UML Drawing Tool

by

Saleh Mohamed Alshepani

B.Sc. (Computer Science)

Acadia University, 1991

Thesis

submitted in partial fulfillment of the requirements for

the Degree of Master of Science (Computer Science)

Acadia University

Fall Convocation 2000

0-612-54520-2

Table Of Contents

List Of Figures

Abstract

Object-oriented design has now become a predominant technology and there is urgent need for tools to assist developers in creating those designs. One area that needs support is object-oriented analysis and design, including the drawing of new diagrams and reengineering of existing ones. In this thesis, we describe a tool that can be used to support the drawing of several types of UML (Unified Modeling Language) diagrams for software design and their integration into a larger software development environment and reengineering of existing code. This tool is part of a large project called the Binder developed by students and faculty at Acadia University. It provides special *'page styles'* and an editor for such pages for the Binder. The UML diagrams implemented by page styles and supported by this tool (called UML Drawing Tool or UDT) are: Class and Object Diagrams, Use Case Diagram, Sequence Diagram, State Transition Diagram and Package Diagram.

UDT allows creation, editing, display and storage of UML diagrams and partially automatic conversion of class diagrams into Smalltalk source code. These diagrams can then be included in the Binder.

This thesis begins with an introduction to object-oriented methodologies followed by a short description of the UML notation and a description of the Binder program. The following chapters include the description, the design and implementation of the drawing tool. The final chapter summarizes the thesis and includes suggestions for future work.

Acknowledgements

I would like to thank my wife for her support and encouragement for making this possible. I would also like to express my sincere appreciation and gratitude to my supervisor Dr. Ivan Tomek for his help, support, and suggestion during the preparation of this thesis. I would like also to thank Dr. Rick Giles for being my internal examiner and Dr. Arthur Sedgwick for being my external examiner. I am very thankful to the staff of the School of Computer Science at Acadia University.

Last, but not least, I would like to extend my thanks to my family and friends whose help and moral support encouraged me to complete this work. A special thanks to the Libyan Educational Secretary for their sponsorship.

Chapter 1

Introduction

1.1 The need for a methodology

In developing a project, regardless of its size or purpose, everybody involved in the project should agree on and follow a methodology or combination of methodologies that is comprehensive enough to include all aspects of the project. This methodology, a body of methods with a set of rules and assumptions, should be flexible enough to match the uniqueness of each project [Networld 1999].

The popularity and effectiveness of disciplined object-oriented analysis and design is clearly shown by the emergence of several competing methodologies. Each of these methodologies has strengths and weaknesses, and the choice of which one to follow depends to a large extent on the type of organization and business involved. The most widely spread methodologies [Technology 1997] are described in the Object Modeling Technique (OMT) [Rumbaugh 1991], the Booch Method [Booch 1993, 1994], Object-Oriented Software Engineering (OOSE) [Jacobson 1992], Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) [Coad 1991a, 1991b], Fusion Method [Coleman 1994], Designing Object-Oriented Software (DOOS) [Wirfs-Brock 1990] Object Oriented Analysis and Design (OOAD) by Martin and Odell [Martin 1993] and Object Lifecycles (OL) by Shlaer and Mellor [Shlaer 1992].

The choice of which methodology to use can be decided by considering at least the following selection criteria:

- The methodology is suitable for the application requirements.
- It covers all software lifecycle phases.
- It fits the programming language and more generally the program development environment.
- The developers have experience with it or can acquire sufficient knowledge about it.
- The methodology is widely supported. The support could be tools for creating models, technical help, or a mentor.
- It is easy to use and understand.

1.2 Object Oriented Methodologies

The first references to object-oriented methodologies first appeared in the late 1980s [Booch 1995]. After that there was an explosion of object-oriented methods as various methodologists experimented with different approaches to object-oriented analysis and design [Booch 1995]. Experience with these methods grew, accompanied by a growing maturation of the field as a whole as more and more projects applied these ideas to the development of production-quality mission-critical systems. By the mid 1990s a few second-generation methods [Demmer 1997], borrowing from other methodologies, began to appear, most notably the Booch method [Booch 1994] which replaced the Booch 91 version, BON [Waldén 1995], Firesmith [Firesmith 1993] and Fusion [Coleman 1994]. Because both Booch and OMT methods were independently growing together and were collectively recognized as the most dominant methods world-wide, the two authors, Booch and Rumbaugh, started working together when Rumbaugh joined the Rational Software Corporation in October 1994. Their goal was to unify at least the various notations used in different methods in what was originally called the Unified Method (UM) and has since become the UML - Unified Modeling Language [Demmer 1997]. Later in 1995, Jacobson, the author of OOSE, joined Rational and the three authors expanded the scope of UML to cover the needs of OOSE notations.

We summarize the three most influential methodologies in the order Booch method, OMT and OOSE and illustrate the symbols of their notations in the remaining sections of this chapter. For each method, we will give a description of the method followed by the graphical representation of the symbols of the method's notation.

1.2.1 Booch Method

The Booch method [Booch 1993, 1994] is a widely used OO-method for designing systems using the object paradigm. It is one of the earliest recognizable object-oriented design methods and covers the analysis and design phases of an OO-system. The Booch method defines many symbols for documenting almost every design decision and perspective and most developers using this method will never use all of its symbols and diagrams. A designer will usually start with class and object diagrams in the analysis phase and refine them in a series of steps.

Booch uses four models to describe an OO system: **logical** and **physical** structure, and its **static** and **dynamic** semantics. Figure 1-1 shows the models of the Booch method. The **logical model** (problem domain) is represented in the class and object structure. In the **class diagram**, the architecture (**static model**) is constructed. The **class diagram** shows existing classes and relationships among them including cardinalities, concurrency and visibility aspects. The **object diagram** shows the existing objects and the relationships among them, including visibility and synchronization aspects. The **physical model** is represented in the **module** and **process** architectures. The **module** and **process** architecture describes the physical allocation of classes and objects to modules and processes. It deals with the association of concrete hardware with the software components of a system. **Model diagrams** show the physical packaging of classes and objects into modules and **Process diagrams** show the allocation of processes to processors.

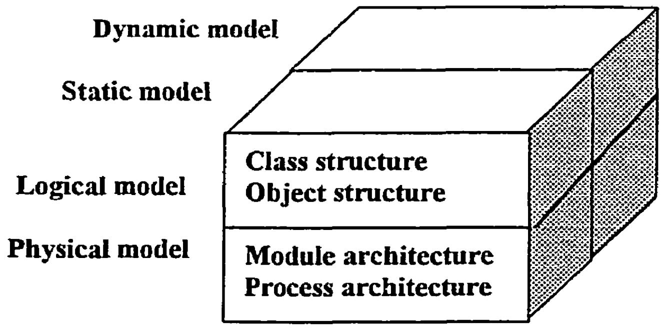


Figure 1-1: Models of the Booch Method

1.2.1.1 The Development Process

In addition to providing models, the Booch method defines a development process. It supports the iterative and incremental development of a system in the analysis and design phases. The view of the development process is divided into a macro and micro processes.

1.2.1.1.1 Macro development process

This process is the controlling framework for the micro process. It allows for the improvement of the micro process. It is designed to support the incremental development of the system and represents the activities of the entire development team. It requires the following activities:

- **Establish core requirements** – In this phase, the vision for the general requirements ideas is established for some application and its assumptions are validated. An idea springs fourth a new business venture, complimentary products and set of features for an existing product.
- **Develop a model of the desired behavior for the system** – In this phase, the classes and objects that form the vocabulary of the problem domain are identified and the system's behavior is emphasized. This phase consists of domain analysis and scenario planning. In domain analysis, classes and objects that are common to a particular problem domain are identified. In scenario planning, the primary function points are identified and scenarios are documented. State machines for classes are developed where life-cycles are clear.
- **Create an architecture** – In this phase, an architecture is created for the evolving implementation and common tactical policies are established. This phase consists of architectural planning, tactical design and release planning. In architectural planning, the aim is to create very early in the life cycle a domain-specific application framework that can be successively refined. In tactical design, decisions are made about the common policies. In release planning, a

formal development plan is yielded for identifying the stream of architectural releases, team tasks and risk assessments.

- **Evolve the implementation** – In this phase, the growth and change in the implementation through successive refinement is established until the production system is reached. This phase consists of application of the micro process and change management. Application of the micro process starts with an analysis of the requirements for the next release, after which it leads to the design of an architecture, and then classes and objects are invented that are necessary to implement this design. The main product is a stream of executable releases representing successive refinements to the first release of the architecture. Change management attempts to recognize the incremental and iterative character of the object-oriented system. It is possible to change the class hierarchies and protocols, or mechanisms as long as it is not a threat for the strategic architecture and the development team.
- **Manage post-delivery evolution** – This phase is mainly a continuation of evolution by making more localized changes to the system as new requirements are added and bugs are being eliminated.

1.2.1.1.2 Micro development process

The micro process describes the day-to-day activities by a single developer or group of software developers and tracks the following activities:

- **Identify classes and objects at a given level of abstraction** - Classes and objects are identified by finding the significant classes and objects in the problem space. The result is a data dictionary of candidate classes and objects and a document describing object behavior.

- **Identify their semantics** - The aim is to establish the state and behavior of each abstraction identified in the previous phase. Semantics are represented in a top-down way in and, where it concerns system function points, strategic issues are addressed. Also commonality in patterns of behavior are discovered, because it may contribute to reusability.
- **Identify their relationships** – In this phase, the boundaries of each abstraction are solidified and co-operating classes and objects are identified. This phase consists of specifying associations, identifying various collaborations and refining associations. The identification of associations results in a class diagram. The identification of collaborations results in object and module diagrams. The refinement of associations results in a more specified description of semantics and relationships.
- **Specify the interface and the implementation of these classes and objects** - This phase consists of the selection of the structures and algorithms that provide the semantics of the earlier identified abstractions. The first three phases of the micro process discuss the outside view of abstractions; this final step focuses on their inside view. This results in artifacts capturing representational issues of each abstraction and their mapping to the physical model.

1.2.1.2 Graphical Notation

This section shows the graphical notation for the elements of the following Booch diagrams: the **class diagram**, the **object diagram**, **state transition diagram**, **interaction diagram**, **module diagram** and **process diagram**.

Figures 1-2 and 1-3 show all the elements and relations for representing Booch **class diagrams**. Figure 1-2 shows the elements of the class diagram with the graphical representations of the following icons: class, class utility, class category (a logical collection of classes), parameterized class, instantiated class, metaclass (a class whose instances are themselves classes) and class nesting. These icons represent the nodes in Booch class diagrams.

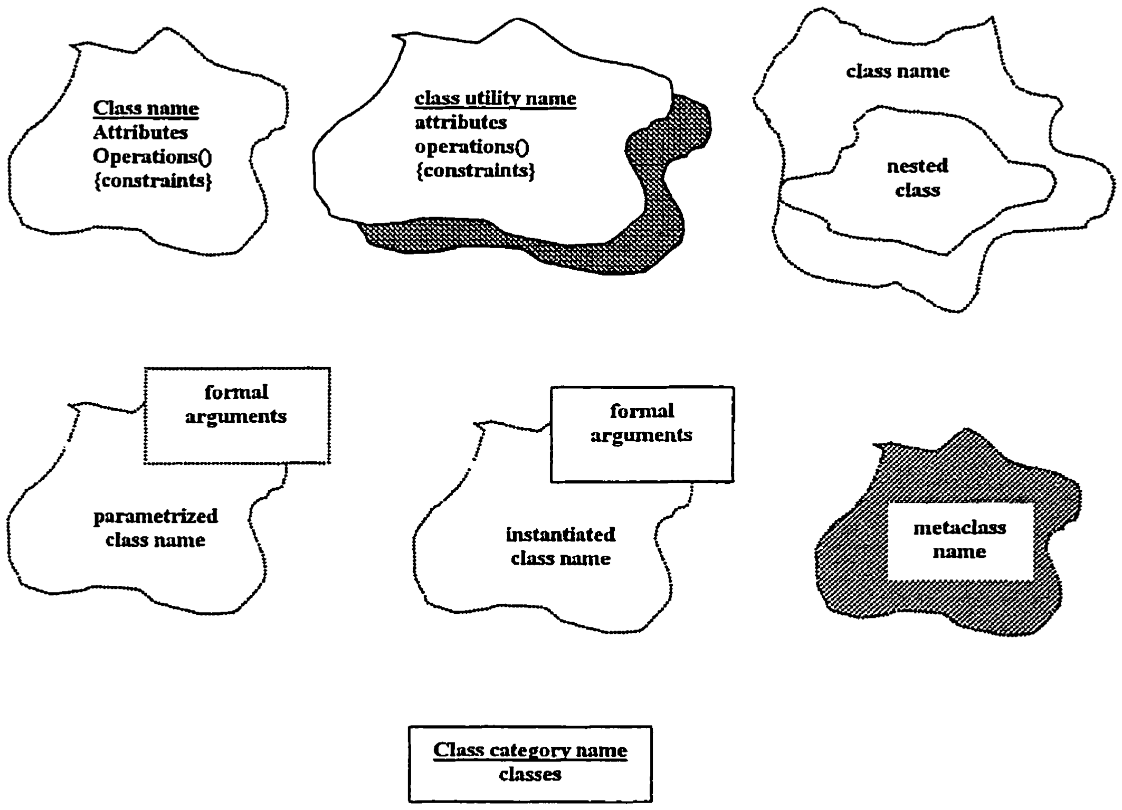


Figure 1-2: Class Icons

Figure 1-3 shows how notes are represented graphically using Booch notation. Notes are attached to the above icons to add documetation to them.

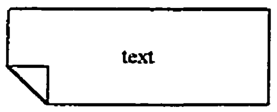


Figure 1-3: Notes

Figure 1-4 shows realtionships represented as edges in a Booch **class diagram**. They clarify the kinds of relationships between classes. The association relation can have a label, a cardinality or a role or a combination of any of them. The has relation can be by reference or value.

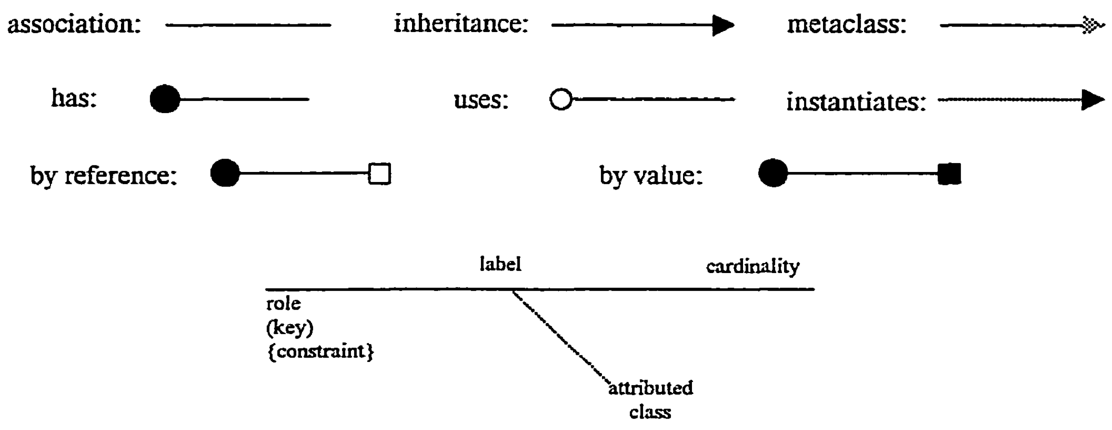


Figure 1-4: Relationships in Booch Class Diagram

Figures 1-5 through 1-6 show the elements and relations used in Booch **object diagrams**. Figure 1-5 shows the graphical representation of objects using the Booch method.

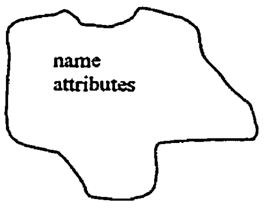


Figure 1-5: Graphical representation of objects in the Booch method

Figure 1-6 shows relationships represented as edges in Booch **object diagrams**. Relationships are used for showing the kind of relationship between objects. Synchronous means wait forever until message is accepted. Timeout means wait for a specified amount of time then abandon if message is not serviced. Asynchronous means 'queue the message and proceed without waiting'.

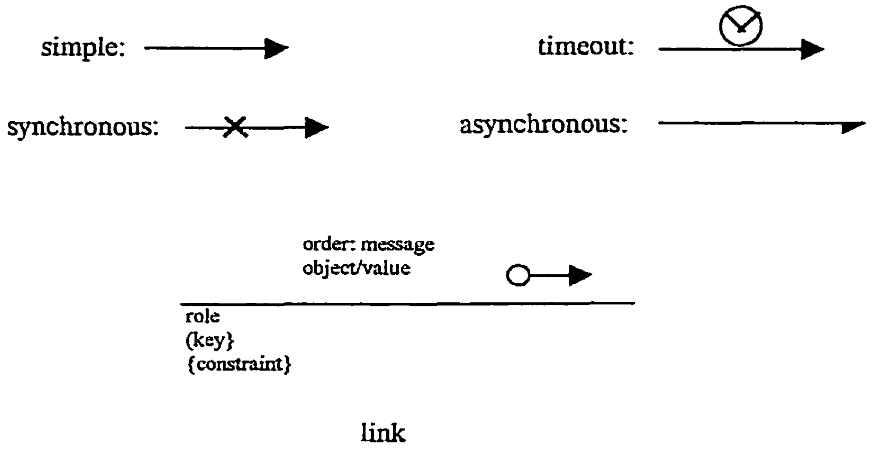


Figure 1-6: Relationships in Booch Object Diagrams

Figure 1-7 shows the elements and relations of the **state transition diagram**. It shows how states, superstate, start state, stop state and history icons are represented graphically. States are connected using directed lines. **State transition diagrams** show the states of an object, the events that cause transitions and actions resulting from transitions.

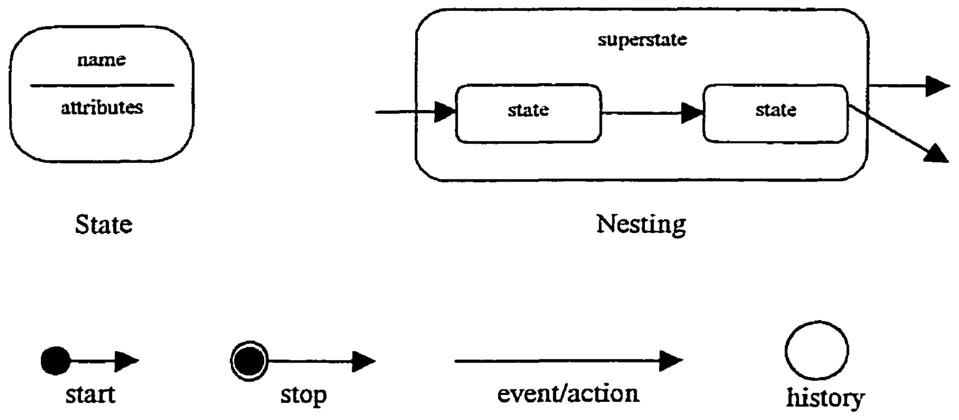


Figure 1-7: Booch State Transition Diagram

Figure 1-8 shows the representation of **interaction diagrams** using the Booch method. Objects are represented by dotted vertical lines with a rectangle showing the life of that object. Interaction diagrams describe how scenarios are executed in the same context as an object diagram but they show the dynamic aspects not the static aspects.

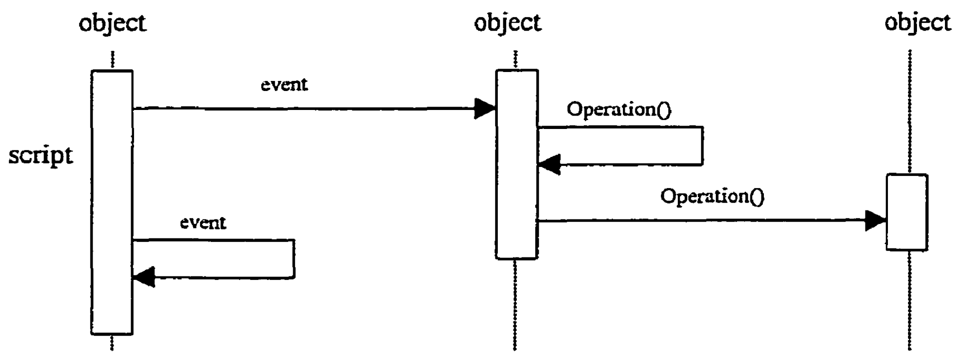


Figure 1-8: Booch Interaction Diagram

Figure 1-9 shows the representation of the **module diagram** in the Booch method. These are the icons for the different components of the module. They are connected using a directed line.

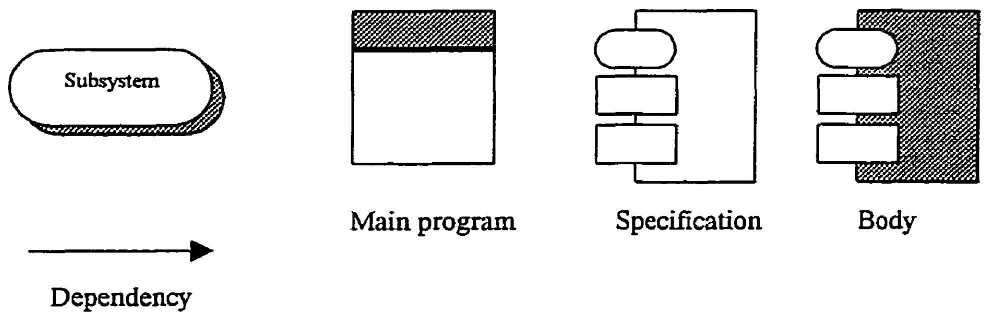


Figure 1-9: Booch Module Diagram

Figure 1-10 shows the elements and relations of the **process diagram**. These icons represent the physical nodes of the **process diagram**. They are connected using a solid line.

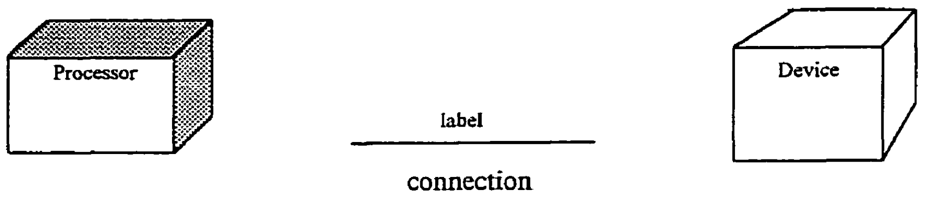


Figure 1-10: Booch Process Diagram

1.2.2 Object Modeling Technique (OMT)

Object Modeling Technique (OMT) [Rumbaugh 1991] is an object-oriented software development methodology that extends from analysis through design to implementation, as shown in Figure 1-11. It starts by building the analysis model to abstract essential aspects of the application domain. After that, design decisions are made and more details are added to the model. Finally, the design model is implemented using a programming language, a database and hardware.

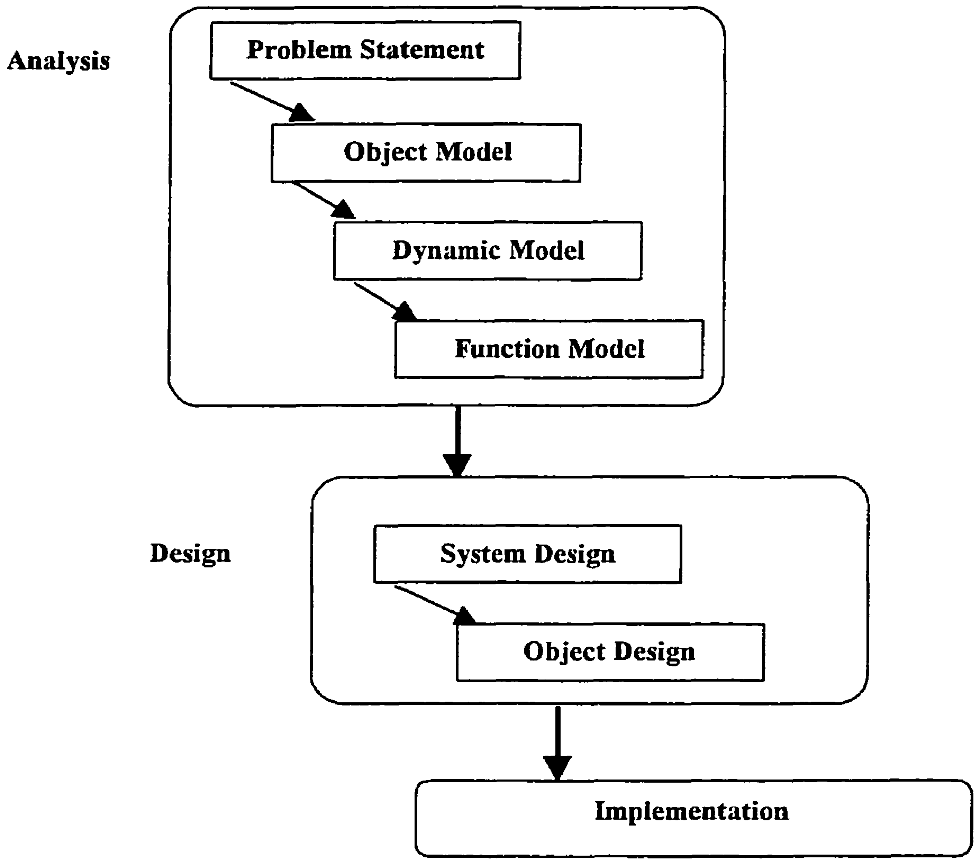


Figure 1-11: OMT : Process

1.2.2.1 Analysis

In the analysis phase, the development team writes or obtains an initial description of the problem statement. After that three models are created. They are the **object model**, the **dynamic model** and the **functional model**.

The **object model** is a description of the structure of the objects in a system including their identity, relationships, attributes and operations. Building an object model requires identifying object classes, starting a data dictionary, adding associations between classes and attributes for objects and links, organizing object classes using inheritance, testing access paths using scenarios, and grouping classes into modules. The data dictionary describes classes, attributes and association. The object model is represented graphically using object diagrams. Figures 1-12 through 1-14 show the elements and relations for representing object diagrams.

The **dynamic model** is a description of aspects of a system concerned with control including time, sequencing of operations and interaction of objects. Developing a dynamic model requires preparing scenarios, identifying events capturing interactions between objects and developing a state diagram for each class that has important dynamic behavior. The dynamic model is represented graphically using state diagrams. Figures 1-15 and 1-16 show the elements and relations for representing state diagrams.

The **functional model** is a description of those aspects of a system that transform values using functions, constraints and functional dependencies. It describes how output values in a computation are derived from input values. Constructing a functional model will require identifying input and output values, using data flow diagrams, describing what each function does and identifying constraints. Data flow diagrams show functional dependencies between values and the computation of output values from input values. The function model is represented graphically using data flow diagrams. Figures 1-17 and 1-18 show the elements and relations for representing data flow diagrams.

1.2.2.2 System Design

System design is the first stage of design. At this stage, high-level decisions are made about the overall structure of the system. The system design phase requires organizing the system into subsystems, identifying concurrency inherent in the problem, allocating subsystems to processors and tasks, choosing the basic strategy for implementing data stores, identifying global resources, choosing an approach to implementing software control and considering boundary conditions.

1.2.2.3 Object Design

In the object design stage, a shift from the real-world orientation of the analysis model towards software perspective is required for a practical implementation. The object design phase will require obtaining operations for the object model, designing algorithms to implement operations, optimizing access paths to data, adjusting class structure to take advantage of inheritance, designing implementation of associations and organizing classes into modules.

1.2.2.4 Implementation

In this phase, the established design is translated into code using the selected programming language.

1.2.2.5 Graphical Notation

This section shows the graphical notation for the elements of the following OMT models: the **object model**, the **dynamic model** and the **functional model**.

Figure 1-12 shows the graphical representation of a **class** in OMT. Classes are the main components for representing the object model graphically.

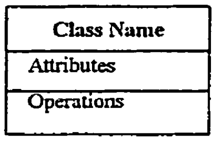


Figure 1-12: Graphical representation of Classes in OMT

Cardinality is used to show how classes are related in the aspect of one-to-one, one-to-many and other relationships. Figure 1-13 shows how cardinality is represented in OMT.

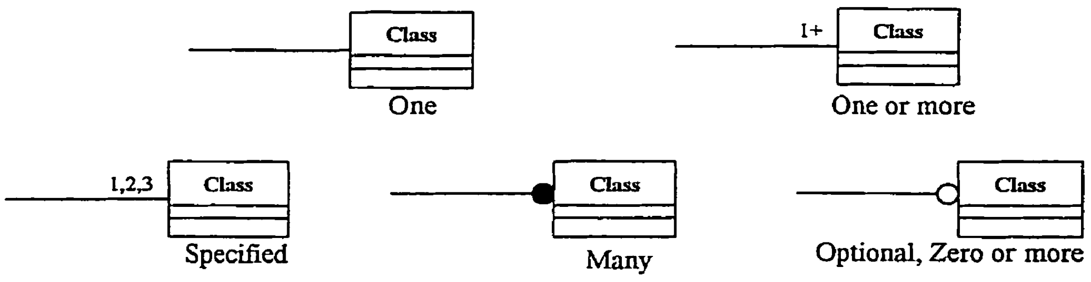


Figure 1-13: Graphical representation of Cardinality in OMT

Figure 1-14 shows the graphical representation of relations in OMT. Relationships are used to show how classes are related. They are the edges of the Object model diagram.

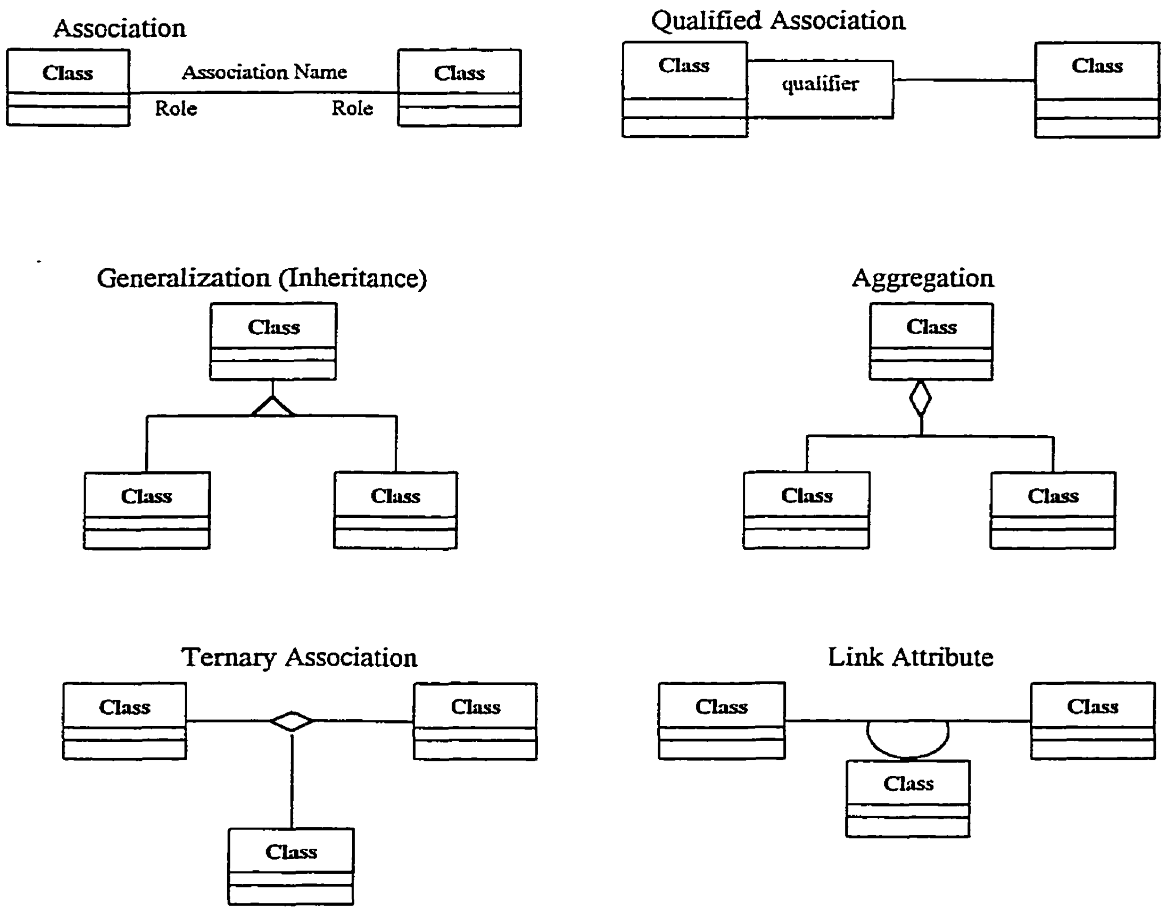


Figure 1-14: Relations in OMT Object diagram

Figure 1-15 shows the graphical representation of **states**, **start states** and **stop states** in OMT. These are used for constructing the **state diagrams**, which represent the dynamic model graphically. States are connected using solid lines. **Entry/exit** actions are actions that will take place when a state is entered/left. An **activity** is an operation with side effects on objects, which has duration in time.

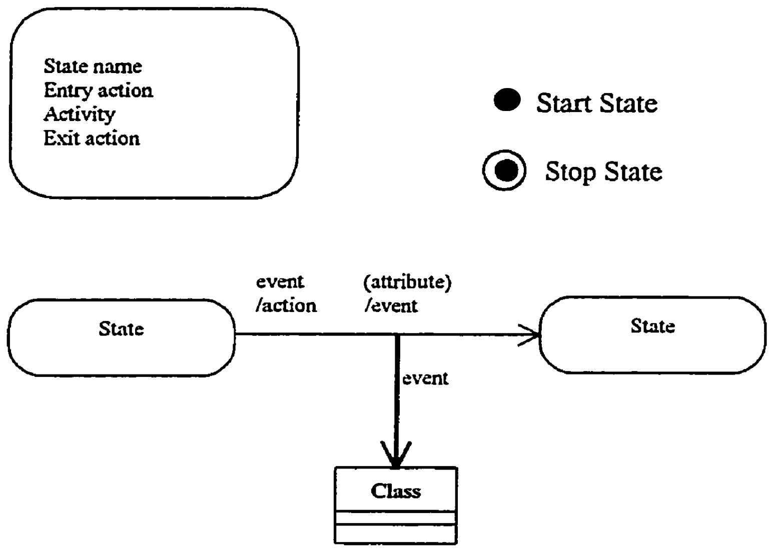
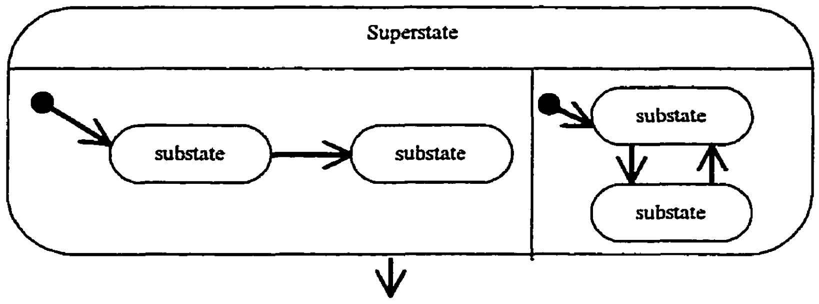
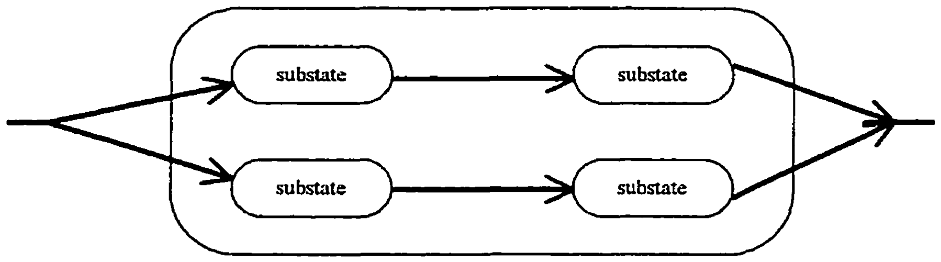


Figure 1-15: Graphical representation of states in OMT

Figure 1-16 shows state nesting and the splitting and synchronization of control in substates.



State nesting



Splitting and Synchronization of control

Figure 1-16: States nesting in OMT

Figure 1-17 shows the graphical representation of **actors, data stores** and **files**. It also shows how **processes** are represented. All these icons are used for constructing data flow diagrams. An **actor** is an active object that drives the data flow graph by producing or consuming data values. A **data store** is a passive object that stores data for later access. A **process** is something that transforms data values.

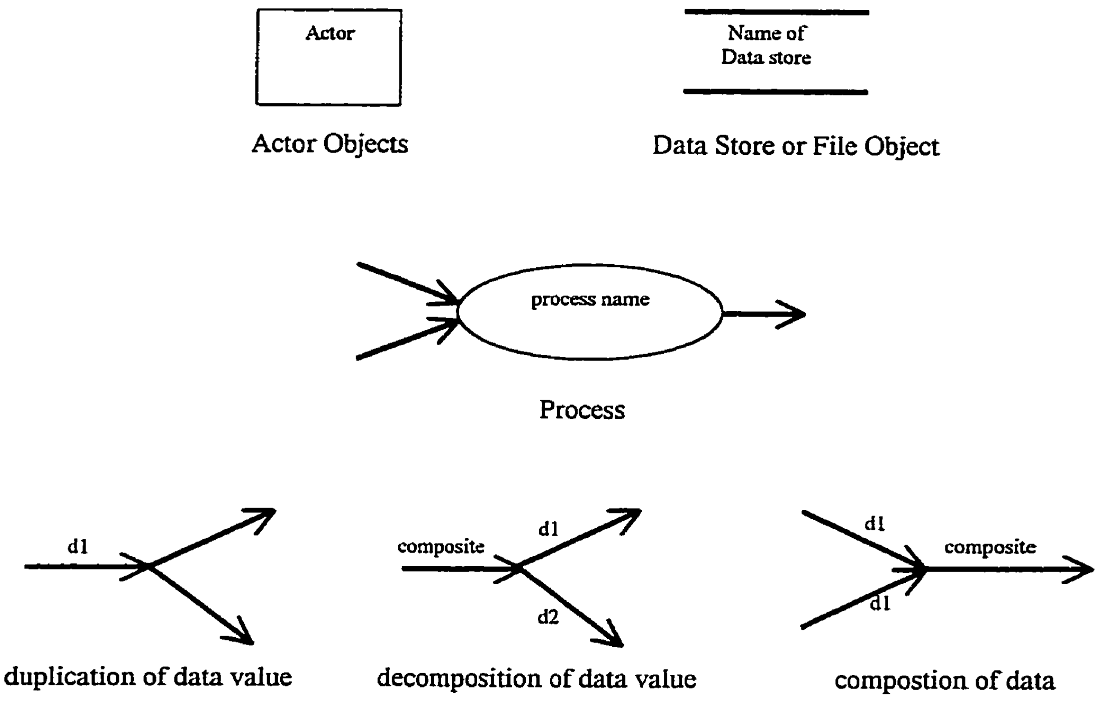


Figure 1-17: Processes, Actors, Data Store and File Objects in OMT

Figure 1-18 shows how **data flow** and **control flow** between processes is represented graphically. **Data/Control flow** is a connection from output of an object or process to input from another object or process.

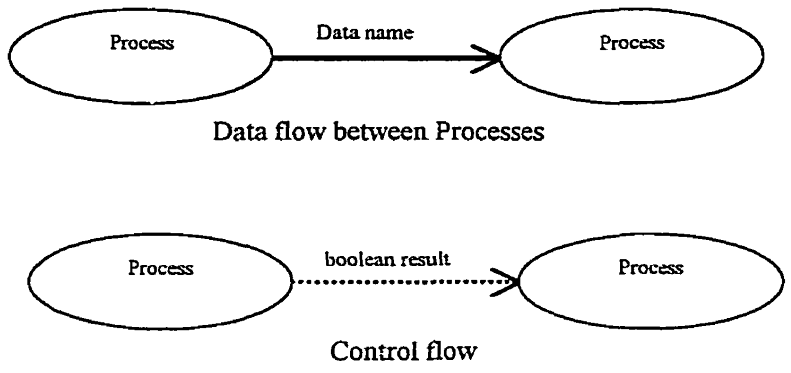


Figure 1-18: Data and Control flow between Processes

1.2.3 Object-Oriented Software Engineering (OOSE)

OOSE [Jacobson 1992] is another popular object-oriented development methodology. It was specifically designed to be used for the development of large real-time systems. It uses "use-cases" for most phases of development, including analysis, design, validation and testing. A use case is a complete course of events specifying interaction between the user and the system. Use cases are initiated by actors. They describe the flow of events that involve these actors. Actors are the things that interact with use cases such as human users, external hardware or other systems.

The process recommended by OOSE for the development of OO systems is summarized in the diagram shown in Figure 1-19. In the following sections we will outline the OOSE process and then show the notation that it uses.

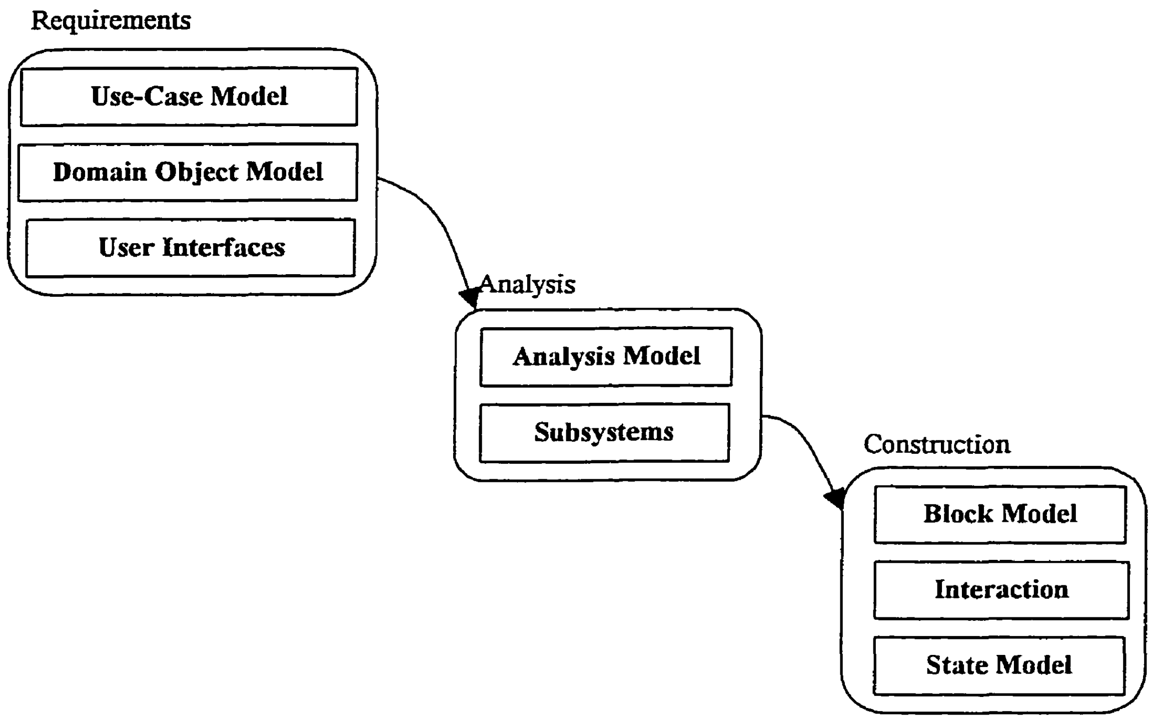


Figure 1-19: OOSE Models

1.2.3.1 Requirements

In the requirements phase, the functionality of the system is defined. This phase consists of developing a use case model, a domain object model and user interfaces. The use case model describes actors and use cases that specify all the interactions between the user and the system. Actors define the roles that users or external entities play in exchanging information with the system. The domain object model represents a logical view of the system to support specifying the use cases. User interfaces compliment use cases by showing what the system looks like when executing these use cases.

1.2.3.2 Analysis

The analysis model structures the system by modeling interface objects, entity objects and control objects. It provides a foundation for the design. In this model subsystems are defined. Subsystems group related objects and may include further subsystems.

1.2.3.3 Construction

This phase consists of the construction of design and implementation models. The design model refines the analysis models with regard to the selected implementation environment. Blocks, groups of design objects, are used to describe system implementation. The state model is developed for individual objects within blocks. The interaction model is used for showing inter-object messages and stimuli for the use cases. The implementation model consists of the source code implementing the blocks.

1.2.3.4 The Graphical Notation

This section shows the graphical notation for the components of the following OOSE models and diagrams: the **domain object model**, the **design model**, the **analysis model**, the **use case model**, the **state transition graph** and the **interaction diagram**.

Figure 1-20 shows the graphical representation of objects in OOSE.

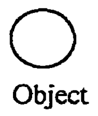


Figure 1-20: OOSE Domain Object Model

Figure 1-21 shows how blocks, which are used to describe system implementation, are represented graphically in OOSE.

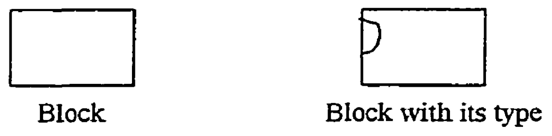


Figure 1-21: OOSE Design Model

Figure 1-22 shows the elements and relations of the OOSE **analysis model**. An entity object is an object that holds information for a long time, even when a use case is completed. An interface object is an object that contains functionality of a use case that interacts directly with the environment. A control object is an object that models functionality that is not in any other object (e.g. calculating taxes using several different criteria). An attribute contains information of some type.

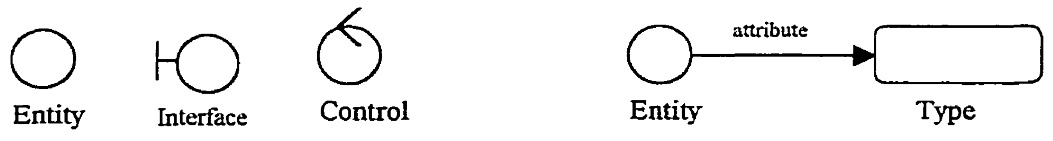


Figure 1-22: OOSE Analysis model

Figure 1-23 shows the elements of an OOSE **use case model**. The model includes the graphical representation of actors, use cases and the relationships between them.

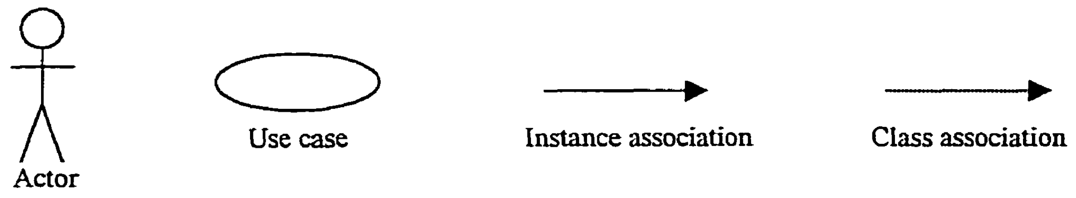


Figure 1-23: OOSE Use case model

Figure 1-24 shows components of an OOSE **state transition graph**. It shows how different types of nodes of the state transition graph are represented graphically in OOSE. A signal is an inter-process stimulus: it is sent between two processes. A message is an intra-process stimulus: a normal call inside one process.

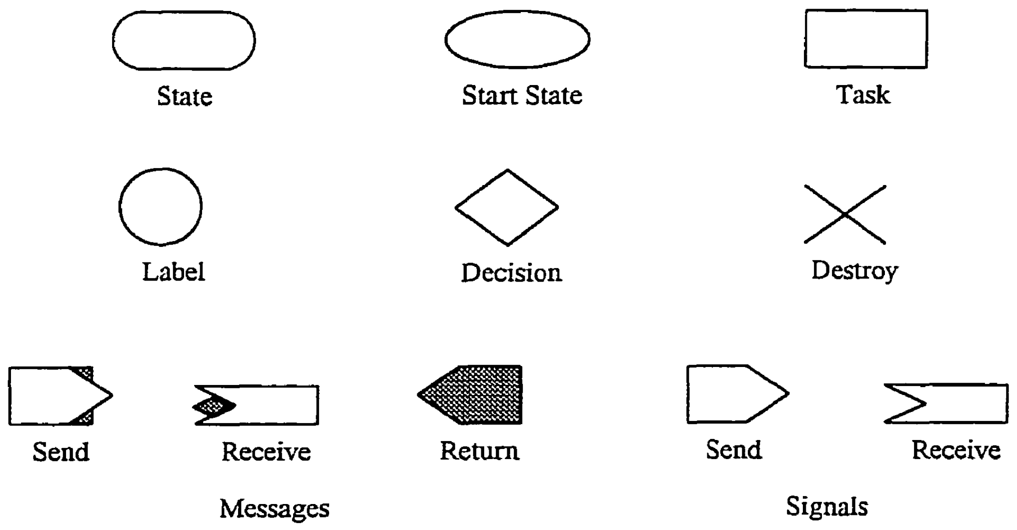


Figure 1-24: OOSE State transition graph

Figure 1-25 shows components of an OOSE **interaction diagram.**

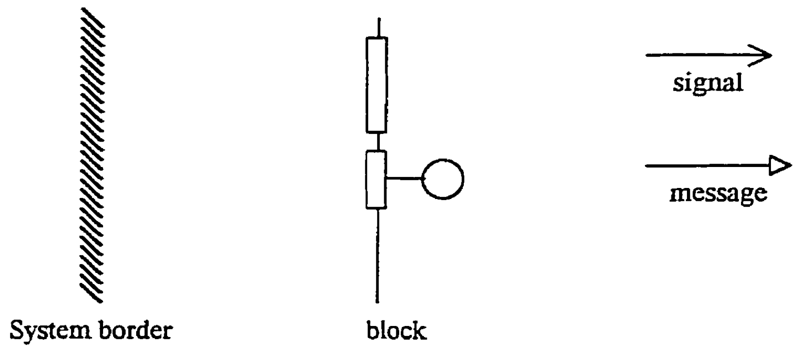


Figure 1-25: OOSE Interaction diagram

1.3 Conclusion

There are many more methodologies for the development of a software project, but the three represented above have been most influential. The decision as to which one is appropriate is more or less a matter of personal taste and design culture. However, it appears certain that those who use these techniques to the best of their abilities will be able to write better projects than those who do not use them. Learning analysis and design methodologies is a very good investment that will serve the learner well for the rest of his or her life in developing any project. Following a methodology builds a group skill and intelligence and reduces dependence on each individual.

Although the process of developing a project may differ from one team to another, the components of the various artifacts, such as classes, events, and packages, are shared by all methodologies. This is reflected in the sets of symbols used by Booch, Rumbaugh and Jacobson. This suggests that a unified representation of concepts would make development easier. This recognition led to the proposal of several unifying notations, which are the subject of the next chapter.

Chapter 2

Unified Modeling Language (UML)

2.1 Introduction

The similarities and differences of notations used by different OO methodologies led to several attempts to a unified standard notation including the Unified Modeling Language (UML) [Rational 2000] and OPEN Modeling Language (OML) [Firesmith 1998]. The most dominant and successful of these is the Unified Modeling Language (UML). This chapter introduces the elements of UML and few illustrative examples. We will give more examples in subsequent chapters. We will also include a brief description of OML later in this chapter.

UML is a standard graphical language for visualizing, specifying, constructing and documenting the artifacts of a software system. It provides a standard notation for expressing a system's blueprint [Booch 1999] – it is not a methodology.

The work on UML started in 1994 when J. Rumbaugh, the author of OMT (Object Modeling Technique), joined G. Booch, the author of Booch method, as a partner at Rational Software Corporation [Rational 2000]. Their aim was to unify their methods and the first version of the Unified Method was released in October 1995 [Booch 1995]. In the same year, I. Jacobson, the author of OOSE (Object-Oriented Software Engineering), joined Booch and Rumbaugh at Rational and the three authors expanded the scope of UML to include OOSE. Their work led to the release of the next version of UML in 1996 [Booch 1996]. After getting feedback from the software engineering community and with the support of several organizations, another version of UML was released in 1997 [Booch 1997]. This version was accepted by the Object Management Group (OMG) in 1997 [Booch 1999] as the standard for expressing analysis and design of software products developed by the object oriented approach.

2.2 Basic UML Building Blocks

Figure 2-1 shows that there are three kinds of building blocks in UML. They are: things, relationships and diagrams. We will explain them in the following sections.

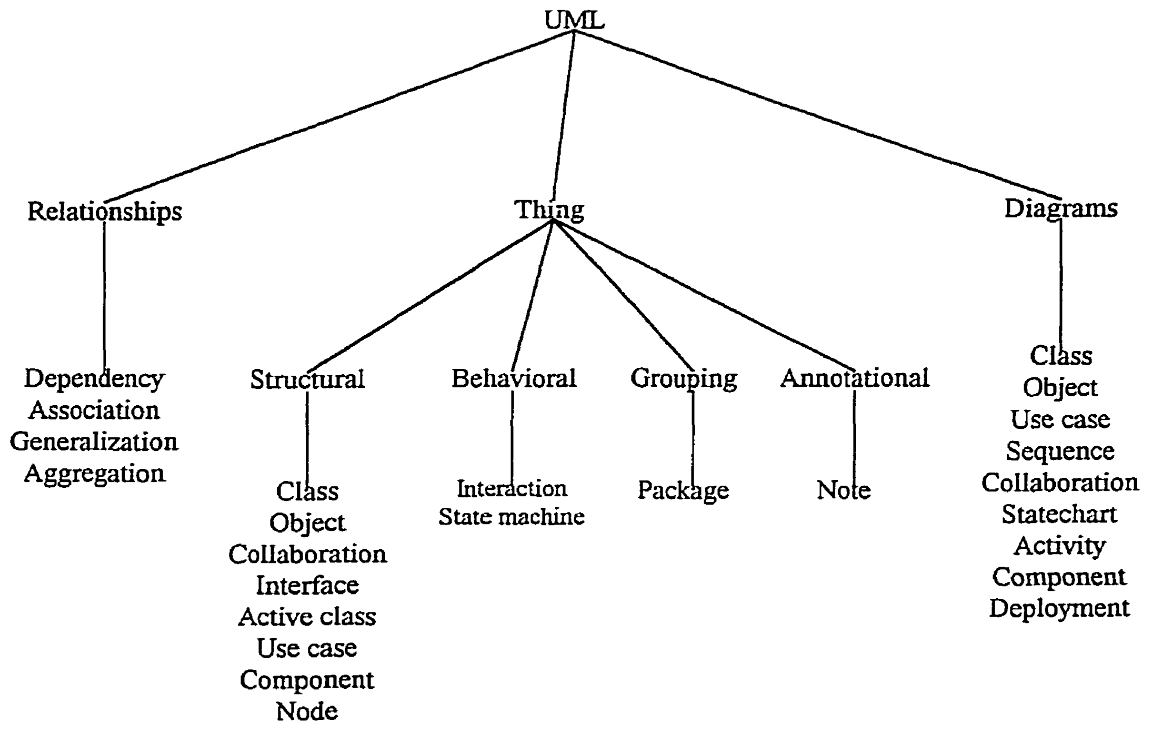


Figure 2-1: Basic Building Blocks (vocabulary) of UML

2.2.1 Things

Things are the basic object-oriented building blocks of the UML. There are four kinds of things: structural things, behavioral things, grouping things and annotational things [Booch 1999].

2.2.1.1 Structural Things

Structural Things represent the conceptual or physical elements in a model. There are seven kinds of structural things in UML: a class, an object, an interface, a collaboration, a use case, an active class, a component and a node.

A **class** describes a set of objects that share the same attributes, operations and relations with objects from other classes. A class is represented graphically by a rectangle that is divided into up to three compartments, the name compartment, the attributes compartment and the operations compartment, as shown in Figure 2-2(a). Only the top compartment is required. Figure 2-2(b) shows an example representing class **Shape** with attributes corner, color, width, origin and name and operations draw, displayOn: and kind.

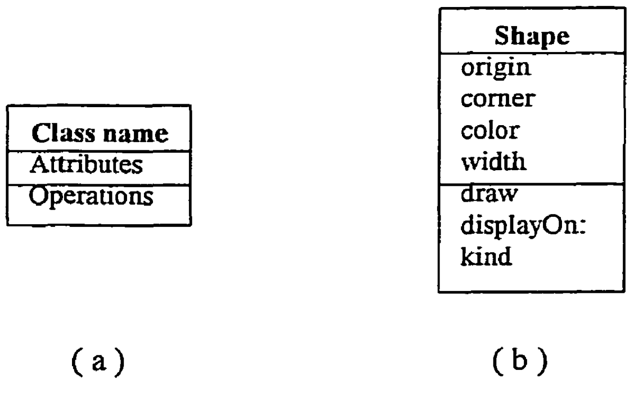


Figure 2-2: Graphical representation of a class in UML – definition (a) and an example (b)

An **object** is an instance of a class. It is present at execution time and allocates memory for its instance variables. An object is graphically represented by a rectangle with an underlined object name, as shown in Figure 2-3(a). An object name may be an object name, the class name preceded by a colon, or both the object name and the class name separated by a colon, as shown in the example in Figure 2-3(b).

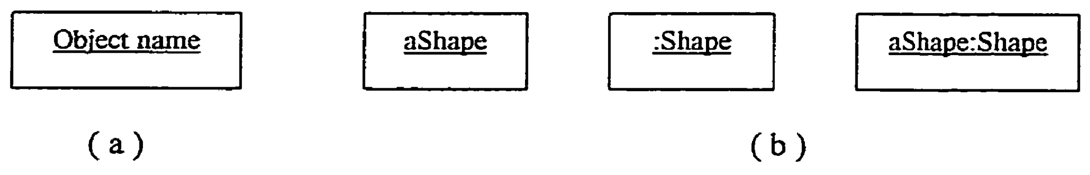


Figure 2-3: Graphical representation of an object in UML – definition (a) and an example (b)

An **interface** is a collection of operations that specify a service provided by a class or a component. It defines a set of operation specifications described by their signatures only. It is usually attached to the class or component that realizes it via a solid line and is represented graphically as a circle with its name, as shown in Figure 2-4 (a). Figure 2-4(b) shows an example of the use of interface **Sortable** with the operations = and > by class **String**. Figure 2-4 (c) shows the interface **Sortable**; note that it is represented by the same symbol as a class but marked with the 'stereotype' <<interface>>. (Stereotypes are covered in Section 2.3.1)

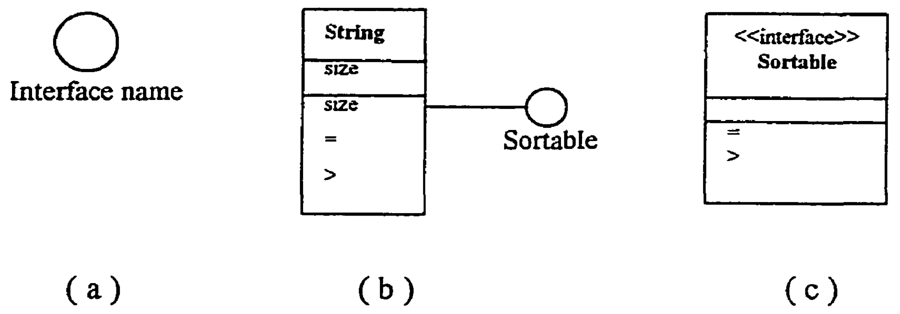


Figure 2-4: Graphical representation of interfaces in UML – definition (a) and examples (b) and (c)

A **collaboration** is a combination of roles and elements that work together to provide some cooperative behavior. Collaborations represent the implementation of patterns that make up the system. Collaborations have a structural part that specifies the classes, interfaces and other elements that work together, and a behavioral part that specifies the dynamics of the interaction of these elements. The structural part is rendered using a class diagram (Section 2.1.3.1) and the behavioral part is rendered using an interaction diagram (sections 2.1.3.4 and 2.1.3.5). A collaboration can be represented graphically by a dotted ellipse that includes the name of the collaboration, as shown in Figure 2-5 (a). Figure 2-5 (b) shows an example of a collaboration called **Internode messaging** which represents secure messaging among nodes in a Web-based retail system.

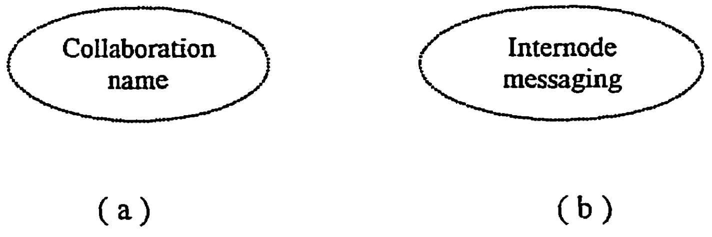


Figure 2-5: Graphical representation of collaborations in UML - definition (a) and an example (b)

A **use case** describes a sequence of actions that a system performs from the perspective of system's users. It describes the system activities from the point view of its actors. A use case is always initiated by an actor - a human user, a physical sensor or a class located outside the system that is involved in the interaction with the system described in a use case. A use case is represented graphically by a solid ellipse, as shown in Figure 2-6 (a). Figure 2-6 (b) shows an example of a use case called **Money withdrawal** in an ATM system. This use case might represent the dialog between a user and an ATM resulting in withdrawing money from an ATM.

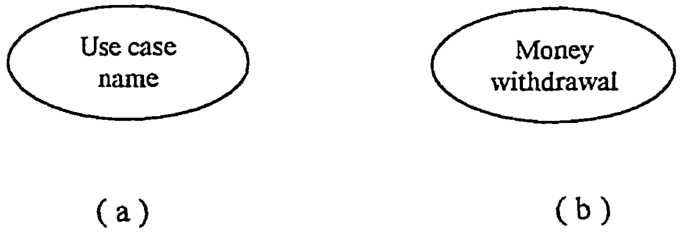


Figure 2-6: Graphical representation of use cases in UML – definition (a) and an example (b)

An **active class** is a class whose objects own one or more processes or threads. Objects from an active class represent elements whose behavior is concurrent with other elements. Graphically, it is represented as a class with heavy lines, as shown in Figure 2-7.

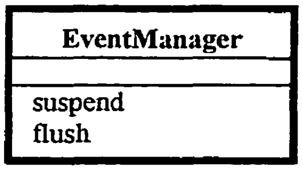


Figure 2-7: Graphical representation of active class in UML

A **component** is a physical part of a system that provides the realization of a set of interfaces. It represents the physical packaging of classes, interfaces and collaborations such as a file containing the source code of some part of the system, libraries, tables or documents. It is represented graphically by a rectangle with tabs with the inclusion of its name, as shown in Figure 2-8 (a). Figure 2-8 (b) shows an example of a component called **uml.st**, a Smalltalk source code file.

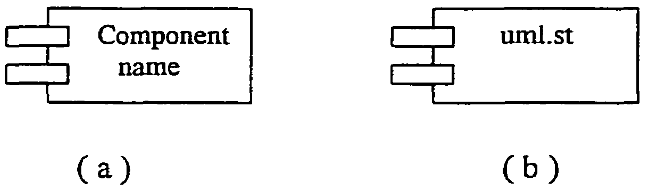


Figure 2-8: Graphical representation of a component in UML - definition (a) and example (b)

A **node** is a physical element representing a computational resource that exists at runtime. A node may contain a set of components. An example of a node is the implementation of a client in a client-server system. It is represented graphically by a cube that includes its name, as shown in Figure 2-9.

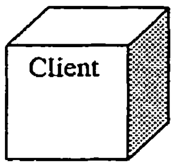


Figure 2-9: Graphical representation of nodes in UML

2.2.1.2 Behavioral Things

Behavioral things are the dynamic parts of UML models. They represent behavior over time and space. They are usually connected to structural things like classes, objects and collaborations. There are two kinds of behavioral things in UML: interaction and state machine.

An **interaction** is a sequence of messages exchanged among a set of objects. An interaction involves messages, action sequences and links. Messages are the stimuli exchanged between objects in the system. Action sequences define the order in which messages are sent. Links capture the relationship between a message sender and a receiver. The graphical representation of a message is a directed line labeled with the name of its operation, as shown in figure 2-10.

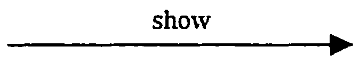


Figure 2-10: Graphical representation of a message in UML

A **state machine** is a sequence of states that an object goes through in response to a sequence of events. A state machine involves states, state transitions and events. The state of an object represents the cumulative history of the object's behavior. State encompasses all of the object's static properties and their current values. A state transition is a change of state caused by an event. State transitions connect two states in a state diagram or show state transitions from a state to itself. An Event is an occurrence that causes the state of a system to change. It can convey data values or information from one object to another. The graphical representation of a state is a rounded rectangle with its name and its substates, as shown in Figure 2-11. Section 2.1.3.6 includes an example of a statechart diagram that includes states, transitions and events in a state machine.

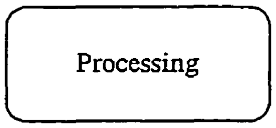


Figure 2-11: Graphical representation of states in UML

2.2.1.3 Grouping Things

Grouping things are the organizational parts of UML models. There is only one grouping thing in UML and is called a package.

A **package** is a construct for organizing structural, behavioral and grouping things into groups. It is represented graphically by a tabbed folder with a name and optional contents, as in Figure 2-12.

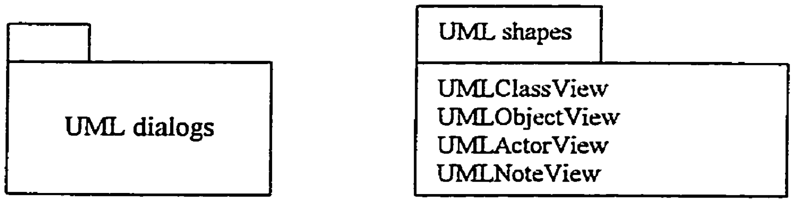


Figure 2-12: Graphical representation of packages in UML

2.2.1.4 Annotational Things

Annotational things are the explanatory parts of UML models. There is only one annotational thing in UML and it is called a note.

A note is a graphical symbol containing textual information, such as constraints, comments, method code bodies, and tagged values, about an element or a collection of elements. It is represented graphically by a rectangle with a dog-eared corner with a textual or graphical comment, as shown in Figure 2-13 (a). Figure 2-13 (b) shows how a note is attached to an element; in this case a class called **Shape**.

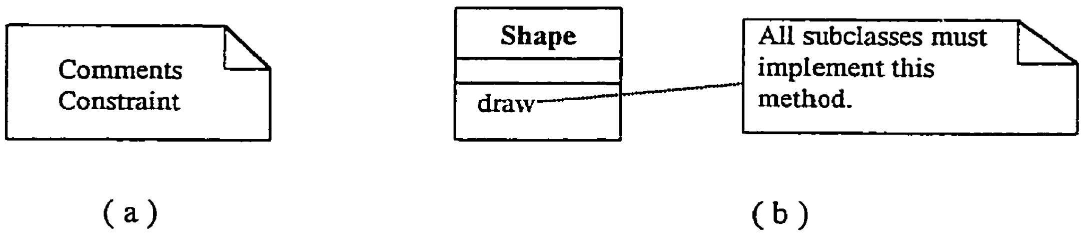


Figure 2-13: Graphical representation of a note in UML – definition (a) and example (b)

2.2.2 Relationships

Relationships tie things together. There are four relationships in UML: dependency, association, aggregation and generalization relationships.

A **dependency** is a relationship between two things in which one thing is dependent on the other. Any change to one thing may affect the state of the other thing. Dependency is a visibility relation where one thing is visible to the other. It is most often between a class that uses another class as a parameter to an operation. An example of dependency relationship is the relation between a **Client** and a **Supplier** where the client depends on the supplier to provide it with certain services (Figure 2-14 (b)). Dependency relationship is represented graphically with a directed dotted line with an optional name, as shown in Figure 2-14 (a).

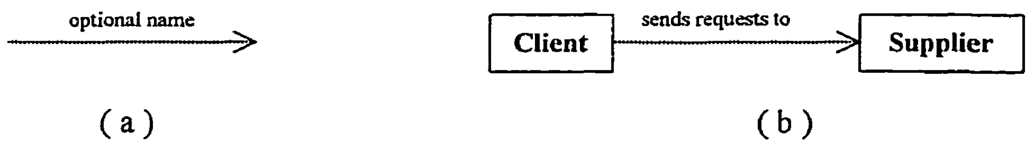


Figure 2-14: Graphical representation of a dependency in UML - definition (a) and an example (b)

A **generalization** is a relationship that represents inheritance between two things. It shows a relationship between a general thing (a superclass) and a specific kind of that thing (a subclass). A generalization relationship means that the subclass shares and extends the structure or behavior defined in one or more superclasses. An example of a generalization is the relation between the **Account** class (the superclass) and the **SavingAccount** class (the subclass) as shown in Figure 2-15 (b). It is represented graphically by a solid line with a hollow arrowhead including an optional name to identify the type or purpose of the relationship, as shown in Figure 2-16 (a). The hollow arrowhead always points to the superclass.

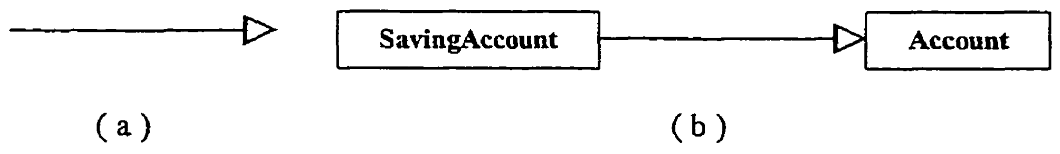


Figure 2-15: Graphical representation of generalizations in UML – definition (a) and an example (b)

An **association** is a relationship that describes a set of links between objects. It shows that objects of one kind of thing collaborate with objects of another kind of thing, or that one object uses the services of other objects. An example of an association relationship is the relation between a **Person** and the **Company** he or she works for, as shown in Figure 2-16 (b). It is represented graphically by a solid line that may have a direction, a name, a cardinality and a role name on either side, as shown in Figure 2-16 (a). The notations (1..*) means one or more and (*) means zero or more. In Figure 2-16, it means that the company can have one or more persons and a person can work for zero or more companies.

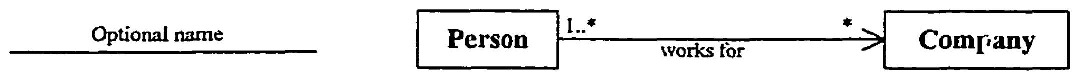


Figure 2-16: Graphical representation of associations in UML – definition (a) and an example (b)

An **aggregation** is a special kind of association that represents a relation between a whole and its parts. It shows that one thing represents a larger thing, the whole part - a container, consists of smaller things, the parts. It relates an assembly class to its components classes. For example, the relation between a **Document** and a **Paragraph** is an aggregation relationship, where the whole is the document and the parts are the paragraphs (Figure 2-17 (b)). An aggregation is represented graphically by a solid line with a diamond at the whole side, as shown in Figure 2-17 (a).

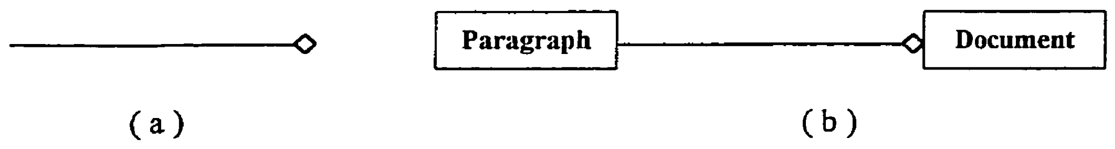


Figure 2-17: Graphical representation of an aggregation relationship in UML – definition (a) and an example (b)

2.2.3 Diagrams

A diagram is a connected graph with things representing the nodes and relations representing the edges. A diagram shows the elements that make up the system. There are nine diagrams in UML and they are explained in the following sections.

2.2.3.1 Class diagram

The class diagram is the most common diagram in object-oriented models. It shows the relationships between classes, interfaces and collaborations. The nodes in a class diagram are classes, interfaces and collaborations and the edges are dependency, association, generalization and aggregation relationships. Class diagrams may contain packages or subsystems, notes, and constraints. Class diagrams are used for modeling the static design view of the system that supports the function requirements of the system. An example of a simple class diagram is shown in Figure 2-18.

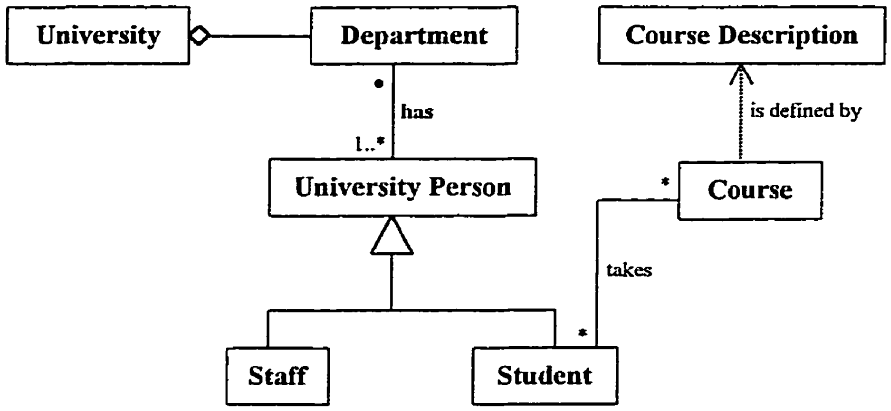


Figure 2-18: A Class diagram in UML

2.2.3.2 Object diagram

An object diagram shows the relationship between objects in a system. An object is an instance of a class. It is present at execution time and allocates memory for its instance variables. An object diagram shows a snapshot of instances found in the class diagram. The nodes of the object diagram are objects and the edges are links. Object diagrams are instances of class diagrams or the static part of interaction diagrams. They are used for modeling the static design view of a system from the perspective of real cases. An example of an object diagram is shown in Figure 2-19.

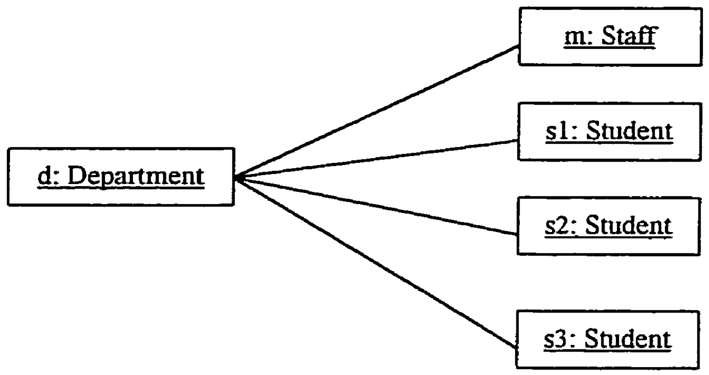


Figure 2-19: An Object diagram in UML

2.2.3.3 Use case diagram

A use case diagram shows the relationship between use cases and actors. The nodes of use case diagrams are use cases and actors and the edges are dependencies, generalizations and associations relationships. Use case diagrams organize and model the behaviors of a system. They describe what the system is supposed to do from the point view of its actors. They may contain notes and constraints and are used for modeling the static use case view of a system. Figure 2-20 shows an example of a use case diagram for a catalog order system [Richter 1997].

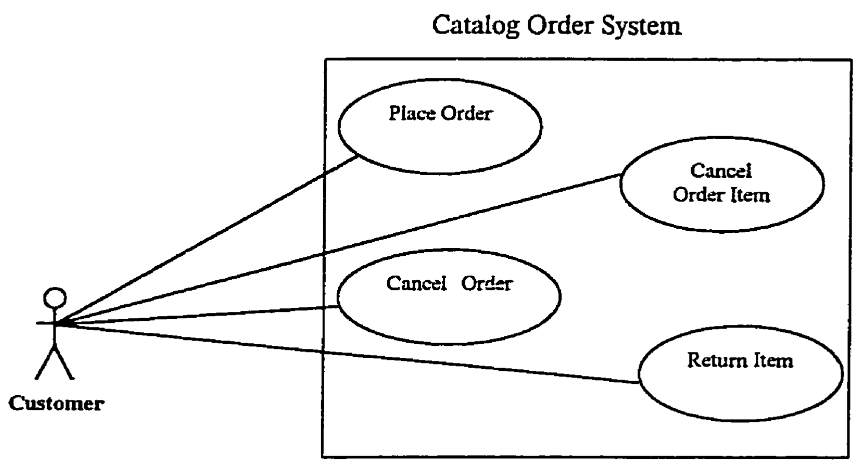


Figure 2-20: A Use case diagram in UML

2.2.3.4 Sequence diagram

A sequence diagram is an interaction diagram that shows ordering of messages in time. An interaction diagram shows the messages dispatched between objects and it is used for modeling the dynamic view of a system. The contents of sequence diagrams are objects, links and messages. Sequence diagrams give detailed description of use cases in a system. That means each sequence diagram is an instance of a use case. Sequence diagrams may contain notes and constraints. A vertical dashed line in a sequence diagram represents the existence of an object over a period of time and a thin rectangle shows the time taken for an object to perform an action. Figure 2-21 shows an example of a sequence diagram for a simple **Automated Teller Machine withdrawal**. The directed line with the filled arrow head represents sending a message and the directed line with an arrow only represents returning a value.

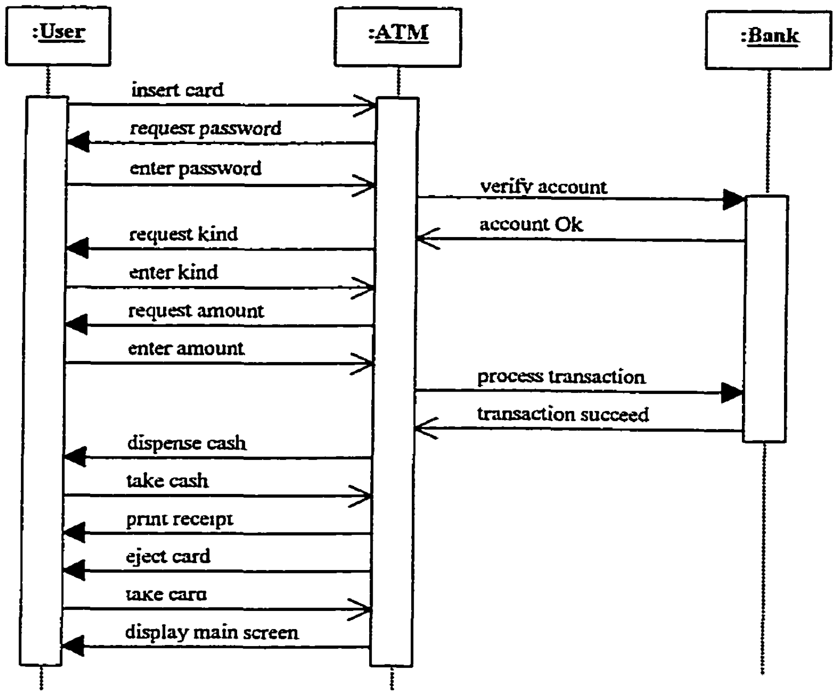


Figure 2-21: A Sequence diagram in UML

2.2.3.5 Collaboration diagram

A collaboration diagram is an interaction diagram that shows sending or receiving messages in the context of the structural organization of objects. It can always be transformed into a sequence diagram and vice versa. The contents of collaboration diagrams are objects, links and messages. Collaboration diagrams, like any other UML diagram, may contain notes and constraints. Sequence numbers are used to indicate the order of messages in time. Figure 2-22 shows a simple example of a collaboration diagram, a part of the sequence diagram in Figure 2-21. Note how the messages are numbered. The directed line in Figure 2-22 that has a circle at one end to represents a return value, the line without a circle indicates a message.

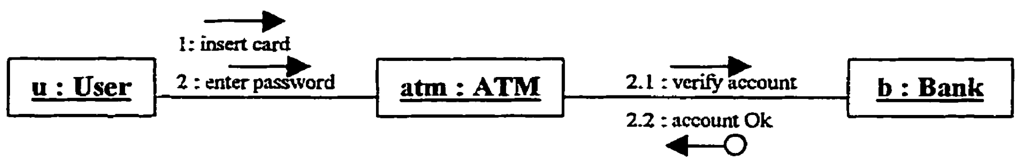


Figure 2-22: A Collaboration diagram in UML

2.2.3.6 Statechart diagram

A statechart diagram shows the definition of a state machine. It consists of states, events and activities and may contain notes and constraints. It shows the sequences of states of an object in response to outside stimuli, together with its responses and actions. Statecharts show the behavior of an interface, class or collaboration and the event-ordered behavior of an object. They show the flow of control from state to state and model the dynamic view of a system. Figure 2-23 shows an example of a statechart diagram using state machine. The example shows the states associated with operation of a telephone. The initial state is `Idle`, the final state is `Active`. The `Active` state is a state machine with an initial state `Dial Tone` and final states `Time out`, `Ringing`, `Invalid` and `Busy`.

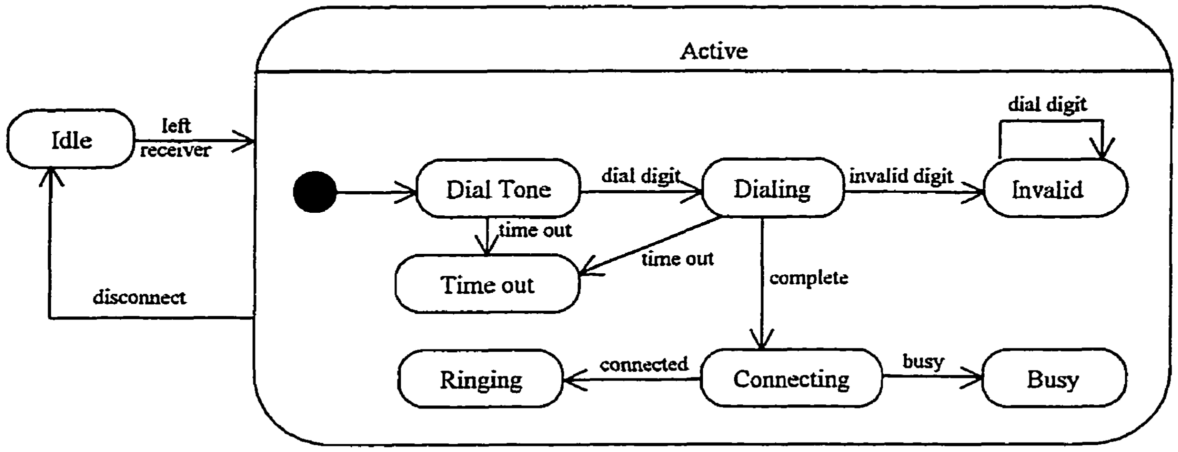


Figure 2-23: A Statechart diagram in UML

2.2.3.7 Activity diagram

An activity diagram is a special kind of statechart diagram. It shows the flow of activities in a system where an activity is an ongoing execution within a state machine. It shows the sequences of states for an object in response to completion of internal state operations. Activity diagrams show the flow of control among objects in a system.

They emphasize the flow of control from activity to activity. They contain activity states, action states, transitions and objects and may contain notes and constraints. They are used for modeling the dynamic view of a system by modeling the sequential steps in a computational process and the flow of an object as it moves from state to state. Figure 2-24 shows an example of an activity diagram for password validation. The hollowed diamond is a decision symbol.

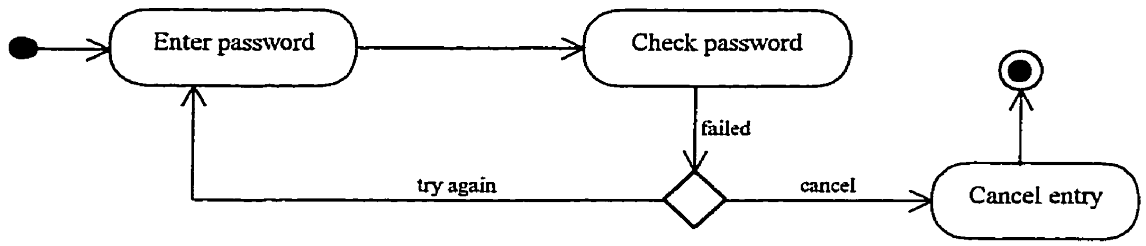


Figure 2-24: An Activity diagram in UML

2.2.3.8 Component diagram

A component diagram is a language/system-dependent implementation diagram that shows dependencies among components. Any component in a component diagram maps into one or more classes, interfaces or collaborations. Component diagrams are used for modeling the static implementation view of a system by modeling physical things such as libraries, files, tables and documents. The nodes of component diagrams are components and interfaces and the edges are dependencies, generalizations, associations and realizations. Component diagrams may contain packages, subsystems, notes and constraints. Figure 2-25 shows an example of a component diagram for a Smalltalk source file using two data files. It shows that the component test.st is dependent on two components, namely file1.bos and file2.bos.

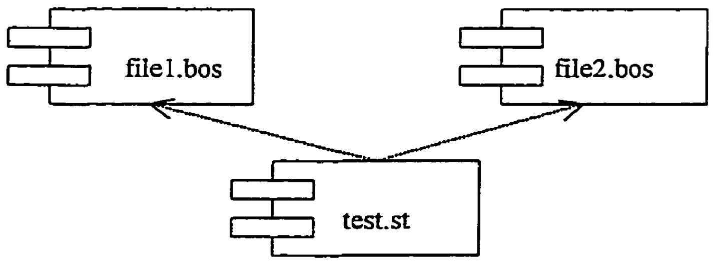


Figure 2-25: An Component diagram in UML

2.2.3.9 Deployment diagram

A deployment diagram shows the system in terms of its hardware nodes. A hardware node may be a physical processor, such as a CPU, or a device such as a printer. A node in the deployment diagram contains one or more components. The nodes of the deployment diagrams are nodes and the edges are dependencies and associations relationships. Deployment diagrams may contain packages, subsystems, notes and constraints. They are used for modeling the static deployment view of a system by describing the topology of the hardware on which a system executes. Figure 2-26 shows an example of a deployment diagram.

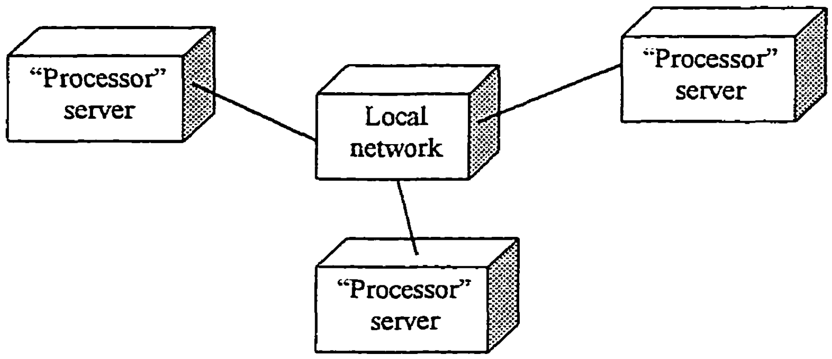


Figure 2-26: A Deployment diagram in UML

2.3 Views

A view in UML is a set of UML diagrams. Views link the modeling language to the method chosen for developing a system. Views are very important for capturing the complete picture of the system to be constructed. Each view shows a particular aspect in describing the system. The IEEE Draft Standard [IEEE 1998] refers to a view as something which "address one or more concerns of the system stakeholder, an individual or a group that shares concerns or interests in the system such as developers, users, customers, etc." A view is a piece of the model that is still small enough for us to comprehend and that also contains all relevant information about a particular concern. Views do not have a graphical representation; they are only conceptual or physical groupings of UML diagrams. There are five views in UML [Eriksson 1998]: logical view, use case view, component view, concurrency view and deployment view. What type of view should be used and when it should be used is strongly dependent on the person who is using it and the tasks that are needed to be accomplished. UML views capture both structural and behavioral aspect of software development. Structural views make use of classes, packages, use cases and so forth. Behavioral views are represented through scenarios, states and activities. The five UML views are explained in the following sections.

2.3.1 Logical View

This view captures the system's static structure and dynamic behavior. The static structure is described in class and object diagrams. The dynamic behavior is described in state, sequence, collaboration and activity diagrams. It is used by designers and developers. It defines interfaces and the internal structure of classes.

2.3.2 Use case View

This view captures the functionality of the system as seen by external actors. It is used by customers, designers and developers. It is also used by testers to validate the system by testing the use case view against the finished system. It contains use case diagrams and activity diagrams.

2.3.3 Component View

This view shows the organization of code components. It describes the implementation modules and their dependencies. It is used by developers and contains component diagrams.

2.3.4 Concurrency View

This view shows concurrency in the system and addresses communication and synchronization problems. It divides the system into processes and processors, describing parallel execution and handling of asynchronous events. The concurrency view is for the developers and integrators of the system. It consists of state, sequence, collaboration, activity, component and deployment diagrams.

2.3.5 Deployment View

This view shows the deployment of the system into the physical architecture. It encompasses the nodes that form the system's hardware topology on which the system executes. It is for developers, integrators and testers of the system. It consists of the deployment diagram. It also shows which programs are executed on which computer.

2.4 Extending UML

All types of analysis and design documents that may ever be needed in the future can not be predicted and this is why UML is an open-ended language. This characteristic allows for extending the language to cover possible new types of models. It allows UML to grow to meet any project's needs and adapt to any software technology. UML can thus be adapted to a specific application domain methodology, user or organization. UML defines three extension mechanisms: stereotypes, tagged values and constraints. Figure 2-27 shows examples of each of them.

2.4.1 Stereotypes

This extensibility mechanism deals with extending the `vocabulary` of UML. It allows for the creation of new building blocks from the basic ones. The graphical representation of a stereotype in UML is a name enclosed in guillemets as in <<abstract>>, and placed above the name of another element, as shown in Figure 2-27. In some languages like C++ or Java, the designer might want to model abstractions where abstractions are just classes that can not be instantiated. In Figure 2-27, **LinkedList** is a class that is marked with an appropriate stereotype <<abstract>> to identify it as an abstract class.

2.4.2 Tagged values

This extensibility mechanism deals with extending the `properties` of UML building blocks. It allows for the inclusion of new types of information in the specification of certain elements. Tagged value is represented graphically by a string enclosed by braces written under the name of another element such as {version 1.1}, as shown in Figure 2-27. If you are working on a product that undergoes many releases over time, you often want to track its version. Since version is not a UML concept, it can be added to a class like **LinkedList** by introducing new tagged value, {version 1.1}, to the class.

2.4.3 Constraints

This extensibility mechanism deals with extending the semantics of a UML building block. It allows for adding or modifying UML rules. A constraint is represented graphically by a string between braces and placed near the element or connected to the element by dependency relationship such as {ordered}, as shown in Figure 2-27. Figure 2-27 shows that the **LinkedList** class is constrained so that all additions are done in order.

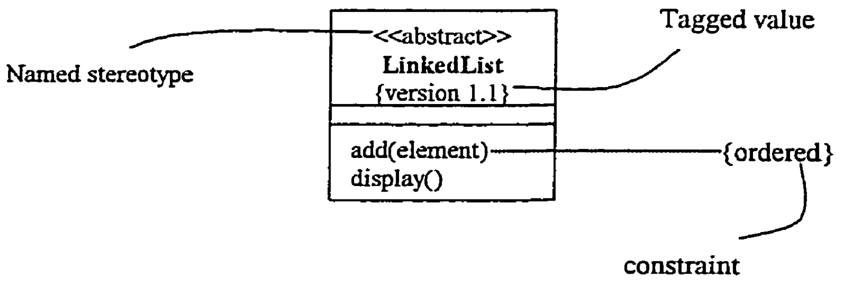


Figure 2-27: Extensibility in UML

2.5 OPEN Modeling Language (OML)

OPEN (Object-Oriented Process, Environment and Notation) is an alternative object-oriented development method developed and maintained by the OPEN Consortium (consists of 26 internationally recognized OO methodologists, researchers, who endorse, develop and teach the OPEN approach to OO development) [Firesmith 1998]. It consists of a modeling language (OML) and a process. It is based on the unification of the methods of Henderson-Sellers, Graham and Firesmith [Firesmith 1998].

OML (OPEN Modeling Language) is a notation for depicting object-oriented systems. It was developed by Don Firesmith, Brian Henderson-Sellers and Ian Graham, with considerable input from other members of the OPEN consortium. It consists of a metamodel for specifying the syntax and semantics of underlying concepts of object-oriented modeling and a notation, COMN (Common Object Modeling Notation), for documenting the models produced by using OPEN. Henderson-Sellers describes it as being designed from the bottom up by a small team of methodologists who were able to propose a notation free from the hereditary biases of earlier data-modeling methods. The intention of the designers of OML is that it should concentrate on describing the commonly understood elements of object technology such as encapsulation, interfaces, inheritance and a discrimination between objects, classes and types.

Despite the wider acceptance of the UML, OML continues to be enhanced and may be used as a superset to the UML. OPEN, however, has not yet been backed by the commercial forces that backed the UML. Thus OPEN has not yet gained much commercial attention. However, Brian Henderson-Sellers, one of OPEN's authors, believes "we are now entering into the realization by industry developers that they DO need a methodology and when they do, OPEN's mindshare is set to grow by leaps and bounds" [Gottesdiener 1998].

In general, OML and UML are quite similar. Their notations differ but they can be used to show the same concepts, Figure 2-28 shows the OML equivalent of the UML class diagram in Figure 2-18.

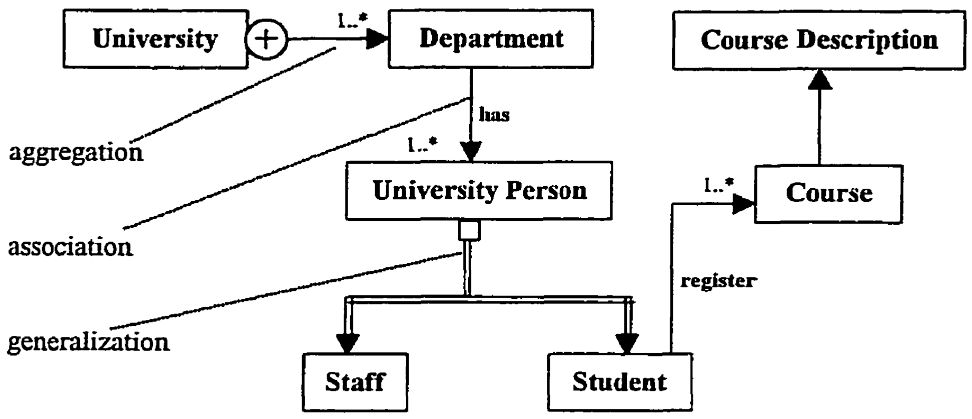


Figure 2-28: OML representation of the UML Class Diagram in Figure 2-18

One of the main differences between the UML and OML is the UMLs lack of a process. The reason that UML is not bound to any process will be explained briefly in the next section. OML on the other hand has a full lifecycle process, which according to Henderson-Sellers can be used together with UML. UML has the advantage that it is already familiar to the vast majority of the software community and dominates the industry.

2.6 Conclusion

UML is a modeling language, a collection of definitions and universal symbols for use in the description of OO systems. It is not a methodology and its use is method-independent. It is basically a modeling language. The Unified Process [Jacobson 2000] is an attempt to provide such a process for the Unified Method [Booch 1995]. Since the aim was to make UML standard and acceptable, it was unnecessary to bind UML to a certain process because it is almost impossible to define a process that would be equally suitable for the different factors involved in the development of varying types of software. UML was designed to be applicable to any process.

UML provides support for modeling classes, objects, the many kinds of relationships among them, dynamic system behavior and features of physical implementation. UML is extensible through the definition of additional stereotypes, tagged values and constraints. It defines a number of diagrams, their structure and their use. Larger diagrams are shown in the description and design chapters. UML is becoming the standard notation for object-oriented system development because of the support it already has from many methodologists, software developers and programmers, management consulting firms, system analysts and CASE tool vendors. As an example, the following companies support UML: Digital Equipment, IBM, Hewlett-Packard, ICON Computing, I-Logix, Microsoft, MCI Systemhouse, IntelliCorp, ObjecTime, Sterling Software, Texas Instruments, James Martin & Co, and Unisys. The recognition of UML is apparent from the huge number of books that have been written about UML and by the thousands of people that have taken training courses in the language. All this progress and UML's use is still expected to grow substantially in the years to come.

The best source for up to date information on UML is Rational's site where Grady Booch, Ivar Jacobson, and Jim Rumbaugh continually develop and extend the UML notation. The most recent updates on the Unified Modeling Language are available via the worldwide web at http://www.rational.com/uml.

Chapter 3

The Binder

3.1 Introduction

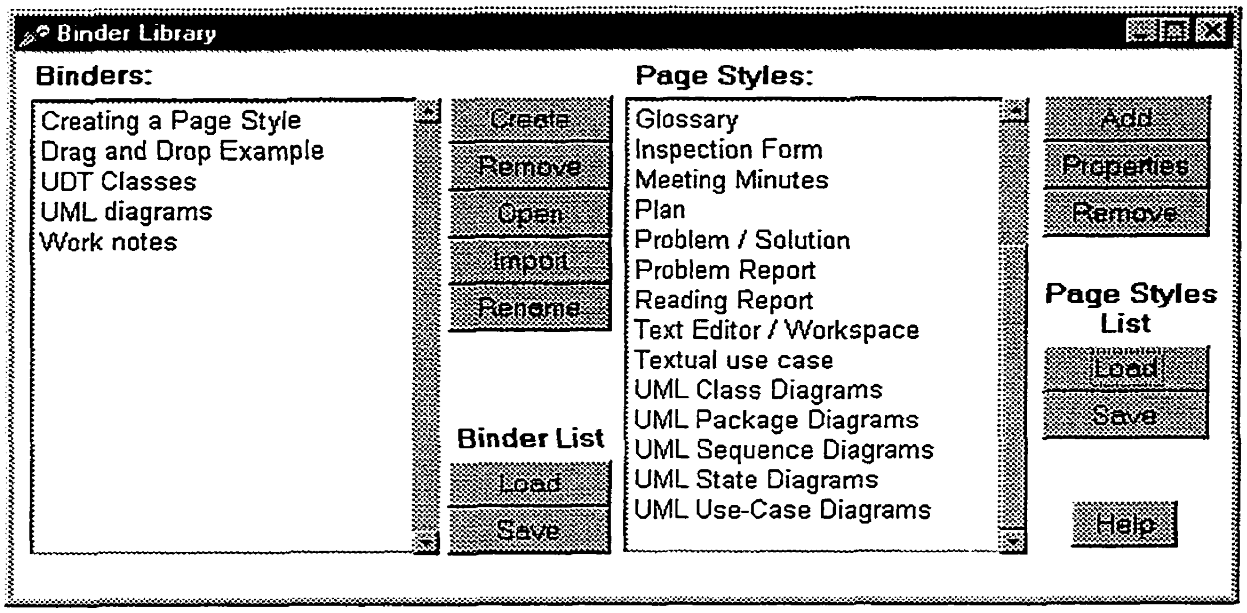
After the overview of methodologies presented in the first two chapters, we will now begin a detailed presentation of the main subject of this thesis – the design and implementation of UML tools in a larger framework.

Documentation is a very important part in any project development and keeping all project documentation in an integrated whole for easy access is equally as important. Project documentation is usually kept in a combination of electronic and paper. Even in an electronic form, the variety of formats results in a set of incompatible files that are hard to integrate and process in a uniform way. Hence there is a need for a tool that can keep everything organized and bound together. An attempt to address this problem resulted in a project called an Electronic Binder [Tomek 2000], a tool that was developed by Acadia students and faculty to provide various kinds of page-styles and allow a developer to organize and keep all project documentation together for easy access and printing. The Binder also provides search mechanisms for searching the whole content, a section, or a specific page-style. The Binder program is extendable and allows for the adding of new page-styles.

The following sections briefly describe the Binder to provide a context for the UDT tool, which has been implemented as a collection of Binder page styles to support the drawing of selected UML diagrams. The details of UDT are explained in the next three chapters.

3.2 The Binder – a Description

A Binder is a collection of pages and/or sections and each section can have any number of sections and/or pages. A Binder is a part of a binder Library. We will now explain the operation and user interfaces of a binder to provide a clear context for the rest of the thesis.



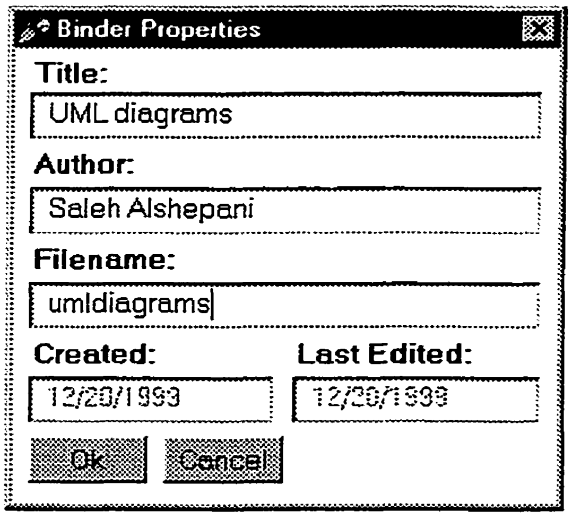
Creating a Page Style
Drag and Drop Example
UDT Classes
UML diagrams
Work notes

Glossary
Inspection Form
Meeting Minutes
Plan
Problem / Solution
Problem Report
Reading Report
Text Editor / Workspace
Textual use case
UML Class Diagrams
UML Package Diagrams
UML Sequence Diagrams
UML State Diagrams
UML Use-Case Diagrams

Figure 3-1: Binder Library Window

Figure 3-1 shows the first window that will be opened when the Binder program is run. It displays two lists with buttons on the right side of each list. The list on the left-hand side shows all binders stored in a library catalog. The buttons on the right side of this list have the following functions: The *Create* button is for creating a new binder. It opens a special window (Figure 3-2). The *Remove* and *Rename* buttons operate on the binder selected in the binder's list. The *Import* button imports a new binder from a file into the library catalog, allowing users to exchange binders. The *Load* and *Save* buttons are for loading and saving entire library catalogs. The *Open* button is used for opening the selected binder and clicking it opens the window in Figure 3-4.

The list on the right-hand-side of Figure 3-1 lists all available page styles. The buttons on the right side of this list have the following functions: the *Add* button is for adding a new page-style (Figure 3-3). The *Remove* button removes the selected page-style. The *Properties* button shows the properties of the selected page-style (Figure 3-3). The *Save* and *Load* buttons allow users to save and load page-styles to or from files. The *Help* button opens a help window on the selected page-style.


Binder Properties

Title: UML diagrams

Author: Saleh Alshepani

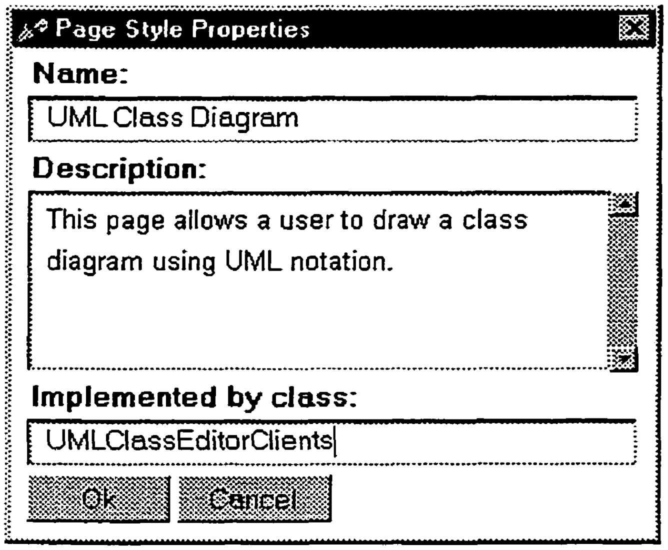
Filename: umldiagrams

Created: 12/20/1999

Last Edited: 12/20/1999

Figure 3-2: Binder Properties Dialog

The Binder Properties window in Figure 3-2 allows the user to create a new binder by filling in the required fields.


Page Style Properties

Name:

UML Class Diagram

Description:

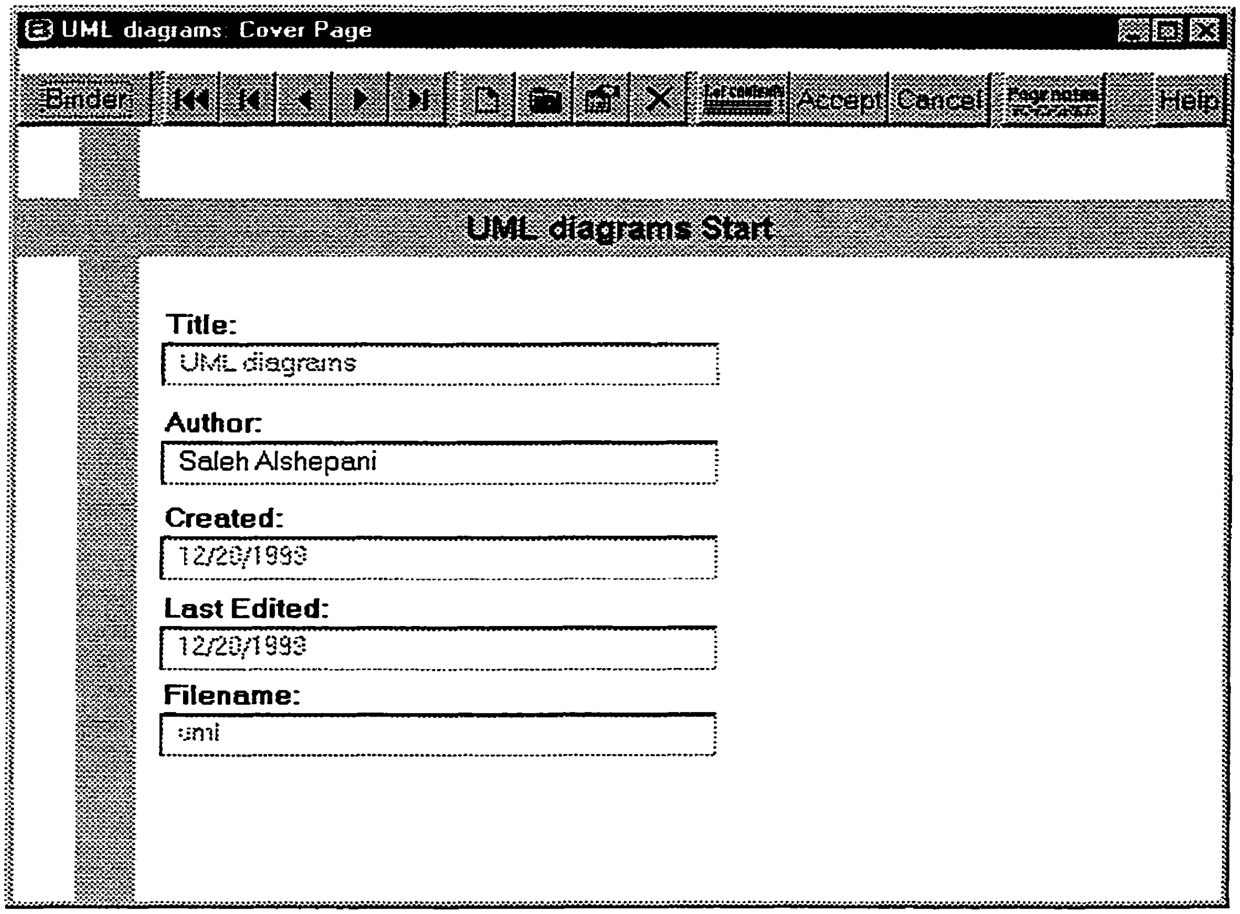
This page allows a user to draw a class diagram using UML notation.

Implemented by class:

UMLClassEditorClients

Figure 3-3: Page-Style Properties

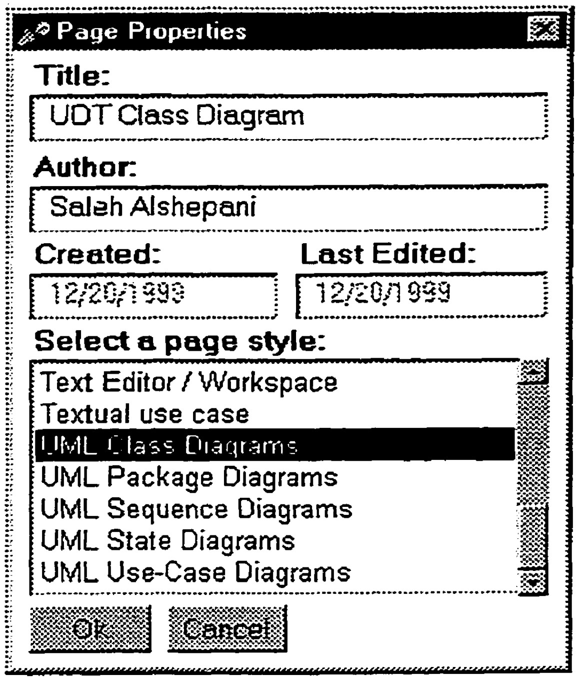
The Page Style Properties window in Figure 3-3 is displayed when the user clicks either the *Add* or the *Properties* button on the right side of the page's list (Figure 3-1). It allows the user to view, define and edit the selected page-style properties - its name, description and the Client class implementing the page. The Client class is used to provide the interface between the page-style application and the binder. Its function will be explained in more detail later. After clicking *Ok* on a new style, the new page-style is added to the list of page-styles.



Title:
UML diagrams
Author:
Saleh Alshepani
Created:
12/20/1999
Last Edited:
12/20/1999
Filename:
uml

Figure 3-4: A Binder's Cover Page

The window in Figure 3-4 is displayed when the user clicks on the *Open* button with a selected binder. It displays the title page of the binder. Using the buttons at the top of the window, the user can move among pages and sections and create, edit or delete them. The user can also view a collapsible table of contents, move pages and sections around using the table of contents, and print or search the whole binder or a selected portion of it.


Page Properties

Title:
UDT Class Diagram

Author:
Saleh Alshepani

Created: 12/20/1999

Last Edited: 12/20/1999

Select a page style:
Text Editor / Workspace
Textual use case
UML Class Diagrams
UML Package Diagrams
UML Sequence Diagrams
UML State Diagrams
UML Use-Case Diagrams

Figure 3-5: New Page Window

The window in Figure 3-5 is displayed when the user requests the creation of a new page. It allows the user to specify the name of the page and select its style from the list of available page-styles. Clicking *Ok* opens a new blank page with the selected style as in Figure 3-6. The basic user interface is the same for all page styles, but each style can add its special purpose buttons and provides its own layout.

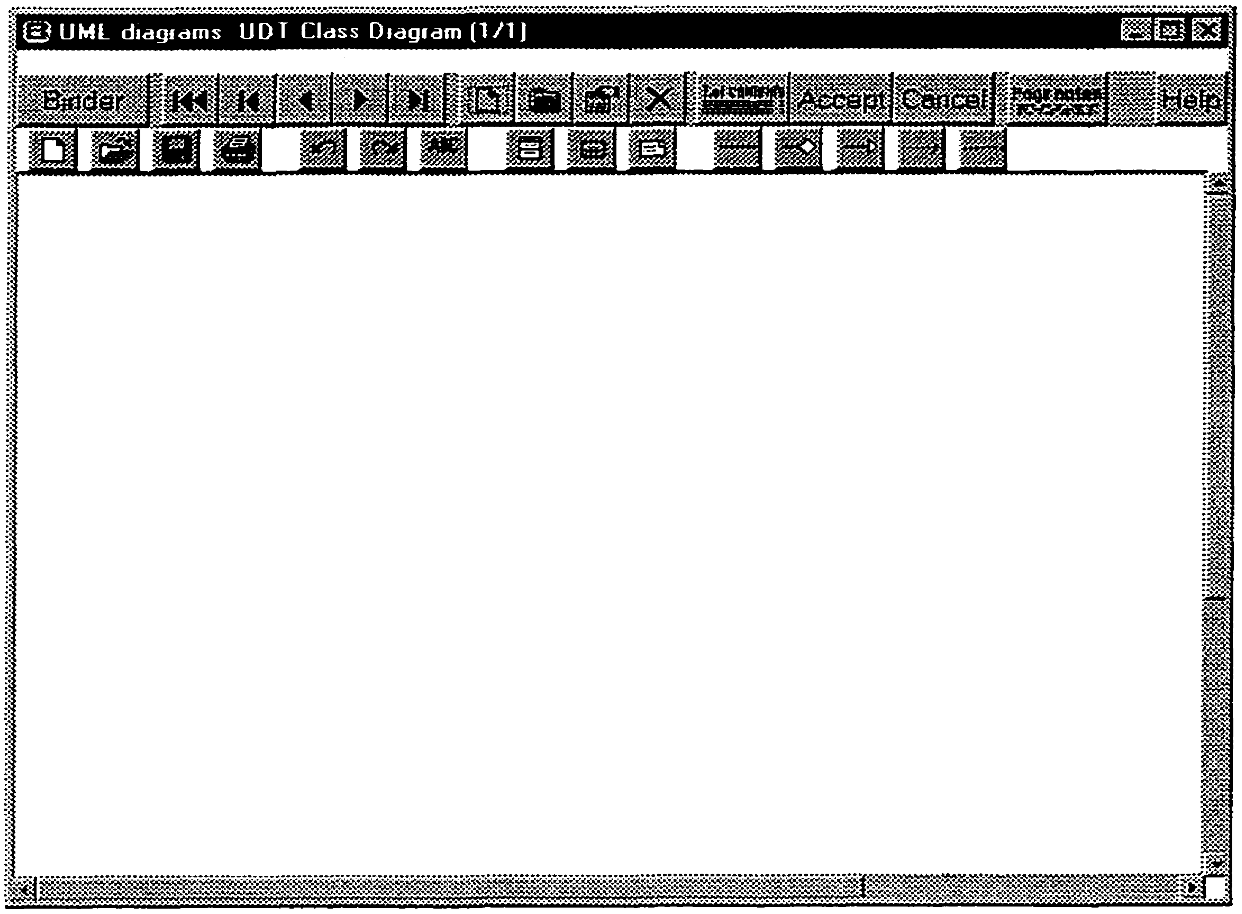


Figure 3-6: A blank Class Diagram Page-Style with style-specific buttons

3.3 Creating new Page-Styles

Creating a new page–style requires the implementation of two or more classes. The first required class is an application class, which defines the user interface for that page. Its instance is plugged into the universal binder page. This class must be a subclass of the **ApplicationModel** class. The second required class is a client class, which provides the interface between the application class and the binder. This class must be a subclass of the **ClientApplication** class. The **ClientApplication** class provides most of the functionality needed to accomplish the interface with the binder and defines the searching and printing operations for the binder. When creating a new page-style for the Binder, the developer must implement certain methods described in the following subsections.

3.3.1 Methods for the application class

The application class must implement method *`newWith`*: `anObject`. This is a class method and is sent to the application class from the client class to create a new instance of the application class on `anObject`.

3.3.2 Methods for the client class

The client class must implement methods *`newWithOutContents`*, *`newWithContents`*: `anObject`, `help`, and `isDirty`. Method *`newWithOutContents`* is a class method and is sent to the client class to return a new initialized instance of the application. Method *`newWithContents:`* `anObject` is a class method and is sent to the client class to return a new instance of the application class opened on `anObject`. Method `help` is a class method and is sent to the client class to return a help string for this page style. Method `isDirty` is an instance method that is sent to the client class instance to check whether the contents have changed. It is used to determine whether the user should be warned before proceeding to another page.

Methods `menu`, `accept`, `preClose`, `displayFromSearch`: `anObject`, and `searchFor`: `aString` `using`: `aStringSpec` should be overridden by the client class if some action other than the default is required. Instance method `menu` should be overridden if the application has a menu. Instance method `accept` should be overridden if a page needs to store anything between viewings. It returns whatever information the application wants to store between viewing of the page (e.g. Editor page should return its text). Instance method `preClose` should be overridden for clients that need to internally accept something before closing, or by any client, which needs something to happen before closing. It is called before the current page is closed or method `dirty` is called. It returns #continue if it is ok to continue closing or anything else otherwise. Instance method `displayFromSearch: anObject` should be overridden to do whatever is required to display this client when its being displayed as the result of a search. Instance method `searchFor: aString using: aStringSpec` should be overridden if the page wants to be searchable for the text `aString` and any additional arguments, such as wildcard options are supplied in `aStringSpec`.

3.4 Conclusion

Gathering all project-related information into a single document can easily be accomplished using the Binder tool described in this chapter. Besides application development, the Binder can be used for other applications as well. The author, Dr. Ivan Tomek, used it as an integrating medium for course materials. It has also been used to gather on-line help for Smalltalk.

Chapter 4

UML Diagrams Tool - Description

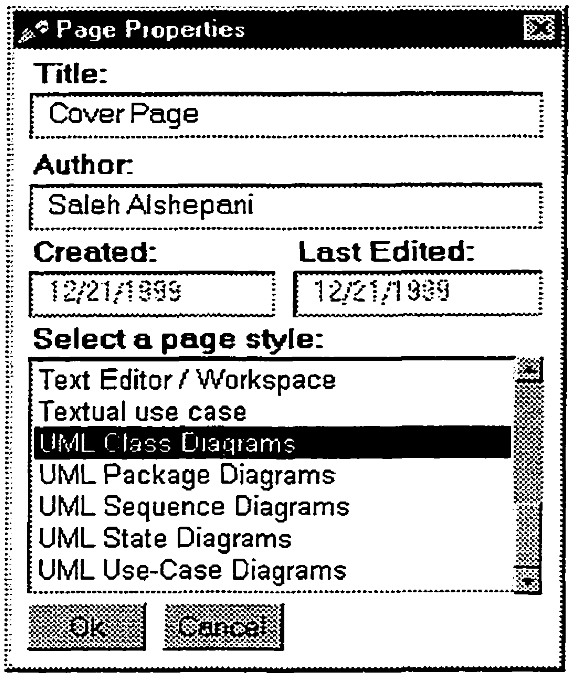
4.1 Introduction

Object-oriented design requires tools to assist developers. One area that needs support is the drawing of design diagrams, which can best be accomplished by a drawing tool. Tools are available that support the drawing of some UML diagrams. One such tool is Together Java [Together 1998] by Object International. This is a very elegant tool and is considered one of the best Java products available. The tool can be used to construct UML diagrams and generate Java code automatically. Another documentation tool is Rational Rose [Rose 1996] from Rational Software [Rational 2000]. This tool can also be used to construct UML diagrams and generate Java and other code automatically.

UML Diagrams Tool (UDT), implemented in this thesis, is a drawing tool that supports the drawing of several types of UML diagrams and their integration with the Binder (Chapter 3). It provides special *'page styles'* and an editor for such pages for the binder. The UML diagrams supported by UDT are:

- Class and Object Diagrams.
- Use Case Diagram.
- Sequence Diagram.
- State Transition Diagram.
- Package diagram.

UDT allows creation, editing, displaying, storing and deleting UML diagrams. It allows for converting the class diagrams into Smalltalk source code, which will help software developers in the implementation phase and their saving in a binder as part of complete project description. It also supports limited reverse engineering, allowing automatic creation of selected UML diagrams from Smalltalk code. Figure 4-1 shows how a UDT style is selected for a Binder page.


Page Properties

Title: Cover Page

Author: Saleh Alshepani

Created: 12/21/1999

Last Edited: 12/21/1999

Select a page style:

- Text Editor / Workspace
- Textual use case
- UML Class Diagrams
- UML Package Diagrams
- UML Sequence Diagrams
- UML State Diagrams
- UML Use-Case Diagrams

Figure 4-1: New Page window

As with every Binder style, UDT is a complete pluggable application and this means that it can also be used independently of the Binder program, performing the same tasks and providing the same functionality. Figures 4-3, 4-6, 4-7, 4-8 and 4-9 included later in this chapter show the editing windows of the supported UML diagrams.

4.2 UML Diagrams Tool (UDT)

The use of UDT in the Binder is as a drawing tool for selected UML diagrams. The users of this tool are expected to know UML notation but the tool provides on-line help.

UDT functional requirements can be divided into basic functional requirements and drawing functional requirements. These will be described next.

4.2.1 Basic functional requirements

UDT provides several features that are shared among all of its diagrams. These features are: saving diagrams in files, loading diagrams from files, opening new files, printing diagrams, scrolling windows to allow for larger diagrams, providing a tool bar for fast saving, loading and drawing of shapes, providing a pop-up menu for each shape in the diagram, allowing multiple undo and redo of previous actions and entering text anywhere in the diagram.

4.2.2 Drawing functional requirements

UDT allows the drawing of UML symbols and text. All symbols have a pop-up menu for performing symbol-specific actions. A shape can be moved anywhere inside the window and moving a shape drags it with all the lines (edges) that are connected to it. The size of a shape automatically adjusts to the size of the text inside it except for the **note** shape, which has a fixed size. Deleting a shape deletes all the lines that are connected to it. In accordance with UML rules, a shape can have a note attached to it using dashed line. The following shapes and lines can be drawn using UDT: A class, an object, a package, a use case, an actor, a state, a state machine, a start sate, an end state, a note, an association, an aggregation, a generalization and a dependency. Using a pop-up menu which becomes available when the user clicks the right mouse button inside a shape, any shape can be deleted or renamed. For the class shape, the pop-up menu can also be used to define a class with all or some of its methods as a Smalltalk class, toggle between class and instance sides and open a Smalltalk browser. For the note shape, the pop-up menu can also be used to change the text inside the note. Also all lines, edges, can be deleted or renamed using a pop-up menu. For the association and aggregation lines, the pop-up menu can also be used to change their properties (role name in either side, cardinality (none, zero or more, one or more, discrete numbers, range of numbers) in either side, direction (left, right, both, none)).

4.3 UML Diagrams Supported

UDT supports the following UML diagrams (page styles): Class and Object Diagram, Use case Diagram, Sequence Diagram, State Transition Diagram and Package Diagram. These diagrams (except the package diagram) were described in Chapter 2. A package diagram is part of a class diagram but is drawn separate in UDT. It shows the relationship between packages - collections of classes. It allows the following shapes and lines: A package shape, a note, and directed and undirected dashed lines.

4.4 UDT User Interface

Each kind of UDT diagram has an editor with a tool bar for easy access. When the user selects a style page when creating a new page for the binder, the appropriate UML diagram editor will be opened. All buttons on the tool bar are supported by pop-up bubble help. The first seven buttons of the tool bar (Figure 4-2) are common to all UDT diagrams. They are from left to right: `New`, `Load`, `Save`, `Print`, `Undo`, `Redo` and `Text`. The first six are self-explanatory and `Text` is used for entering a text anywhere in the diagram. The `Load` button, which is the second button from left, differs in the Class and Object Diagrams window by allowing the user to choose to load from a file or a Smalltalk code (for reverse engineering) – the other diagrams, only loading from a file is possible. We will now give examples of each of these diagrams and explain their operation.

4.4.1 Class & Object Diagrams window

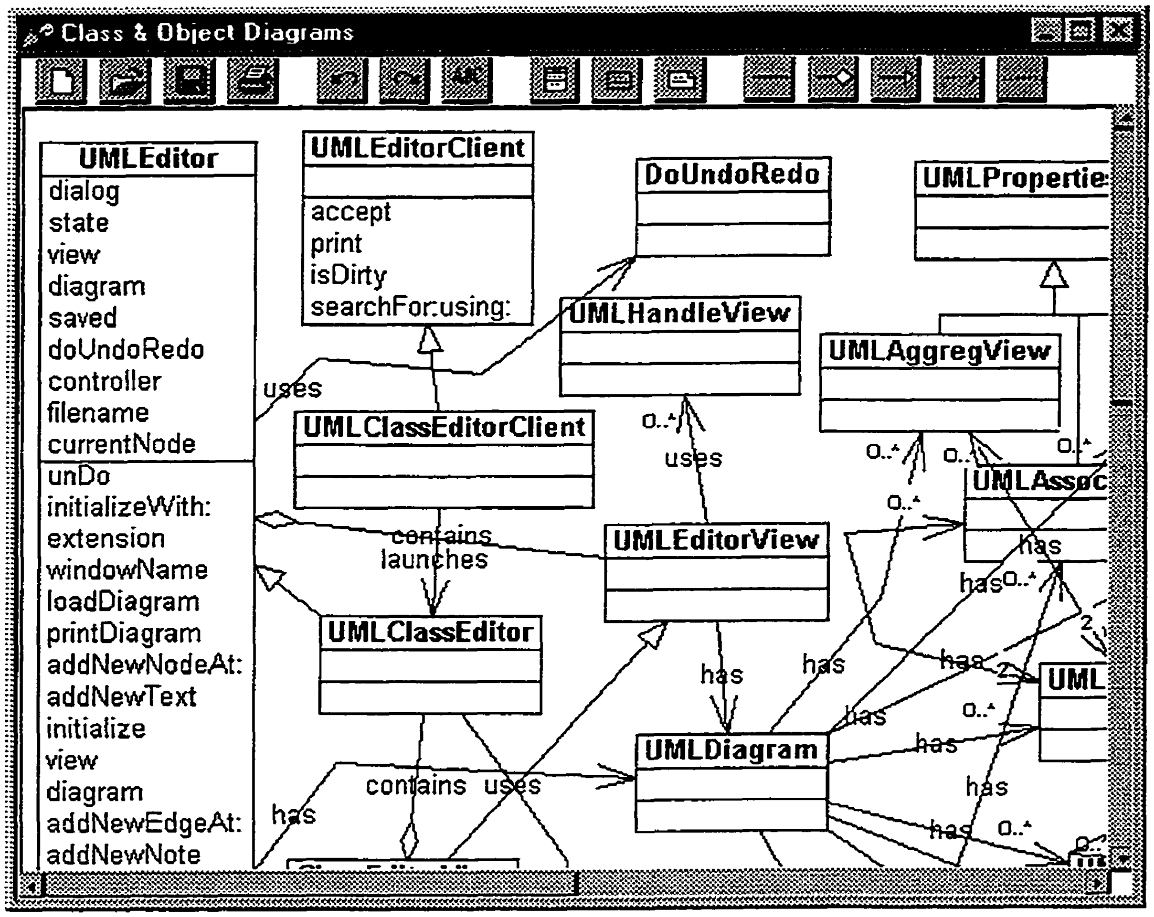


Figure 4-2: Class & Object Diagrams window

The Class & Object Diagram window, Figure 4-2, allows the user to draw UML Class and Object diagrams. The buttons from number eight to fifteen (left to right) are `Class`, `Object`, `Note`, `Association`, `Aggregation`, `Generalization`, `Dependency` and `Note Connection`. Clicking a button displays a dialog asking for a name, except the `Note Connection` which does not have a name. Clicking anywhere in the drawing area will then display that shape in the selected position. Clicking the

<operate> mouse button on any shape in the drawing area shows a pop-up menu specific to the selected shape.

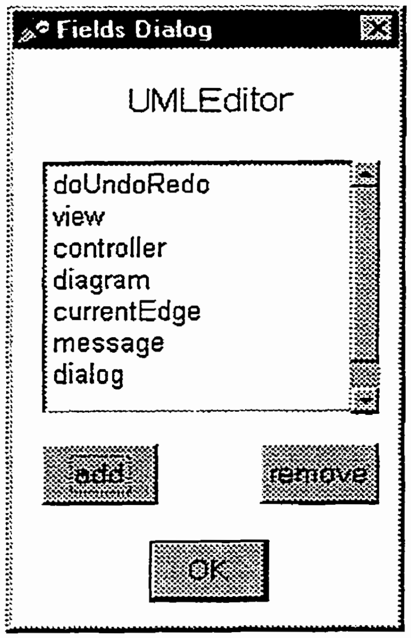
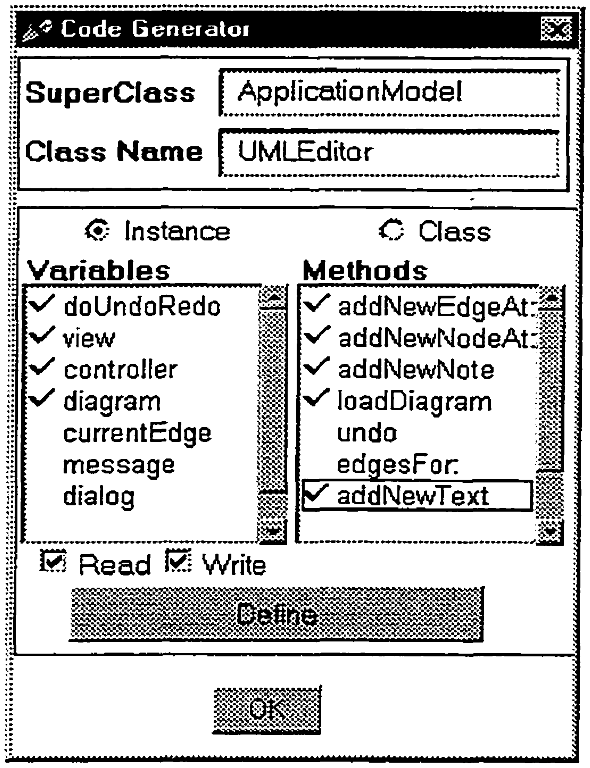


Figure 4-3: The Fields Dialog window

The Fields Dialog window, Figure 4-3, opens when the user selects 'change variables' or 'change methods' from a pop-up menu which becomes active when the user clicks the left mouse button inside a class shape. It allows adding or removing methods or variables to the class or instance side.



SuperClass: ApplicationModel

Class Name: UMLEditor

[✓ Instance] Class

Variables	Methods
✓ doUndoRedo	✓ addNewEdgeAt:
✓ view	✓ addNewNodeAt:
✓ controller	✓ addNewNote
✓ diagram	✓ loadDiagram
currentEdge	undo
message	edgesFor:
dialog	✓ addNewText

☑ Read ☑ Write

Figure 4-4: Code Generation window

To generate code from a diagram, the user can use the code generator window (Figure 4-4). This window (opened via selecting 'define' from the class shape pop-up menu) allows the user to generate Smalltalk code for the selected class in the Class and Object Diagram window. It defines the selected class, its instance and class variables and their accessor methods, as well as stumps of other methods listed in the class node.

4.4.2 Package Diagram window

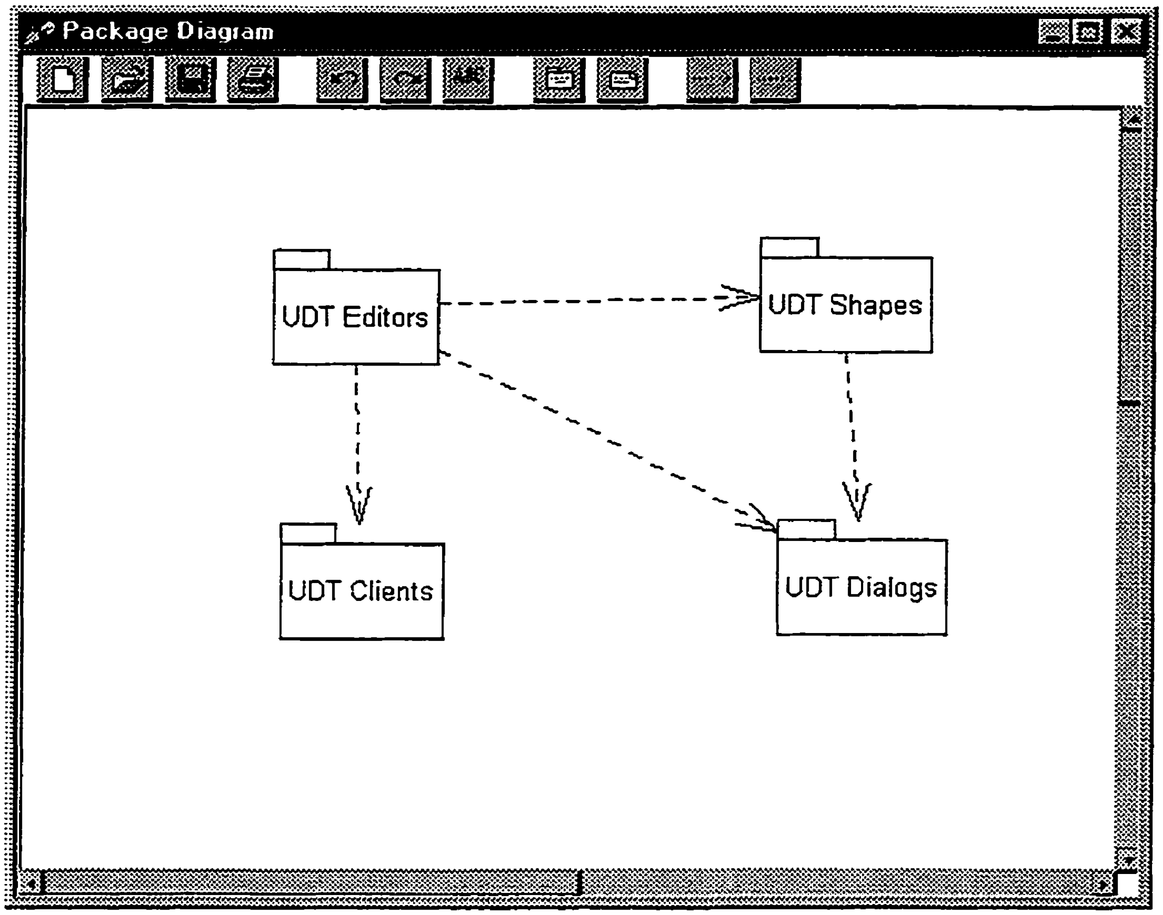


Figure 4-5: Package Diagram window

The Package Diagram window, Figure 4-5, allows the user to draw package diagrams using UML notation. The only new button here is the `Package` button (eighth button from left). It is used to display a UML Package shape in the drawing area.

4.4.3 Use Case Diagram window

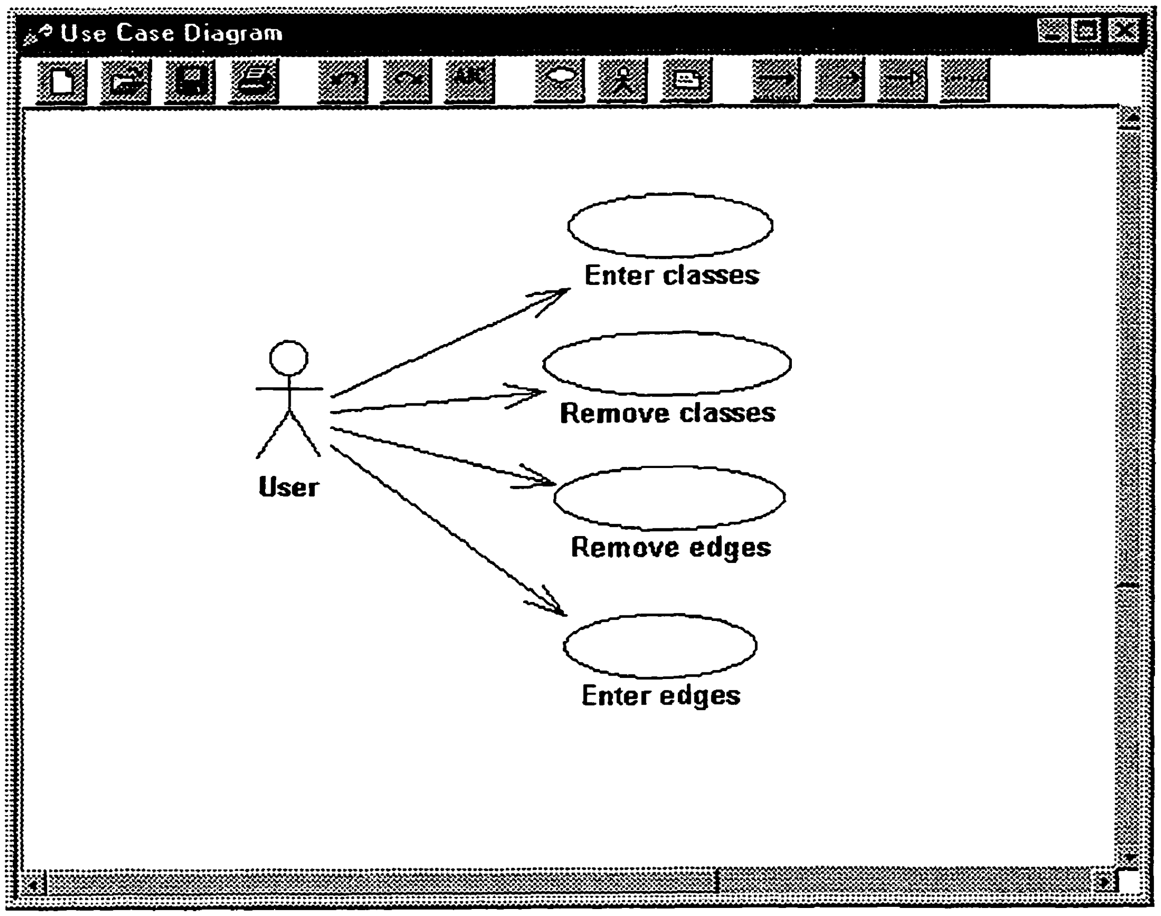


Figure 4-6: Use Case Diagram window

The Use Case Diagram window, Figure 4-6, opens when the user selects the use case diagram *page-style*. It allows the user to draw UML use case diagrams. The only new buttons here (from left to right) are the eighth, `Use case` button which is used to draw `Use case` shapes, and the ninth, `Actor` button which is used to draw `Actor` shapes.

4.4.4 Sequence Diagram window

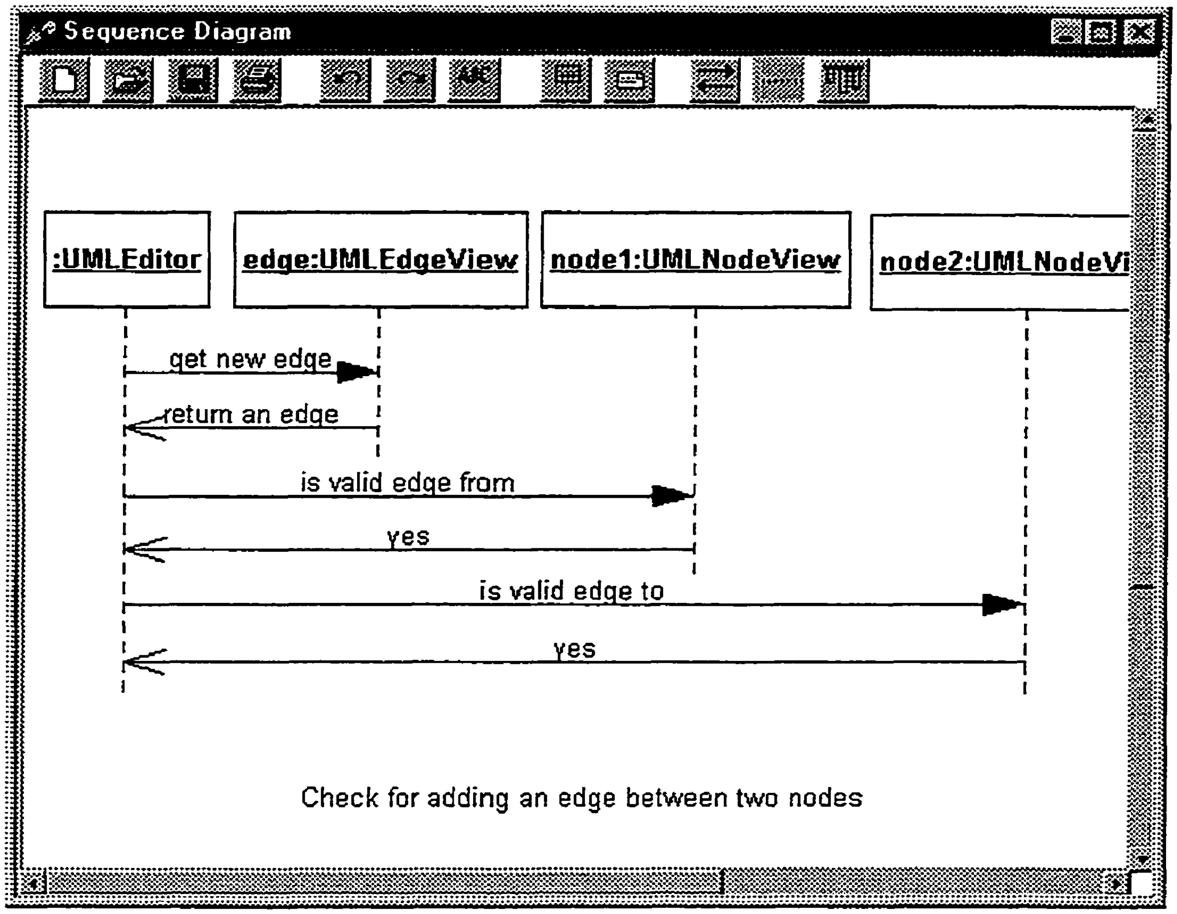


Figure 4-7: Sequence Diagram window

The Sequence Diagram window, Figure 4-7, opens when the user selects the sequence diagram *page-style*. It allows the user to draw UML sequence diagrams. The eighth button from left is used for drawing an `Object` shape with a vertical dashed line. The tenth button is used to draw a `Message` shape. The first button on the right is the `Align` button. It is used to align all the objects in the diagram with the top of the selected object.

4.4.5 State Transition Diagram window

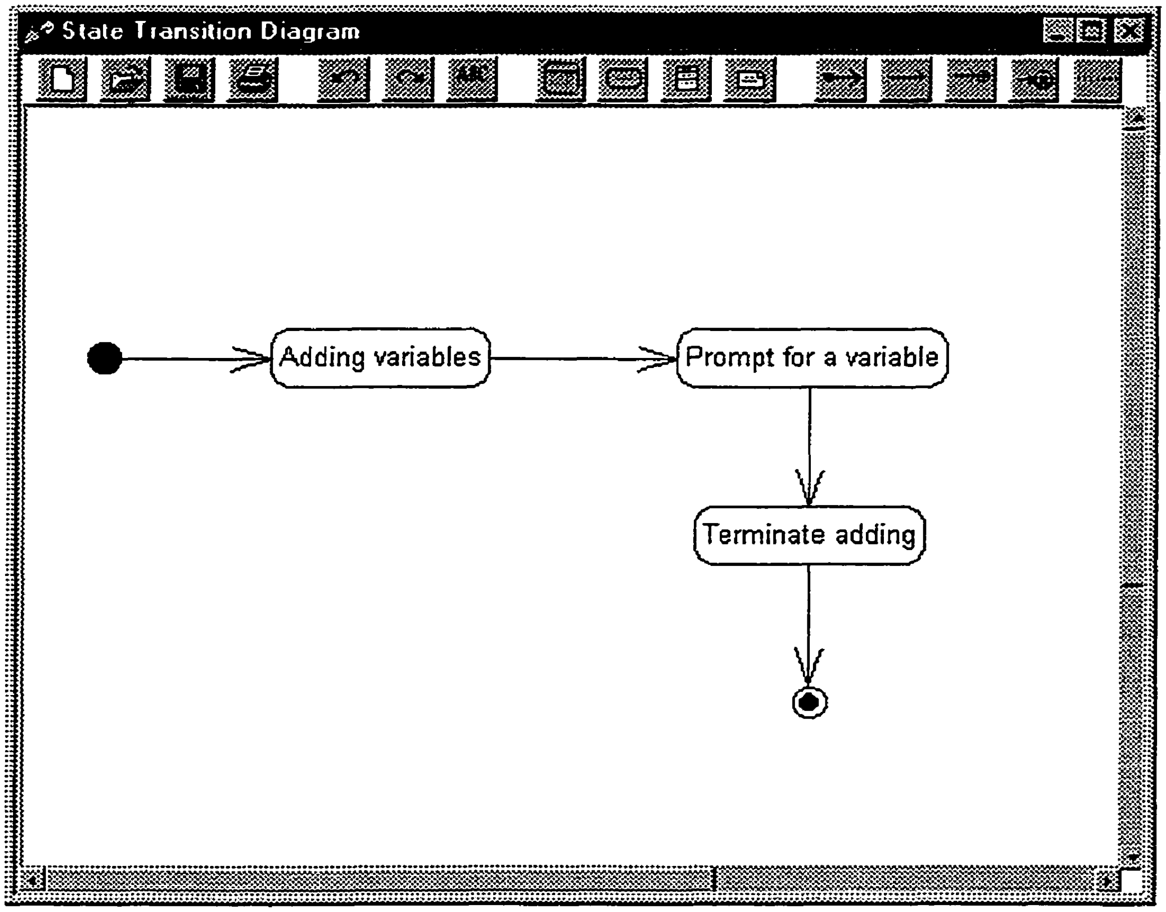


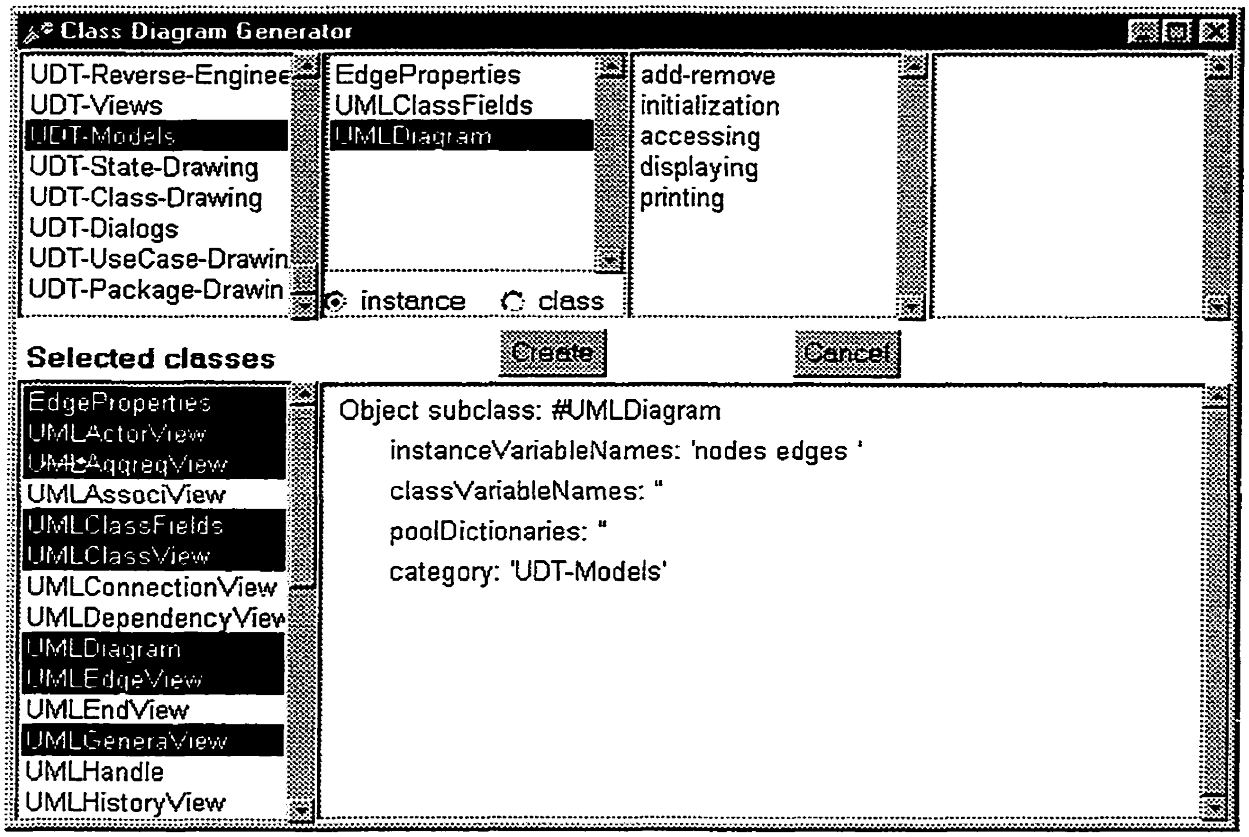
Figure 4-8: State Transition Diagram window

The State Transition Diagram window, Figure 4-8, opens when the user selects the state transition diagram *page-style*. It allows the user to draw UML state diagrams. The buttons from eight to fifteen (left to right) are `State Machine`, `State`, `Class`, `Note`, `Start State`, `State Connection`, `End State`, `History State` and `Note Connection`. All of these buttons are used for drawing State Transition Diagram shapes and are self-explanatory.

4.4.6 Class Diagram Generator window – Reverse Engineering in UDT

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Any tool designed to support the drawing of any design diagrams should support some kind of reverse-engineering for at least the static design diagrams (inheritance, association and aggregation relationships) and dynamic design diagrams (interaction diagrams).

Reverse engineering inheritance and association diagrams (class diagrams) from Smalltalk code is both aided and impeded by the nature of the Smalltalk language and its programming environment. While reverse engineering for other OO languages proceeds from an analysis of the source text of a program, this need not be the case for Smalltalk due to its fully reflective nature. In a Smalltalk environment, it is very simple to determine which classes inherit from whom by asking the class for its superclass but it is more problematic to determine association relationships. This is due to the fact that Smalltalk is not a statically typed language. In any statically typed language, the same strategy of parsing the program source to determine both types of relationships would work. But this will not work in Smalltalk because in Smalltalk variables are not typed until runtime. As a consequence, deriving a full set of associations in a Smalltalk program is more complicated and requires runtime traces. UDT is restricted to identifying some associations and class hierarchy relationship only.



UDT-Reverse-Enginee
UDT-Views
UDT-Models
UDT-State-Drawing
UDT-Class-Drawing
UDT-Dialogs
UDT-UseCase-Drawin
UDT-Package-Drawin

EdgeProperties
UMLClassFields
UMLDiagram

add-remove
initialization
accessing
displaying
printing

EdgeProperties
UMLActorView
UMLAggreqView
UMLAssociView
UMLClassFields
UMLClassView
UMLConnectionView
UMLDependencyView
UMLDiagram
UMLEdgeView
UMLEndView
UMLGeneraView
UMLHandle
UMLHistoryView

```
Object subclass: #UMLDiagram
    instanceVariableNames: 'nodes edges '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'UDT-Models'
```

Figure 4-9: Class Diagram Generator window

To generate parts of a class diagram from Smalltalk code, the user can use the class diagram generator window (Figure 4-9). This window opens when the user clicks on the `Load` button (which loads from a file or Smalltalk code) in Figure 4-2 and selects Code. It allows the user to generate a class diagram for the classes in the `Selected classes` list. First, the user must add classes to the `Selected classes` list by dragging a category from the category list (upper left), which will add all of its classes to the `Selected classes` list, or dragging specific class from the classes list (second upper left), which will add that class to the `Selected classes` list. After that, the user selects one or more classes (if no selection is made, all the classes in the list will be considered) from the `Selected classes` list and clicks on the `Create` button. This will create a class diagram for all the selected classes. The class diagram will contain inheritance and association relationships only for the selected classes. The class diagram is placed in the current Class and Object Diagram page of the Binder.

Chapter 5

UML Drawing Tool – Design

5.1 Introduction

The purpose of this chapter is to provide insight into UDT design and to provide a context and foundation that will be useful for future UDT extension.

There are many design methodologies for object-oriented software development and some of them are mentioned in Chapter 1. The methodology that we followed in designing UML Drawing Tool is Responsibility Driven Design (RDD) [McKean 1995]. This methodology divides design into two stages: exploratory design and final (detailed) design. In the exploratory design stage, the stress is on finding classes and their general responsibilities and collaborations. In the final design stage, the stress is on abstraction, inheritance relations and responsibilities shared between classes.

In the following, we summarize UDT design using UML class diagrams and descriptions of UDT classes. Since UDT implements five page-styles for the Binder program and includes about fifty classes, only the design of one page-style, namely the Class and Object Diagram page-style, will be explained in detail. The other page-styles, which include Use Case Diagram, State Diagram, Sequence Diagram and Package Diagram page-styles, are similar in their design. In the next sections we will provide the following for the Class and Object Diagrams page-style: the Object Model, some Sequence Diagrams, and textual class descriptions. All the diagrams in this chapter were produced by UDT.

5.2 Class and Object Diagrams Page-style Object Model

The class and object diagrams page-style Object Model describes the static structure of classes. It shows how classes are related to each other. Figures 5-1 and 5-2 show the Object Model for the class and object diagrams page-style using UML notation. Figure 5-1 shows the main classes of UDT and how they are related to the Binder. These classes are the base for any page-style. For any new page-style all that is needed is the substitution of **UMLClassEditorClient** with the new page-style client and the **UMLClassEditor** with the new page-style editor.

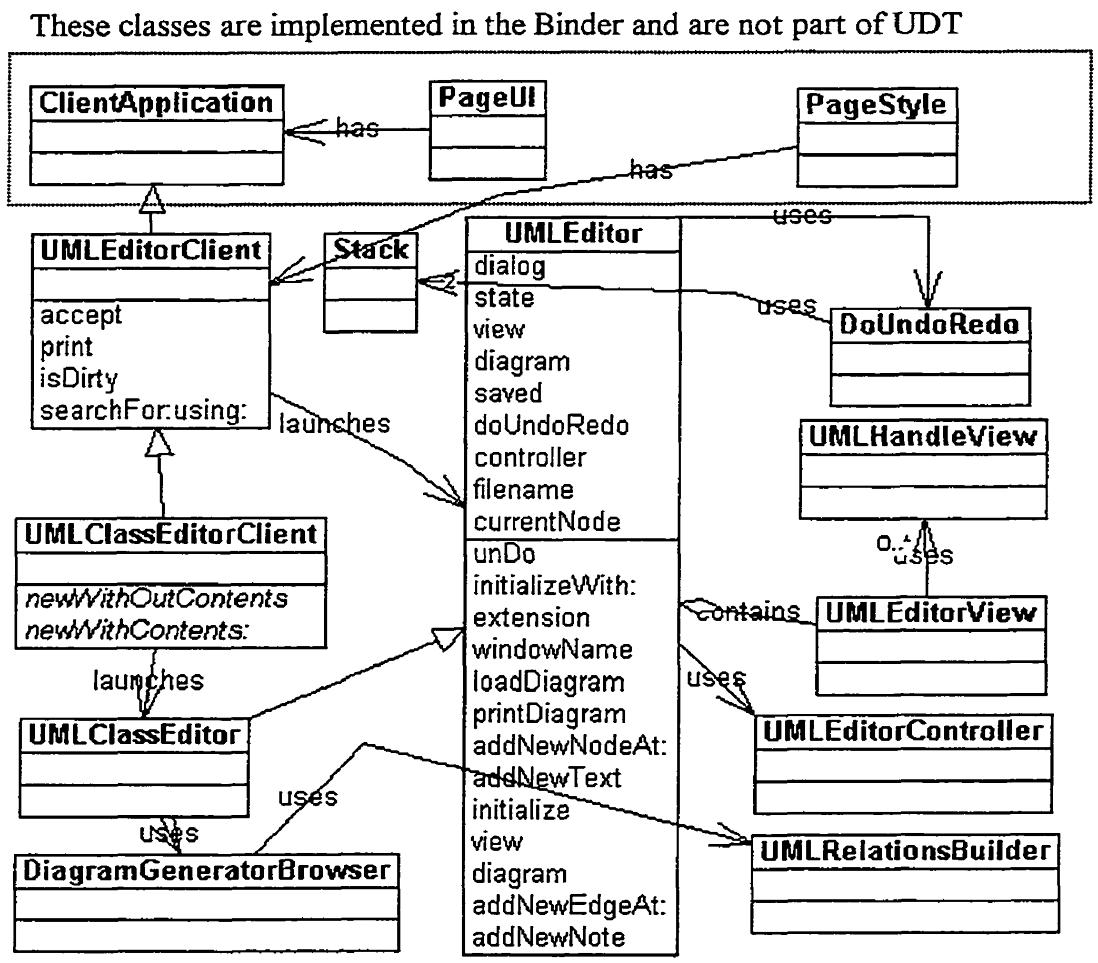


Figure 5-1: Part 1 of the class diagram for the Class and Object diagrams Page-style

Figure 5-2 shows the remaining classes that are used in the Class and Object Diagram page-style. They represent all the shapes (nodes and edges) that can be drawn in this diagram. The classes **UMLEditor**, **UMLDiagram**, **UMLNoteView** and **UMLConnectionView** are common to all diagrams but the other classes are different for each diagram.

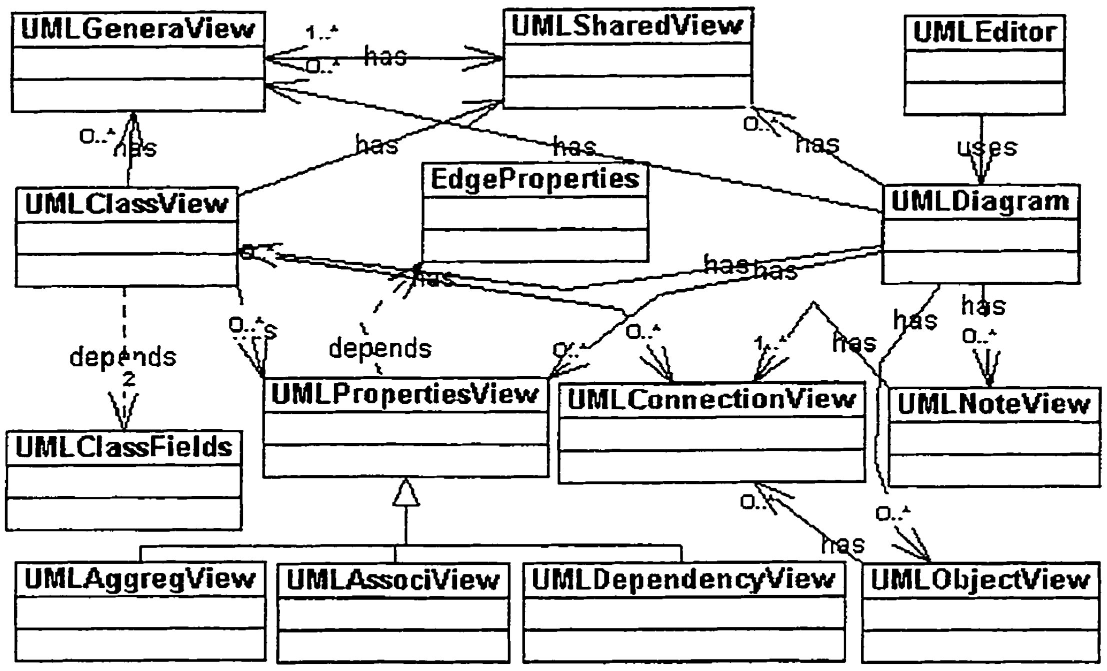


Figure 5-2: Part 2 of the class diagram for the Class and Object diagrams Page-style

5.3 Class and Object Diagrams Page-style Sequence Diagrams

In this section we show three UML Sequence Diagrams for the Class and Object diagrams page-style. Figure 5-3 shows the Sequence Diagram for adding a new class. It is the same for adding all other shapes include lines. The only difference is substituting **UMLClassView** with that shape's view.

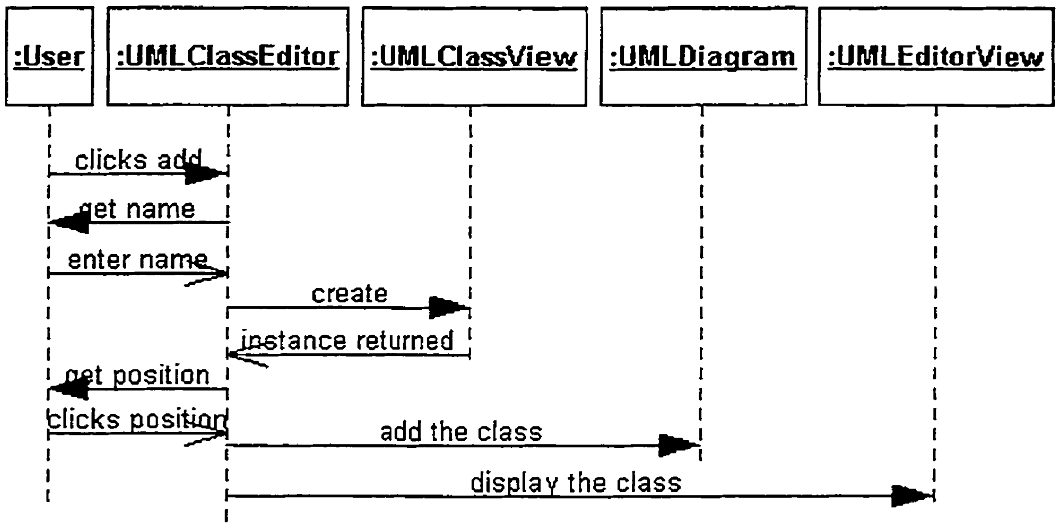


Figure 5-3: Sequence Diagram for adding a new class

Figure 5-4 shows the Sequence Diagram for adding variables to a class. It is the same for adding methods to a class.

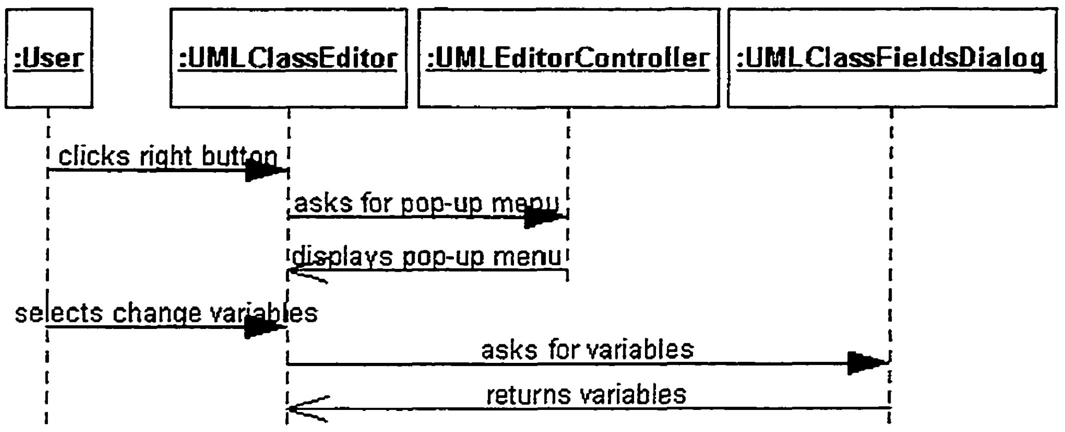


Figure 5-4: Sequence Diagram for adding class variables

Figure 5-5 shows the Sequence Diagram for undoing the last action (adding or removing a shape). It is the same for redoing the last action. The only difference is that, **DoUndoRedo** object sends the message `redo` instead of `undo`.

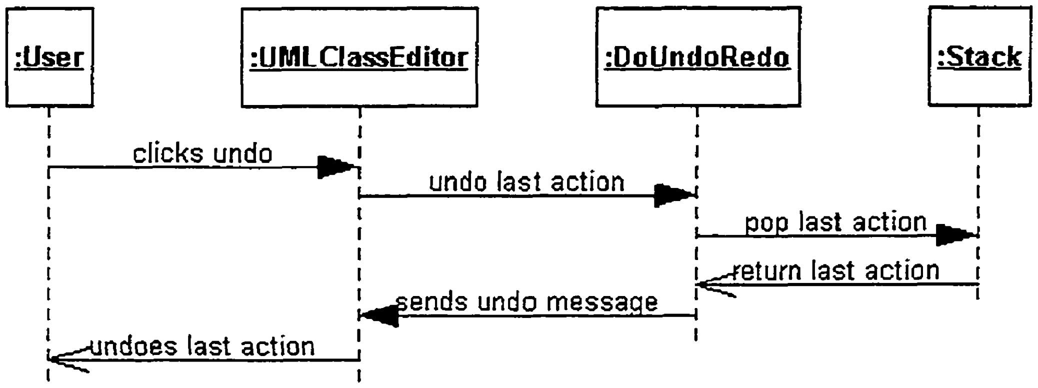


Figure 5-5: Sequence Diagram for undoing last action

5.4 Class Descriptions of Class and Object Diagrams

The following sections present a full description of all classes used in this page-style. Other page-styles use similar classes with similar states and behaviors.

5.4.1 UMLDiagram Class

- Class: **UMLDiagram**
- Superclass: **Object**
- Purpose: **UMLDiagram** class is used for representing a UML diagram. It contains all the information about all the nodes and edges and their positions on the window. It consists of two ordered collections. The first collection is used for storing the nodes of the UML diagram and the second collection is used for storing the edges of the UML diagram.
- Instance Variables:

nodes	**<OrderedCollection>**	A collection of nodes objects in a diagram.
edges	**<OrderedCollection>**	A collection of edges objects in a diagram.

- Class Behaviors:

creation

new	Create a new instance of **UMLDiagram** and initialize it.

- Behaviors:

accessing	Setters and getters for the instance variables.

initialization

initialize Initialize the receiver by initializing its nodes and edges as empty ordered collections.

displaying	
*displayOn:*aGraphicsContext	Display the diagram in a window by asking each node and edge to display itself at its origin.
add – remove	
*addNode:*aNode	Add a node to the diagram.
*removeNode:*aNode	Remove a node from the diagram.
*addEdge:*anEdge	Add an edge to the diagram.
*removeEdge:*anEdge	Remove an edge from the diagram.
searching	
*componentAt:*aPoint	Return the node or edge that contains aPoint. Returns nil if no shape is found at aPoint
*findEdgeNamed:*aName	Return an edge object named aName.
*findNodeNamed:*aName	Return a node object named aName.
testing	
= *aDiagram*	Return true if aDiagram is equal to the receiver.
enumerating	
*nodesDo:*aBlock	For each node in the diagram evaluate aBlock.
*edgesDo:*aBlock	For each edge in the diagram evaluate aBlock.
*nodesInsiade:*aRectangle	Return all the nodes inside aRectangle.
*edgesInside:*aRectangle	Return all the edges inside aRectangle.
*shapesInside:*aRectangle	Return all the shapes inside aRectangle.
*edgesFor:*aNode	Return all edges that are connected to aNode.

5.4.2 UMLClassFields Class

- Class: **UMLClassFields**
- Superclass: **Object**
- Purpose: **UMLClassFields** class holds information about a UML class variables or methods. It contains information about the class and instance side for the variables or methods. It has a dictionary with two keys. It stores the instance variables or methods at the #instance key and stores the class variables or methods at the #class key. The value for each key is a collection of the corresponding variables or methods.
- Instance Variables:

  Field **<Dictionary>** A dictionary of the names of the instance and class methods or variables of this class.

- Class Behaviors:

  creation

  new Create a new instance of **UMLClassFields** and initialize it.

- Behaviors:

  accessing Setters and getters for the instance variables.

  initialization

  initialize Initialize the receiver by setting the instance and class sides of *field* to empty sets.

  add – remove

  *add:*aName *at:*aSymbol Add aName to the set at aSymbol in the dictionary *field*.

`remove:aName at:aSymbol` Remove `aName` from the set at `aSymbol` in the dictionary *field*.

5.4.3 EdgeProperties Class

- <u>Class:</u> **EdgeProperties**
- <u>Superclass:</u> **Object**
- <u>Purpose:</u> **EdgeProperties** class holds information about edge properties. The properties are the role name, the cardinality and the value or values for that cardinality.
- <u>Instance Variables:</u>

role	**<String>**	A string for the role name of the edge.
cardinality	**<Symbol>**	A symbol for the cardinality type (#OneCardinality, #DiscreteCardinality, #FixedCardinality, #ZorOCardinality, #ZorMCardinality, #RangeCardinality).
startValue	**<Number | String>**	A number to hold the start value for a range or a string to hold the discrete values.
EndValue	**<Number | nil>**	A number to hold the end value for a range.

- <u>Class Behaviors:</u>

creation

`new`	Create a new instance of **EdgeProperties** and initialize it.

- <u>Behaviors:</u>

accessing	Setters and getters for the instance variables.
initialization	

`initialize`	Initialize the receiver by setting its cardinality to default (#OneCardinality).

5.4.4 UMLEditor Class

- Class: **UMLEditor**
- Superclass: **ApplicationModel**
- Subclasses: **UMLClassEditor**, **UMLUseCaseEditor**, **UMLSequenceEditor**, **UMLStateEditor**, **UMLPackageEditor**.
- Purpose: This class is the super class for all UDT editors. It provides general user interfaces such as toolbar buttons and drawing area for a selected UML diagram. It provides the common behavior for all UML diagrams that are supported by this application.
- Instance Variables:

doUndoRedo	**<DoUndoRedo>** A **DoUndoRedo** object to handle all the undo and redo operations.
view	**<UMLEditorView>** A **UMLEditorView** object for providing a drawing window for the selected UML diagram.
diagram	**<UMLDiagram>** A **UMLDiagram** object for storing the nodes and edges of the selected UML diagram.
controller	**<UMLEditorController>** A controller for handling the keyboard and mouse inputs.
currentNode	**<UMLNodeView>** The node that has the focus.
currentEdge	**<UMLEdgeView>** The edge that has the focus.

state	**<Symbol>**	The action to be performed like adding or deleting a node or an edge.
filename	**<String>**	The file name of the current diagram.
saved	**<Boolean>**	True if the file is saved, false otherwise.

- <u>Class Behaviors:</u>

creation

`newWith:aDiagram` Answer a new instance of **UMLEditor** with diagram set to `aDiagram`.

interface spec

`windowSpec`	Provides the user interface to allow for the drawing of UML diagrams.
`buttonSpec`	Provides the toolbar buttons.

- <u>Behaviors:</u>

accessing Setters and getters for the instance variables.

initialize-release

`initialize` Initialize the receiver by setting *controller* to the view's controller and setting the controller's menu to the main menu.

menu accessing Pop-up menus for the seleceted UML diagram.

menu message

`changeEdgeName`	Change the current edge's name.
`changeNodeName`	Change the current node's name.
`changeNoteText`	Change the note's text.
`changeText`	Change the selected text contents.

edgeProperties	Manipulate the current edge properties.
HardCopy	Print a hard copy of the current diagram as a text.
printScreen	Print a hard copy of the current diagram as a diagram.
removeEdge	Remove the current edge from the diagram.
removeNode	Remove the current node and all of its edges from the diagram.

actions

addNewConnection	Add an instance of **UMLConnectionView** (dashed line) between an instance of **UMLNoteView** and an instance of **UMLNodeView**.
*DragShapeAt:*aPoint	If shape at aPoint is selected then drag it to where the mouse is inside the window, otherwise make that shape the current selection.
*addNewConnectionAt:*aPoint	Add a connection at aPoint.
*addNewEdgeAt:*aPoint	Add the current edge between two nodes at aPoint.
*addNewNodeAt:*aPoint	Add the current node at aPoint.
addNewNote	Add a **UMLNoteView** object to the diagram.
addNewText	Add a **UMLTextView** object to the diagram.
loadDiagram	Load a UML diagram from a file.
newDiagram	Create a new UML diagram.
saveDiagram	Save the current UML diagram.

printDiagram	Print the current UML diagram.
redo	Redo the last action (adding or deleting).
undo	Undo the last action (adding or deleting).

undoing-redoing

redoAddEdge:`anEdge`	Add the last deleted edge.
undoAddEdge:`anEdge`	Remove the last added edge.
redoAddNode:`aNode`	Add the last deleted node.
undoAddNode:`aNode`	Remove the last added node.

private

windowName	Return the window name (subclass responsibility, subclasses must implement this).
extension	Return a valid file name extension for this type of diagram. Valid extensions are: '.uco' for Class and Object diagrams, '.uuc' for UseCase diagrams, '.ust' for State diagrams, '.upd' for Package diagrams, '.use' for Sequence diagrams and etc.
validView:`aShape`	Return true if `aShape` is a valid shape in the current UML diagram.
validShapes	Return a set of all the valid shapes for the current UML diagram (subclass responsibility, subclasses must implement this).
isValidFrom:`s1` *to*:`s2` *connection*:`s3`	Return true if `s3` can connect `s1` and `s2`.

5.4.5 UMLClassEditor Class

- Class: **UMLClassEditor**
- Superclass: **UMLEditor**
- Purpose: This class provides a toolbar buttons for drawing the shapes needed for UML class and object diagrams. It only allows for the drawing of the shapes that are valid for the Class and Object diagrams.
- Class Behaviors:

interface spec

`buttonSpec`	Provide toolbar buttons description.

- Behaviors:

menu accessing Pop-up menus for UML class diagram shapes.

menu message

`browseClass`	Open a Smalltalk browser on the current class.
`changeClassAttributes`	Manipulate the current class's attributes by adding or removing attributes.
`changeClassOperations`	Manipulate the current class's operations by adding or removing operations.
`defineClass`	Define the current class and its attributes and operations in the **Smalltalk** library. It generates part of the code for this class.
`instanceSide:aBoolean`	Work with the instance side of the class if `aBoolean` is true, otherwise work with the class side.

actions

addnewAggregation Add an aggregation edge to the diagram, an instance of **UMLAggregView**.

addNewAssociation Add an association edge to the diagram, an instance of **UMLAssociView**.

addNewClass Add an instance of **UMLClassView** to the diagram.

addNewDependency Add a dependency edge to the diagram, an instance of **UMLDependencyView**.

addNewGeneralization Add a generalization edge to the diagram, an instance of **UMLGeneraView**.

addNewObject Add an instance of **UMLObjectView** to the diagram.

private

windowName Return 'Class and Object Diagrams'.

extension Return '.uco'.

validShapes Return the set [#class, #object, #note, #text, #association, #generalization, #dependency, #connection, #shared]

addGeneraFromClass:c1 *toClass*:c2 Add the current edge which is a generalization between two classes.

addGeneraFromClass:c1 *toGenera*:c2 Add the current edge which is a generalization between a class and a generalization to create a shared generalization.

addGeneraFromClass:c1 *toShared*:c2 Add the current edge which is a generalization between a class and a shared generalization.

removeGenerEdge Remove the current edge which is a generalization.

removeSharedEdge Remove the current edge which is a shared generalization and remove all the edges that are connected to it.

textClass Return the current class information.

5.4.6 UMLEditorClient Class

- Class: **UMLEditorClient**
- Superclass: **ClientApplication**
- Subclasses: **UMLClassEditorClient**, **UMLUseCaseEditorClient**, **UMLStateEditorClient**, **UMLSequenceEditorClient**, **UMLPackageEditorClient**.
- Purpose: This class is the super class for all UDT editor clients. It acts as an interface between the Binder program and the **UMLEditor** application.
- Class Behaviors:

creation

newWithOutContents Answer a new initialized instance of the application (**UMLEditor**).

newWithContents:`newContents` Answer an instance of the application (**UMLEditor**) opened on `newContents`.

help

help Return a help text for this client's application.

- Behaviors:

accessing — Setters and getters for the instance variables.

accepting

`accept` — Accept the current changes to this diagram (see Chapter 3 for more details).

`isDirty` — True if the contents have been changed, false otherwise diagram (see Chapter 3 for more details).

searching

`searchFor:``t1` *`using:`*`t2` — Search the application for `t1` and return `#notfound` if `t1` is not found in the application. `t2` is used as an additional argument when searching for `t1`.

Printing

`print` — Print the application's contents.

5.4.7 UMLClassEditorClient Class

- Class: **UMLClassEditorClient**
- Superclass: **UMLEditorClient**
- Purpose: This class acts as an interface between the **Binder** program and the **UMLClassEditor** application.
- Class Behaviors:

creation

`newWithContents:newContents` — Answer an instance of the application opened on `newContents` (class or object diagram).

5.4.8 UMLEditorController Class

- Class: **UMLEditorController**
- Superclass: **ControllerWithMenu**
- Subclasses: **UMLClassController, UMLSequenceController, UMLStateController.**
- Purpose: This class is the super class for all UDT editors controllers. It handles all mouse events like pressing the yellow or red buttons. It shows the appropriate pop-up menu when pressing the left mouse button.
- Instance Variables:

cursor	**<Cursor>**	the cursor shape for a specific action.

- Behaviors:

accessing Setters and getters for the instance variables.

control defaults

`redButtonActivity` Invoke the appropriate action (sending a message, dragging a shape, selecting a shape and dropping a shape) when the left mouse button is pressed..

`yellowButtonActivity` Activate a specific pop-up menu when the right mouse button is pressed depending on the position of the cursor.

events

`enterEvent`:event Set the cursor to a cross hair shape when entering the current window if there is an action to be taken.

exitEvent:`event` Reset the cursor shape when existing the current window.

redButtonPressedEvent:`event` send the receiver the instance message *redButtonActivity*

yellowButtonPressedEvent:`event` send the receiver the instance message *yellowButtonActivity*

private

getLineFromUserAt:`aPoint` Return an end point of a line starting at the point `aPoint`.

rectangleAt:`aPoint` Get a rectangle from the user starting at the point `aPoint`.

uml drag

drag:`aShape` *start*:`aPoint` Drag `aShape` starting at the position `aPoint` by sending *dragAt*:`aPoint` *for*:`self` to `aShape`.

5.4.9 UMLEditorView Class

- Class: **UMLEditorView**
- Superclass: **View**
- Subclasses: **ClassEditorView**, **SequenceEditorView**, **StateEditorView**.

- Purpose: This class is the super class for all UDT editors views. It provides a drawing area for the selected UML diagram. It has a vertical and horizontal scrollers for drawing large diagrams.
- Instance Variables:

selectedShapes **<OrderedCollection>** The collection for the selected shapes inside this view.

- Behaviors:

controller

defaultControllerClass Return the controller (**UMLEditorController**) for this view.

displaying

*displayOn:*aGraphicContext Paint the view.

accessing

selectedShapes Return an ordered collection of the selected shapes in the diagram.

*handlesAt:*aPoint Return the handle at aPoint.

*visualComponentAt:*aPoint Return the shape at aPoint.

*unSelect:*aShape Remove aShape from the selected shapes.

*unSelectExcept:*handles Make handles the only selected shapes.

*updateEdges:*edges *using:*aNode Change the positions of edges according to the position of the node aNode.

*updateEdge:*anEdge *using:*nodes Change the position of anEdge according to the positions of nodes.

5.4.10 UMLHandle Class

- Class: **UMLHandle**
- Superclass: **View**
- Purpose: This class is responsible for displaying handles to mark the selected shape as selected. It displays a rectangle with a different color on the shape selected if it is a node and displays small rectangles a long the line if the selected shape is an edge.
- Instance Variables:

shape	**<UMLShapeView>**	The selected UML shape.
points	**<OrderedCollection>**	A collection of the points that should be displayed in a different color.

- Class Behaviors:

creation

`on:aShape` Return an instance of the receiver on the UML shape `aShape`.

- Behaviors:

accessing Getters and setters for the instance variables.

testing

`containsPoint:aPoint` Test whether this shape contains `aPoint`.

displaying

`displayOn:aGraphicsContext` Display the receiver on the window.

`displaySquareOn:aGraphicsContext at:aPoint` Display small square at the point `aPoint`.

add – remove

`add:aPoint` Add the point `aPoint` to the collection *points*.

`remove:aPoint` Remove the point `aPoint` from the collection *points*.

converting

`asOrderedCollection` Return an ordered collection on this handle.

5.4.11 UMLShapeView Class

- Class: **UMLShapeView**
- Superclass: **DependentPart**
- Subclasses: **UMLNodeView, UMLEdgeView.**
- Purpose: This class is the superclass for all UDT views and that includes both shapes and edges. It provides the behavior required by all shapes and edges in any UML diagram.
- Instance Variables:

kind	**<Symbol>**	A shape kind (#class, #object, #association, ...).
isSelected	**<Boolean>**	True if the shape is selected, false otherwise.
origin	**<Point>**	The origin of the shape.
corner	**<Point>**	The corner of the shape.
name	**<String>**	The name of the shape.

- Behaviors:

accessing

`extent`	Return the width and height of the receiver as a point.
`layout`	Return the smallest rectangle that contains the shape.
`fullName`	Return the name and the kind of the shape.
`prefferedBounds`	Returns the bounds of the shape.

searching

searchFor:`t1` *using*:`t2` Search the shape for `t1` and return #notfound if `t1` is not found in the shape. `t2` is used as an additional argument when searching for `t1`.

nearestPointTo:`aPoint` Return the nearest point on the shape's bounds that is located on the straight line between `aPoint` and the center of the receiver.

transforming

moveBy:`aPoint` Move the shape by `aPoint`.

testing

containsPoint:`aPoint` Test whether this shape contains `aPoint`.

isEdge Test if this shape is an edge, returns false.

isNode Test if this shape is a node, returns false.

copying

postCopy Make a deep copy, copy all instance variables.

converting

asOrderedCollection Return an **OrderedCollection** with this shape.

comparing.

= `aShape` Check for equality.

5.4.12 UMLNodeView Class

- Class: **UMLNodeView**
- Superclass: **UMLShapeView**

- Subclasses: **UMLClassView**, **UMLEndView**, **UMLTextView**, **UMLNoteView**, **UMLActorView**, **UMLStateView**, **UMLPackageView**, **UMLObjectView**, **UMLStartView**, **UMLHistoryView**, **UMLUseCaseView**.
- Purpose: This class is the super class for all UDT nodes views. It provides the behavior required by all nodes in any UML diagram.
- Behaviors:

accessing

handle	Return array of the four corners of the node.
height	Return the height of the node.
width	Return the width of the node.

testing

isNode	Since this is a node return true.

converting

asHandle	Return a **UMLHandle** object on this node.

drag

dragAt:`aPoint` *for:*`aController` Drag the receiver (current node) starting at the position `aPoint`.

dragWith:`edges` *at:*`aPoint` *for:*`aController` Drag the receiver (current node) with all the edges in `edges` starting at position `aPoint`.

5.4.13 UMLClassView Class

- Class: **UMLClassView**
- Superclass: **UMLNodeView**
- Purpose: This class is responsible for drawing a class in UML notation on a window. It draws a rectangle with three sections. The first section for the class name, the second section for the variables and the third section for the methods. The size of the rectangle depends on the contents of the largest section.
- Instance Variables:

variables	**<UMLClassFields>**	A Class that has a dictionary for holding information about the instance and class variables.
methods	**<UMLClassFields>**	A Class that has a dictionary for holding information about the instance and class methods.
side	**<Symbol>**	Which side of the field (#class or #instance).
superClass	**<String>**	The name of the super class.

- Behaviors:

accessing

`side`	Return #class or #instance depends on which side to process.
`superClass`	Return the receiver's super class.
`variables`	Return the variables (class and instance) of the receiver.
`methods`	Return the class and instance methods of the receiver.
`firstLine`	Return the *y position* of the end of the first section.
`secondLine`	Return the *y position* of the end of the second section.

displaying

displayOn:`aGraphicsContext` Display the receiver on the window.

private

nameAsTextList Return the name as a **TextList** object.

variablesAsTextList Return the variables as a **TextList** object.

valuesAsTextList Return the variables and methods as a **TextList** object.

drag

dragAt:`aPoint` *for:*`aController` Drag the receiver (current class) starting at the position `aPoint`.

5.4.14 UMLObjectView Class

- Class: **UMLObjectView**
- Superclass: **UMLNodeView**
- Purpose: This class is responsible for drawing an object shape in UML notation on a window. It draws a rectangle with the name of the object inside it. The size of the shape depends on the width of the name with a fixed height.
- Behaviors:

accessing Getters and setters for the instance variables.

private

centerBottom Return the point at the middle of the bottom line of the reactangle.

asTextList Return the name of the object as a **TextList** object.

displaying

`displayOn:aGraphicsContext`	Display the receiver on the window.

5.4.15 UMLNoteView Class

- Class: **UMLNoteView**
- Superclass: **UMLNodeView**
- Purpose: This class is responsible for drawing a note shape in UML notation on a window. The size of the shape is fixed.
- Instance Variables:

text	**<Text>**	A text that holds the note contents.

- Behaviors:

accessing	Getter and setter for the instance variable.
displaying	
`displayOn:aGraphicsContext`	Display the receiver on the window.

5.4.16 UMLTextView Class

- Class: **UMLTextView**
- Superclass: **UMLNodeView**
- Purpose: This class is responsible for displaying a text anywhere on a window. It asks for the text and then display it on a window.
- Instance Variables:

string	**<String>**	The text to be displayed on a window.

- Behaviors:

accessing	Getter and setter for the instance variable.

private

`asTextList`	Return the text as a **TextList** object.

displaying

`displayOn: aGraphicsContext`	Display the receiver on the window.

5.4.17 UMLEdgeView Class

- Class: **UMLEdgeView**
- Superclass: **UMLShapeView**
- Subclasses: **UMLPropertiesView**, **UMLSharedView**, **UMLGeneraView**, **UMLSequenceLineView**, **UMLConnectionView**, **UMLStateConnectionView**, **UMLSequenceConnectionView**.
- Purpose: This class is the super class for all UDT edges views. It provides the behaviors required by all edges in any UML diagram.
- Instance Variables:

from	**<String>**	The name of the node at the from-side.
to	**<String>**	The name of the node at the to-side.
showName	**<Boolean>**	If true show the name of the edge.
edge	**<OrderedCollection>**	An ordered collection of the points of this edge.

- Behaviors:

accessing

`from`	Return the name of the node at the from-side.
`to`	Return the name of the node at the to-side.

fromKind	Return the kind of the node at the from-side.
toKind	Return the kind of the node at the to-side.
at:i	Return the point at index `i` of *edge*.
at:i put:point	Put the Point `point` at index `i` of *edge*.
endsOf: `aPoint`	If `aPoint` exists in *edge* then return the index of the point before `aPoint` in *edge*, otherwise if `aPoint` is in the middle between two points in the *edge* then return the index of `aPoint` in *edge*, otherwise return nil.
handle	Return array of the points of *edge*.
size	Return the size of *edge*.

testing

isEdge	Since this is an edge, return true.
connectedTo: `name`	Return true if the this edge is connected to the node named name.
isLine	Returns true if *edge* has only two points.
isPolyline	Return true if *edge* has more than two points.

converting

asHandle	Return a **UMLHandle** object on this edge.
reset	Make *edge* straight line by removing all the middle points between the first and the last point.

displaying

displayOn: `aGraphicsContext`	Display the receiver as a solid line on a window.

displaySolidOn: `aGraphicsContext` Display the receiver as a solid line on a window.

displayDottedOn: `aGraphicsContext` Display the receiver as a dashed line on a window.

displayLeftArrowOn: `aGraphicsContext` Display a left arrow shape at the end of the receiver on a window.

displayRightArrowOn: `aGraphicsContext` Display a right arrow shape at the beginning of the receiver on a window.

displayNameOn: `aGraphicsContext` Display the name of the receiver on a window.

private

displayOnX:`aGraphicsContext` *extent:*`extent` *at:*`aPoint`

Used for displaying a dashed line horizontally.

displayOnY:`aGraphicsContext` *extent:*`extent` *at:*`aPoint`

Used for displaying a dashed line vertically.

fromName Return the name of the node at the from-side.

toName Return the name of the node at the to-side.

add – remove

add:`point1` *after:*`point2` Add the point `point1` to *edge* after the point `point2`.

add:`point1` *before:*`point2` Add the point `point1` to this edge after the point `point2`.

remove:`aPoint` Remove the point `aPoint` from this edge.

drag

dragAt:`aPoint` *for*:`aController` Drag the receiver (current edge) starting at the position `aPoint`.

dragEndPointAt:`p1` *start*:`p2` *for*:`aController` Drag the end point `p1` of the receiver (current edge) starting at position `p2`.

dragMiddelPointAt:`p1` *start*:`p2` *for*:`aController` Drag the middle point `p1` of the receiver (current edge) starting at position `p2`.

update

updateUsing:`aNode` *for*:`aController` Update the state of the receiver (current edge) according to the state of `aNode` (an instance of **UMLNodeView**).

5.4.18 UMLGeneraView Class

- Class: **UMLGeneraView**
- Superclass: **UMLEdgeView**
- Purpose: This class is responsible for drawing a line representing a generalization edge in UML notation.
- Instance Variables:

shared	**<Boolean>**	True if this edge is shared between classes.

- Behaviors:

accessing	A setter and getter for the instance variable.

converting

asMulit Set the *shared* variable to true.

asSingle Set the *shared* variable to false.

asSharedView Return an instance of **UMLSharedView** on the receiver.

update

*updateUsing:*aNode *for:*aController Update the state of the receiver (generalization edge) according to the state of aNode.

5.4.19 UMLSharedView Class

- Class: **UMLSharedView**
- Superclass: **UMLEdgeView**
- Purpose: This class is responsible for drawing a shared edge for representing a connection between a group of subclasses and a superclass. It represents a shared generalization in UML
- Instance Variables:

fromSide **<SortedCollection>** A sorted collection of associations where the keys are the edges full names and values are points.

- Behaviors

accessing A setter and a getter for the instance variable.

add-remove

*add:*anAssociation Add the association anAssociation to the *fromSide* collection.

remove:anAssociation Remove anAssociation from the *fromSide* collection.

update

updateUsing:aNode *for*:aController Update the state of the receiver (shared edge) according to the state of aNode.

5.4.20 UMLConnectionView Class

- Class: **UMLConnectionView**
- Superclass: **UMLEdgeView**
- Purpose: This class is responsible for drawing a dashed line representing a connection edge in UML notation. This edge is used for connection a note object and any UML shape.

5.4.21 UMLPropertiesView Class

- Class: **UMLPropertiesView**
- Superclass: **UMLEdgeView**
- Purpose: This class is the super class for all edges views that have some extra properties like cardinality and role names.
- Instance Variables:

direction **<Symbol>** A symbol to show the direction of the edge. The available directions are: #none, #left, #right and #both.

properties **<Dictionary>** A dictionary for holding this edge's properties with two associations. One for the *to-side* of the edge and the other for the *from-side* of the edge. The value for the dictionary elements is an **EdgeProperties** object.

current **<Symbol>** A symbol for indicating which side of the edge should have its property displayed.

- Behaviors:

accessing A setter and getter for the instance variable.

displaying

displayDiscreteCardinalityOn:`aGraphicsContext` Display the discrete cardinality for this edge on the *current* side.

displayFixedCardinalityOn:`aGraphicsContext` Display the fixed cardinality for this edge on the *current* side.

displayOneCardinalityOn:`aGraphicsContext` Display the one cardinality (nothing) for this edge on the *current* side.

displayRangeCardinalityOn:`aGraphicsContext` Display the range cardinality for this edge on the *current* side.

displayZorMCardinalityOn:`aGraphicsContext` Display the zero or more cardinality for this edge on the *current* side.

displayZorOCardinalityOn:`aGraphicsContext` Display the zero or one cardinality for this edge on the *current* side.

displayRoleNameOn:`aGraphicsContext` Displays the role name for this edge on the *current* side.

private

`cardinalityPosition:aText` Return a point on the window for displaying the cardinality of this edge.

`rolePosition:aText` Return a point on the window for displaying the role name of this edge.

5.4.22 UMLAssociView Class

- Class: **UMLAssociView**
- Superclass: **UMLPropertiesView**
- Purpose: This class is responsible for drawing a line representing an association edge in UML notation.

5.4.23 UMLAggregView Class

- Class: **UMLAggregView**
- Superclass: **UMLPropertiesView**
- Purpose: This class is responsible for drawing a line representing an aggregation edge in UML notation.

5.4.24 UMLDependencyView Class

- Class: **UMLDependencyView**
- Superclass: **UMLPropertiesView**
- Purpose: This class is responsible for drawing a line representing a dependency edge in UML notation.

5.4.25 UMLClassDefinningDialog Class

- Class: **UMLClassDefinningDialog**
- Superclass: **SimpleDialog**
- Purpose: This class is responsible for drawing the interface needed for defining a class and its variables and methods.
- Instance Variables:

name	**<String>**	The name of the class to be defined.
umlClass	**<UMLClassView>**	The UMLClassView object to be defined.
className	**<ValueHolder>**	A value holder on an input field for the class name.
methodsList	**<MultiSelectionList>**	A list of all the methods. If *choice* is #class then class methods are listed otherwise instance methods are listed.
choice	**<Symbol>**	Used to indicate instance or class side. Possible values are: #instance or #class.
superClassName	**<ValueHolder>**	A value holder on the superclass's name.
readAccessing	**<Boolean>**	If true define getters for the selected variables.
writeAccessing	**<Boolean>**	If true define setters for the selected variables.
variablesList	**<MultiSelectionList>**	A list of all the variables. If *choice* is #class then class variables are listed otherwise instance variables are listed.

- Class Behaviors:

interface specs

windowSpec The description of the user interface.

creation

class:`aClassView` Create an instance of the receiver from `aClassView` which is an instance of the class **UMLClassView**.

- Behaviors:

accessing

name The name of the **UMLClassView** object.

initialize-release

umlClass:`aClassView` Initialize the receiver with the state of `aClassView`.

action

apply Define the class named *name* with the selected variables and methods.

defining

addMethodsNames:`names` Define the methods in the collection `names` for the class named *name*.

addReadingAccessFor:`vars` Define getters for the variables in the `vars` list.

addWritingAcessFor:`vars` Define putters for the variables in the `vars` list.

private

changedChoice Used to switch between the class and instance sides of the class.

5.4.26 UMLRelationsBuilder Class

- Class: **UMLRelationsBuilder**
- Superclass: **Object**
- Purpose: This class provides all the functionality for dealing with reverse engineering. It implements all the necessary functions for finding the associations and generalizations for a collection of classes.
- Instance Variables:

mainClasses	**<Set>** A set of the classes for which a reverse engineering should take place.
associations	**<Set>** A set of all the associations for the classes in *mainClasses*.
associations	**<Set>** A set of all the generalizations for the classes in *mainClasses*.

- Class Behaviors:

creation

new	Create a new instance of **UMLRelationsBuilder** and initialize it.
on:`aCollection`	Create an instance of the receiver on the classes in `aCollection`.

- Behaviors:

accessing	Setters and getters for the instance variables.

initialize

initialize	Initialize the receiver by setting the instance variables *mainClasses*, *associations* and *aggregations* to empty sets.

add - remove

*add:*aSymbol Add aSymbol which is a class name to *mainClasses*.

*addAssociation:*anAssociation Add anAssociation (class1->class2, class1 uses class2) to *associations*.

*addGeneralization:*anAssociation Add anAssociation (class1->class2, class1 subclass of class2) to *generalizations*.

*remove:*aSymbol remove aSymbol which is a class name from *mainClasses*.

*removeAssociation:*anAssociation Remove anAssociation (class1->class2, class1 uses class2) from *associations*.

*removeGeneralization:*anAssociation Remove anAssociation (class1->class2, class1 subclass of class2) from *generalizations*.

converting

asUMLDiagram Convert the receiver to an instance of **UMLDiagram** by adding all the nodes (classes) and edges (associations and generalizations) to a new instance of **UMLDiagram**.

class relations

buildAssociations For each class in *mainClasses* find the classes that use this class and the classes that this class uses and add them to *associations*.

buildGeneralizations For each class in *mainClasses* find its superclass and add it to *generalizations*.

*getAssociations:*umlDiagram Add all the edges in *associations* to umlDiagram which is an instance of **UMLDiagram**.

getGeneralizations:umlDiagram Add all the edges in *generalizations* to umlDiagram which is an instance of **UMLDiagram**.

getEdges:umlDiagram Add all edges (associations and generalizations) to umlDiagram.

getNodes:umlDiagram Add all the nodes (the classes in *mainClasses* and their superclasses) to umlDiagram.

private

allMessages:aClass Return a collection of all messages that are sent by aClass.

classesUses:aClass Return a collection of all the classes that use aClass.

classesReferences:aClass Return a collection of all the classes that aClass uses.

classes Return a set of all the classes that have an association relation with the classes in *mainClasses*.

superClasses Return a set of the superclasses of all the classes returned by the previous method.

5.4.27 DiagramGeneratorBrowser Class

- Class: **DiagramGeneratorBrowser**
- Superclass: **Browser**

- Purpose: This class provides a browser that is used for reverse engineering. It's layout is identical to the Smalltalk Browser. It adds a list for selecting the classes for which a class diagram should be drawn.
- Instance Variables:

selectedClassesList	< **MultiSelectionInList**> A **MultiSelectionInList** object to store the selected classes for reverse engineering. It allows a multi selection of classes for which the reverse engineering should happen.
caller	<**UMLClassEditor**> The application that creates an instance of the receiver.

- Class Behaviors:

interface opening

`openFrom:anApplication`	Answer a new instance of **DiagramGeneratorBrowser** from `anApplication`.

interface spec

`windowSpec`	Provide the user interface to allow for selecting classes for reverse engineering.

resources

selectedClassMenu	Return a pop-up menu for the selected classes list.

- Behaviors:

accessing	Setters and getters for the instance variables.

actions

`createDiagram` Create a partial class diagram for all the selected classes in *selectedClassesList* and close the Browser.

`cancelDiagram` Close the reverse engineering browser.

private

`addSelectedCategory` Add all the classes of the selected category to *selectedClassesList.*

`addSelectedClass` Add the selected class to *selectedClassesList.*

`removeSelectedClasses` Remove all the selected classes from *selecetedClassesList.*

`getSelectedClasses` Return a collection of all the selected classes in *selectedClassesList* or all the classes if there were no selections.

5.5 Extending UDT

Several UML diagrams are not supported by this version of UDT, but it is easy to extend UDT to implement them since most of the common functionality is already implemented in the base classes **UMLEditor**, **UMLEditorClient**, **UMLNodeView** and **UMLEdgeView**. For each new diagram, all that is needed is subclassing of all or some of these base classes or their subclasses.

Subclassing **UMLEditor** is needed to provide an editor for the new diagram. The class method `buttonSpec` of **UMLEditor,** which provides a specific toolbar buttons for the new diagram, should be overridden. Other methods that should be implemented by **UMLEditor** subclasses are: `windowName`, `validShapes` and `extension`. Method windowName is an instance method that returns the name of the new page-style. Method validShapes is an instance method that returns an ordered collection of the names of the shapes that can be drawn in this diagram. Method extension is an instance method that returns the extension of the file that stores this diagram.

In order to be able to use the new diagram as a page-style in the Binder program, the new diagram's editor must have a client class that is a subclass of **UMLEditorClient** (Refer to Chapter 3 for more detail on the client class). This client class should have the new diagram's editor as its application and should implement the methods described in Chapter 3.

Every node in the new diagram should be represented by a view that is a subclass of **UMLNodeView**. For example, UML Class Diagram has the following nodes: class and note. The class shape has a view (**UMLClassView**) that is a subclass of **UMLNodeView** and the note shape also has a view (**UMLNoteView**) that is a subclass of **UMLNodeView**. Instance methods *displayOn*: aGraphicsContext and *kind* must be overridden. Method *displayOn*: is used to paint the new node on aGraphicsContext. Method *kind* is used to return the kind of node (#class, #note, #object, etc). Method *dragAt*:aPoint *for*:aController might be overridden if the new node should be dragged differently. As an example of that, the class node overrides this method to allow the dragging of the tree format of the generalization edge which differs in shape from other edges like association or aggregation.

Every edge in the new diagram should be represented by a view that is a subclass of **UMLEdgeView**. For example, UML Class Diagram has the following edges: association, generalization, dependency and aggregation. All these edges have views

(**UMLAssociView**, **UMLGenerView**, **UMLDependencyView**, **UMLAggregView**) that are subclasses of **UMLEdgeView**. Instance methods *displayOn:* aGraphicsContext and *kind* must be overridden. Method displayOn: is used to paint the new edge on the aGraphicsContext. Method *kind* is used to return the kind of edge (#association, #generalization, #aggregation, etc). Method *dragAt:*aPoint *for:*aController might be overridden if the new edge should be dragged differently. As an example of that, the message edge in the Sequence Diagram overrides this method to allow the dragging to be up and down only. Another method that might have to be overridden is *updateUsing:*aNode *for:*aController which updates the current edge according to the position of aNode. As an example of that, the message edge in the Sequence Diagram overrides this method to allow the update to be according to the dashed vertical line from aNode, an object node, and not aNode itself.

Chapter 6

UML Drawing Tool Implementation

6.1 Introduction

The UML Drawing Tool (UDT) is implemented using Smalltalk, VisualWorks 3.0 [Cincom 2000]. The implementation is based on the design described in Chapter 5. The following sections present samples of the implementation of selected UDT classes. In source code classes are in **Bold** and methods are in *Italic.*

6.2 UMLDiagram Class

This class is the model for all UDT page-styles, an object that manages (calculates, sorts, stores, retrieves, simulates, converts and so on) information [Howard 1995]. It holds all domain information about a diagram displayed in a single window or Binder page. It contains two ordered collections, one for storing all the nodes and the other for storing all the edges of the diagram. In the following subsections, the implementation of some of the methods of this class will be shown.

6.2.1 Method displayOn:

This instance message is used to display the receiver on an instance of **GraphicContext**. It is sent to an instance of **UMLDiagram** to display all the nodes and edges on the drawing window for that diagram using the window's **GraphicsContext** object. It first asks each node to display itself on the drawing area relative to its origin and then asks each edge to display itself.

```
displayOn: aGraphicsContext
```

```
    self nodes do: [:node | aGraphicsContext display: node
                                  at: node origin].
    self edges do: [:edge | aGraphicsContext display: edge
                                       at: edge origin]
```

6.2.2 Method componentAt:

This instance message is used when the user wants to select, drag or display a pop-up menu for a specific shape. It is called when a mouse button is pressed inside the drawing area. It is sent to an instance of **UMLDiagram** to return a component at a specific point. If there are more than one component, the most recently added component is returned. The argument `aPoint` represents a point on the drawing area of the diagram, the position of the mouse cursor. The method first searches the edges collection and returns the first edge that contains this point in its bounding area. If no edge is found, it searches the nodes collection for that point.

```
componentAt: aPoint
    self edges do: [:edge | (edge containsPoint: aPoint)
                                  ifTrue: [^edge]].
    self nodes do: [:node | (node containsPoint: aPoint)
                                  ifTrue: [^node]].
    ^nil
```

6.2.3 Method shapesInside:

This instance message is used when all shapes inside a specific rectangle are to be dragged together. It is sent to an instance of **UMLDiagram** to return a collection of all the shapes (nodes and edges) that are displayed inside a specific rectangle. Argument `aRectangle` represents a rectangle on the drawing area of the related diagram. The method first creates a collection of all the nodes inside `aRectangle` and then adds all the edges inside the rectangle to that collection. It then returns the final collection.

```
shapesInside: aRectangle
   | aCollection |
   aCollection := self nodesInside: aRectangle.
   aCollection addAll: (self edgesInside: aRectangle).
   ^aCollection
```

6.2.4 Method nodesInside:

This instance message is used when all nodes inside a specific rectangle are to be dragged together. It is sent to an instance of **UMLDiagram** to return a collection of all the nodes that are displayed inside a specific rectangle. The argument `aRectangle` represents a rectangle on the drawing area of the diagram. Method `layout` returns the smallest rectangle that completely covers the receiver.

```
nodesInside: aRectangle
   ^self nodes select: [:node | aRectangle contains: node
                                                  layout]
```

6.3 UMLEditor Class

This class is the superclass of all UML diagram editors. It provides a drawing area and a toolbar with buttons, initiates all UDT tool operations and collaborates with other classes to execute them. Following is the description of some of the methods implemented in this class.

6.3.1 Method newWith:

This class message is used for opening a page with a specific diagram. It is sent to a subclass of the **UMLEditor** class to create a new instance of the class on a specific diagram. The argument `aDiagram` represents a **UMLDiagram** object, such as a Class and Object Diagram, a Sequence Diagram, etc., that this editor should display. It first creates a new instance of the receiver and then initializes the receiver with the nodes and edges of `aDiagram`.

```
newWith: aDiagram
   ^self new initializeWith: aDiagram
```

6.3.2 Method removeNode

This instance message is sent to an instance of a subclass of **UMLEditor** when the user selects "remove node" from a pop-up menu. It removes the currently selected node and all its edges. It first asks the **UMLDiagram** object to delete the current node and then asks the **UMLEditorView** object to remove the current selection. After that it updates the **DoUndoRedo** object. The next step is the deletion of all the edges that are attached to the current node. The last step is to tell its dependents that it has changed so that the diagram can redisplay itself.

removeNode

```
| edges node |
node := self currentNode.
self diagram removeNode: node.
self view unSelect: node.
self doUndoRedo undo: #reDoAddNode: redo: #unDoAddNode:
                arguments: (Array with: node).
edges := self diagram edgesFor: node.
edges do: [:edge |  self currentEdge: edge.
                    self removeEdge].
self changed: node kind with: node
```

6.3.3 Method executeOperationWith:

This instance message is sent to an instance of a subclass of **UMLEditor** by the **UMLEditorController** when the user clicks the right mouse button inside the drawing area of the current diagram. It first needs to determine which action is to be performed. The `message` value is usually an action, message, related to the type of action that should be performed. If `message` is undefined, nothing happens. The following are some valid values for the `message` instance variable: *#addNewNodeAt:*, *#addNewEdgeAt:*, *#dragShapeAt:* and *#addNewConnectionAt:*. The argument `aPoint` is the point in the window where the mouse button was clicked.

```
executeOperationWith: aPoint
   self message isNil ifFalse: [self  perform: self message
                                       with: aPoint]
```

6.3.4 Method dragShapeAt:

This instance message is sent to an instance of a subclass of **UMLEditor,** such as **UMLClassEditor,** by the **UMLEditorController** when the user clicks the right mouse button inside the drawing area of the current diagram. It first asks the view, an instance of **UMLEditorView**, for the selection at `aPoint`. The selection is not `nil` if a shape at `aPoint` was previously selected. If no selection is found, the method makes the shape at `aPoint` the current selection. It clears all selections if no shape is found. If a selection is found, the method asks the controller to drag the shape inside the selection to where the mouse is inside the window.

```
dragShapeAt: aPoint
   | handle umlShape |
   message := nil.
   handle := self view handlesAt: aPoint.
   handle isNil ifFalse: [^controller drag: handle shape
                                      start: aPoint].
   umlShape := self view visualComponentAt: aPoint.
   umlShape isNil ifTrue: [^self view unSelectExcept:
                                 OrderedCollection new].
```

```
  self view unSelectExcept: umlShape asHandle
                                  asOrderedCollection
```

6.3.5 Method addNewEdgeAt:

This instance message is sent to an instance of a subclass of **UMLEditor** by the instance method *executeOperationAt:* aPoint when the user executes the new edge command. It adds a new edge between two nodes. It first checks whether the from-side of the edge is on a valid node by asking the view to return the visual component at aPoint and checking whether the returned shape is a shape valid for this edge. Then it asks the controller object to draw a line from aPoint to the last point where the left mouse button was down and checks whether that point is on a shape valid for this edge. If the shape is valid, it updates the current edge with the appropriate origin, corner, from-side node's name and to-side node's name values and adds it to the **UMLDiagram** object edges collection. Finally, it updates the diagram by sending its receiver the message updateChanges:.

```
addNewEdgeAt: aPoint
  | shape1 endPoint shape2 |
  shape1 := self view visualComponentAt: aPoint.
  (self validView: shape1) ifFalse: [^self].
  endPoint := controller getLineFromUserAt: aPoint.
  shape2 := self view visualComponentAt: endPoint.
  (self validView: shape2) ifFalse: [^self].
  shape1 = shape2 ifTrue: [^self].
  (self isValidFrom: shape1 kind to: shape2 kind
        connection: self currentEdge kind)
        ifFalse: [^self].
  self currentEdge connectFrom: shape1 to: shape2.
  self diagram addEdge: self currentEdge.
  self updateChanges: self currentEdge
```

Method *updateChanges*: aShape sets the value of message to nil and sends its receiver the message changed: with: so its view updates itself accordingly.

```
updateChanges: aShape
  message := nil.
  saved := false.
  self controller cursor show.
  self changed: aShape kind with: aShape
```

6.3.6 Method changeNodeName:

This instance message is sent to an instance of a subclass of **UMLEditor** when the user selects "change name" from the <operate> menu. It changes the name of the currently selected node in the diagram. It first asks the user to enter a new name and then asks the current node to change its name to the new name and redisplay itself. Then it asks the view of this class to update all the edges that are connected to this node. After that it sends the receiver the message `changed: with:` so that all its dependents will be updated accordingly.

changeNodeName

```
| name nodeFullName edges node |
name := Dialog request: 'Enter new Name : '
                initialAnswer: self currentNode name.
(name isEmpty or: [name = self currentNode name]) ifTrue:
          [^self].
nodeFullName := self currentNode fullName.
self currentNode name: name.
node := self currentNode.
self view   invalidateRectangle: (node layout expandedBy:
     3 @ 3) repairNow: true.
self diagram edgesDo: [:edge | edge changeSideName:
     nodeFullName with: node fullName].
edges := self edgesFor: node.
self view updateEdges: edges using: node.
self changed: node kind with: node
```

6.4 UMLEditorView Class

This class defines the view (the object responsible for display) for all UDT diagrams editors. It provides a drawing area for drawing a selected type of a UML diagram. It scrolls its contents horizontally and vertically to allow the drawing of large diagrams and provides all operations that deal with the drawing of nodes and edges. It holds **UMLHandle** objects for marking a node or an edge as selected.

6.4.1 Method updateEdges: using:

This instance message is sent to an instance of **UMLEditorView** by its controller when dragging a **UMLNode** object from one position to another. It updates the origins and corners of all edges in the first argument, `edges`, to match the second argument, `aNode`, which is a **UMLNode** object. Message *updateUsing: for:* is sent to each edge in `edges`.

```
updateEdges: edges using: aNode
   edges do: [:edge | edge updateUsing: aNode for: self
                        controller]
```

Method *updateUsing: for:* is sent to an instance of a subclass of **UMLEdgeView** to check whether the edge is a line segment or a polyline. If the edge is segment (line with only two points), the instance message *updateLineUsing: for:* is sent to that edge. If the edge is a polyline with more than two points, the instance message *updatePolyLine:* `is sent` to that edge. This method will be explained in the next subsection.

```
updateUsing: aNode for: aController
   self isLine
      ifTrue: [self updateLineUsing: aNode for: aController]
      ifFalse: [self updatePolyLineUsing: aNode]
```

Method *updateLineUsing: for:* is sent to an instance of a subclass of **UMLEdgeView** to find the **UMLNode** object connected to this edge and adjusts the edge to the shortest distance between the centers of the two nodes.

```
updateLineUsing: aNode for: aController
    | node1 node2 |
    (self from sameAs: aNode fullName)
        ifTrue: [ node1 := aNode.
                  node2 := aController model diagram
                  findNodeNamed: self to]
        ifFalse:[ node1 := aController model diagram
                  findNodeNamed: self from.
                  node2 := aNode].
    self origin: (node1 nearestPointTo: node2 layout center).
    self corner: (node2 nearestPointTo: node1 layout center)
```

Method *updatePolyLine:* is sent to an instance of a subclass of **UMLEdgeView** to check if it is connected at the from-side, its origin is updated and if it is connected at the to-side, its corner is updated.

```
updatePolyLineUsing: aNode
    | point |
    (aNode fullName sameAs: self from) ifTrue: [
            point := self at: 2.
            self origin: (aNode nearestPointTo: point)]
        ifFalse: [(aNode fullName sameAs: self to) ifTrue: [
            point := self at: self size - 1.
            self corner: (aNode nearestPointTo: point)]]
```

6.4.2 Method updateEdge: using:

This instance message is sent to an instance of **UMLEditorView** by its controller when dragging a segment of **UMLEdge** object from one position to another inside the view drawing area. It updates the origin and corner of `anEdge` according to the positions of the nodes in the `nodes` argument. For each node in the `nodes` collection, it sends the instance message *updatePolyLine*: (the code is shown in the previous subsection) to `anEdge`.

```
updateEdge: anEdge using: nodes
   nodes do: [:node | anEdge updatePolyLineUsing: node]
```

6.5 UMLEditorController Class

This class is the controller for all UDT editor views. It uses polling to handle all the mouse events. It defines the <operate> menu and launches some operations when the left mouse button is pressed. Some of the operations associated with the left mouse button are: dropping a shape on the diagram, selecting or unselecting a shape and dragging a shape inside the window.

6.5.1 Method yellowButtonActivity

This instance message is sent to an instance of **UMLEditorController** when the user presses the <operate> ('yellow') mouse button. It opens a pop-up menu related to the shape in that position. It first asks the view to find the shape at the cursor point, by sending the message *visualComponentAt:* aPoint to the view, and then displays the right pop-up menu for that shape. If there is no shape at that point, the method displays a pop-up menu that enables the user to add new UML shapes specific to that diagram. If the shape is a node, it sets the current node to be this node and asks the model of the view to supply the pop-menu associated with this node and displays it. If the shape is an edge, it sets the current edge to be this edge and asks the view's model to supply the pop-up menu associated with this edge and displays it.

```
yellowButtonPressedEvent: event
   self yellowButtonActivity


yellowButtonActivity
   | umlObject |
   umlObject := self view visualComponentAt:
                              self sensor cursorPoint.
   umlObject isNil
        ifTrue: [self menuHolder value: self model viewMenu]
        ifFalse: [self updateChanges: umlObject]
   super yellowButtonActivity.
```

```
updateChanges: aShape
   aShape isNode
      ifTrue: [self model currentNode: aShape]
      ifFalse: [self model currentEdge: aShape].
   self menuHolder value:
      (self model perform: (aShape kind , 'Menu') asSymbol)
```

6.5.2 Method getLineFromUserAt:

This instance message is sent to an instance of a **UMLEditorController** by its view's model to draw a line following the cursor movement on the screen. It is used when adding an edge between two **UMLNode** objects. It asks the **Screen** to show a line on the screen starting at `aPoint` and returns the last point on the screen where the left mouse button was up.

```
getLineFromUserAt: aPoint
   | line |
   line := Array with: aPoint with: aPoint.
   Cursor crossHair showWhile: [
      [self sensor redButtonPressed] whileTrue: [
         Screen default displayShape: line at: self sensor
                        globalOriginforMilliseconds: 0.
         self viewHasCursor ifTrue: [
              line at: 2 put: self sensor cursorPoint]]].
   ^line at: 2
```

6.5.3 Method drag: start:

This instance message is sent to an instance of a **UMLEditorController** by its view's model when the user clicks the left mouse button on a selected shape. The first argument `umlShape` is the shape to be dragged inside the drawing area and the second argument `aPoint` is the starting point. The method sends the method *dragAt: for:* to `umlShape` and the method *changed* to its model so its view will change accordingly.

```
drag: umlShape start: aPoint
        umlShape dragAt: aPoint for: self.
        self model changed
```

The next two methods are required by the above definition if `umlShape` is a node. They are sent to an instance of a subclass of **UMLNodeView.** Method *dragAt: for:* asks the model for all the edges that are connected to this node and then removes them. After that it sends the message `dragWith: at:` to its receiver which drags the selected node to follow the cursor position. This process stops when the left mouse button is released. Finally all the updated edges of this node are added back to the diagram.

```
dragAt: aPoint for: aController
    | oldPoint oldLayout edges |
    oldPoint := aPoint.
    edges := aController model edgesFor: self.
    edges do: [:edge | aController model diagram
                                            removeEdge: edge].
    aController view invalidate.
```

```
oldLayout := self layout expandedBy: 3 @ 3.
Cursor hand showWhile: [[aController sensor redButtonPressed]
   whileTrue: [aController viewHasCursor ifTrue: [
      oldPoint := self dragWith: edges at: oldPoint
                        inside: oldLayout for: aController]]].
edges do: [:edge | aController model diagram addEdge: edge]
```

Method *dragWith: at:* drags the receiver from its original position to follow the cursor position and draws lines on the screen for each edge that is connected to this node.

```
dragWith: edges at: aPoint inside: aRectangle for: aController
   | newPoint oldLayout |
   oldLayout := aRectangle.
   newPoint := aController sensor cursorPoint.
   newPoint = aPoint ifFalse: [
      self moveBy: newPoint - aPoint.
      aController view updateEdges: edges using: self.
      edges do: [:edge | Screen default
                  displayShape: edge edge
                  at: aController sensor globalOrigin
                  forMilliseconds: 10].
      aController view invalidateRectangle: oldLayout
                        repairNow: true.
      oldLayout := oldLayout moveBy: newPoint - aPoint.
```

```
    aController view invalidateRectangle: oldLayout
                         repairNow: true].
    ^newPoint
```

The next three methods are required by the definition in 6.5.3 if `umlShape` is an edge. They are sent to an instance of a subclass of **UMLEdgeView.** Method *dragAt: for:* asks the receiver (an edge) for the nearest point on its drawing area to `aPoint`. If the nearest point was the first or the last point on the edge then no dragging is allowed. If the nearest point is an inner point then if it is the second or the one before the last then *dragEndPointAt: start: for:* is sent. If the point was between the second and the one before the last then *dragMiddlePointAt: start: for:* is sent.

```
dragAt: aPoint for: aController
  | point index1 index2 |
  point := self nearestPointTo: aPoint.
  (point = self origin or: [point = self corner]) ifTrue: [^self].
  index1 := self endsOf: point.
  index1 isNil ifTrue: [^self].
  (self edge indexOf: point) isZero ifTrue: [
     self add: point after: (self at: index1)].
  index2 := index1 + 1.
  (index2 = 2 or: [index2 = (self size - 1)])
  ifTrue: [self  dragEndPointAt: index2   start: aPoint
                 for: aController]
```

```
    ifFalse: [self dragMiddelPointAt: index2 start: aPoint
                        for: aController]
```

Method *dragMiddlePointAt: start: for:* is sent to drag the receiver from the middle.

```
dragMiddelPointAt: pos start: aPoint for: aController
    | newPoint oldPoint oldLayout |
    oldPoint := aPoint.
    Cursor hand showWhile: [[aController sensor
redButtonPressed]
            whileTrue: [aController viewHasCursor ifTrue: [
                newPoint := aController sensor cursorPoint.
                newPoint = oldPoint ifFalse: [
                    oldLayout := self layout expandedBy: 3 @ 3.
                    self at: pos put: (self at: pos)
                            + (newPoint - oldPoint).
                    oldPoint := newPoint.
                    aController view invalidateRectangle:
                            oldLayout repairNow: true.
                    aController view invalidateRectangle:
                        self layout repairNow: true]]]]
```

Method *dragEndPointAt: start: for:* is used to drag the receiver (an edge) from one of its ends.

```
dragEndPointAt: pos start: aPoint for: aController
  | newPoint oldPoint oldLayout col |
  oldPoint := aPoint.
  col := OrderedCollection new.
  pos = 2 ifTrue: [col add: (aController model diagram
                                  findNodeNamed: self from)].
  pos = (self edge size - 1) ifTrue: [col add:
    (aController model diagram findNodeNamed: self to)].
  Cursor hand showWhile: [[aController sensor redButtonPressed]
    whileTrue: [aController viewHasCursor ifTrue: [
      newPoint := aController sensor cursorPoint.
        newPoint = oldPoint ifFalse: [
          self moveAt: pos from: oldPoint to: newPoint
               with: col for: aController.
      oldPoint := newPoint]]]]
```

6.6 DiagramGeneratorBrowser Class

This class provides a browser that is used for reverse engineering. Its layout is almost identical to the Smalltalk Browser but adds a list for selecting the classes for which a class diagram should be drawn (see Figure 4-9). It invokes all the operations for selecting the desired classes either by dragging the whole category or just a class to the selection list. In the following subsections, the implementation of some methods will be shown.

6.6.1 Method createDiagram

This instance method is used to create a partial class diagram for the selected classes. It is sent to an instance of **DiagramGeneratorBrowser** when the user clicks the `Create` button in Figure 4-9. It first creates a new instance of **UMLRelationsBuilder** on the selected classes and then sends that instance the message *asUMLDiagram* which converts that instance to an instance of **UMLDiagram**. Finally, it assigns this instance of **UMLDiagram** to the callers diagram for display on the caller's window.

```
createDiagram
   | associations |
   associations := UMLRelationsBuilder on: self
                                        getSelectedClasses.
   self caller diagram: associations asUMLDiagram.
   self cancelDiagram
```

6.7 UMLRealtionsBuilder Class

This class provides all the functionality for dealing with reverse engineering. It implements all the necessary functions for finding the associations and generalizations for a collection of classes and provides information necessary for drawing the diagrams. It contains three ordered collections, one for storing the selected classes, another for storing all the associations and the third one for storing all the generalizations. In the following subsections, the implementation of some methods will be shown.

6.7.1 Method on:

This class message is used to create an instance of **UMLRelationsBuilder** on classes selected by the user. It is sent to **UMLRelationsBuilder** by **DiagramGeneratorBrowser** to return an instance initialized to the classes in the argument `aCollection`.

```
on: aCollection
     ^self new mainClasses: aCollection
```

6.7.2 Method asUMLDiagram

This instance method is used to convert the receiver to an instance of **UMLDiagram**. It is sent to an instance of **UMLRelationsBuilder** by **DiagramGeneratorBrowser** to return an instance of **UMLDiagram**. This instance then can be displayed in a Class Diagram editor's window.

```
asUMLDiagram
   | umlDiagram |
   umlDiagram := UMLDiagram new.
   self getNodes: umlDiagram.
   self getEdges: umlDiagram.
   ^umlDiagram
```

6.7.3 Method classReferences:

This instance method is used to return a collection of all classes that reference the class in the argument. It is sent to an instance of **UMLRelationsBuilder** with the name of the class as an argument. It first gets the 'user' classes from the class **SmalltalkClasses** and then searches the selectors of all these classes for aSymbol.

```
classReferences: aSymbol
   | calls userClasses |
   calls := Set new.
   userClasses := SmalltalkClasses userClasses asSet.
   userClasses := userClasses collect: [:class |
                (Smalltalk associationAt: class) value].
   userClasses do: [:class | (((class whichSelectorsReferTo:
         (Smalltalk associationAt: aSymbol)) asSet) addAll:
         (class whichSelectorsReferTo: aSymbol); yourself)
              isEmpty ifFalse: [
                 calls add: class instanceBehavior name]].
   ^calls asOrderedCollection
```

Chapter 7

Conclusion

7.1 Summary

The purpose of this thesis was to develop a tool that could be used to support the drawing of several types of UML diagrams, their integration into a larger software development environment (the Binder) and reengineering of existing code. This was achieved through the following:

- Studying object-oriented methodologies. Several object-oriented methodologies were studied to accomplish two things: understanding the need for a methodology in object-oriented software development and understanding the differences and similarities between object-oriented methodologies.
- Studying the Unified Modeling Language (UML). Since the aim was to allow for the drawing of selected types of UML diagrams, a comprehensive study of UML was done.
- Designing and implementing UDT.
- Studying the Binder program. A full study and understanding of the Binder allowed the integration of the tool into the Binder to be accomplished easily.

The result of these activities is UDT – UML Drawing Tool. UDT enables the drawing of the following UML diagrams: Class and Object Diagrams, Use Case Diagram, Sequence Diagram, State Diagram and Package Diagram. UDT implements these diagrams as page styles for the Binder program. The drawing is accomplished by providing an editor and a toolbar for each page style. The toolbar provides general buttons for creating, storing, loading and printing of selected UML diagrams, undoing and redoing of previous actions and adding a note and note connection to the selected UML diagram. It also provides buttons for adding edges and nodes for each supported UML diagram. The editor provides a drawing area for the selected diagram. It enables the selection, deleting and dragging of any shape in the diagram. For each shape, the editor provides a specific pop-up menu for that shape.

UDT also provides the following forward and reverse engineering mechanisms: It allows for the defining of classes and their variables and the creation of empty methods from diagrams. It also enables partial creation of UML class diagrams from Smalltalk code including inheritance and association.

7.2 Future Work

Based on the work done in this thesis, many topics can be suggested for future work. Some of these topics are:

- Extending UDT to include the remaining UML diagrams. Other UML diagrams can be implemented as page-styles to the Binder. This can be accomplished by subclassing **UMLEditor** class to provide an editor for the new diagram and subclassing **UMLEditorController** and **UMLEditorView** classes if necessary. All the nodes in the diagram should be represented by views that are subclasses of **UMLNodeView** and all edges should be represented by views that are subclasses of **UMLEdgeView**. Also each diagram's editor should have a client class that is a subclass of **UMLEditorClient** (to be used for the Binder).

- Automatic creation of other diagrams from code. Parts of other UML diagrams can be generated from a Smalltalk code. An example is generating Sequence Diagrams from use cases.
- Applying changes to the Smalltalk library. In the current version of UDT some changes to a class in a UDT diagram, such as removing or renaming a class or changing its state or behavior, are not applied to the library. In future versions, these changes could be made to allow for a stronger link between the diagram and the code.
- Design control capturing the evolution of diagrams could be implemented as a part of the Binder and linked to source code versions.
- Dynamic diagrams such as sequence and state diagrams could be used to support and partially automate the creation of texts.
- Additional reverse engineering tasks on the basis of more sophisticated code analysis or runtime tracing.

Bibliography

[Booch 1993]	Booch, G., Object-Oriented Analysis and Design with Applications, 2nd edition, Benjamin Cummings, Redwood City, California, 1993.
[Booch 1994]	Booch, G., Object-Oriented Analysis and Design with Applications, Benjamin Cummings, Redwood City, California, 1994.
[Booch 1995]	Booch G. and Rumbaugh J., Unified Method for Object-Oriented Development, Documentation Set Version 0.8, October 1995.
[Booch 1996]	Booch G., Jacobson I. and Rumbaugh J., The Unified Modeling Language for Object-Oriented Development, Documentation Set Version 0.91, September 1996.
[Booch 1997]	Booch G., Jacobson I. and Rumbaugh J et. al., The Unified Modeling Language for Object-Oriented Development Version 1.0, UML Notation Guide, UML Summary, UML Semantics, Rational Software Corporation, January 1997 and the UML 1.1 update of Sept. 1997.
[Booch 1999]	Booch, G., Rumbaugh, J. and Jacobson, I., The Unified Modeling Language User Guide, Addison Wesley Longman, Inc, 1999.
[Cincom 2000]	Cincom's VisualWorks – Smalltalk Software,2000. Available via: http://www.cincom.com/visualworks/
[Coad 1991a]	Coad, P. and Yourdon, E., Object Oriented Analysis (2nd Edition), Yourdon Press, Englewood Cliffs, New Jersey, 1991.
[Coad 1991b]	Coad, P. and Yourdon, E., Object Oriented Design, Yourdon Press, Englewood Cliffs, New Jersey, 1991.
[Coleman 1994]	Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremaes, P., Object-Oriented Development The Fusion Method, Prentice Hall, Englewood, New Jersey 07632, 1994.
[Demmer 1997]	Demmer, C., UML 1.1 vs. MWOOD, using material from Booch, G., Rumbaugh, J. and Jacobson, I., 1997. Available via: http://stud2.tuwien.ac.at/~e8726711//ummw1.html
[Eriksson 1998]	Eriksson, H. and Penker, M., UML Toolkit, John Wiley & Sons, ISBN: 0471191612, 1998.

[Firesmith 1993]	Firesmith D.G., Object-oriented Requirements Analysis and Logical Design - ASE Approach, John Wiley & Sons NY, 1993.
[Firesmith 1998]	Firesmith, D., Henderson-Sellers, B. and Graham, I., OPEN Modeling Language (OML) Reference Manual, Cambridge University Press, 1998.
[Gottesdiener 1998]	Gottesdiener, E., OO-Methodologies: Process & Product Patterns, Component Strategies, Vol. 1, No 5, 1998.
[Howard 1995]	Howard, T., The Smalltalk Developer's Guide to VisualWorks, SIGS Publications, Inc., New York, 1995.
[IEEE 1998]	IEEE "Recommended Practice for Architectural Description," Draft Std. P1471, IEEE, 1998.
[Jacobson 1992]	Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G., Object-Oriented Software Engineering, Addison-Wesley, 1992.
[Jacobson 2000]	Jacobson, I., Rumbaugh, J. and Booch, G., The Unified Software Development Process, Addison Wesley Longman, Inc., 1999.
[Martin 1993]	Martin, J. and Odell, J., Object Oriented Analysis and Design, Prentice Hall, Englewood, New Jersey, 1993.
[McKean 1995]	McKean, A., and Wirfs-Brock, R., Responsibilities-Driven Design "Tutorial Notes, Tutorial 42". ParcPlace-Digitalk, Inc., 1995.
[Networld 1999]	Networld Solutions - All rights reserved. QualIT and POC are registered service marks of Paladin Enterprises, Inc., 1999. Available via: http://www.networld-solutions.com/qualit/default.htm
[Oestereich 1999]	Oestereich, B., Developing Software with UML, Object-oriented analysis and design in practice, Addison Wesley Longman Ltd, 1999.
[Rational 2000]	Rational Software, The Unified Modeling Language (UML), 2000. Available via: http://www.rational.com/
[Richter 1997]	Richter, C., Exploring the Unified Modeling Language (UML) by Example, Object Engineering, Inc, 1997.

[Rose 1996]	The Rational Rose, Rational Software, 1996. Available via: http://www.rational.com/
[Rumbaugh 1991]	Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., Object-Oriented Modeling And Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
[Shlaer 1992]	Shlaer, S. and Mellor, S.J, Object Lifecycles: Modeling the world in states, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.
[Technology 1997]	Object Oriented Technology, 1997. Available via: http://disc.cba.uh.edu/~rhirsch/spring97/lam1/hope.htm
[Together 1998]	Together J/C++, Object International, Peter Coad's Company, 1998. Available via: http://www.oi.com/
[Tomek 2000]	Tomek, I., An Electronic Binder for an Object-Oriented Analysis and Design course, SIGCSE, Symposium, Austin, 2000.
[Waldén 1995]	Waldén, K. and Nerson, J., Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems, Prentice Hall, 1995.
[Wirfs-Brock 1990]	Wirfs-Brock, R., Wilkerson, B. and Wiener, L., Designing Object-Oriented Software, Prentice Hall, Englewood, New Jersey 07632, 1990.