered or deleted by a management group member. When altering the account, the manager may enable or disable

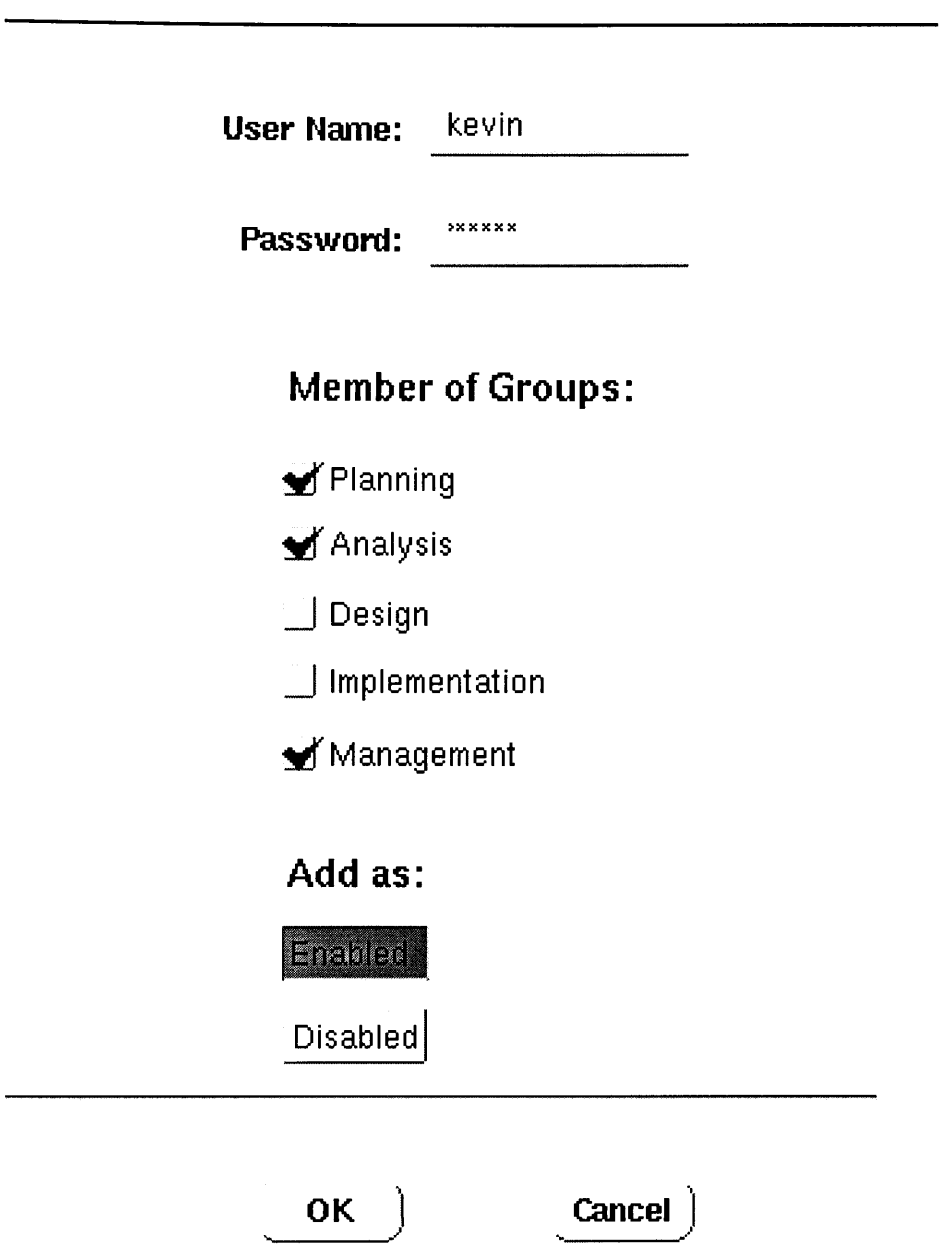


Figure 8: The window to add a user

the account, change the groups the user is in, change the password or change the user's

name. To delete a user, the manager only needs to verify the removal. When a user is deleted, all information concerning the account is removed from the system and the user is removed from all of the groups that he belonged to. As a result, the user may no longer log in to the system.

Any user of the system, whether he is a management group member or not, may view the user account information. The user may select any account and examine what groups the user belongs to and whether the account is enabled or disabled.

5.4.4 Software Products

To perform operations on deliverables, a product must first be started. To begin a product, a management member enters a product name, a beginning release number, a starting date and a project manager's name. The account for the project manager must already exist in the system. If it does not, the user will be able to add an account for him. The only restriction for starting a product is that the product name must not already exist in the system.

Since starting a product actually starts the first release of the product, the manager may start additional releases as well. Each release that he begins must have a larger number than the previous releases' number. The manager must also enter a starting date and the project manager's name. As specified in starting a product, the project manager's name must already exist in the system list of users. If it does not, the user will be able to add it.

The manager will also be able to change any information pertaining to the last release of a

product. Should he choose to alter the release number, the new number must not be less than the previous existing release number. The manager may also alter the starting date and the project manager name. As before, the project manager's name must already exist in the system. One additional piece of information the manager may supply is the ending date for the release.

Another function a manager may perform is to delete the last release. Upon choosing this option, he will be asked to confirm his decision. By deleting a release, all corresponding deliverables and subsystems will be removed from the system. The manager, however, will not be able to remove any release other than the latest one.

One function that all users may perform is viewing the product information. The user must first select a product and one of it's releases. He then is presented with the start date, the product manager's name and if one exists, the ending date. The sequence of steps is outlined in scenario two for all users from Appendix 1, section 9 and results in the window in figure 9.

5.4.5 Subsystems

One of the MOSES analysis activities is to identify subsystems. Any member of the analysis group is able to perform basic subsystem management activities such as adding and deleting subsystems and changing the subsystem's names. When entering a new subsystem name, the user is not allowed to enter a name that already exists since each subsystem is a separate component of the overall product and therefore must be unique.

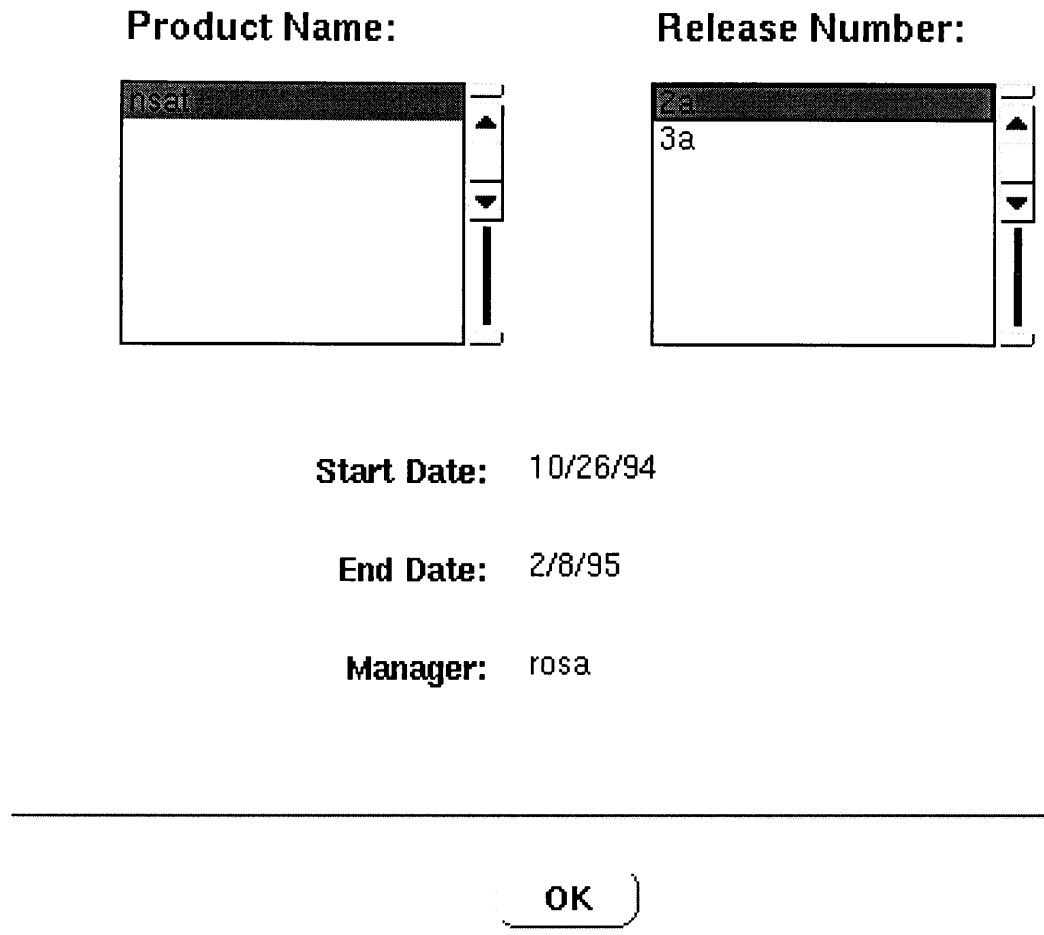


Figure 9: The window to view product information

Since the purpose of identifying subsystems is to divide the development work, each subsystem may have one or more deliverables assigned to it. Should this be the case, the

user will be notified as to which deliverables are assigned to the subsystem. He may then choose to continue with the deletion or cancel it. If he deletes the subsystem then any deliverables that are assigned to it will be assigned to no subsystem after the deletion. If there are no deliverables assigned to the subsystem then the user will only receive a prompt to verify the subsystem deletion.

5.4.6 Deliverables

Depending upon which group a user belongs to, he may operate on one or more types of deliverables. The user will only be allowed to access those deliverables for which he has read only or read and write permission. If the user does not belong to a particular group then he may not access any of the deliverables for which that group has responsibility. A warning message will be displayed if he attempts to access those deliverables. For example, if the user does not belong to the design group then he will not be able to access any of the Design Diagrams.

To actually operate on the deliverables, the user must first select a product and one of it's releases. Upon supplying this information, the user will be presented with the screen in figure 10. The user may then become involved in one of the deliverable scenarios from Appendix 1, section one. Initially, the user will not be able to perform any operations. He must first select a deliverable from the menu button labeled **Deliverable**. The menu presented to the user will consist only of deliverables corresponding to the function he selected from the main menu. For example, if he elected to perform planning operations

then he will only be able to operate on the Business Planning Report and Iteration Plans.

Once the user selects a deliverable, additional information is presented. This includes the subsystem the deliverable is assigned to, if any, and the latest version number. If the deliverable is a textual document then the contents will be displayed in the read only text display at the bottom of the window. For signed off deliverables, the associated box is marked with a check. By pressing the **Version** button, the user is able to select any version that he has created thus far. Should he choose a previous version, the version number in the window is updated. If the deliverable is textual then the text is displayed in the read only text area. For which ever version is selected, the user may also elect to display relevant information such as the author and the start date by pressing the **Version Info** button .

For the special case of a multiple deliverable, the **Name&Number** button is enabled. By pressing this button, the user is able to select one of the existing numbered multiple deliverables. Before pressing this button, the user may elect to create another diagram or document by pushing the **Operation** button and selecting the create option. When the user does select one of the multiple deliverables, he is presented with the same information as for a single deliverable. However, the deliverable name and number that he has chosen are displayed in the window. If the deliverable is a hierarchical diagram and is assigned to a class in another diagram then the class name that it is assigned to and the diagram number that the class is in are also displayed.

Depending on whether the deliverable is signed off or not, the state and type of the

deliverable selected, the permissions of the group the user is logged in as and whether the phase is completed or not, the user is presented with a list of operations that he may perform on the deliverable. If the user has read only permission for the deliverable he has chosen then he is restricted to viewing, printing and performing annotation operations. In particular, he may not sign off the deliverable or alter it's contents.

Should the developer have read and write permission to the deliverable then he is able to perform additional operations. First, he is able to create the deliverable. If the deliverable is a single deliverable then he is only able to do this once. If it is a multiple deliverable, then he is able to create as many as he needs. Once a deliverable exists, the user may also edit it's contents or delete it. The user may also "sign off" the deliverable. This is done by using the **Sign Off** button in the lower left corner of the window. If the deliverable is already signed off then this button serves to sign the deliverable on again so that it may be modified.

For multiple deliverables, users with read and write permission may perform two additional sets of operations. The first one is assigning the deliverable to a subsystem. When choosing this option, the user selects an existing subsystem from a list. The subsystem name is then displayed in the window. Alternatively, the user may unassign a deliverable to a subsystem. This is reflected in the window by clearing the subsystem field. If the deliverable is a Design Diagram or Analysis Diagram then the user may opt to assign it to a class in another diagram of the same type. For example if the current deliverable is an Analysis Diagram then he may only assign it to a class in another Analysis

Diagram. If no other Analysis Diagrams exist then the user will not be given this option. Once he selects the diagram, the user is prompted to enter the name of the class in the other diagram. This information is then displayed in the appropriate fields in the window.

One of the operations that the user may select is to create a deliverable. For textual deliverables, the user is able to enter text in a text editor which contains the standard options to cut, copy and paste text as well as repeat and undo the last text operation performed. Upon saving this text, the user is returned to the deliverable window with the newly created text placed in the read only text box. For graphical deliverables, the user is presented with a drawing editor. He may then select graphical objects such as lines, boxes and arrows and place them on a drawing canvas. This diagram is then saved under a name input by the user.

Four of the documents are created by using more than the standard text editor. The first is the Actor Glossary. The user is able to enter a name and definition for each actor. This list is sequentially numbered and then saved. The second document is the Scenario List. The user is presented with a standard text editor but with the additional option to obtain a definition for an actor as he is preparing each of the scenarios. The third document is the numerical User Requirements Specification. The user is able to enter a requirement and have it associated with the next available sequential number beginning at one. In addition, he will be able to prefix and postfix the requirements with some text that can be entered into a text editor. The numerical Subsystem Requirements Specifications are able to be prepared in the same fashion. The additional feature is provided that the user may select a

requirement from the numerical User Requirements Specification and have it copied into the numerical Subsystem Requirements Specification. The user, of course, may elect to create a non-numeric User Requirements Specification and non-numeric Subsystem Requirements Specification. The last special document is the Test Report. For this deliverable, the user may enter a test case and at the same time view the user requirements from the numerical User Requirements Specification. He may also denote that the test case corresponds to a particular user requirement.

Another operation is editing a deliverable. To alter the document's contents, the user is presented with the document in the text editor. Once altered and saved, the new text is displayed in the read only text field in the deliverable window. If the document the user selects is one of the special documents listed above, the user is presented with the same window used to create the text. If the deliverable is graphical then the user is presented with a graphical editor in which he can alter the placement of the graphical components. Each time the user edits a deliverable, a new version is created and assigned the next sequential version number. In this way, the user may obtain any past version of a deliverable in the system.

One other important operation is deleting a deliverable. Should the deliverable have links from other deliverables in the system, they will be presented to the user in a list. By performing the actual deletion, the deliverable will have all of it's links removed. If the deliverable being deleted is a hierarchical diagram then the user will be displayed a list of diagrams that are assigned to one of the classes in the deliverable. If the deliverable is

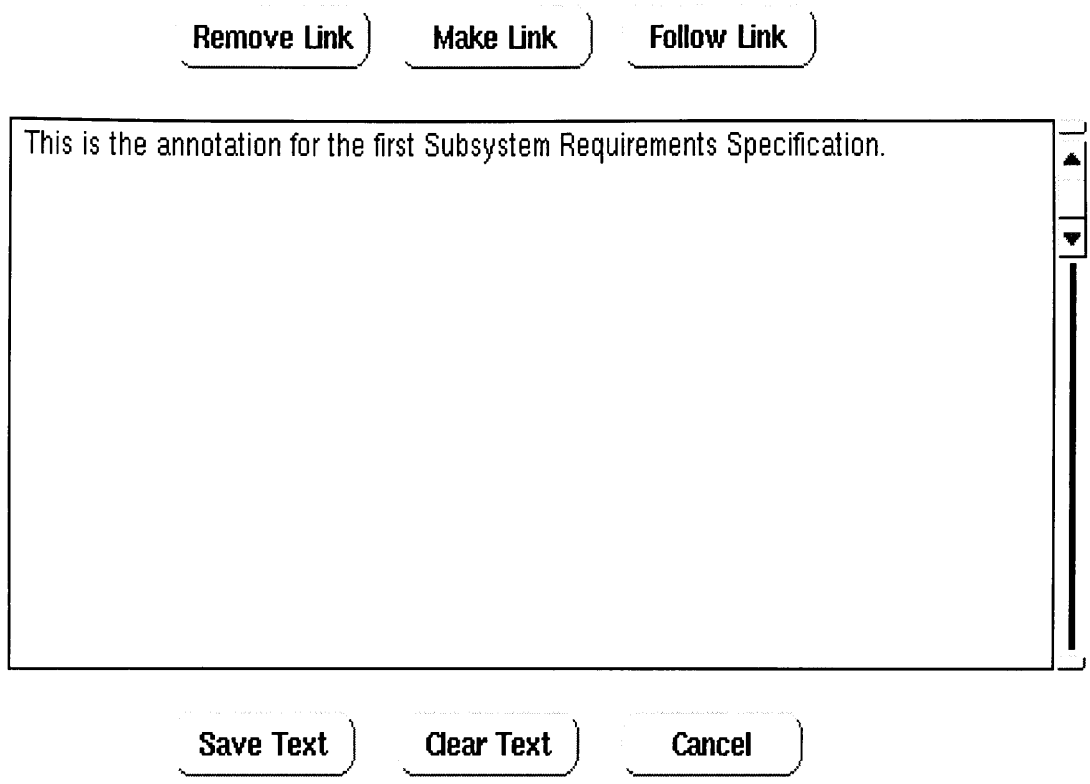


Figure 11: The annotation window

removed then the diagrams will no longer be assigned to the class in the deliverable. If the deliverable to be deleted has no dependencies then the user will only be prompted to verify it's removal.

The last essential operation is annotation. When selecting this option, the window in figure 11 is displayed. If the user has read only permission for the deliverable then he is only able to view the annotation. Should the user have read and write permission then he may perform additional operations. One of these operations is creating the annotation. To

accomplish this, the user must enter some text in the text editor and press the **Save Text** button. To edit the annotation, the user needs to follow these same sequence of steps. If at any time the user must clear the contents of the text editor, he may press the **Clear Text** button. By pressing the **Save Text** button, the empty document that was just created is saved.

One portion of the annotation window are the options to make, remove and follow links. Every user with read only or read and write permission is able to follow a link that a deliverable has. However, only users with read and write permission are able to make and remove links. When the user invokes one of these operations, he is presented with a list of deliverables. If he is removing a link, the selected deliverable is deleted from the list. If he is adding a link then the selected deliverable is added to the list of links that may be followed.

5.4.7 Source Code

The user may implement the analysis and design by creating classes. The user must first select a product and release which then becomes the category for which the user may add and remove classes. When choosing to add a class, the user is provided with an editor in which he may enter the code. He then may compile the code thus retaining it as a class in the system. In addition, the user may add and remove methods for each class he has entered. Should the user decide to remove a class, he will be asked to confirm his deletion. At any point during the implementation, the user may browse through the class

hierarchy and view the source code. When compiling the source code, if an error occurs, the user is presented with a debugger in which he may step through each message passed between objects. As each object receives a message, he may inspect the values of the parameters as well as the instance variables.

5.4.8 Agenda Editor

Any user will be able to view the progress of a product's development or the results of developing a past product by using the agenda editor. This editor is displayed in figure 12 and presents an overall outline of the steps of MOSES. To arrive at the editor, the user must follow scenario 3 from Appendix 1, section 9. For each phase, namely planning, analysis, design and implementation, the total number of modifications made to the deliverables is displayed. To obtain a more detailed view of each phase, the user may press the **View Deliverable Statistics** button. For each deliverable, the number of modifications (versions) made as well as whether the deliverable is "signed off" or not is then presented. If each of the deliverables in a phase has been created and "signed off" then the user will be able to specify that the phase is complete. When a phase has been completed, no additional modifications to the deliverables are allowed. To modify a deliverable, the user will first have to denote the phase as being incomplete and then "sign on" the deliverable. When the developer has completed each of the phases, the product is considered to be complete and the manager should assign it an ending date.

5.5 Implementation

The implementation of this CASE tool was done on a Sun Sparc-10 workstation running the SunOS 4.1.3 version of Unix. The language used is Smalltalk-80 which is part of the VisualWorks development environment.

5.5.1 VisualWorks Smalltalk

Smalltalk is an object-oriented language in which the user is able to create classes and have them interact via message passing [GR83]. Each class may have methods on both the class side and the instance side. When the user accepts a method the code is compiled and is made available to all classes in the system. Classes also may have both class variables and instance variables. The difference is that class variables may be accessed by any instance of the class whereas instance variables may only be accessed by the individual object.

The version of Smalltalk that is packaged with VisualWorks already has approximately three hundred classes predefined for the user to access. Examples of these classes include graphical support classes, data structures and data types. All of these classes are located in what is known as a Smalltalk image. This image records the state of the classes and changes that are made to them. Each time the user saves the environment, the image is updated to reflect all of the classes that were added, deleted or altered since the last update. Therefore, when the user re-enters the environment, the state that he left it in is returned.

VisualWorks is a complete development environment based on Smalltalk. This allows the user to easily integrate graphical user interface development with the model supporting

it. Provided to the user are tools such as the canvas editor, a menu editor, a color tool and a position tool. The canvas editor allows the user to select graphical components such as push buttons and labels and place them on a drawing canvas. This canvas is then installed on a class so that it can be opened by the user. When the canvas is still opened for editing, the user may utilize the color tool to alter the colors of the background and the foreground. In addition, the components of the window can be aligned with one another as well as relatively placed in the window by using the position tool. The other tool, the menu editor provides a simple way to construct a menu which may then be attached to a menu button.

To browse through both the user classes and system classes there is a class browser. This tool allows the user to select a class and browse through it's methods and variables. Since the methods in a class are divided into categories, the user must first select a category before viewing a method. The browser may be opened on all classes in the system or on a particular class and it's hierarchy. While in the browser, the user may add classes as well as methods and variables.

5.5.2 Windows

The windows the users interact with in this CASE tool were developed by using the canvas editor from the VisualWorks environment. This graphically-oriented component allows the user to drag window components such as input fields, text boxes, radio buttons and labels from a palette to a drawing canvas. Here the user may arrange the components and define their properties. Each of the components have either aspects or actions associated with

them. An aspect is a value holder for an attribute of the object. An action is a method that is invoked when the component is activated. For example, when a button is pressed a corresponding action is invoked. When a component with an aspect is modified, such as a radio button changing state, the corresponding value of the attribute is changed.

When the user has completed building the window, he installs the canvas on an application class that he names. He may also select the class to be the parent of the new class. Typically, the ApplicationModel class is selected since it has many of the actions that windows need such as the method **closeRequest** which closes the specified window. The user may also select the method name that holds the geographical layout. Typically, this is called **windowSpec**. The window is then opened by sending the **open** method to the class. If the class is entitled DeliverableWindow then the message DeliverableWindow **open** will open the window on the screen.

5.5.3 User Management

All user account information is located in the User class. This class has instance variables, **name, password, access** and **groupList** to record the user name, his password, the status of the account and a list of development groups that the user belongs to. In addition there is a class variable **UserList** to record all instances of the User class. The instances of User class are passive, meaning their sole purpose is to provide information to the other objects in the system. An example of this is when a user logs into the system. First, the user is presented with two input fields in which he may enter his user name and password.

After the developer types in his user name, the message **nameExists:** is sent to the User class. If the user name is in the system, the User class responds with true, otherwise it responds with false. If the user is in the system, the user is allowed to enter the password. This time the message, **getPassword:** is sent from the LoginWindow to the User class requesting the password for the provided user name. At no time does the User class need to request information from any other object in the system.

The User class contains class methods to add a user to the **UserList**, to remove a user from the **UserList** and to search for a user in the **UserList**. In addition, instance methods exist to set and retrieve the values of the instance variables for each class instance. When the list is searched, a private method is used, **find:** which attempts to locate the object with the desired user name in the **UserList**. Once the object is located, an instance method is used to retrieve the value of the desired instance variable. By convention, this method is given the same name as the instance variable.

5.5.4 Product Management

There are two classes involved in managing products and their releases. The first class, Product, has instance variables **productName** and **startDate**. The Release class has instance variables **startDate, endDate, manager, productName** and **releaseNumber**. Both of these classes have a class variable that holds all instances of the class. The user is able to add class instances to this list.

When the user starts a new product, he inputs the starting date, product name, starting

release number and project manager name into the input fields displayed by the class StartProduct. This class, upon receipt of the information, sends the message **addProduct:for:with:startingOn** to Product with the information from the input fields. The code for this method is shown in Figure 13. The first few sections of this code perform error checking. If an error is found then false is returned. If the parameter values are acceptable then an instance of Product is created and instance methods are invoked to set the instance variable's values. This object is then added to the OrderedCollection, **ProductList**. In addition, the message **addRelease:for:with:StartingOn** is sent to the class Release in order to start a new release. The Release class then follows the same sequence of steps that Product does. It creates an instance of itself, sets the values of the instance variables and adds the object to the class list, **ReleaseList**.

When the user wishes to delete a product, the DeleteProduct class displays a list of products in the system by sending the Product class the message **getNames**. This method merely iterates over the **ProductList** and for each object, adds the value of the instance variable **productName** to a new OrderedCollection which is then returned. The name selected by the user is then sent to Product by sending the message **removeFromList:**. This message searches the **ProductList** for the given name. When the object is located, it is removed from the OrderedCollection, thus removing it from the system.

The user may also start a release for a created product. The user enters the information into the input fields displayed by the class StartRelease. This information is then sent to the Release class via the message **addRelease:for:with:StartingOn**. The Release class

```
addProduct: aProductName for: aReleaseNumber with: aProjectManager
startingOn: aDate

" Add a product to the system.  When a product is added, a new release must be
started as well"

"Check for mistakes in the input information"
(Product nameExists: aProductName)
        ifTrue: [ DialogView warn: 'The product ', aProductName, 'already
                exists'.
                ^false ].
(aProductName size = 0)
        ifTrue: [ DialogView warn: 'You must enter a product name'.
                ^false ].
(aReleaseNumber size = 0)
        ifTrue: [ DialogView warn: 'You must enter a release number'.
                ^false ].
(aProjectManager size = 0)
        ifTrue: [ DialogView warm: 'You must enter a name of a manager'.
                ^false ].
(User nameExists: aProjectManager)
        ifFalse: [ DialogView warn: 'You need to add ', aProjectManager, ' to the
                user list'.
                AddUser openWith: aProjectManager.
                ^false ].

" The input information is valid so add the new product to the class list"
ProductList add: ( (self new)
                setProductName: aProductName;
                setStartDate: aDate ).

"Start a new release for this product"
Release addRelease: aReleaseNumber for: aProductName
                with: aProjectManager startingOn: aDate.
^true
```

Figure 13: Adding a product to the system

responds by checking to make sure that the release number sent in the parameter is greater than the last existing release number for the product. If it is not then a message is sent to

the Dialog class to display a warning message. If it is then an instance of Release is created with the instance variables set to the parameters that were supplied. This object is then added to the class list, **ReleaseList**.

5.5.5 Subsystem Management

All subsystem information is stored in the Subsystem class. The only attributes that need to be recorded are the name and the product and release that the subsystem is for. Correspondingly, there are six instance methods **setName, setProduct, setRelease, name, product** and **release** that set and return the values of these instance variables.Like previously discussed classes, there is a class variable that records the instances of Subsystem. This variable is named **SubsystemList**.

This class has class methods **addSubsystem:forProduct:withRelease:** and **removeSubsystem:forProduct:withRelease:** which add and remove objects from the **Subsystem List**. To determine whether a subsystem exists already, the method **subsystemExists:forProduct:withRelease:** is used. The code for this method is displayed in figure 14. This method finds all occurrences of Subsystem objects for a provided product name and places them in an OrderedCollection. The list is then refined by searching for objects having the provided release number. This list is then searched for an object with the given name. The list is then tested to determine if it's size is greater than zero. If it is then true is returned, signifying the subsystem does exist. Otherwise, false is returned.

```
subsystemExists: aName forProduct: aProductName
withRelease: aReleaseNumber

" Examine the list of subsystems and see whether a specified subsystem exists
or not"

| productList releaseList nameList |
productList := SubsystemList select: [:each | each product = aProductName ].
releaseList := productList select: [:each | each release = aReleaseNumber ].
nameList := releaseList select: [:each name = aName ].

"nameList should contain the subsystem we are looking for.  If it was not found,
the list will be empty"

nameList size > 0
        ifTrue: [ ^true ]
        ifFalse: [ ^false ]
```

Figure 14: Testing the existence of a subsystem

A subsystem name can be altered once an object exists in **SubsystemList** by using the method **setSubsystemName:to:forProduct:withRelease:**. This method adds a new Subsystem class instance to the class list. If the **addSubsystem:forProduct:withRelease:** method returns true, meaning a duplicate name was not added, then the Subsystem object with the old name is removed from the list. If the user attempts to change a subsystem name to the same name, then the **addSubsystem:forProduct:withRelease:** method would return false and the user would receive an instance of the Dialog class warning him of his error.

5.5.6 Deliverable Management

There are two inheritance hierarchies that handle all activities concerning the MOSES

deliverables and their versions. The first hierarchy is displayed in figure 15. The Deliverable class is at the top of the hierarchy and has two subclasses, SingleDeliverable and MultipleDeliverable. In addition, MultipleDeliverable has a subclass HierarchicalDiagram. The SingleDeliverable class is used for deliverables that can only have one instance for each product, such as the User Requirements Specification. In contrast, the MultipleDeliverable class is used for deliverables such as the Iteration Plan for which there can be many per product. The HierachicalDiagram class is for Analysis Diagrams and Design Diagrams only, since they can be assigned to a class in another diagram of the same type.

The Deliverable class contains the instance variables, **deliverableType**, **productName, releaseNumber** and **signedStatus**. The **signedStatus** variable corresponds to whether a deliverable has been signed off or not. There also is a class variable **DeliverableList** which serves the same purpose as the class lists previously discussed.

The Deliverable class contains a category of methods entitled testing. These methods determine the type of deliverable received as a parameter. For example, i**sDiagram:** returns true if the deliverable is a diagram, otherwise it returns false. These tests need to be performed so that messages can be directed to the appropriate subclass of Version. For example, if a deliverable is a diagram then messages should be directed towards the DiagramVersion class. Version and it's subclasses will be discussed after the Deliverable hierarchy. There are also messages to test whether a deliverable is a single deliverable, a

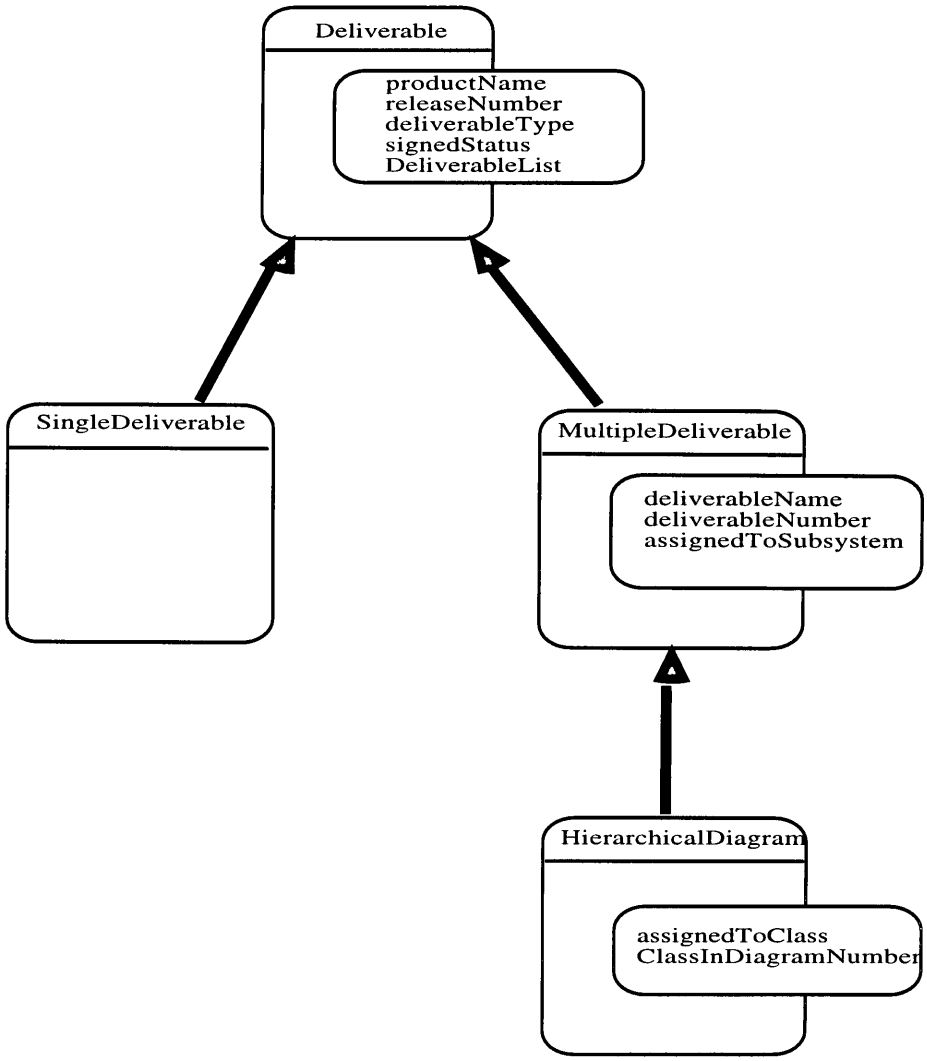


Figure 15: The Deliverable class hierarchy

document, a multiple deliverable, a hierarchical diagram or a numeric requirements specification.

There also exists a method **deliverableExists:forProduct: withRelease:** which

uses the private method **findDeliverable:forProduct:withRelease:**. If the deliverable is found then the method returns true meaning that the deliverable does indeed exist. Otherwise, it returns false. The private method **findDeliverable: forProduct:withRelease:** searches the class list for objects with the provided **productName** and places the results in a new instance of OrderedCollection. This list is then searched for objects with the provided **releaseNumbe**r. These objects are then placed in a list which is searched for an object with the appropriate **deliverableType**. If one is found, an OrderedCollection of one element is returned. The one element is the object corresponding to the search parameters. If the object is not found, a new instance of OrderedCollection is returned.

The MultipleDeliverable class has three additional instance variables, **deliverableName, deliverableNumber** and **assignToSubsystem** which are added to the variables that are inherited from the Deliverable class. Since multiple deliverables, by definition, are uniquely identified by a number as well as as the type, the methods used to search the class list need to search according to one additional parameter.

This class provides class methods to locate an object in the class list and access it's instance variables. Examples are **getSignedStatusFor:forProduct:withRelease: withNumber** and **getSubsystemFor:forProduct:withRelease:withNumber**. These methods simply search the class list for the appropriate object and return the value of the sought after instance variable. There also are class methods to set the values of the instance variables, **assignToSubsystem:forDeliverable:forProduct:withRelease:**

withNumber and **setSignedStatusTo:forDeliverable:forProduct:withRelease: withNumber**.

The methods to start a multiple deliverable follow the same basic idea as previously discussed classes. The code for **startDeliverable:forProduct:withRelease: withName:withNumber:withText** is displayed in figure 16. This method starts a textual document. First, an instance of Multiple Deliverable is created. This object then has it's instance variables set and is added to the class list. However, unlike previous start methods, one additional step is necessary. The deliverable type is tested to determine which subclass of Version to send the start message to. Since the actual text of a document is stored in a version, the text does not get stored in the MultipleDeliverable class. The text is sent to the appropriate subclass of Version. For diagrams, the information is stored in the window specification for a class. This window specification is sent to the DiagramVersion class.

The Multiple Deliverable class also has methods to remove deliverables. The method **removeDeliverable:forProduct:withRelease:withNumber:** is similar to the message to start a deliverable except the method **removeVersionsFor:forProduct: withRelease:withNumber:** is sent to the appropriate subclass of Version. First, however, a check is made to make sure the deliverable does exist. Also, the message **removeFromListFor:forProduct:withRelease:withNumber:** is sent to itself to remove the MultipleDeliverable object from the class list. This method simply searches the class list to locate the appropriate object. It is then removed from the list by using the

```
startDeliverable: aDeliverableType forProduct: aProductName withRelease:
aReleaseNumber withName: aName withNumber: aNumber withText: aText

"Add a new deliverable to the system.  Create an instance of the class, set the
attributes' values and add the object to the class list"
DeliverableList add: ( (self new)
        setDeliverableType: aDeliverableType;
        setProductName: aProductName;
        setReleaseNumber: aReleaseNumber;
        setDeliverableName: aName;
        setDeliverableNumber: aNumber;
        setSignedStatus: false ).

"Determine which subclass of Version to send the version creation message
to"

(aDeliverableType = 'Numeric Subsystem Requirements Specification')
        ifTrue: [ SubsystemRS startVersionFor: 'Numeric Subsystem
                        Requirements Specification' forProduct: aProductName
                        withRelease: aReleaseNumber withNumber: aNumber
                        withName: aName.
                ^self ].
(aDeliverableType = 'Test Report')
        ifTrue: [TestReportVersion startVersionFor: 'Test Report'
                        forProduct: aProductName
                        withRelease: aReleaseNumber withNumber: aNumber
                        withName: aName.
                ^self ].
(self isDocument: aDeliverableType)
        ifTrue: [ DocumentVersion startVersionFor: aDeliverableType
                        forProduct: aProductName withRelease: aReleaseNumber
                        withNumber: aNumber withName: aName withText: aText.
                 ^self ].
```

Figure 16: Starting a new deliverable

message **removeIfAbsent:**.

There are also a few private methods for the MultipleDeliverable class. The first method is **getNumbersFor:forProduct:withRelease:**. The code for this method is shown in

figure 17. This method searches the **DeliverableList** to find all objects of a particular deliverable type for a given product and release. These objects are placed in an OrderedCollection which is then iterated over. Each number is extracted and placed in a new OrderedCollection which is then returned. The other method is **getNameFor:forProduct:withRelease:withNumber:**. This method simply locates the appropriate object in the **DeliverableList** and returns the value of **deliverableName**. If the deliverable does not exist then false is returned.

```
getNumbersFor: aDeliverableType forProduct: aProductName
withRelease: aReleaseNumber

"Retrieve all of the deliverable numbers of a single type of deliverable.
For example, retrieve all analysis diagram number is the system"

| productList releaseList numberList |
numberList := OrderedCollection new.
(productList := DeliverableList select: [:each | each productName =
aProductName]) size > 0
        ifFalse: [ ^OrderedCollection new ]
        ifTrue: [ (releaseList := productList select: [:each | each releaseNumber
= aReleaseNumber] ) size > 0
                ifFalse: [ ^OrderedCollection new ]
                ifTrue: [ (deliverableList := releaseList select: [:each | each
deliverableType = aDeliverableType]) size > 0
                        ifFalse: [ ^OrderedCollection new ]
                        ifTrue: [ (deliverableList do: [:each | numberList add: (each
deliverableNumber) ]) size > 0
                                ifFalse: [ ^OrderedCollection new ]
                                ifTrue: [ ^numberList ] ] ] ]
```

Figure 17: Retrieving all deliverable numbers for a specific type

The HierarchicalDiagram class inherits the majority of it's methods from MultipleDeliverable. It provides the additional instance variables **assignedToClass** and **classInDiagramNumber**. These record the class name and diagram number that the diagram is assigned to. To access and set these variables, there are three methods. The first, **assignToClass:inDiagram:forDeliverable:forProduct:withRelease: withNumber:**, finds the object in the class list and uses the instance methods **setClass:** and **setDiagram:** to set the instance variables. The second method, **getClassDiagramInFor:forProduct:withRelease:withNumber:** finds the object in the list and retrieves the value of the instance variable **classInDiagramNumber**. The last method **getClassSignedToFor:forProduct:withRelease:withNumber:** does the same thing except that it returns the value of the variable **assignedToClass**.

The last subclass of Deliverable is SingleDeliverable. This class has no additional variables. The methods implemented in this class provide a functionality similar to the Deliverable class. There are methods to start deliverables and remove deliverables. In addition, there are methods to obtain the values of the inherited instance variables.

Every deliverable' s text or window specification is actually stored in a subclass of Version. The instance variables for Version are **author, date, releaseNumber, productName, deliverableType** and **versionNumber**. As for previously discussed classes there is a class list variable, **VersionList**, that stores all the instances of the class. There are instance methods to set and retrieve the values of these instance variables as well. There are also class methods to start a version, remove all versions, test if a version exists,

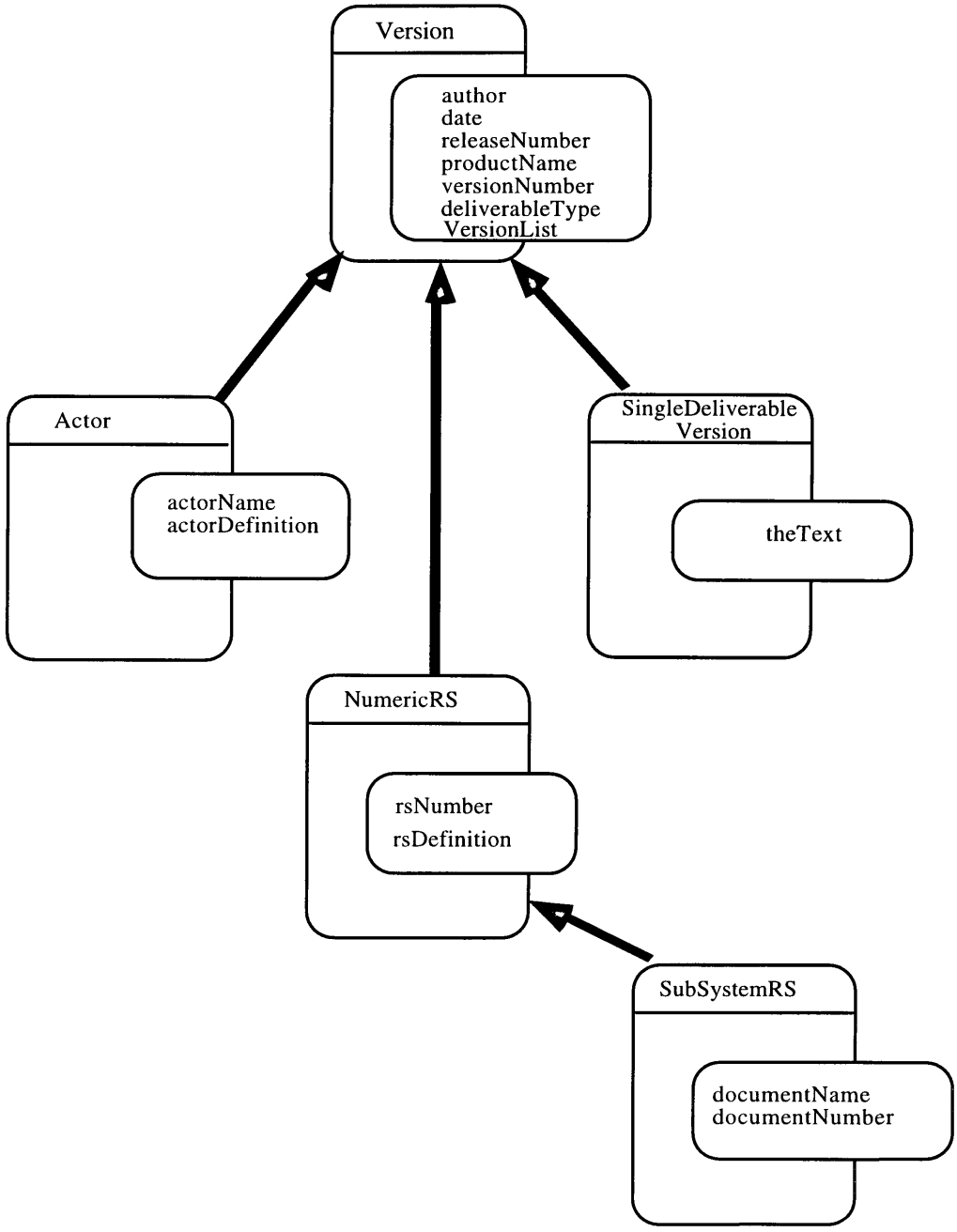


Figure 18a: The Version class hierarchy

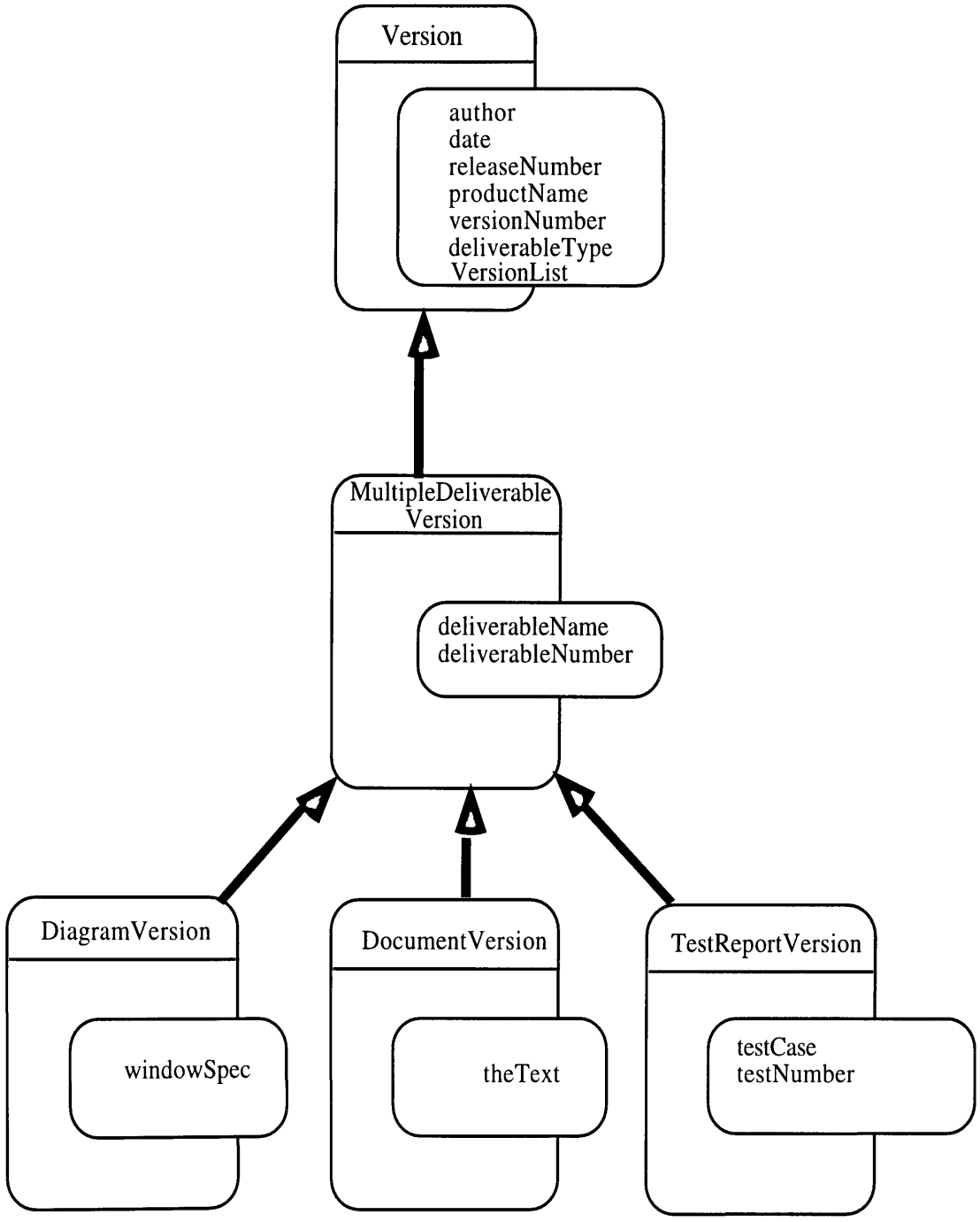


Figure 18b: The MultipleDeliverableVersion class hierarchy

get the last version number and to find a particular version. These methods are very similar to the methods of previously discussed classes that perform these functions. However, there is no method to remove a single version. There is the method **removeVersionsFor:forProduct:withRelease:** though, which removes all versions for a particular deliverable.

There are many subclasses of Version. The inheritance hierarchy is shown in both figures 18a and 18b. MultipleDeliverableVersion class contains the instance variables **deliverableName** and **deliverableNumber**. This class has no instances but serves as a template for it's subclasses and provides a variety of methods. The DiagramVersion class is one of it's subclasses. This class holds information pertaining to all diagram deliverables. The instance variable **windowSpec** contains the actual diagram created with the drawing editor. The other subclass is DocumentVersion. This corresponds to any deliverable that is a MultipleDeliverable and is textual such as Iteration Plans, Subsystem Requirements Specifications and Review Reports. The last subclass is TestReportVersion which contains information concerning Test Reports. It has instance variables **testNumber** and **testCase**. All of these classes provide methods to add instances to the class list and to remove all versions.

Another subclass that inherits from Version is SingleDeliverableVersion. This class corresponds to documents that are single deliverables. The instance variable, **theText** is provided to hold the actual text of the document. Another subclass is Actor. This class corresponds to the Actor Glossary. Two additional instance variables are defined,

actorName and **actorDefinition** which hold the actor's name and definition. Additional methods are provided to set and retrieve the values of these variables.

The last subclass to inherit from Version is NumericRS. This class corresponds to the numeric User Requirements Specification. There is one subclass which inherits from this class, SubsystemRS. This class corresponds to the numeric Subsystem Requirements Specification. It defines two instance variables **deliverableName** and **deliverableNumber**. This is due to the fact that there can be many occurrences of this deliverable per product. Were Smalltalk to have multiple inheritance capabilities, this class would not need to define these variables. Instead it would inherit from both NumericRS and MultipleDeliverableVersion.

5.5.7 Annotation Management

The class Annotation contains information regarding the annotation and the links that are associated with each deliverable. There are instance variables **annotation**, which contains the actual annotation text, **deliverable, product, release, number** and **deliverableLinks**, which is an OrderedCollection of links to other deliverables. There are also instance methods to set and retrieve the values of these variables. In addition, there is a class variable **AnnotationList** holding all instances of the Annotation class.

Similar to the other classes, there are class methods to start and remove an annotation for a deliverable. The method to start an annotation is **setAnnotationTo: forDeliverable:forProduct:withRelease:**. The code for this method is shown in

figure 19. Since the user may elect to save an annotation even if there is no text in the text editor, starting an annotation is treated like setting an annotation's text to a particular value. First, a check is made to see if an annotation already exists for the deliverable by invoking the private method **annotationExistsFor:forDeliverable:forProduct: withRelease:**. If this returns true then the annotation already exists in the **AnnotationList**. In this case, the **AnnotationList** must be searched for the correct object by using the method **findAnnotationFor:forDeliverable: forProduct:withRelease:**. This object then has the instance method **setAnnotation:** invoked to set the annotation to the new value. If the annotation does not exist then the

```
setAnnotationTo: anAnnotation forDeliverable: aDeliverableType
forProduct: aProductName withRelease: aReleaseNumber

"Create an annotation if one doesn't exist, otherwise set the old
annotation to the new annotation"

(Annotation annotationExistsFor: aDeliverableType forProduct:
aProductName withRelease: aReleaseNumber)
        ifTrue: [ (Annotation findAnnotationFor: aDeliverableType
forProduct: aProductName withRelease: aReleaseNumber)
                setAnnotation: anAnnotation ]
        ifFalse: [ AnnotationList add: ( (self new)
                        setDeliverable: aDeliverableType;
                        setProduct: aProductName;
                        setRelease: aReleaseNumber;
                        setDeliverableLinks: OrderedCollection new;
                        setAnnotation: anAnnotation) ]
```

Figure 19: Setting the annotation

AnnotationList has the new annotation appended. First, though, **deliverableLinks** is

set to a new instance of OrderedCollection.

Another set of methods relate to creating links for a deliverable. The method for this is **AddLink:forDeliverable:forProduct:withRelease:**. If the annotation already exists then an instance of the class DeliverableLink is added to the instance variable **deliverableLinks**. This class only contains two instance variables, **deliverableNumber** and **deliverableType**. It also has instance methods to set and retrieve the values of these variables. If the annotation does not exist then an instance of the OrderedCollection class is created. The new annotation is then added to the **AnnotationList**. When a link needs to be removed, the method **removeLink:forDeliverable:forProduct:withRelease** is invoked. This method first finds the appropriate annotation in the **AnnotationList.** If the annotation is found then **deliverableLinks** is searched for the object corresponding to the provided **deliverableNumber**. Once found, the object is removed from **deliverableLinks**.

5.5.8 Window and Model Class Interaction

The classes implemented for this tool are of two types, the model classes and the window classes. The window classes receive input from the user and pass that information to the model classes. They also receive messages from the model classes and display the output in the window components. A simple example is the IdentifySubsystemWindow. It displays an input field in which the user may input the name of the subsystem. The aspect that this is stored in is **subsystemName**. When the user presses the apply action button,

the method **applyPushed** is invoked. This method sends the message **addSubsystem:forProduct:withRelease:** to the Subsystem class with the value of the aspect. If the cancel action button is pressed instead, the message **closeRequest** is sent to self which in this case is the IdentifySubsystemWindow class. This is an inherited method from the ApplicationModel class whose effect is to close the specified window.

6 Conclusion

6.1 Summary

This CASE tool provides the user with top-level guidance throughout the entire software development process. By following the phases and activities outlined by MOSES, the developer is able to gauge both the amount of work completed and the amount of work still needed. To develop the necessary deliverables, editors both graphical and textual, are provided to the user.

The methodology utilized by this tool is both flexible yet demanding. The user is able to step through the development process in any sequence that he sees fit. However, he is encouraged to first plan then analyze, design and implement. This flexibility is a direct effect of using the fountain lifecycle model. Instead of requiring the user to follow a rigid set of steps like the waterfall model does, the developer may perform a little of each step and then return to previous phases. This allows for more robust software products since the developer may first design and implement a piece of the product or a prototype. The

software user may then comment and produce feedback, thus allowing the developer to take into account any changes or mistakes.

At the same time, MOSES demands that specific deliverables need to be produced in order to complete a product. For every phase of development, the user must perform certain activities whose end product results in a deliverable. The deliverables are both textual and graphical in nature. Since each phase requires the user to produce specified deliverables, he can easily judge which phases need additional work. When all phases are complete and all deliverables produced, the developer can be assured that the product is in a final state.

The CASE tool in this thesis allows the user to take advantage of these benefits of MOSES. Editors as well as supporting features such as deliverable versioning and security are provided. As a means of viewing the effort put into product development, the developer may use the agenda editor. In order to follow an idea from inception to implementation, the user may link deliverables to one another as well as annotate them. These are only a few of the features that allow the user to proceed through the entire product lifecycle while maintaining a clear notion of where he stands in the development process.

6.2 Future Work and Enhancements

To extend the capabilities of this CASE tool, a shared object-oriented database needs to be added. This would allow multiple developers to simultaneously operate on different

portions of a project thus realizing MOSES' objective of having multiple development teams. In it's current state, this environment will allow only one user at a time to operate on a product. However, since the security feature and development groups already exist, this CASE tool could easily be extended for distributed use.

The other feature that would make this environment more useful is the addition of a drawing editor with expanded functionality. Such an editor has been developed by Henderson-Sellers himself. He provides the capability to construct diagrams specifically using the notation that MOSES advocates.

Combining these two features with the current capabilities would make this tool more suitable for commercial environments.

References

[CRS92] Casais, Eduardo, Michael Ranft, Bernhard Schiefer, Dietmar Theobald and Walter Zimmer. STONE. *FZI Technical Report FZI.039.1*, 1992.

[EH93] Edwards, J.M. and Brian Henderson-Sellers. A graphical notation for analysis and design. In *Journal of Object-Oriented Programming*, 5(9): 53-74, 1993.

[EHe93] Edwards, J.M. and Brian Henderson-Sellers. Application of an Object-Oriented Analysis and Design Methodology to Engineering Cost Management. In *Journal of Systems Software*, 23: 123-128, 1993.

[GR83] Goldberg, Adele and David Robinson. *Smalltalk-80. The Language and it's Implementation.* Addison-Wesley, Reading, MA, 1983.

[HE93] Henderson-Sellers, B. and J.M. Edwards. The O-O-O methodology for the object-oriented lifecycle. In *Software Engineering Notes*, 4: 54-63, 1993.

[Hen92] Henderson-Sellers, B. *A Book of Object-Oriented Knowledge*. Prentice-Hall, Sydney, 1992.

[Hen93] Henderson-Sellers, B. *MOSES (Methodology for Object-Oriented Software Engineering of Systems).* In TOOLS USA' 93, pages 561-571, 1993.

[Kad93] Kadia, R. Issues encountered in building a flexible software development environment. *University of Colorado Technical Report*, 1993.

[You94] Yourdon, Edward. *Object-Oriented Systems Design - an Integrated Approach.* Yourdon Press, 1994.

Appendix 1

1 Scenarios for operating on deliverables

Scenario 1

The user creates a deliverable. The date, author and version number one are recorded. The user then enters the text for a textual deliverable or constructs a diagram for a graphical deliverable.

Scenario 2

The user views the deliverable. By viewing a deliverable, the user is not able to alter the contents. The deliverable is displayed in a window.

Scenario 3

The user edits the deliverable. The selected deliverable is placed in the appropriate editor and presented to the user. The user may then perform one of the editor operations.

Scenario 4

The user deletes the deliverable. For safety precautions, the user is asked to verify the removal of the deliverable. If the user responds affirmatively, all existing versions of the deliverable will be removed from the system. Should the user respond negatively, no removal will take place.

Scenario 5

The user prints the document.

Scenario 6

The user annotates the deliverable. The user can create, edit, save or delete an annotation. Each deliverable may only have one annotation associated with it. By editing the annotation, the user alters the contents and either saves or discards the changes. The user can also choose to delete the annotation. By doing this no annotation will be associated with the document.

Scenario 7

The user links the deliverable to another existing deliverable in the system. The user selects the deliverable to which he will be linking. The link is then constructed.

Scenario 8

The user removes a link associated with the deliverable. The user selects one of the linked deliverables. This link is then removed.

Scenario 9

The user follows a link to a deliverable. He selects a link that has already been created. He is then placed in an environment in which he is able to perform operations on the deliverable that he selected.

Scenario 10

The user "signs off" a deliverable. This indicates that the deliverable has been completed and is ready for review. By default, the latest version is used as the "signed off" deliverable.

Scenario 11

The user "signs on" a deliverable. This indicates that additional operations may be performed on this deliverable.

Scenario 12

The user selects a previous version. All subsequent deliverable operations will be performed on this version.

Scenario 13

The user assigns the multiple deliverable to a subsystem. He then selects the subsystem to assign it to.

Scenario 14

The user assigns the multiple deliverable to no subsystem. The multiple deliverable is then denoted as being assigned to no subsystem.

Scenario 15

The user assigns the hierarchical diagram to a class in another hierarchical diagram of the same type. The user selects the diagram.

Scenario 16

The user assigns the hierarchical diagram to no class. The hierarchical diagram is then denoted as being assigned to no class.

2 Scenarios involving the Planner

The Planner enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Planner chooses a product and one of it's releases. Next, the Planner elects to operate on either the Business Planning Report or the Iteration Plan. The Planner then selects one of the deliverable operations. After the Planner chooses an operation, he either selects another document or diagram or exits the CASE tool.

3 Scenarios invloving the Requirement Gatherer

The Requirement Gatherer enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Requirement Gatherer chooses a product and one of it's releases. Next, the Requirement Gatherer selects a document or diagram.

Scenario 1

The Requirement Gatherer chooses to operate on the formal User Requirements Specification. He then chooses to generate a numerical list of requirements.

Scenario 1a

The Requirement Gatherer chooses to prefix the numerical list with some text. He then enters the text in the text editor and saves it. The formal User Requirements Specification now consists of the text followed by the numeric list.

Scenario 1b

The Requirement Gatherer chooses to postfix the numerical list with some text. He then enters the text in the text editor and saves it. The formal User Requirements Specification now consists of the numeric list followed by the text.

Scenario 1c

The Requirement Gatherer chooses to enter a numbered requirement. He then enters text which will be associated with the next available sequential number in the list.

Scenario 1d

The Requirement Gatherer chooses to edit or delete a numbered requirement. The Requirement Gatherer then selects a requirement number. The text associated with the chosen number is displayed and the Requirement Gatherer may either alter or delete it. If the requirement is deleted, the list will be re-numbered to retain a sequential ordering.

Scenario 1e

The Requirement Gatherer chooses any deliverable operation for the entire formal User Requirements Specification.

Scenario 2

The Requirement Gatherer chooses to operate on the formal User Requirements Specification. The Requirement Gatherer then selects a non-numerical list of requirements. The Requirement Gatherer chooses any deliverable operation.

Scenario 3

The Requirement Gatherer chooses to operate on the Subsystem Requirements Specification. He then chooses to generate a numerical list of requirements.

Scenario 3a

The Requirement Gatherer chooses to prefix the numerical list with some text. He then enters the text in the text editor and saves it. The Subsystem Requirements Specification

now consists of the document followed by the text.

Scenario 3b

The Requirement Gatherer chooses to postfix the numerical list with some text. He then enters the text in the text editor and saves it. The Subsystem Requirements Specification now consists of the numeric list followed by the text.

Scenario 3c

The Requirement Gatherer chooses to enter a numbered requirement. He then enters text which will be associated with the next available sequential number in the list.

Scenario 3d

The Requirement Gatherer chooses to edit or delete a numbered requirement. The Requirement Gatherer then selects a number. The text associated with the chosen number is displayed and the Requirement Gatherer may either alter or delete it. If the requirement is deleted, the list will be re-numbered to retain a sequential ordering.

Scenario 3e

The Requirement Gatherer chooses to retrieve a numbered requirement from the User Requirements Specification. He selects the number to retrieve and it is appended to the numerical list with the next available sequential number.

Scenario 3f

The Requirement Gatherer chooses any deliverable operation for the entire Subsystem Requirements Specification.

Scenario 4

The Requirement Gatherer chooses to operate on the Subsystem Requirements Specification. The Requirement Gatherer then selects a non-numerical list of requirements. He chooses any deliverable operation.

After the Requirement Gatherer chooses one of these scenarios, he either chooses another document or diagram or exits the CASE tool.

4 Senarios involving the Analyzer

The Analyzer enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Analyzer chooses a product and one of it's releases.

Scenario 1

The Analyzer chooses to perform subsystem management operations.

Scenario 1a

The Analyzer chooses to add a subsystem. The Analyzer enters the subsystem name.

Scenario 1b

The Analyzer chooses to delete a subsystem. He then selects the name of the subsystem. When asked to verify the deletion, the Analyzer responds affirmatively and the subsystem is removed. All documents, diagrams and source code modules associated with the subsystem will be associated with no subsystem.

Scenario 1c

The Analyzer chooses to change a subsystem name. The Analyzer selects a name to change and then enters the new name. All documents, diagrams and source code modules

associated with the previous name will now be associated with the new name.

Scenario 2

The Analyzer chooses to perform operations on analysis diagrams. He then selects an analysis diagram number.

Scenario 2a

The Analyzer chooses to create the analysis diagram. He then enters the class names, attributes, operations, relationships (aggregation, association and generalization) and cardinalities. The Analyzer then enters the diagram's name. A unique number is assigned to the diagram.

Scenario 2b

The Analyzer chooses to edit the analysis diagram. The Analyzer enters, deletes or alters the class names, attributes, operations, relationships (aggregation, association and generalization) and cardinalities.

Scenario 2c

The Analyzer chooses any deliverable operation other than edit or create.

Scenario 3

The Analyzer chooses to perform operations on the actor glossary. He then selects a deliverable operation.

Scenario 3a

The Analyzer chooses to create an Actor Glossary. He then enters the actor names and definitions.

Scenario 4

The Analyzer chooses to perform operations on the scenario list. He then selects a deliverable. He may elect to view the definition of each actor as he edits or creates the scenario list.

After the Analyzer chooses one of these scenarios, he either chooses another document or diagram or exits the CASE tool.

5 Scenarios involving the Designer

The Designer enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Designer chooses a product and one of it's releases. Next, the Designer selects a document or a diagram.

Scenario 1

The Designer chooses to perform operations on design diagrams. He then selects a design diagram number.

Scenario 1a

The Designer chooses to create the design diagram. He then enters the class names, relationships (client-server), cardinalities and services (public and private). The Designer then enters the diagram name. A unique number is assigned to the diagram.

Scenario 1b

The Designer chooses to edit the design diagram. The Designer enters, deletes or alters the class names, relationships (client-server), cardinalities and services (public and private).

Scenario 1c

The Designer chooses any deliverable operation other than edit or create.

Scenario 2

The Designer chooses to perform operations on inheritance diagrams. He then selects an inheritance diagram number.

Scenario 2a

The Designer chooses to create the inheritance diagram. He then specifies where to place classes in the inheritance diagram. The Designer then enters the diagram name. A unique number is assigned to the diagram.

Scenario 2b

The Designer chooses to edit the inheritance diagram. He then alters the positioning of classes in the inheritance diagram or deletes classes in the diagram.

Scenario 2c

The Designer chooses any deliverable operation other than edit or create.

Scenario 3

The Designer chooses to perform operations on contract diagrams. He then selects a contract diagram number.

Scenario 3a

The Designer chooses to create the contract diagram. He then enters the contract requirements, preconditions, post-conditions and the results of the contract. The Designer then enters the diagram name. A unique number is assigned to the diagram.

Scenario 3b

The Designer chooses to edit the contract diagram. He then enters, edits or deletes the contract requirements, preconditions, post-conditions and the results of the contract.

Scenario 3c

The Designer chooses any deliverable operation other than edit or create.

Scenario 4

The Designer chooses to perform operations on class interface diagrams. He then selects a class interface diagram number.

Scenario 4a

The Designer chooses to create the class interface diagram. He then enters the class names that the specified class interacts with, operations, local variables, the interface and class features. The Designer then enter the diagram name. A unique number is assigned to the diagram.

Scenario 4b

The Designer chooses to edit the class interface diagram. He then chooses to enter, delete or alter the class names that the specified class interacts with, operations, local variables, the interface and class features.

Scenario 4c

The Designer chooses any deliverable operation other than edit or create.

After the Designer chooses one of these scenarios, he either chooses another document or diagram or exits the CASE tool.

6 Scenarios involving the Programmer

The Programmer enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Programmer chooses a product and one of it's releases. Next, the Programmer chooses to operate on source code.

Scenario 1

The Programmer chooses to edit the source code. The Programmer then selects the class and alters the contents.

Scenario 2

The Programmer chooses to compile the source code. He then selects the class and compiles.

Scenario 3

The Programmer chooses to debug the source code. The Programmer selects the class and invokes the debugger.

Scenario 4

The Programmer selects a class to browse. The class and it's features are displayed on the screen. The Programmer then chooses to browse through a superclass or a subclass of the current class. He enters the name of this class and the features are displayed on the screen. After the Programmer chooses one of these scenarios, he either chooses another class or exits the CASE tool.

7 Scenarios involving the Quality Assurance Person

The Q&A person enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Q&A person chooses a product and one of it's releases. Next, the Q&A person selects a document or diagram.

Scenario 1

The Q&A person chooses to edit or create a Review Report. The Q&A person then selects the document or diagram that he is going to review. He enters his assessment of the quality of this document or diagram.

Scenario 2

The Q&A person chooses to edit or create a Test Report. The Q&A person then selects the class that he is going to test. He lists the test cases in the Test Report and records whether the class passed or failed the test.

Scenario 3

The Q&A person chooses a deliverable operation other than edit or create.

After the Q&A person chooses one of these scenarios, he either chooses another document or diagram or exits the CASE tool.

8 Scenarios involving the Project Secretary or Project Manager

The Project Secretary enters his user name, password and group that he wishes to log in as. If he has authorization to continue, the Project Secretary chooses a product and one of

it's releases.

Scenario 1

The Project Secretary chooses whether to set or unset the security feature. When the security feature is enabled, each user is prompted for his name and password upon entering the system.

Scenario 2

The Project Secretary chooses to perform operations on products and releases.

Scenario 2a

The Project Secretary chooses to start a product. He then enters the product name, start date, project manager's name and beginning release number.

Scenario 2b

The Project Secretary chooses to delete a product. He then enters the product name. Upon verification, all documents, diagrams and source code modules associated with the product are removed.

Scenario 2c

The Project Secretary chooses to start a new release for the product. He then enters the new release number, start date and project manager's name. The release number must be greater than the previous release number.

Scenario 2d

The Project Secretary chooses to delete the last release of a product. Upon verification, all associated documents, diagrams and source code modules are removed from the system.

Scenario 2e

The Project Secretary chooses to change the number of the latest release. He then enters the new number. This number cannot be less than or equal to the previous release number. All documents, diagrams and source code associated with the old release number will now be associated with the new number.

Scenario 2f

The Project Secretary chooses to alter the starting date for a product release. He then enters the new date.

Scenario 2g

The Project Secretary chooses to alter the project manager's name for a product release. He then enters the new name.

Scenario 2h

The Project Secretary chooses to enter an ending date for a product release. He then enters the ending date.

Scenario 3

The Project Secretary chooses to alter the user information for the security feature.

Scenario 3a

The Project Secretary chooses to start a user account. He then enters the user name, password, the groups the user is a part of and whether the account is enabled or disabled. The user name must not already exist in the system.

Scenario 3b

The Project Secretary chooses to remove a user from the system. He then enters the user name. Upon verification, the user is removed from the system and any groups that he may have belonged to.

Scenario 3c

The Project Secretary chooses to change a user's password. He then selects a user's name. Next, he enters the password for that user.

Scenario 3d

The Project Secretary chooses to assign a user to one or more groups. He first selects theuser name. Next, he selects the groups and assigns the user to them.

Scenario 3e

The Project Secretary chooses to enable or disable a user account. He first selects an account and then enables or disables it.

Scenario 3f

The Project Secretary chooses to change a user's name. He first selects the user and then enters the new name. All account information is now associated with the new user name.

Scenario 4

The Project Secretary chooses to set the permissions a group has for each of the deliverables. He first selects a group. Next, he selects a deliverable and assigns either read only, read and write or no permissions for this group.

After the Project Secretary chooses one of these scenarios, he either chooses another document or diagram or exits the CASE tool.

9 Scenarios involving All Users

Scenario 1

All users may view user account information. The user first selects a user account. He then is able to examine the groups the user is in and whether the account is enabled or disabled.

Scenario 2

All users may view product information. The user first selects a product. He may then display the start date, end date and project manager's name for each of the releases.

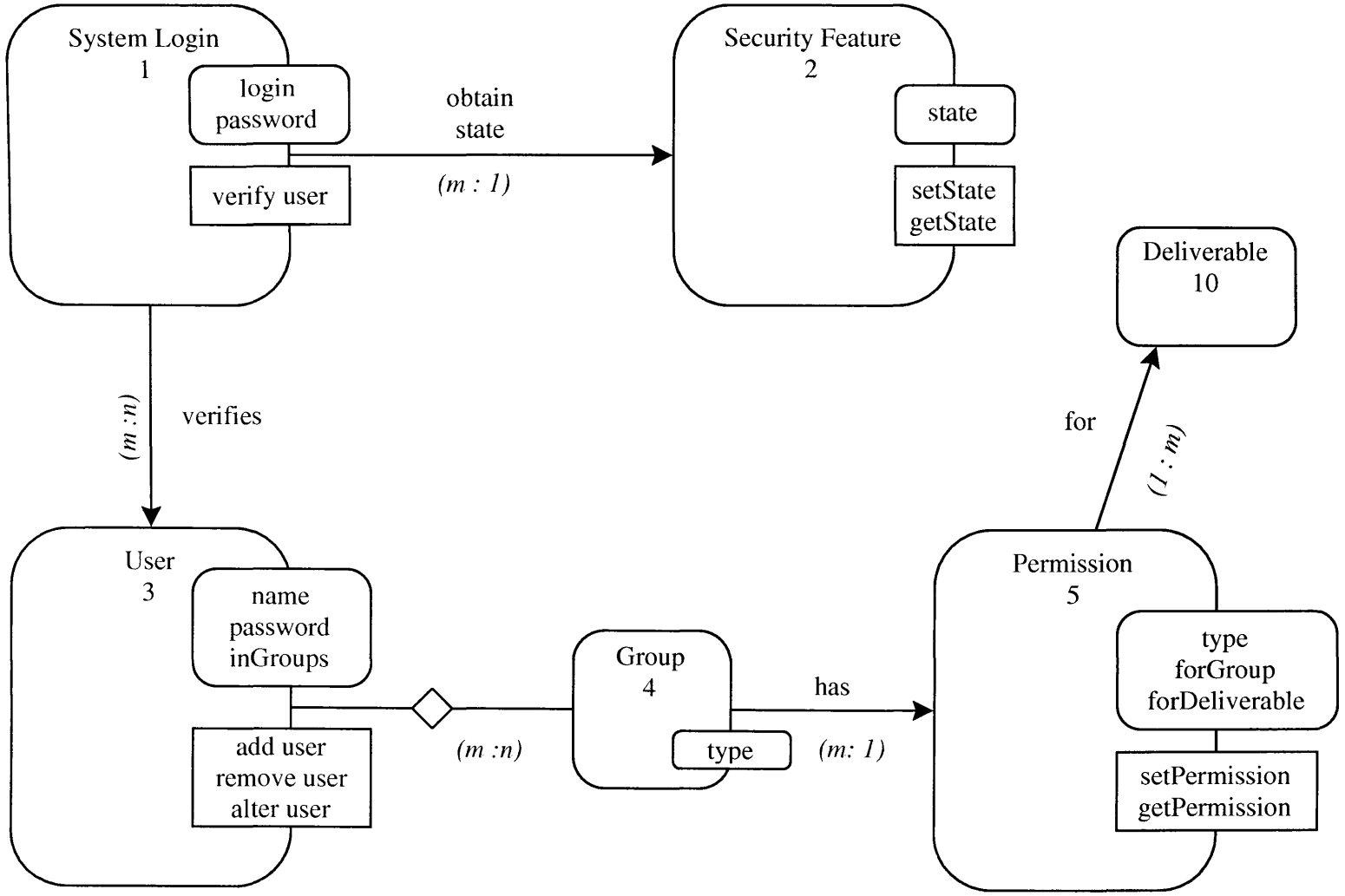
Scenario 3

All users may view the MOSES agenda. The user first selects a product and release. He may then display the number of modifications made to each deliverable, whether each deliverable is signed off and whether each phase is completed.

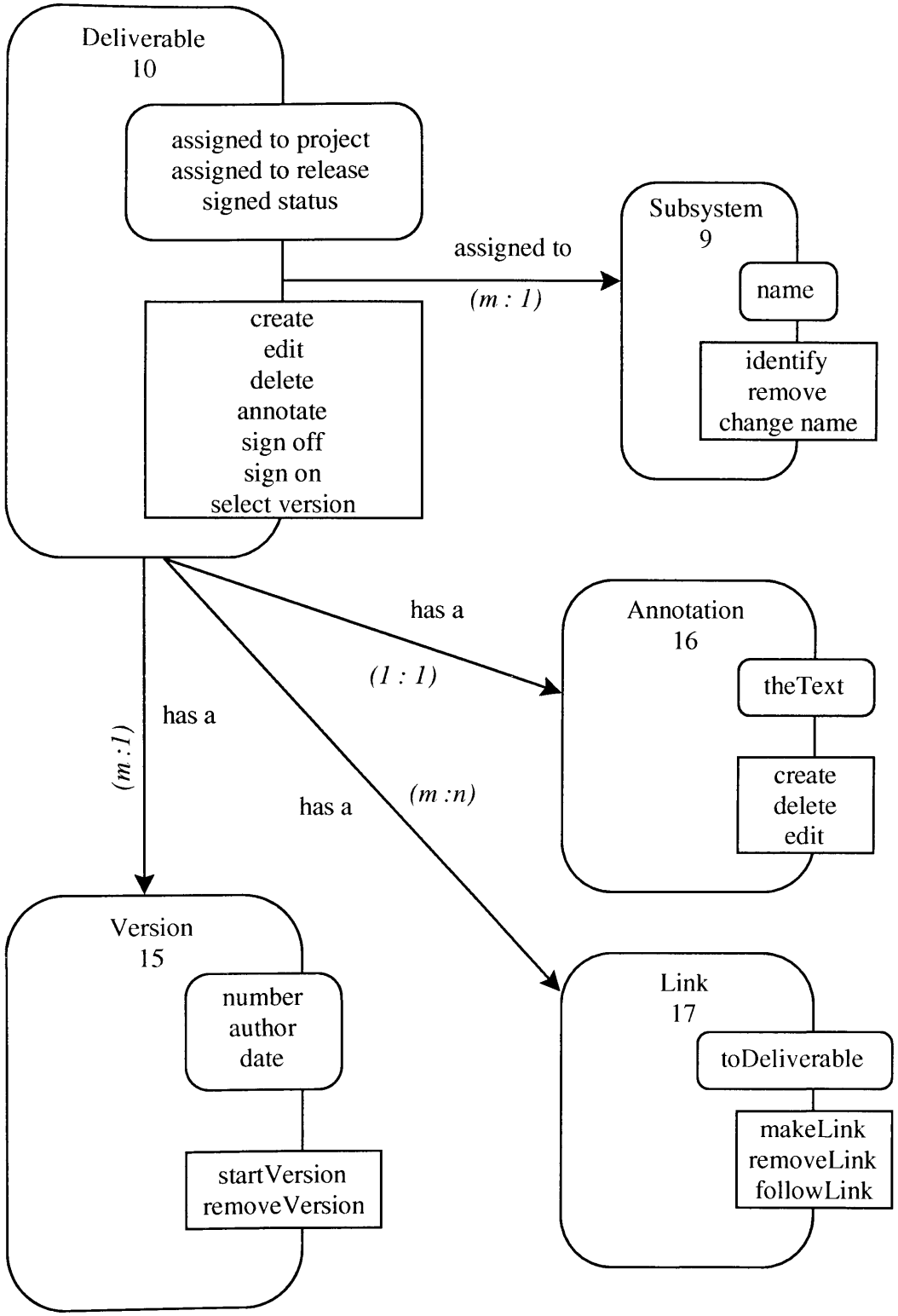
Appendix 2

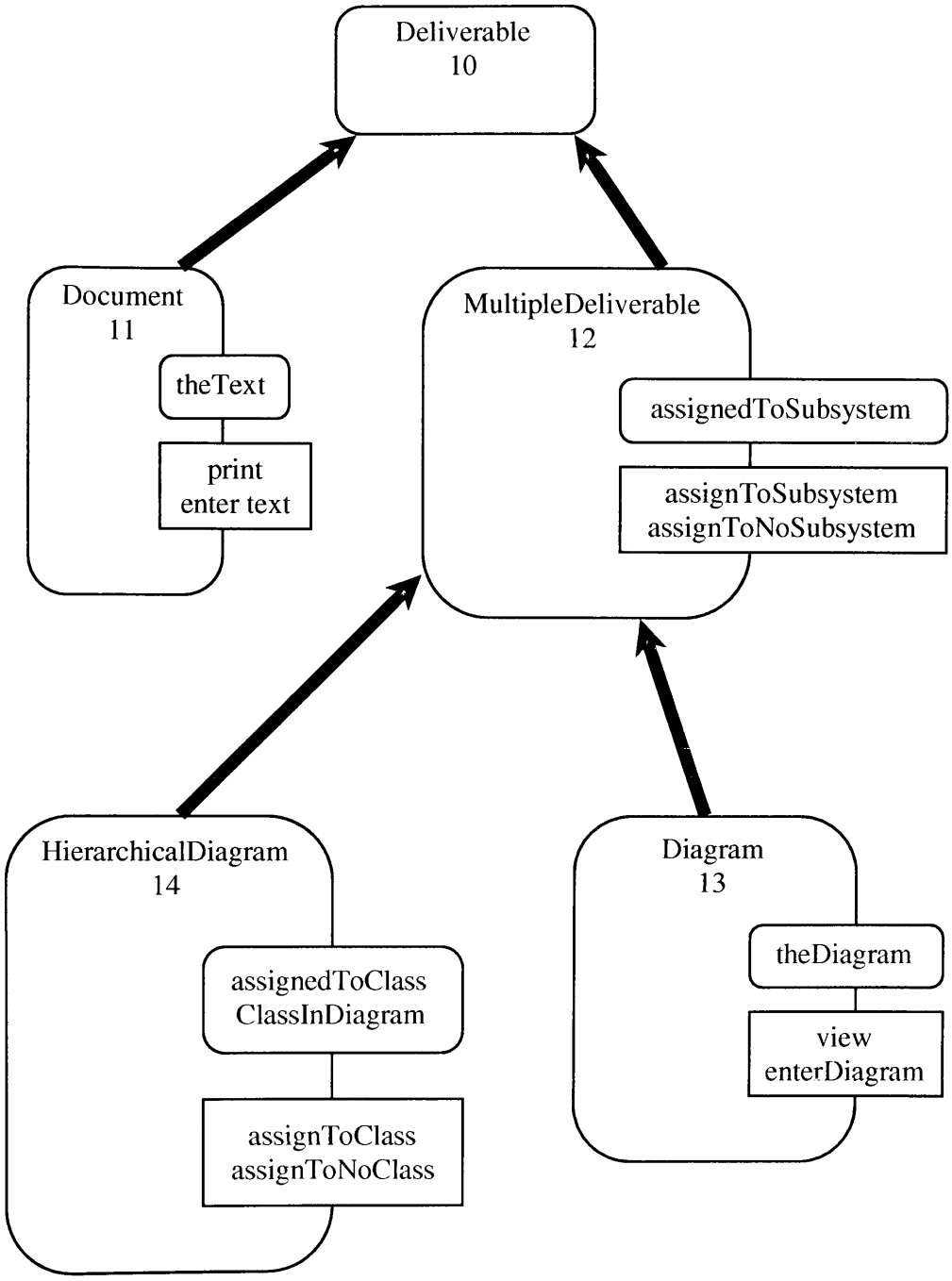
Analysis Diagrams

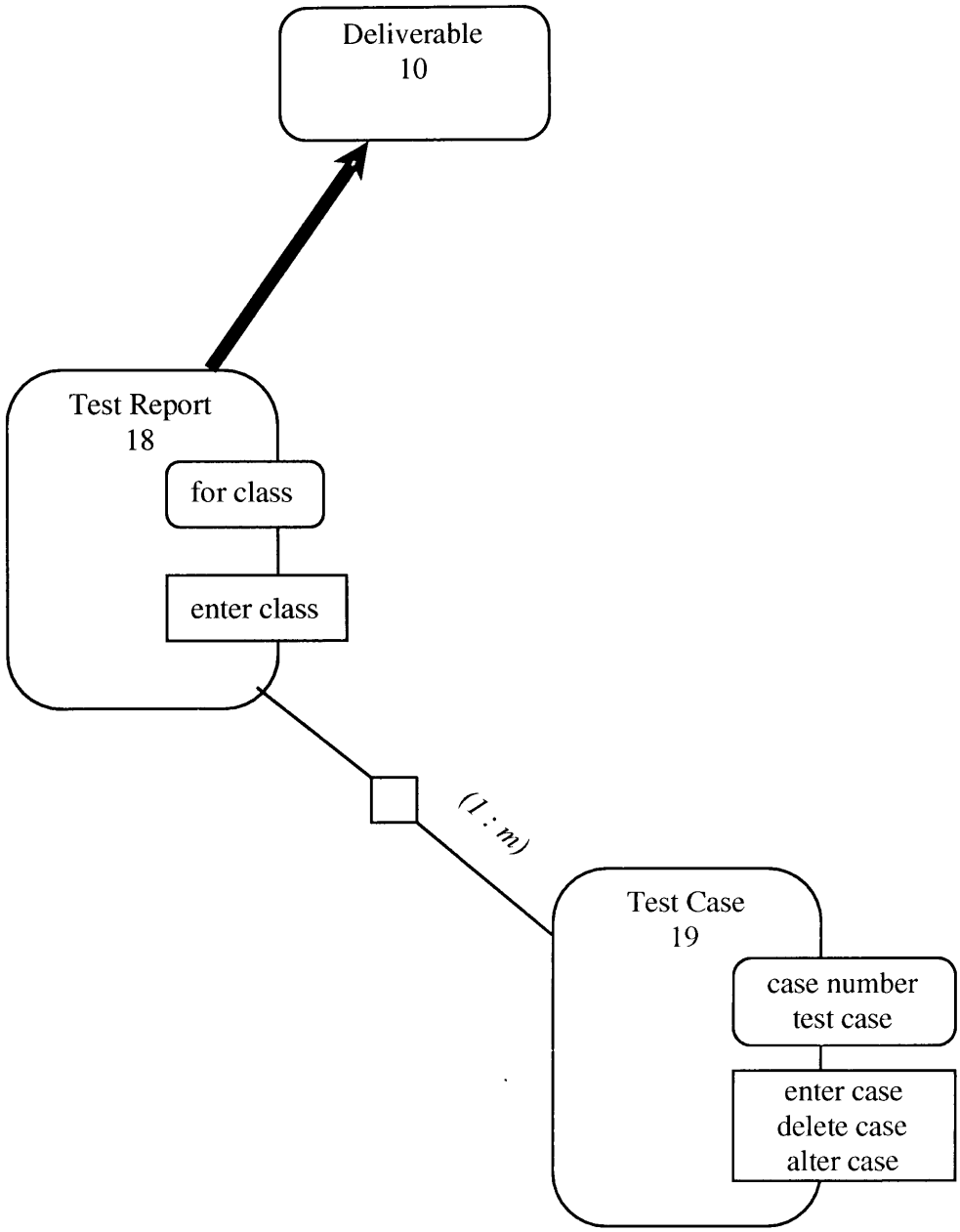
The analysis diagrams identify real-world O/C's and the relationships between them. In addition, each O/C has a list of perations and attributes listed. The notation used is taken directly form MOSES.

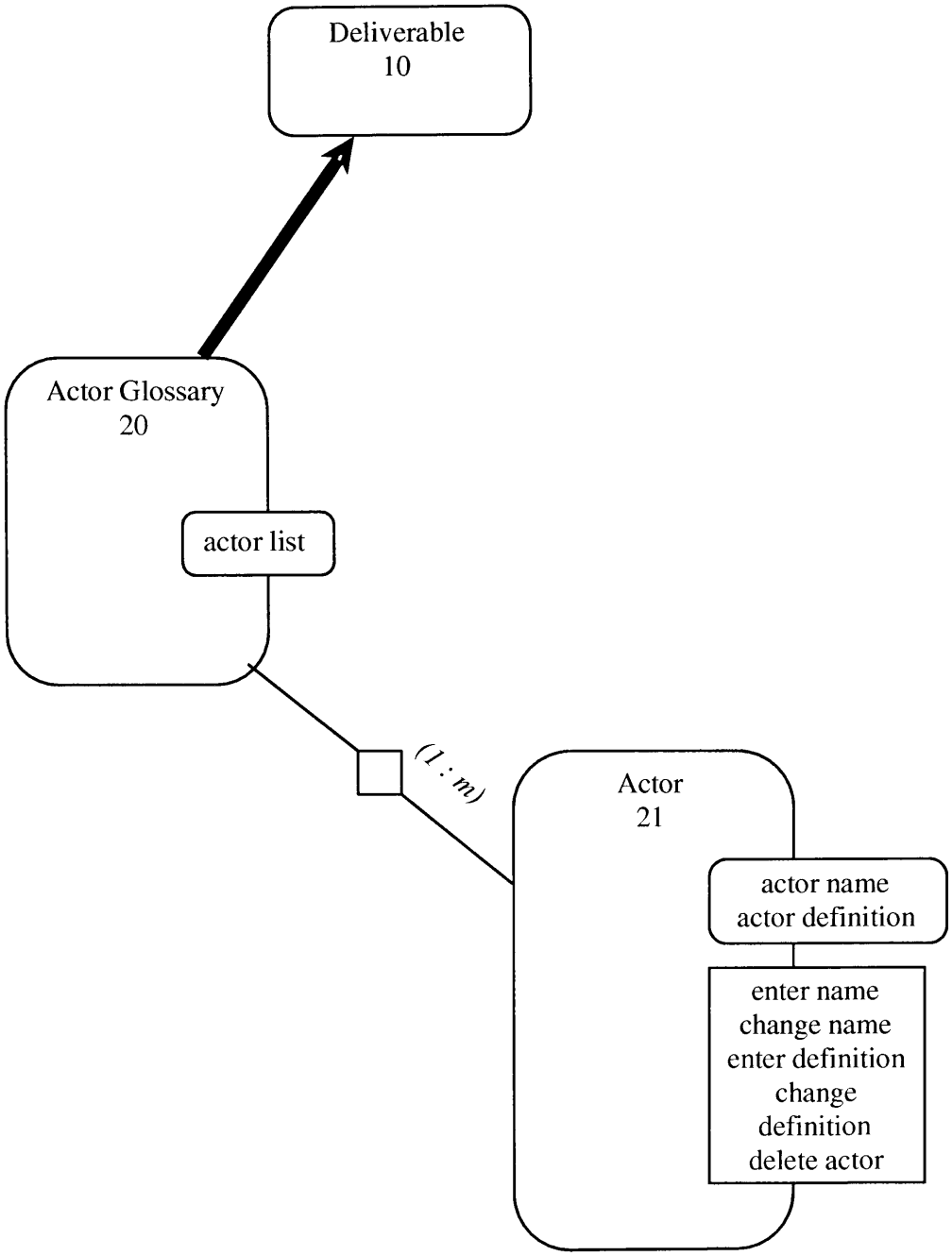

System Login
1
login
password
verify user
obtain
state
(m : 1)
Security Feature
2
state
setState
getState
Deliverable
10
(m :n)
verifies
for
(1 : m)
User
3
name
password
inGroups
add user
remove user
alter user
(m :n)
Group
4
type
has
(m: 1)
Permission
5
type
forGroup
forDeliverable
setPermission
getPermission

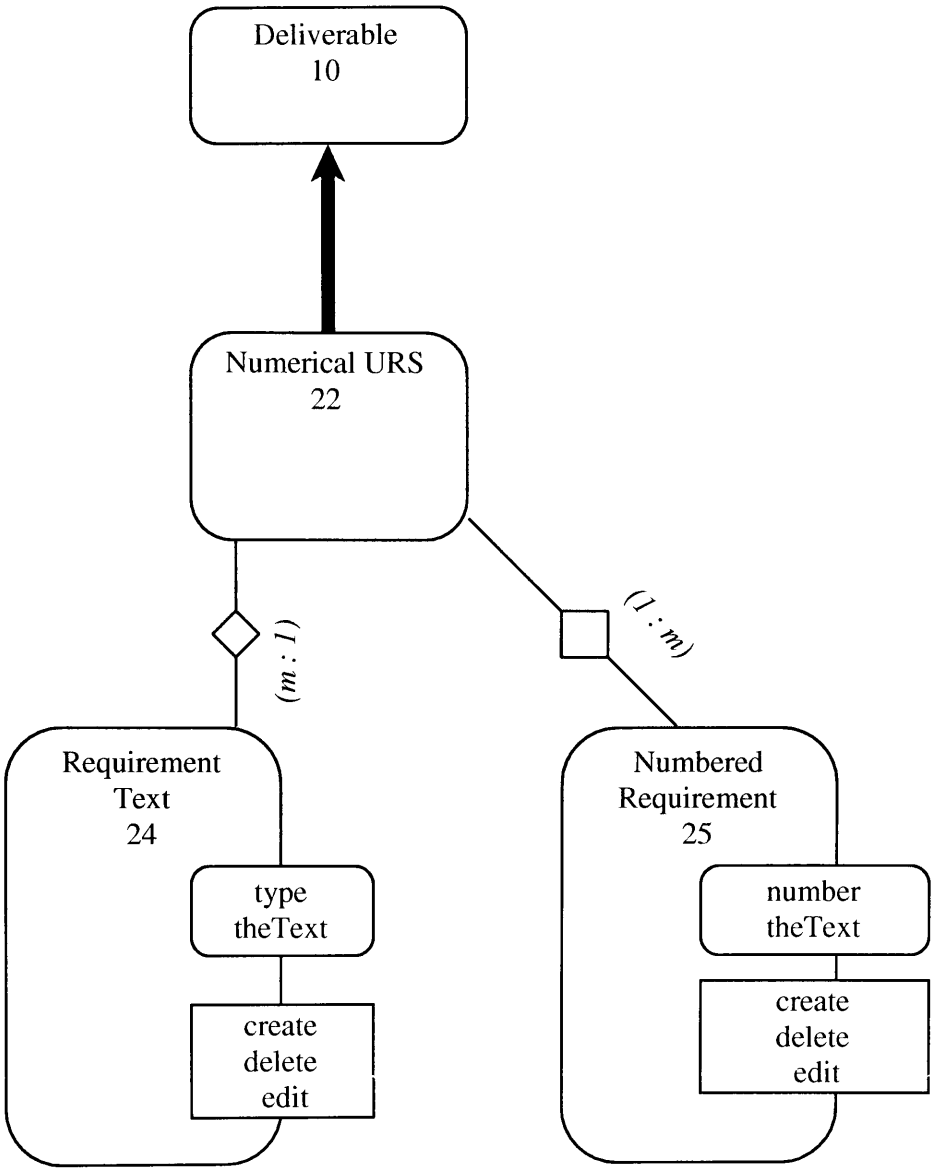
Agenda
6
phase
completedStatus
setCompletedStatus
getCompetedStatus
get
signed off
status
(1 : m)
Deliverable
10
(m :1)
for
Product
7
name
startProduct
deleteProduct
alterProduct
(1: m)
Release
8
number
startDate
endDate
manager
startRelease
deleteRelease
alterRelease

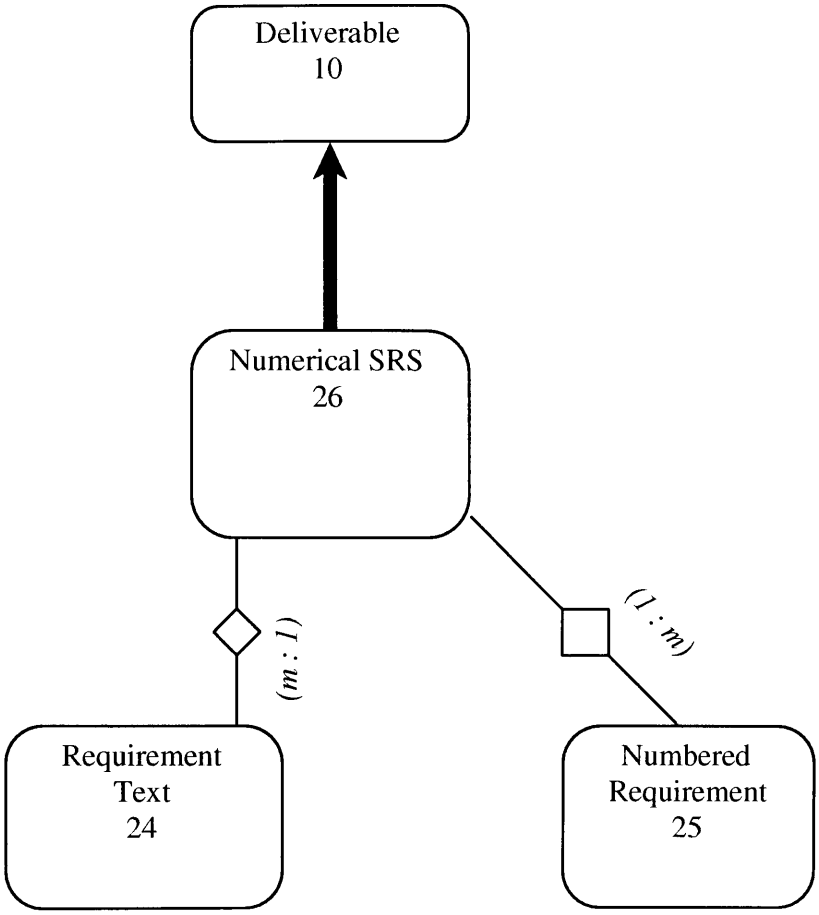

Deliverable
10
assigned to project
assigned to release
signed status
create
edit
delete
annotate
sign off
sign on
select version
assigned to
(m : 1)
Subsystem
9
name
identify
remove
change name
has a
(1 : 1)
Annotation
16
theText
create
delete
edit
has a
(m :1)
has a
(m :n)
Version
15
number
author
date
startVersion
removeVersion
Link
17
toDeliverable
makeLink
removeLink
followLink


Deliverable
10
Document
11
theText
print
enter text
MultipleDeliverable
12
assignedToSubsystem
assignToSubsystem
assignToNoSubsystem
HierarchicalDiagram
14
assignedToClass
ClassInDiagram
assignToClass
assignToNoClass
Diagram
13
theDiagram
view
enterDiagram


Deliverable
10
Test Report
18
for class
enter class
(1 : m)
Test Case
19
case number
test case
enter case
delete case
alter case


Deliverable
10
Actor Glossary
20
actor list
(1 : m)
Actor
21
actor name
actor definition
enter name
change name
enter definition
change
definition
delete actor

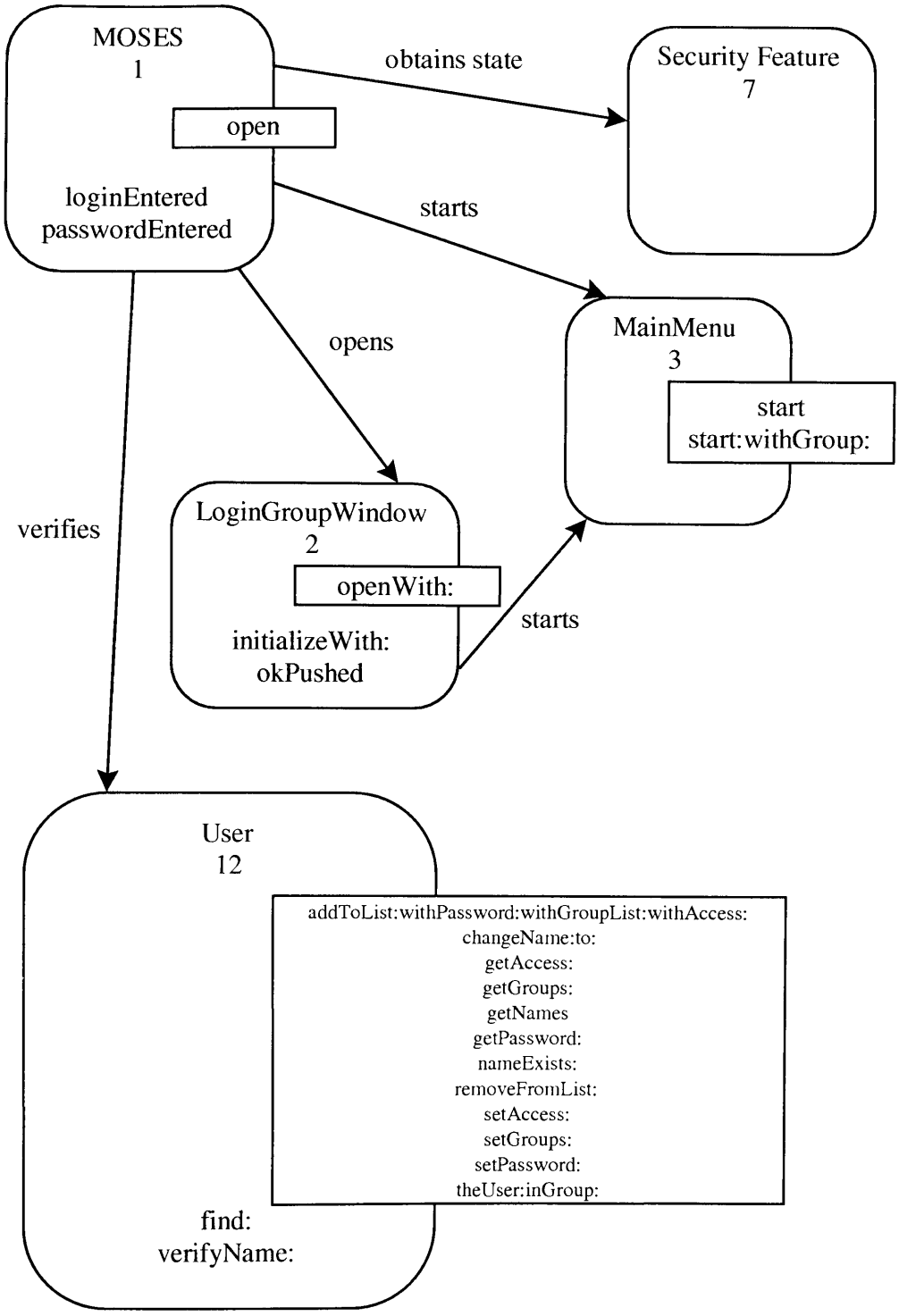

Deliverable
10
Numerical URS
22
(m : 1)
(1 : m)
Requirement
Text
24
type
theText
create
delete
edit
Numbered
Requirement
25
number
theText
create
delete
edit

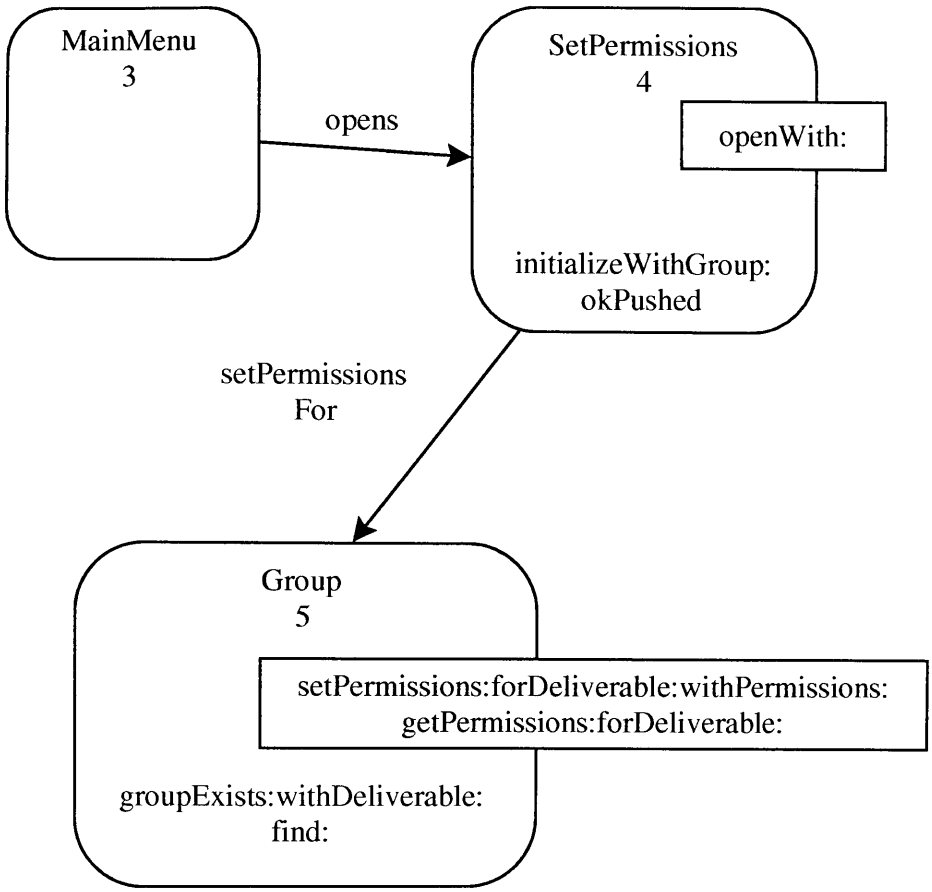

Deliverable
10
Numerical SRS
26
(m : 1)
(1 : m)
Requirement
Text
24
Numbered
Requirement
25

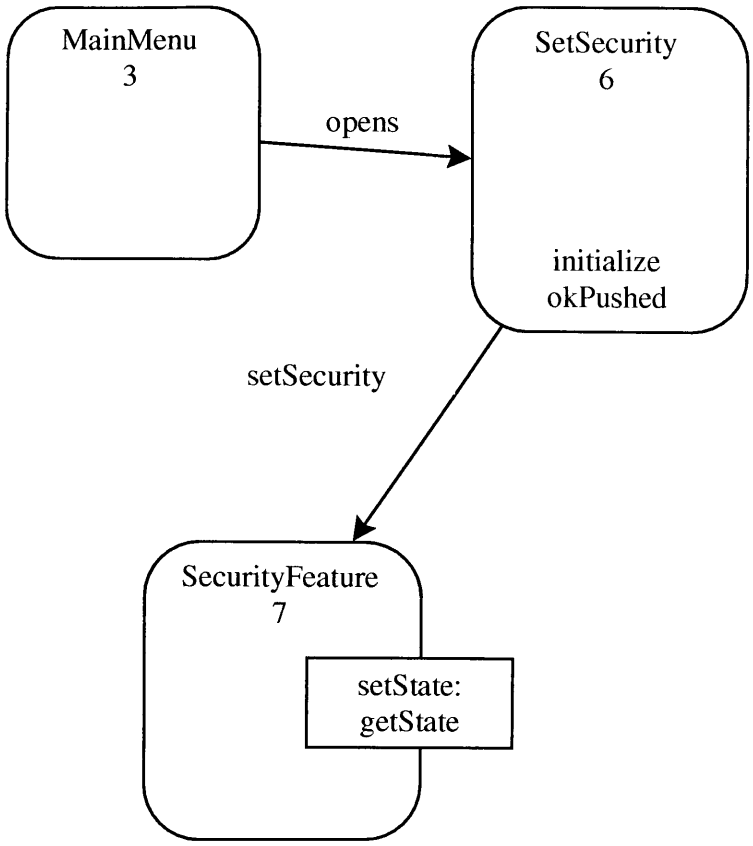
Appendix 3

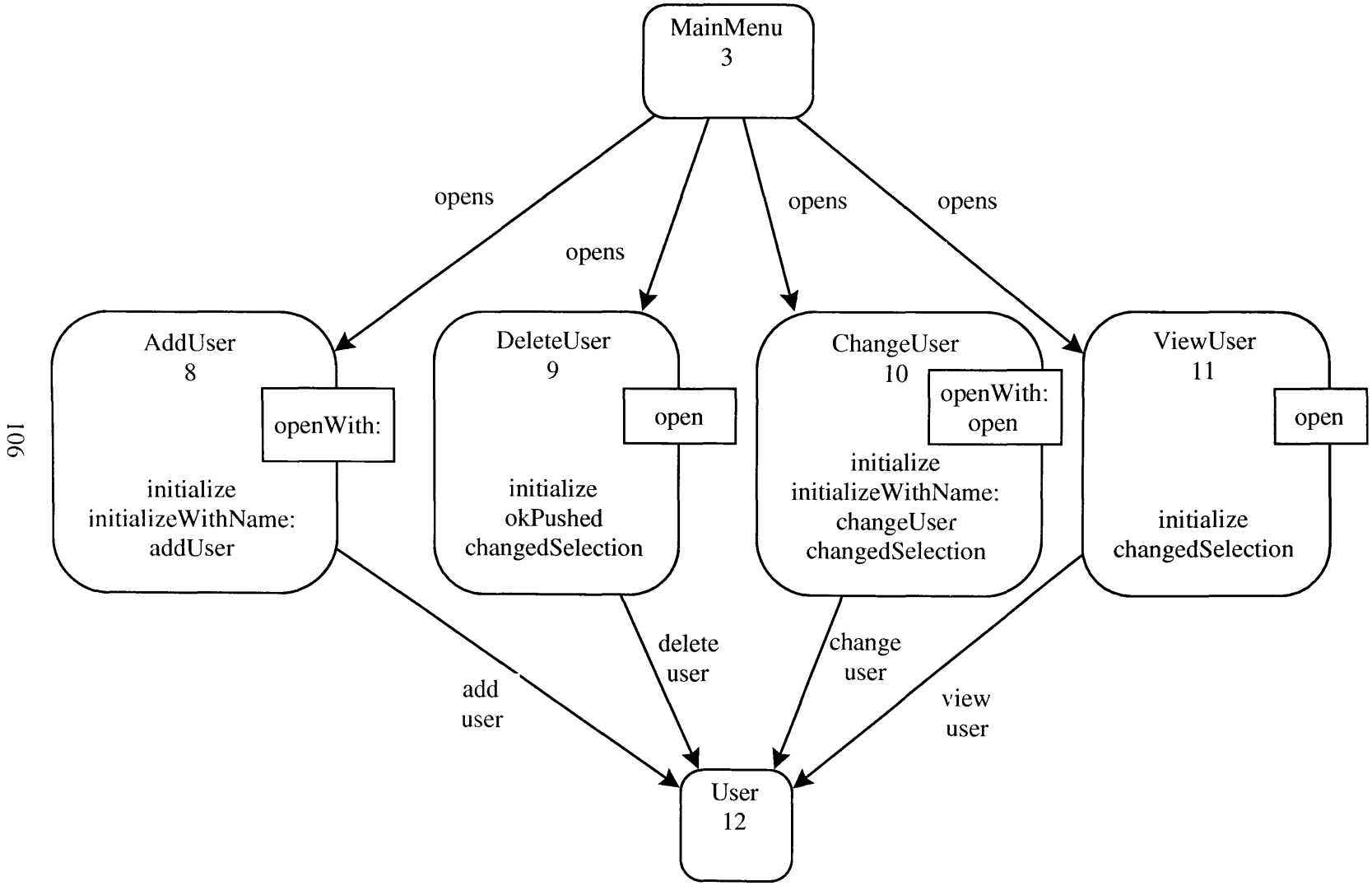
Design Diagrams

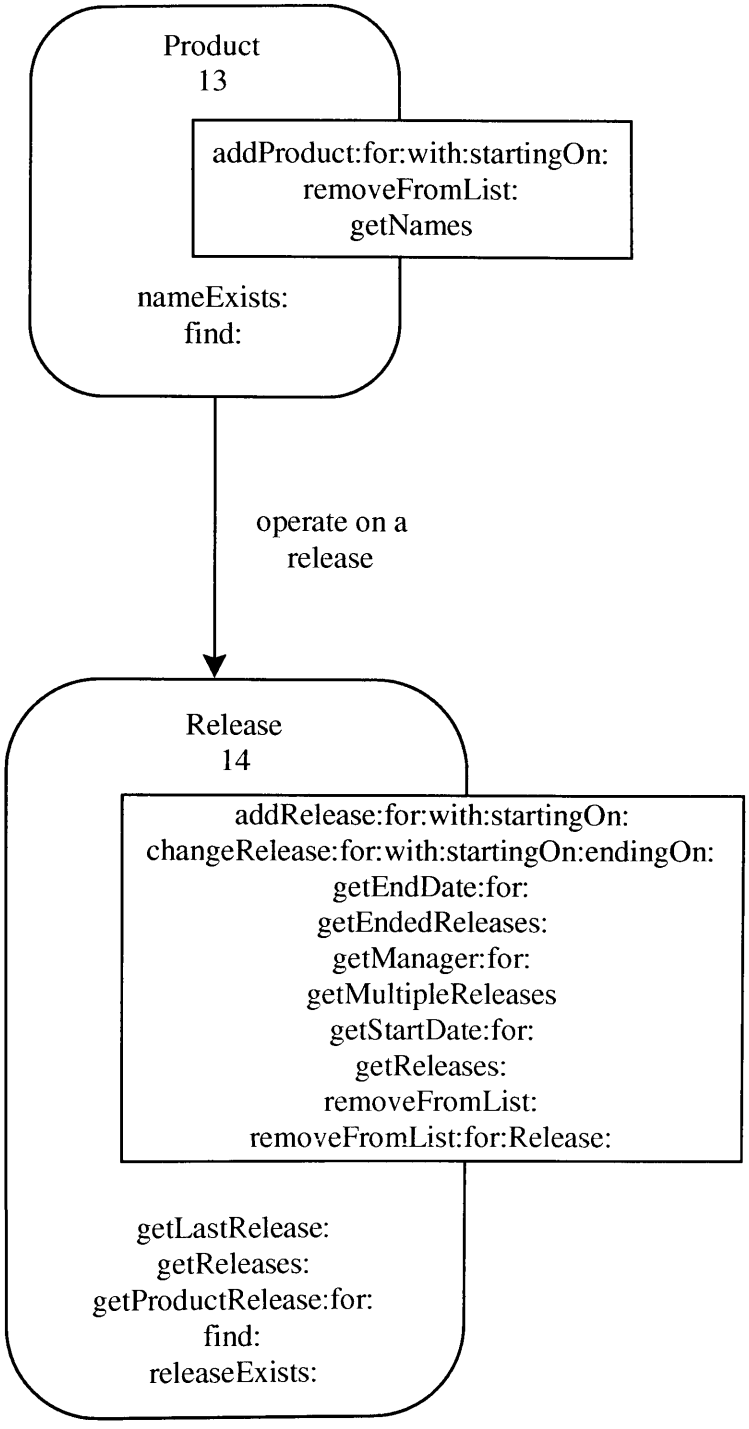
The design diagrams further refine the real-world OC's specified in the analysis diagrams. In addition, the library classes and the user interface O/C's are incorporated. The relationships between the O/C's are either client-server or inheritance.

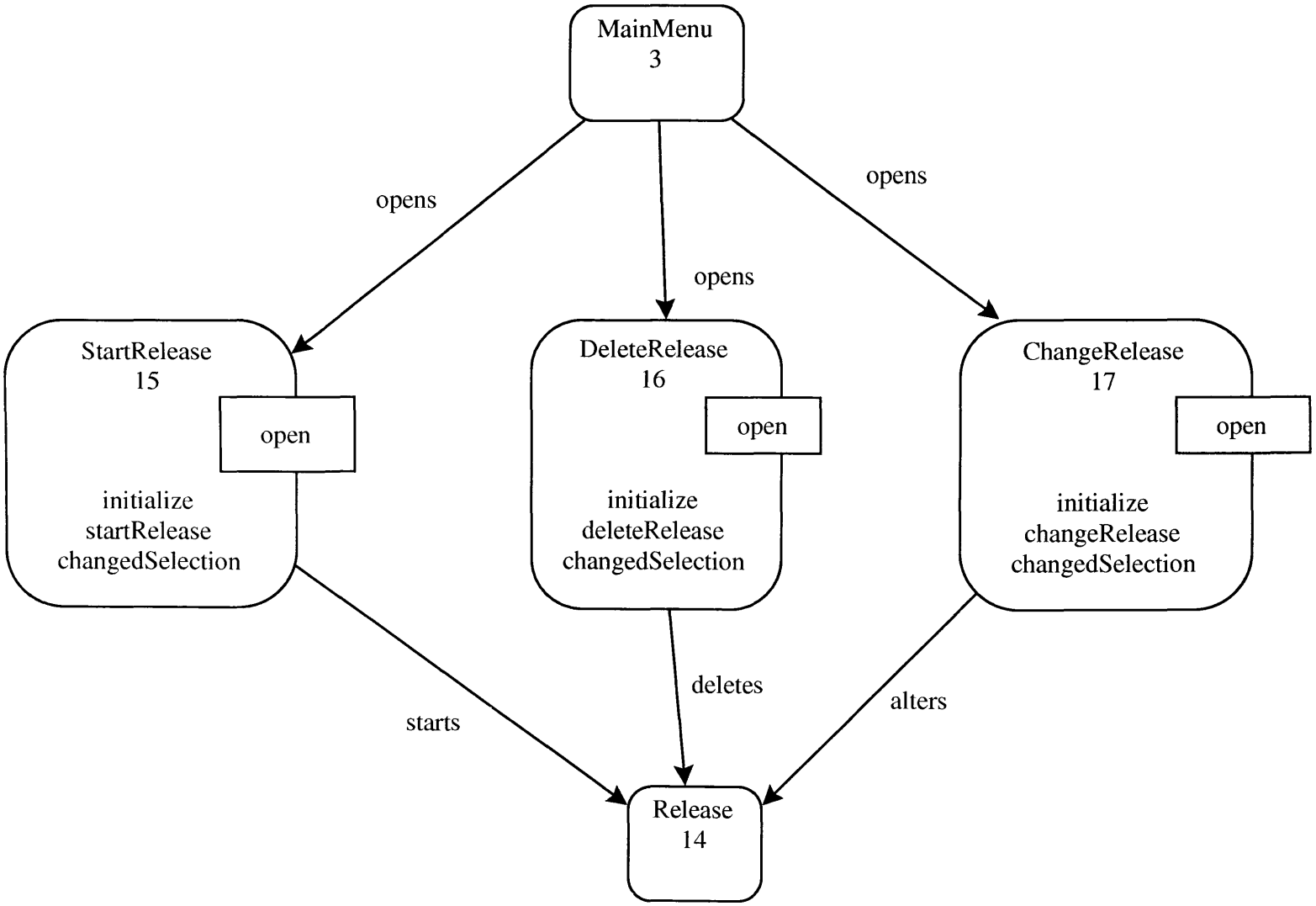

MOSES
1
open
loginEntered
passwordEntered
obtains state
Security Feature
7
starts
opens
MainMenu
3
start
start:withGroup:
verifies
LoginGroupWindow
2
openWith:
initializeWith:
okPushed
starts
User
12
addToList:withPassword:withGroupList:withAccess:
changeName:to:
getAccess:
getGroups:
getNames
getPassword:
nameExists:
removeFromList:
setAccess:
setGroups:
setPassword:
theUser:inGroup:
find:
verifyName:

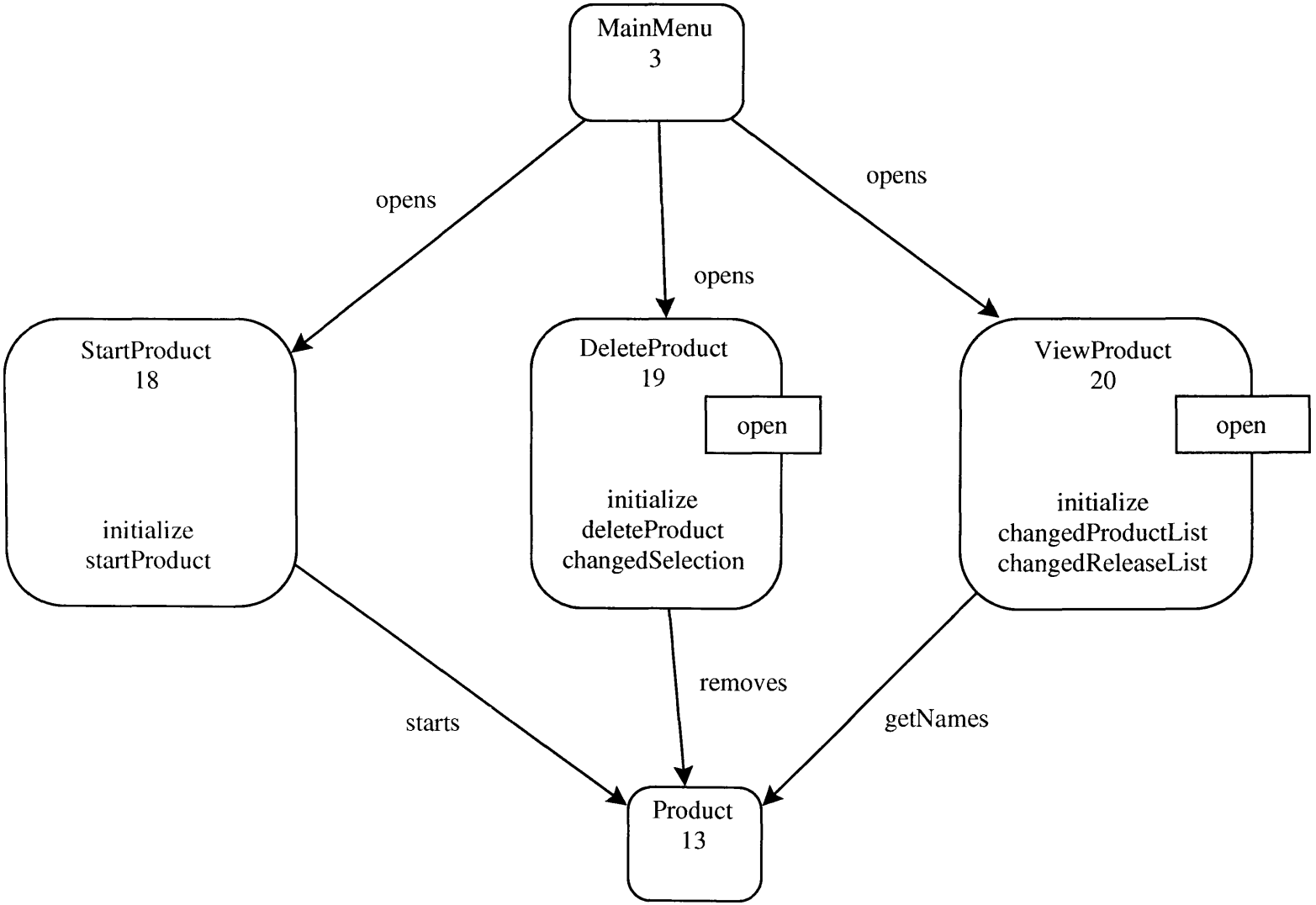

MainMenu
3
opens
SetPermissions
4
openWith:
initializeWithGroup:
okPushed
setPermissions
For
Group
5
setPermissions:forDeliverable:withPermissions:
getPermissions:forDeliverable:
groupExists:withDeliverable:
find:

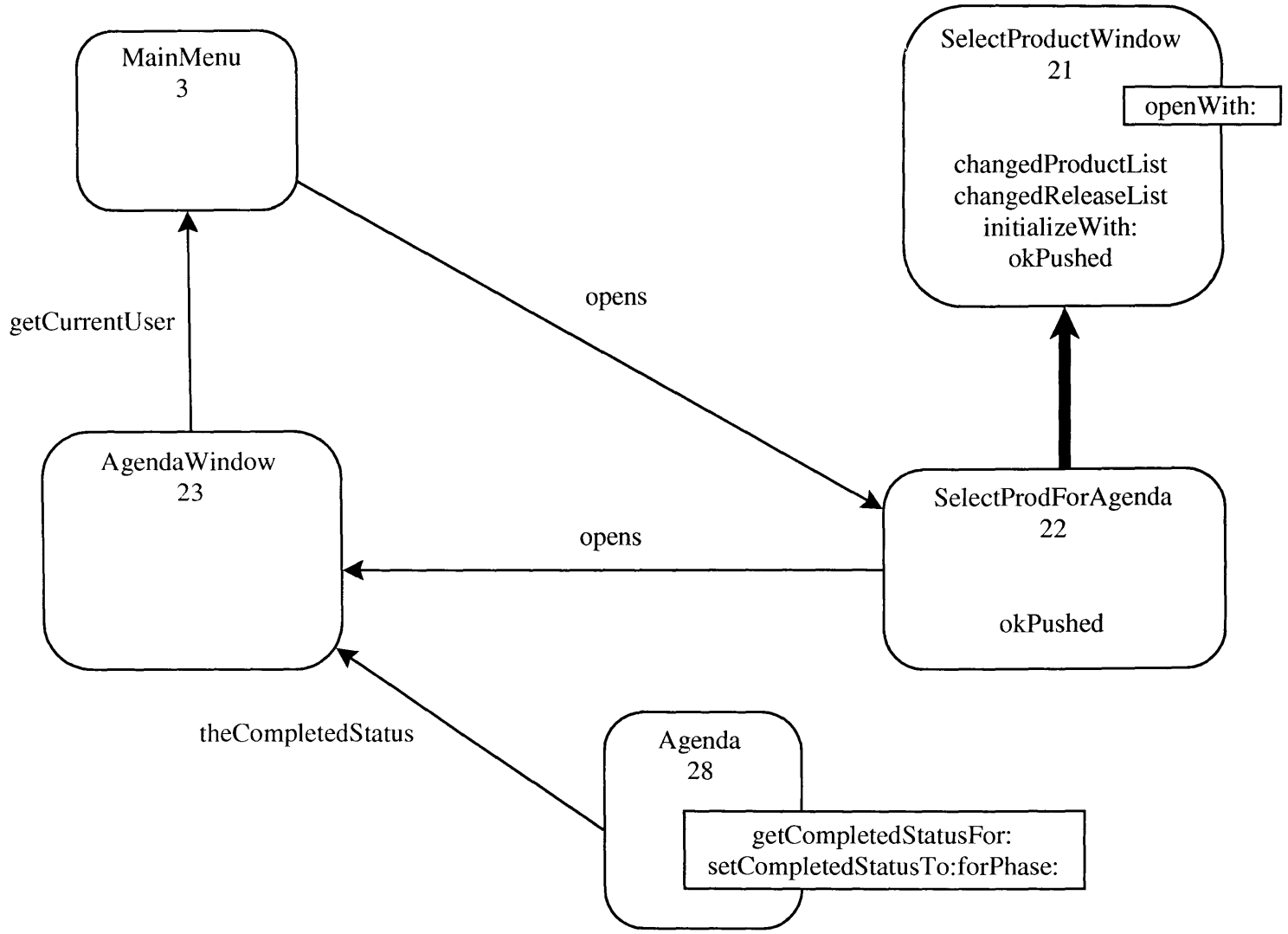

MainMenu
3
opens
SetSecurity
6
initialize
okPushed
setSecurity
SecurityFeature
7
setState:
getState

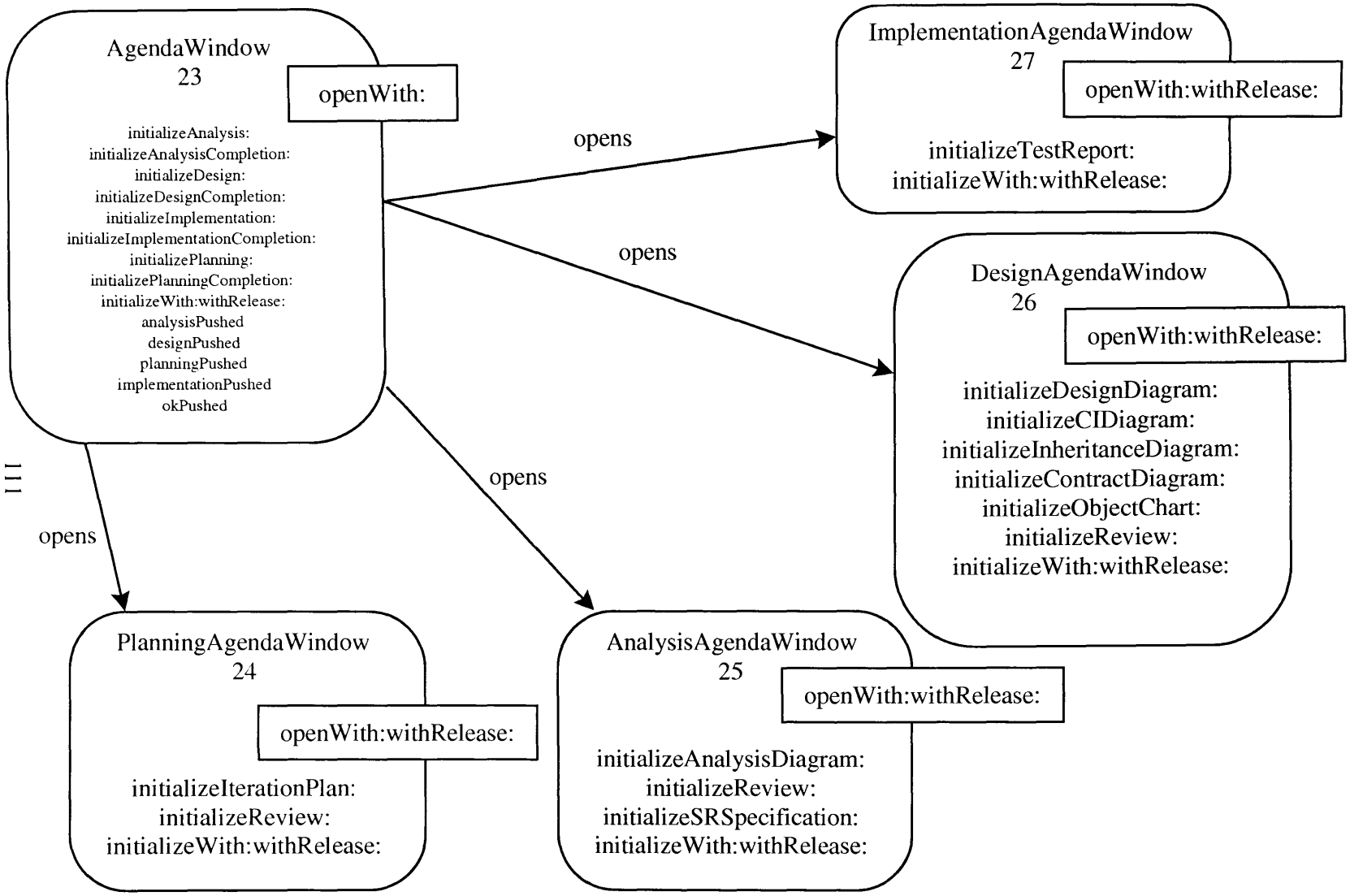

MainMenu
3
opens
opens
opens
opens
AddUser
8
openWith:
initialize
initializeWithName:
addUser
DeleteUser
9
open
initialize
okPushed
changedSelection
ChangeUser
10
openWith:
open
initialize
initializeWithName:
changeUser
changedSelection
ViewUser
11
open
initialize
changedSelection
add
user
delete
user
change
user
view
user
User
12

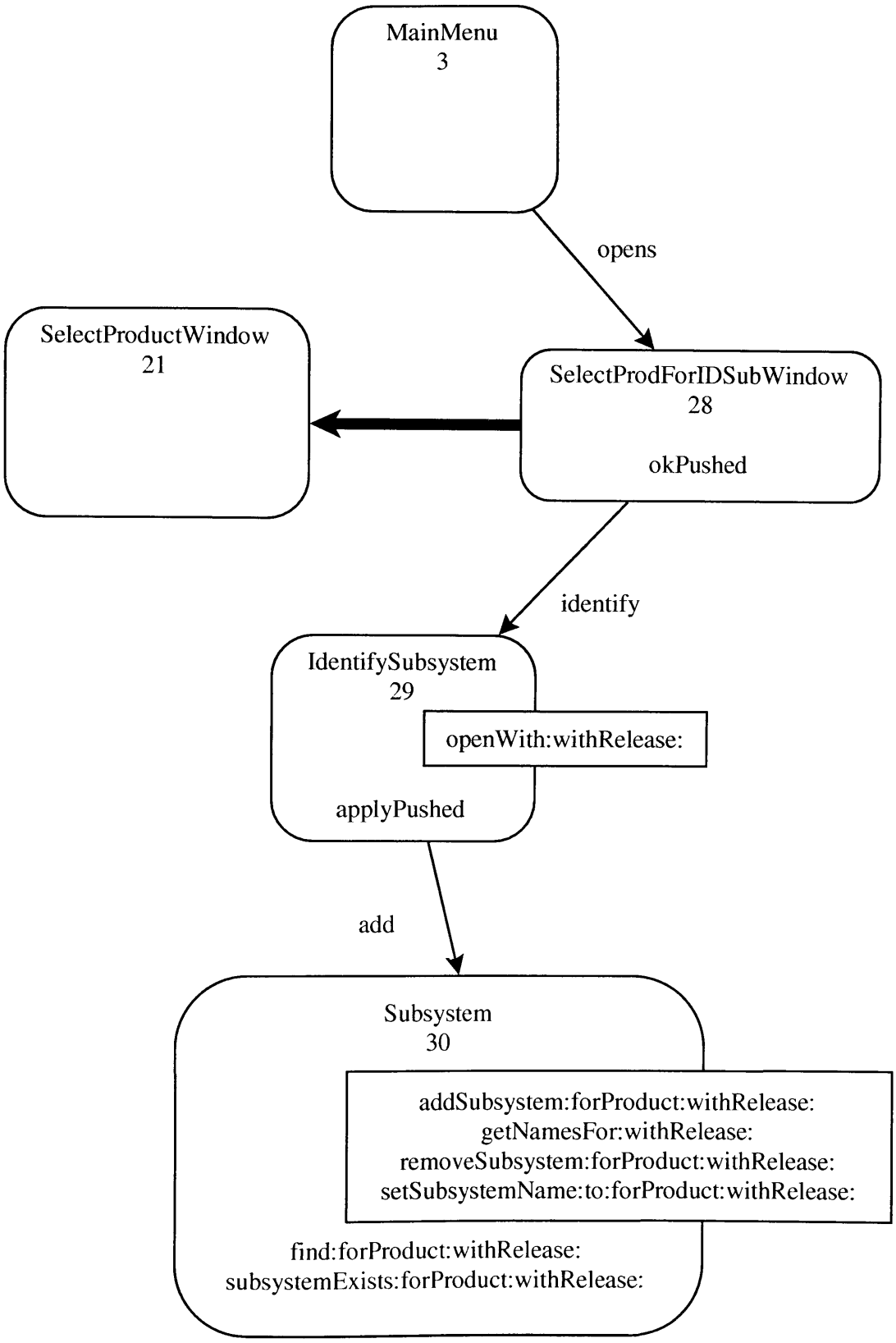

Product
13
addProduct:for:with:startingOn:
removeFromList:
getNames
nameExists:
find:
operate on a
release
Release
14
addRelease:for:with:startingOn:
changeRelease:for:with:startingOn:endingOn:
getEndDate:for:
getEndedReleases:
getManager:for:
getMultipleReleases
getStartDate:for:
getReleases:
removeFromList:
removeFromList:for:Release:
getLastRelease:
getReleases:
getProductRelease:for:
find:
releaseExists:

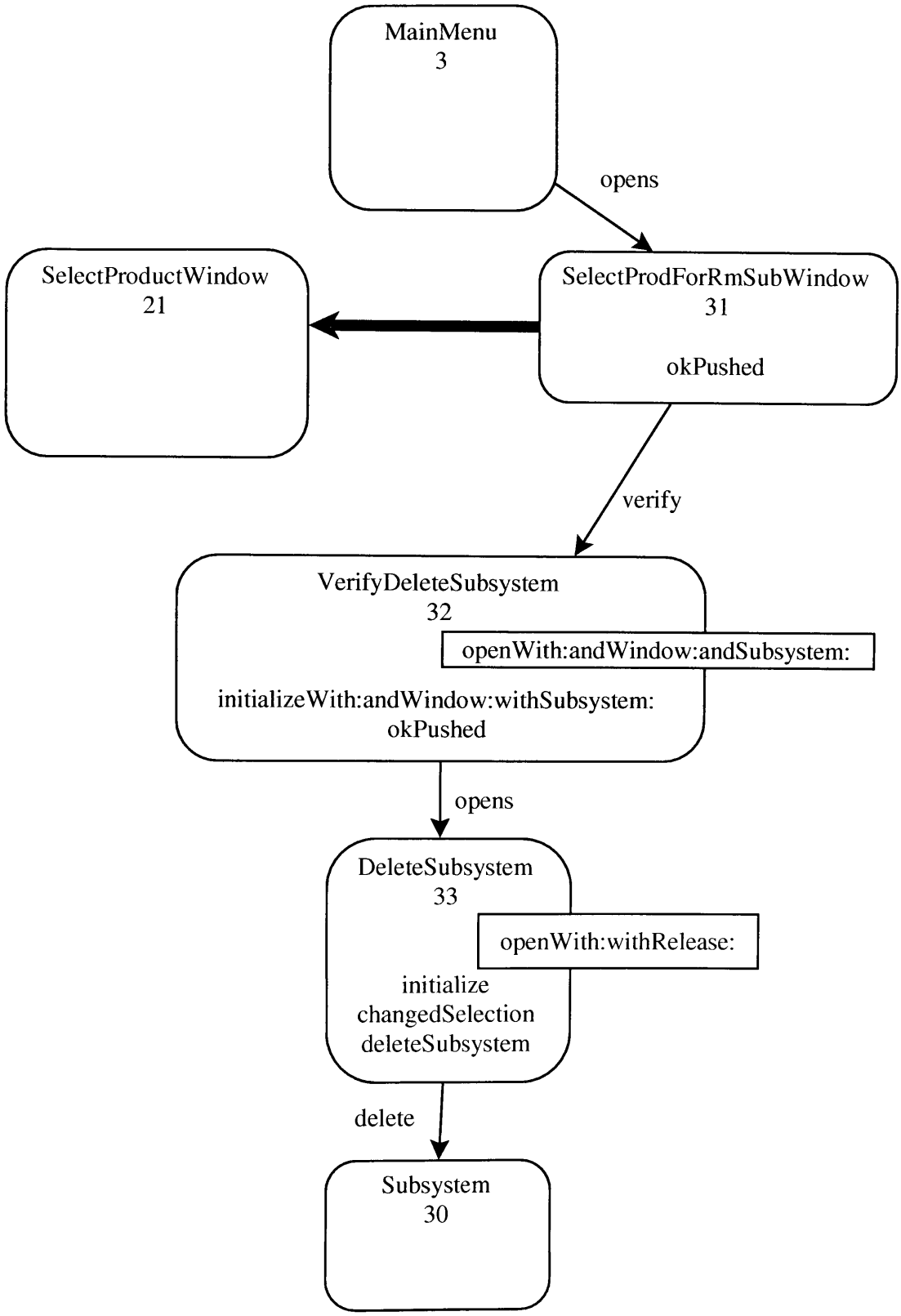

MainMenu
3
opens
opens
opens
StartRelease
15
open
initialize
startRelease
changedSelection
DeleteRelease
16
open
initialize
deleteRelease
changedSelection
ChangeRelease
17
open
initialize
changeRelease
changedSelection
starts
deletes
alters
Release
14

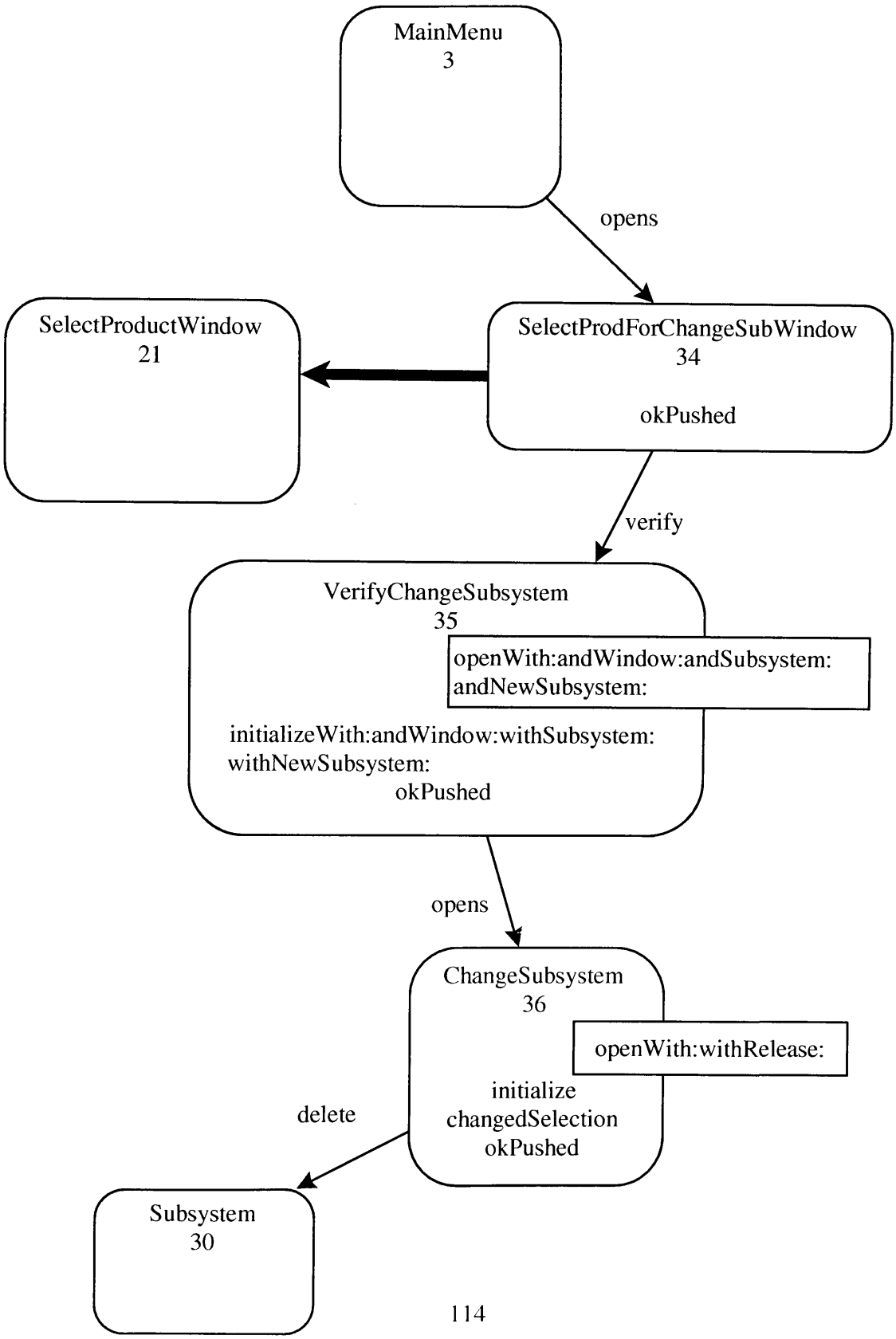

MainMenu
3
opens
opens
opens
StartProduct
18
initialize
startProduct
DeleteProduct
19
open
initialize
deleteProduct
changedSelection
ViewProduct
20
open
initialize
changedProductList
changedReleaseList
removes
starts
getNames
Product
13

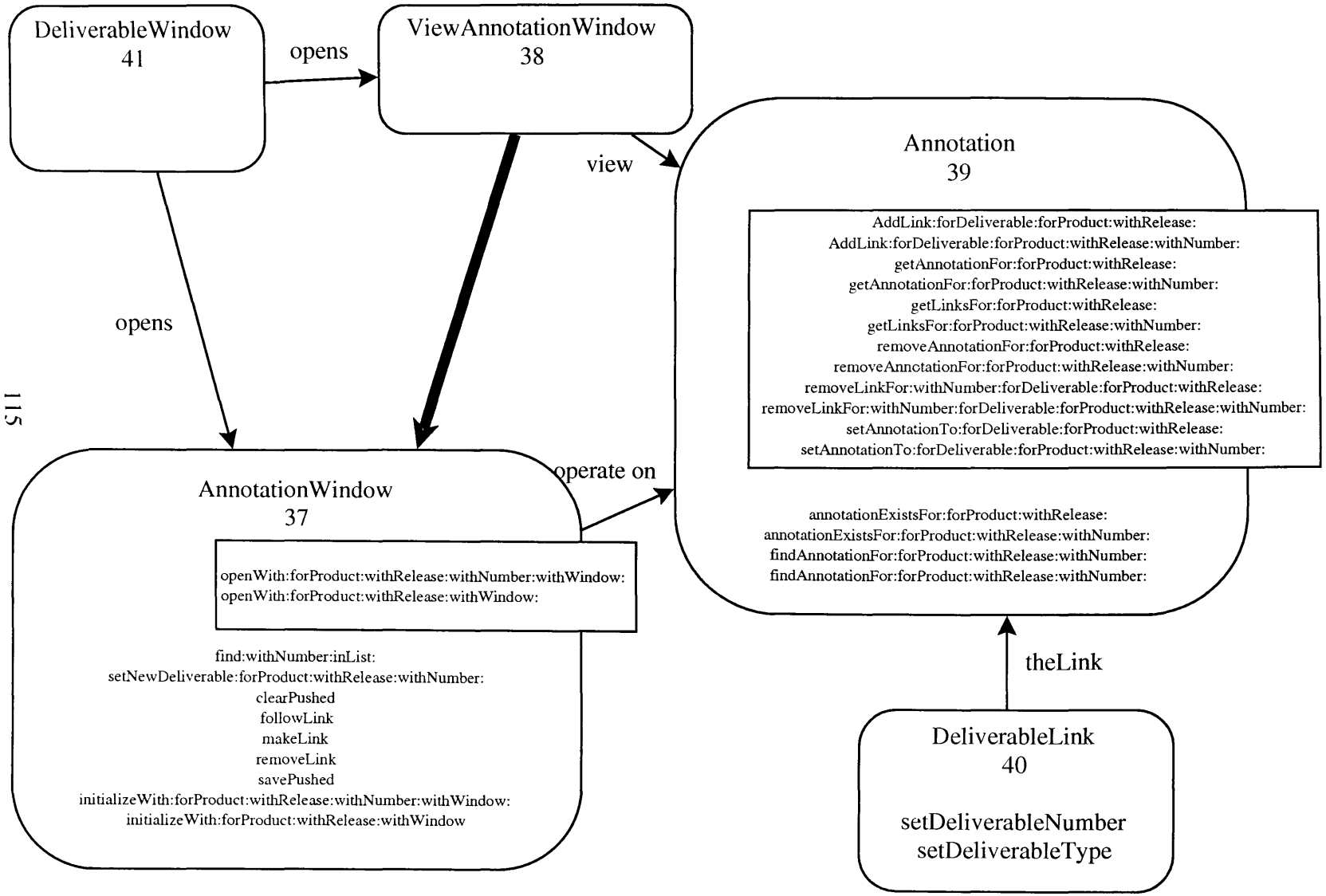

MainMenu
3
SelectProductWindow
21
openWith:
changedProductList
changedReleaseList
initializeWith:
okPushed
opens
getCurrentUser
AgendaWindow
23
SelectProdForAgenda
22
okPushed
opens
theCompletedStatus
Agenda
28
getCompletedStatusFor:
setCompletedStatusTo:forPhase:

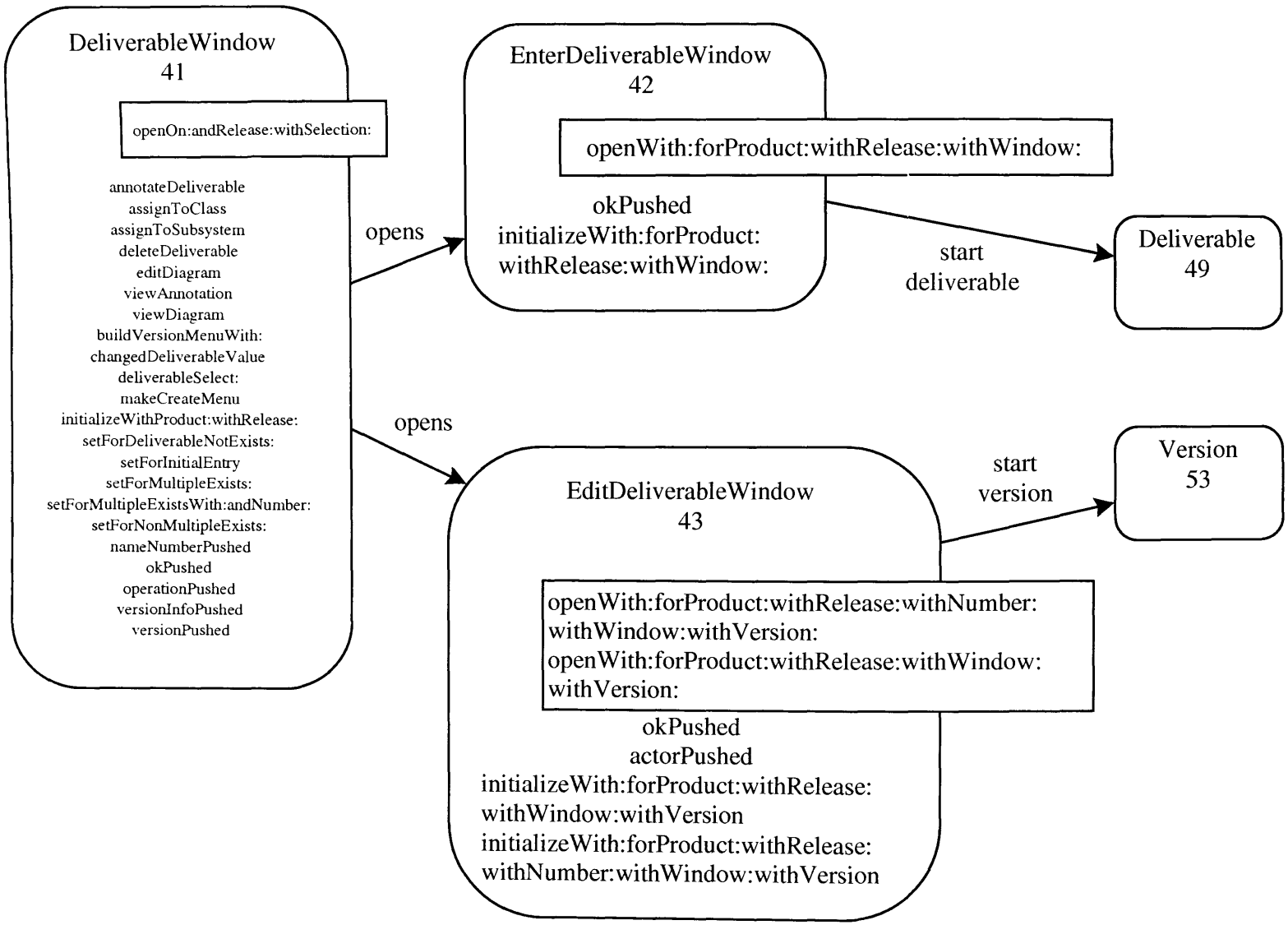

AgendaWindow
23
openWith:
initializeAnalysis:
initializeAnalysisCompletion:
initializeDesign:
initializeDesignCompletion:
initializeImplementation:
initializeImplementationCompletion:
initializePlanning:
initializePlanningCompletion:
initializeWith:withRelease:
analysisPushed
designPushed
planningPushed
implementationPushed
okPushed
opens
ImplementationAgendaWindow
27
openWith:withRelease:
initializeTestReport:
initializeWith:withRelease:
opens
DesignAgendaWindow
26
openWith:withRelease:
initializeDesignDiagram:
initializeCIDiagram:
initializeInheritanceDiagram:
initializeContractDiagram:
initializeObjectChart:
initializeReview:
initializeWith:withRelease:
opens
opens
PlanningAgendaWindow
24
openWith:withRelease:
initializeIterationPlan:
initializeReview:
initializeWith:withRelease:
AnalysisAgendaWindow
25
openWith:withRelease:
initializeAnalysisDiagram:
initializeReview:
initializeSRSpecification:
initializeWith:withRelease:

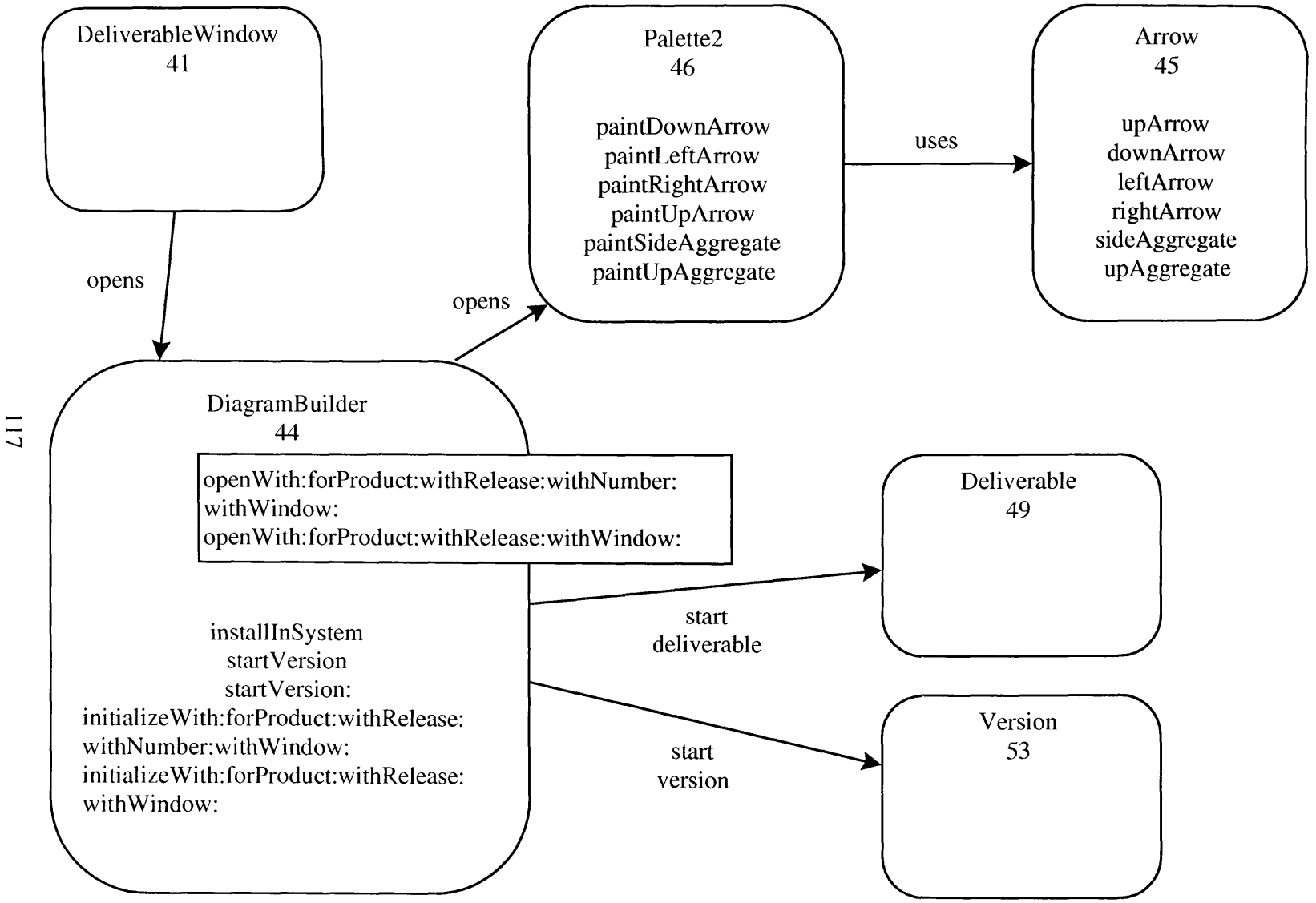

MainMenu
3
opens
SelectProductWindow
21
SelectProdForIDSubWindow
28
okPushed
identify
IdentifySubsystem
29
openWith:withRelease:
applyPushed
add
Subsystem
30
addSubsystem:forProduct:withRelease:
getNamesFor:withRelease:
removeSubsystem:forProduct:withRelease:
setSubsystemName:to:forProduct:withRelease:
find:forProduct:withRelease:
subsystemExists:forProduct:withRelease:

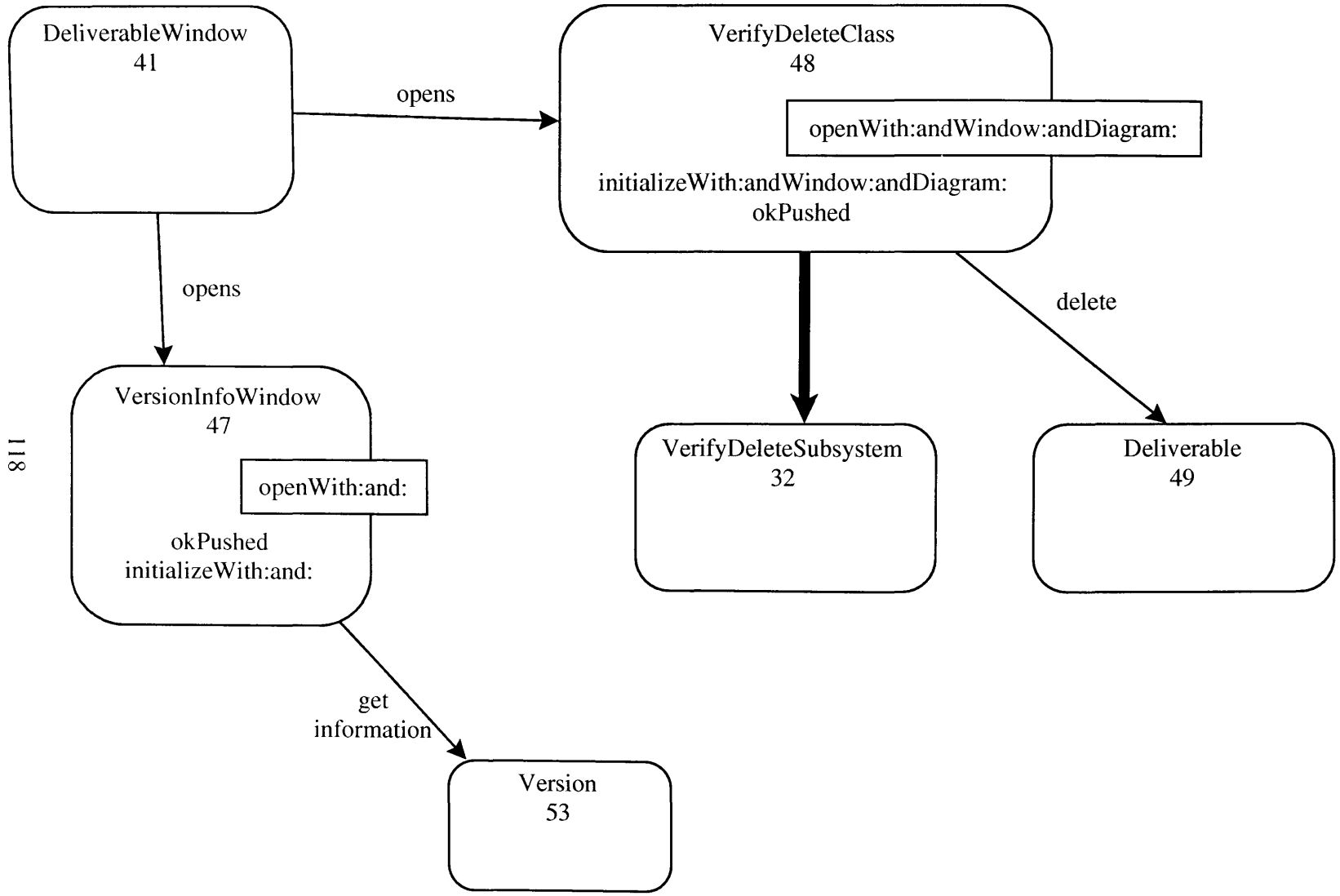

MainMenu
3
opens
SelectProductWindow
21
SelectProdForRmSubWindow
31
okPushed
verify
VerifyDeleteSubsystem
32
openWith:andWindow:andSubsystem:
initializeWith:andWindow:withSubsystem:
okPushed
opens
DeleteSubsystem
33
openWith:withRelease:
initialize
changedSelection
deleteSubsystem
delete
Subsystem
30


MainMenu
3
opens
SelectProductWindow
21
SelectProdForChangeSubWindow
34
okPushed
verify
VerifyChangeSubsystem
35
openWith:andWindow:andSubsystem:
andNewSubsystem:
initializeWith:andWindow:withSubsystem:
withNewSubsystem:
okPushed
opens
ChangeSubsystem
36
openWith:withRelease:
initialize
changedSelection
okPushed
delete
Subsystem
30

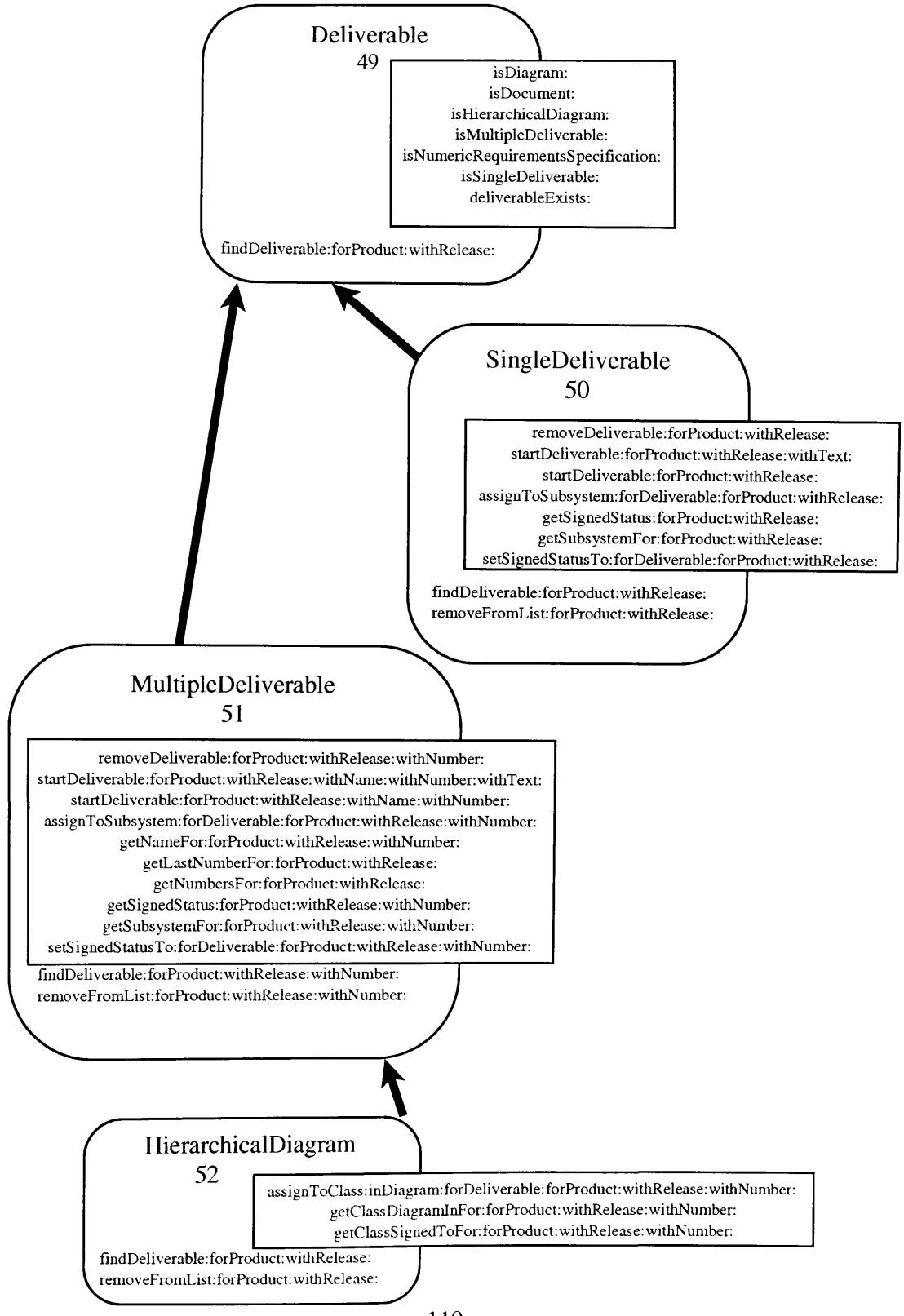

DeliverableWindow
41
opens
ViewAnnotationWindow
38
view
Annotation
39
AddLink:forDeliverable:forProduct:withRelease:
AddLink:forDeliverable:forProduct:withRelease:withNumber:
getAnnotationFor:forProduct:withRelease:
getAnnotationFor:forProduct:withRelease:withNumber:
getLinksFor:forProduct:withRelease:
getLinksFor:forProduct:withRelease:withNumber:
removeAnnotationFor:forProduct:withRelease:
removeAnnotationFor:forProduct:withRelease:withNumber:
removeLinkFor:withNumber:forDeliverable:forProduct:withRelease:
removeLinkFor:withNumber:forDeliverable:forProduct:withRelease:withNumber:
setAnnotationTo:forDeliverable:forProduct:withRelease:
setAnnotationTo:forDeliverable:forProduct:withRelease:withNumber:
opens
operate on
AnnotationWindow
37
openWith:forProduct:withRelease:withNumber:withWindow:
openWith:forProduct:withRelease:withWindow:
find:withNumber:inList:
setNewDeliverable:forProduct:withRelease:withNumber:
clearPushed
followLink
makeLink
removeLink
savePushed
initializeWith:forProduct:withRelease:withNumber:withWindow:
initializeWith:forProduct:withRelease:withWindow
annotationExistsFor:forProduct:withRelease:
annotationExistsFor:forProduct:withRelease:withNumber:
findAnnotationFor:forProduct:withRelease:withNumber:
findAnnotationFor:forProduct:withRelease:withNumber:
theLink
DeliverableLink
40
setDeliverableNumber
setDeliverableType

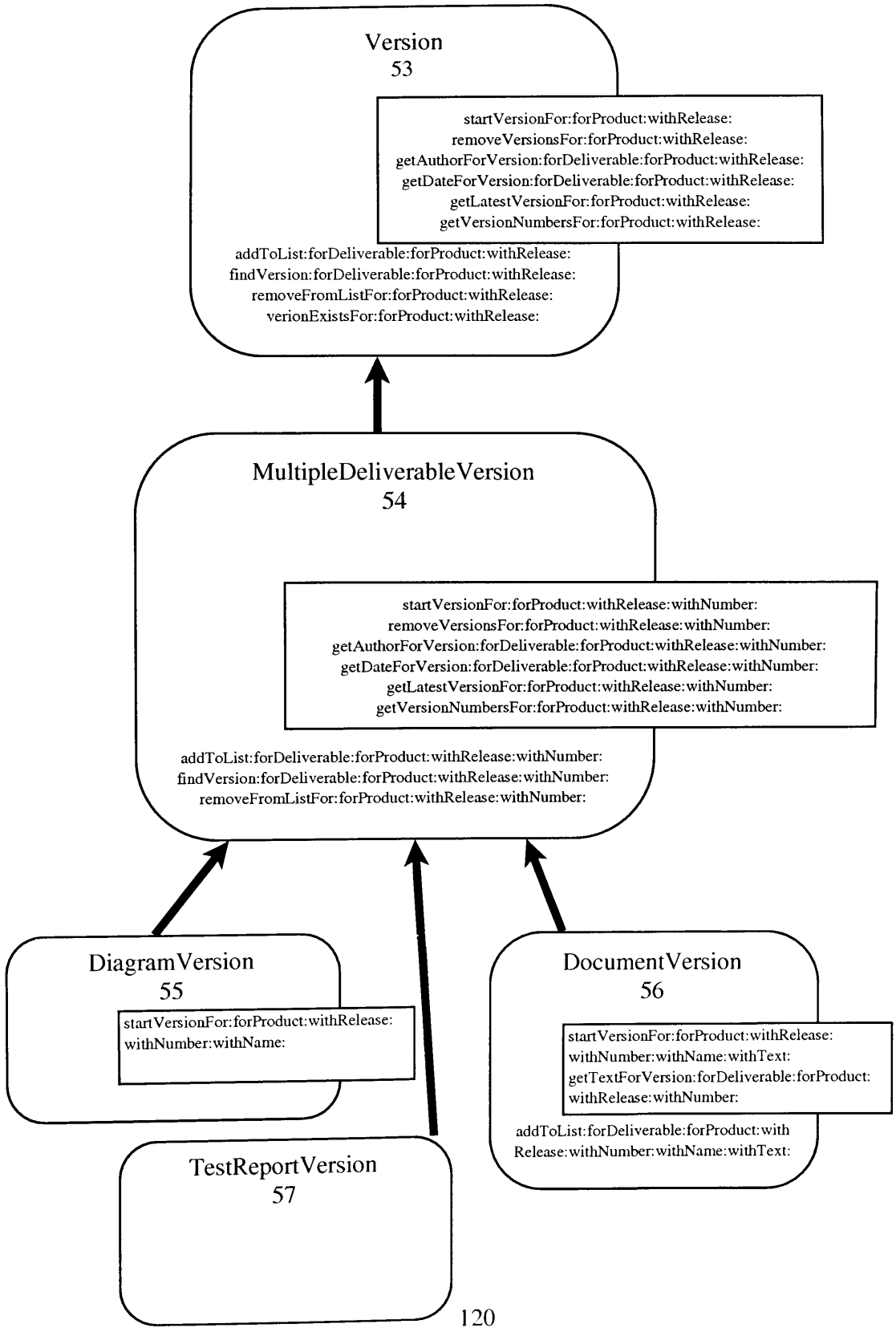

DeliverableWindow
41
openOn:andRelease:withSelection:
annotateDeliverable
assignToClass
assignToSubsystem
deleteDeliverable
editDiagram
viewAnnotation
viewDiagram
buildVersionMenuWith:
changedDeliverableValue
deliverableSelect:
makeCreateMenu
initializeWithProduct:withRelease:
setForDeliverableNotExists:
setForInitialEntry
setForMultipleExists:
setForMultipleExistsWith:andNumber:
setForNonMultipleExists:
nameNumberPushed
okPushed
operationPushed
versionInfoPushed
versionPushed
opens
EnterDeliverableWindow
42
openWith:forProduct:withRelease:withWindow:
okPushed
initializeWith:forProduct:
withRelease:withWindow:
start
deliverable
Deliverable
49
opens
EditDeliverableWindow
43
start
version
Version
53
openWith:forProduct:withRelease:withNumber:
withWindow:withVersion:
openWith:forProduct:withRelease:withWindow:
withVersion:
okPushed
actorPushed
initializeWith:forProduct:withRelease:
withWindow:withVersion
initializeWith:forProduct:withRelease:
withNumber:withWindow:withVersion

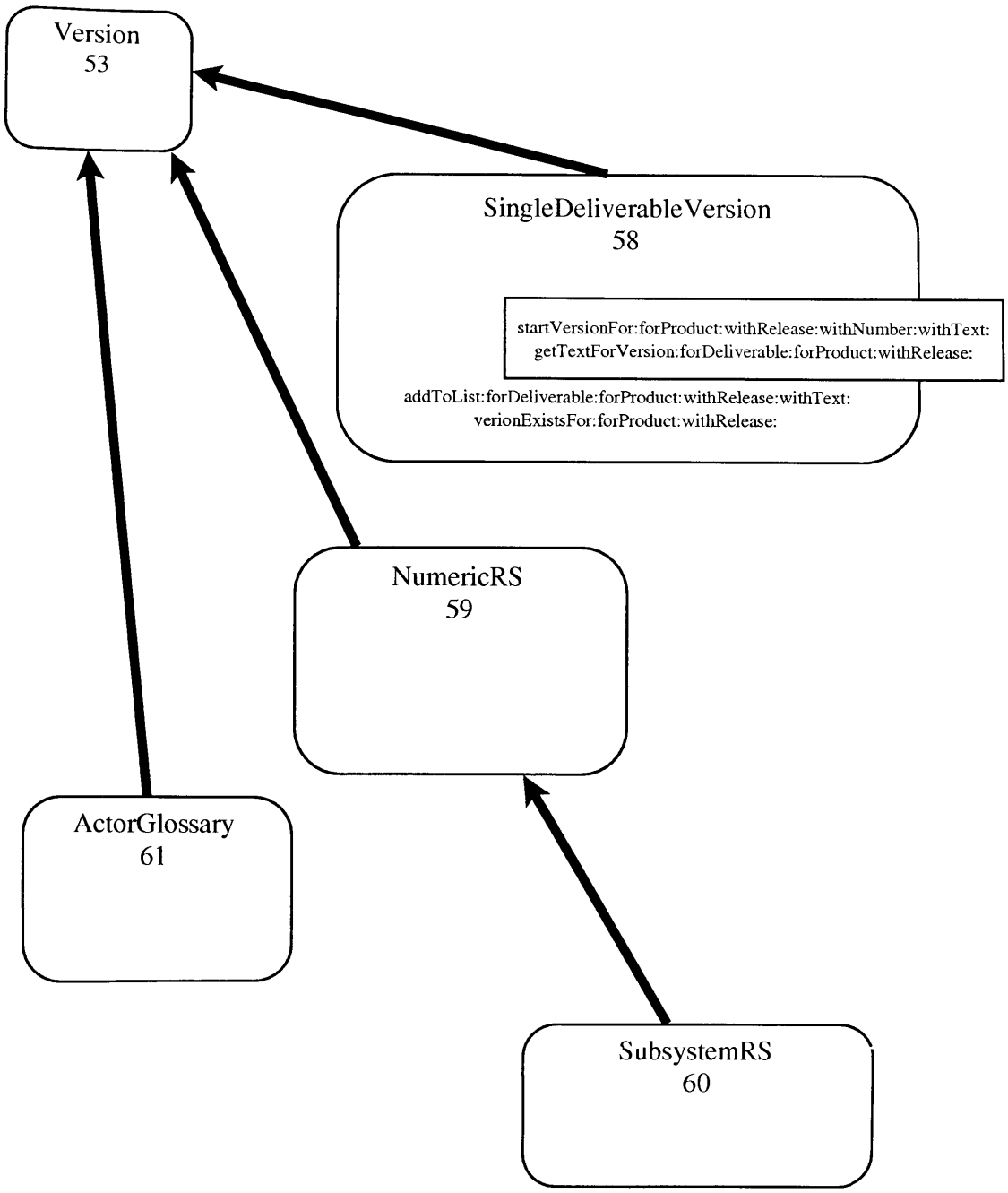

DeliverableWindow
41
Palette2
46
paintDownArrow
paintLeftArrow
paintRightArrow
paintUpArrow
paintSideAggregate
paintUpAggregate
uses
Arrow
45
upArrow
downArrow
leftArrow
rightArrow
sideAggregate
upAggregate
opens
opens
DiagramBuilder
44
openWith:forProduct:withRelease:withNumber:
withWindow:
openWith:forProduct:withRelease:withWindow:
installInSystem
startVersion
startVersion:
initializeWith:forProduct:withRelease:
withNumber:withWindow:
initializeWith:forProduct:withRelease:
withWindow:
start
deliverable
Deliverable
49
start
version
Version
53

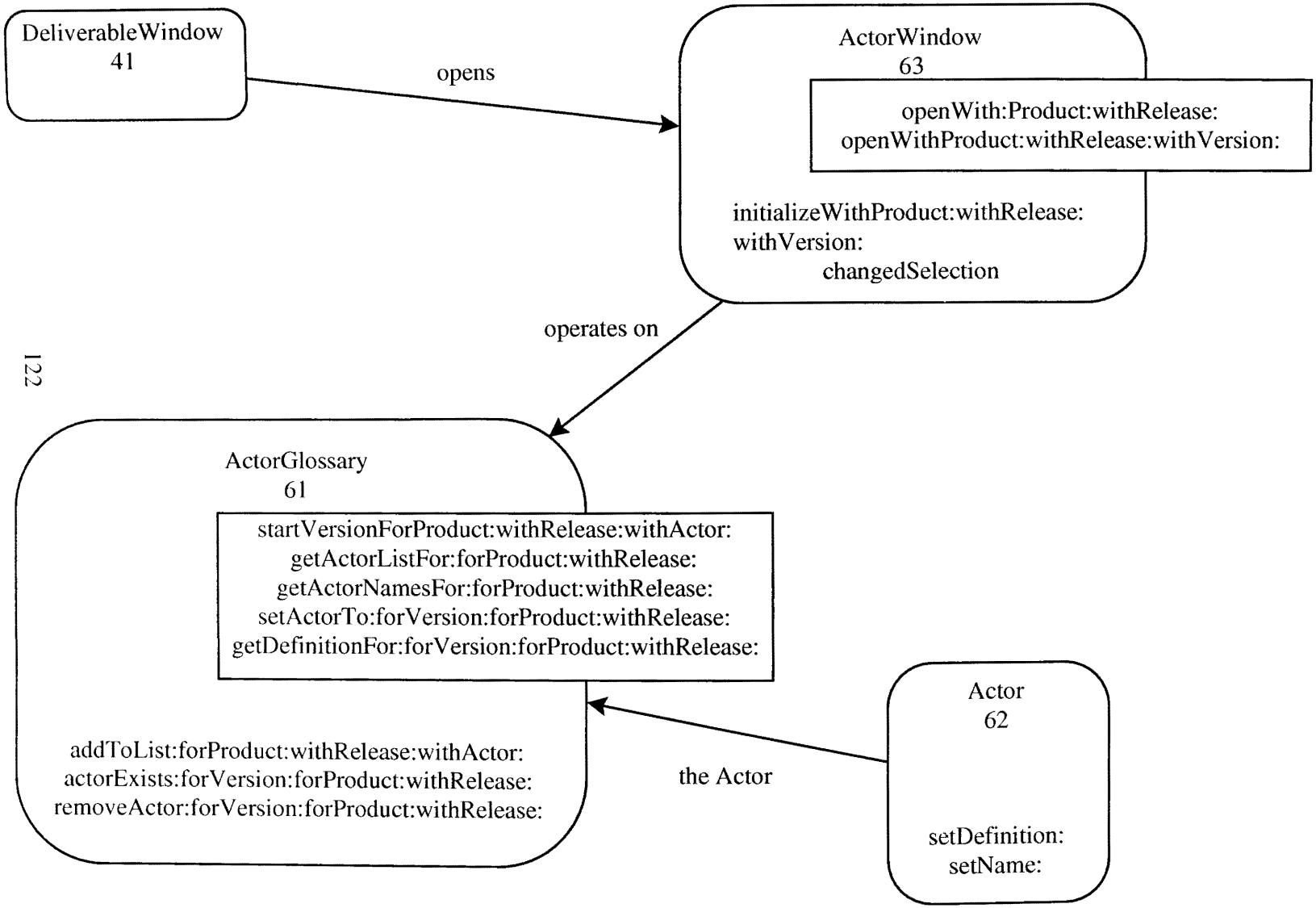

DeliverableWindow
41
opens
VerifyDeleteClass
48
openWith:andWindow:andDiagram:
initializeWith:andWindow:andDiagram:
okPushed
opens
delete
VersionInfoWindow
47
openWith:and:
okPushed
initializeWith:and:
VerifyDeleteSubsystem
32
Deliverable
49
get
information
Version
53

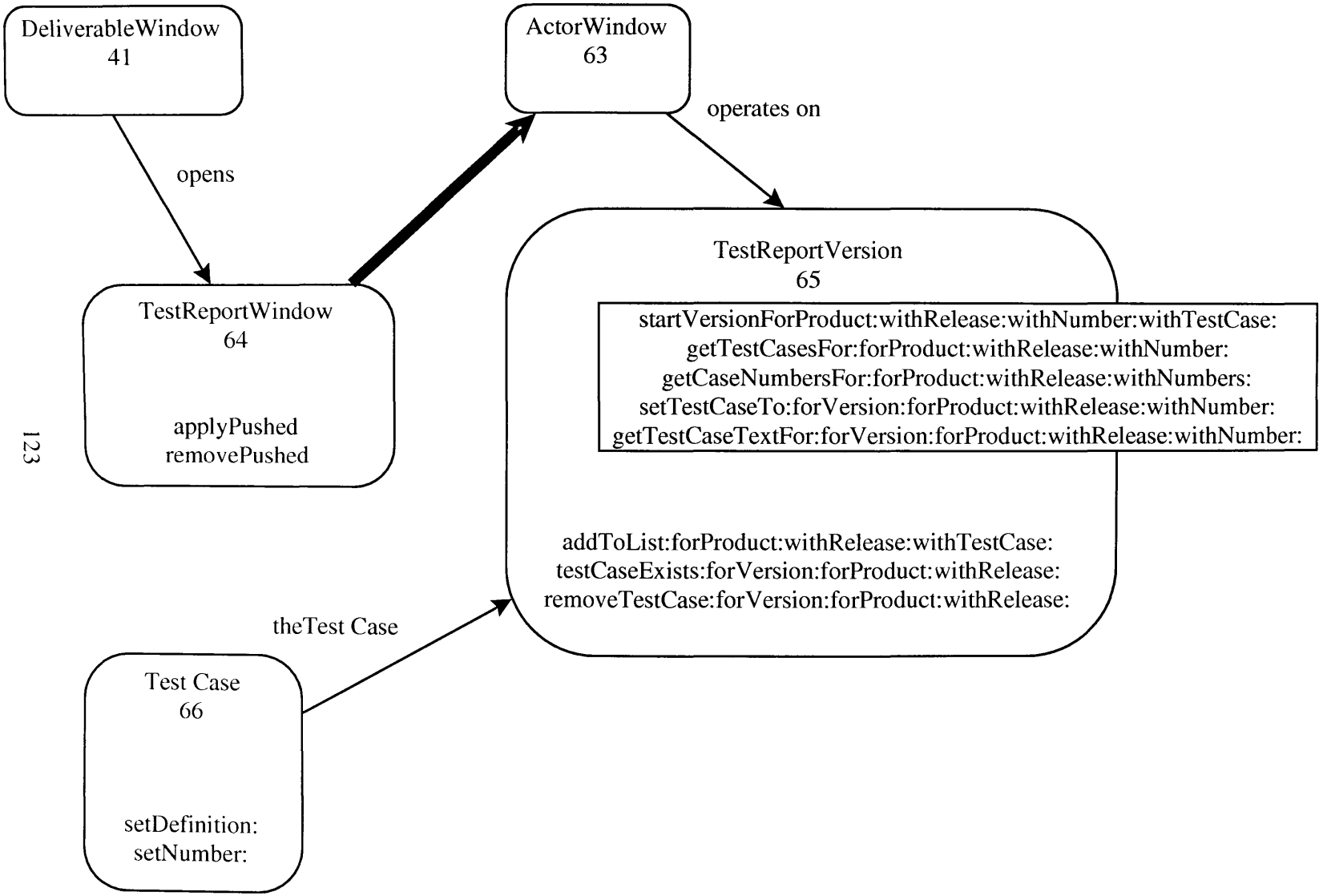
118

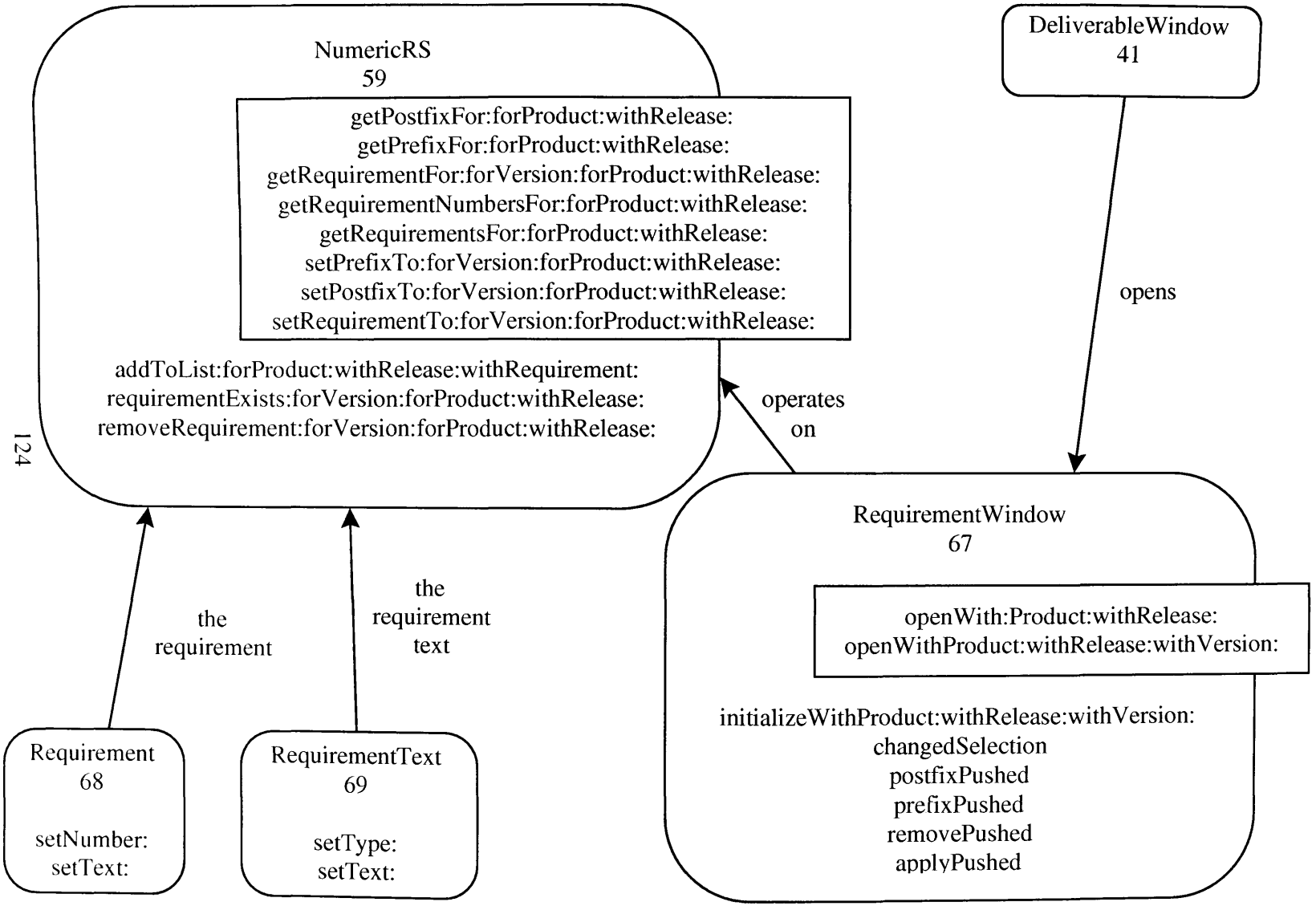

Deliverable
49
isDiagram:
isDocument:
isHierarchicalDiagram:
isMultipleDeliverable:
isNumericRequirementsSpecification:
isSingleDeliverable:
deliverableExists:
findDeliverable:forProduct:withRelease:
SingleDeliverable
50
removeDeliverable:forProduct:withRelease:
startDeliverable:forProduct:withRelease:withText:
startDeliverable:forProduct:withRelease:
assignToSubsystem:forDeliverable:forProduct:withRelease:
getSignedStatus:forProduct:withRelease:
getSubsystemFor:forProduct:withRelease:
setSignedStatusTo:forDeliverable:forProduct:withRelease:
findDeliverable:forProduct:withRelease:
removeFromList:forProduct:withRelease:
MultipleDeliverable
51
removeDeliverable:forProduct:withRelease:withNumber:
startDeliverable:forProduct:withRelease:withName:withNumber:withText:
startDeliverable:forProduct:withRelease:withName:withNumber:
assignToSubsystem:forDeliverable:forProduct:withRelease:withNumber:
getNameFor:forProduct:withRelease:withNumber:
getLastNumberFor:forProduct:withRelease:
getNumbersFor:forProduct:withRelease:
getSignedStatus:forProduct:withRelease:withNumber:
getSubsystemFor:forProduct:withRelease:withNumber:
setSignedStatusTo:forDeliverable:forProduct:withRelease:withNumber:
findDeliverable:forProduct:withRelease:withNumber:
removeFromList:forProduct:withRelease:withNumber:
HierarchicalDiagram
52
assignToClass:inDiagram:forDeliverable:forProduct:withRelease:withNumber:
getClassDiagramInFor:forProduct:withRelease:withNumber:
getClassSignedToFor:forProduct:withRelease:withNumber:
findDeliverable:forProduct:withRelease:
removeFromList:forProduct:withRelease:

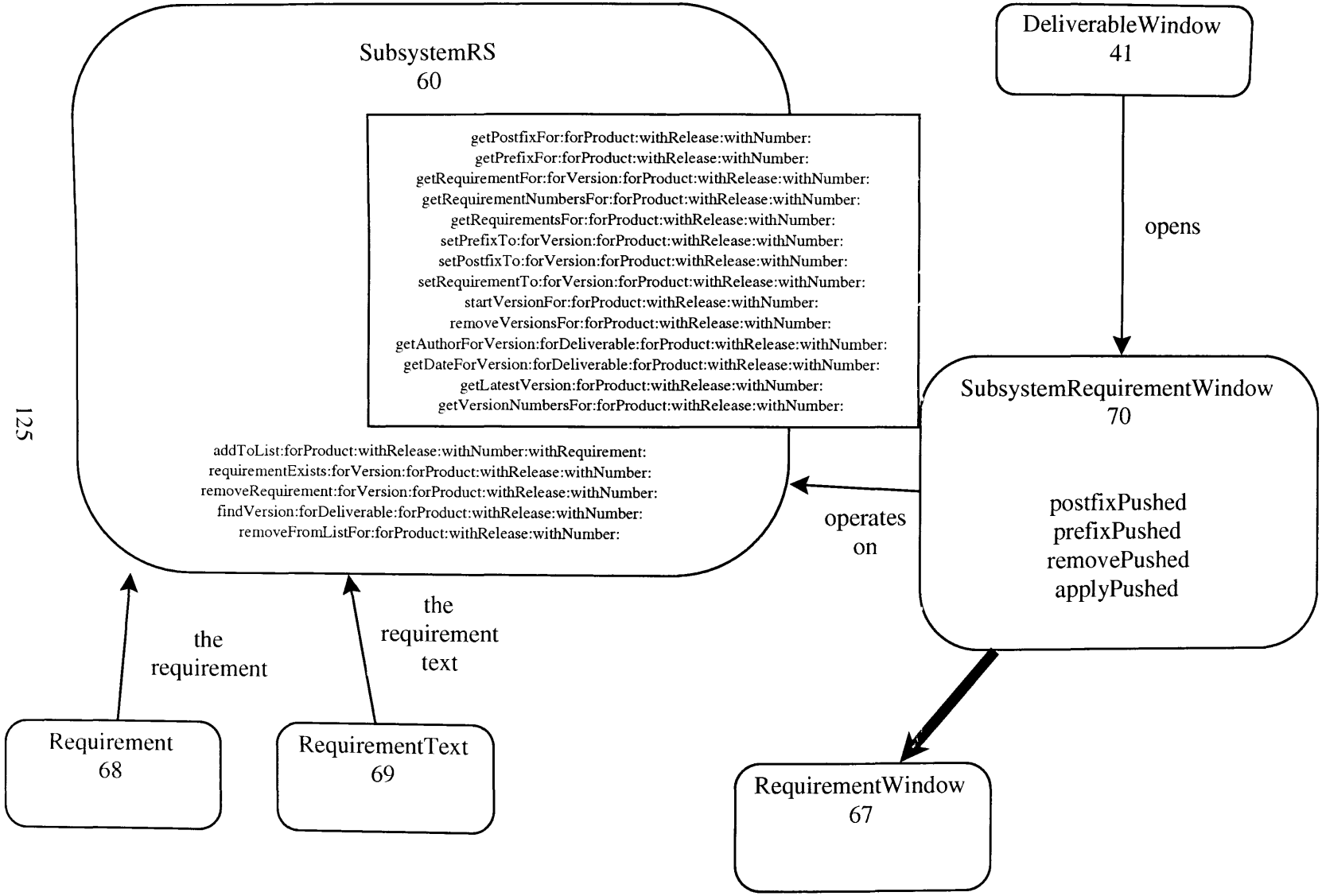

Version
53
startVersionFor:forProduct:withRelease:
removeVersionsFor:forProduct:withRelease:
getAuthorForVersion:forDeliverable:forProduct:withRelease:
getDateForVersion:forDeliverable:forProduct:withRelease:
getLatestVersionFor:forProduct:withRelease:
getVersionNumbersFor:forProduct:withRelease:
addToList:forDeliverable:forProduct:withRelease:
findVersion:forDeliverable:forProduct:withRelease:
removeFromListFor:forProduct:withRelease:
verionExistsFor:forProduct:withRelease:
MultipleDeliverableVersion
54
startVersionFor:forProduct:withRelease:withNumber:
removeVersionsFor:forProduct:withRelease:withNumber:
getAuthorForVersion:forDeliverable:forProduct:withRelease:withNumber:
getDateForVersion:forDeliverable:forProduct:withRelease:withNumber:
getLatestVersionFor:forProduct:withRelease:withNumber:
getVersionNumbersFor:forProduct:withRelease:withNumber:
addToList:forDeliverable:forProduct:withRelease:withNumber:
findVersion:forDeliverable:forProduct:withRelease:withNumber:
removeFromListFor:forProduct:withRelease:withNumber:
DiagramVersion
55
startVersionFor:forProduct:withRelease:
withNumber:withName:
DocumentVersion
56
startVersionFor:forProduct:withRelease:
withNumber:withName:withText:
getTextForVersion:forDeliverable:forProduct:
withRelease:withNumber:
addToList:forDeliverable:forProduct:with
Release:withNumber:withName:withText:
TestReportVersion
57

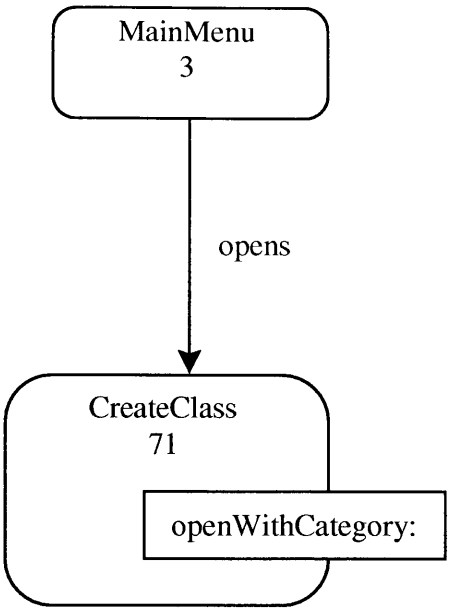

Version
53
SingleDeliverableVersion
58
startVersionFor:forProduct:withRelease:withNumber:withText:
getTextForVersion:forDeliverable:forProduct:withRelease:
addToList:forDeliverable:forProduct:withRelease:withText:
verionExistsFor:forProduct:withRelease:
NumericRS
59
ActorGlossary
61
SubsystemRS
60


DeliverableWindow
41
opens
ActorWindow
63
openWith:Product:withRelease:
openWithProduct:withRelease:withVersion:
initializeWithProduct:withRelease:
withVersion:
changedSelection
operates on
122
ActorGlossary
61
startVersionForProduct:withRelease:withActor:
getActorListFor:forProduct:withRelease:
getActorNamesFor:forProduct:withRelease:
setActorTo:forVersion:forProduct:withRelease:
getDefinitionFor:forVersion:forProduct:withRelease:
addToList:forProduct:withRelease:withActor:
actorExists:forVersion:forProduct:withRelease:
removeActor:forVersion:forProduct:withRelease:
the Actor
Actor
62
setDefinition:
setName:


DeliverableWindow
41
opens
ActorWindow
63
operates on
TestReportWindow
64
applyPushed
removePushed
TestReportVersion
65
startVersionForProduct:withRelease:withNumber:withTestCase:
getTestCasesFor:forProduct:withRelease:withNumber:
getCaseNumbersFor:forProduct:withRelease:withNumbers:
setTestCaseTo:forVersion:forProduct:withRelease:withNumber:
getTestCaseTextFor:forVersion:forProduct:withRelease:withNumber:
addToList:forProduct:withRelease:withTestCase:
testCaseExists:forVersion:forProduct:withRelease:
removeTestCase:forVersion:forProduct:withRelease:
theTest Case
Test Case
66
setDefinition:
setNumber:


NumericRS
59
getPostfixFor:forProduct:withRelease:
getPrefixFor:forProduct:withRelease:
getRequirementFor:forVersion:forProduct:withRelease:
getRequirementNumbersFor:forProduct:withRelease:
getRequirementsFor:forProduct:withRelease:
setPrefixTo:forVersion:forProduct:withRelease:
setPostfixTo:forVersion:forProduct:withRelease:
setRequirementTo:forVersion:forProduct:withRelease:
addToList:forProduct:withRelease:withRequirement:
requirementExists:forVersion:forProduct:withRelease:
removeRequirement:forVersion:forProduct:withRelease:
124
DeliverableWindow
41
opens
operates
on
RequirementWindow
67
openWith:Product:withRelease:
openWithProduct:withRelease:withVersion:
initializeWithProduct:withRelease:withVersion:
changedSelection
postfixPushed
prefixPushed
removePushed
applyPushed
the
requirement
the
requirement
text
Requirement
68
setNumber:
setText:
RequirementText
69
setType:
setText:


SubsystemRS
60
getPostfixFor:forProduct:withRelease:withNumber:
getPrefixFor:forProduct:withRelease:withNumber:
getRequirementFor:forVersion:forProduct:withRelease:withNumber:
getRequirementNumbersFor:forProduct:withRelease:withNumber:
getRequirementsFor:forProduct:withRelease:withNumber:
setPrefixTo:forVersion:forProduct:withRelease:withNumber:
setPostfixTo:forVersion:forProduct:withRelease:withNumber:
setRequirementTo:forVersion:forProduct:withRelease:withNumber:
startVersionFor:forProduct:withRelease:withNumber:
removeVersionsFor:forProduct:withRelease:withNumber:
getAuthorForVersion:forDeliverable:forProduct:withRelease:withNumber:
getDateForVersion:forDeliverable:forProduct:withRelease:withNumber:
getLatestVersion:forProduct:withRelease:withNumber:
getVersionNumbersFor:forProduct:withRelease:withNumber:
addToList:forProduct:withRelease:withNumber:withRequirement:
requirementExists:forVersion:forProduct:withRelease:withNumber:
removeRequirement:forVersion:forProduct:withRelease:withNumber:
findVersion:forDeliverable:forProduct:withRelease:withNumber:
removeFromListFor:forProduct:withRelease:withNumber:
DeliverableWindow
41
opens
SubsystemRequirementWindow
70
postfixPushed
prefixPushed
removePushed
applyPushed
operates
on
the
requirement
the
requirement
text
Requirement
68
RequirementText
69
RequirementWindow
67
125


MainMenu
3
opens
CreateClass
71
openWithCategory: