University of Sussex



A Learner Centred CASE Tool for Software Engineering

Lamya Mohammed Aljasmi

Submitted for the degree of Doctor of Philosophy

April 2000

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other University for a degree.

Signature:

University of Sussex

Lamya Aljasmi Doctor of Philosophy

A Learner Centred CASE Tool for Software Engineering

Summary

LCCASE is a learner centered computer-aided software engineering tool for assisting in the learning of object-oriented software engineering and is, in addition, a powerful and effective software development environment.

LCCASE is intended to support the teaching of courses in software engineering and aims to improve the pedagogical efficiency of the process through its learner-friendly approach.

The main objectives of the LCCASE tool are: supporting the development of large software systems, improving the learning of software engineering, enhancing the productivity of learners, and making software development a satisfying task.

The LCCASE tool achieves its objectives by providing intelligent guidance to its users, suggesting the models that should be developed throughout the software life cycle, checking the consistency and the completeness of those models, and by reusing information.

The LCCASE tool allows the developer to link concepts throughout the software development life cycle. It automates the *Next Generation of the Fusion Method*, understands the semantics of the models of that method, realizes the connection between the information in the several models, and provides this information to the user, as required.

In addition, the LCCASE tool provides graphical assistance, allowing the developer to construct diagrams easily, navigational assistance by providing the developer with information about all the components of a project and their relationships, and maintaining a data dictionary for a project. The design of LCCASE itself is based on object-oriented software engineering principles and is capable of being developed to support code generation and reverse engineering.

The LCCASE tool is powerful and introduces new techniques for overcoming the challenges encountered while developing large software systems.

Acknowledgement

Throughout this work and this thesis, I was fortunate to have had many people by my side.

I would like to thank Professor Richard Grimsdale for providing me with his precious guidance, sharing his invaluable experience, and his ultimate patience.

Thanks are extended to my mother and late father. I am deeply indebted to them.

I am especially grateful to my husband, without his continued support, encouragement, and love, this work would not have been possible.

To our children, Fatema and Mohammed, they have made our life a very pleasant one.

I would like to thank the University of Bahrain, for providing me with the scholarship and giving me the opportunity to carry out this work.

I appreciate the efforts of the University of Sussex Crèche and Nursery staff, who took very good care of our children.

Table of Contents

1 INTRODUCTION 1

1.1 DEFINITION OF THE FIELD 2
1.1.1 Software Engineering and Object-Oriented Software Engineering 2
1.1.2 Computer Aided Software Engineering (CASE) Tools 3
1.1.3 Software Engineering Education 4
1.2 AN OVERVIEW OF PREVIOUS WORK 5
1.2.1 The ExSpect Integrated Tool 5
1.2.2 ROME 6
1.2.3 UML CASE TOOL 6
1.3 DESCRIPTION OF THE WORK 6
1.3.1 Inspirations 6
1.3.2 The LCCASE Tool 8
1.3.3 The Design of the LCCASE Tool 10
1.3.4 The Software Engineering Process 11
1.4 AN OVERVIEW OF THE CHAPTERS OF THE THESIS 12

2 BACKGROUND 14

2.1 SOFTWARE ENGINEERING 14
2.2 OBJECT-ORIENTATION 15
2.2.1 Abstraction 16
2.2.2 Decomposition 16
2.2.3 Modelling 16
2.2.4 Encaps 17
2.2.4 ulation 17
2.3 OBJECT-ORIENTED SOFTWARE DEVELOPMENT 17
2.4 THE STYLES OF SOFTWARE DEVELOPMENT LIFE CYCLES 18
2.4.1 Waterfall Life Cycle 18
2.4.2 Incremental and Iterative Development Life Cycle 19
2.4.3 Evolutionary Life Cycle 20
2.5 OBJECT-ORIENTED METHODOLOGY 21
2.5.1 Advantages of the Object-Oriented Methods 22
2.5.2 The Fusion Method 23
2.5.3 Evolutionary Fusion 32
2.5.4 The Next Generation of the Fusion Method 33
2.5.5 Object Modelling Technique (OMT) 38
2.5.6 The Objectory Method 42
2.5.7 The Booch Method 45
2.5.8 The Unified Modelling Language (UML) 51
2.5.9 OO Methodologies can be Adopted as Needed to Meet the Requirements of a Project 54
2.6 COMPUTER-AIDED SOFTWARE ENGINEERING (CASE) TOOLS 55
2.6.1 The Benefits of CASE 55
2.6.2 Typ 56
2.6.2 es Of CASE Tools 56
2.6.3 The CASE Evolution 56
2.6.4 Philosophies for Supporting the CASE Tools Methodology 59
2.7 SOME EXAMPLES OF OBJECT-ORIENTED CASE TOOLS 60
2.7.1 The ExSpect Integrated Tool 60
2.7.2 ROME 63
2.7.3 UML CASE TOOL 64

2.8 TEACHING SOFTWARE ENGINEERING 65
2.8.1 *Goals of a Course on Software Engineering* 65
2.8.2 *To* 67
2.8.2 *ols* 67
2.8.3 *Difficulties Involved in Teaching Software Engineering Courses* 67
2.8.4 *S* 68
2.8.4 *olutions* 68
2.9 CHAPTER SUMMARY 68

3 LEARNER CENTRED COMPUTER AIDED SOFTWARE ENGINEERING (LCCASE) TOOL 70

3.1 THE NEED FOR THE LCCASE TOOL 71
3.2 THE PHILOSOPHY OF THE LCCASE TOOL 72
3.2.1 *Guided Philosophy* 72
3.2.2 *Interactivity* 73
3.2.3 *Step by* 74
3.2.3 *Step Construction* 74
3.2.4 *Intermediate Construction* 74
3.2.5 *Navigation* 74
3.2.6 *Flexibi* 75
3.2.6 *lity and Individualisation* 75
3.2.7 *Ease of Use* 75
3.2.8 *The Nex* 76
3.2.8 *t Generation of the Fusion Method* 76
3.3 THE LCCASE TOOL MAIN FEATURES 76
3.3.1 *Intelligent Guidance* 76
3.3.2 *Navigational Assistance* 78
3.3.3 *Graphical Assistance* 79
3.3.4 *Code Generation Facility* 79
3.3.5 *The Data Dictionary* 80
3.3.6 *Report Generation Facility* 80
3.3.7 *Support the Maintenance Phase* 80
3.4 THE BENEFITS OF THE LCCASE TOOL 81
3.4.1 *The Benefits of the LCCASE Tool to the Professional Developer* 81
3.4.2 *The Benefits of LCCASE Tool for Learners of a Course on SE* 81
3.5 CHAPTER SUMMARY 82

4 THE DESIGN OF THE LCCASE TOOL 84

4.1 THE ARCHITECTURE OF THE LCCASE TOOL 84
4.1.1 *Model S* 88
4.1.1 *ubsystem* 88
4.1.2 *D* 91
4.1.2 *irectory Subsystem* 91
4.1.3 *Development Life Cycle Subsystem* 96
4.1.4 *Data Dic* 99
4.1.4 *tionary Subsystem* 99
4.2 THE CONT 106
4.2 ROL STRATEGY BETWEEN THE SUBSYSTEMS OF THE LCCASE TOOL 106
4.3 THE ENVIR 107
4.3 ONMENT OF THE LCCASE TOOL 107
4.3.1 *The U* 109
4.3.1 *se-* 109
4.3.1 *Case Model* 109
4.3.2 *Class Diagram* 110
4.3.3 *Scenario* 112
4.3.4 *Contract* 114

4.3.5 Collaboration Diagram *116*
4.3.6 Desi *118*
4.3.6 gn Class Diagram *118*
4.3.7 Code Generation *119*
4.3.8 Information Linkage in the LCCASE Tool *120*
4.4 THE CHAPTER SUMMARY 123

5 AN EXAMPLE TO SHOW 125

5.1 OVERVIEW 125
5.1.1 The Library System Requirement Statement *126*
5.1.2 The Use-Case Model *126*
5.1.3 The Class Diagram *139*
5.1.4 The Scenario *169*
5.1.5 The Contract *186*
5.1.6 The User Use-Case Model Revisited *190*
5.1.7 The Window Menu *197*
5.1.8 The File Menu *200*
5.1.9 The Data Dictionary *202*
5.2 CHAPTER SUMMARY 206

6 IMPLEMENTATION 207

6.1 REQUIREMENTS 210
6.2 THE USE-CASE MODELS 210
6.2.1 Model Use-Case Model *211*
6.2.2 File Use-Case Model *213*
6.2.3 Window Use-Case Model *215*
6.2.4 Development Life Cycle Use-Case Model *216*
6.2.5 Directory Use-Case Model *217*
6.2.6 Data Dictionary Use-Case Model *219*
6.2.7 Code Generation Use-Case Model *220*
6.3 CLASS DIAGRAMS 221
6.3.1 AllModels Class Diagram *222*
6.3.2 Model Class Diagram *225*
Use-Case Model Class Diagram *226*
6.3.4 Use-Case Model Objects Class Diagram *232*
6.3.5 The Relationships of the ClassDiagramCanvas Class Diagram *233*
6.3.6 ClassClass Class Diagram *235*
6.3.7 Relationship Class Diagram *237*
6.3.8 Scenario Class Diagram *240*
6.3.9 Directory Class Diagram *243*
6.3.10 AllDirectories Class Diagram *244*
6.3.11 Window Organiser Class Diagram *247*
6.3.12 Development Life Cycle Class Diagram *248*
6.3.13 Database Class Diagram *250*
6.4 SCENARIOS 251
6.4.1 Model Scenarios Group *252*
6.4.2 File Scenarios Group *252*
6.4.3 Window Scenarios Group *252*
6.4.4 Development Life Cycle Scenarios Group *252*
6.4.5 Directory Scenarios Group *253*
6.4.6 Create UseCase Scenario *253*
6.5 CONTRACTS 257
6.5.1 The Contract, addModel *257*

6.6 COLLABORATION DIAGRAMS 259
6.7 DESIGN CLASS DIAGRAMS 260
6.8 CODING 260
6.8.1 Flickering When Dragging an Object *261*
6.8.2 Re *263*
6.8.2 dr *263*
6.8.2 aw the Inherited Relations *263*
6.8.3 Sa *265*
6.8.3 vi *265*
6.8.3 ng Images on Disk *265*
6.9 THE 267
6.9 IMPLEMENTATION OF THE CLASSCLASS CLASS 267
6.10 THE 280
6.10 IMPLEMENTATION OF LINKING THE INFORMATION IN THE MODELS 280
6.10.1 Implementing the link between the Use-Cases and Scenarios *281*
6.10.2 I *288*
6.10.2 mplementing the Link between the actors in a Use-Case Model and the actors in a Scenario *288*
6.10.3 Im *296*
6.10.3 plementing the Link between the Parameters of a System Operation and the Classes *296*
6.11 SU 298
6.11 MMARY 298

7 CONCLUSION 300

7.1 THE LCCASE TOOL 300
7.1.1 The Characteristics of the LCCASE Tool *302*
7.1.2 The Design of the LCCASE Tool *305*
7.2 THE IMPACT OF THE LCCASE TOOL ON THE DESIGN OF NEW CASE TOOLS 307
7.3 CONCLUSIONS REACHED FROM DEVELOPING THE LCCASE TOOL 307
7.4 FUTURE WORK 308

REFERENCES 310

APPENDICES 315

1 Introduction

This thesis describes the design, implementation, and use of a learner centred computer aided software engineering (LCCASE) tool to assist in developing successful object-oriented software.

The objective of my work is to design and develop a powerful and effective CASE tool for developing successful object-oriented software as well as assisting the learning of software engineering.

The LCCASE tool provides a fully integrated computer aided software engineering environment. It supports the whole development life cycle of a system by automating the Next Generation of the Fusion Method [Coleman 1997a, Coleman 1997b].

The LCCASE tool applies the semantics and rules of the models in the Next Generation of the Fusion Method. It has the ability to logically link the information in the developed project and provide a full and intelligent guidance to the user through the software development process.

The unique characteristics of the LCCASE tool of intelligent guidance and logical assistance in an integrated environment provide means to support the needs of the software developers and overcome the difficulties that arise in the development and learning processes of software engineering.

This chapter starts with an introduction to object-oriented software engineering, CASE tools, and software engineering education. Next, the chapter discusses several previously developed CASE tools such as the ExSpect Integrated Tool, ROME, and the UML CASE Tool. The chapter then describes the inspirations of my work and the objectives, design

characteristics, and implementation of the LCCASE tool. Finally, an overview of the chapters in the thesis is presented.

1.1 Definition of the field

1.1.1 Software Engineering and Object-Oriented Software Engineering

The software development process consists of three phases: analysis, design, and implementation. The primary purpose of the software life cycle is to provide a controlled process for managing the development of software.

Object-oriented software development is based on abstractions that exist in the real world. In object-oriented software development, the object-orientation concept is applied to all phases of the development process. The aim of the analysis phase is to understand the requirements of the problem domain and recognise them as a collection of classes linked by relationships. The design phase provides a comprehensive description of the software solution for the system and clarifies the concept of objects and the communication between them. During the implementation phase, programs are written with an object-oriented language. Object-oriented programming views the programs as a co-operative collection of objects, which are instances of classes that combine both data and behaviour.

There are many different object-oriented methodologies available for software development such as Fusion [Coleman 1994], OMT [Rumbaugh 1991], Booch [Booch 1994], and UML [Booch 1998]. Each methodology has its own terminology, products, processes, and techniques for implementation.

The methodology should provide guidelines through all phases of the software life cycle to control and reduce the software development complexity and produce high quality software.

1.1.2 Computer Aided Software Engineering (CASE) Tools

The development of large software is complex, expensive, labour intensive, and time consuming. Consequently, most of the large software projects are delivered over budget, behind schedule, and rarely meet the original objectives.

In order to overcome the problems associated with the software development process, object-oriented software development techniques and methodologies combined with CASE tools are used to support and automate the development of the software life cycle phases.

CASE tools increase the software development effectiveness. They improve the software development productivity, increase the quality of the developed system, and reduce the development and maintenance cost.

A typical CASE tool should include a graphical editor which supports developing the diagrams, utilise method rules and semantics to provide logical assistance, guide the user through the software development, ensure consistency and traceability between the method phases, store the information in a common repository, automatically generate documentation, and support team work.

Unfortunately, the current situation of the CASE tools is unsatisfactory. Many CASE tools are drawing tools with few consistency checks and little knowledge of the method they implement, slow, and erroneous. In addition, many CASE tools perform a single

function such as drawing diagrams or documentation. However, few CASE tools perform multiple functions, share data, and automate a software development methodology.

1.1.3 Software Engineering Education

Teaching software engineering is currently being re-evaluated and analysed because of the rapid increase in the software development complexity and the availability of new object-oriented methodologies and techniques [Tewari 1995, Abran 1996, Abu Al-Maati 1997, Bezivin 1992, Concepcion 1998, Dodani 1992, Maibaum 1997].

In a course on software engineering, principles and concepts of building large software systems are taught.

The objectives for a course on software engineering include learning and experiencing all the phases of the software life cycle, understanding and exercising formal methodologies and models, experiencing real world projects, working and co-operating with a team, and preparing the students with knowledge and experience that will accommodate their transition from academia to industry.

There are several difficulties in presenting a course on software engineering, as the students are not able to experience the complete software engineering development process. In most cases, they learn and practice analysis and design but there is insufficient time to complete the implementation of their design, which does not allow them to test and verify the effectiveness and quality of their systems. The production of real software at the end of the learning period, motivates the students and increases their satisfaction of the course. In addition, it is difficult to practice the iterative approach due to the poor tools available and time limitation. Furthermore, the software development process itself

is complex and the analysis and design is a synthesis process with no unique answers. Moreover, most object-oriented methods do not have a clear process that gives detailed descriptions of the activities and their control in the method, which makes them too complex to apply to a software project in a course on software engineering.

To aid learning of software engineering and overcome the above problems a well-defined object-oriented software engineering method combined with a powerful and easy to use CASE tool should be used.

1.2 An Overview of Previous Work

Although most available CASE tools provide the same degree of support to both the new and expert developer, there are several CASE tools that have important features to be discussed such as the ExSpect Integrated Tool [Broek 1992], ROME [Parsons 1994], and the UML CASE Tool [Booch 1998].

1.2.1 The ExSpect Integrated Tool

The ExSpect Integrated Tool consists of a mathematical model with an associated language, an object-oriented data model, and a CASE tool.

The ExSpect CASE tool is a shell comprising a graphical editor, a type checker for checking the consistency, an interpreter for executing specification, and some supporting tools. The tool set is implemented using the C language and runs under UNIX on SUN machines.

The ExSpect Integrated Tool has been designed for the development of systems in which the time and the physical location of its components are very important, as in

distributed information systems. It provides a partial support for all the phases of the software life cycle and can be considered a good example of an early attempt to implement the integrated CASE tool philosophy.

1.2.2 ROME

ROME is an object-oriented CASE tool. It was developed to aid the learning of object-oriented software engineering.

ROME assists users in drawing class and object diagrams. It supports code generation and reverse engineering and provides a partial support to the analysis phase but it does not provide any support for the design phase, which means that students will not be able to experience all the phases of the software life cycle.

1.2.3 UML CASE TOOL

UML CASE TOOL is an object-oriented CASE tool developed by Rational Rose to support the development of object-oriented software using the UML method. The UML CASE tool has many important features such as graphical assistance, report generation, code generation, and reverse engineering which are discussed in detail in chapter 2.

1.3 Description of the work

1.3.1 Inspirations

It is now evident that software engineering techniques and methodologies are very important to reduce software development complexity and produce high quality software. However, existing CASE tools lack the intelligent support for software developers, have

partial knowledge of the method they automate, provide the same support to the experienced developer and learner of software engineering, and do not necessarily speed up the development process.

This inspired me to design and develop a CASE tool that provides intelligent support, understands the semantics of the method it implements, and guides the developer through the development process to produce successful software in a short period of time.

Furthermore, while the discipline of software engineering is at the end of its third decade, teaching methods of software engineering are still being developed.

The students of software engineering must learn that the software development life cycle is not just the development of class diagrams in the analysis phase and the programming in the implementation phase. The students need to learn and experience all phases of the software life cycle.

If the students develop real life projects in a course on software engineering, they will have the chance to learn by doing and will be able to practice their theoretical studies and acquire an important experience.

It has been claimed that the iterative style for the software development life cycle has many advantages over other life cycle styles [Cotton 1996], which are discussed in detail in chapter 2.

Most real life projects are too complex to be used as a software project in a course on software engineering. Moreover, developing successful software is a creative process and does not have a unique and exact solution, but the main aim of the students is to satisfy the project requirements, in real life the user needs, which increase the complexity and

the difficulty of the course. In addition, the students have to develop the project or at least a prototype in relatively short time.

Most CASE tools that are currently used to assist students suffer from the classic deficiencies in CASE tools that are discussed above. Moreover, these CASE tools are designed for experienced software developers and thus do not address the special needs of the students.

All the above reasons beside increasing the pedagogical efficiency of a course on software engineering, defeating the time constraints, improving the quality of the students' projects, and overcoming the difficulties in teaching software engineering that are discussed above inspired me to develop a CASE tool that assists the experienced software developer as well as students of a software engineering course in the development of a software project.

1.3.2 The LCCASE Tool

LCCASE is an integrated CASE tool which automates all the phases of the software life cycle in the Next Generation of the Fusion method, provides intelligent guidance to speed up the software development process, understands the semantics of the Fusion method, realises the connection between the models, is simple enough to be used by students, and supports the iterative style of the software development life cycle.

The main objectives of the LCCASE tool are assisting the development of successful object-oriented software, improving the productivity of the software developer, improving the learning of software engineering, involving the student, and making software development practice a more satisfying task.

The LCCASE tool has many features that facilitate and aid the software developer in completing his software project. These features include providing intelligent guidance to the software developer. The LCCASE tool can realise the connection between the models and provide this information to the developer as required. It has the ability to make many consistency checks between the information in the models and suggest the models that should be created by the developer according to information obtained from the models that have already been developed.

In addition, it supports multiple views of the models, advises the developer to reuse items or diagrams and has the ability to reuse items and diagrams automatically. It provides a facility to track the requirements specified in the Use-Case Model of the Next Generation of the Fusion method and maps these requirements to other models. Moreover, the LCCASE tool has the ability to analyse the models and inform the user about insufficient information in the models.

Furthermore, the LCCASE tool provides graphical assistance so that the software developer can draw the diagrams of the Next Generation of the Fusion method, logical assistance through understanding the semantics of the models and performing consistency checks, and navigational assistance by allowing users to navigate through the models and the class and use-case hierarchies.

The LCCASE tool creates and maintains a data dictionary for the project and generates reports of classes and models as required. In addition, the LCCASE tool could be developed in the future to have code generation.

1.3.3 The Design of the LCCASE Tool

A structured approach has been used in designing and developing the LCCASE tool to meet the objectives, which are discussed in the previous section.

The architecture of the LCCASE tool consists of co-operating subsystems: Model, Directory, Development Life Cycle, Data Dictionary, Database, and User-Interface.

Each subsystem has specific and clearly defined responsibilities. The subsystems are collection of classes, relationships, operations, and constraints. They have well-defined relationships with each other.

The main objectives of the design of the LCCASE tool are maintaining links between the subsystems of the LCCASE tool to facilitate the functionality of each subsystem without increasing the dependency between them nor affecting their integrity, the ability to adapt the functionality of the LCCASE tool to the needs of the students, and the availability of guidance and support to students.

The Model is the largest subsystem in the LCCASE tool and is responsible for developing all the models in all the phases of the software development life cycle, drawing and manipulating the graphical components of the models, guiding the user while developing the models, checking the correctness and the consistency of the information shared between the models, and supporting reuse of information between the models.

The main function of the Directory subsystem is to present the names of the objects in the project under development and their relations with each other, such as the use-cases and the classes.

The task of the Development Life Cycle subsystem is to show the learner of object-oriented software engineering the models that have been and should be developed, which give him an idea about his progress in the development process.

The Data Dictionary subsystem generates and updates the documentation of the project automatically during the software development life cycle.

The Database subsystem operates as a central repository for all the information in a project. It manages the information in the project and facilitates the communication between the LCCASE tool subsystems.

The User Interface subsystem is responsible for making the LCCASE tool easy to use, learner centred, and efficient.

1.3.4 The Software Engineering Process

The LCCASE tool has been developed using object-oriented software engineering techniques. The life cycle of the LCCASE tool includes the analysis, design, and implementation phases. The Next Generation of the Fusion method [Coleman 1997a], which is described in chapter 2 has been followed in organising the software development process of the LCCASE tool in a systematic manner.

The analysis and design phases have been carried out for all the models supported by the LCCASE tool. As time was limited, the Use-Case Model, Class Diagram, Scenario, and Contracts are fully implemented, whereas Collaboration Diagram, Design Class Diagram, and Code Generation have not been implemented. However, the LCCASE tool has been designed to support the development of these models in the future.

1.4 An Overview of the Chapters of the Thesis

The following is a brief description of each chapter in the thesis.

Chapter 2: Background

This chapter covers the background of various issues involved in the philosophy, design, and development of the LCCASE tool. The chapter provides an overview of the software engineering and object-oriented software engineering fields, explains important features of object-orientation, and discusses various styles of the software development life cycle.

The chapter then gives an overview and evaluation of famous object-oriented software methods. The chapter then discusses the important issues of CASE tools such as their definition, benefits, types, evolution, and philosophies. Next, examples of CASE tools are reviewed and evaluated.

Finally, the chapter explains the goals, difficulties, and solutions of teaching software engineering.

Chapter 3: Learner Centred Computer Aided Software Engineering (LCCASE) Tool

This chapter discusses the need and the philosophy of the LCCASE tool. In addition, it describes the features and the benefits for the experienced developer and the learner.

Chapter 4: The Design of the LCCASE Tool

This chapter discusses the design of the architecture of the LCCASE tool and the interface between its subsystems. Later the chapter describes the environment of the LCCASE tool and explains the linkage of information between the models.

Chapter 5: An Example to Show

This chapter shows an example of developing a project using the LCCASE tool. It describes the process of developing the models. In addition, the important features of the LCCASE tool and the roles of its subsystems are highlighted.

Chapter 6: The Implementation of the LCCASE Tool

This chapter discusses the implementation of the LCCASE tool using the Next Generation of the Fusion method. The models developed during the analysis, design, and coding phases are explained. In addition, the implementation of the linkage of information between the models is shown.

Chapter 7: Conclusion

This chapter discusses the conclusions, impacts, and contributions of the work and provides an insight into future work.

2 Background

This chapter discusses the most important features of software engineering in general and addresses object-oriented software engineering in particular. In addition, a brief description and evaluation is given of several well-known object-oriented methods such as the Fusion, Objectory, OMT, and Booch methods. Furthermore, CASE tools are discussed in detail and some examples of CASE tools are reviewed. Finally, the current situation of teaching object-oriented software engineering is discussed.

2.1 Software Engineering

The software development process refers to the life cycle of the software that starts with preparing a requirements specification and ends with producing a successful software application.

The software development process is classically split into three phases to reduce the complexity of developing large software. These phases include analysis, design, and implementation.

The analysis phase involves defining the problem domain and determining what the system is required to do. The analysis phase produces a Requirements Specification.

The design phase transforms the Requirements Specification into a Design Specification. The design recognises the solution to the problem and how the system will behave.

The implementation phase receives the Design Specification as input and produces the code. Figure 2-1 shows the phases of the software development process.

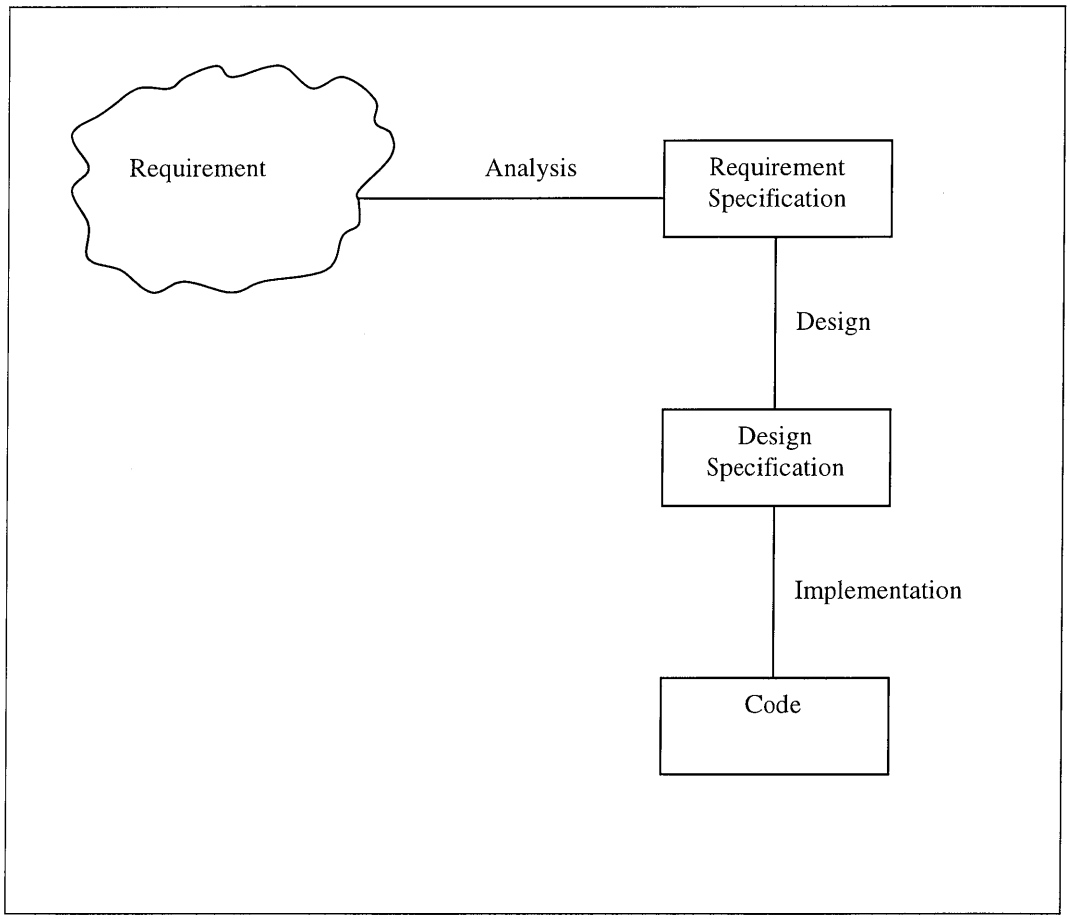


Figure 2-1 The Process of Software Development

2.2 Object-Orientation

Object-Oriented means that the software is organized as a collection of discrete objects that combine data and behaviour. A class is the basis of a group of objects with similar properties and common behaviour [Rumbaugh 1991].

The relationships between classes such as inheritance and aggregation are very powerful features of object-orientation. The attributes and methods of the parent class are inherited by the children classes. The child class can add more attributes and methods of its own. Inheritance reduces duplication and allows existing classes to be reused.

Aggregation is the "part-whole" or "a-part-of" relationship. It is a composition relationship where objects are parts of other objects [Parsons 1994].

There are many important object-orientation concepts that support the software development process.

2.2.1 Abstraction

Abstraction is "the extraction of the essential properties while omitting the inessential details" [Ross 1975]. It is a fundamental concept for controlling complexity [Booch 1983]. Complex systems can be easily understood as levels of abstraction. The higher level specifies the important concepts and the details appear in lower levels.

2.2.2 Decomposition

Like abstraction, decomposition is a concept for controlling complexity. It is the process of finding the parts to be used when building an abstraction. Object-oriented development breaks down a complex system into parts that preserve their real-world characteristics [Anderson 1991].

2.2.3 Modelling

Abstraction and decomposition are powerful concepts for controlling and reducing the complexity of system development but require clear, easy, and rich communication to be effective. Modelling is a communication mechanism that represents a selected set of abstractions [Anderson 1991].

2.2.4 Encapsulation

Encapsulation is the process of sectioning the components of abstraction, which are the structure and behaviour. Encapsulation separates the interface (behaviour) of an abstraction from its implementation (structure), which is usually achieved through information hiding [Booch 1994].

2.3 Object-Oriented Software Development

The object-oriented software development process starts with the analysis phase by recognizing the problem domain as a collection of classes of real-world objects linked by relationships. The relationships between classes are described using inheritance to show the classes hierarchy, association to clarify the interaction and visibility between the classes, and aggregation to construct complex classes out of small ones. In addition, the attributes of the classes are specified.

Next, in the design phase the concept of the objects and the communication between them are clarified. The decisions involved in this phase should support the development of quality software that maintains modularity, reusability, extensibility, understandability, cohesion, and reliability.

Finally, during the implementation phase the programs are written in an object-oriented language.

There are many advantages in using the object-orientated software development style: the data abstraction concept of object-orientation makes the details of classes representation visible only through its methods so different implementations of a class can be used without changes to the class code. The object-oriented analysis and design

models improve the communication between the development team and make the evaluation of alternative designs easy [Malan 1995]. In addition, the software components are compatible because formal procedures are used for constructing classes and their interfaces. Moreover, the combination of methods and data in the classes helps in developing reusable software and some object-oriented techniques such as inheritance support system extension. Finally, the modularity feature of the classes structure helps in containing the effects of changes during the system maintenance [Coleman 1994].

The advantages of object-oriented software development discussed above have contributed to the choice of object-oriented software development to be the development style of the LCCASE tool.

2.4 The Styles of Software Development Life Cycles

There are many styles to manage the processing of the phases of the software life cycle. The next subsections discuss the Waterfall, Iterative and Incremental, and Evolutionary life cycles.

2.4.1 Waterfall Life Cycle

In the Waterfall life cycle (Figure 2-2) the phases of the software development process are developed sequentially. First, the developer starts with the analysis phase where system requirements are defined. Then the developer proceeds to the next phase, which is the design phase. Finally, the implementation phase is performed. Within the Waterfall approach there is no intermediate software product. Moreover, it assumes that all the user requirements are known at the beginning so that well-defined processes can be executed sequentially. But this assumption is not valid since it is very difficult to know all the

requirements ahead of the project development. Even if all the requirements are known, they often change during the project development period [Williams 1996].

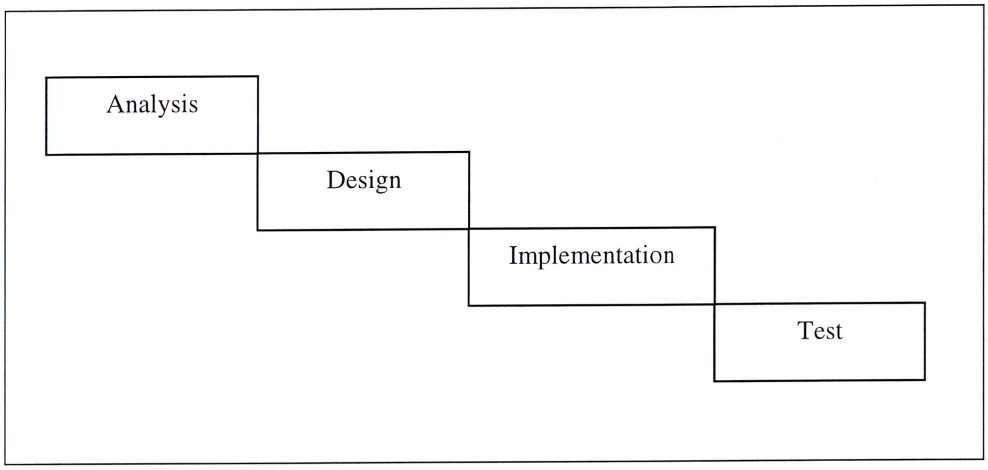


Figure 2-2 The Waterfall Life Cycle

2.4.2 Incremental and Iterative Development Life Cycle

The iterative approach replaces the implementation phase of the Waterfall life cycle process with smaller implementation cycles. Each cycle contains all the life cycle phases. The first process of the incremental approach is to find user needs and system requirements. The next process is to develop the life cycle phases, which are the analysis, the design, and the implementation in sequence [Cotton 1996]. Figure 2-3 shows the Incremental and iterative life cycle. The iterative approach produces an early prototype of the system, which allows the developer to perform several tests against the system requirements. This reduces risks during the software development process.

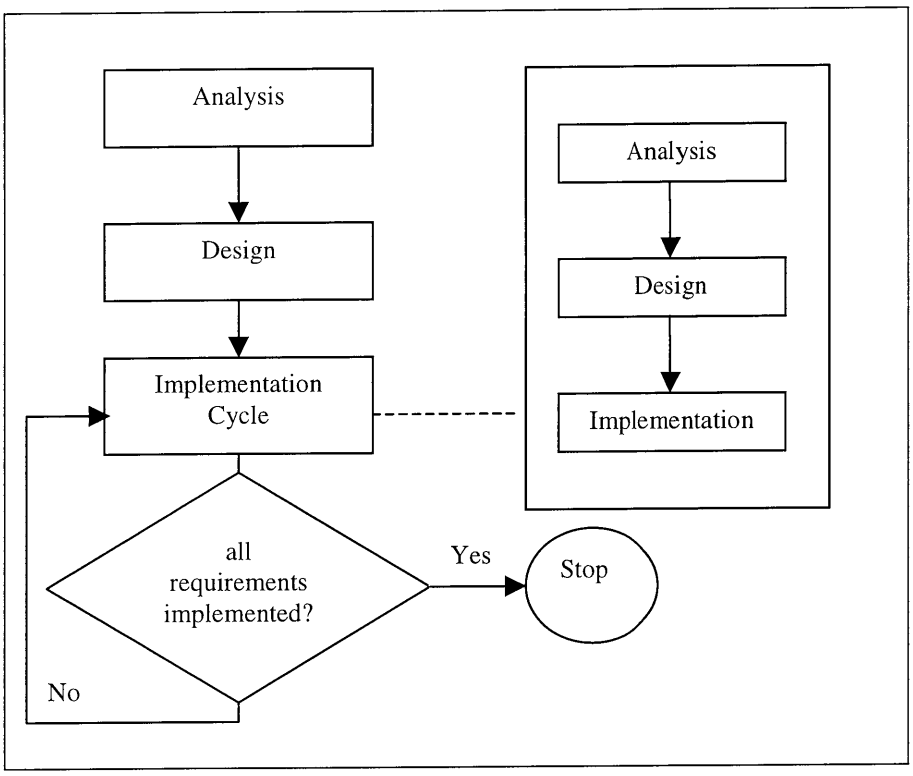


Figure 2-3 The Incremental and Iterative Life Cycle

2.4.3 Evolutionary Life Cycle

Evolutionary development differs from the incremental and iterative development approach with its beliefs that all the user requirements and needs cannot be defined at the beginning. The initial user requirements are defined at the beginning and they are refined at each cycle. The Evolutionary approach adds a stronger user base as it receives user feedback at the end of each cycle [Cotton 1996]. Moreover, it can adapt to changes in user requirements [Williams 1996]. Figure 2-4 shows the evolutionary life cycle.

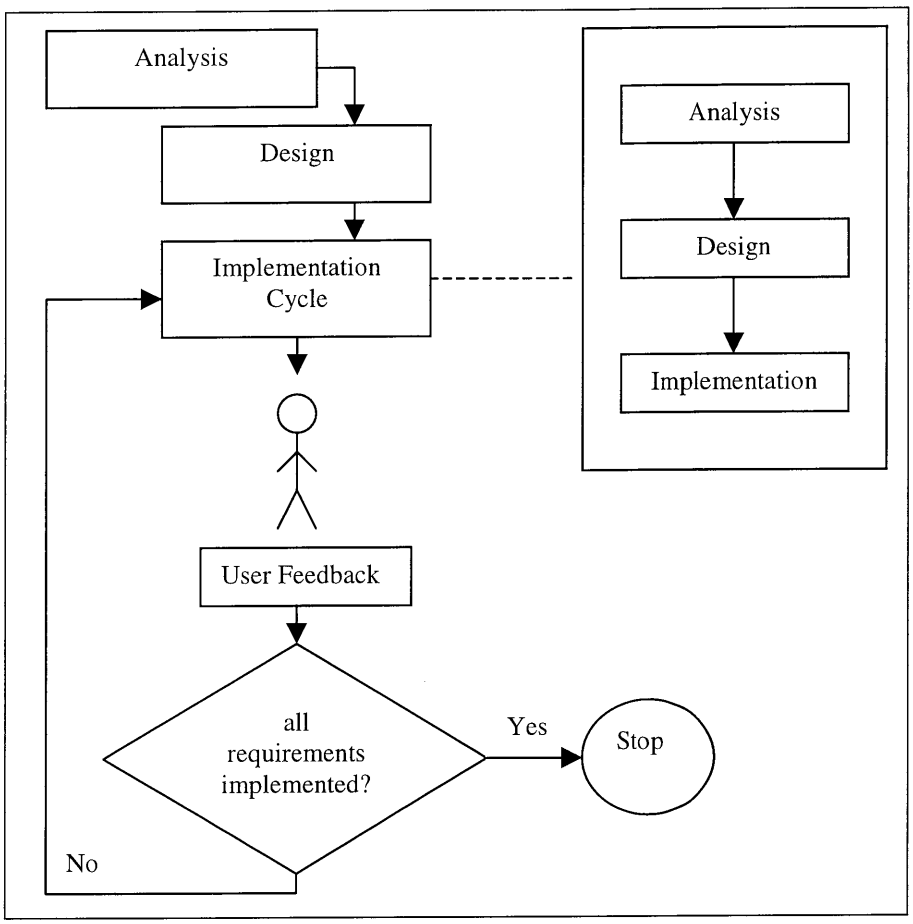


Figure 2-4 The Evolutionary life cycle

2.5 Object-Oriented Methodology

A methodology is a systematic approach used to accomplish a task [Vessey 1992]. Object-oriented methodology means that object-orientation features are implemented within the methodology. There are many object-oriented methodologies available for software development. Each methodology has its own terminology, concentrating on some phases and activities of the life cycle, and using different techniques and products for implementation [Jurik 1992]. Object-oriented methods use popular and well-defined notations, implement the entire system development life cycle, support team management [Coleman 1994], and support system maintenance. Finally, recent project experiences

found that Object-Oriented Analysis and Design methods are mature enough to improve the systems development processes [Malan 1995].

2.5.1 Advantages of the Object-Oriented Methods

There are many benefits obtained from using the object-oriented methods [Coleman 1994, Jurik 1992], which are summarised as follows:

- During the analysis phase and with the help of the system modelling techniques, the requirements can be checked thoroughly, many ambiguous features in the system requirements are explained clearly, and the ignored or omitted features can be found easily.
- The different models of the analysis phase increase the level of understanding of the problem domain and clarify its concepts, which help in adapting the changes in the system requirements into the system.
- The models developed in the analysis and design phases discuss the problem at different level of abstraction so that different alternatives of the system design can be produced which leads to a strong basis for implementation.
- The analysis and design phases facilitate dividing the problem into related parts, which help in managing and developing large systems by a team of developers.
- The notation used in the analysis phase helps improve the communication between the developers themselves and between the developers and the users.
- Software maintenance is generally done by people different from the original developers. The analysis and design documents help in understanding the system and the code.

- The use of the preconditions, post-conditions, assertion-derived operations, invariant assertions, and exceptions within the object-oriented methodologies has lead to the development of more reliable software.
- The focus on the reuse in all of the software life cycle phases is an important requirement within object-oriented methodologies.
- Object-oriented methodologies cover all the life cycle phases. Moreover they provide a systematic approach to the development of the system phases.
- Documents developed by object-oriented methods are more readable, understandable, and testable than those of functional methods.

In the following sub-sections some object-oriented methods are discussed, a brief description is given about each method with an evaluation of its object-oriented features and support. This evaluation was necessary to decide which method the LCCASE tool automates.

2.5.2 The Fusion Method

The Fusion method [Coleman 1994] is a software development method for object-oriented software. It uses a concise but comprehensive set of well-defined notations. It covers all the life cycle phases of the system development process from the analysis phase to the implementation phase. It supports re-use and re-engineering. In addition, the Fusion method has many consistency checks that should be done at the end of each phase to make sure that the results of each phase are consistent and complete. Furthermore, the Fusion method has a data dictionary where all the entities of the system are listed and described to support the system documentation, which will support the teamwork.

2.5.2.1 The Fusion Life Cycle

The Fusion method covers all the three phases of the software development process, analysis, design, and implementation (Figure 2-5). Each phase involves developing several models [Coleman 1994].

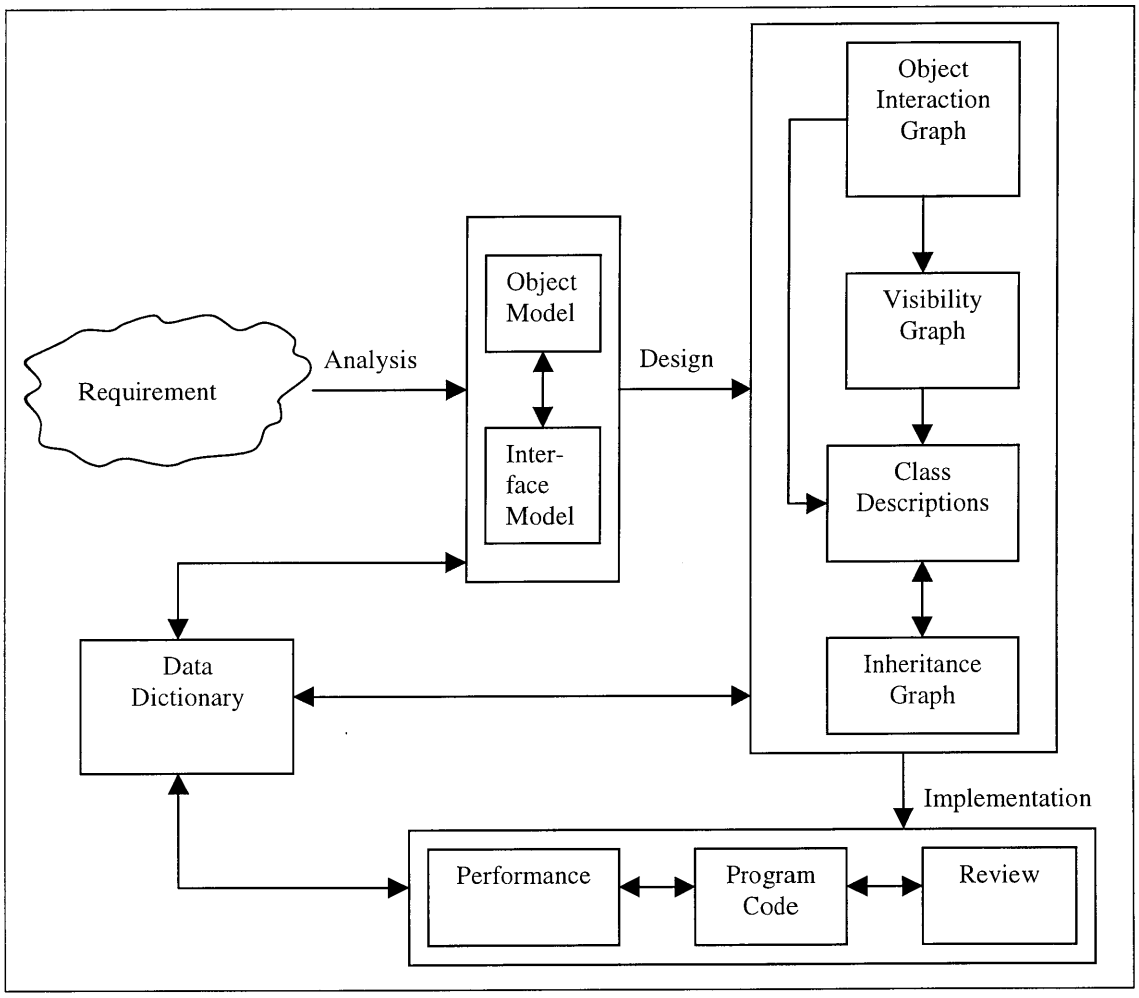


Figure 2-5 The Fusion life cycle.

The Analysis Phase

The analysis phase describes what the system does. The input to the analysis phase is an informal requirements document and the output is a clear and precise set of models. The developer starts with developing the Object model, then develops the Interface model. Finally, the analysis models are checked to find if they are complete and consistent.

Object Model

The Object Model is developed from the requirement specifications by finding the concepts in the problem domain and the relations between them. It consists of classes, attributes, and the relationships between classes, which are association, inheritance, and aggregation. Moreover it shows the cardinalities and the roles within the relationships. Figure 2-6 shows the components of the Object Model.

Interface Model

The Interface Model is concerned about the system behaviour and identifies the system users, system operations, and events. The user of the system performs some operations (inputs) on the system that might change its state, then the system respond to them with some events (outputs). These operations are called system operations. The Interface Model consists of two models, which are the Operational Model and the Life Cycle Model. In addition, Coleman recommends the development of scenarios for the system operations to help with the development of the models.

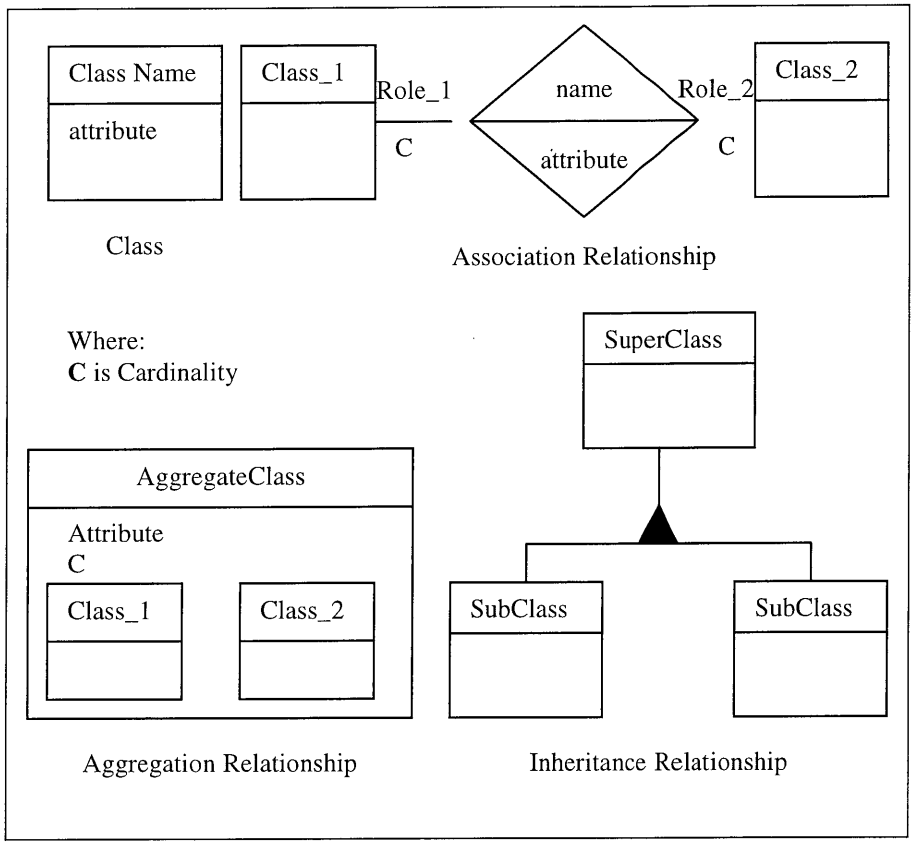


Figure 2-6 The Components of the Object Model

Operational Model

The Operational Model (Figure 2-7) is a very valuable model, which can provide the system developer with a great help during the design and the implementation phases. The Operational Model is concerned with the semantics of a system operation. It describes the system operation inputs, results, and attributes that change as a result of the system operation, pre-conditions, and post-conditions.

Life Cycle Model

The Life Cycle Model (Figure 2-8) shows the sequences of the system operations. It can easily be developed from the system scenarios.

Operation:	operation name
Description:	Description of operation
Reads:	attributes
Changes:	attributes
Sends:	attributes
Assumes:	preconditions
Result:	postconditions

Figure 2-7 The Operational Model

Life cycle name : Regular_Expression
(name = regular_Expression)*
where:

Regular_Expression	Name, Operation
Name	Any operation name
Operation	Concatenation: . , Alternation: |, Repetition: * , Optional: [], Interleaving: ||, Grouping: ( ).

Figure 2-8 The Life Cycle Model

The Design Phase

Four design models are developed during this phase: Object Interaction Graphs, Visibility Graphs, Class Descriptions, and Inheritance Graphs. Consistency checks with the models developed at the analysis phase should be made for each model in the design phase. A brief description of each model follows:

Object Interaction Graphs

The Object Interaction Graphs (Figure 2- 9) show how objects interact at run time to support the operational model functionality. An object interaction graph is developed for each system operation. It consists of sequences of messages that occur between a collection of objects in a specific operation.

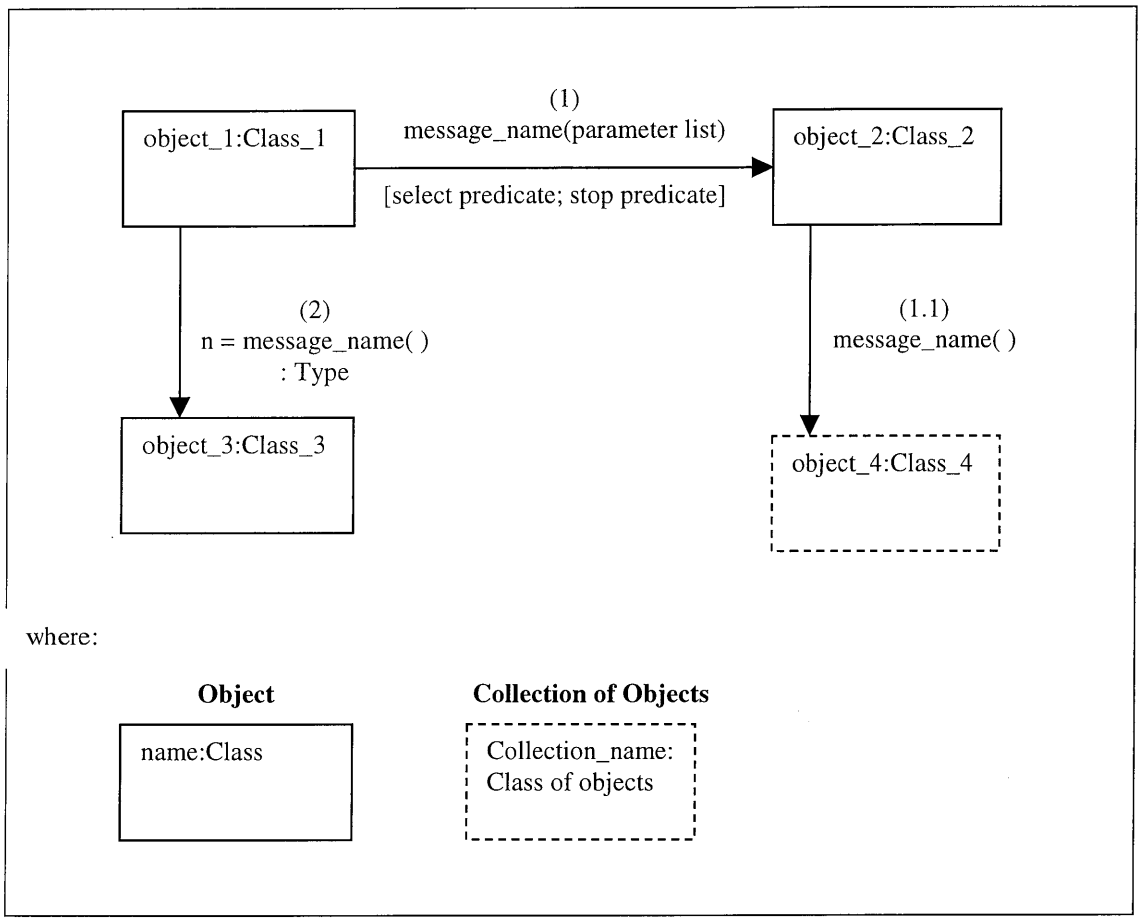


Figure 2- 9 The Object Interaction Graph.

Visibility graphs

The Visibility Graph (Figure 2-10) shows the object communication path. It defines the reference structure of the classes in the system by finding the class instances that need to be referenced and the suitable kinds of references to those objects.

Class Description

The Class Description is a specification with data attributes, class interface, object reference attributes, and method signature for all classes in the system.

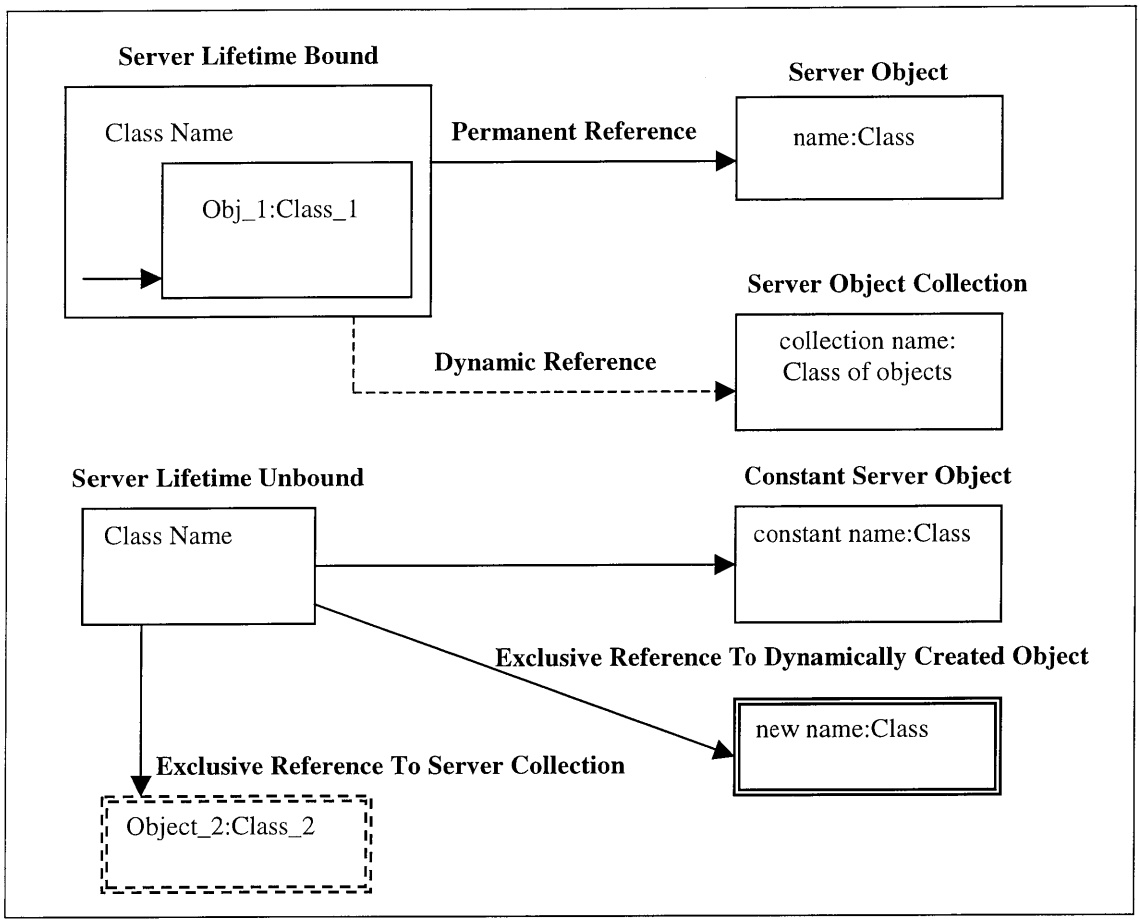


Figure 2-10 The Visibility Graph

Inheritance Graph

The Inheritance Graph shows the inheritance structure of the classes. The inheritance structure is developed by exploring the classes to find abstractions and commonalties.

The Implementation Phase

The implementation phase is the last phase of the Fusion method where the design is mapped into code. The input to the implementation phase is the Class Descriptions, Object Interaction Graphs, the data dictionary produced by the design phase, and the Life Cycle model generated by the analysis phase. The implementation phase consists of three parts, coding, performance, and review.

Coding

Coding means translating the output of the design into the implementation language. The Class Descriptions are translated into the class notation of the implementation language. The Object Interaction Graphs define the bodies of the methods. The Life Cycle model is translated into a state machine. The methods should implement the functions, types, and predicates available in the data dictionary.

Performance

The performance of the system should be considered throughout the implementation phase. If the system is wasting any resource such as space or time, the code should be traced to find the place where the resource is being consumed. Then the code should be modified.

Review

The code should be reviewed, inspected, and tested to find and remove the errors before the system is used.

Data Dictionary

The Data Dictionary is a central repository that stores the definition of terms and concepts. It supports and increases the semantics and clarity of the Fusion models. It provides considerable support while crosschecking of the models for consistency and completeness.

2.5.2.2 An Evaluation of the Fusion Method

The Fusion method is successful by capturing the objects in the environment of the system using the Object Model and specifying the behaviour of the system using the Interface Model.

While the Object model is able to implement object-oriented techniques such as decomposition, inheritance, aggregation, and association, it fails to represent the decomposition of complex systems into subsystems.

I consider the Operational model very valuable for describing the behaviour of the system and has been very useful for constructing the models of the design phase during my implementation of the LCCASE tool. The pre- and post-conditions clarified the message passing between the models. The Operational model has been included in the Next Generation of the Fusion method and the UML notation in the form of the Contracts.

The Life Cycle model includes many grammatical notations, which are not easy to remember. This can add an unnecessary burden to the developer.

The Object Interaction Graph supports the description of the messages between the objects but the message lines are crowded with information, which might make them difficult to construct.

The construction of Class Description involves gathering information from the Object model, Object Interaction Graphs, and Visibility Graphs and writing it in the form of a class definition using a pseudo language. The system engineer then converts the Class Description to a source code of a programming language. This adds unnecessary

work to the developer since the developer can write the code directly especially since the gathered information has been well specified in the analysis and design models.

The Fusion method has an important feature, in that it provides step-by-step support to the developer and has many completeness and consistency checks, which make it suitable for both inexperienced and experienced developers.

2.5.3 Evolutionary Fusion

A variation of the Fusion method called Evolutionary Fusion has been developed by Todd Cotton, a colleague of Derek Coleman, the developer of the Fusion method. The Evolutionary Fusion method consists of two major phases, definition and development [Cotton 1996].

The Definition Phase

The Definition Phase is distinguished by its communications. It emphasizes the communication between the system development team themselves and between them and the users of the system. During this phase the Scenario, Object model and Operation model of the Fusion method are developed. At the end of this phase a plan for the next phase is developed. The plan consists of three parts: Firstly, the key roles' ownership assignment such as the project manger and the technical manager. Secondly, defining the standard of the evolutionary approach cycle such as determining the cycle length. Finally, grouping and giving priority to functionality by completing the high risk functions first.

Development Phase

The Development Phase contains the evolutionary cycles. Each evolutionary cycle consists of the same steps, refining the analysis models, developing the design models such as the Object Interaction Graphs, Visibility Graphs, Class Descriptions, and the Inheritance Graphs, coding and validation, and obtaining the customer feedback which is done in parallel with the other tasks of the evolutionary cycle.

2.5.4 The Next Generation of the Fusion Method

A new version of the Fusion method has been developed, called the Next Generation of Fusion by the author of the Fusion method and others. The Next Generation of the Fusion method (Figure 2-11) has the same fundamental ideas of the Fusion method but it adopts the UML notation. In addition, features are added to enhance the support to the architectural design and the teamwork [Coleman 1997a].

The phases of the Next Generation of the Fusion are requirements, analysis, architecture, design, and implementation. In addition, it has two more supplementary specialist activities, GUI and database design [Coleman 1997a, Coleman 1997b].

Requirements

The goal of the requirements phase is to capture the behaviour of the system and present it in a form that the user can understand. The requirement phase defines the functional and non-functional requirements of the system in terms of use-cases, use-case specifications, and non-functional requirements.

The non-functional requirements include quality measures for the system, such as usability and reliability. In addition, it defines the constraints of the system.

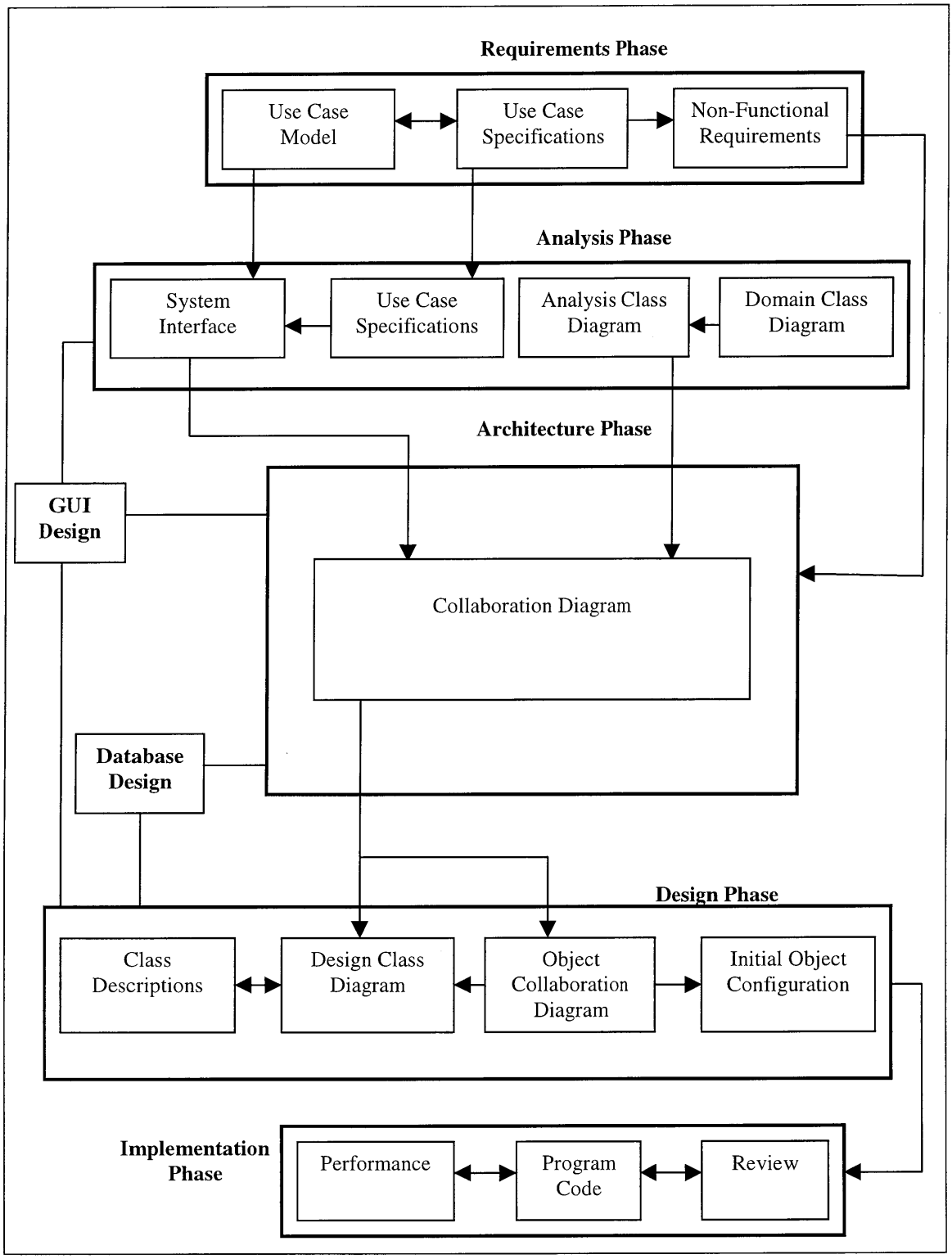


Figure 2-11 The Next Generation of the Fusion method

The requirements phase include:

- the Use-Case models from the requirements specification by finding the actors, use-cases, and drawing the diagrams;
- the use-case specifications, which define the purpose of the use-cases, its constraints, and the sequence of interactions between the actor and the system;
- the non-functional requirements, such as the system quality measures and its constraints.

Analysis

The analysis phase defines what the system does and describes its behaviour. During the analysis phase Class Diagrams and system interface in terms of system operations and events are developed.

The activities include developing:

- Class Diagrams by defining the classes from the system environment and their attributes and finding the association, inheritance, and aggregation relationships;
- system operations from the developed use-cases. This requires finding the inputs, outputs, changes, and pre- and post-conditions of the system operation.

Architecture

The architecture of the system specifies the system components and their interconnection. The architecture phase describes the collaborations between the system components, and a logical architecture, which specifies the messaging interfaces. The output of this phase is the Collaboration Diagrams in UML notation.

A Collaboration Diagram is developed for each system operation that has postconditions. It shows the sequence of messages between the objects in the project.

Consistency checks must be made between the collaboration diagrams and the class diagrams to verify the objects.

Design

The design phase specifies object-orientation structures such as Design Class Diagrams and object Collaboration Diagrams, which satisfy the abstract definitions produced from analysis and design phases.

The Design Class Diagrams are developed by adding methods such that each system operation is a method, aggregations to the class diagrams and specifying the visibility of the classes.

Implementation

The implementation phase translates the design into code. The system developer maps the Design Class Diagrams, object Collaboration Diagrams, and information from the data dictionary into useful implementation.

Coding starts by writing classes definitions, which involves finding the attributes and methods headings from the Design Class Diagrams and the parent class of the class. Next, the code is written for the method bodies. Finally, the code is reviewed to check its correctness and quality.

GUI Design

The design phase of the graphical user interface produces use-cases and Class Diagrams, which define windows and user interface navigation. It is carried out in parallel with the analysis and design phases.

Database Design

The database design phase is performed if the system uses a database. It produces a Design Class Diagram to design the database schema and system operations to define database transactions.

2.5.4.1 An Evaluation of the Next Generation of the Fusion Method

The Next Generation of the Fusion method possesses all the advantages of the Fusion method. In order to meet the requirement for standardisation in the language and notation for modelling, the Next Generation of the Fusion method adopted the UML notation.

The Next Generation of the Fusion method captures the behaviour of the system using use-cases and system operations. The use-cases provide the developer with a complete view about the interaction between the user and the system. In addition, the ability to map the use-cases to Scenarios and Contracts in the analysis phase, then to its Collaboration Diagrams in the design phase, and then to code in implementation phase has a great value in verifying the correct and sufficient implementation of the requirements.

Collaboration Diagrams in the Next Generation of the Fusion method replace Object Interaction Graphs of the Fusion method. Even though Collaboration Diagrams are useful for describing the message passing between the objects, they suffer from the same

drawback of the Object Interaction Graphs discussed in the evaluation of the Fusion method.

The Next Generation of the Fusion Method includes Design Class Diagrams to describe the visibility, attributes, and methods of a class within the design view. However, it still requires the development of Class Descriptions, which the author feels unnecessary, as discussed in the previous section.

The Next Generation of the Fusion method implements the iterative life cycle, which assists in managing the development process of large software. It supports the GUI and Database design, which are necessary for developing any current software system.

Like the Fusion method, the Next Generation of the Fusion method provides step-by-step support to the developer and has many completeness and consistency checks.

Due to the benefits and advantages of the Next Generation of the Fusion method, it has been chosen by the author to be automated by the LCCASE tool and to develop the LCCASE tool.

2.5.5 Object Modelling Technique (OMT)

The OMT method [Rumbaugh 1991] consists of four phases: the analysis, system design, object design and implementation. The object-oriented concepts can be applied throughout the system development life cycle. Figure 2-12 shows the OMT different phases and the models developed in each phase.

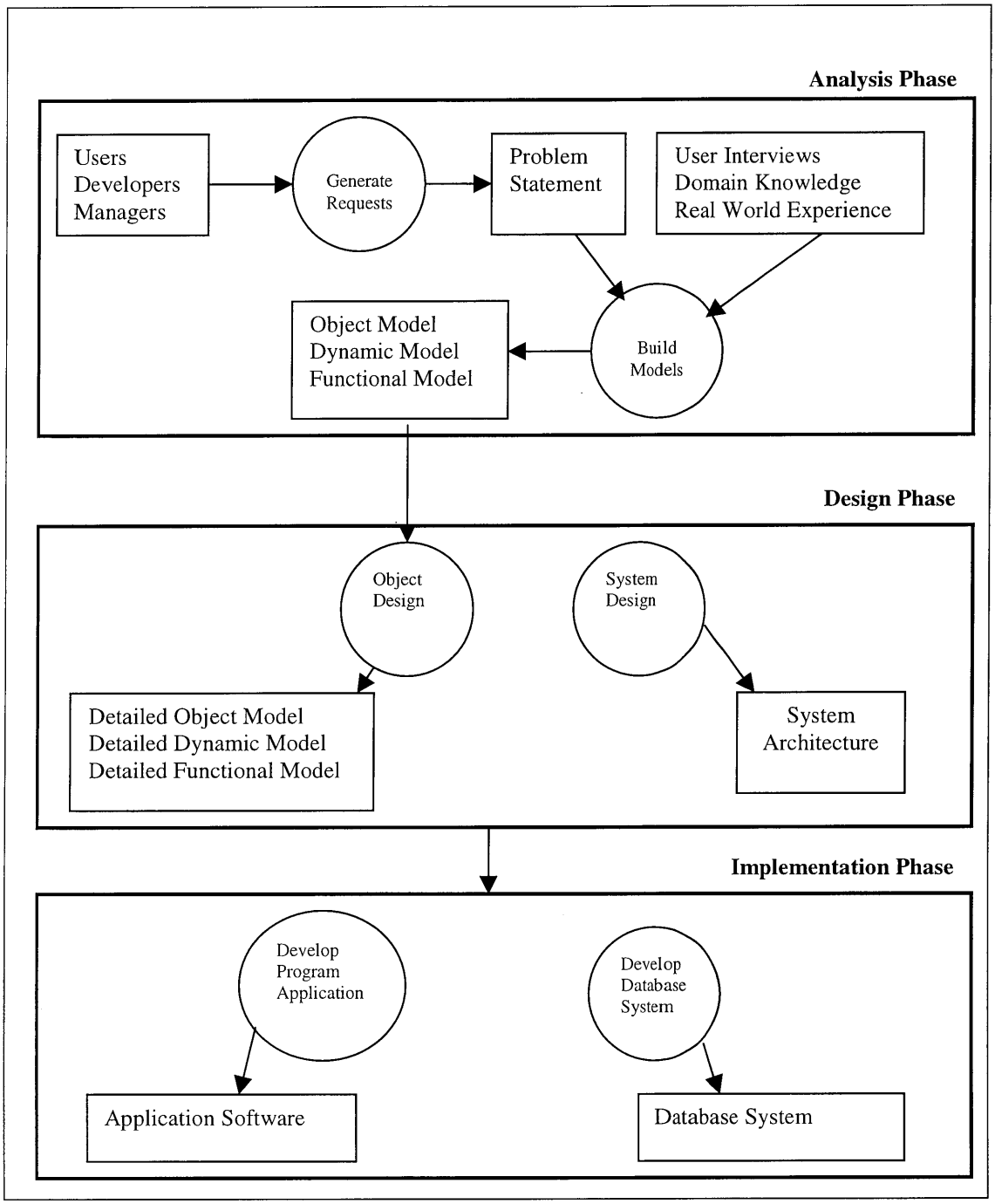


Figure 2-12 The OMT Method

2.5.5.1 The OMT Life Cycle

Analysis

The input to the analysis phase is the problem statement. The analyst then builds a model of the real world showing its important properties.

System Design

In the System Design phase, the system designer makes high level decisions about the system architecture. The system is divided into subsystems according to the analysis structure and the proposed system architecture.

Object Design

The object designer builds a design model based on the analysis model by adding implementation details, concentrating on the data structures and the algorithms required to implement each class.

Implementation

During implementation, the object classes and the relationships developed during object design are translated using a programming language into code.

The OMT methodology produces three models [Rumbaugh 1991]: Object Model, Dynamic Model, and Functional Model. These models describe the system being developed through the stages of the system development.

Object Model

The Object Model shows the static structure of the system and describes the structure of its objects such as identity, attributes, operations, and relationships. The Object Model is represented graphically using object diagrams.

Dynamic Model

The Dynamic Model illustrates the behaviour of the system in terms of how the system changes over time and the sequencing of operations. The Dynamic Model is represented graphically using state diagrams that show how the class objects react to events by changing their states and by performing operations.

Functional Model

The Functional Model describes the functions of the system and explains the transformations of the data value. It contains data flow diagrams [DeMarco 1979], which show the computation of the output values from the input values and the dependencies between them.

2.5.5.2 An Evaluation of the OMT Method

The analysis phase of the OMT method captures the object and the behaviour of the system. It links the Object Model with the Dynamic Model by corresponding the events in the Dynamic Model to the methods in the Object Model. However, OMT does not provide linkage between the Functional Model and other models.

The OMT method supports the construction of software with good object-oriented structure such as inheritance and aggregation. Its notation is very concise and clear. In addition, the Object Model supports teamwork.

The OMT method has been one of the first methods that provided support to developing software with rich object-oriented principles. In addition, it attempted to develop models in all phases of the software life cycle. However, they are mainly developed in the analysis phase.

The OMT method has provided a solid base for successor object-oriented methods, such as Booch and the Fusion methods.

2.5.6 The Objectory Method

The Objectory method [Jacobson 1992] (Figure 2-13) consists of three phases: requirements, analysis, and construction.

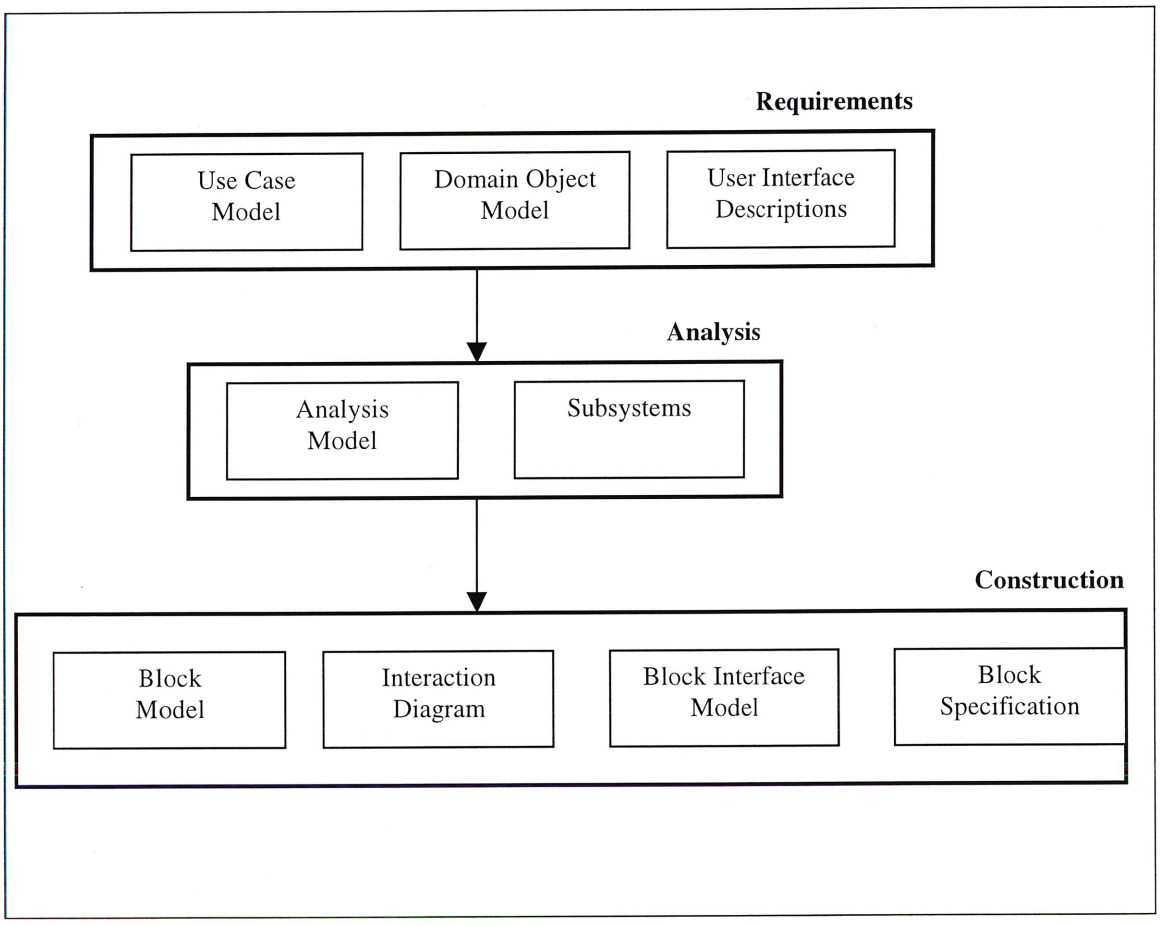


Figure 2-13 The Objectory method

2.5.6.1 The Objectory Life Cycle

The life cycle of the Objectory method consists of the requirements, analysis, and construction phases.

Requirements

The requirement phase has three models:

Use-Case Model

. The Use-Case Model is a dialog between the system and the user. The important components in this model are the use-case and the agent. The use-case represents a functionality of the system. The Agent is the user of the system, which can be a human, software, or another system. Each use-case should be documented with a textual description.

Domain Object Model

The Domain Object Model defines the problem domain. Its notation is similar to the entity-relationship model. It contains classes, object instances, and relationships such as associations, and inheritance.

User Interface Descriptions

The User Interface Descriptions show examples of user interface windows. The user participates in the development of User Interface Descriptions.

Analysis

The analysis phase improves and elaborates the requirement models. It is a form of the entity-relationship model. The analysis phase is developed incrementally by extracting the classes, objects, and relationships to implement the use-cases.

Construction

During the construction phase the analysis models are refined. The communication between the objects is clearly defined and the implementation details are discussed.

This phase produces four models, which are:

Block Model

In the Block Model the blocks of the system, which are abstractions of code modules, are developed from one or more of the analysis objects.

Interaction Diagrams

An Interaction Diagram, developed for each use-case, is a timeline diagram that shows how the blocks communicate by sending messages to each other or to the system.

Block Interfaces

The Block Interface describes the operational interface for each block. It is developed by determining the operations of a block in all interaction diagrams.

Block Specification

The Block Specification is an optional model that shows the internal behavior of the blocks as state machines.

2.5.6.2 An Evaluation of the Objectory Method

The Objectory method is a use-case driven method. The use-cases are not just used to capture the behaviour of the system but drive the whole development process. They heavily support the implementation of the requirements to develop a system that meets the user's requirements. The benefits and success of the use-case driven design introduced by the Objectory method have encouraged other methods such as the UML and Next Generation of the Fusion methods to adopt it.

The Objectory method provides strong support to capture the behaviour of the system but it lacks this strong support in capturing the classes and their relationships in the system.

2.5.7 The Booch Method

Booch method [Booch 1994] is used to develop object-oriented systems based on the Object Model. The method consists of two parts, which are the notation and the process.

2.5.7.1 Notation

Booch believed that clear, well-defined and rich notation is fundamental to the process of software development. On the other hand, he stated that drawing diagrams alone does not form the analysis and design. The notation is a method for extracting the reasoning about the behaviour and architecture of a system.

Class Diagrams

The Class Diagrams (Figure 2-14) consist of classes and their relationships. The Class Diagrams can be used in both the analysis and the design phases. It is used during

the analysis phase to explain the roles and responsibilities that specify the system behaviour. In addition, it is used during the design phase to find the structure of the classes that form the system architecture.

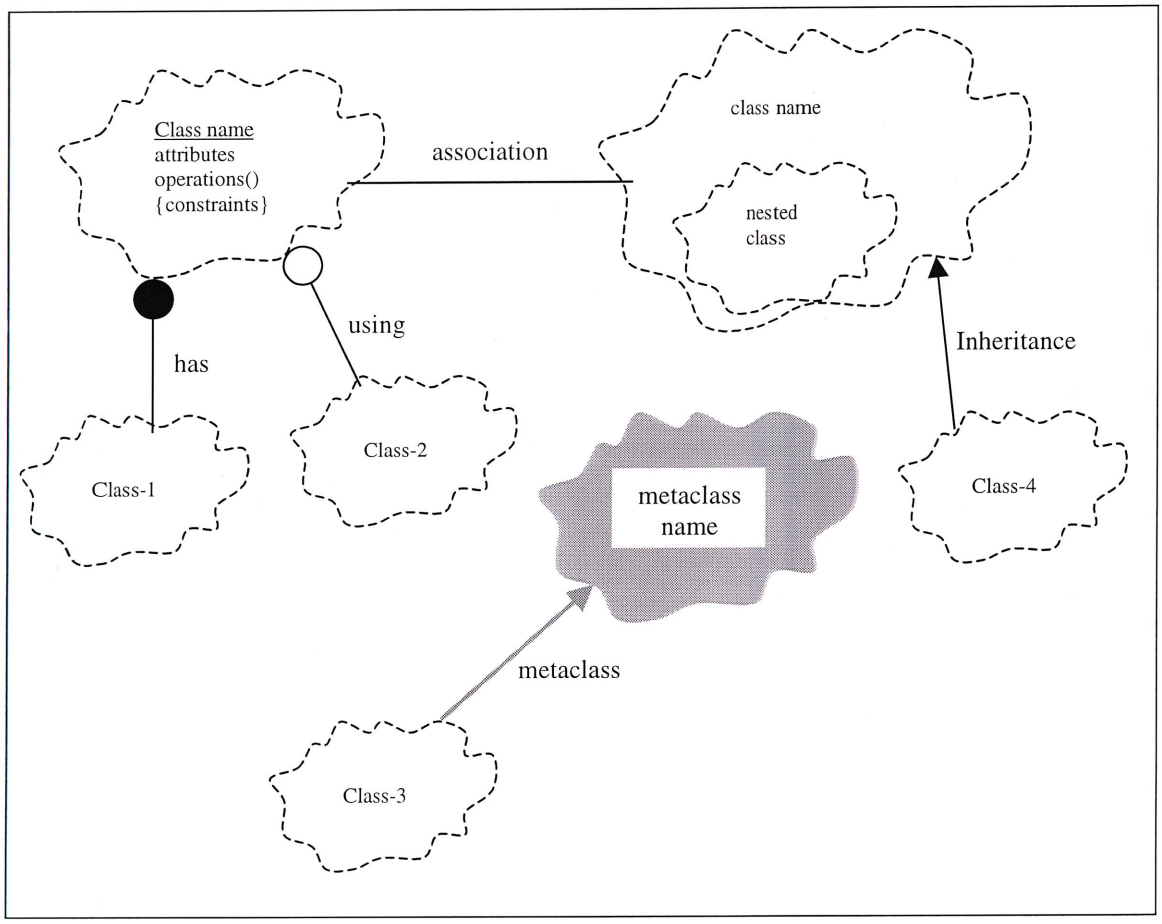


Figure 2-14 Booch Class diagram

State Transition Diagrams

The State Transition Diagram (Figure 2-15) shows the state of a specific class, the events that lead to the change in the state, and the actions that occur during a state change. Booch [Booch 1994] adopted the notation used by Harel [Harel 1987]. The two fundamental elements of a State Transition Diagram are states and states transition.

States

The states show the cumulative results of class behaviour. Each state has a name, while actions are optional.

State Transitions

The state transitions show the change in the state of the system as a result of an event that occurred. The state transition icon connects two states.

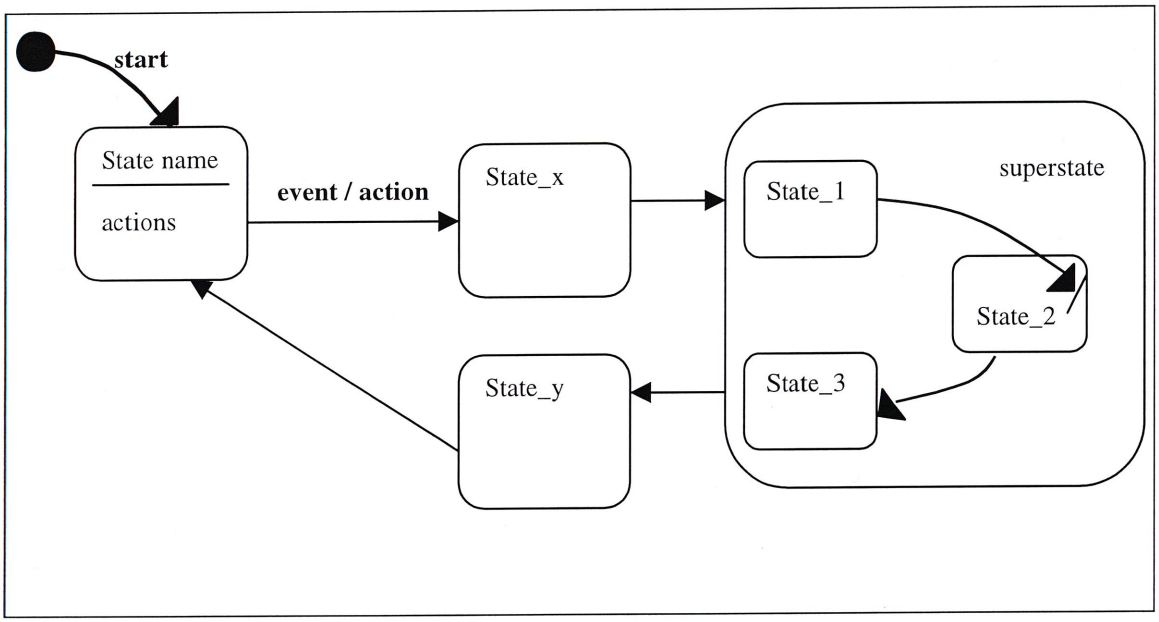


Figure 2-15 State Transition Diagram

Object Diagrams

Object Diagrams (Figure 2-16) show the existence of objects and their relationships in the logical design of a system. The two fundamental elements of object diagrams are objects and their relationships. Object diagrams facilitate the description of objects with complex invocations and permit the inclusion of other information such as links, attribute values, roles, data flow, and visibility.

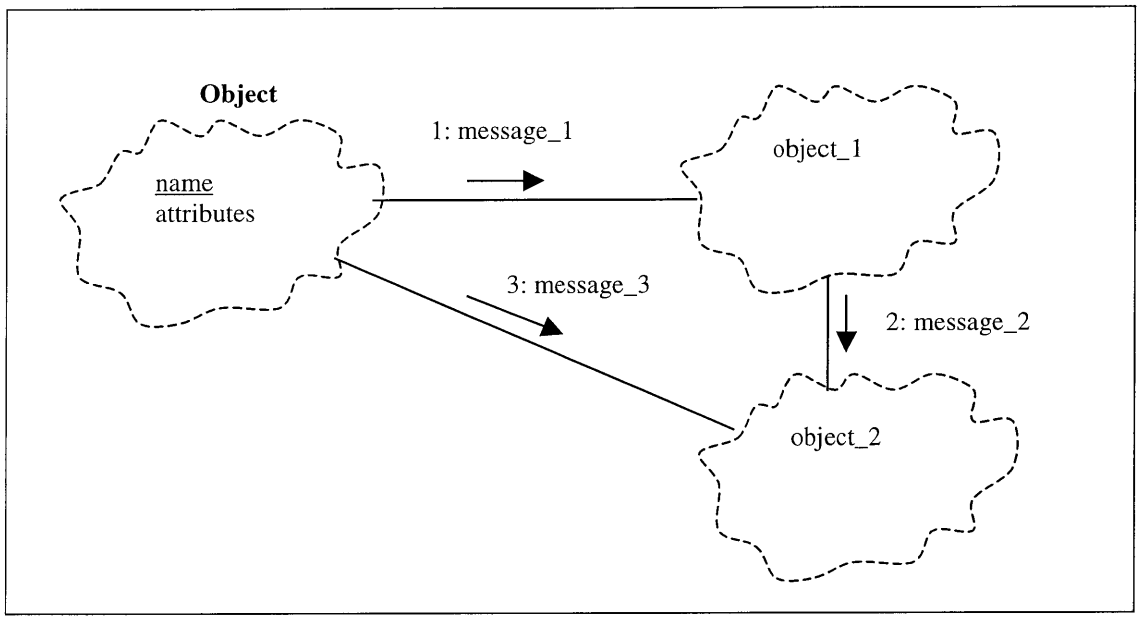


Figure 2-16 The Object Diagram

Interaction Diagrams

The Interaction Diagram (Figure 2-17) is an alternative way to represent the Object Diagram. Using such diagrams make messages passing in a sequence easier to read.

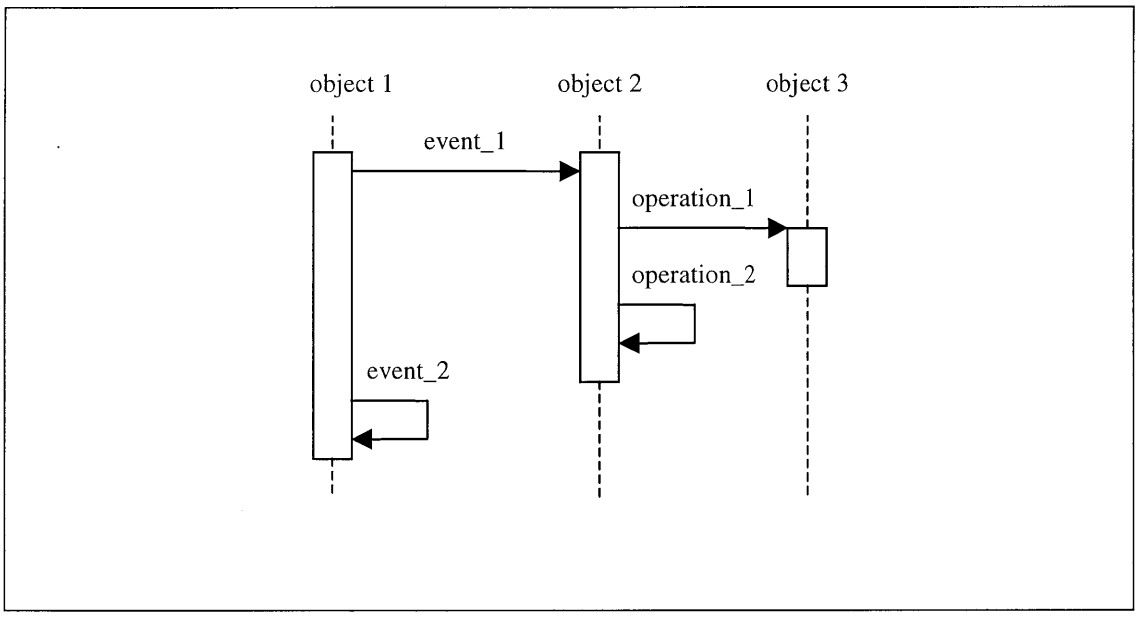


Figure 2-17 Interaction Diagram

Module Diagram

A Module Diagram shows the assignment of classes and objects to modules in the physical design of a system. The Module Diagram consists of modules and their dependencies.

Process Diagrams

A Process Diagram shows the assignment of processes to processors in the physical design of the system. The Process Diagram consists of processors, devices, and their connections.

2.5.7.2 The Process

The Micro Development Process

The scenarios and the architectural products produced in the macro process have an affect on the development of the micro process. In the micro process the analysis and the design phases are combined. Figure 2-18 shows the micro development process.

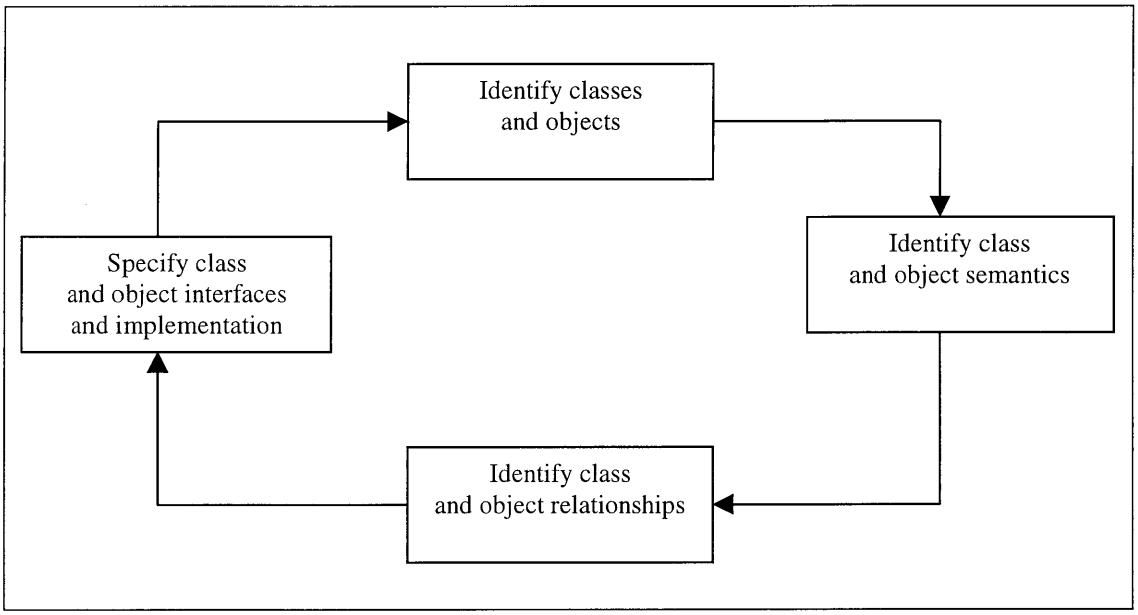


Figure 2-18 The Micro Development Process

The Macro Development Process

The macro process controls the framework of the micro process. The technical management of the development team is in charge of the macro process. In the macro process the analysis and design phases are separated and the process is reasonably well ordered. Figure 2-19 shows the macro development process.

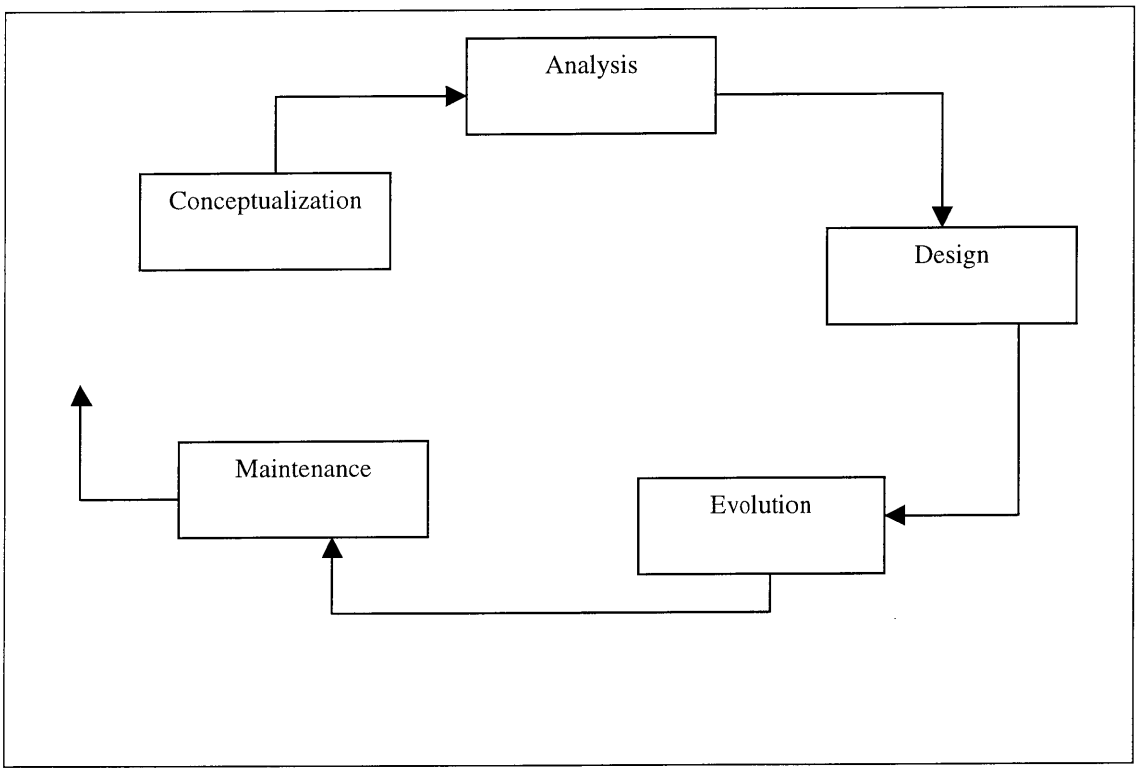


Figure 2-19 The Macro Development Process

2.5.7.3 An Evaluation of Booch Method

Booch method has a very rich and comprehensive notation but has poor process. The process does not follow the step-by-step approach, which makes it unsuitable to be used by the learners. The argument provided by Booch about this is that following a method for developing a system is not like following a cookbook, every problem is different. He might be right to a certain extent, but the method should contain general steps to guide

the developer. Booch notation is highly maintained. It supports the object-oriented structure, and covers almost all the object-oriented concepts. In addition, it supports the maintenance of the system and the teamwork because of its rich notation.

2.5.8 The Unified Modelling Language (UML)

The UML is a " language for specifying, visualization, and constructing the artifacts of software systems" [Rumbaugh 1998]. UML contains notations and their semantics for modelling object-oriented systems. In 1997 the UML was submitted as a candidate for a standard language and notation for modeling. It is widely accepted by industry. It has been developed by Grady Booch, Jim Rumbaugh, and Ivar Jacobson [Larman 1998].

2.5.8.1 The Notation of UML Method

The UML method adopts OMT, Booch, Objectory, and most other object-oriented methods. The notation of the UML method (Figure 2-20, Figure 2-21) includes:

- Use-Case Diagrams, which are similar to those of the Objectory method.
- Class Diagrams, which are hybrid of the Class Diagrams of those of OMT, Booch, and most other object-oriented methods.
- State Diagrams that are based on the state charts presented by David Harel with little modification.
- Sequence Diagrams, which can be found in other object-oriented methods bearing different names such as interaction, message trace, and event trace.
- Collaboration Diagrams were adapted from the Object Diagram in Booch method and the Object Interaction Graph in the Fusion method.

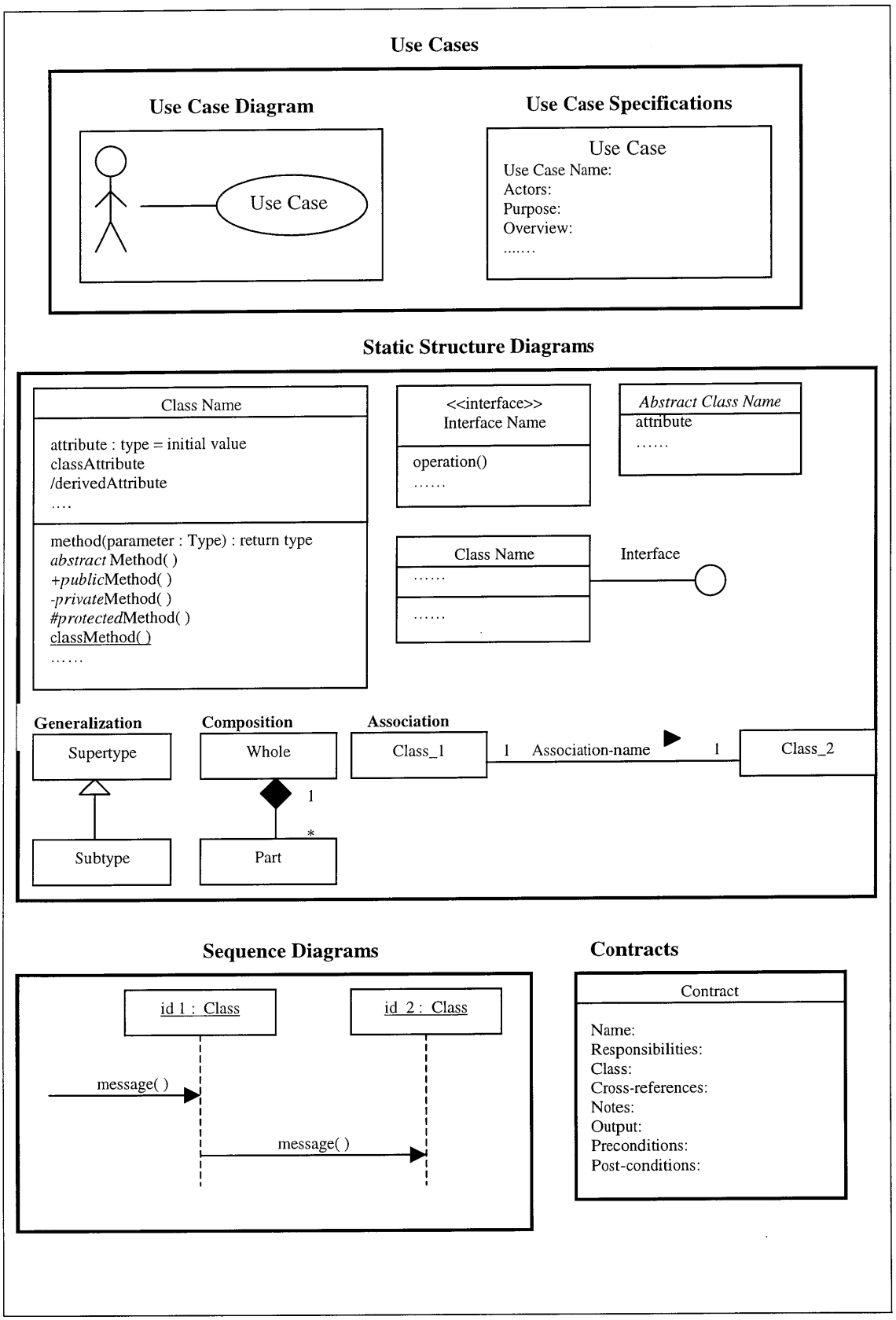


Figure 2-20 Sample of the UML notation

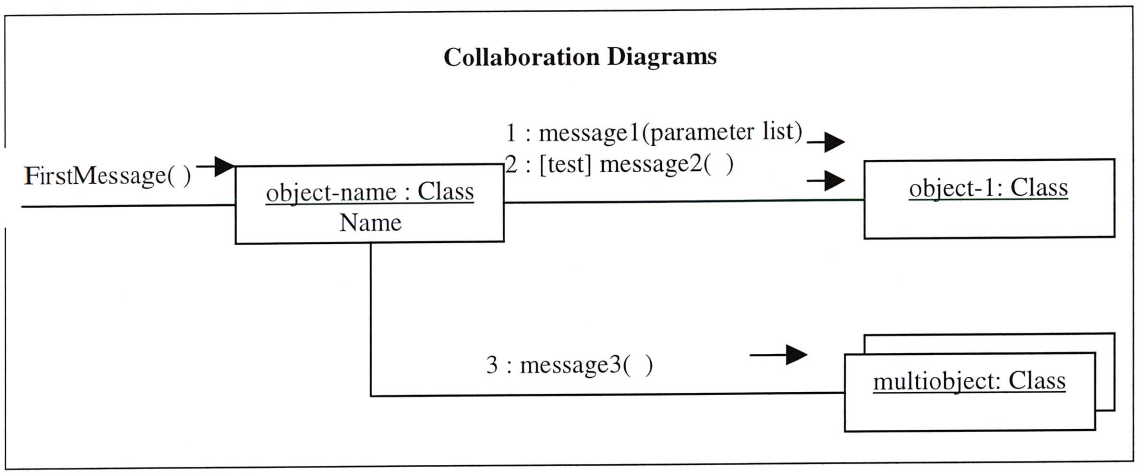


Figure 2-21 Sample of the UML Notation

2.5.8.2 An Evaluation of the of UML

The notation of the UML method is very rich since it has been derived from other well-known object-oriented methods and covers most OO concepts. It has the ability to represent subsystems using packages. However, the author feels it suffers from some drawbacks.

First, some fields are overcrowded with many important details. I feel if such fields were split up into more specific ones, they would be clearer and easier to construct. One example is the *Name* field of the Contract. It also contains the name of the Contract and a list of inputs for the system operation. The inputs include descriptions such as type, access modifier, and initial value. Another example is the post-condition field. It describes different details such as object creation, attributes modification, and formed or broken associations.

Second, Collaboration Diagrams can be crowded and difficult to construct. Messages sent from one object to another are represented by a line. Each message has an arrow associated with it indicating the direction. The diagrams can become crowded when there

are several messages between two objects. Instead this should be represented with a single pointing line. In addition, the messages contain many details such as, the parameters, output, and preconditions, which makes them difficult to construct.

While I was developing the LCCASE tool, the UML method was being introduced. At that time there has not been any evidence for the effectiveness of the method, especially in the teaching of software engineering field. In addition, there has not been any discussion in the literature regarding the experience of developing real and large projects using the UML method.

2.5.9 OO Methodologies can be Adopted as Needed to Meet the Requirements of a Project

The previous discussions about several well-known object-oriented methodologies show that every method has good and poor features. The opinion of the author is that object-oriented methods can be adopted as needed to fit the project requirement. Some methods contain models that do not have sufficient value to warrant the cost of developing them. For example, the Class descriptions in the Fusion method are not worth the time and cost of developing them as discussed in the previous section. In addition, most methods specify the order in which its different models should be implemented, but this is not always maintained. Moreover, models from other methods can be incorporated in other methods such as the use of the Use-Case Model from the Objectory method with the Fusion method [Malan 1995].

2.6 Computer-Aided Software Engineering (CASE) Tools

Since the emergence of computers, one of the main obstacles to their wider use is the great human effort required in developing systems. As a result of this, a major research area in computing has emerged to focus on building computing technologies that ease and automate the analysis, design, implementation and management of the software development process. This field is called computer-aided software or systems engineering (CASE) [Huang 1996, Lyytinen 1992, Stobart 1996]. An Object-Oriented CASE (OOCASE) tool is a CASE tool that applies the object-orientation concepts, features, and structures through the software development process from the analysis phase to the implementation phase to produce object-oriented software.

2.6.1 The Benefits of CASE

Yourdon says "Just as CAD/CAM technology has helped revolutionize various engineering disciplines over the past 25 years, so CASE technology will help revolutionise the software industry" [Yourdon 1986].

CASE tools have a great impact on the software development life cycle. They speed up the development process, increase the software quality, reduces the development and maintenance cost, improve the development productivity, support managing the software development process, and make the object-oriented concepts and techniques practical [Blanc 1992, Ho 1992, Iivari 1996, Gibson 1988, Norman 1989, Ramiller 1993, Stobart 1996, Pallato 1989].

2.6.2 Types Of CASE Tools

[Jarzabek 1998, Blanc 1992] described several types of CASE tools:

- Upper CASE tools support the analysis and design phases of the software development process.
- Lower CASE tools support the implementation phase of the software development process.
- MetaCASE tools generate CASE tools from a method specifications.
- MultipleCASE tools support multiple methods.
- Integrated CASE tool support all the software development phases.

2.6.3 The CASE Evolution

CASE tools passed through three successive generations since they were born. These generations are referred to as first, second, and next generation of the CASE tools.

2.6.3.1 First Generation of CASE Tools

First generation CASE tools developed in the early and mid 70's, including research prototypes, to automate the building process completely [Teichroew 1971]. The production function of CASE is considered thoroughly by these tools that were mostly laboratory systems to produce optimized code for large mainframe applications. The interfaces of those tools were text-based that had poor or non-existing support for graphical design methods. A few tools were developed commercially such as ISDOS system [Teichroew 1977].

2.6.3.2 Second Generation CASE Tools

The second generation CASE tools [Lyytinen 1992] developed since the mid 80's, have been used until the late 80's and early 90's to support the graphical design methods that were developed in the 70's. These CASE tools work mostly on personal computers or workstations. They include graphical interfaces for method support. Furthermore, second generation CASE tools have started developing and adopting disciplined and methodical approaches to systems development.

The second generation CASE tools support the development of dataflow diagrams, structure charts, state-transition diagrams or data modeling notations. Some offer functionality for designing forms and screen layouts. In addition, they cover "back-end operations" including schema and code generation or project management support.

The most important characteristics of the second generation CASE tools are: automation of the software development methods, providing support to individual analysts to enhance their productivity, and providing partial support to the system development process.

CASE tools of the second generation have many limitations: First, they do not provide support to the whole software development process, such as the lack of support for automatic code generation. Next, they are based on methods that are inappropriate or not well formalised for computer support. Finally, they do not provide support for teamwork.

2.6.3.3 Next Generation CASE Tools

The next generation CASE tools [Lyytinen 1992] adopt and use an integrated set of tools that support the development of all the phases of the software life cycle. In addition,

they support checking, reporting, storage, and sharing of system development descriptions. Moreover, they follow a well-defined object-oriented method that covers all the phases of the software life cycle.

The next generation CASE tools have three important functions:

- Production technology that impacts on the ability to generate planning or design decisions and subsequent products. The next generation CASE tools help to increase productivity, quality, and rationalization.
- Co-ordination technology that enables, facilitates, and expands the interaction between people involved in the analysis, design, and implementation phases.
- Organisational technology that maintains procedures, standards and practices to support the production and coordination technologies.

So far research and the commercial tools focus on the production function.

Important Characteristics of the Next Generation CASE Tools

The next generation CASE tools should:

- support the use of the formal methods that cover analysis, design and implementation paradigms including object oriented analysis, design and programming paradigms;
- support different representation forms such as graphics and text;
- support new representation media such as visualisation and three-dimensional representation;
- have improvements in their flexibility and adaptability. This stresses the need for better formal representation of methods and their use of linkages between methods and better integration of methods into the larger system development environment;

- have improvements in the management of different types of representation and their transformations including facilities for code generation, method transformations, and documentation;
- apply better features to manage design objects in the environment that would allow for reuse;
- support multi-user functionality by including features for identifying, notifying, and supporting several simultaneous users in the environment;
- support automatic and intelligent interactions between different users;
- have improvements in usability by using advanced graphical interfaces that provide access to all necessary tools in the system development environment.

2.6.4 Philosophies for Supporting the CASE Tools Methodology

It is agreed that the next generation CASE tools should follow a well-defined method. However, there are different philosophies [Vessey 1992] in the way that the CASE tool should support method implementation.

2.6.4.1 Restrictive Philosophy

A restrictive CASE tool forces the user to use it in a normative way, such tools restrict the decisions of the user at any point during the system development. For example [Vessey 1992] finds that unconstrained user actions lead to poor performance and that: When a person is to be creative with no direction or constraints whatsoever, the result is often a sense of powerlessness or even complete paralysis of the imagination. Constraints provide the security net that enables people to take imaginative leaps.

2.6.4.2 Guided Philosophy

A guided CASE tool encourages, but does not force, the user to use it in a normative way, such CASE tools guide its users to make decisions by assisting them in choosing and using the support of the methodology. The developmental literature supports the use of guided methodology companion. Moreover, research in computer-assisted learning reveals that learner-controlled systems are more effective than program-controlled systems in learning especially in the following situations: higher-order creative tasks, and subjects with prior knowledge. In addition, the research advice is that the learner control should be accompanied with advice-giving system to help students in decision-making [Gay 1986, Patrick 1983, Steinberg 1985, Taylor 1987].

2.6.4.3 Flexible Philosophy

A flexible CASE tool gives the user the complete freedom in its use. The user can choose his own process, such as top-down or bottom-up processes

2.7 Some Examples of Object-Oriented CASE Tools

2.7.1 The ExSpect Integrated Tool

The ExSpect [Broek 1992] framework, developed at Eindhoven University of Technology at the Netherlands, consists of a mathematical model with an associated language, an object-oriented data model, and a CASE tool. It is intended for the development of systems in which the time and physical location of its components are very important issues, such as in distributed information systems.

ExSpect

ExSpect (EXecutable SPECification Tool) is a model based on a typed functional language for describing the state space and the transformations of a system. It is intended for describing the structural concepts of the applications.

The SimCon Model

The SimCon (Simple Integrated Model for Complex Objects Networks) Model is an object model and is associated with a manipulation language called SimCon Algebra (SCA). Both the SimCon model and the language are integrated within the EXSpect framework.

The SimCon model consists of objects, attributes, and relationships. In addition, it has container objects where the distribution and timing aspects are treated.

2.7.1.1 The ExSpect CASE Tool Set

The tool set consists of a shell from which different tools can be accessed. The tools are a graphical editor, a type checker for checking the consistency, an interpreter for executing a specification, and some supporting tools. The tool set is implemented using C language and runs under UNIX. The user interface of the tool set based on one interaction style. The commands are given using a mouse and a selection is made from a pop-up menu.

The Graphical Editor I

The graphical editor is used for creating and manipulating Petri Nets. A Petri Net is a biparite network of transitions and places. The transitions communicate with each other by sending tokens through the places.

The graphical editor has a standard mouse driven interface. It can create and edit the Petri Nets components such as nets, transitions, and places.

The Type Checker

The type checker receives its input, which is a source file that contains definitions. It then checks the type correctness of the definitions in the input file and produces an object file.

The Interpreter

The interpreter simulates a specification. It has an asynchronous user interface, which allows the users to interact with the simulation, while it is running.

The Graphical Editor II

This graphical editor has the ability to create and edit the schemas of the SimCon model. The editor creates simple and container types, defines references to connect the types, provides inheritance, and has the ability to define constraints. It is windows-based system and the interaction is made with a mouse in a menu driven environment.

2.7.1.2 The Evaluation of the ExSpect CASE Tool

Although the ExSpect CASE tool supported object-oriented techniques and captured the objects through its SimCon Model and behaviour of the system by using a Coloured Petri Net-like model, it made no clear distinction between the phases of the software life cycle.

Since the ExSpect CASE tool provides the developer with additional facilities to support the development process such as the ability to display multiple views of models, a type checker to check the types of the objects in the models, and an interpreter, which

simulates the interaction of the objects in the models, it can be considered as good example of an early attempt to implement the integrated CASE tool philosophy.

2.7.2 ROME

ROME [Barclay 1997] is an object-oriented CASE tool. It is developed by the authors of the book "Object-Oriented Design with C++" to support and automate the method described in their book. It is meant to aid the learning of object-oriented software engineering.

ROME supports the development of Class Diagrams. It generates code in the LOOM and C++ programming languages. In addition, it performs reverse engineering. A change in the LOOM script is reflected in the Class Diagrams.

2.7.2.1 An Evaluation of ROME

Although ROME successfully supports some of the goals of teaching object-oriented software engineering such as allowing the students to experience the iterative life cycle style and develop object model in the analysis phase, it does not support the development of any model that helps in capturing the behaviour of the system nor supports the design phase of the software life cycle. Thus, it does not allow the learners to practice all phases of the software life cycle.

In addition, the features and facilities of ROME do not support the development of large and complex software.

2.7.3 UML CASE TOOL

UML CASE tool is an object-oriented CASE tool developed by Rational Rose to support the development of object-oriented software using their UML method. The UML CASE tool has the following features:

- The tool applies the rules of the UML method.
- It allows the user to develop most the models of the UML method.
- It helps the user to draw the diagrams.
- It stores the information about the models and their components. For example, it stores all the information about the class diagrams such its contents of classes, attributes, methods, and the relationships.
- It generates reports.
- It supports automatic code generation and reverse engineering.

2.7.3.1 An Evaluation of the UML Tool

Even though the UML tool supports the development life cycle graphically by allowing the users to draw the diagrams of the models, it only provides partial logical support. The UML tool stores the information of the models but does not recognise and make a relation between the models. For example, it does not connect a use-case with its scenario. Linking the information between the models has been one of the important concepts that is implemented by the LCCASE tool.

Unlike the LCCASE tool, which displays the inherited attributes in the child classes, the UML CASE tool does not display them in the Class Diagram. The developer has to open the class dialog in order to view these attributes. The display of all the attributes of a

class in the Class Diagram is more clear and helpful to the developer. In addition, the UML tool allows the child class to inherit the private attributes of the parent class, which is against the object-oriented concepts.

2.8 Teaching Software Engineering

There are many challenges involved in teaching software engineering such as course administration, leading the students to develop real, complex, and successful projects, and updating the teaching material to cover the changes that occur periodically in the field. The following sub-sections discuss the goals, difficulties, and solutions of teaching software engineering.

2.8.1 Goals of a Course on Software Engineering

The main objective of a course in software engineering is to provide knowledge and experience in both the theory and practice of object-oriented software engineering. This objective implies several goals [Grimsdale 1997, Knudsen 1996, Dodani 1992, Grimsdale 1998, Biffl 1997, Concepcion 1998], which are discussed in the following paragraphs.

The students should be able to obtain a basic understanding of some of the current techniques of object-oriented software engineering. In most universities, the theme of a course on software engineering is that an object-oriented design methodology such as the Fusion, OMT, Booch, or the UML methods is presented to be used by the students in developing the models of the method to produce a prototype of an application.

The students should understand, exercise, and develop models in all the phases of the software life cycle, analysis, design, implementation, and testing. Furthermore, Dodani [Dodani 1992] has found that students can experience the maintenance phase by performing maintenance on each other's projects.

In addition, the students need to develop real projects and experience the development of complex systems. There have been several attempts in developing these kinds of projects in software engineering courses. Biffl [Biffl 1997] required the students in his software engineering course to develop a working commercial product for real customers. Concepcion [Concepcion 1998] introduced the implementation of the software project, Algorithma, Algorithm Animation, which animates the effect of each step in an algorithm to the data structure it manipulates. The development of large projects requires the knowledge of different fields such as GUI, data structures, file systems, graphics, and compilers.

Furthermore, the students should work in and co-operate with a team. The team size range from 3 to 4 persons in some courses and even 5 to 7 persons in others. Concepcion [Concepcion 1998] introduced the idea of running a software engineering class like a virtual software company, which is composed of teams and team leaders.

The students should experience the process involved in software development such as managing the time, producing documentation, and effectively communicating with each other via invited meetings, phone, ftp, WWW, news mail, and e-mail.

2.8.2 Tools

Several tools were used to support the learning of the object-oriented techniques and methods. In general, the choice of tools depends on their availability and course budget. In some courses general drawing tools are used to draw the diagrams such as HyperCard combined with other tools for database support such as Oracle [Knudsen 1996] and Lotus. Dodani [Dodani 1992] utilised in his course MacProject II for project planning and tracking, version control software, email for communication, and MsWord for documentation. In other courses CASE tools were used such as the OOA tool for the Macintosh [Knudsen 1996], ROME [Grimsdale 1997], UML Toolkit [Grimsdale 1998].

2.8.3 Difficulties Involved in Teaching Software Engineering Courses

Although the Software Engineering course has usually been successful and students in the previous years have produced work of commendable quality [Knudsen 1996, Grimsdale 1997], most courses suffered from several problems, which include:

- The students were not able to experience complete software engineering development process. They learn and practice the analysis and design but in most cases there is not enough time to complete the implementation of their design.
- It is difficult to practice the iterative approach due to the poor tools available and time limitation.
- The complexity of the software development process itself and the analysis and design being a synthesis process with no unique answers.
- Finding adapted CASE tool to the teaching and learning purposes.

2.8.4 Solutions

To aid the learning process and overcome the problems that occur during the object-oriented software development, well-defined object-oriented software engineering method is needed. This method should highly support the object-oriented structures and the iterative approach such as the Next Generation of the Fusion method combined with a powerful and easy CASE tool support.

2.9 Chapter Summary

This chapter discussed the most important features of software engineering, explained the different phases of the software engineering process including analysis, design, and implementation, and highlighted the problems occurring during the software development life cycle.

The chapter explained object-orientation, its important concepts, and object-oriented software engineering. In addition, the different life cycle styles such as the Waterfall, the Incremental and Iterative, and the Evolutionary were discussed.

Next, some object-oriented methodologies such as the Fusion, Next Generation of the Fusion, OMT, Objectory, Booch, and UML methods were overviewed and evaluated.

Due to the advantages of the Next Generation of the Fusion method and its suitability for the use of the experienced and learners of software engineering, I have chosen it to be automated by the LCCASE tool.

The definition of CASE tools and their types, evolution, and philosophies were presented. Moreover, some existing CASE tools such as ExSpect Integrated Tool, ROME, and UML were discussed.

Finally, the current situation of teaching software engineering, related issues, and problems facing the students were highlighted and a solution was proposed.

3 Learner Centred Computer Aided Software Engineering (LCCASE) TOOL

LCCASE tool is a comprehensive CASE tool that applies object-oriented software engineering principles and integrated technical and management approaches throughout the software development life cycle.

The environment of the LCCASE tool has been designed to support the development scheme of the Next Generation of the Fusion Method. It constructs the models of the method for each development phase of the software life cycle.

While designing the LCCASE tool, I was aiming for a CASE tool that is powerful and supports the development of a successful object-oriented software, and on the other hand, assists the learning process of object-oriented software engineering, is practical and easy to be used by the students, and helps the students in developing software within the short time span of a course.

The LCCASE tool includes many features and characteristics to make it suitable for use by both the experienced and inexperienced software developers. It provides a full and intelligent guidance during the development of a system from a collection of objects and models with well-defined interfaces and links between them.

Moreover, the LCCASE tool can be used for developing both small and large complex applications. While the developer of a small application has the ability to ignore the development of some models in the method, the developer of a large and complex application can benefit from the support of the tool to develop all the models and use its facilities.

The following sections start by discussing the need for the LCCASE tool. Next, an explanation of the philosophy of the LCCASE Tool is given. After that, the main features of the LCCASE Tool are discussed. In addition, the benefits of the LCCASE tool to both the professional and learner users are described. Finally, a summary of the chapter is given.

3.1 The Need for the LCCASE Tool

The process of developing large software is creative and complex in nature, costly, and time consuming. Thus, many large software projects suffer from serious deficiencies, such as being released behind schedule, being above estimated cost, and not supporting the original objectives.

Many software engineering techniques, methodologies, and CASE tools have been produced to attempt to eliminate the problems associated with developing large software projects.

Unfortunately, the support of existing CASE tools for the software development process neither improves the quality of the software nor reduces the time necessary for development.

Current CASE tools suffer from many deficiencies, such as they do not provide support to the complete life cycle of the software development process and they do not adapt to the needs of their users in general and to the needs of the learners in particular. In addition, many CASE tools are erroneous and have no or only partial knowledge of the semantics of the method they implement.

all the models and requirements of the method it supports. It guides the developer through a sequence of models and the user can choose the level of the model.

Large software development process is complex, creative, and has no unique solution. Adopting the guided philosophy in the LCCASE tool encourages and helps the developer to explore different possible answers, which increases the developer creativity. In addition, the guided philosophy motivates the learner by allowing him to try and use his own ideas and be responsible for the decision making.

Boyle [Boyle 1994] stated that Carroll [Carroll 1990] observed that learners tend to skip over important material if it was not significant, in their view, to complete the current task in hand. Thus, the LCCASE tool guidance reminds the developers about the optimum procedure to finish their tasks and focuses on important design concepts.

3.2.2 Interactivity

The developer interacts with the LCCASE tool and constructs the software. Interactivity is an important feature because the reactions of the developer control the behaviour of the tool's environment. Moreover, the interactivity feature helps the developer to apply a step-by-step strategy, discussed in the next sub-section, in solving complex problems and applying their ideas and refining them, which reduces the complexity of the software development process.

During the construction of the models, the developer interactively develops the models using the LCCASE tool guides and support. Once the development of a model is completed, the LCCASE tool provides the developer with feedback regarding the models that should be developed next.

3.2.3 Step by Step Construction

The LCCASE tool allows the user to develop the models and concepts incrementally until a final solution in the form of a Java program is produced with links being made at each step between the developed models and objects.

3.2.4 Intermediate Construction

The LCCASE tool supports the iterative strategy during the construction of the models to help the developer in producing several prototypes of the software. This provides the developer with intermediate steps during the software life cycle, which reduces the software development complexity and risks.

3.2.5 Navigation

The developers can navigate through the LCCASE tool environment very easily through direct and clear paths such as accessing any model from the Model menu located in the main menu.

In addition, the LCCASE tool interface includes a dynamic menu that can be accessed from the Windows menu. The dynamic menu shows the names of the currently active models. The developer can easily view a model by selecting its name from this menu.

Moreover, the developer can view an information dialog about any component in the Directory with double clicks.

3.2.6 Flexibility and Individualisation

The LCCASE tool is a rich adaptable tool. It provides many facilities that individual developers can use to meet their development needs. The developer can use these facilities according to his needs and experience in a development phase.

These facilities include the Development Life Cycle, which is a facility that shows the developer his progress in the development process by displaying the models that have been developed and the models that need to be developed.

Another facility is the Directory. It displays the names of all objects that are created in the project and their relationships with each other.

There are many other facilities that can be adapted to the developer needs, which are discussed in the next chapters.

3.2.7 Ease of Use

A key objective of the LCCASE tool is to retain an overall simplicity and intuitive ease of use. There are several features in particular, which contributed to this aim such as, accessing any facility or command is gained through clicking on an icon, button, or menu item.

The developer can display more than one model at the same time so that the he can benefit from the relevant information in different models.

The LCCASE tool allows fast and easy construction of diagrams. The developer can draw the objects in the models by simply accessing an icon and specifying the location of the object in the diagram with another mouse click. The developer can move the objects by dragging them and delete them by clicking on the delete button.

3.2.8 The Next Generation of the Fusion Method

The rationale behind using the Next Generation of the Fusion method to automate the LCCASE tool is its concise and comprehensive set of well-defined notation, which is the standard UML notation. It covers all the software development life cycle, supports reuse and reengineering. In addition, the Next Generation of the Fusion method has many consistency checks and measures that are performed at the end of each phase. The Next Generation of the Fusion method is described and evaluated in detail in chapter 2. The use of such clear and powerful method increases the effectiveness and usability of the tool.

Moreover, the Next Generation of the Fusion method has been used to develop the LCCASE tool.

3.3 The LCCASE TOOL Main Features

The LCCASE tool has many features that are important to be included in a useful and reliable CASE tool in general and in a CASE tool that is adapted to the needs of the learners in particular. These features include providing intelligent guidance, navigational assistance, graphical assistance, and code generation facility, Data Dictionary, report generation facility, and support to the maintenance phase. The nature of these features and how they are included in the tool support are discussed below:

3.3.1 Intelligent Guidance

The LCCASE tool provides the developer with intelligent guidance through all the development process of the models in different aspects.

When creating any model the tool has the intelligence to realise the connection between the new model and the models previously created and provide this information to the user, as required. This feature decreases the complexity of the software development process. It draws the attention to the important concepts, allowing the developer to link between different concepts during the software development process, and helps him to interpret this relevant information correctly.

The LCCASE tool has the ability to understand the semantics of the models and perform consistency checks. Moreover, it has the ability to detect and report errors and any conflicts within and between the models. For example, the tool will check that the system classes and relationships have unique names, and all attributes and methods within the same class are unique.

The LCCASE tool suggests the models that should be created by the developer according to the information obtained from the models that have already been developed. For example after the developer has created a Use-Case Model, the tool will suggest the required Scenarios. It will then suggest the Contract models from the information in the Scenarios. This feature draws the developer attention to what should be done next, decreasing the complexity of the software development without requiring action from him.

The LCCASE tool advises the user to reuse items or diagrams. Moreover, it has the ability to reuse items or diagrams. This feature speeds up the process of software development in general and allows the learner to finish the development process within a limited time period. In addition, the learner will gain good software development habits since reusing components is a very useful software engineering technique.

The LCCASE tool provides a facility, which is implemented in the Directory, to track the requirements specified in the Use-Case Model of the Next Generation of the Fusion method, and map these requirements in other models. Providing the developer with feedback about the effectiveness of his approach, allows him to verify that the requirements are correctly implemented.

The LCCASE tool has the ability to analyse the models. It informs the user about insufficient information in the models, such as creating an object without giving it a name. In addition, the tool warns the user about the contradictions in the models. This feature allows the learners of software development to profit from their mistakes.

3.3.2 Navigational Assistance

The LCCASE tool contains a Directory tool that works as a browser, which allows the developers to navigate through the models and the use-cases and the classes hierarchies. It displays the names of all the classes and their relationships, the use-cases and their scenarios, system operations, and collaboration diagrams.

This feature provides the developer with the means to view the relations and the dependency between the project components. In addition, it provides him with the ability to trace through all the components to the requirement stage, which assists the developer in his decision making process about the new components that should be developed and the changes needed to be made to any component. Moreover, it improves the pedagogical efficiency of the tool, because the demonstration of the relations between the objects is essential for understanding the software development process.

The LCCASE tool supports multiple views of the models, where each view focuses on a model in the application. Moreover, the tool suggests the models that should be viewed to help develop the current model. This feature helps the developer in general and the learner in particular, to link the information at different models and reuse the concepts and structures.

3.3.3 Graphical Assistance

The LCCASE tool provides a graphical assistance, whereby the developer can draw and manipulate diagrams of different models in the Next Generation of the Fusion method. The diagrams include boxes, lines, text, and any symbol used in that method notation. In addition, the diagrams are sufficiently complex to convey all the information within the method, but simple enough to be easily drawn and understood.

The drawing of diagrams using this tool stimulates the developer. The tool provides excellent mechanisms for manipulating the diagrams and assists the developer creating flawless diagrams. Moreover, the developer can easily load, edit, save and print diagrams.

The provision of a tool that allows the developer and the learner to draw the method diagrams very easily, allowing them to concentrate on the actual development process.

3.3.4 Code Generation Facility

The design of the LCCASE tool has the ability to support code generation. Although this has not been implemented, it can be developed in the future. The LCCASE tool can generate Java source code after the creation of the design class models.

The code generation feature would help the learner to produce a real software application at the end of the development process. The fact that the learner knows that he will be able to produce a real software product motivates him and increases his interest in the topic. Moreover, it provides him with the opportunity of testing and validating the software, which are very important phases of the software life cycle.

3.3.5 The Data Dictionary

The LCCASE tool automatically creates and maintains a Data Dictionary (DD) that contains information about all the models in a project. If the user changes information in any model, the tool changes the corresponding information in the DD. The tool uses the information in the DD to check the consistency between the models. The DD helps the developer clarify concepts and ideas during the whole of the development process. Moreover, it is very important for the software maintenance and extension phase.

3.3.6 Report Generation Facility

The LCCASE tool generates reports on classes and models as required. This feature helps the learners to connect the software development concepts and ideas and serves as a communication mechanism between the learners themselves and the learners and their teacher. In addition, it provides good communication between the members of a professional development team.

3.3.7 Support the Maintenance Phase

The system is altered according to the changes in the requirements and needs of the user. The Use-Case Models, which represent the needs of the users, are linked to most

other models, such as, the Scenarios, Contracts, and Collaboration Diagrams. If changes occur in the user needs that affect the Use-Case Model, the system gives a list of all the changes that need to be made to other connected models. The linkage between the models is discusses in chapter 4. In addition, the Data Dictionary contains a full description of all the information about the system development, which provides help to the system maintainer who might lack knowledge of the system.

3.4 The Benefits of the LCCASE Tool

3.4.1 The Benefits of the LCCASE Tool to the Professional Developer

The rich features of the LCCASE tool, which are discussed in the previous section, make it a preferable choice for the professional developers.

The LCCASE tool automates a powerful method, which is the Next Generation of the Fusion method. The developers can easily develop the models of the software life cycle in an integrated environment. In addition, the linkage of information supported by the LCCASE tool helps the developers to connect and remember the concepts in different models. Moreover, the LCCASE tool allows the developer to navigate between the models and access their information easily and produce an updated documentation about the project.

3.4.2 The Benefits of LCCASE Tool for Learners of a Course on SE

The LCCASE tool provides the learner with the opportunity to experience the complete software engineering development process by building software. Ram [Ram 1994] stated

that people could learn through many different ways, but prefer to gain knowledge and skills through experiential opportunities.

Since the LCCASE tool supports the incremental and iterative approach for software development, it allows the learner to practice the iterative and incremental development approach. The LCCASE tool helps the learner to develop the software system incrementally until the final phase in the form of a Java program is produced and at each phase links are made between the models. In addition, the learner always can go back and add to or change the system models.

Moreover, the LCCASE tool, through its learner centred approach, allows the learner to concentrate on the actual development task rather than wasting his effort and creativity on trivial tasks, it helps the learners to implement the object-oriented software engineering method correctly, and improves the learner productivity and creativity.

3.5 Chapter Summary

Current CASE tools suffer from many deficiencies that reduce their efficiency and make them below the expectation of their users. In addition, they lack the provision of the adequate support to the inexperienced users.

The LCCASE tool provides its users with intelligent guidance throughout the software development process, which reduces the complexity of the software development and draws the attention of the developers to the important issues. In addition, the LCCASE tool allows the developer to control the development environment, apply step-by-step and intermediate construction strategies, and navigate through clear and direct paths. The

LCCASE tool is a flexible tool that can adapt to the needs of its individual users. Moreover, it retains simplicity and ease of use through all its operations.

The LCCASE tool provides intelligent guidance, navigational assistance, graphical assistance, code generation facility, data dictionary, report generation facility, and a strong support to the maintenance phase.

Finally, the benefits of the LCCASE tool to both the experienced and inexperienced users have been discussed. While the LCCASE tool supports the professional users by understanding and applying the rules of a powerful method and providing a complete, effective, and easy to use CASE tool, it allows the learners to experience the software engineering techniques, learn and practice the iterative development approach, concentrate on the actual development task, implement an object-oriented software engineering method, and improves learners productivity and creativity.

4 The Design of the LCCASE Tool

The purpose of the design is to create an architecture for the implementation and to specify control policies that co-ordinate the functions of the different elements in the system. The architecture and control of the LCCASE tool have a large impact on its quality and completeness.

There are several activities that need to be accomplished during the design. The architectural design involves identifying the subsystems of the system and their relationships, services, and constraints. The interface design involves finding the interface of each subsystem with other subsystems and how they communicate together. The component design establishes the components of each subsystem, their services, and how they communicate with each other.

This chapter starts by describing the architecture of the LCCASE tool. It then discusses the control strategy between the subsystems. Finally, the chapter explains the environment of the LCCASE tool.

4.1 The Architecture of the LCCASE Tool

While the features of the LCCASE tool, discussed in chapter 3, guided the process of identifying the subsystems of the LCCASE tool, the philosophy and objectives of the LCCASE tool have great impact on deciding the general strategy within the subsystems, their constraints and relationships with other subsystems. These features, objectives, and philosophy have been implemented as co-operating subsystems.

The architecture of the LCCASE tool (Figure 4-1) consists of two layers. The first layer is partitioned into six principal and independent subsystems, which are the Model, Directory, Development Life Cycle, Data Dictionary, Database, and User Interface. Each subsystem is responsible for a specific task. The second layer contains the Message Services Subsystem that allows the subsystems of the first layer to communicate with each other.

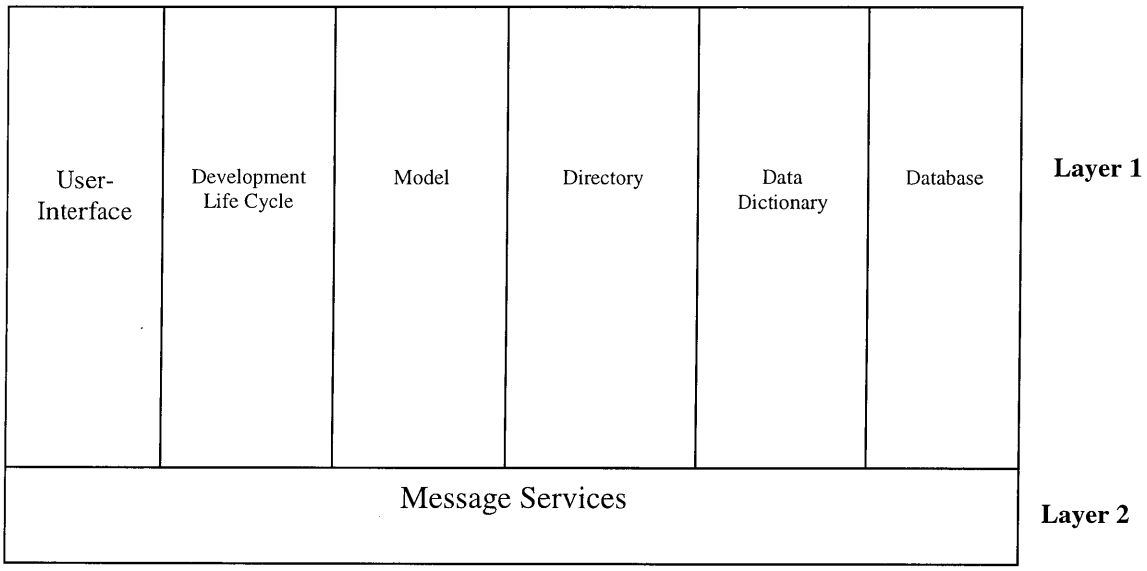


Figure 4-1 The Architecture of the LCCASE Tool

The LCCASE tool supports the development of the models in all the phases of the life cycle of the *Next Generation of the Fusion* method. As the developer constructs the models, the LCCASE tool assists him in drawing the diagrams, updating the information of the models, and providing him with intelligent guidance. These activities are related to the model development and integrated with each other, which raise the need for the Model subsystem that supports developing the models and maintains the facilities of the LCCASE tool during the development process of a model.

Due to the complexity involved in developing large software systems, especially if there are numerous models to be developed, the developer might forget the contents of those models. For instance, the developer might attempt to create a class that already exists or create several classes with relatively similar attributes and methods. Therefore, the developer needs a tool that shows the objects created during the software life cycle and provides him with navigational capability. The Directory Subsystem is included in the LCCASE tool to support this purpose.

The documentation of the software life cycle is very useful during the analysis, design, and implementation phases of the software life cycle, particularly through the maintenance phase since it describes the components of the software life cycle. However, the production of documentation is an activity that is disliked by software engineers [Welsch 1997]. Moreover, the iterative style of the software development life cycle requires frequent update of the documentation to keep it consistent with models and components of the software life cycle. Since the documentation involves many activities and has bounded domain, the Data Dictionary Subsystem was necessary to support these issues.

The subsystems of the LCCASE tool communicate and co-operate together in performing their tasks. In addition, much of the data is shared among these subsystems. This requires the design of complicated management rules to ensure that all subsystems perform their work and all shared information is consistent. To reduce the management complexity, facilitate the communication between the subsystems, ensure the consistency and correctness of the shared information, and support tool extensibility, a common

repository, the Database Subsystem, is needed to store and manage all the information in the project.

One fundamental objective of the LCCASE tool is to assist the learners of software engineering in a learner centred environment. The Development Life Cycle Subsystem has been specially designed to benefit the user who is learning object-oriented software engineering and inexperienced in developing object-oriented software.

Another important objective of the LCCASE tool is to develop a fully interactive and user-friendly CASE tool. Thus, the LCCASE tool needs to have an easy to use and efficient user interface. The User Interface Subsystem of the LCCASE tool is based on rich graphical contexts to support and guide the developer through the software development life cycle. It is an interactive user interface, which provides the developer with prompt and correct response upon each act or enquiry. In addition, it is easy to learn.

While designing the architecture of the LCCASE tool, I specified several design goals:

- The LCCASE tool should maintain proper links between its subsystems that do not effect the integrity of individual subsystem or increase the dependency between them. This strategy is implemented by the use of a Database Subsystem, which holds all shared data and can be accessed by all subsystems, thus there is no need to transmit data explicitly from one subsystem to another.
- The LCCASE tool use and functionality should be adapted to the users needs.
- The LCCASE tool provides proper guidance and support so that the users with insufficient knowledge are able to perform their tasks.

The following sub-sections explain each subsystem in detail.

4.1.1 Model Subsystem

Responsibilities

The Model Subsystem, the largest and most important subsystem in the LCCASE tool, is responsible for many services provided by the tool. These services include developing all the models in all the phases of the life cycle, namely the Use-Case Model, Class Diagram, Scenario, and Contract in the analysis phase; and the models of the design phase, which are Collaboration Diagram, and the Design Class Diagrams. The LCCASE too is designed to be able to support code generation in the future. It supports the iterative life cycle style.

The Model Subsystem develops all the models in one formal way. This makes the tool easy to use and once the user learned to develop one type of model, all other models can be developed in the same way.

In addition, the Model Subsystem is responsible for generating guidance to the users during the development of the models. Moreover, the guidance is adapted to the needs of the users. The Model Subsystem provides guidance and support to the user in many ways:

- Finds, for each model, information that can be reused from other models and where and how it should be reused. This feature is especially important for the inexperienced user because it shows him the links between the models that he might not know and it saves him time and effort of developing each model from scratch.
- Checks the correctness and the consistency of the information shared between the models.

- Changes the mouse shape according to a requested function, which reminds the user of the command he is executing.
- Displays an information dialog about any object in the Work Space when it is double clicked. This reminds the user about the detailed information while developing a complex system.
- Checks the correctness of the sequence of commands during the development of the models.

The Model Subsystem maintains links between the models giving many benefits such as supporting reuse and allowing the subsystem to perform the consistency checks. These links are discussed in section 3.

Moreover, the Model Subsystem is responsible for drawing and manipulating the graphical diagrams to simplify the task of creating the models and reduce their production time. Each model has its own graphical notation. However, the Model Subsystem provides the same drawing procedure for all the objects in the models.

Constraints

The Model Subsystem operates when the developer requests the creation or editing of a model. Before creating a Use-Case Model or a Class Diagram the developer should provide a name for those models. However, the Model Subsystem suggests names for other models such as Scenarios, Contracts, and Collaboration Diagrams. These names are obtained from the related Use-Case Models and Class Diagrams.

Interface Design

The Model Subsystem communicates with the Database Subsystem via the message services that are provided by the tool. The message services allow communication between the two models without the need for each system to know about the structure and the internal operations of the others. The Model Subsystem stores, updates, or retrieves information from the Database Subsystem.

Component Design

The structure of the Model Subsystem (Figure 4-2) consists of two layers where each layer contains several components.

The first layer contains ObjectInfo and Event and Message Handler Components.

The main responsibility of the ObjectInfo Component is to maintain references for objects, which represent the elements in a model. The elements of a model are specified by the Next Generation of the Fusion method, such as the Use-Case Model, which contains use-cases, actors, and references. Each object of an element contains information about the attributes, operations, and drawing procedure of the element.

The Event and Message Handler Component triggers the externally generated events that affect the Model Subsystem and perform the required actions.

The second layer contains partitions that represent the moulds of the models constructed by the Model Subsystem. The moulds include the Use-Case Model, Class Diagram, Scenario, Contract, Collaboration Diagram, Design Class Diagram, and Code. Each mould automates its corresponding model in the Next Generation of the Fusion Method. They communicate with each other via the Event and Message Handler.

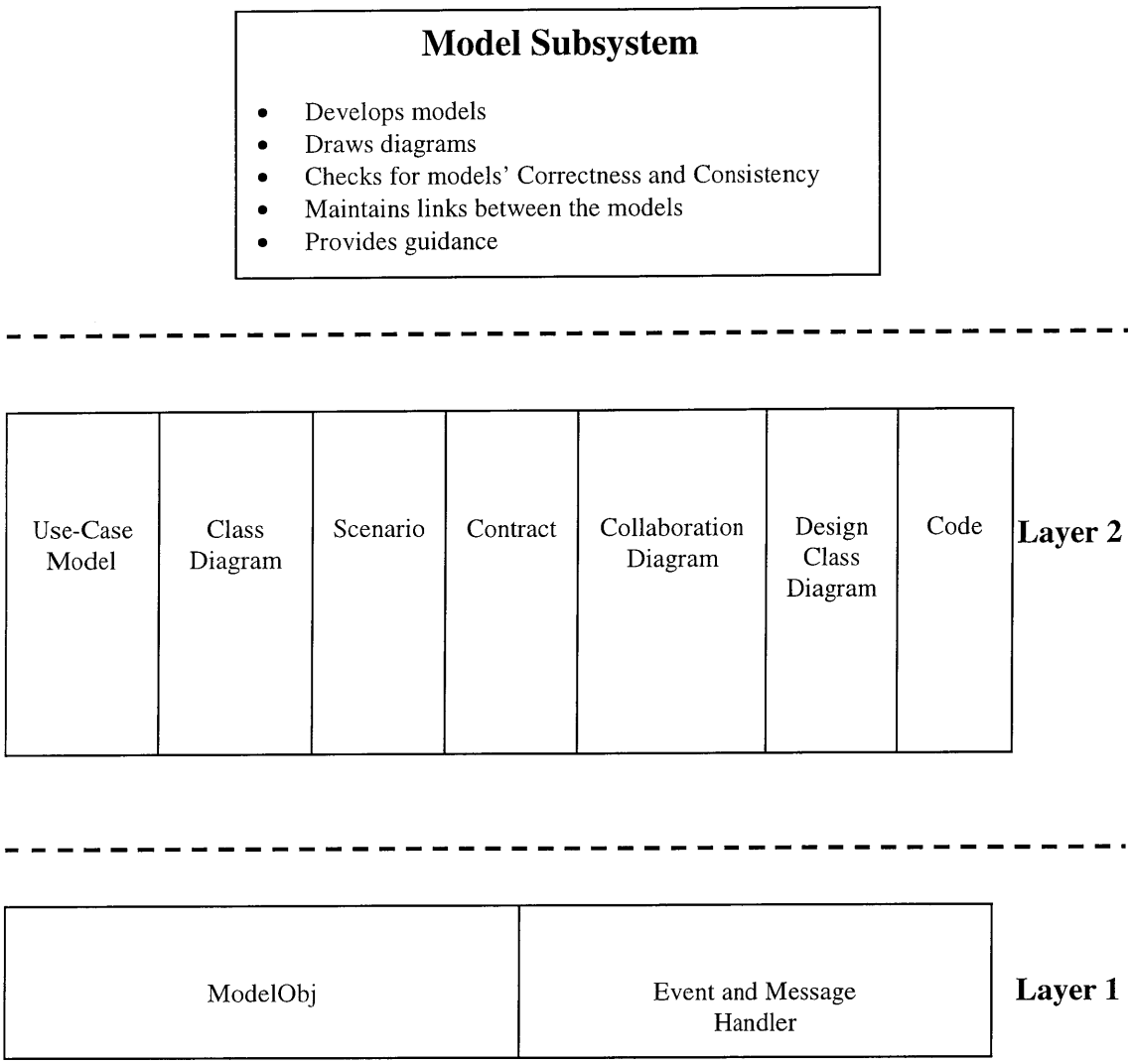


Figure 4-2 The Structure of the Model Subsystem

4.1.2 Directory Subsystem

Responsibilities

The Directory Subsystem displays a Directory window. The Directory window has two views, the Use-Case and the Class views and the developer can switch between these two

views. The reasons behind having the two views are that both the use-cases and classes are very important objects in the project, many other subsystems and models depend on them or have relations with them, and from the use-cases and the classes the developer can traverse all the subsystems in the project.

The Directory Subsystem provides the developer with good navigational capabilities whereby the developer is able to browse all the objects and the models in the project.

Constraints

The Directory Subsystem displays a directory window throughout the project development period.

Interface Design

The Directory Subsystem communicates with the Database Subsystem. It retrieves the information needed about the objects that it displays. If any changes occurred to the objects from the Directory, the Database Subsystem updates these changes and thus the models containing those objects are affected as well.

Component Design

Care has been taken in designing the Directory Subsystem so that important information can easily be perceived. The structure of the Directory Subsystem (Figure 4-3) consists of two layers. The first layer contains the ObjectInfo and Event and Message Handler Components.

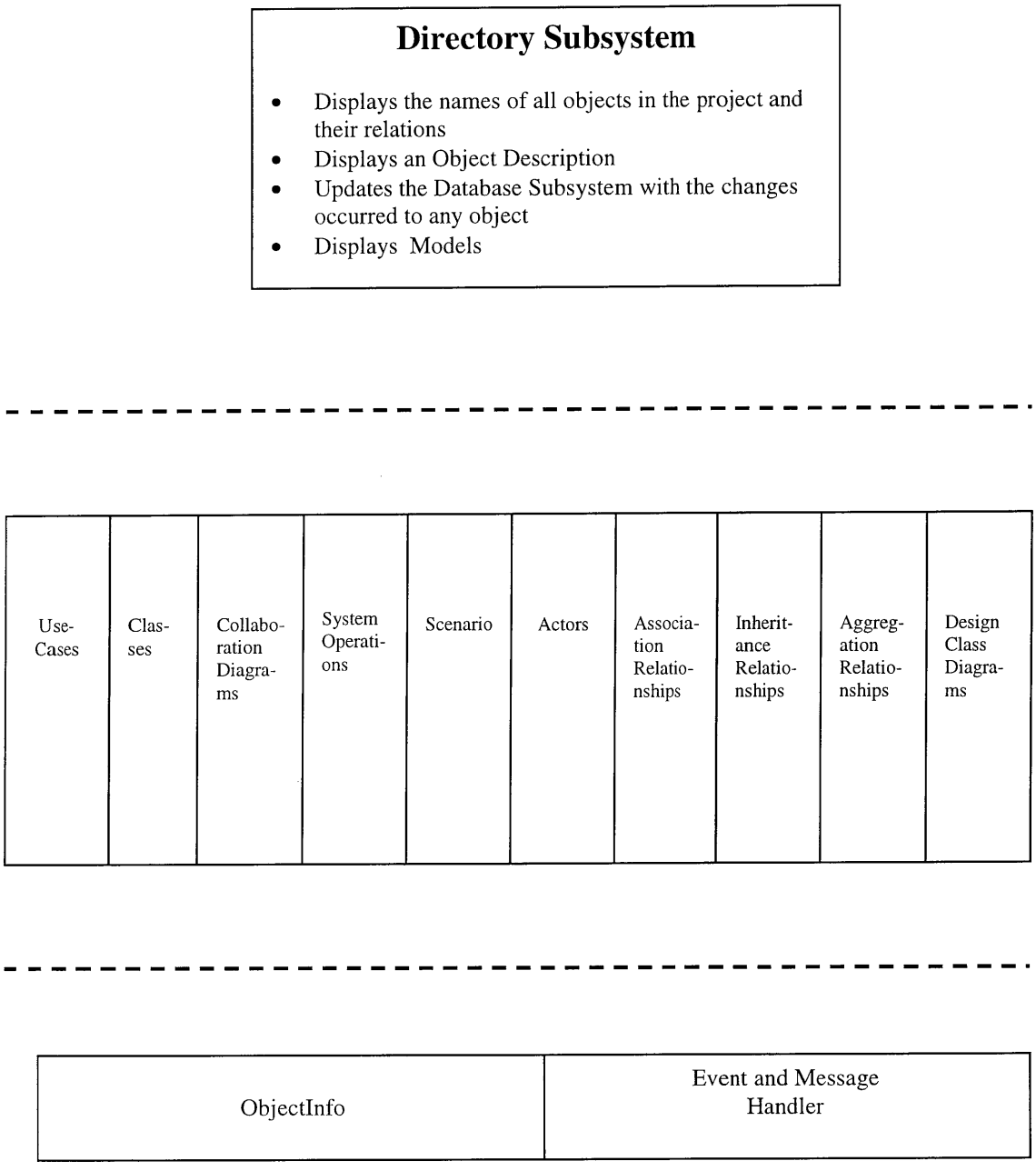


Figure 4-3 The Structure of the Directory Subsystem

The main responsibility of the ObjectInfo Component is to maintain references to the objects so that the Directory Subsystem gets the required information.

The Event and Message Handler Component triggers the externally generated events that effect the Directory Subsystem and perform the required actions. The external events include selecting or double clicking an item from the components in the second layer. In addition, the Event and Message Handler Component controls the messages that are sent from one component to another.

The second layer consists of several components, which include the Use-Cases, Actors, Scenario, System Operations, Collaboration Diagrams, Classes, Associations, Inheritances, and Aggregations.

The Use-Cases component displays the names of the use-cases that are created in the project giving the user a clear view of all the operations performed in the project. The Use-Cases component controls the operations performed by the user on the displayed use-cases' names. Once the user selects a use-case, the Event and Message Handler component sends a message to the:

- Actors component to display the names of the actors who have references with the selected use-case;
- Scenario Component to allow the developer to display the Scenario of the selected use-case if a Scenario has been created for the use-case;
- System Operation to display the names of the system operations in the Scenario.

The Actors component displays the names of the actors who have references to a selected use-case. Knowing the actors of a use-case has an important effect in developing other models in the system such as the Scenario and the Class Diagram.

The Scenario component enables a Scenario Button, which displays the Scenario of a selected use-case once it is clicked. This allows the user to view easily the Scenario of the use-case. This feature is useful when the developer needs to know, remember, or edit operations to perform a specific use-case and when the developer needs to know the sequence of those operations.

The Collaboration Diagrams component displays the names of all Collaboration Diagrams that contain a selected system operation.

The operations of the Use-Cases, Actors, Scenario, System Operations, and Collaboration Diagrams components allow the developer to relate the information in the project.

In addition, these components display the information dialog about their elements. This provides the developer with an alternative way to access the information about the use-case, actor, or system operation in the project. Moreover, the developer can modify the information in the information dialog and the LCCASE tool reflects those changes in the corresponding diagrams.

Viewing the contract is important during the software development especially when the developer needs to know the characteristics of the system operation to develop other models such as the Collaboration Diagram of the system operation.

The Classes component displays the names of the classes that are created in the project. The Associations component displays the association relationships of a selected class. The Inheritances component displays the inheritance relationships of a selected class. The Aggregations component displays the aggregation relationships of a selected class.

All the above components display an information dialog of a selected item once it is double clicked. This allows the user to have a clear view of the class attributes, methods, and their relationships and find the class hierarchies. In addition, the developer can modify the information in the dialog, then the LCCASE tool reflects the changes in the model that contains the object. This provides the user with an alternative way to access the information about a class or a relationship.

4.1.3 Development Life Cycle Subsystem

Responsibilities

The Development Life Cycle Subsystem (DLCSub) maintains and displays the Development Life Cycle window and controls all its operations.

The purpose of the Development Life Cycle is to show the learner of object-oriented software engineering what models he should create and the progress of the development process.

The Development Life Cycle does not control or restrict the student but it guides him to decide which models need to be developed as the student has the freedom to develop any model at any time.

Constraints

The user can activate or deactivate the DLCSub at any time during the software development process.

Interface Design

The DLCSub communicates with the Database Subsystem. It retrieves the information needed to display the names of the models that have been created and make the decision about which models should be developed.

Component Design

The structure of the DLCSub (Figure 4-4) consists of two layers. The first layer contains ObjectInfo and Event and Message Handler components.

The ObjectInfo component retrieves all the information needed by the DLCSub from the Database Subsystem to perform its task.

The Event and Message Handler component responds to the external events that are generated to request from the DLCSub to perform specific task. The Event and Message Handler analyses and then executes those requests.

The second layer of the DLCSub contains the LifeCycle component, which displays Life Cycle Area. The Life Cycle Area contains buttons, which represent the models in the life cycle. These buttons include Use-Case Model, Class Diagram, Scenario, Contract, Collaboration Diagram, Design Class Diagram, and Code Generation Buttons.

The buttons can either be green or red. Green buttons indicate that the user can develop models of that type. Red buttons indicate that the time is not appropriate for developing models of that type. The selection of the red and green colours for the buttons gives the students the feeling of stop and go.

When the user clicks on any button, the Event and Message handler component analyses this request and sends a message to the LifeCycle component to perform the

required task. The LifeCycle component displays a dialog containing a list of the models that are developed and a list of the models that need to be developed.

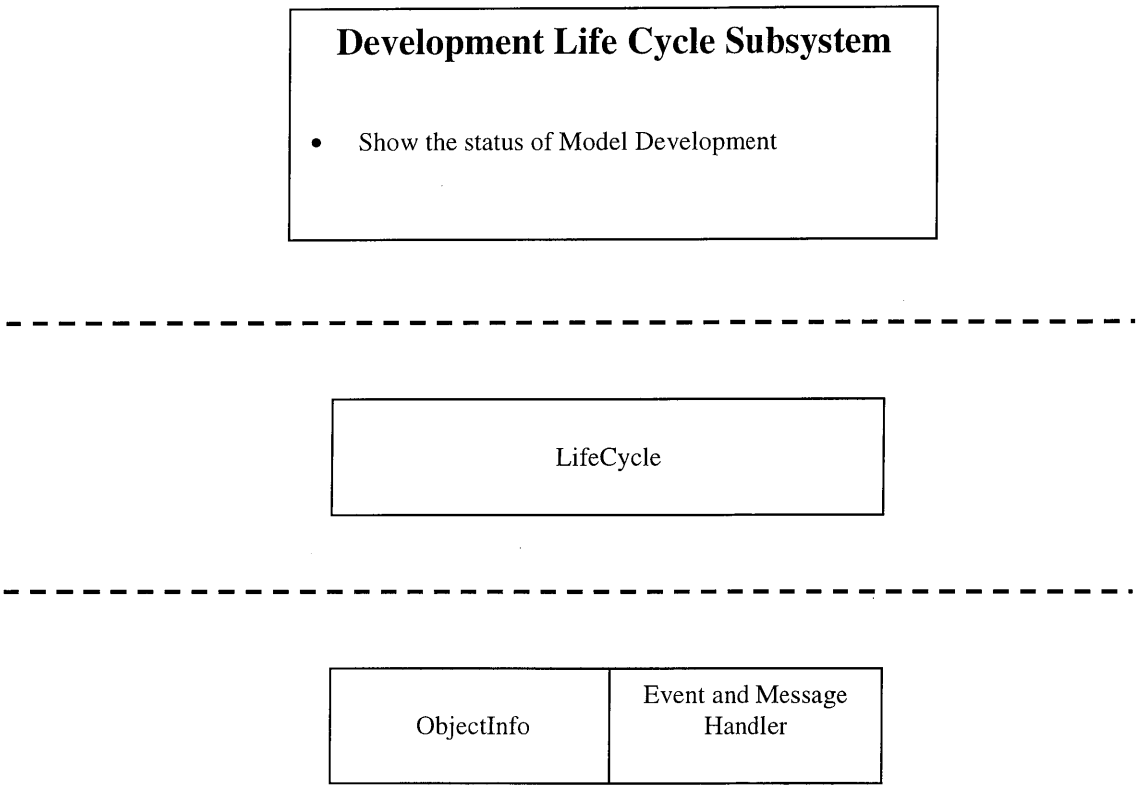


Figure 4-4 The Structure of the Development Life Cycle Subsystem

4.1.4 Data Dictionary Subsystem

Responsibilities

The Data Dictionary Subsystem of the LCCASE tool generates and updates the documentation automatically. Each time the developer creates an object during the software life cycle, the Data Dictionary Subsystem adds the object name, kind, and description to the data dictionary list. In addition, the developer can add more information to the object information. Moreover, the Data Dictionary updates this information whenever a change is made to the object and makes a consistency checks between the information added by the developer and the information in the project.

Furthermore, the Data Dictionary Subsystem provides the developer with a search engine to search in the data dictionary for a specific name or kind.

Constraints

There are no constraints for using the Data Dictionary Subsystem. The developer displays the Data Dictionary at any time during the software development process.

Interface Design

The Data Dictionary Subsystem communicates with the Database Subsystem. It retrieves the information of all the objects created in the project and updates the information stored in the Database Subsystem with any changes.

Component Design

The structure of the Data Dictionary Subsystem (Figure 4-5) consists of two layers. The first layer contains ObjectInfo and Event and Message Handler components. The responsibility of the objectInfo and Event and Message Handler components is the same as their peers in other subsystems.

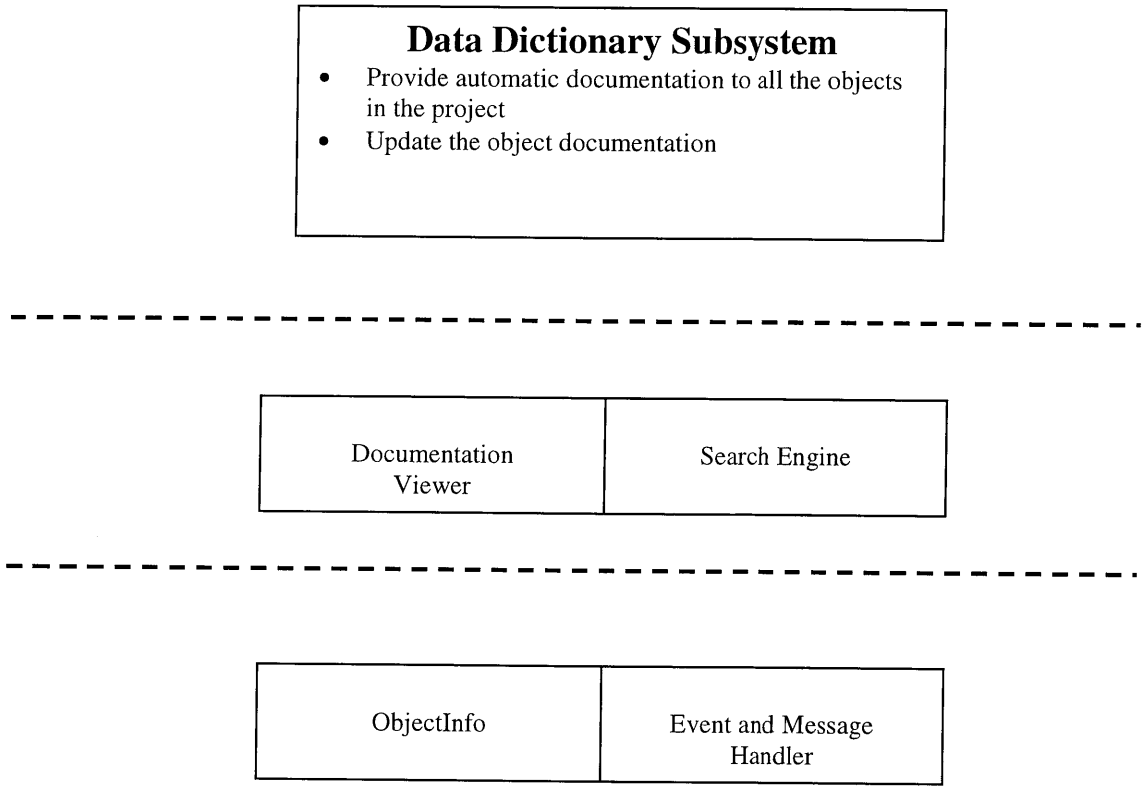


Figure 4-5 The Structure of the Data Dictionary Subsystem

The second layer contains Documentation Viewer and Search Engine components. The Documentation Viewer displays the information about all objects in the project in a

tabular form. The table consists of three columns, which are Name, Kind, and Description. The objects are arranged according to their name in alphabetical order.

The Search Engine component provides the developer with a search facility within the Data Dictionary. The developer searches for a particular object according to its name or group of items according to their kind.

4.1.4.1 Database Subsystem

Responsibilities

The Database Subsystem (Figure 4-6) provides support for creating, reading, updating, and deleting objects. It defines and manages the relationships between the stored objects where each object has unique identifier. In addition, the Database Subsystem provides support for multiple objects of a class and groups them into composite structures.

Interface Design

The Database Subsystem communicates with the other subsystems of the LCCASE tool, either by getting information from other subsystems or updating the information stored in them.

Component Design

The Database Subsystem consists of two layers. The first layer represents the objects of the project. The second layer represents the data structures of the objects.

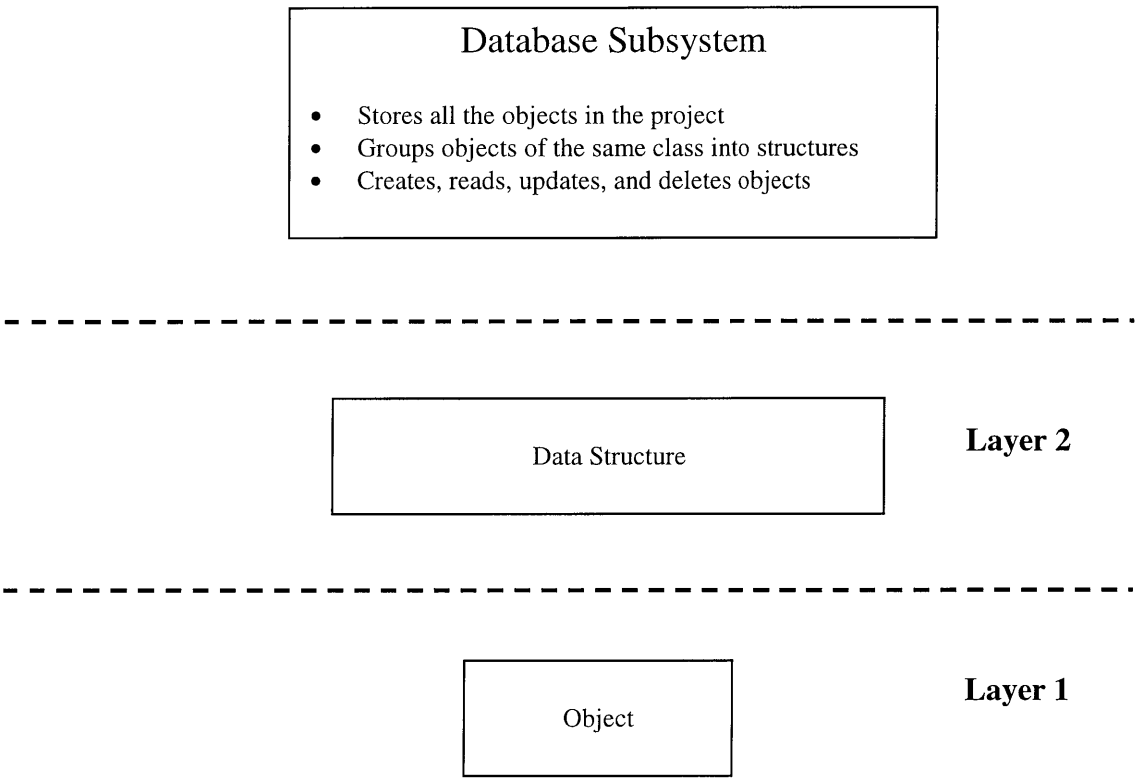


Figure 4-6 The Structure of the Database Subsystem

4.1.4.2 User Interface Subsystem

Responsibilities

The User Interface Subsystem of the LCCASE tool is responsible for producing a graphical user interface environment. This graphical user interface uses visual objects to represent the tool entities, such as windows, buttons, menus, and dialogs.

The ability to display multiple windows makes possible the display of different information simultaneously on the user's screen.

The use of the menus facilitates the use of the commands that the user selects a command from a menu rather than typing it in a command language.

The drawing of diagrams can be mixed with text, which supports the development of the models.

The user interface of the LCCASE tool has standard mouse-driven interface. A mouse is used for selecting items from a menu, pressing a button, or indicating objects of interest in a window.

The commands in the Model Subsystem are executed via buttons to create any object in a model, cancel command to cancel any command that is in execution state, and delete command to delete any component in the model.

In addition, each model that is developed by the Model Subsystem has a Work Space, in which the user creates, draws, or deletes the objects of the model. The Work Space has many facilities to increase the creativity of the user, allowing him to be in control, and making the tool easy to use, improving the quality of his work, and allowing him to develop the models more rapidly. These facilities include:

- the Work Space has vertical and horizontal scrollbars, which allows the user to scroll the Work Space;
- the ability to move objects after creation.

Interface Design

The User Interface Subsystem communicates with other subsystems such as Model, Development Life Cycle, Data Dictionary, and Directory Subsystems.

Component Design

The user interface for each subsystem of the LCCASE tool is designed to facilitate and support the function of that subsystem. The User Interface Subsystem consists of two layers. The first layer contains the main GUI components, which are window, button, list, dialog, menu, icon, graphical presentation, and text presentation. The second layer consists of the components that are responsible for producing the graphical interface for other subsystems that interact with the user. These include Model, Directory, Development Life Cycle, and Data Dictionary. Figure 4-7 shows the structure of the User Interface Subsystem.

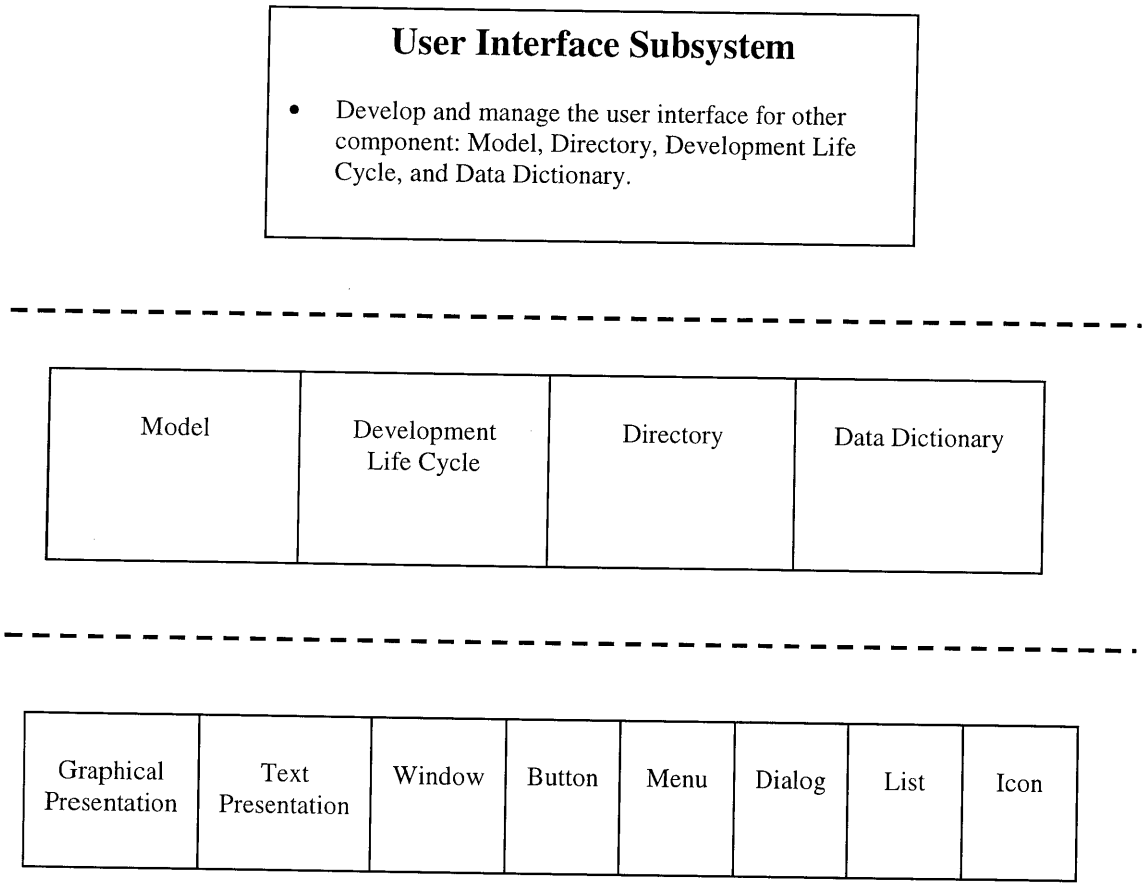


Figure 4-7 The Structure of the User Interface Subsystem

Design Goals

The goals of the design of the components of the GUI and their responsibilities include interface consistency. The system commands, menus, graphical representations, etc. have the same format across all the subsystems of the tool. The consistency of the user interface reduces the user learning time of the system that is knowledge learnt about one command is applicable to all other subsystems.

The user is asked to confirm an action that might cause a serious error, which draws the user attention to the possibility of making a mistake.

The presentation of information shows the relationships between the project's objects very easily, which reflects one of the main objectives of the LCCASE tool.

Colour is used to draw the user attention to the important information.

The provision of the user guidance is done through the use of dialogs and lists.

The error messages are linked to the context of the system.

4.2 The Control Strategy Between the Subsystems of the LCCASE Tool

The approach to control the operations of the subsystems of the LCCASE tool is event-based. There are subsystems that respond to external events generated by the developer such as the User Interface, Model, Directory, Data Dictionary, and Development Life Cycle Subsystems. There are messages sent between the subsystems such as the Database Subsystem and most other subsystems.

The Messages Services Subsystem controls the process of passing the events between the subsystems. Each subsystem registers an interest in specific events. When these events occur, the Messages Services Subsystem transfers the control to the subsystem that can handle the event. Subsystems specify which events they require and the Message Services Subsystem ensures that these events are sent to them. Figure 4-8 shows the events and messages control.

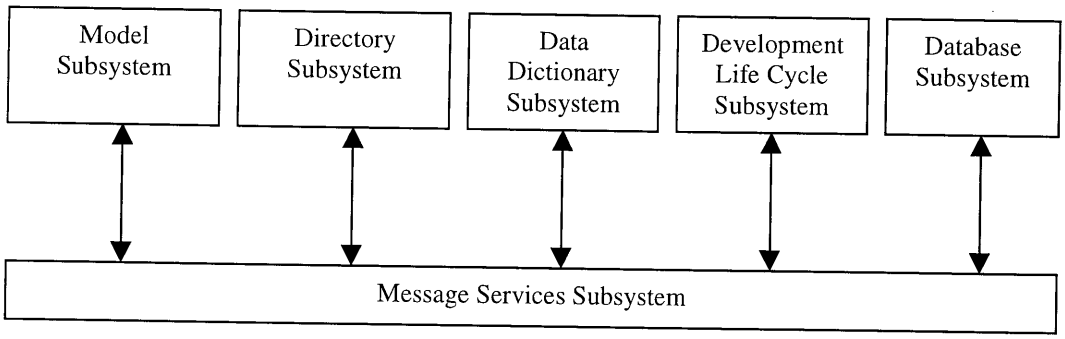


Figure 4-8 Event and Messages Control

4.3 The Environment of the LCCASE Tool

During the development process of the models, the directory shows the names of the project objects and their relations, the Development Life Cycle guides the user with the updated information about the developed models and those to be developed, and the Data Dictionary maintains an updated documentation about the project. Figure 4-9 shows the LCCASE tool environment.

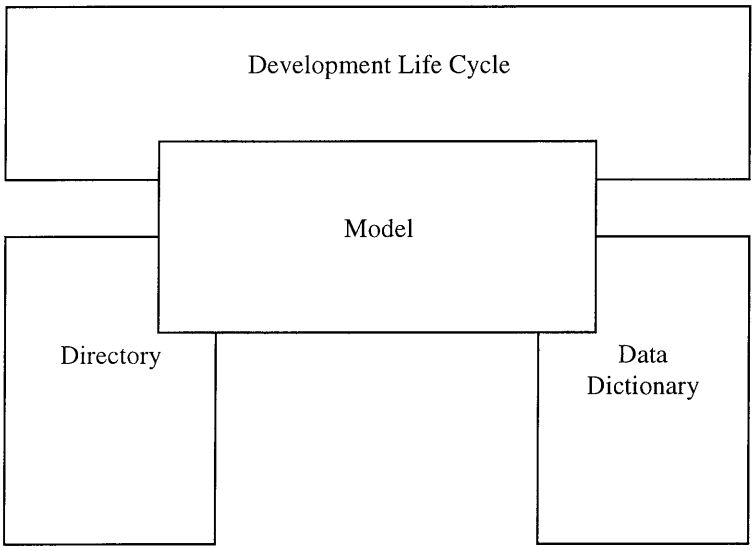


Figure 4-9 The LCCASE Tool Environment

This section starts by describing how the subsystems of the LCCASE tool support the development of each of the models (Figure 4-10), and explains the choices for the development environment. The section then discusses how the LCCASE tool links the information between the models and a discussion about what information can not be linked by the LCCASE tool is given.

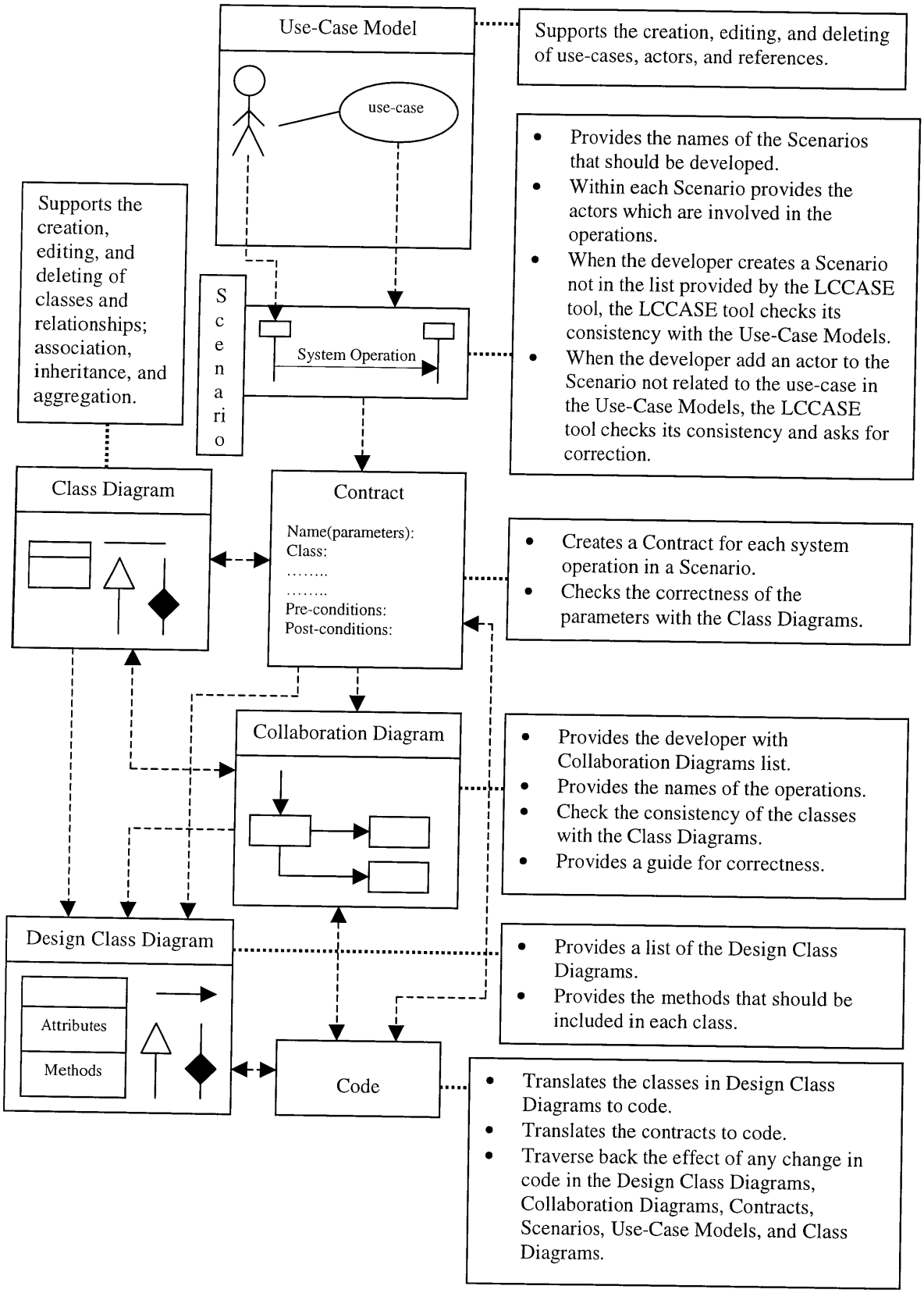


Figure 4-10 The Support of the LCCASE Tool to Models Development

4.3.1 The Use-Case Model

Usually the Use-Case Model is the first model in the life cycle. The Use-Case Model shows the interactions between the system and its users. Thus, it illustrates the requirements of the system. The Use-Case Model contains use-cases and actors. A use-case shows what operations the user can perform on the system. An actor is the user of the system.

This sub-section discusses the support of the LCCASE tool to the development of the Use-Case Models. It shows how each subsystem aids the development of the Use-Case Model.

Development Life Cycle

The Development Life Cycle views the Use-Case Models status button with green colour, which means that the developer can proceed in developing the Use-Case Models.

Model

In each Use-Case Model the developer can create, edit, draw, and delete use-cases, actors, and references. The reference connects a use-case and an actor. The actors can be used in other Use-Case Models. The use-cases and actors can be moved in the diagram while retaining their references.

Directory

Once the developer creates a use-case, the Directory displays its name in the Use-Case List. When the developer select a use-case from the Use-Case List, the name of the actors that are linked to the selected use-case are displayed in the Actor List.

Data Dictionary

The Data Dictionary creates documentation for the Use-Case Model and each of its components such as the use-cases, actors, and references. If any change occurs to the Use-Case Model, the Directory updates its information. In addition, the developer can search the Data Dictionary for a particular item according to its names or for group of items according to their kind.

4.3.2 Class Diagram

The Class Diagrams are developed during the analysis phase. It consists of classes and their relationships such as association, inheritance, and aggregation. The Class Diagrams shows the attributes and the roles of the classes within the relationships.

This sub-section describes how each subsystem of the LCCASE tool participates in developing the Class Diagrams.

Development Life Cycle

The Development Life Cycle displays the Class Diagram Models status button with green colour, which means that the developer can proceed in developing the Class Diagrams.

Model

The LCCASE tool provides means for constructing Class Diagrams. The developer can create, edit, and delete classes and link the classes with association, inheritance, and aggregation relationships. The LCCASE tool supports the developer in drawing the components of the Class Diagram according to the UML notation.

The name, attributes, and methods of a class are displayed within the class box. The name of the class is printed in bold to draw the attention of the developer. The name, access modifier, type of the attribute and method are printed within the class box. The data about the attributes and methods are printed in different colour. Thus the developer can easily recognise them.

The association relationship can be represented by a line or polyline that joins the two classes, which participate in the relationship. The name of the association and its cardinalities are printed above the line and the role of the classes are printed below the line.

The child class in the inheritance relationship inherits the attributes, methods, and relations of the parent class automatically. The names of the inherited attributes and methods are included within the child class. If an inheritance relationship is deleted, all the derived attributes and methods are removed from the children classes.

The developer can draw the aggregation relationship and include the cardinalities.

The LCCASE tool preserves the relationships between the classes even if they are moved.

Directory

The Directory displays the name of a class in the Class List after its creation. When the developer selects a class from the Class List, the name of the association, inheritance, and aggregation relationships together with the name of other classes that participate in the relationship are displayed in the relationships lists.

Data Dictionary

The Data Dictionary creates and updates the documentation for the Class Diagram and each of its components, which are the classes and the relationships.

4.3.3 Scenario

The Scenario shows the operations between the system and the user to accomplish a particular task. It consists of actors and system operations.

Since the use-cases in the Use-Case Models represent the tasks, a Scenario is developed for each use-case.

This sub-section describes how the subsystems of the LCCASE tool support the development of the Scenarios.

Development Life Cycle

The Development Life Cycle displays the Scenarios button status with red colour if no use-cases have been created in the project. This means that the developer should not proceed in developing Scenarios. After the developer creates a use-case, the Development Life Cycle shows the Scenario button with green colour. When the

developer clicks on the Scenario button, a list with names of the Scenarios that should be developed is displayed.

Model

The LCCASE tool uses the information linkage capability, discussed in the next section, to provide the developer with the names of the Scenarios that should be developed. In addition, the developer can create any Scenario. While the developer adds an actor to the Scenario, the LCCASE tool provides him with the name of the actors that should be added to the Scenario. The developer has the freedom of adding actors, which are not in the suggested list. The LCCASE tool maintains the consistency between the Use-Case Models and the Scenarios. It asks the developer to fix any inconsistency. Moreover, within the Scenario the developer can create, edit, and delete system operations and move the actors while retaining their system operations.

A system operation is represented with a line between the two actors involved in the operation and the name of the system operation is printed above the line.

Directory

When the user selects a use-case from the Use-Case List, the LCCASE tool enables the Scenario button if a scenario has been created for the selected use-case where the developer can press this button to display the Scenario. Moreover, it displays the names of the system operations in the System Operation List. The developer can select a system operation to display its Contract.

Data Dictionary

The Data Dictionary provides an updated documentation for the Scenario and each of its components, the system operations and actors.

4.3.4 Contract

Contracts are very valuable models. They provide the developer with tremendous help during the design and the implementation phases. The Contract describes the inputs, outputs, attributes, preconditions, and post-conditions of a system operation.

This section describes the support of the subsystems of the LCCASE tool to develop a Contract.

Development Life Cycle

The Development Life Cycle views the Contracts button status with red colour if no system operation has been created in the project, which means that the developer should not proceed in developing a Contract. Once the developer creates a system operation, the LCCASE tool creates a Contract for the system operation and the Development Life Cycle displays the Contracts button with green colour.

Model

After the creation of a system operation, the Model Subsystem of the LCCASE tool creates a Contract for that system operation and updates its fields such as the name and cross-references.

The Contract is a dialog window containing several fields for the developer to enter and change the Contract entries. The fields are:

- the Name field which contains the name of the system operation. It is not an editable field and is updated by the LCCASE tool;
- the Responsibilities field where the developer enters the responsibilities of the system operation;
- the Class field, which is updated by the user to hold the name of the class that contains the system operation;
- the Cross-Reference field is an editable field. It is updated by the LCCASE tool to hold the name of the models that are related to the system operation. Moreover, it can be edited by the user to enter more information;
- the Notes field, which holds the name of any special algorithms used in the system operation;
- the Exceptions field is updated by the user to hold any exceptions during the execution of the system operation;
- the Output field is updated by the user to contain output messages or dialogs.
- the Pre-Conditions field is updated by the user. It holds the conditions that should be satisfied before executing the system operation;
- the Post-Conditions field is updated by the user. It contains the main operations that are performed by the system operation such as the names of any objects that are created, attributes changes, and association creation or deletion.

Directory

When the user selects a use-case from the Use-Case List, the LCCASE tool displays the names of the system operations in the System Operation List. When the developer double clicks on the system operation name, the Contract window is displayed.

Data Dictionary

The Data Dictionary creates documentation for the Contract and each of its fields. If any change occurs to the Contract and its fields, the Data Dictionary updates its information.

4.3.5 Collaboration Diagram

The Collaboration Diagram shows how objects interact at run time to support the functionality of an operation. A Collaboration Diagram is developed for each system operation. It can be derived from the post-conditions field in the Contract. It consists of sequences of messages that occur between a collection of objects in a specific operation.

Next, the support of the subsystems of the LCCASE tool to develop the Collaboration Diagrams are discussed.

Development Life Cycle

The Collaboration Diagrams button status remains red until the post-condition field of a Contract in the project has been updated with method calls. Once the developer fills the post-condition field of a Contract with the names of the methods that will be called by the method described in the Contract, the Development Life Cycle views the Collaboration Diagrams button with green colour. When the developer clicks on the button, a list with the names of the Collaboration Diagrams that should be developed is displayed.

Model

The LCCASE tool provides the developer with the names of the Collaboration Diagrams that should be developed where each Contract contains method calls in its post-condition field that corresponds to a Collaboration Diagram. In addition, it provides the developer with a list of the methods that are involved in the collaboration.

The developer can add, edit, and delete classes and messages in the Collaboration Diagram while the LCCASE tool maintains the consistency between the Collaboration Diagram, Contracts, Scenarios, and Class Diagrams.

Directory

When the user selects a use-case from the Use-Case List, the LCCASE tool enables the Scenario button. It displays the names of the system operations which are created in the Scenario in the System Operation List and the names of the Collaboration Diagrams in the Collaboration Diagram List which correspond to the selected use-case.

The viewing of the names of all subsystems in the project that are related to or determined from a use-case, allows the developer to trace the subsystems and their relations in the project.

Data Dictionary

The Data Dictionary creates and updates the documentation for the Collaboration Diagrams and each of its components.

4.3.6 Design Class Diagram

The Design Class Diagrams show the Class Diagrams from the design phase view. They show the visibility of the classes and the methods.

This sub-section shows how the subsystems of the LCCASE tool supports the development of the Design Class Diagrams.

Development Life Cycle

When a Class Diagram, Contract, or Collaboration Diagram has been created in the project, the Design Class Diagrams button status changes to green.

Model

The Model Subsystem of the LCCASE tool provides the developer with a list, which contains the names of the Design Class Diagrams that should be developed, a list with names of the methods that should be added to each class in the Design Class Diagram, and a list with the names of the Collaboration Diagrams that contain the classes in the Design Class Diagram.

The LCCASE tool provides the developer with the Class Diagram that corresponds to the Design Class Diagram, the developer then can change it to construct the Design Class Diagram.

Directory

When the developer selects class from the Class List, the Directory displays the relationships' lists containing the names of those relationships in which the selected class is involved.

Data Dictionary

The Data Dictionary creates and updates the documentation for the Design Class Diagrams and each of its subsystems.

4.3.7 Code Generation

Code Generation occurs during the implementation phase. The LCCASE tool is being designed to be able to support the generation of the code.

Next, the support of the subsystems to generate code is discussed.

Development Life Cycle

The colour of the Code Generation button changes to green after the creation of a Design Class Diagram in the project.

Model

The code in the Java programming language can be generated for each class in a Design Class Diagram. It contains the class name, attributes, and methods. Moreover, it contains method calls, which are ordered according to the sequence in the Collaboration Diagrams. The information about the class may be determined from one or more Design Class Diagram.

Directory

When the developer double clicks on a class name in the Class List, the information dialog of the class and the names of the class relations in the relationships lists are

displayed. This allows the developer to view all the information about the class to check or add to the generated code.

Data Dictionary

The Data Dictionary is developed to be able to create and update the documentation for the generated Code.

4.3.8 Information Linkage in the LCCASE Tool

The LCCASE tool has the ability to link the information between the models, as discussed in chapter 3. Dashed arrows in Figure 4-10 represent the links between models. These links are based on the use-cases and classes, which are created by the developer. The following are the links of information supported by the LCCASE tool:

- There is a one-to-one relationship between a use-case and a Scenario such that each Scenario corresponds to a use-case in the project. The use-cases represent the functions of the user on the system. In order to perform these functions, various interactions of the user with the system are usually required. Since the Scenarios are used to explain those interactions, linking a use-case to the Scenario that describes it is very helpful for the developer. This link facilitates tracing the functions. In addition, the LCCASE tool uses this link to suggest the Scenarios that needed to be developed as a result of the created use-cases.
- The actor in a Scenario is the same actor that is linked via a reference to the use-case corresponding to the Scenario. The actors represent the person or software that interacts with the system to perform a particular function. Since this function is the use-case that corresponds to the Scenario as described above, the actors linked to the

use-case should be the actors in the corresponding Scenario. This link is used to check the consistency of the actors within the Use-Case Models and the Scenarios. The LCCASE tool utilises this link to suggest the actors that should be included in a Scenario.

- There is one-to-many relationship between a Use-Case Model and the Scenarios in the project. A Use-Case Model may contain any number of use-cases. Each use-case corresponds to a Scenario. This link is useful to determine the required actors in the Scenario.
- The LCCASE tool automatically creates a Contract for each system operation in a Scenario. There is one-to-one relationship between a Contract and a system operation. Contracts are developed to explain important features about the operation of a system. This relationship links a system operation to its Contract.
- The parameters of a system operation are classes or attributes in the project. The LCCASE tool uses these links to verify the consistency of the parameters with the classes and attributes created previously in the project.
- In the UML notation, the post-conditions in a Contract specify messages between objects, object creation or deletion, and attribute modification. A Collaboration Diagram is used to specify the messages sent between objects. Thus, there is one-to-one relationship between a Collaboration Diagram and a Contract that has post-conditions. As a result of this relationship the LCCASE tool maintains various links, which are the name of the Collaboration Diagram is the same as the name of the Contract, the messages specified by the post-conditions in the Contract are the messages between the objects in the Collaboration Diagram, the controller class in the

Collaboration Diagram is the class specified by the class field in the Contract, and the collaborator classes in the Collaboration Diagram should be classes created in the project. The LCCASE tool obtains most of the information needed to develop a Collaboration Diagram from these links, which reduces the process of creating a Collaboration Diagram tremendously.

- There is one-to-one relationship between Design Class Diagrams and Class Diagrams. A Design Class Diagram shows a Class Diagram in the design view. A Design Class Diagram is developed by altering the corresponding Class Diagram to include more information about the methods and visibility of classes, which are known during the design phase.
- The messages, which are system operations, between the objects in a Collaboration Diagram are methods within classes in the Design Class Diagrams.
- There is one-to-one relationship between system operations and methods. A system operation, which is created in a Scenario and explained by a Contract during the analysis phase, is linked to a method in the design phase. This link facilitates the development of a method since the necessary information about the method (inputs, outputs, responsibility, exceptions, calls to other methods, creation of objects, and changing the value of attributes) is included within the Contract.
- There is correspondence between a class in a Design Class Diagram and its implementation in code. The attributes and methods of the class in a Design Class Diagram specify the attributes and methods of the class during code and the Contract specifies the body of a method.

Further linkage is possible. This includes links between specification requirement statement and use-cases or classes. However, the specification requirement statement has not been implemented in the LCCASE tool. Thus, these links are not currently supported by the LCCASE tool.

4.4 The Chapter Summary

The chapter discusses the architecture design of the LCCASE tool. The architecture of the LCCASE tool contains two layers. The first layer contains the subsystems of the LCCASE tool, which are the Model, Directory, Development Life Cycle, Data Dictionary, Database, and the User-Interface Subsystems. The second layer contains the Message Services Subsystem that allows other subsystems to communicate with each other.

The Model Subsystem is responsible for developing the models during the software development life cycle, generating intelligent guidance that is adapted to the user's needs during the development of the models, maintaining links between the models to support reuse and consistency checks, and drawing the diagrams for the models.

The Directory Subsystem displays the names of all use-cases, actors, system operations, collaboration diagrams, classes, and their relationships in a project. It helps the developer to relate the concepts in different models and draw his attention to the important concepts.

The Development Life Cycle Subsystem is an aiding tool for the learners to perform their task better and learn more. It shows the developer his progress during the software

life cycle by displaying the names of the models that have been developed and the names of the models that should be developed.

The Data Dictionary Subsystem generates and updates the documentation of the developed project automatically. In addition, it provides the developer with a search facility within the Data Dictionary.

The Database Subsystem supports the process of creating, reading, updating, and deleting objects in the project. It defines and manages the relationships between the stored objects, defines and creates structures of group of objects, and provides access and update services to objects.

The User Interface Subsystem produces a rich GUI environment to support the user through the software development life cycle. It is interactive and relatively easy to learn. It uses windows, menus, buttons, dialogs, and graphics to communicate with the user.

Finally, the chapter described the linkage of information in the LCCASE tool environment where the information in the models are linked logically and systematically.

5 An Example to Show

This chapter describes the development of a project (a library system) using the LCCASE tool, illustrating functionality and highlights the characteristics of the LCCASE tool. Moreover, it shows the role of each component of the LCCASE tool during the development of a Library System project.

While the models in the project are being developed, I discuss the operations performed by the user and the response of LCCASE the tool to such operations. In order to avoid the confusion with the user of the project being described in this chapter, the user of the LCCASE tool is called the developer. Common features in different models are discussed when first introduced.

5.1 Overview

The project shows the analysis phase of the life cycle to develop a library system. Each of the following subsections describes the operations in a model created during the analysis phase and discusses its important characteristics.

The Use-Case Model, Class Diagram, and Scenario subsections describe how to create, browse, and delete the models of those types using the LCCASE tool and explain how to create, modify, and delete the components of the models.

The Contract subsection describes how to update a contract since it is created automatically by the LCCASE tool after the user creates a system operation.

The Window Menu subsection illustrates the dynamic structure and the tile across features of the Window menu.

The File Menu subsection shows how the LCCASE tool opens, saves, and prints files.

The Data Dictionary subsection illustrates the functionality of the Data Dictionary Subsystem.

During the development of the life cycle models, the roles of the Directory and the Development Life Cycle Subsystems are explained.

5.1.1 The Library System Requirement Statement

The project is a library system whose requirement statement is adapted from [Barclay 1997]. It states that a library has a name and holds a collection of publications that may be periodicals or books. All publications have a title, publication date, publisher, and a unique reference number. However, periodicals have an editor and a list of articles each with a title and an author. Books have an author and edition number.

There are many users for the library and each has a unique name and address. A user may:

- request a display of the current list of library publications available for loan;
- borrow a periodical or book from the library by supplying its reference number;
- return a periodical or book to the library.

5.1.2 The Use-Case Model

The Use-Case Model is a dialog between the system being developed and its users that represents their needs. It contains use-cases and actors. While a use-case represents a functionality of the system, an actor is a user of the system. Each actor performs a particular function. The actor is linked to the use-case via a reference.

This subsection takes the reader through the development process a Use-Case Model in the project.

5.1.2.1 The User Use-Case Model

The User Use-Case Model shows the interaction between the user of the library and the library system. The Actor in this model is the *User*. The actor performs the following operations:

- Borrow a publication from the library.
- Return a publication to the library.
- Request the library system to display a list of the currently available publications in the library.

Each operation is represented by a use-case in the User Use-Case Model.

5.1.2.2 The Development of the User Use-Case Model

In what follows the creation and editing of use-cases, actors, and references are described.

Use-Case

The creation of use-case, *borrow*, starts once the developer activates the Use-Case Button. The LCCASE tool displays a Use-Case Dialog (Figure 5-1), which contains two fields to enter the name and documentation of the use-case. When the developer specifies the location of the use-case in the diagram by clicking on the location, the LCCASE tool draws the use-case, *borrow*, in the location specified by the developer, and writes the name of the use-case.

Actor

The procedure to create an actor is the same as the use-case, which reflects the uniform strategy of the LCCASE tool. When the developer clicks on the Actor Button, the LCCASE tool starts creating the actor, *User*. It displays an Actor Dialog, which has the same structure of the Use-Case Dialog. The developer enters the name and the documentation of the actor. Figure 5-2 shows the Actor Dialog after entering the information. Finally, the LCCASE tool draws the actor in the specified location in the diagram and writes the name of the Actor, *User*, below it.

Reference

The LCCASE tool allows the developer to define a reference between the actor, *User*, and the use-case, *borrow*. The creation process starts by clicking on the Reference Button. The developer specifies the actor and the use-case by clicking on them, the LCCASE tool then draws the reference line between the actor and the use-case. Figure 5-3 shows the User Use-Case Model.

5.1.2.3 Important Features

Deleting a Component of the Use-Case Model

The LCCASE tool enables the developer to delete any element in a Use-Case Model. Figure 5-4 shows the User Use-Case Model illustrating the results of deleting the reference between the actor, *User*, and use-case, *borrow*. In addition, Figure 5-4 shows the Directory window that demonstrates the selection of the use-case, *borrow*, from the Use-Case list while the Actor list did not display the deleted actor name. When the developer makes any changes in any model, the Directory reflects and shows the changes

directly. This demonstrates the interactivity feature of the LCCASE tool discussed in chapter 3.

To ensure the consistency between the models, the developer cannot delete any actor or use-case involved in a reference. The LCCASE tool displays "CanNot Be Deleted" Dialog. Figure 5-5 shows the "CanNot Be Deleted Dialog", which was displayed when the developer tried to delete the actor, *User*. The developer must delete all the references connected to the actor before deleting the actor. This causes the developer to think before deleting the references since they have enormous effect on the corresponding Scenario and the Contracts.

Preventing Overlapping of Objects

The LCCASE tool does not allow the developer to draw an object on top of an existing object in the diagram. This feature facilitates drawing diagrams especially the complicated ones and prevents objects being hidden behind others.

Deleting the Use-Case Model

When the developer decides to delete any model in the project, the LCCASE tool provides him with Confirm Deletion Dialog (Figure 5-6), which displays the message "Are you sure that you want to delete The User Use-Case Model". According to the response of the user, the LCCASE tool then proceeds to delete the model or cancels the deletion process.

5.1.2.4 The Role of the Directory

Once the user clicks on the Use-Case Button in the Directory window, the LCCASE tool displays the Use-Case list, which contains the names of the use-cases that have been created in the project. Figure 5-7 shows the Use-Case View of the Directory window after selecting use-case, *borrow*, from the Use-Case list.

The Use-Case list displays the names of the use-cases that are created in the project, giving the developer a clear view of all the use-cases in the project without the need to view each Use-Case Model.

When the developer selects a use-case from the Use-Case list, the LCCASE tool:

- displays in the Actor list those actors' names that are connected to the use-case via a reference. Knowing the actors of a use-case has an important effect in developing other models in the system such as the Scenario and the Class Diagram;
- displays the Use-Case Model that contains the use-case;
- if a Scenario has been created for the use-case, the LCCASE tool activates the Scenario Button located in the Directory. The developer can then access the Scenario. This allows the developer to view the Scenario of the use-case easily instead of searching for it in the Scenario list;
- displays the names of the system operations that have been created in the Scenario of the selected use-case in the System Operation list located in the Directory;
- displays the names of the Collaboration Diagram that contain a selected system operation from the System Operation list.

The Directory Subsystem provides the developer with navigational capabilities since the developer can access the Use-Case Models, Scenarios, Contracts, and the Collaboration Diagrams from the Directory.

The operations of the Directory discussed above allow the developer to have clear view of the relations between the information in the project. They assist the developer in tracking the requirements specified by the selected use-case in other models, which provides the developer with a feedback about the correctness of implementing the requirements and the effectiveness of his approach in developing successful software.

When the developer double clicks on a use-case, an actor, or a system operation, the LCCASE tool displays the corresponding information dialog. This provides the developer with an alternative way to access the information about the objects in the project. Moreover, the developer can modify the information in the information dialog and the LCCASE tool reflects those changes in the corresponding drawings.

5.1.2.5 The Role of the Development Life Cycle

Before the developer creates a use-case in the project, the Development Life Cycle window displays the colour of the Use-Case Model, and Class Diagram Buttons in green, which means that Use-Case Models and Class Diagrams can be developed. All other buttons are red which indicates that the developer should not develop models of those types. Figure 5-8 shows the Development Life Cycle window before creating a use-case.

The Development Life Cycle guides the developer through the development phases of the project. It provides him with the information of what models should be developed, which reduces the complexity of the software development process. This implements the

step-by-step strategy discussed in chapter 3. In addition, the Development Life Cycle maintains the individualisation approach since displaying the Development Life Cycle is optional and inexperienced developers benefit the most from this facility.

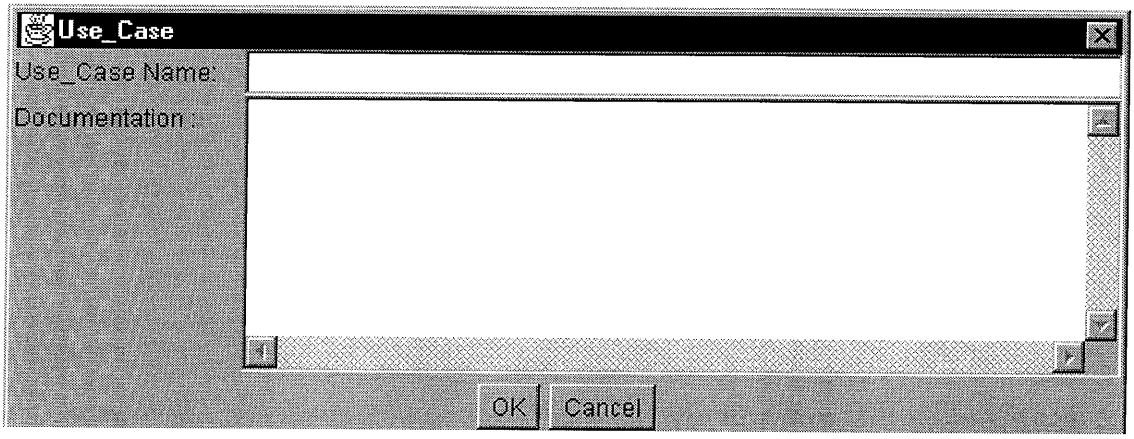


Figure 5-1 The Use-Case Dialog

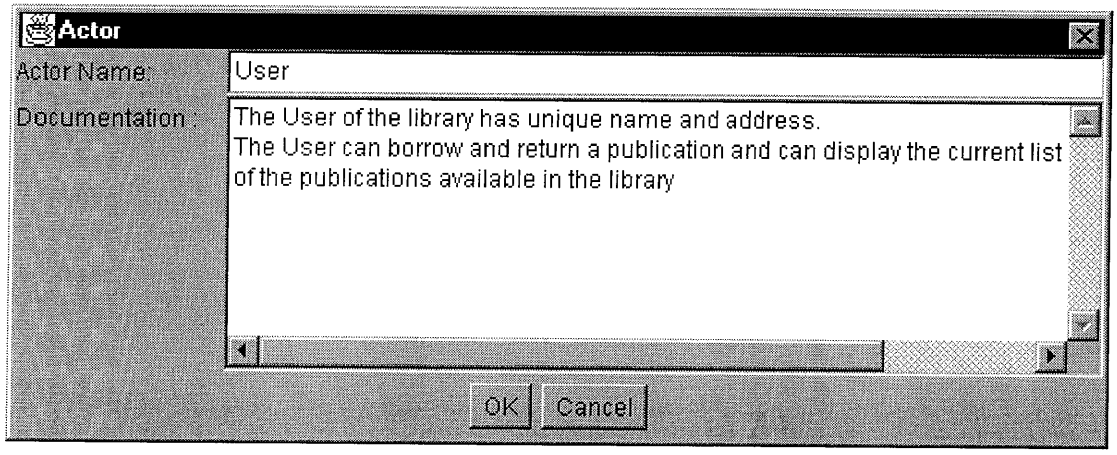


Figure 5-2 The Actor Dialog

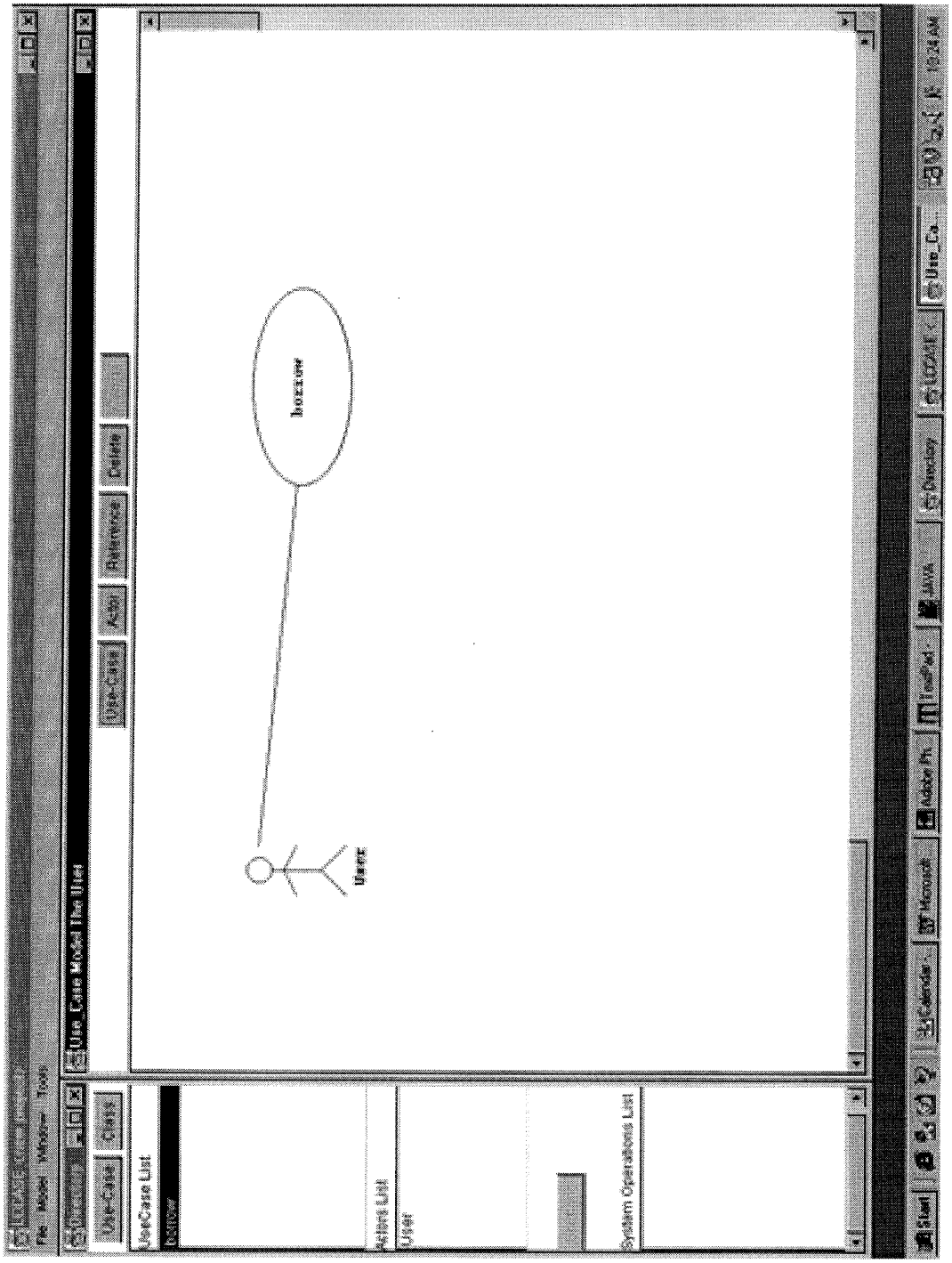


Figure 5-3 The User Use-Case Model

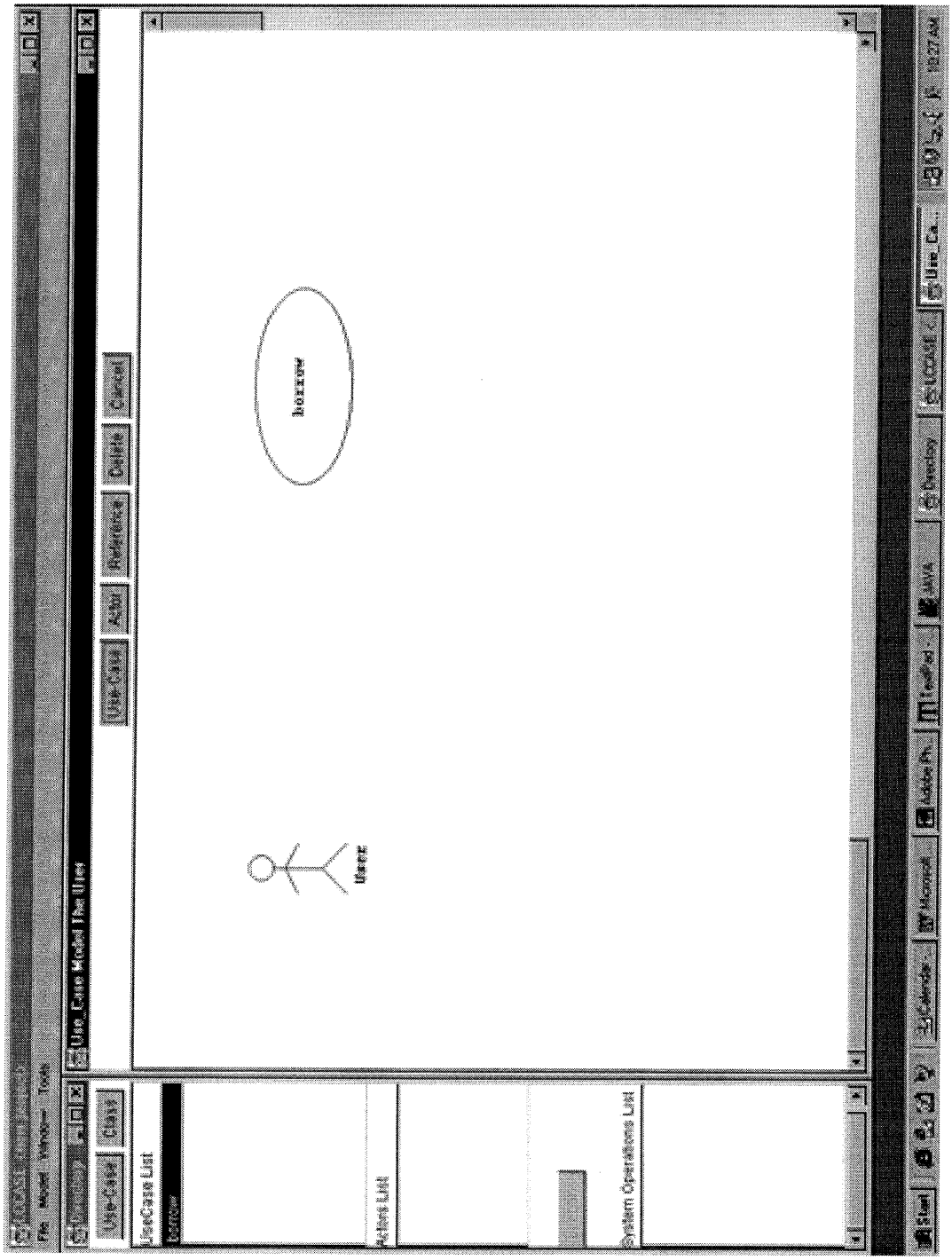


Figure 5-4 The User Use-Case Model after Deleting the Reference



Figure 5-5 CanNot be Deleted Dialog

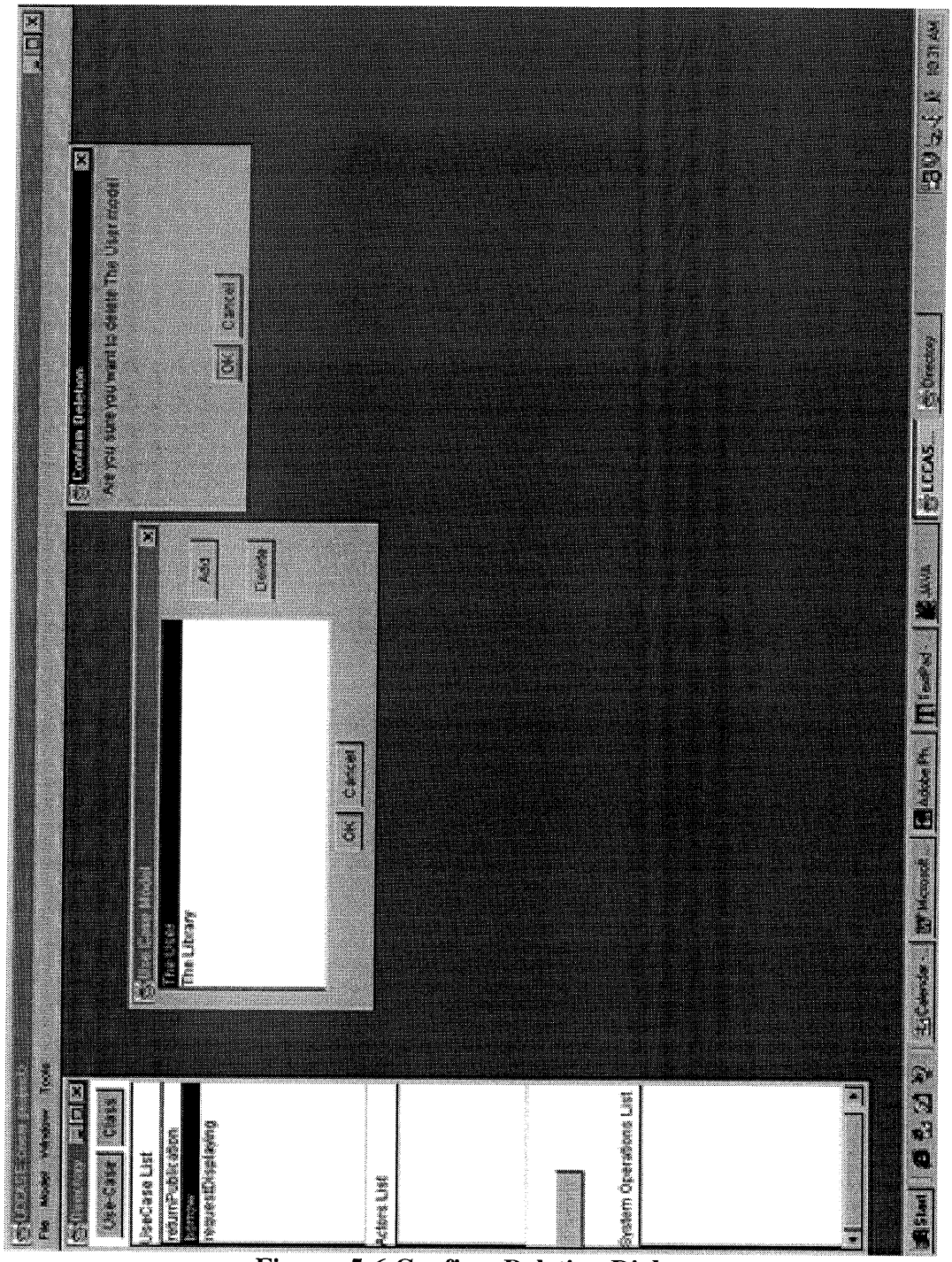


Figure 5-6 Confirm Deletion Dialog

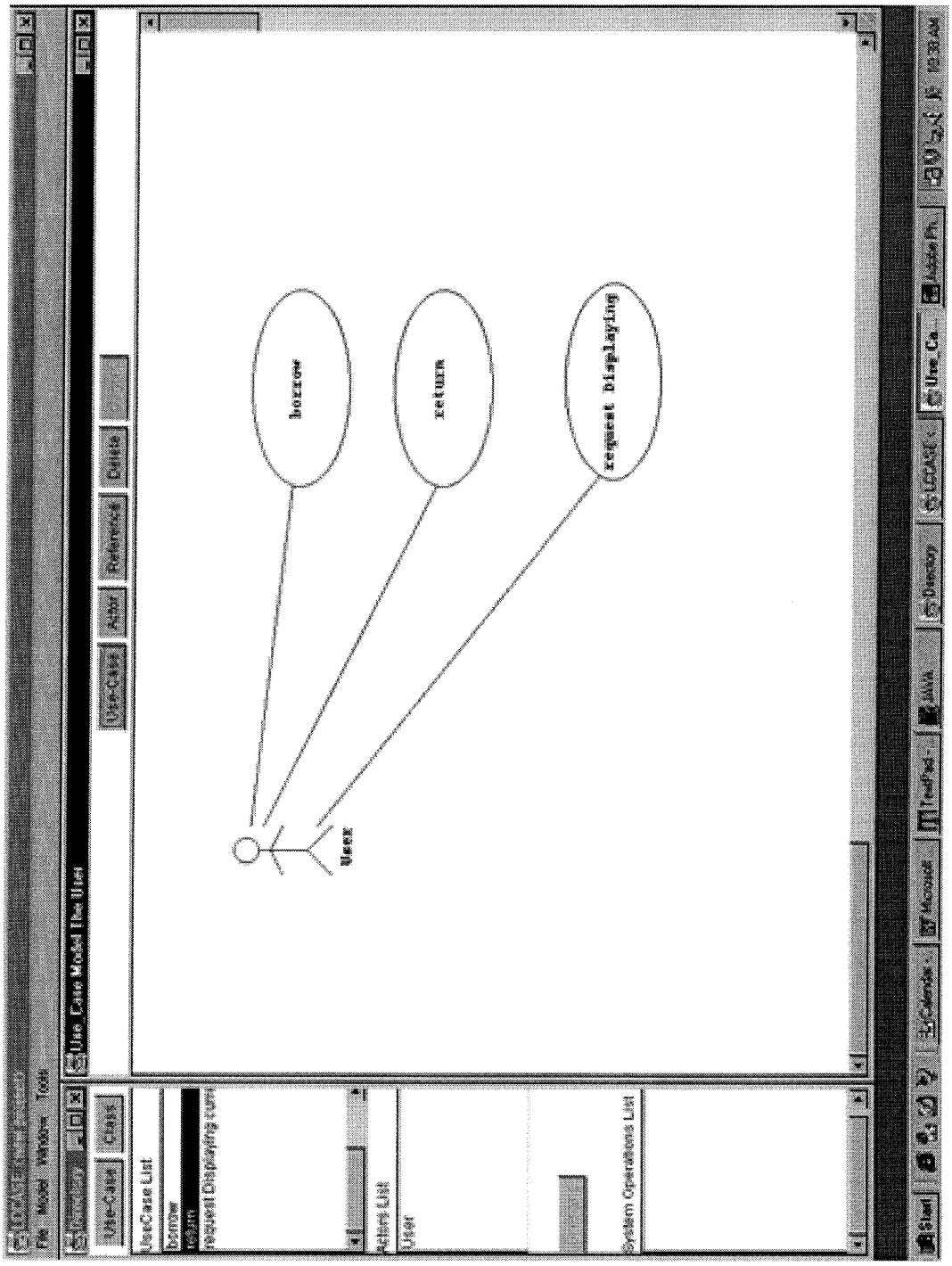


Figure 5-7 The Use-Case View of the Directory

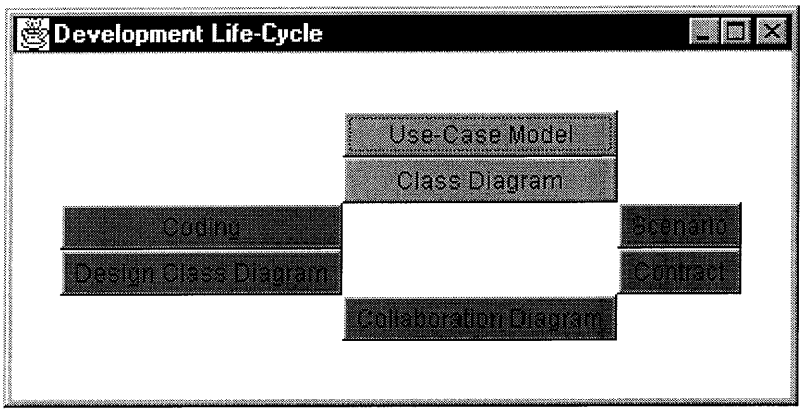


Figure 5-8 The Development Life Cycle before Creating a Use-Case

5.1.3 The Class Diagram

The Class Diagram consists of classes and their relations. Each class has a unique name and can have attributes and operations. The relationships between the classes are association, inheritance, and aggregation. Class Diagrams are very important models in the software life cycle. They have great effect in the production of the system code.

5.1.3.1 The Publication Class Diagram

The Publication Class Diagram contains Publication, Periodical, Book, and Article classes, their attributes, and the relationships between them. The relationships are:

- Inheritance relationship between the parent class, *Publication*, and the child class, *Periodical.*
- Inheritance relationship between the parent class, *Publication*, and the child class, *Book.*
- Aggregation relationship between the container class, *Periodical*, and the part class, *Article.*

5.1.3.2 The Development of the Publication Class Diagram

The creation, editing, and deletion operation on classes and relationships in a Class Diagram are discussed below.

Class

The LCCASE tool allows the developer to create and edit classes. Each class has a unique name and can contain attributes and methods. The developer starts the creation of the class, *Publication*, by clicking on the Class Button.

Once the developer places the class in the diagram, the LCCASE tool displays the Class Dialog. The Class Dialog contains fields for almost all the information about a class in the project. To make the process of entering the class information easy and straightforward, the Class Dialog has a tab layout, which includes General, Details, Attribute, and Method tabs. The developer can enter the information easily by selecting the tab and then filling out the required fields.

The General tab view contains fields to enter the name of the class and documentation and Parent field. The Parent field holds the name of the parent class if any. The LCCASE tool updates this field automatically after the developer has created an inheritance relationship. The Parent field draws the attention of the developer to the availability of the parent class and an inheritance relationship, which may affect the attributes and the methods of the developed class.

The Details tab view contains lists to select the access modifier, cardinality, and concurrency of the class and a check box to specify if the class is an abstract class.

The Attribute tab view includes a list which holds the names of the attributes that have been created and buttons that activate the commands to add a new attribute, modify a selected attributed from the Attribute list, and delete a selected attribute.

The Methods tab view has the same structure as the Attribute tab view. It supports displaying, adding, modifying, and deleting the methods in a class.

Figure 5-9 shows the General tab view in the Class Dialog of class, *Publication*. Figure 5-10 shows the Details tab view.

The developer enters the attributes of the class using the Attribute Dialog. The Attribute Dialog contains fields to enter the important information abut the attribute such as its name, type, and initial value and a list to select the access modifier. In addition, it contains a Class field, which is updated by the LCCASE tool to hold the name of the class that is originally contains the attribute. If an attribute is inherited from another class, the name of the parent class is written in the Class field. Figure 5-11 shows the Attribute Dialog after entering the information. Once the user enters the attribute data, the Attribute tab view adds the attribute name to the Attribute list. Figure 5-12 shows the Attribute tab view after adding the attribute, referenceNo.

When the developer has supplied all the class information, the LCCASE tool draws the class. The names of the attributes are displayed. Figure 5-13 shows the Publication Class Diagram containing the class, *Publication*. The name of the class is displayed in bold, the colour of the attribute name is red, and the colour of the method name is blue to help the user recognise and remember them.

Inheritance Relationship

There is an inheritance relationship between the *Publication* class and the *Periodical* class where *Publication* is the parent class and *Periodical* is the child class. Figure 5-13 shows the Publication Class Diagram before creating the inheritance relationship.

The LCCASE tool supports the developer in creating an inheritance relationship. The inheritance relationship development starts by clicking on the Inheritance Button. When the developer specifies the child and parent classes by clicking on them, the LCCASE tool draws the inheritance relationship and shows that the child class, *Periodical,* inherits all the protected attributes in the parent class, *Publication.* Figure 5-14 shows the Publication Class Diagram after creating the inheritance relationship.

The LCCASE tool redraws the children classes in the inheritance relationship to display the inherited attributes and methods, reminding the developer of the inherited attributes and methods especially when the child class has appeared in a different Class Diagram. In Figure 5-14, the class view of the Directory shows the inheritance relationship between the classes, *Publication* and *Periodical* in the Inheritance list because the *Publication* class has been selected from the class list.

The developer can display and edit the inheritance relationship information dialog. The Inheritance Relationship Dialog contains the fields, name and documentation that need to be filled by the developer and fields that are updated by the LCCASE tool to hold the names of the parent and child classes. Figure 5-15 shows the Inheritance Relationship Dialog. The parent and child classes fields clarify the role of each class in the inheritance relationship.

Aggregation Relationship

The developer starts the creation of an aggregation relationship between the component class, *Article*, and the container class, *Periodical*, by clicking on the Aggregation Button. After the developer specifies the component and the container classes, the LCCASE tool

draws the aggregation relationship between them. Figure 5-16 shows the Publication Class Diagram after creating the aggregation relationship.

The developer can modify the information of an aggregation relationship from the Aggregation Relationship Dialog.

To specify the cardinality of the container and component classes in the aggregation relationship between the classes, *Article and Periodical*, the developer double clicks on the aggregation relationship. The LCCASE tool then displays the Aggregation Relationship Dialog. The Aggregation Relationship Dialog displays the name of the container and part classes and has the same structure of the Inheritance Relationship Dialog.

When the developer modifies the required information, the LCCASE tool redraws the Aggregation relationship in the Class Diagram to show the cardinalities. Figure 5-17 shows the Aggregation Dialog after adding the cardinalities. Figure 5-18 shows the Publication Class Diagram after modifying the aggregation relationship.

Association Relationship

To create the association relationship, *borrows*, between the classes, *User* and *Library*, the developer clicks on the Association Button. When the developer clicks on the two classes that are involved in the association relationship, the LCCASE tool displays the Association Relationship Dialog. The Association Relationship Dialog contains fields to enter the name and roles of the association relationship and lists to select the cardinalities. In addition, it displays the names of the classes involved in the relationship. Figure 5-19

shows the Association Dialog after entering the information. Consequently, the LCCASE tool draws the association relationship and displays its name and cardinalities.

The developer can easily create the association relationship because he has only to fill the Association Relationship Dialog with the information about the relationship and the LCCASE tool draws the relationship automatically.

Moreover, the developer can create an association relationship that can be represented by a polyline to facilitate drawing complex diagrams. The association relationship, *returns*, between the classes, *User* and *Library,* has been represented by polyline. Figure 5-20 shows the Library Class Diagram after creating the association relationship, borrows and *returns*.

5.1.3.3 Important Features

Deleting an Association Relationship

Once the developer clicks on the Delete Button and he then clicks on the association relationship, *returns*. The LCCASE tool deletes the association relationship, *returns*.

The Directory deletes the association relationship, *returns,* from the association list. Figure 5-21 shows the Library Class diagram after deleting the association relationship, *returns*.

Deleting an Inheritance Relationship

Figure 5-21 shows the User Class Diagram. In this Class Diagram the requirement statement of the example has been modified so that the user of the library can either be student or staff. The student can be postgraduate or undergraduate.

The LCCASE tool allows the developer to delete an inheritance relationship. Once the developer clicks on the Delete Button and then clicks on the inheritance relationship between the parent class, *User*, and the child class, *Student*, the LCCASE tool deletes the inheritance relationship and removes all the inherited attributes from the children. The attribute, *name*, is removed from the classes *Student*, *Undergraduate*, and *Postgraduate*. The classes, *Undergraduate* and *Postgraduate*, inherit from the class, *Student*. Figure 5-22 shows the User Class Diagram after deleting the inheritance relationship.

The LCCASE tool redraws all the children classes automatically after the deletion of the inheritance relationship to remove the inherited attributes and methods, which saves the developer the time and effort of redrawing the children classes. Moreover, it clarifies the concept of inheritance.

Modifying an Attribute

The LCCASE tool allows the developer to modify the information of the attributes and methods of a class. To change the access modifier of the attribute, *referenceNo*, from *private* to *protected*, the user double clicks on the class, *Publisher*. The LCCASE tool then displays the Publication Class Dialog. The developer selects the Attribute tab and then selects the attribute, *referenceNo*. When the developer clicks on the Modify Button, the LCCASE tool displays the Attribute Dialog of the attribute, *referenceNo*. The developer selects *Protected* from the Access Modifier list. Next, the LCCASE tool redraws the Publication class. Figure 5-23 shows the Publication Class Diagram window after the modification of the attribute, *publisher*, in the Publication Class.

The developer can easily modify the information of a class by changing the required information in the Class Dialog. The LCCASE tool then redraws the class automatically to reflect the changes in the information of the class and any necessary adjustments. This saves the developer time and effort by not having to perform these tasks manually.

Moving a Class

One of the facilities that the LCCASE tool provides its users is the ability to move objects, i.e. classes that have been drawn in a diagram while preserving their relations with other objects. Figure 5-18 shows the Publication Class Diagram before moving the class, *Book* to a new location in the class diagram. To move a class the developer has to drag the class to the new location. Figure 5-23 shows the Publication Class Diagram after moving the class, *Book.*

Deleting an Aggregation Relationship

The developer can delete relationships or classes that have been created in a model. To delete the aggregation relationship between the classes, *Article* and *Periodical*, the developer clicks on the Delete Button and then clicks on the aggregation relationship. The LCCASE tool then deletes the aggregation relationship and redraws the Class Diagram. Figure 5-24 shows the Publication Class Diagram after deleting the aggregation relationship.

Adding a Method

The LCCASE tool allows the developer to add methods to a class. To add the method, to the class, *Library*, the developer double clicks on the class. The LCCASE tool then

displays the Class Library Dialog. The developer then clicks on the Method tab. After that the LCCASE tool displays the Method tab view. When the developer clicks on the Add Button, the LCCASE tool displays the Method Dialog. Figure 5-25 shows the Method Dialog after entering the information. The LCCASE tool redraws the Library class to show the method, *borrow*. Figure 5-26 shows the Library Class Diagram after adding the method.

5.1.3.4 The Role of the Directory

The LCCASE tool displays in the Class list the names of the classes that are created in all Class Diagrams in the project, which reminds the developer with the classes that are created in the project, allows him to access their information, and helps him to reuse them.

Selecting a Class Name from the Class List in the Directory

The developer can select a class name from the Class list. Once the developer selects the class, *User*, from the Class list, the LCCASE tool displays the association, inheritance, and aggregation relationships of the class in the corresponding lists in the Directory. The ability of the developer to view the names of all the relationships of a class and the name of other classes involved in the relationships provides him with the opportunity to relate the components of the Class Diagrams. Figure 5-26 shows the class view of the Directory after selecting the class, *User*.

Moreover, when the developer selects any of the relationships from the relationships' lists, the LCCASE tool displays the relationship dialog, i.e. when the developer selects the aggregation relationship between the classes, *Periodical* and *Article,*

from the Aggregation list, the LCCASE tool displays its Aggregation Dialog. Figure 5-27 shows the Aggregation Dialog.

The developer can modify the information in the dialog, the LCCASE tool then redraws the aggregation relationship in the Class Diagram to show its cardinalities. Figure 5-28 shows the Library Class Diagram after adding the cardinalities.

When the developer selects a name of a class from the Class list, the LCCASE tool displays in the Directory window:

- in the association list, the names of the association relationships in which the class is involved;
- inheritance relationships in the inheritance list;
- aggregation relationships in the aggregation list.

When the developer double clicks on a class's name or a relationship, the LCCASE tool displays its information dialog. This allows the developer to have a clear view of the class attributes, methods, and their relationships and find the class hierarchies.

When the developer modifies any information in the dialog, the LCCASE tool redraws the class or the relationship to reflect the new changes. This provides the developer with a navigational capability and an alternative way to access the information of class or relationship.

5.1.3.5 The Role of the Development Life Cycle

The Development Life Cycle window shows the status of the model development at this stage. Since use-cases have been created, the colour of the Use-Case Model, Class

Diagram, and Scenario Buttons are green. Figure 5-29 shows the Development Life Cycle window after creating a use-case.

When the developer clicks on the Scenario Button, the LCCASE tool displays the Scenario Dialog, which contains lists to show the names of the developed and should be developed scenarios in the project. This guides the developer through the development process to know the models that need to be developed. Figure 5-30 shows the Scenario Status Dialog.

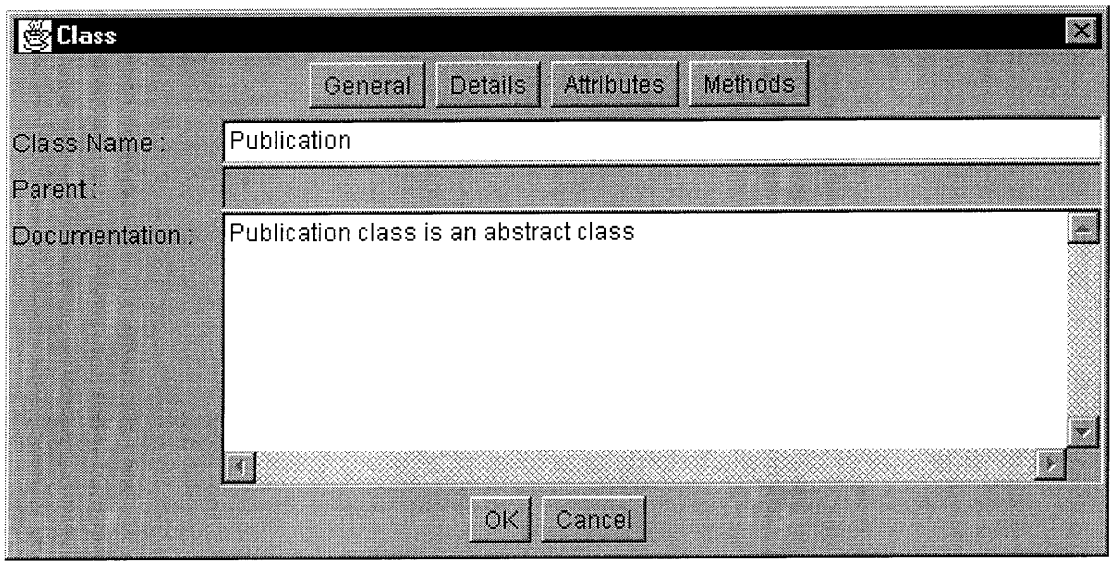


Figure 5-9 The General View of the Class Dialog

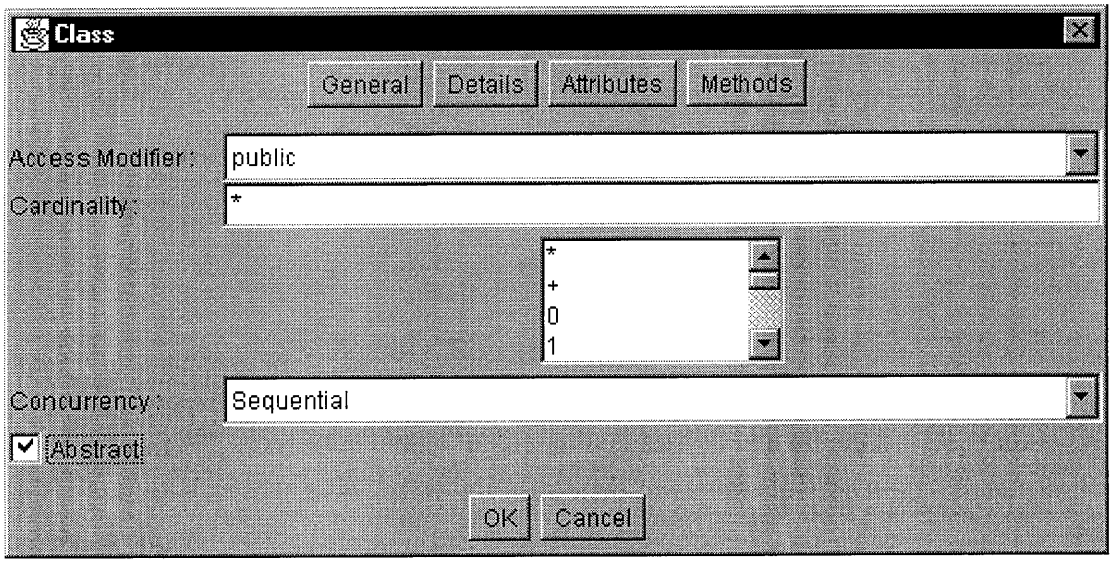


Figure 5-10 The Details View of the Class Dialog

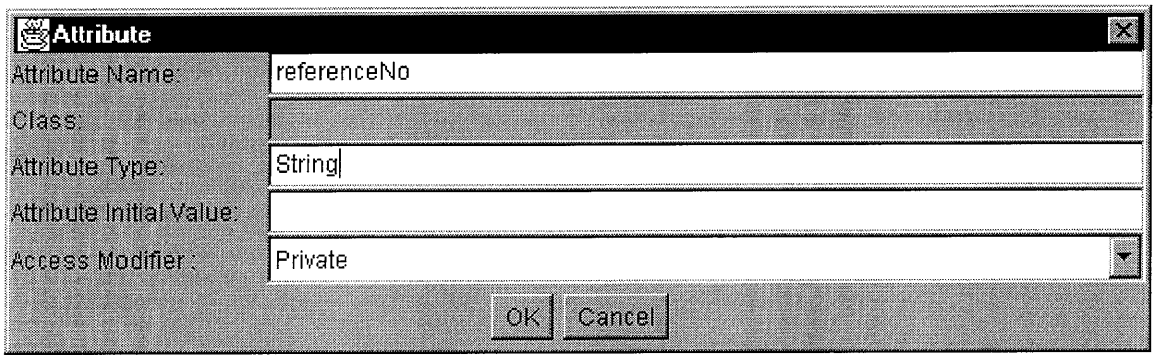


Figure 5-11 The Attribute Dialog

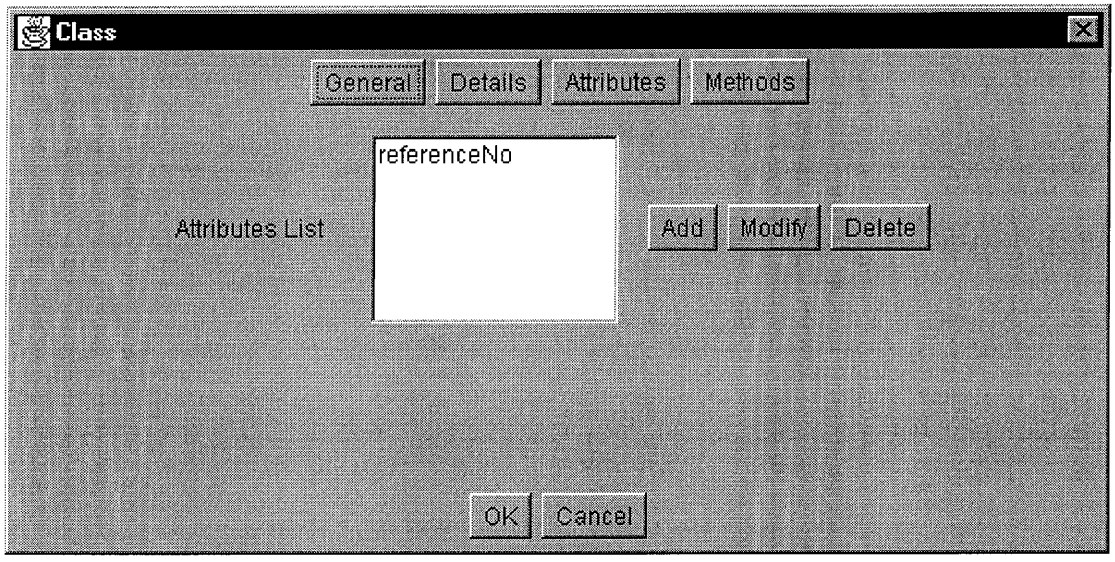


Figure 5-12 The Attribute View of the Class Dialog

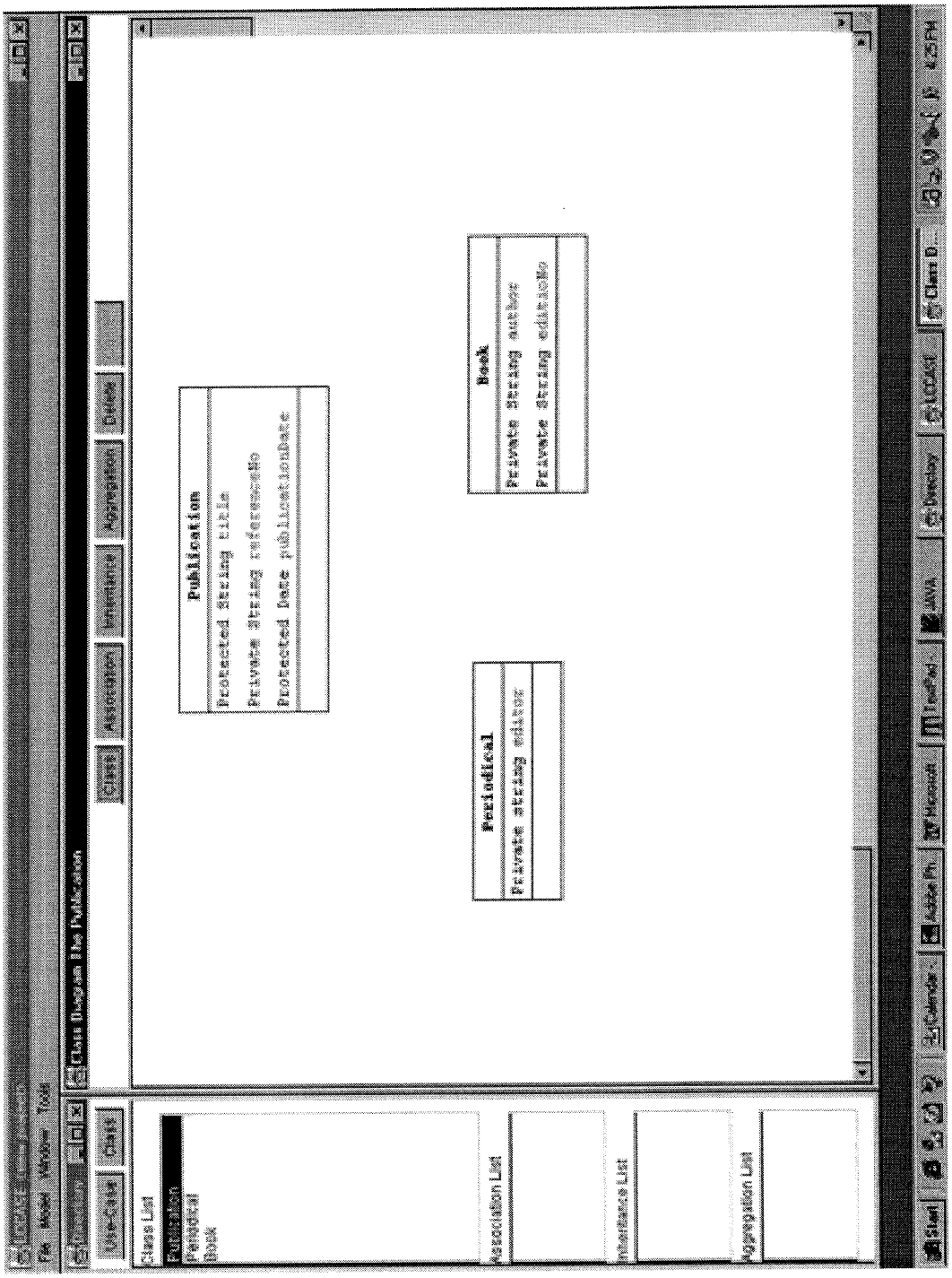


Figure 5-13 The Publication Class Diagram

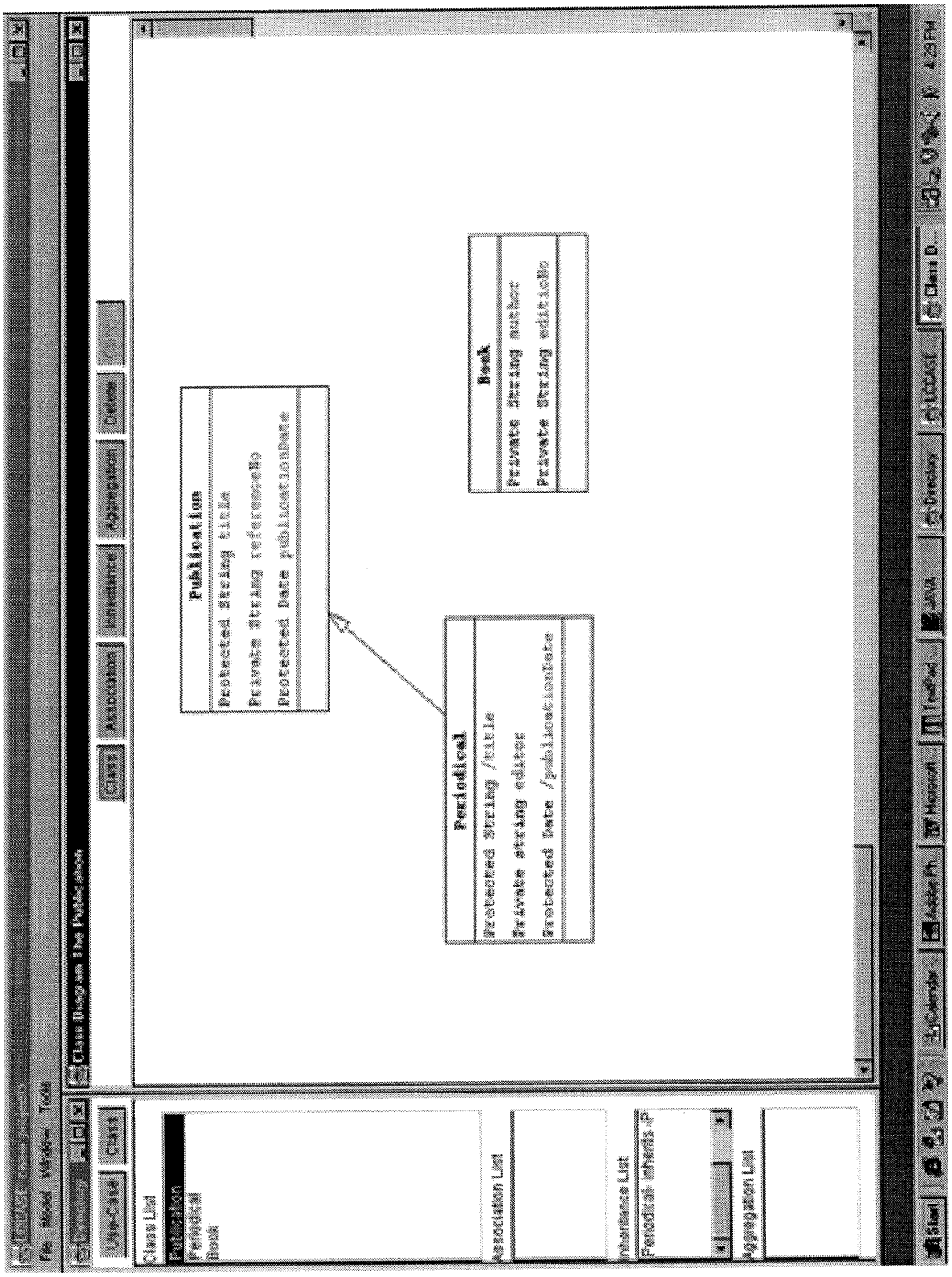


Figure 5-14 The Publication Class Diagram after Creating an Inheritance Relationship

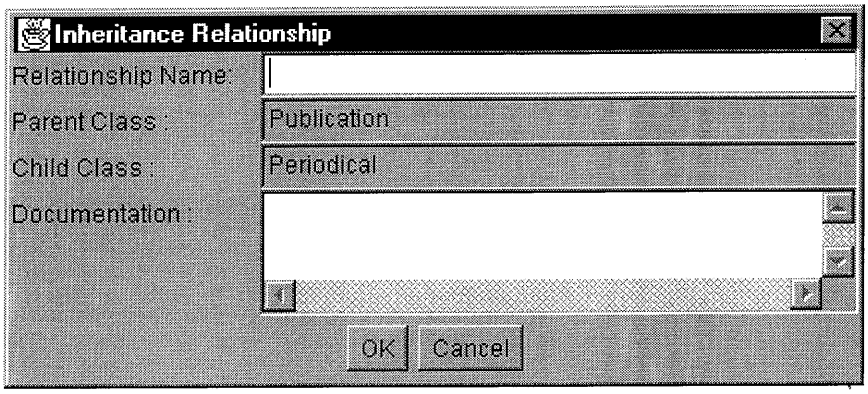


Figure 5-15 The Inheritance Dialog

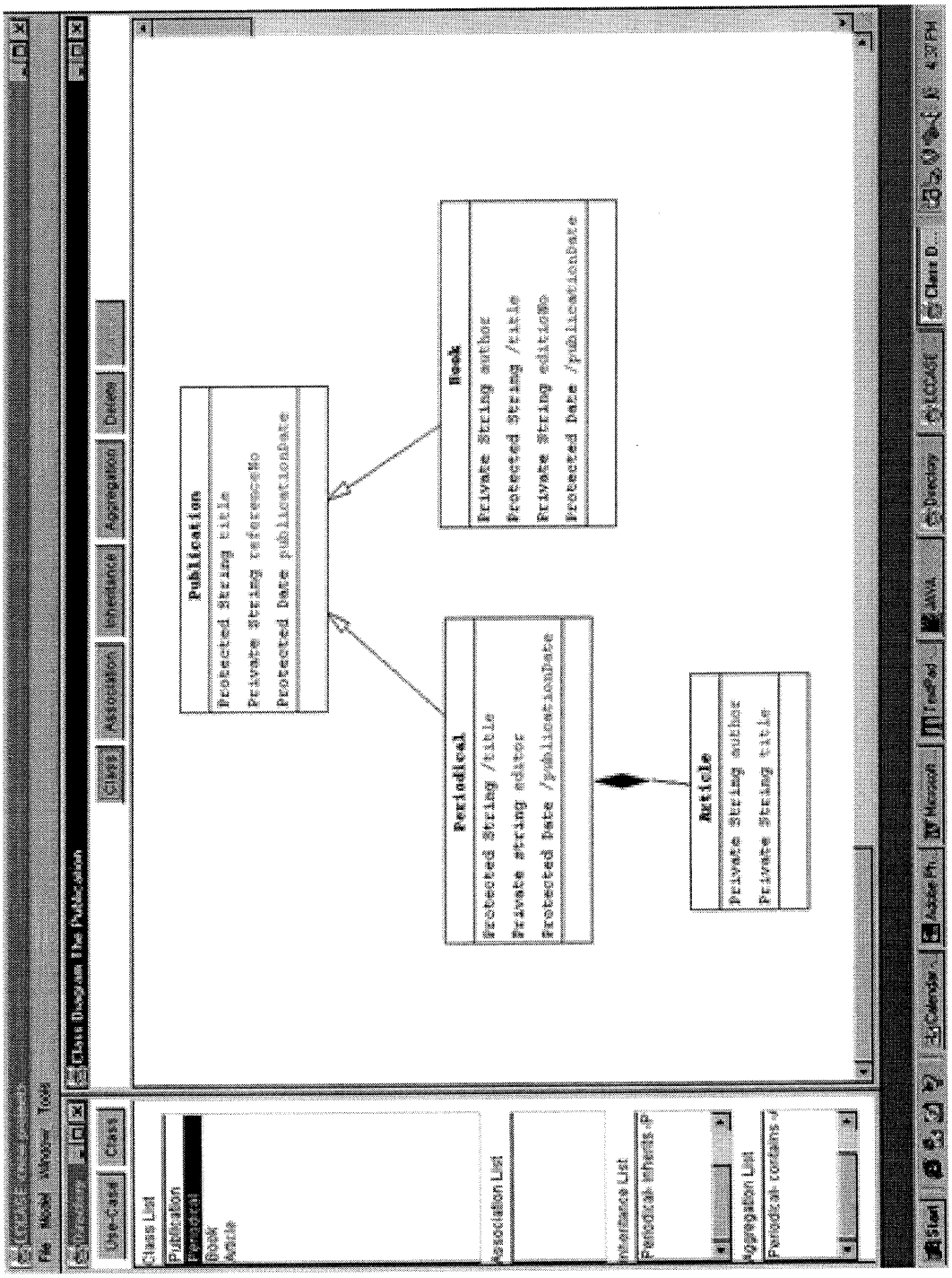


Figure 5-16 The Publication Diagram after Creating an Aggregation Relationship

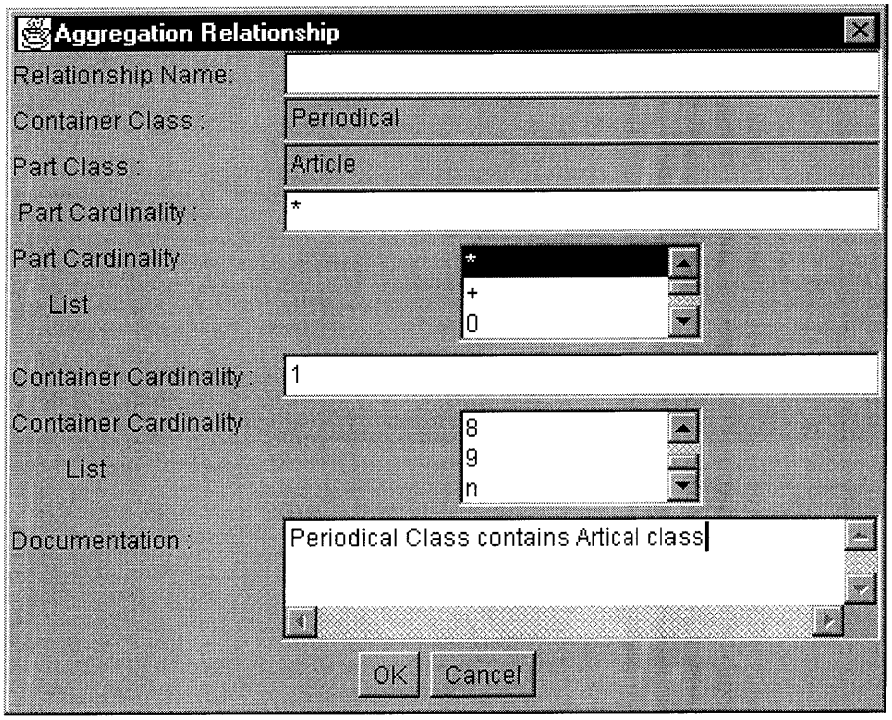


Figure 5-17 The Aggregation Dialog

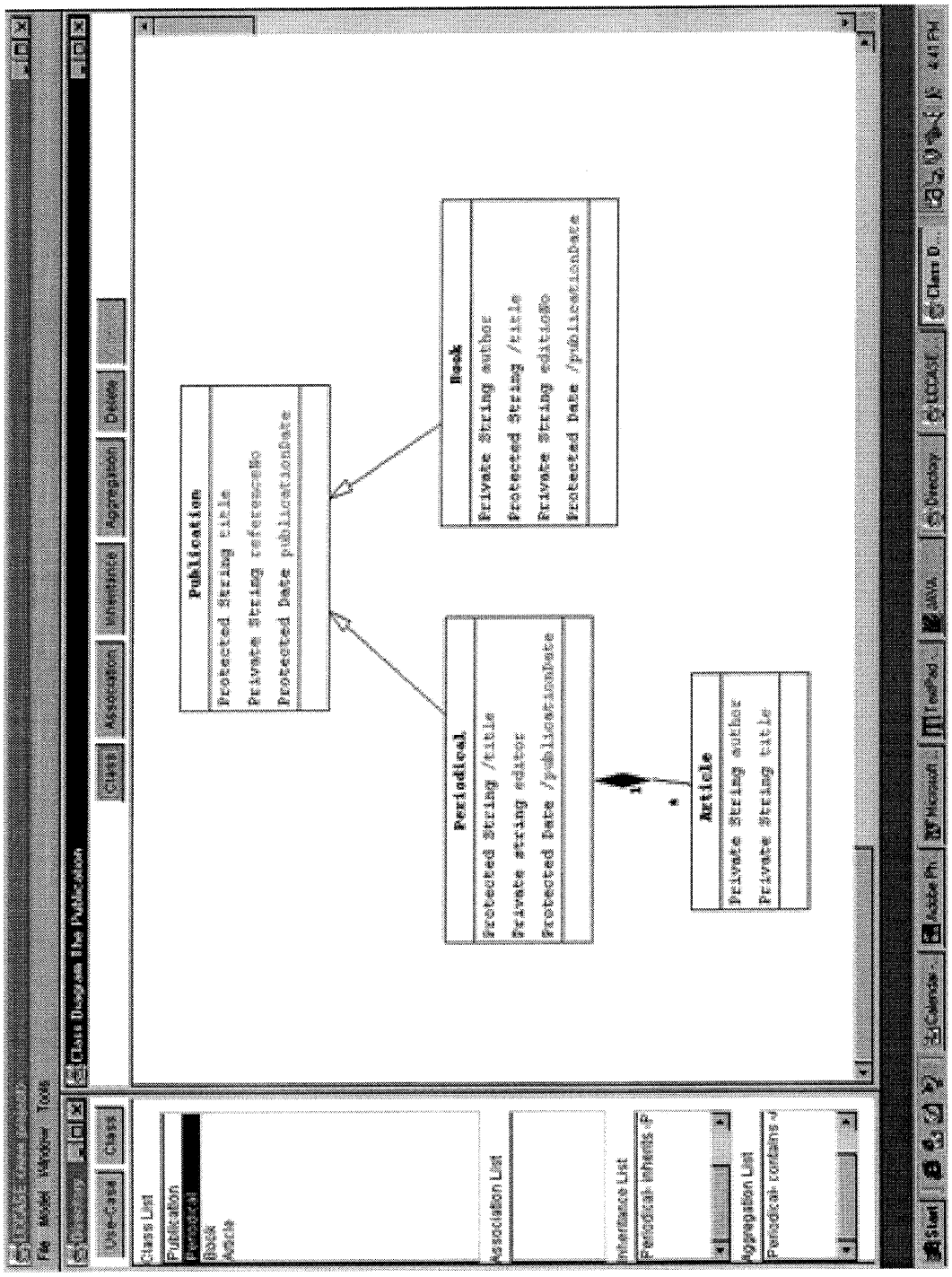


Figure 5-18 The Publication Class Diagram after Modifying the Aggregation Relationship

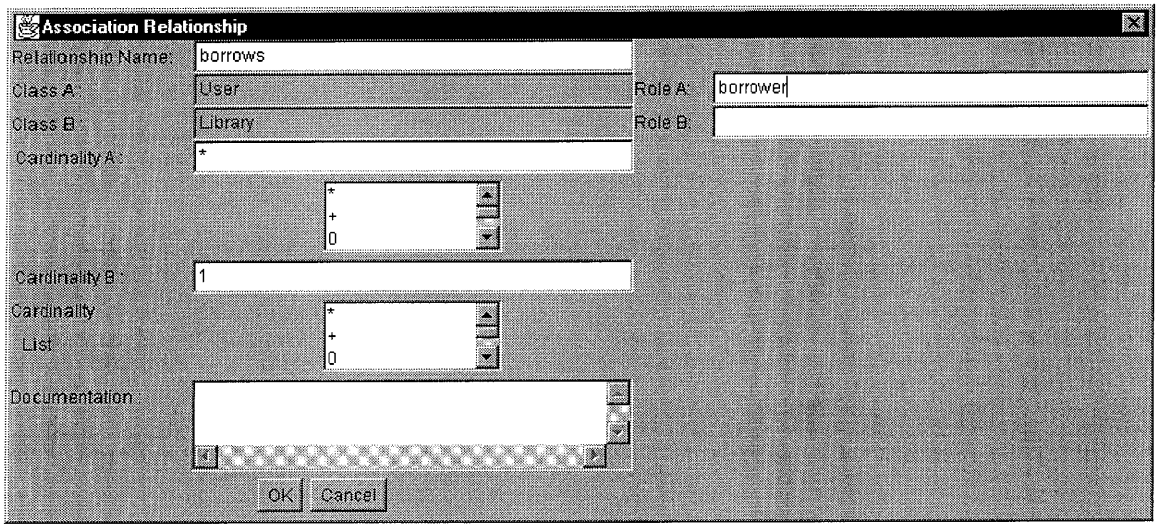


Figure 5-19 The Association Dialog

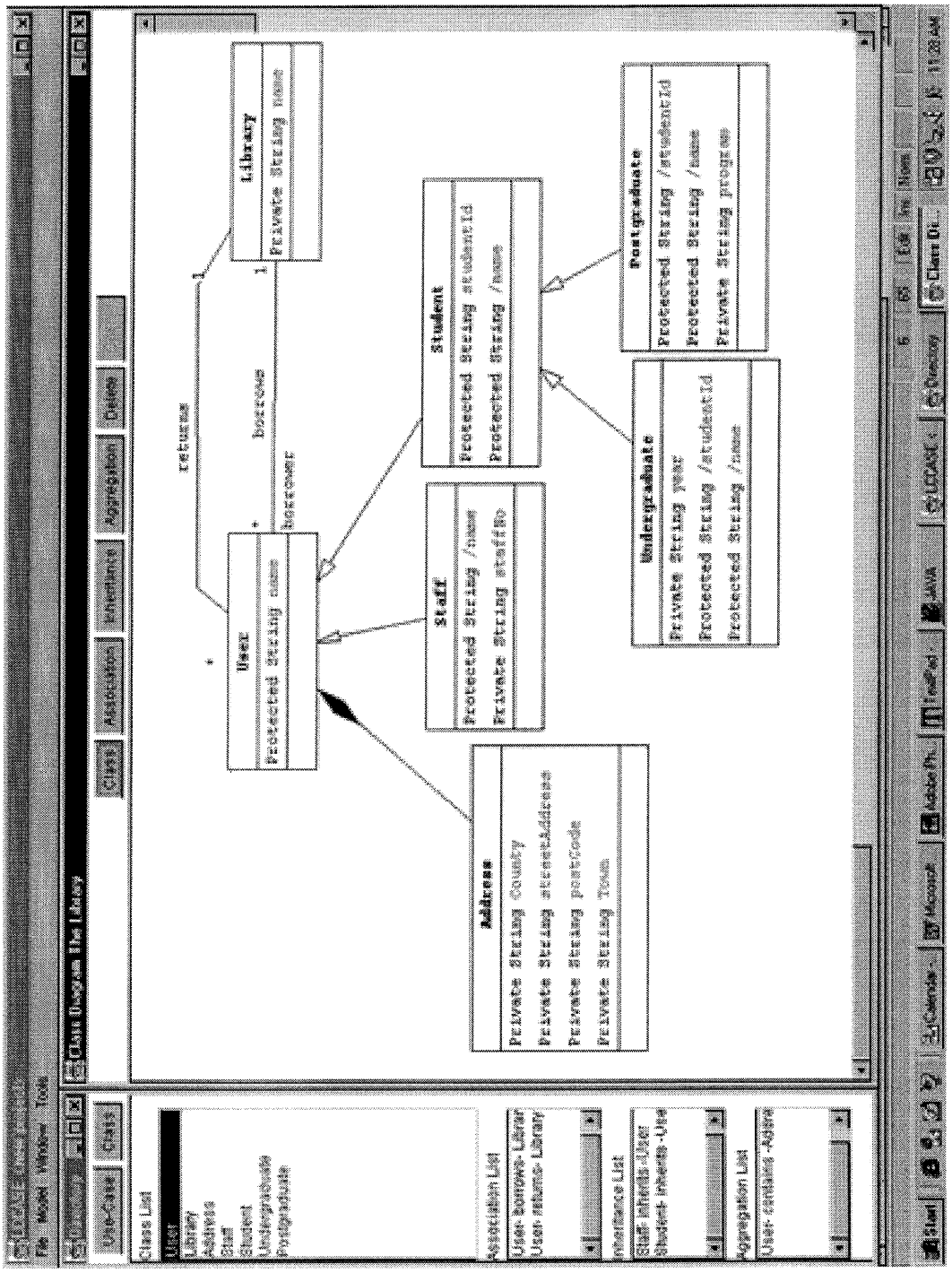


Figure 5-20 The Library Class Diagram

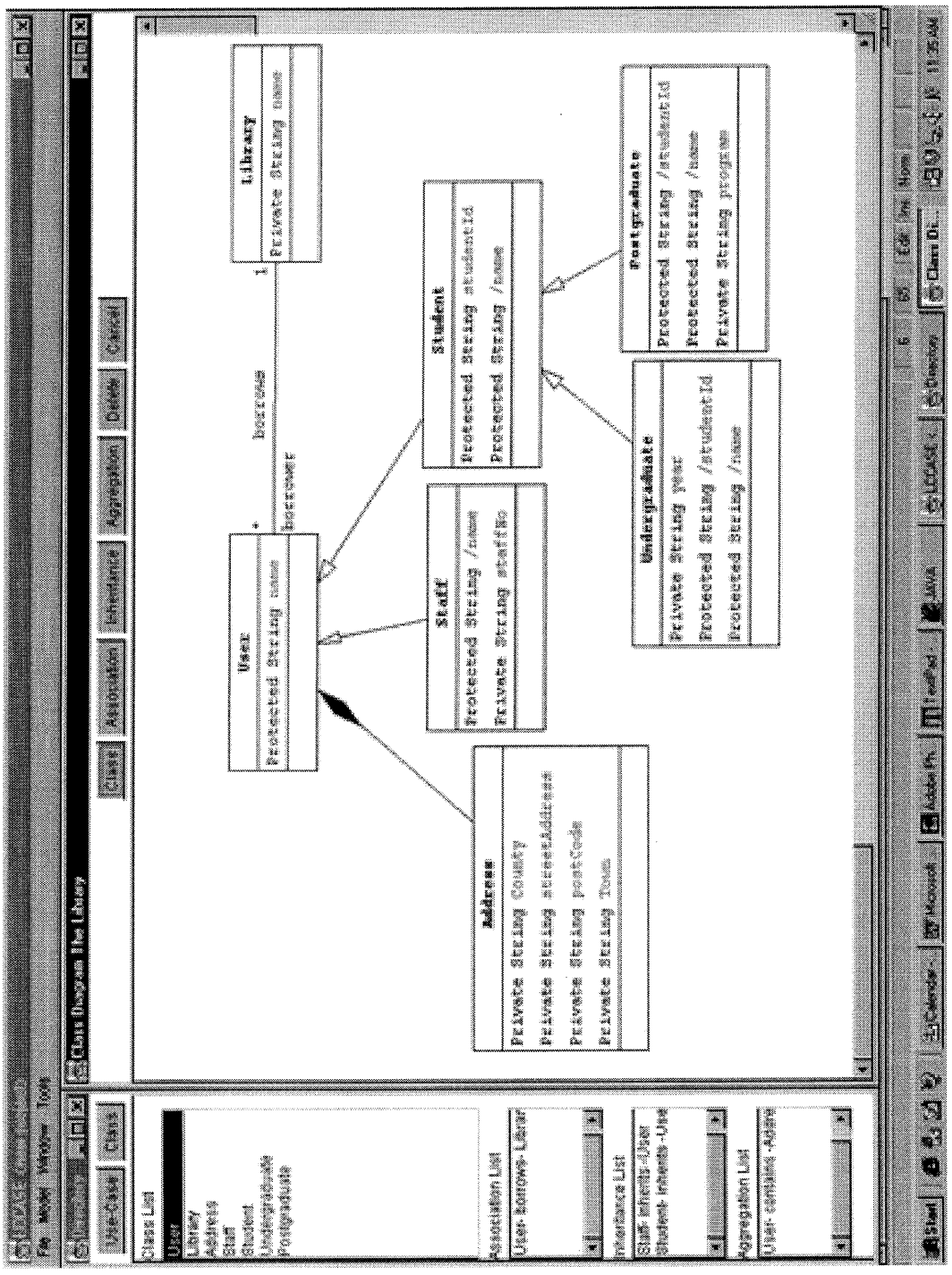


Figure 5-21 The Library Class Diagram after Deleting *returns* Association Relationship

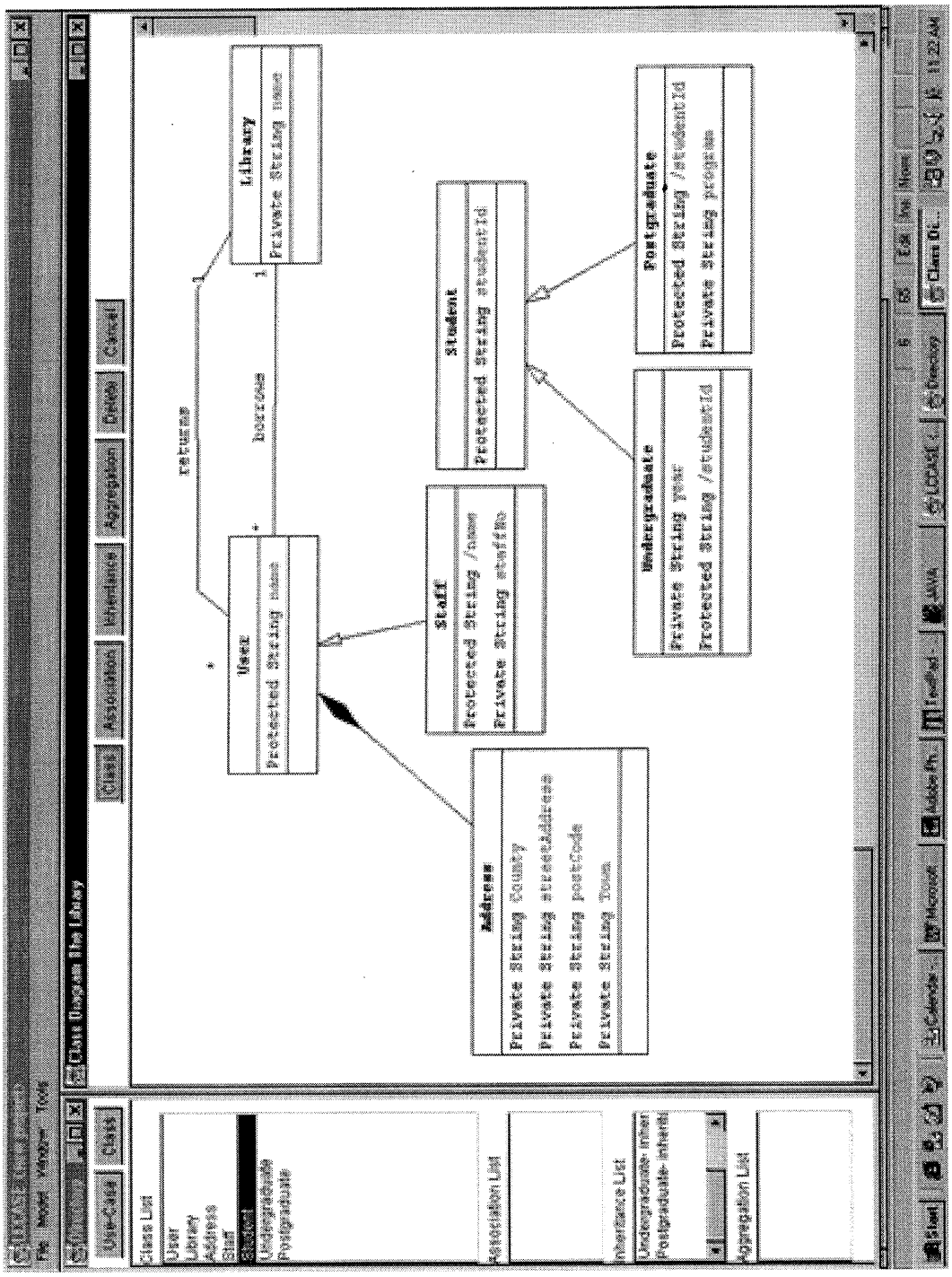


Figure 5-22 The Library Class Diagram after Deleting the Inheritance Relationship between the Classes, *User* and *Student*

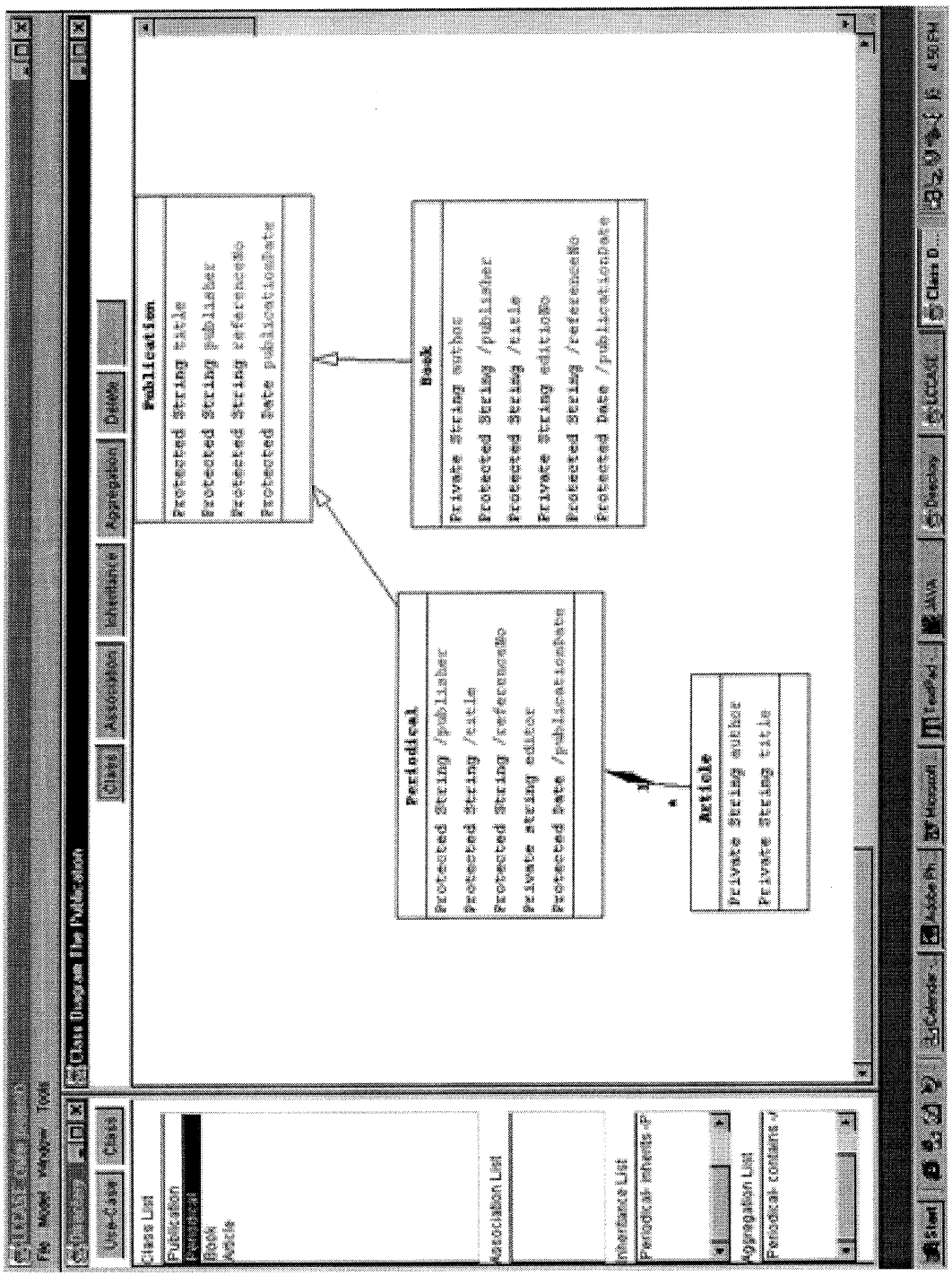


Figure 5-23 The Publication Class Diagram after modifying the *referenceNo* attribute of the *Publication* Class

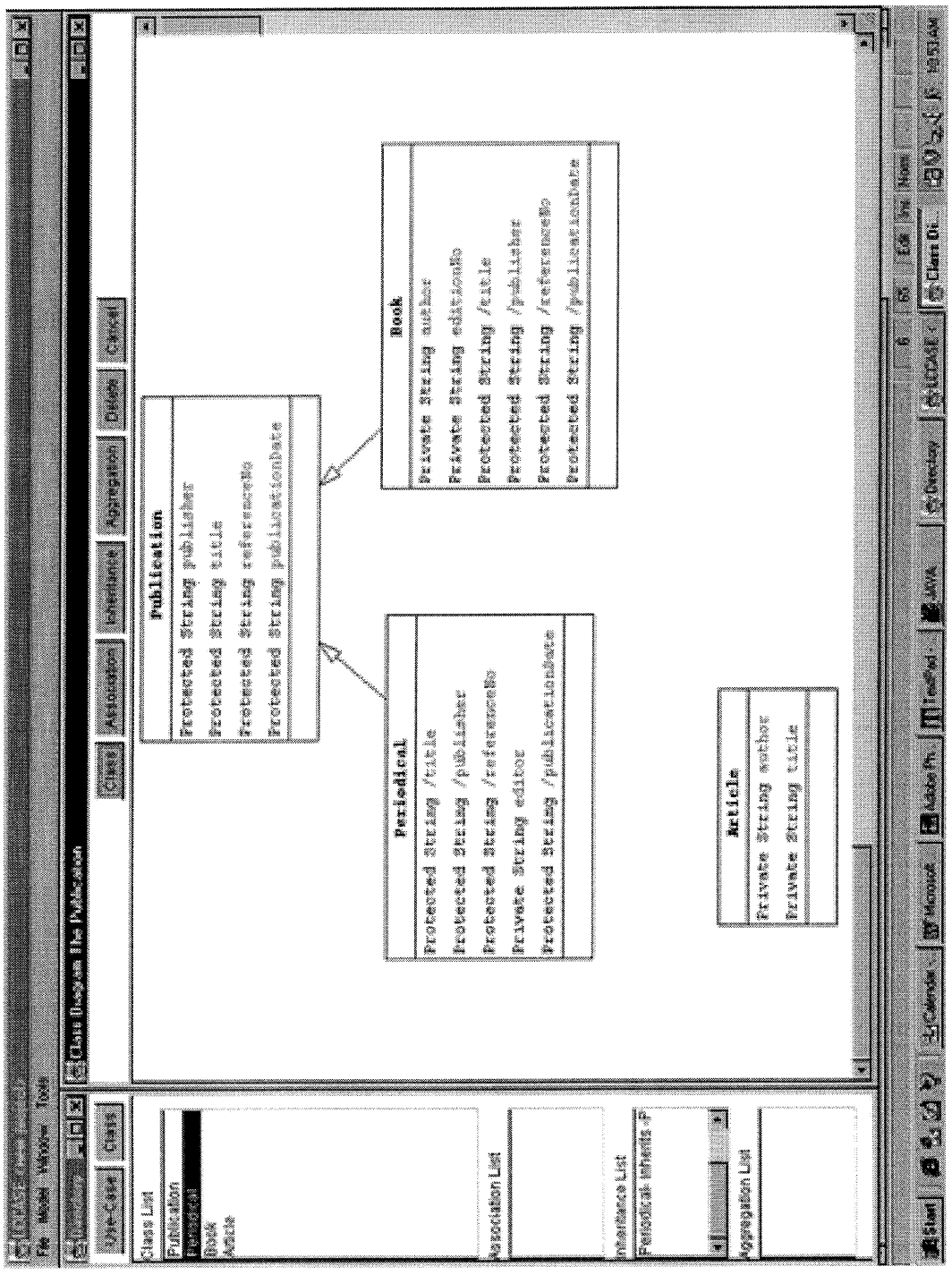


Figure 5-24 The Publication Class Diagram after Deleting the Aggregation Relationship between the Classes, *Periodical* and *Article*

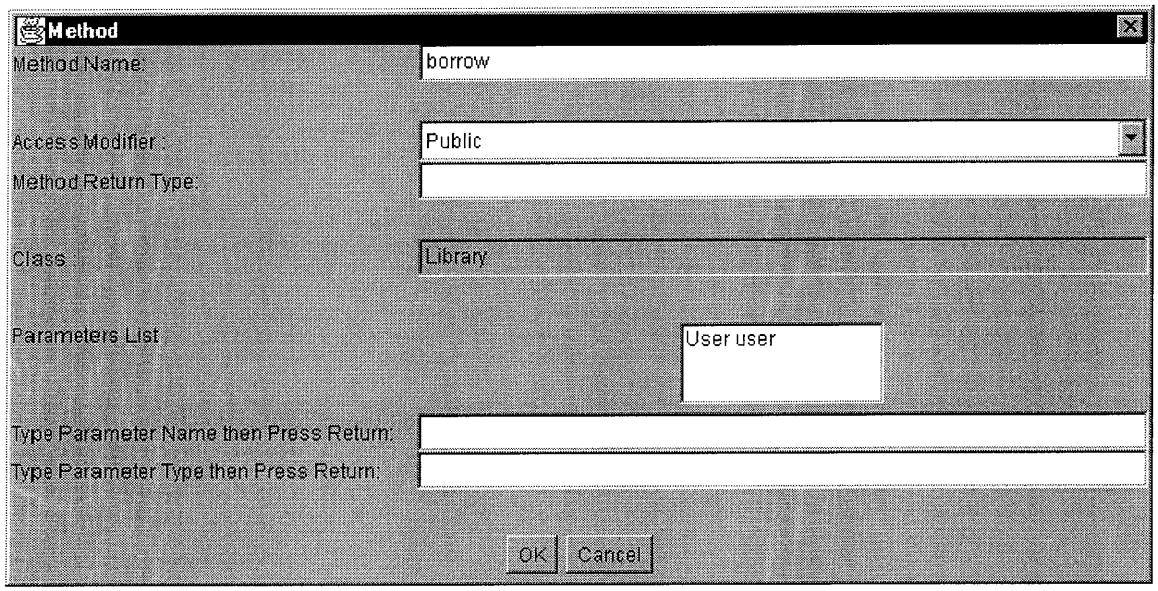


Figure 5-25 The Method Dialog

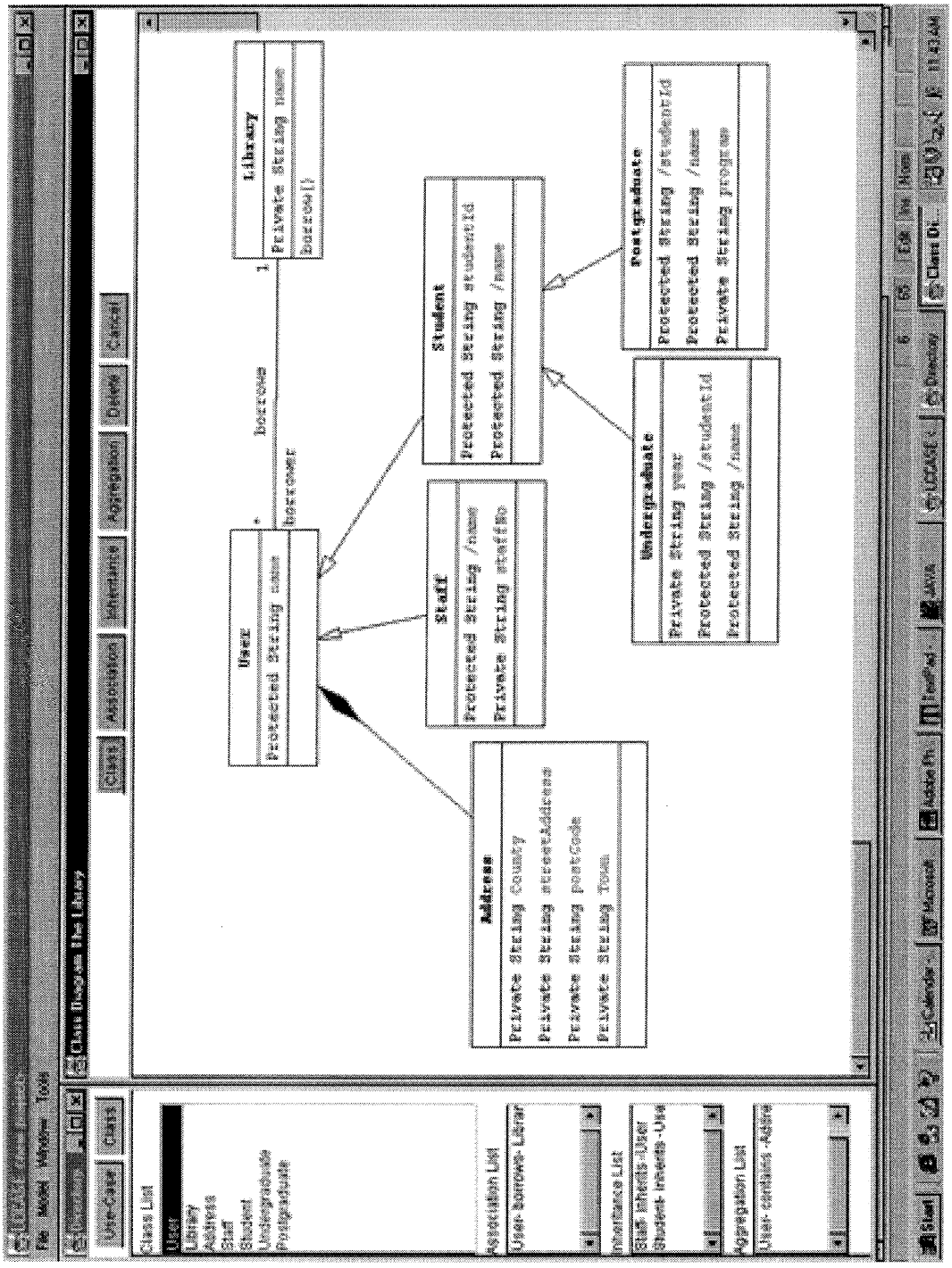


Figure 5-26 The Library Class Diagram after adding the *borrow* Method

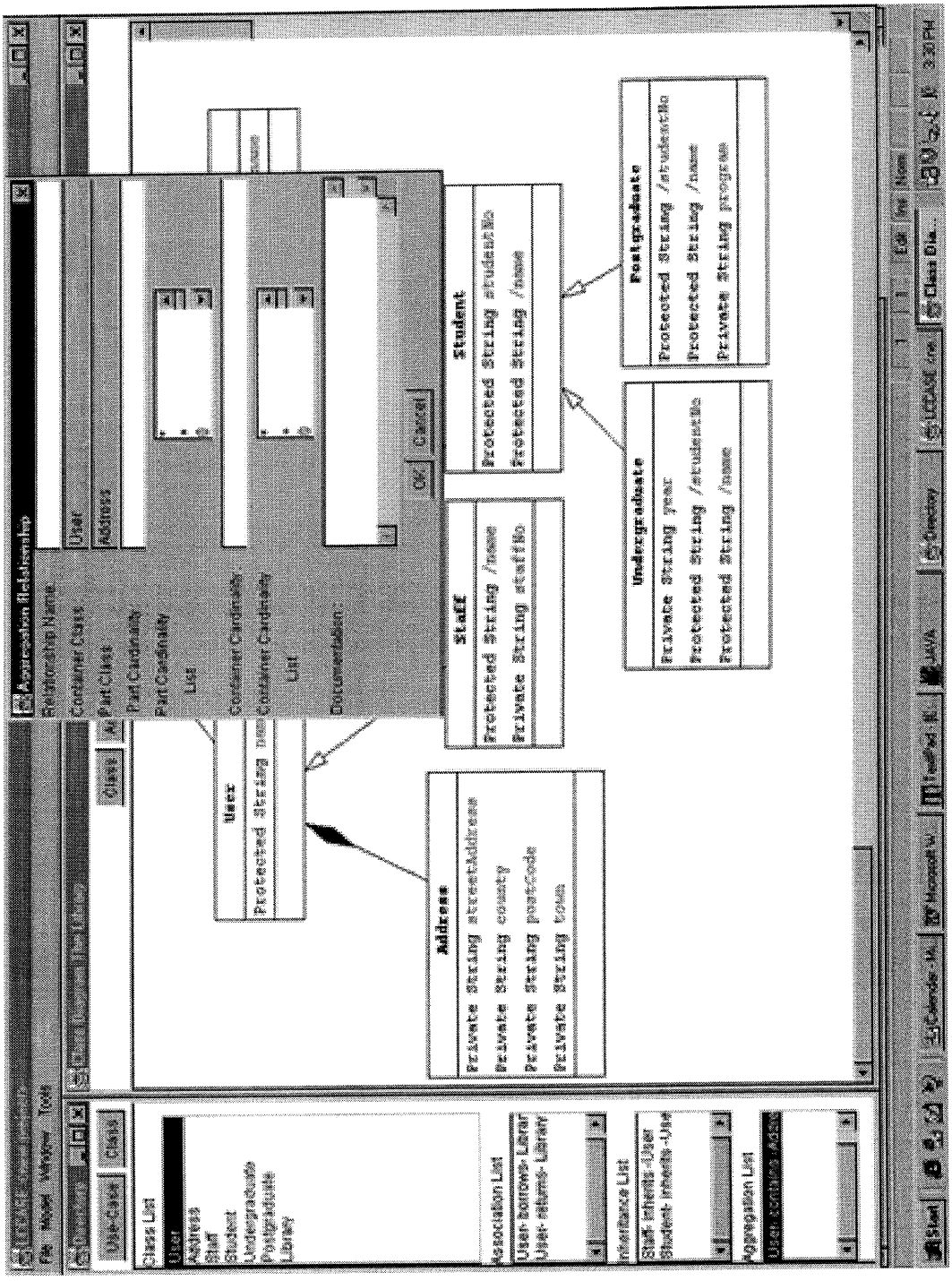


Figure 5-27 Aggregation Relationship Dialog

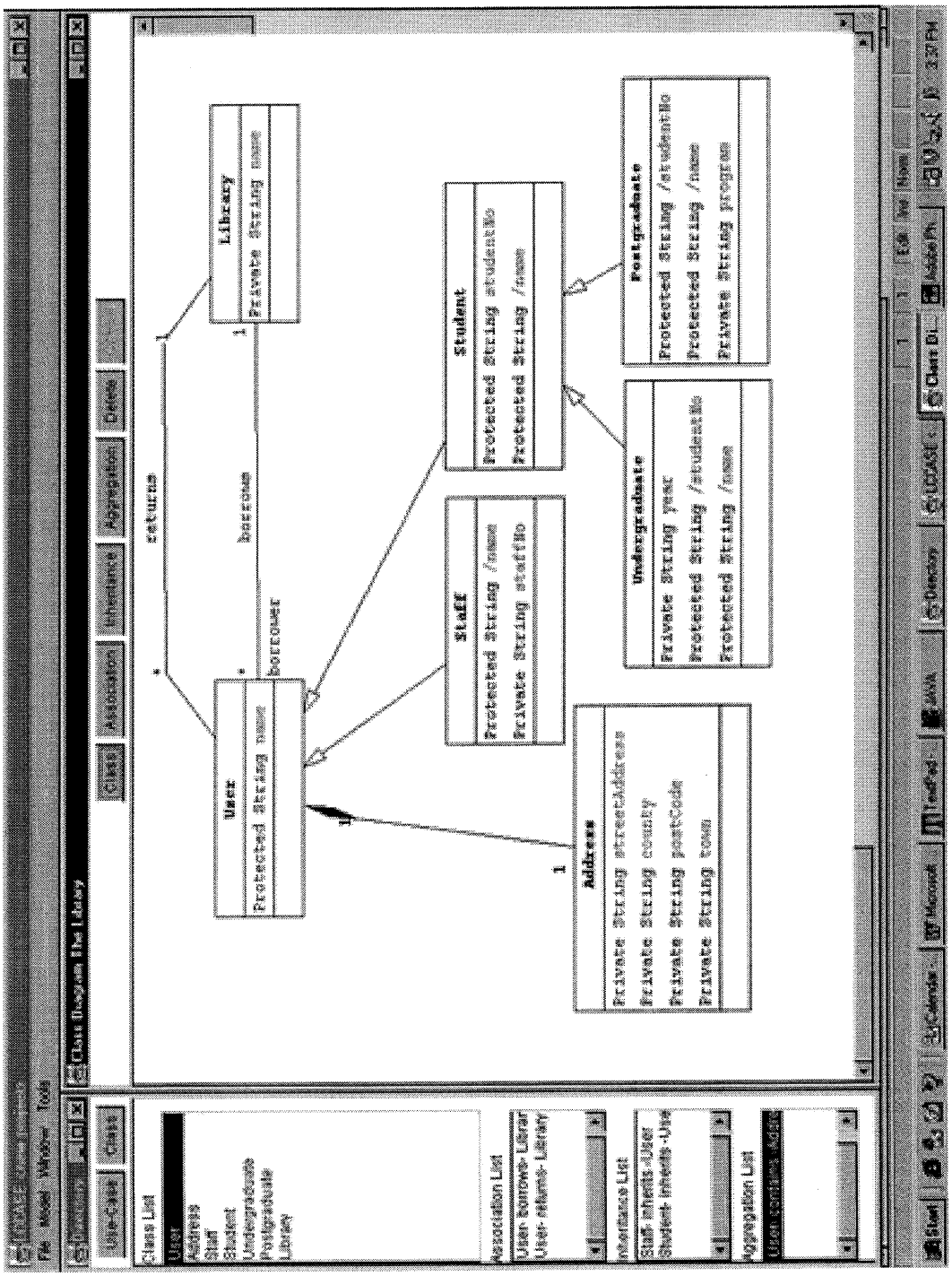


Figure 5-28 The Library Class Diagram after Adding the Cardinalities to the Aggregation Relationship

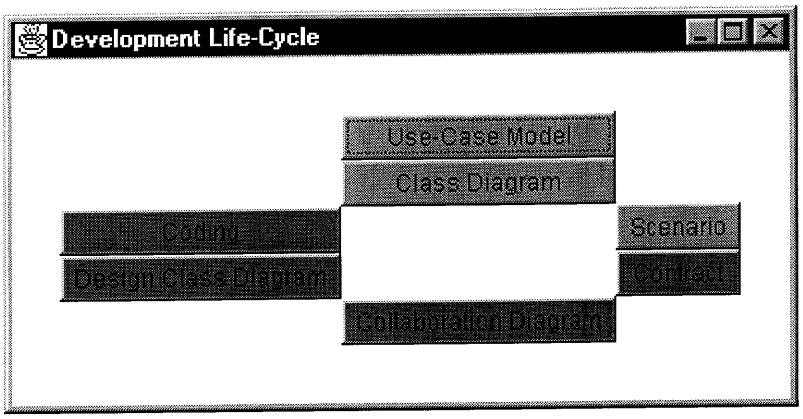


Figure 5-29 The Development Life Cycle after Creating a Use-Case

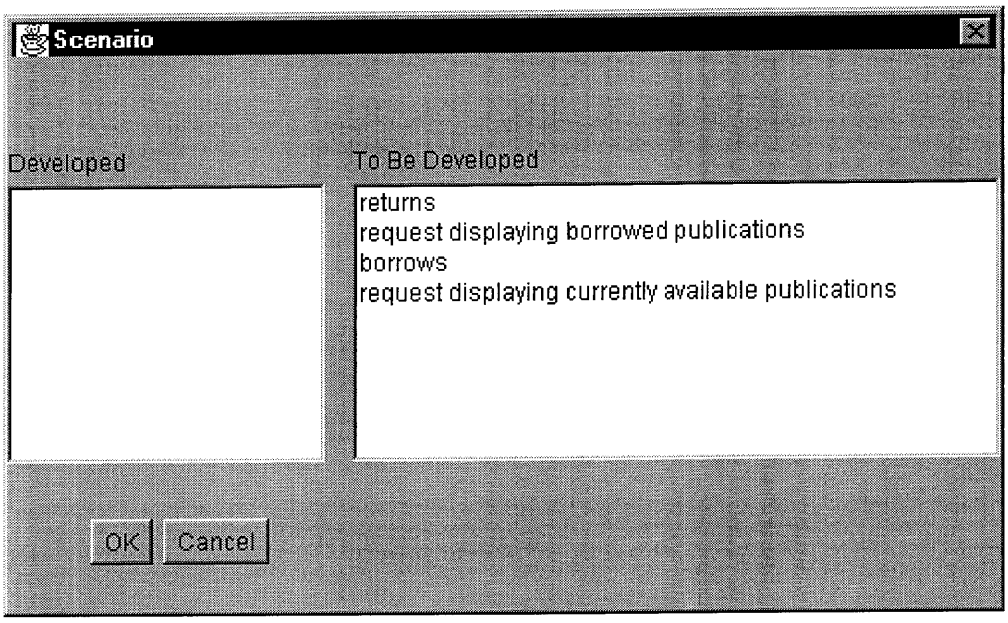


Figure 5-30 The Scenario Status Dialog

5.1.4 The Scenario

A Scenario is a sequence of system operations between actors and the system. Each use-case has its corresponding Scenario. The actors with references to a use-case are the actors in the Scenario.

The LCCASE Tool guides the developer though out the development process of the Scenario. Once the developer decides creating a Scenario, the LCCASE tool provides him with the names of the Scenarios that should be developed. Since the LCCASE tool understands the semantics of the models and links the information between them, it determines the names of the Scenarios from the use-cases as discussed in chapter 4. The flexibility approach of the LCCASE tool discussed in chapter 3 can be noticed clearly in giving the developer the freedom to choose a Scenario from the provided Scenario list or creates a new Scenario. When the developer needs to add an actor to the Scenario, the LCCASE tool provides him with a list of the actors that need to be added to the Scenario. Using the references to the use-case that corresponds to the Scenario, the LCCASE tool determines the actors. The developer has the freedom to choose an actor from the list or add a new actor to the Scenario.

5.1.4.1 The Development of the Borrow Scenario

To develop the Scenario, *borrow*, which shows the interaction between the actor, *User*, and the library system, the developer selects Scenario from the menu bar. The LCCASE tool then displays the Scenario Dialog, which contains a message requesting the developer to choose a Scenario from the lists below or provide a name of a new Scenario,

lists of the names of the use-cases and Scenarios that have been created, an Add Button to add a use-case to the scenario list, a message requesting the developer to enter a name of a new Scenario, and a field to enter the name of a Scenario. When the developer selects the use-case, *borrow*, from the Use-Case list and then clicks on the Add Button, the LCCASE tool adds the Scenario, *borrow*, to the Scenario list. Figure 5-31 shows the Scenario Dialog after adding the name of the Scenario to the Scenario list. Figure 5-32 shows the Borrow Scenario Window.

Adding an Actor

The developer can add an actor to the Scenario by clicking on the Actor Button. When the developer specifies the location of the actor in the diagram, the LCCASE tool displays the Actor Dialog, which contains a message that requests the developer to choose an actor from the list or enter a new actor's name, list of the actors that have reference with the use-case, *borrow*, Add Button to add an actor to the diagram, message to enter new actor name, and a field to enter the actor name. The developer then selects the actor, *User*, from the Actors list and clicks on the Add Button. The LCCASE tool adds the actor to the Actor list in the scenario. Figure 5-33 shows the Actor Dialog after adding the actor, *User*, to the Actor list. Once the developer specifies the actor, the LCCASE tool draws the actor, *User*, box in the diagram. Figure 5-34 shows the window of the Scenario, *borrow*, after adding the actor.

Creating a System Operation

A System Operation can be added to the Scenario by clicking on the System Operation Button. The LCCASE tool draws the system operation line once the developer specified

the actors. When the developer double clicks on the system operation line, the LCCASE tool displays the System Operation Dialog. When the developer enters the name of the system operation, *requestBorrowing*, the LCCASE tool redraws the system operation to write its name above the line. Figure 5-35 shows the system operation.

Deleting a System Operation

Figure 5-36 shows the borrow Scenario window before deleting the system operation, *borrows*, in the Scenario.

The developer deletes a system operation by clicking on the Delete Button, and then specifying the system operation, *borrow*. The LCCASE tool then deletes the system operation from the diagram. Figure 5-37 shows the window of the Scenario after deleting the system operation.

5.1.4.2 Important Features

Creating a Non-Suggested Scenario

The following discusses how to develop a Scenario that has not been suggested by the LCCASE tool.

To develop the Scenario, *addAccount*, which shows the interaction between the actor, *staff*, and the library system, the developer selects Scenario from the menu bar. The LCCASE tool then displays the Scenario Dialog. Since the developer did not create the use-case, *addAccount*, the LCCASE tool did not display *addAccount* within the suggested Scenarios list. The developer enters the name of the Scenario. The LCCASE tool then displays a message stating that the use-case, *addAccount*, has not been created in the

project, and that if the developer insists on creating the Scenario, *addAccount*, the LCCASE tool will create the use-case, *addAccount*, in a new Use-Case Model bearing the same name. Figure 5-38 shows the Scenario Dialog with the previous message. The LCCASE tool then creates the Scenario, *addAccount*. Figure 5-39 shows the addAccount Scenario window. This illustrates how the LCCASE tool makes sure that all the information in the project is consistent.

Adding a Non-Suggested Actor

The following illustrates how the developer adds the actor, *staff*, which is not in the actor's suggested list.

When the developer specifies the location of the actor in the diagram, the LCCASE tool displays the Actor Dialog with an empty actor list. The developer enters the name of the actor, *staff*. The LCCASE tool then displays a message, which states that the actor, *staff*, is not created in the project and if the developer insists on adding the actor, the LCCASE tool will create the actor, *staff*, along with a reference from the actor to the use-case, *addAccount*. Figure 5-40 shows the Actor Dialog with the previous message. Figure 5-41 shows the result of adding the actor, *staff*, to the addAccount Scenario. Figure 5-42 shows the addAccount Use-Case Model, which is created by the LCCASE tool as a result of the previously discussed operations.

5.1.4.3 The Role of the Directory

Figure 5-36 shows the effects of selecting the use-case, *borrow*, from the use-case list in the Directory. Figure 5-37 shows the effect of deleting system operation, *borrow,* from

the Scenario. The System Operation list in the Directory shows the system operation, *requestBorrowing*, but the system operation, *borrow*, has been removed.

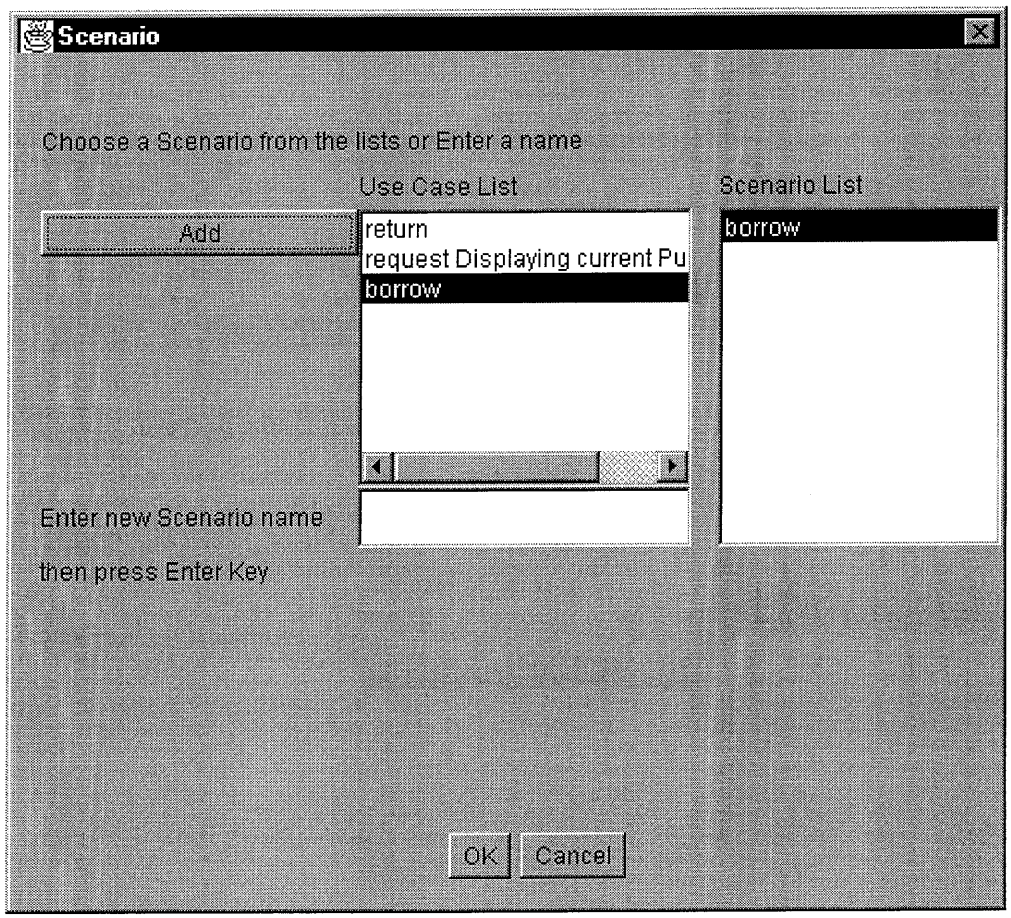


Figure 5-31 The Scenario Dialog after Selecting the *borrow* Use-Case

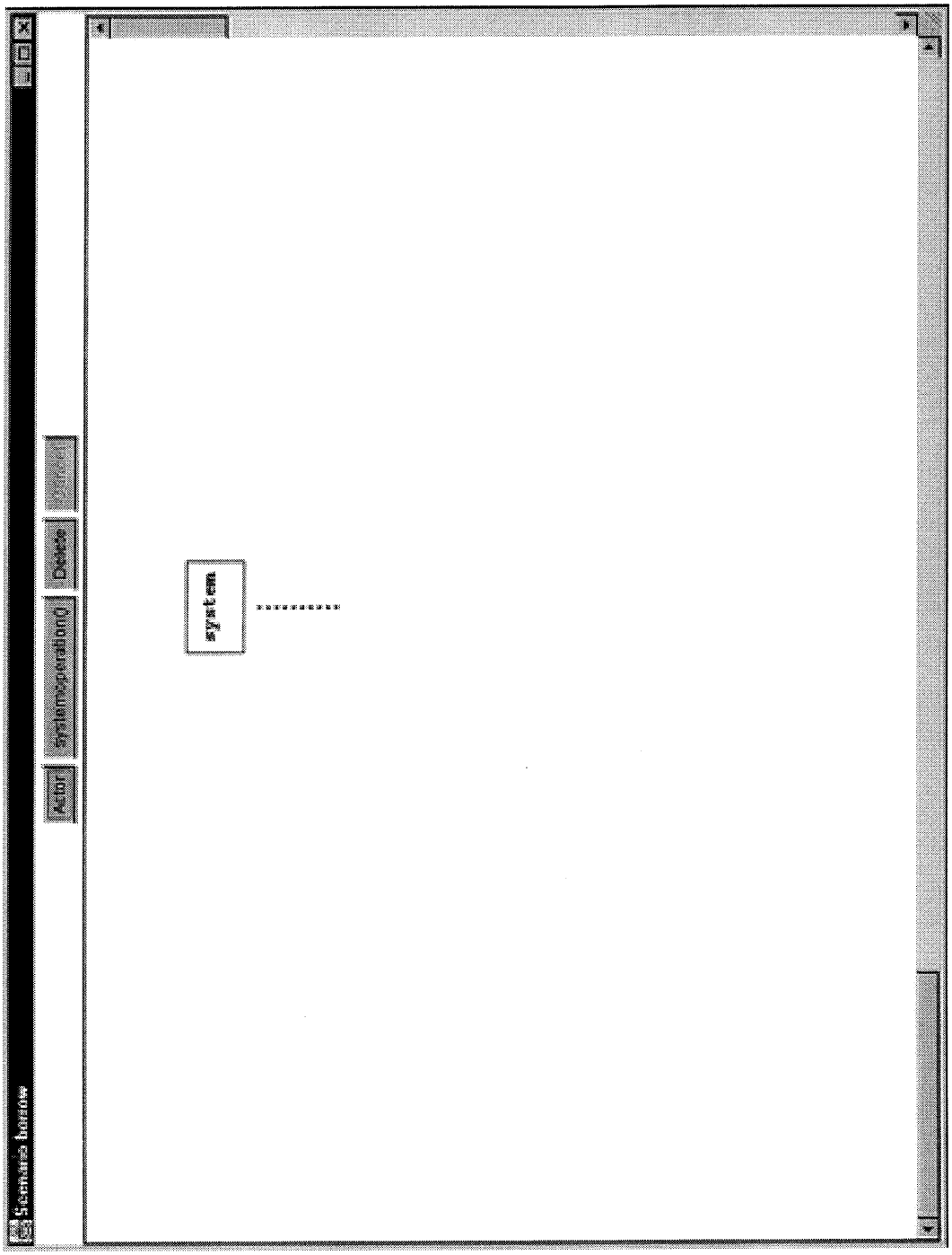


Figure 5-32 The borrow Scenario

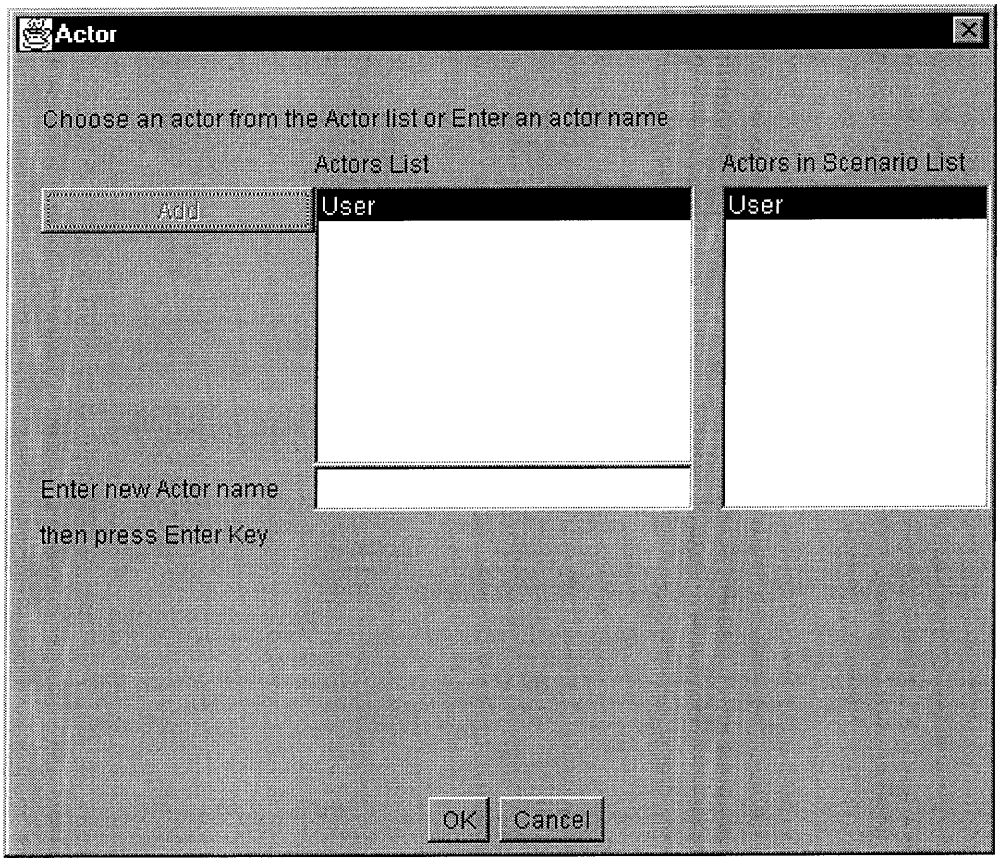


Figure 5-33 The Actor Dialog

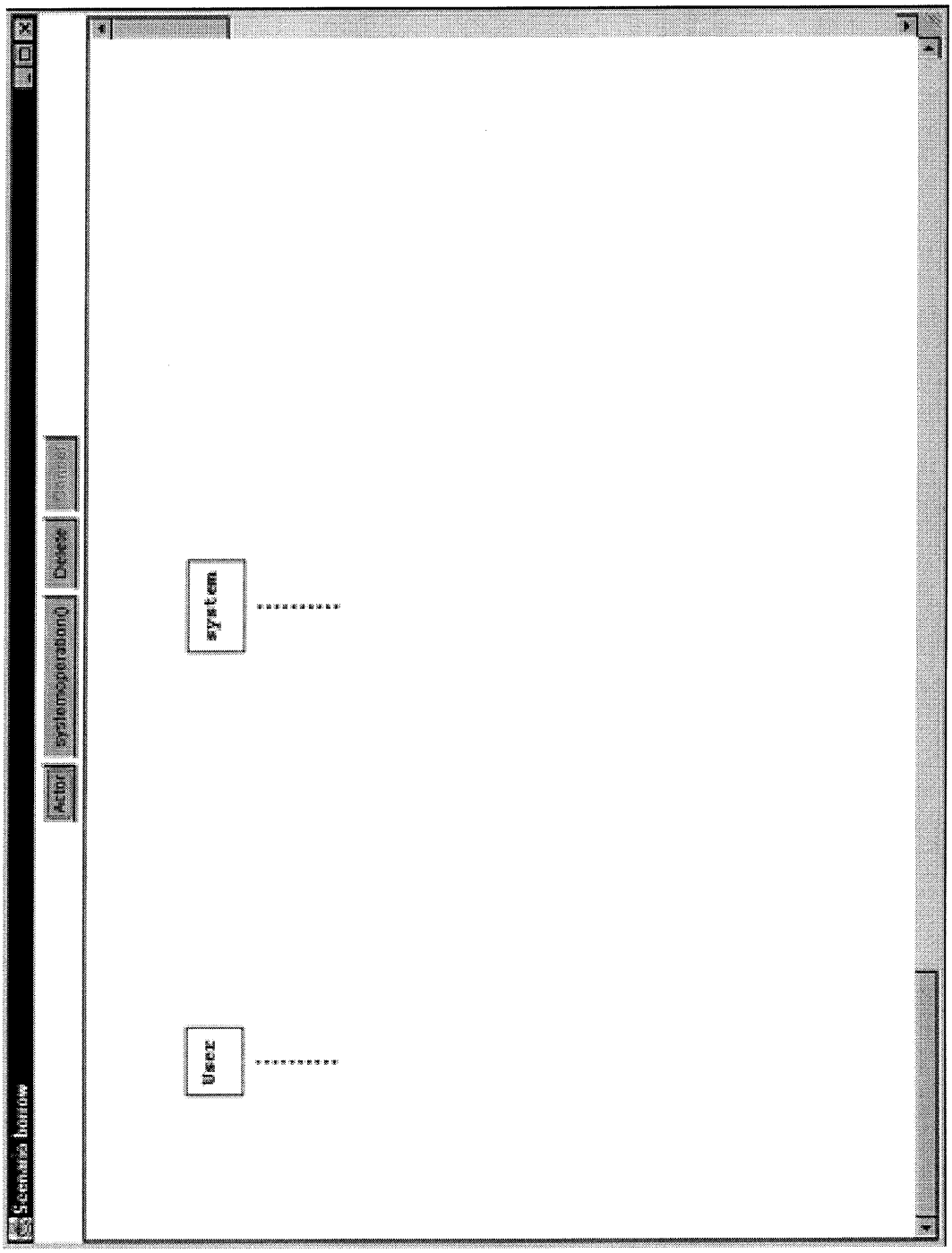


Figure 5-34 The borrow Scenario after Adding the *User* Actor

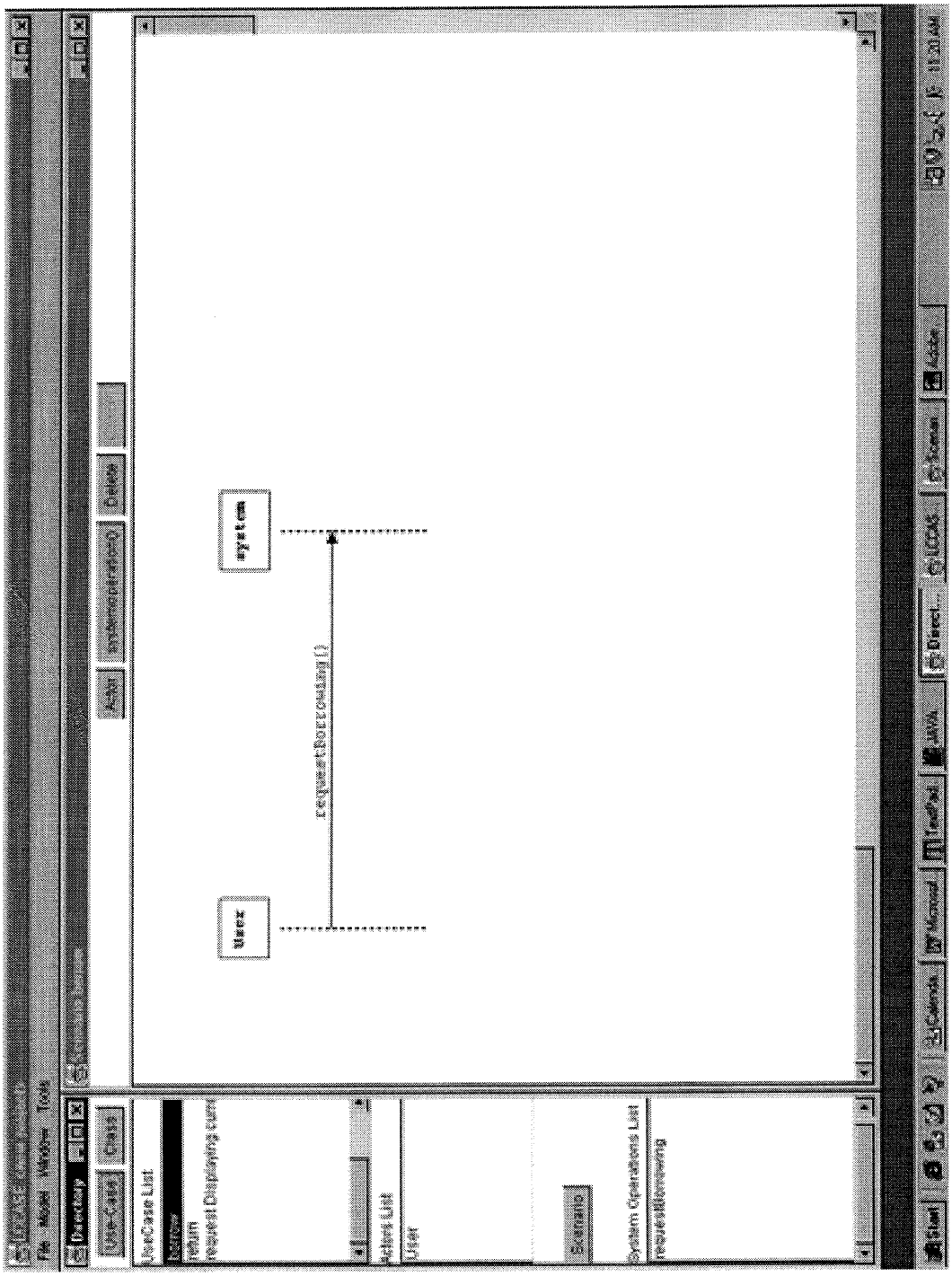


Figure 5-35 The borrow Scenario after Creating the *requestBorrowing* System Operation

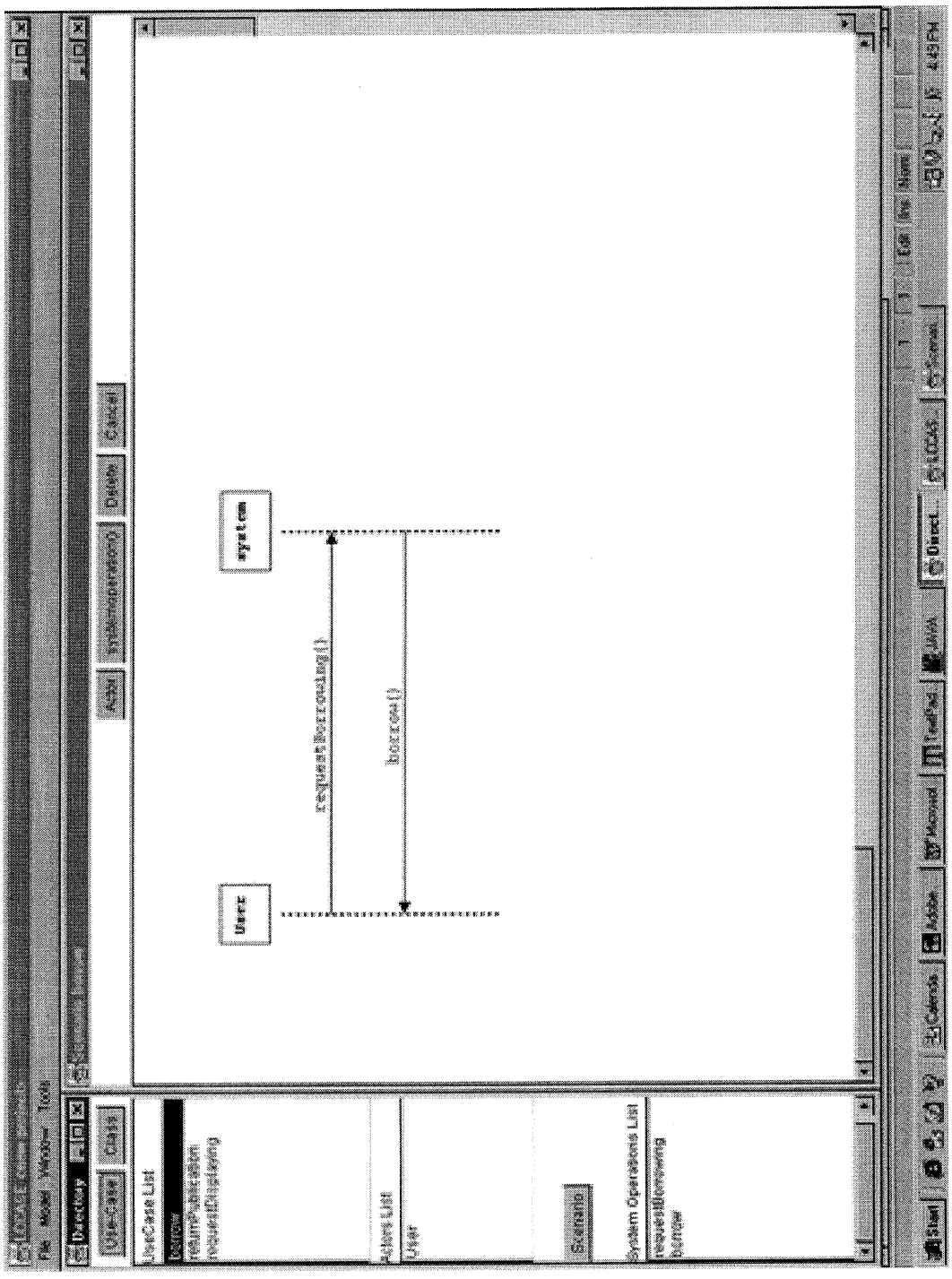


Figure 5-36 The borrow Scenario

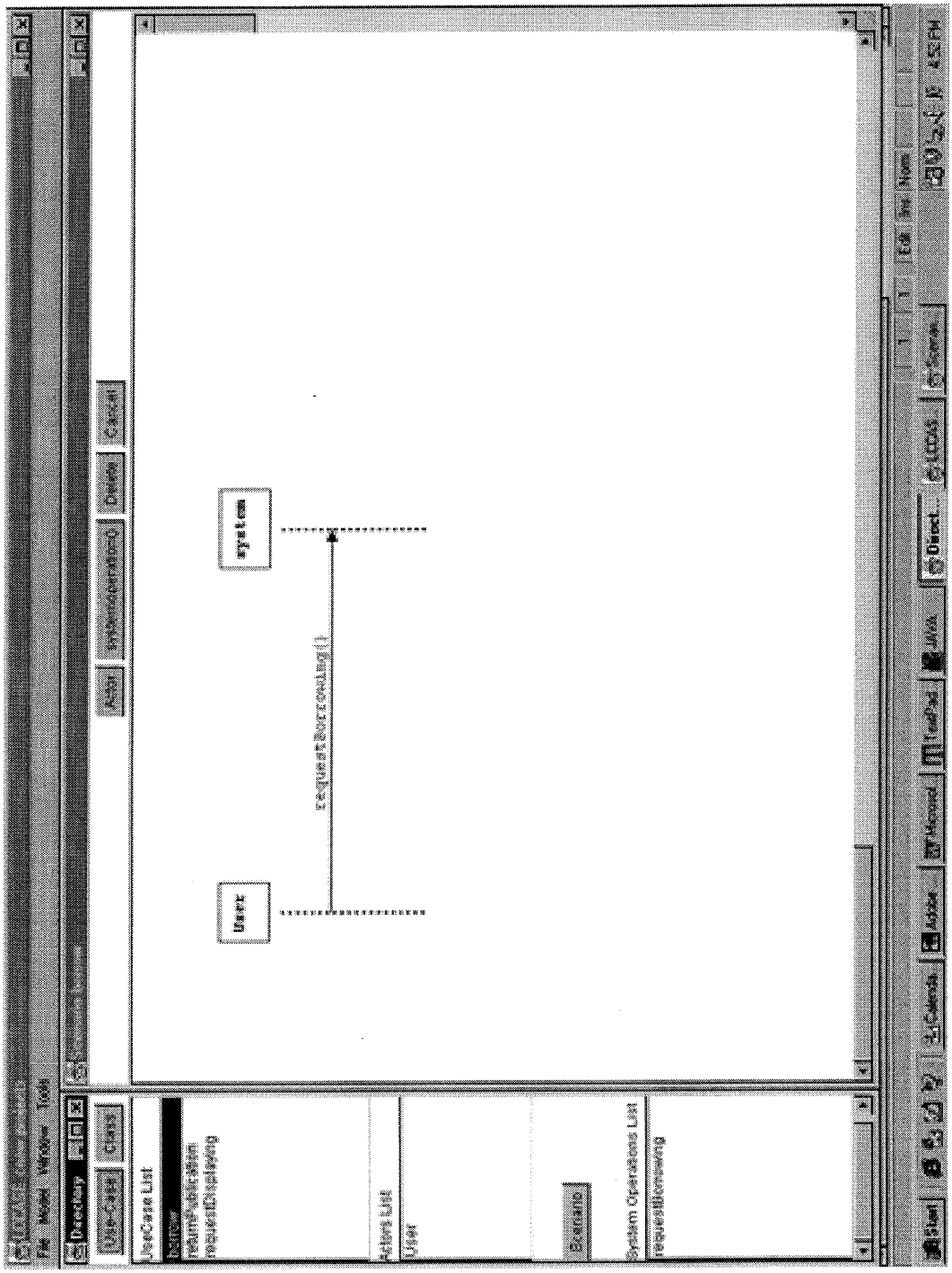


Figure 5-37 The borrow Scenario after Deleting the borrow System Operation

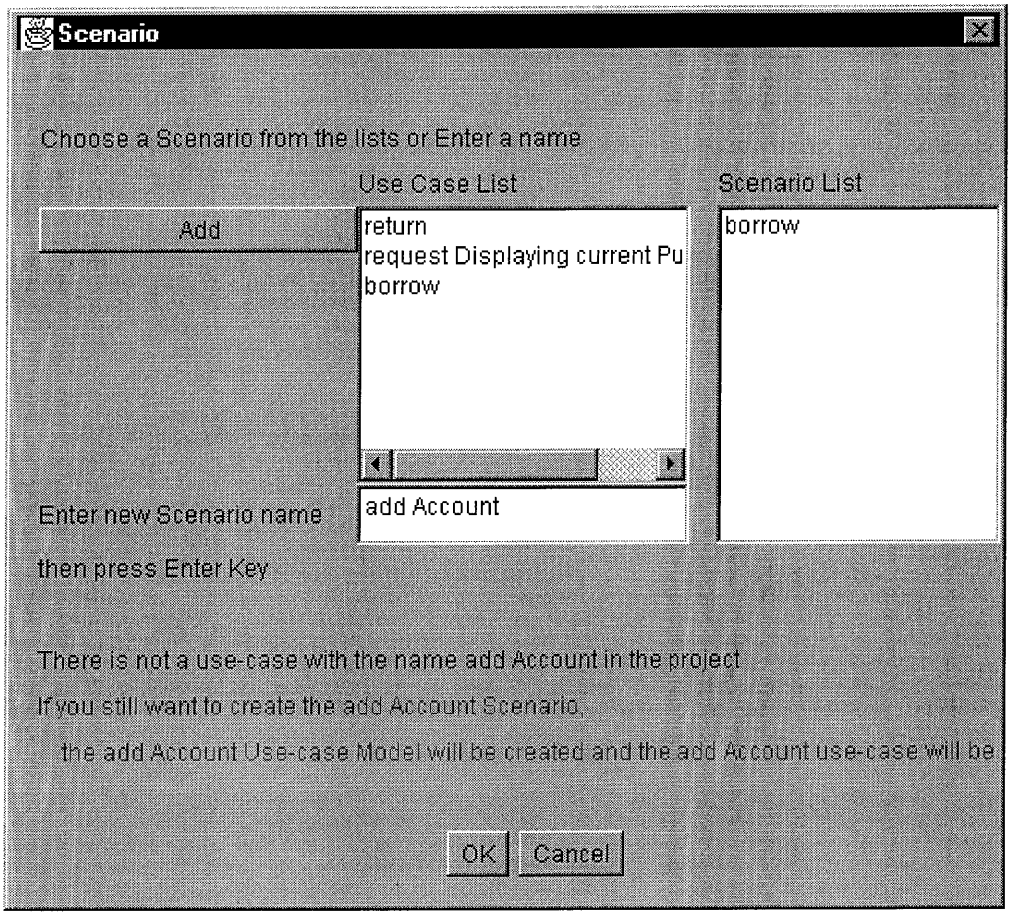


Figure 5-38 The Scenario Dialog

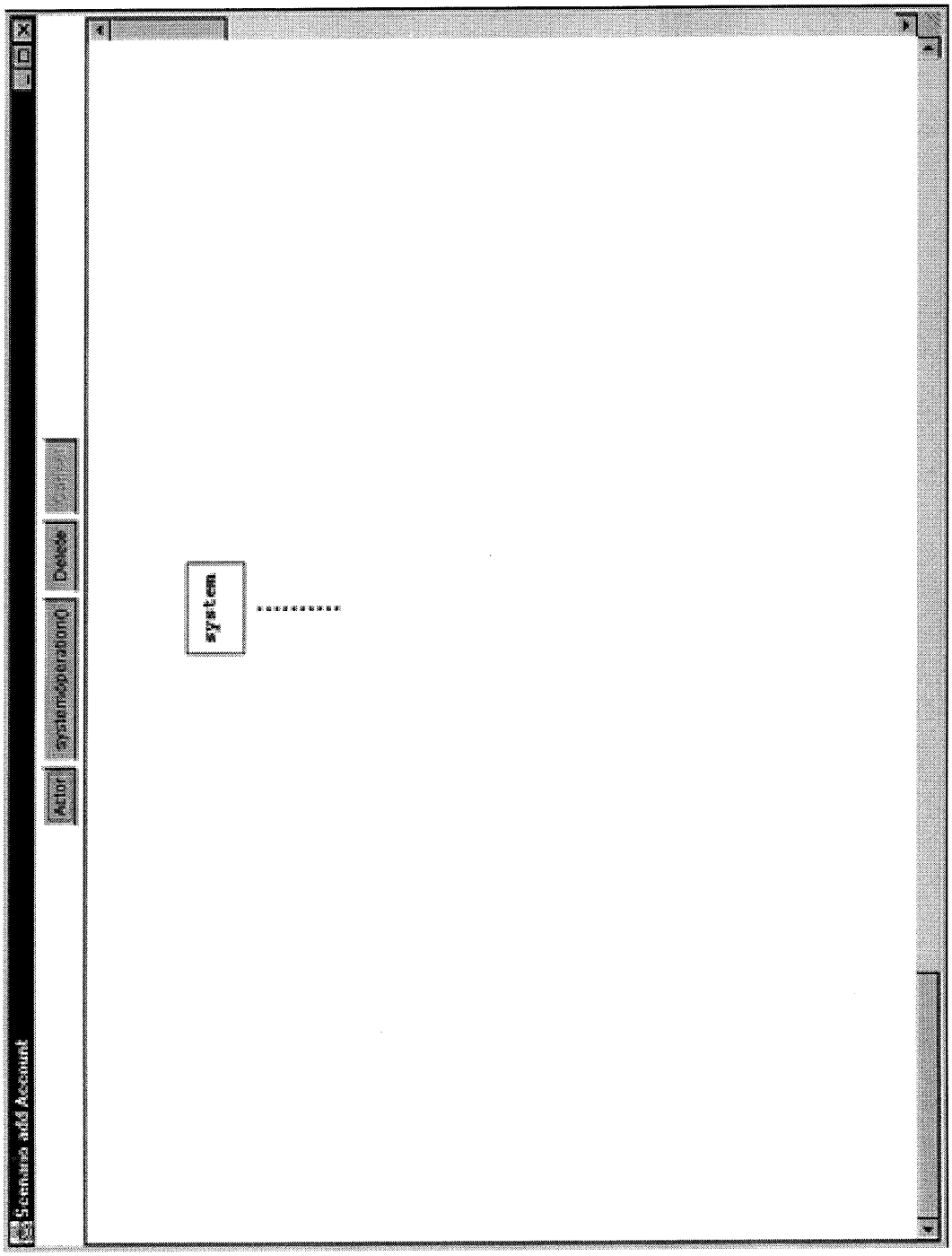


Figure 5-39 The AddAccount Scenario

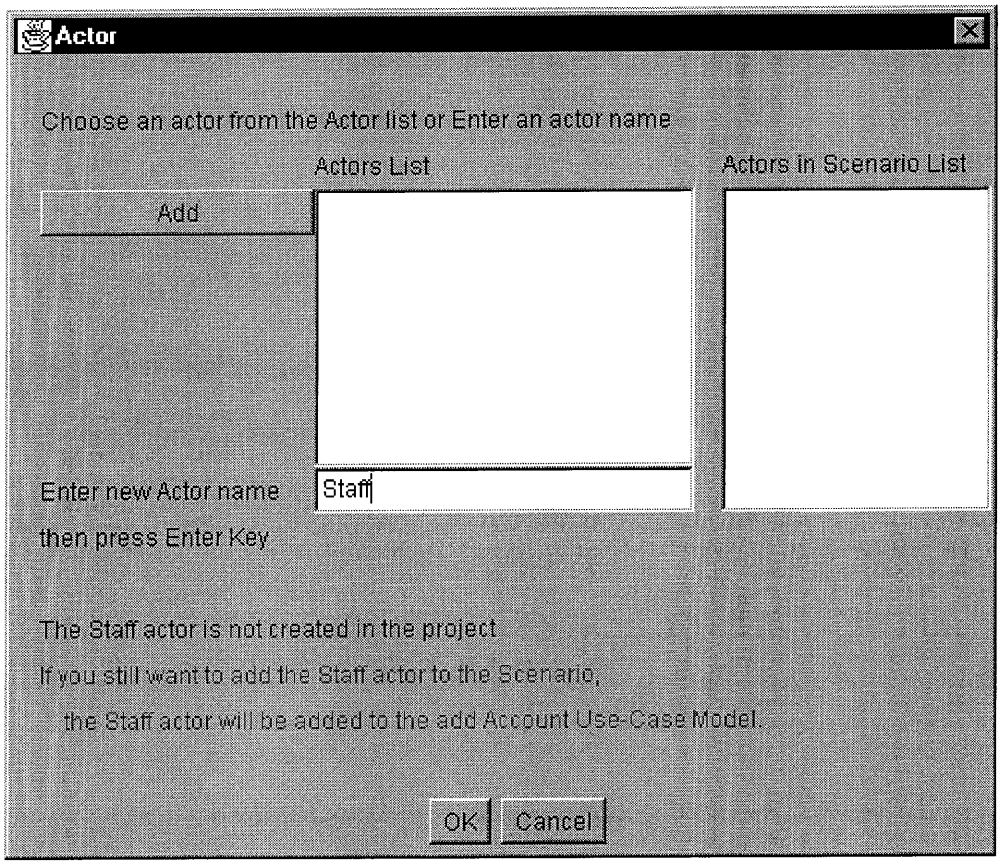


Figure 5-40 The Actor Dialog

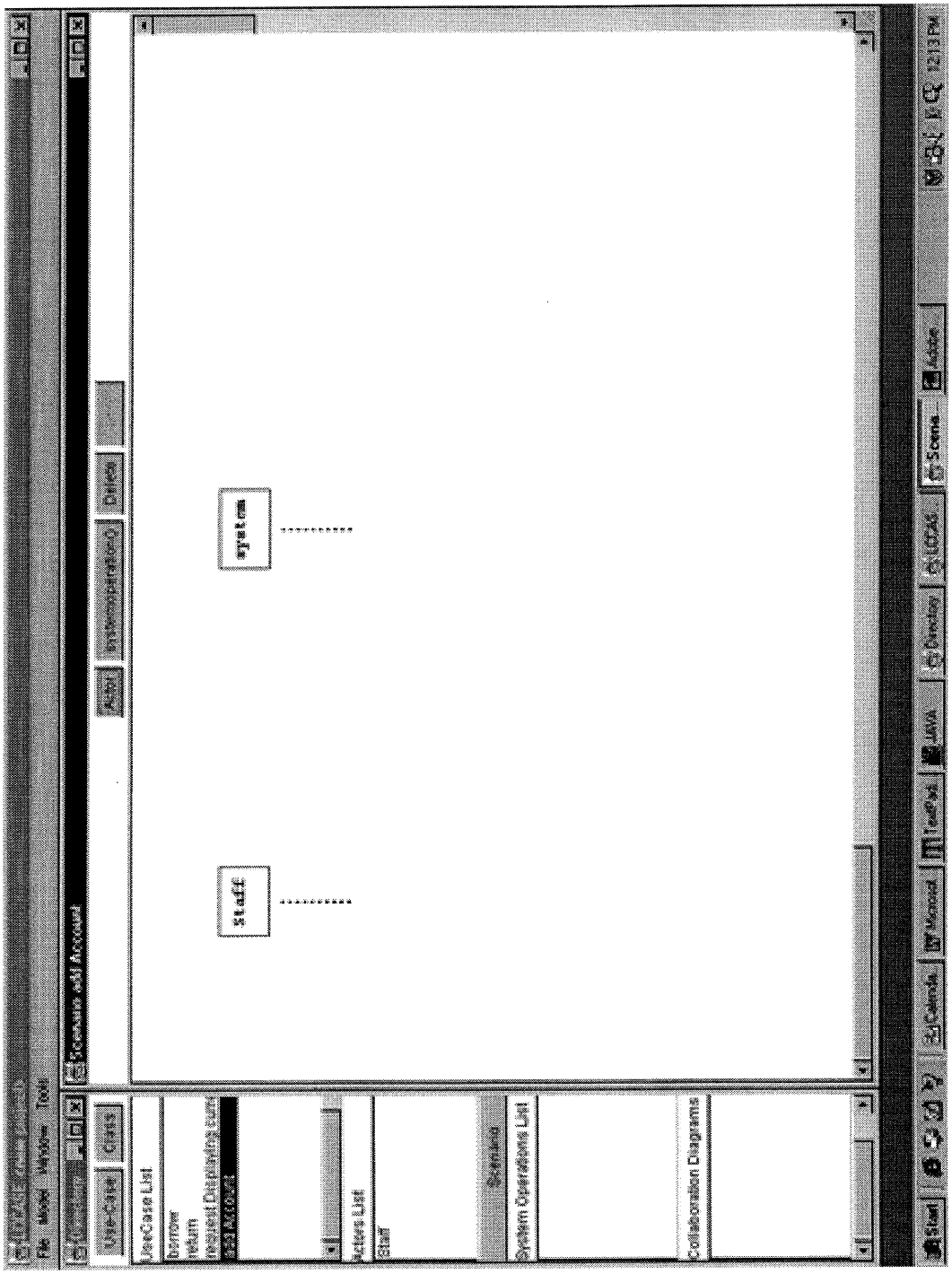


Figure 5-41 AddAccount Scenario after Adding the *staff* Actor

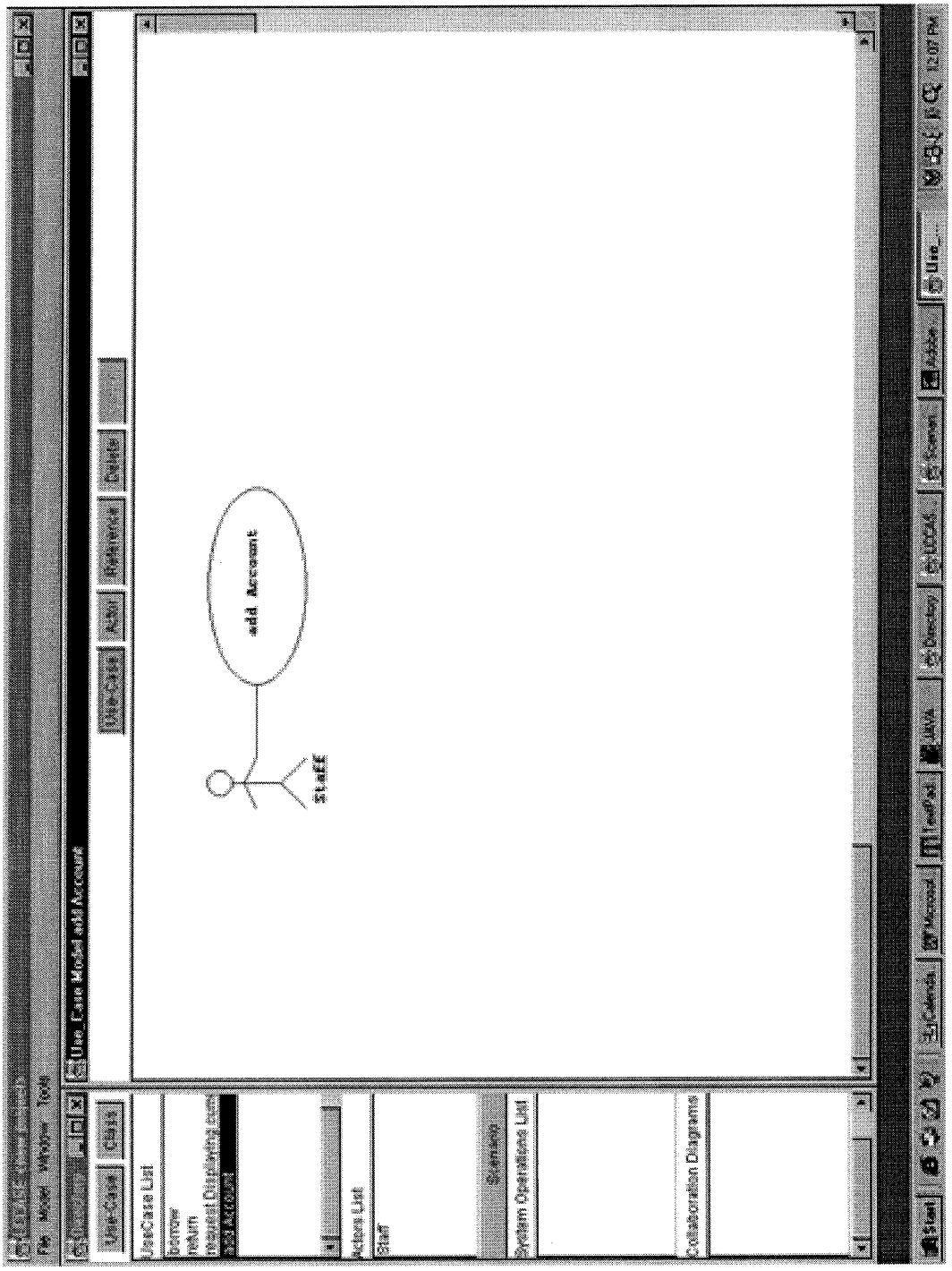


Figure 5-42 The AddAccount Use-Case Model

5.1.5 The Contract

Each Contract in the project corresponds to a system operation and documents its definition. The Contract specifies inputs, outputs, responsibilities, the class that contains the system operation, other objects in the project that relate to the system operation such as use-cases and a scenario, preconditions, and post-conditions.

The LCCASE tool creates a Contract automatically for each system operation. It updates some of the fields such as the name, inputs, and cross-references.

5.1.5.1 Important Features

Updating a Contract

When the user selects "Contract" from the menu bar to update the Contract, *borrow*. The LCCASE tool displays the Contracts List Dialog. It contains lists that display the names of all Scenarios that have been created in the project and the names of the Contracts in each Scenario. When the developer selects *borrow* from the Scenario list, the LCCASE tool displays the names of the Contracts in the Scenario, *borrow*. Figure 5-43 shows the Contracts List Dialog after the Scenario, *borrow*, has been selected. The developer then selects the Contract, *borrows*, from the Contract list. Next, the LCCASE tool displays the Contract Dialog, which contains fields that hold the name of the contract, its purpose, the name of the Class that has the method, output messages, the exceptional cases for the method, preconditions, and post-conditions such as, objects

creation or deletion, attributes changes, and creation or break of associations done by the method. Figure 5-44 shows the Contract, *borrow*, after entering its information.

The LCCASE tool provides the developer with a list that contains the names of all scenarios created in the project. When the developer selects a Scenario from the Scenario list, the LCCASE tool displays a list with all system operations in the Scenario and a list with the names of all created Contracts. This guides the developer to know which contracts should be developed.

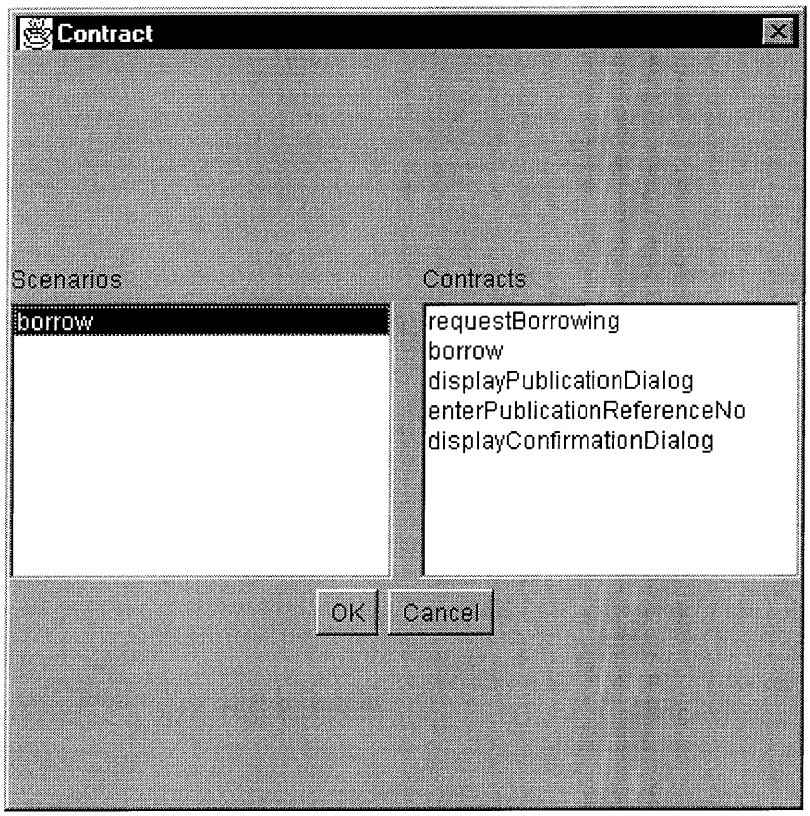


Figure 5-43 The Contracts List Dialog after Selecting the *borrow* Scenario

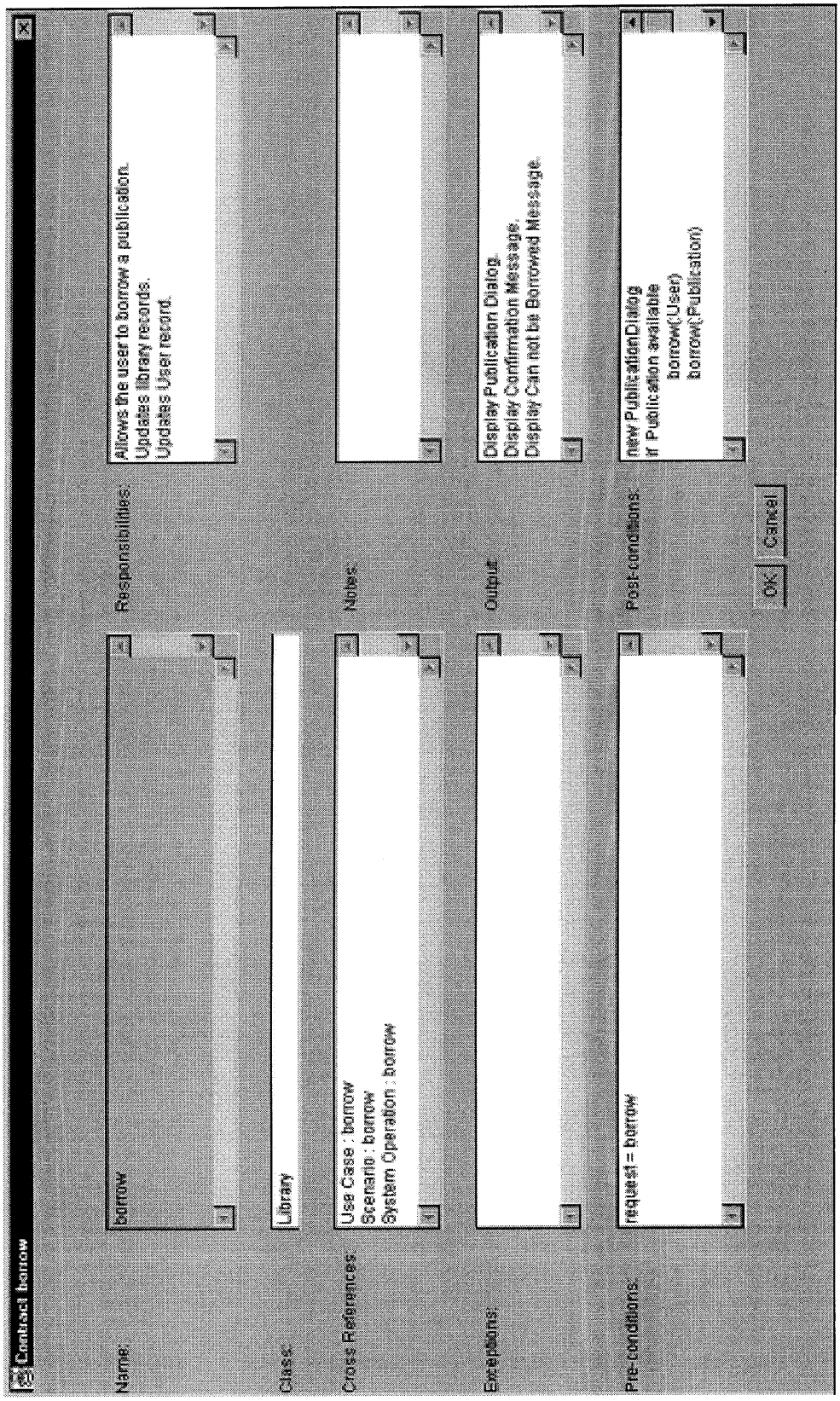


Figure 5-44 The borrow Contract

5.1.6 The User Use-Case Model Revisited

In order to illustrate the support of the LCCASE tool to the iterative life cycle style and to highlight features that could not be shown before developing the related Scenarios and Contracts, the User Use-Case Model is revisited.

Deleting a Reference

The operation, deleting a Reference, is revisited to illustrate the effects of deleting a reference after the creation of a Scenario corresponding to the use-case, connected by the reference.

When the developer needs to delete a reference, linked to a use-case that has a corresponding Scenario, the LCCASE tool informs the developer about the effect of the deletion on the Scenario. To delete the reference between the actor, *User*, and use-case, *borrow*, from the User Use-Case Model, the developer clicks on the Delete Button, then clicks on the reference. The LCCASE tool then displays a confirmation message that draws the developer attention indicating that deletion of the reference requires deleting the actor from the *borrow* Scenario as well as relevant system operations. This is necessary in order to keep the information consistent in the project. If the developer insists on deleting the reference, the LCCASE tool deletes the reference from the Use-Case Model diagram, the *User* actor and the system operations from the Scenario.

In addition, the deletion process saves the developer from manually:

- finding the Scenario which corresponds to the use-case connected to the reference;

- deleting the actor from the Scenario;
- deleting all the system operations in the Scenario connected to the actor;
- deleting the Contracts that correspond to the system operations.

When the developer selects *borrow* from the Use-Case list in the Directory window, the Use-Case list reflects the deletion of the actor, *User*, from the actor list and the system operations between the *User* actor and the system from the System Operations list. Figure 5-45 shows the User Use-Case Model and the Directory after deleting the reference.

Modifying an Existing Use-Case

In order to change the name of the use-case, *return*, to *returnPublication*, the developer double clicks on the use-case, *return*. The LCCASE tool then displays the use-case dialog. The LCCASE tool then redraws the use-case to display the new name (Figure 5-46).

When the developer changes the information of the use-case in the Use-Case Dialog, the LCCASE tool redraws the use-case directly to reflect the new changes. Moreover, it updates the Directory with the changes. This saves the developer time and effort of redrawing the use-case. In addition, the developer can get the latest information about the project from the Directory.

Moving an Actor

To change the location of the actor, *User*, in the User Use-Case Model, the developer points to the actor, *User*, and drags the mouse to the new location. The LCCASE tool

then moves the actor to the new location while retaining its references. Figure 5-47 shows the User Use-Case Model illustrating the effects of moving the actor.

Globally Deletion of a Use-Case

In order to delete the use-case, *requestDisplaying*, completely from the project, the developer selects the use-case, *requestDisplaying*, from the Directory and right clicks the mouse button. The LCCASE tool then displays a pop-up menu above the selected use-case with the menu item, *delete*. Figure 5-48 shows the pop-up menu. The developer selects the delete item. Next, the LCCASE tool displays a Confirm Deletion Dialog (Figure 5-49), which displays a message to the developer. When the developer selects Ok, the LCCASE tool deletes the use-case from the User Use-Case Model as well as from the Directory. Pop-up menus are location sensitive which are more friendly user interface.

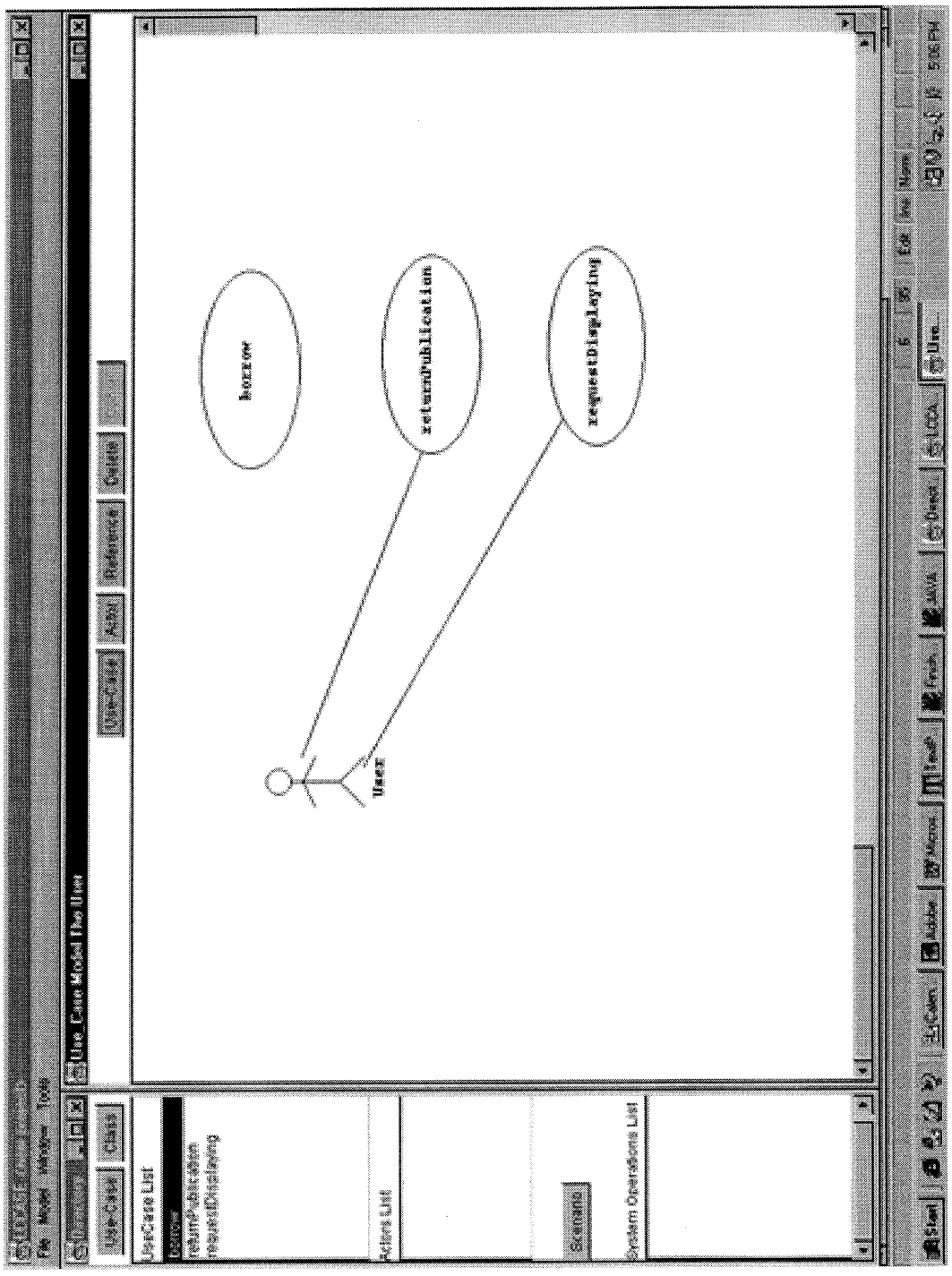


Figure 5-45 The User Use-Case Model after Deleting the Reference between the *User* Actor and the *borrow* Use-Case

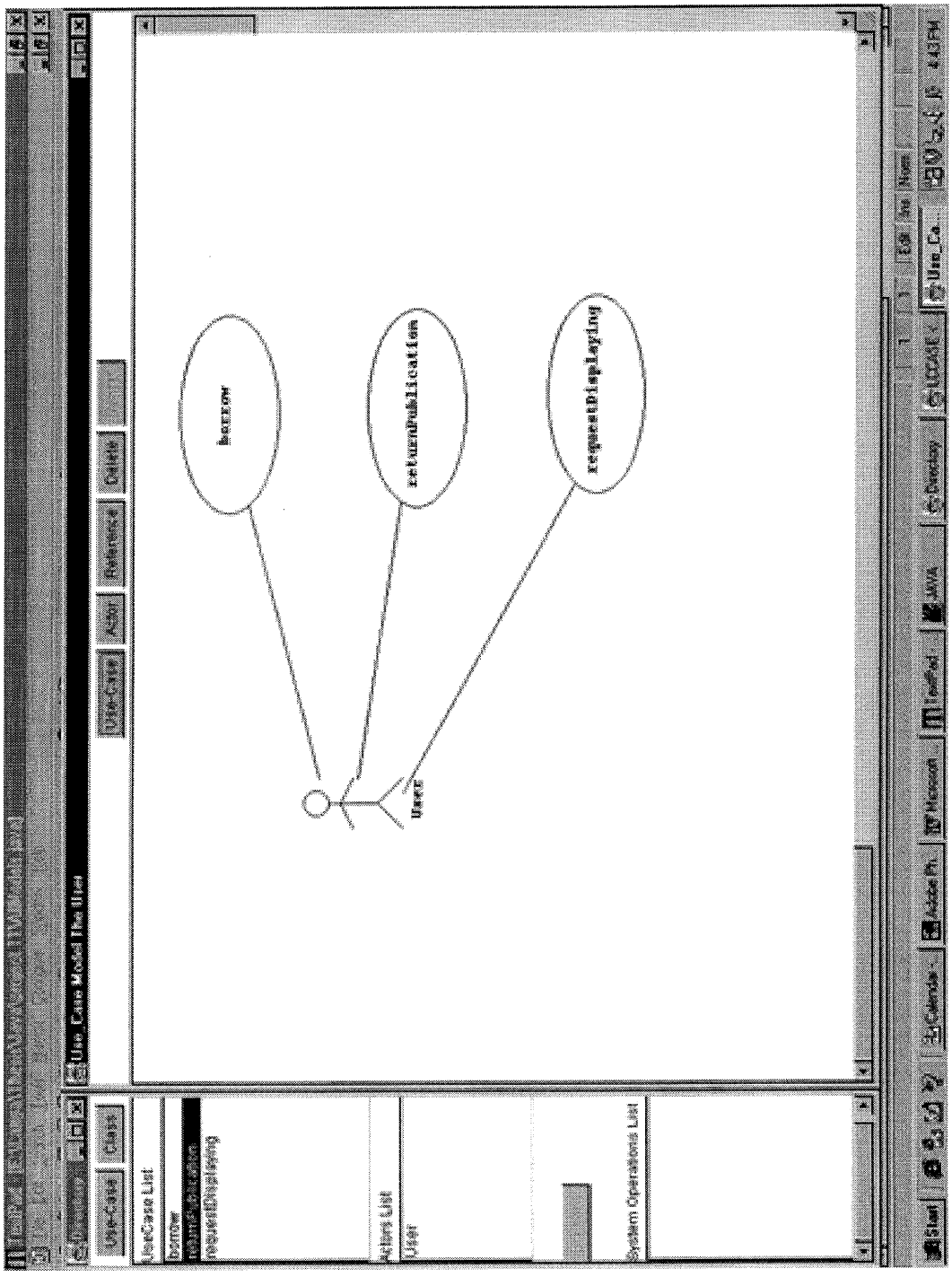


Figure 5-46 The User Use-Case Model after Modifying the *return* Use-Case

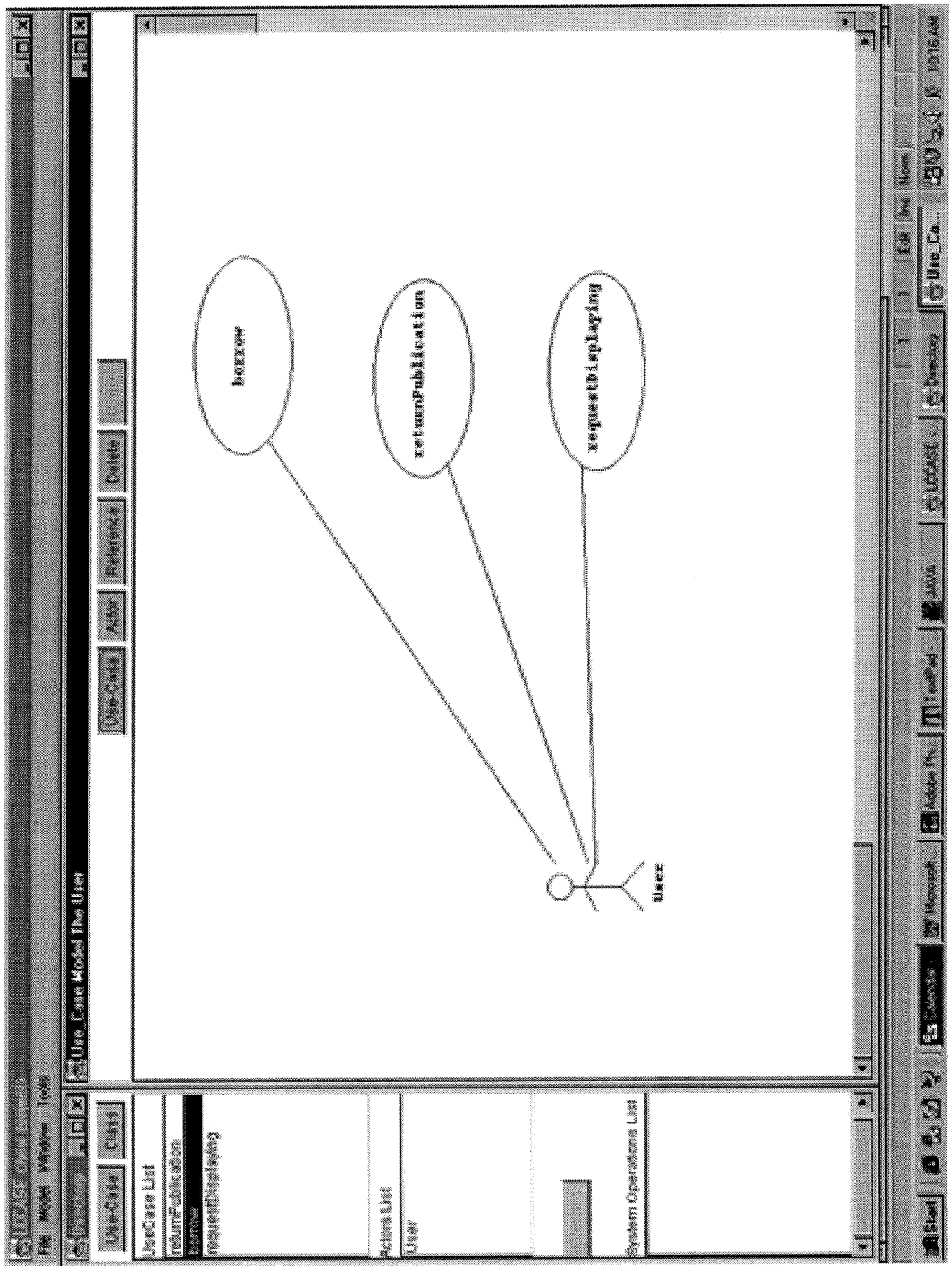


Figure 5-47 The User Use-Case Model after Moving the *User* Actor

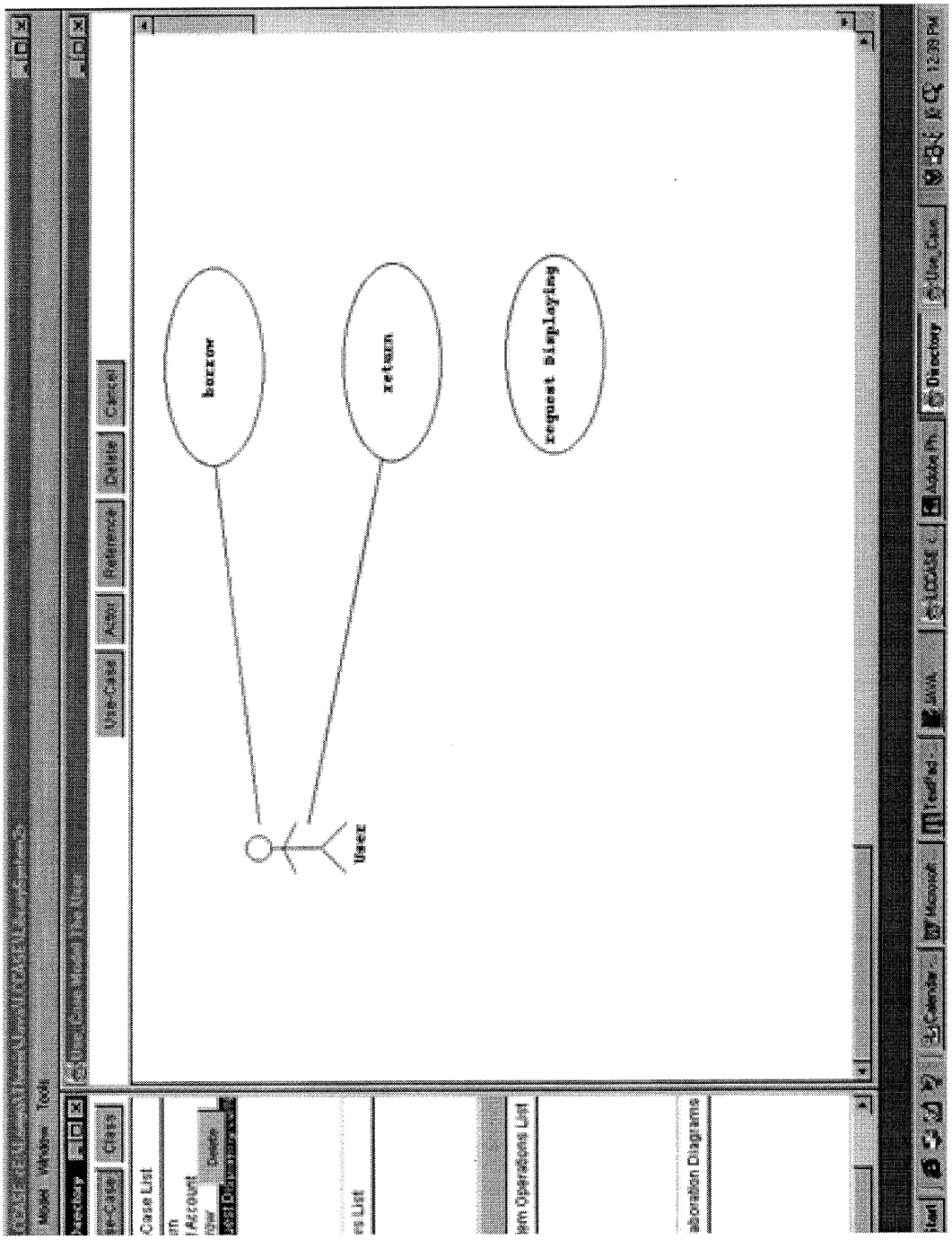


Figure 5-48 The pop-up menu

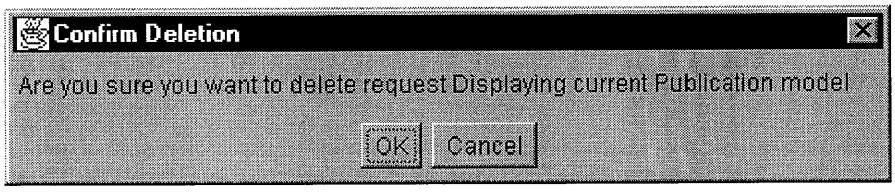


Figure 5-49 Confirm Deletion Dialog

5.1.7 The Window Menu

Checking Open Windows

When the developer selects "Window" from the menu bar, the LCCASE tool displays the menu with a fixed menu item, Tile Across, and dynamic menu items, the names of the currently open models. Figure 5-50 shows the Window menu items and the opened windows.

When the developer selects the menu item, *The User*, the LCCASE tool brings the User window to the front.

The Window menu has a dynamic structure. It displays the names of the models that are currently open in the project. This allows the developer to switch easily between the different models.

Viewing Multiple Models

When the developer clicks on Window menu then selects the menu item, *Tile Across*, the LCCASE tool arranges the currently open windows in the screen. Figure 5-51 shows an example of viewing multiple windows.

This feature helps the developer by automatically arranging multiple models in the screen. Moreover, displaying multiple models is important for the developer during the project development because most of the time the developer needs information from other models to develop the current one.

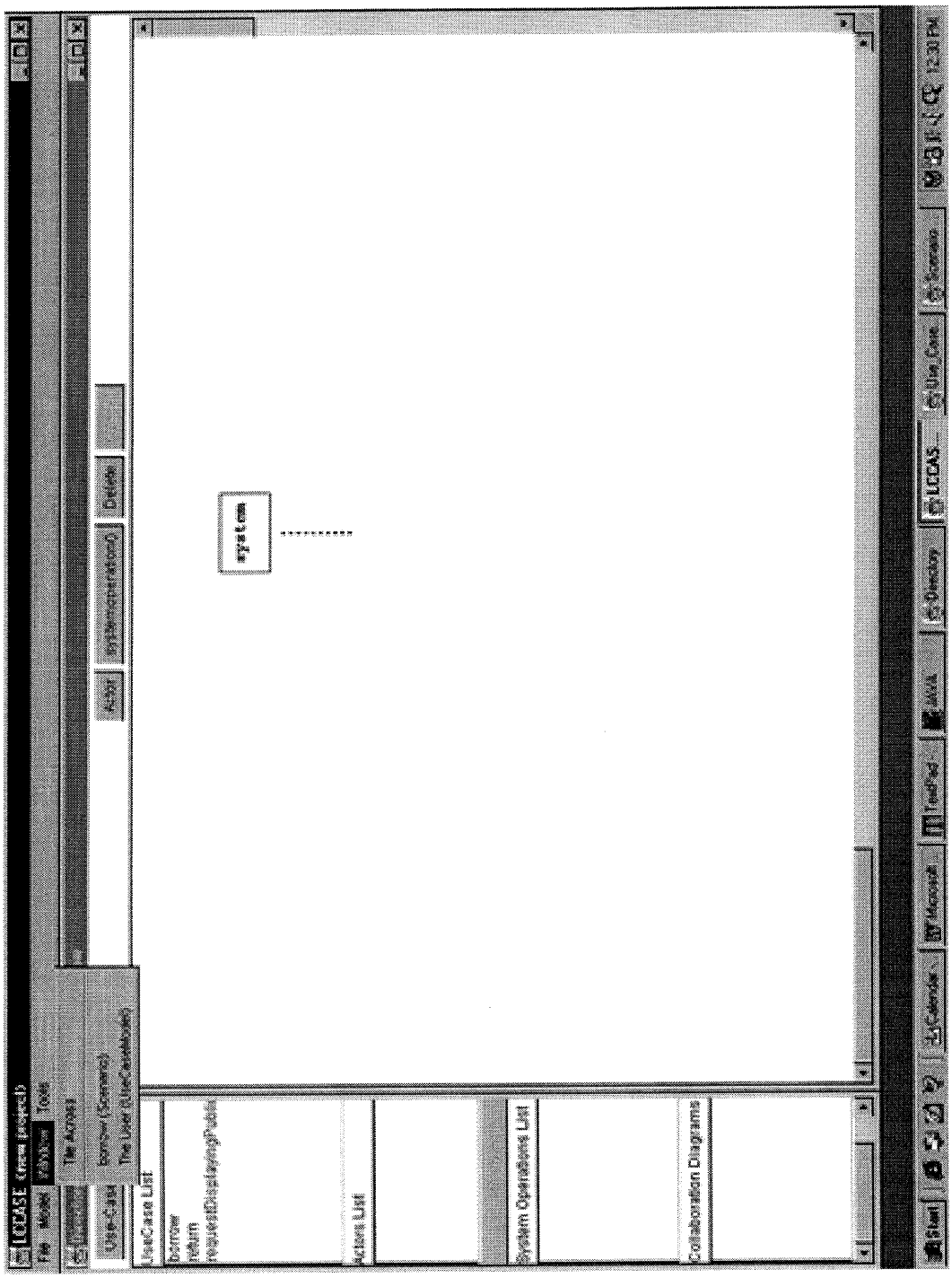


Figure 5-50 The Window Menu Showing the Dynamic Menu

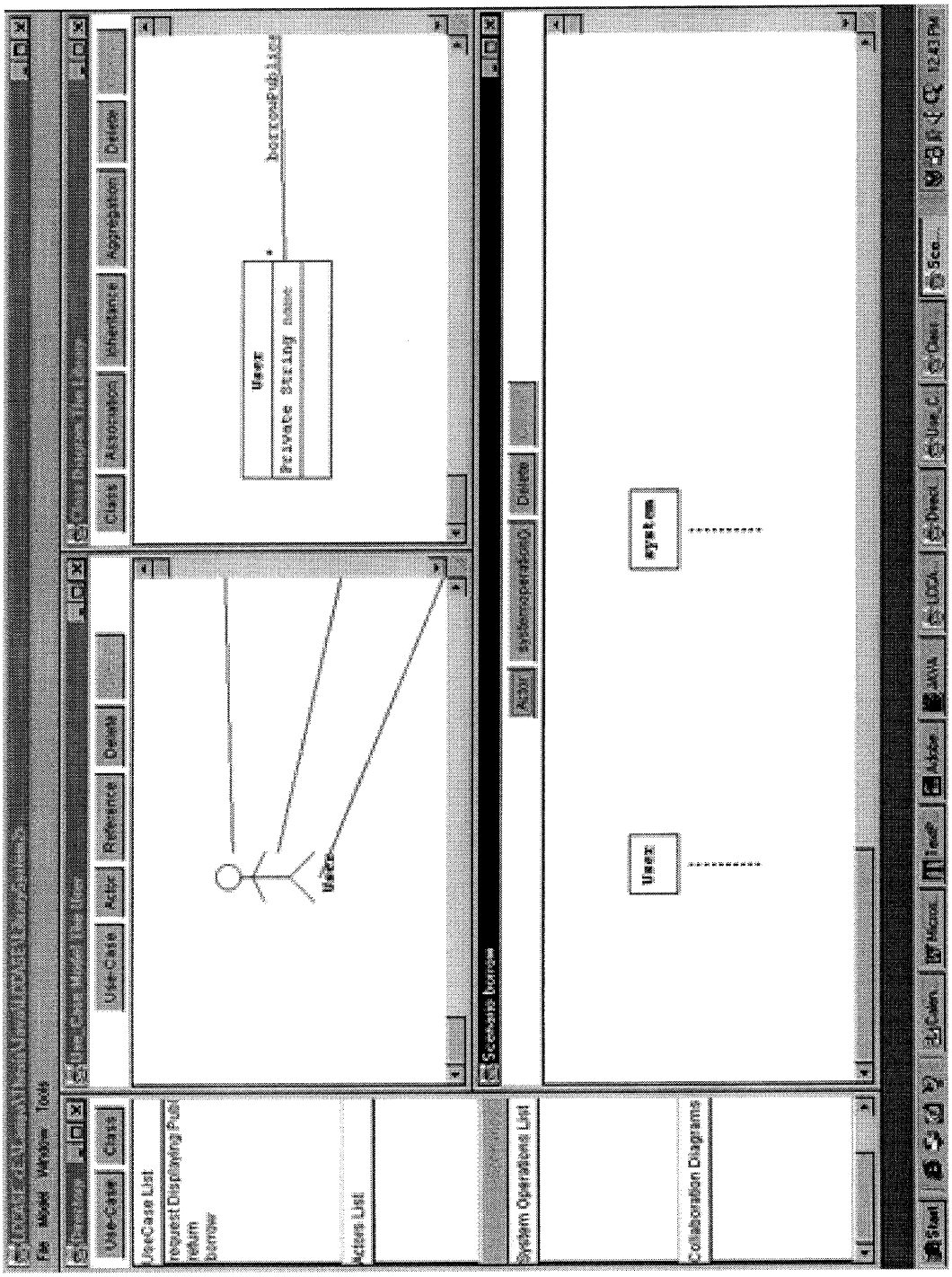


Figure 5-51 View Multiple Windows

5.1.8 The File Menu

The LCCASE tool allows the developer to start new project, open, save, and print a file of a project.

Once the developer saves a file, the LCCASE tool displays the name of the project in the title bar. Figure 5-52 shows the menu items of the File menu.

The LCCASE tool allows the developer to print the diagrams of the models.

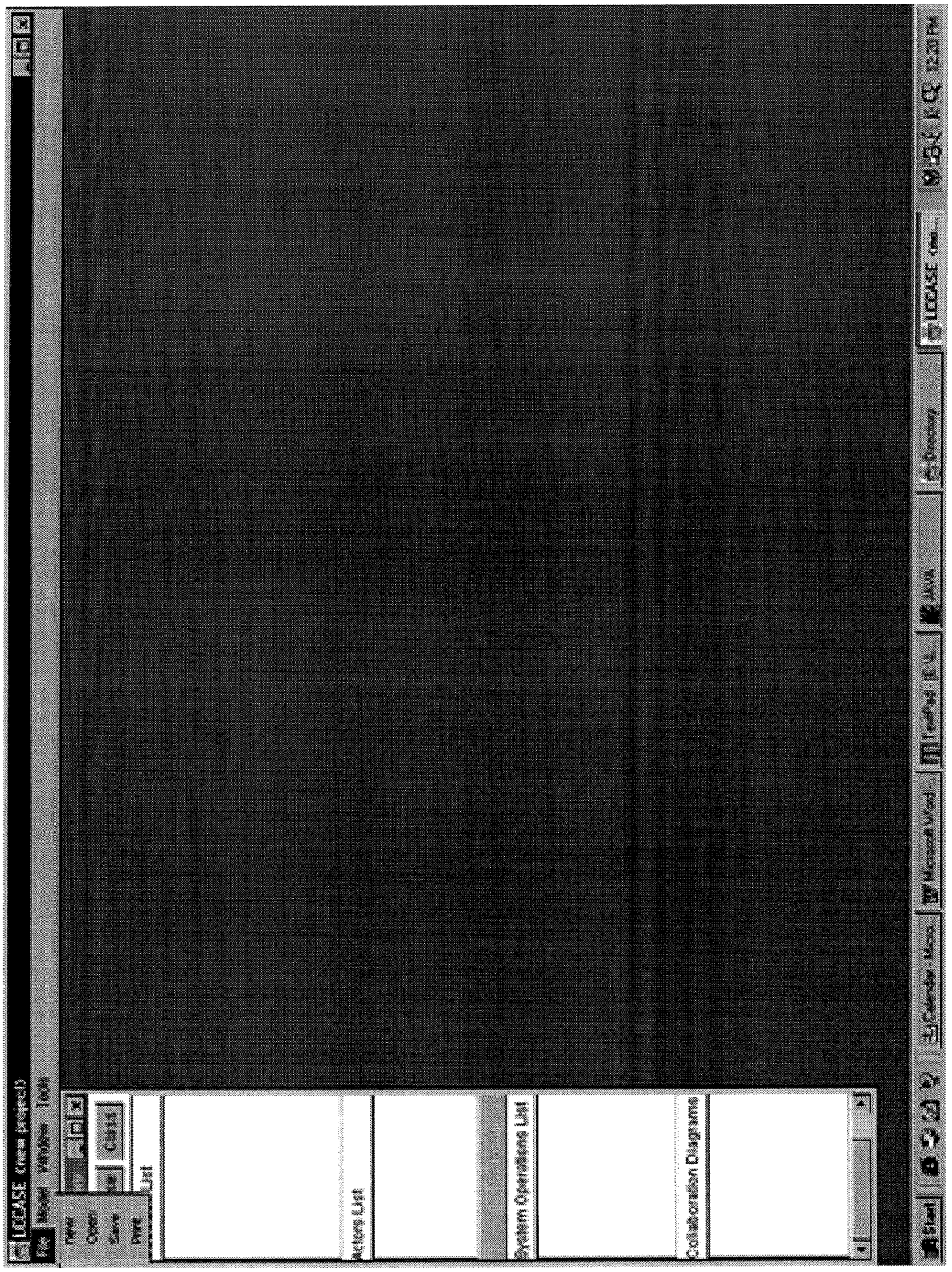


Figure 5-52 The File Menu

5.1.9 The Data Dictionary

The developer activates the Data Dictionary from the Tools menu. Figure 5-53 shows the menu items of the Tools menu. The LCCASE tool displays the Data Dictionary window, which contains a message requesting a key word to be entered, a field to be filled with a key word, a list of the different available categories of key words, a button to search for the key word, a list containing the key words of the project under development with their type, and a text area to display the description of the selected key word.

When the developer enters a key word, chooses the category of the search, and clicks the Search Button, the LCCASE tool highlights the keyword and its type in the key word list and displays its description in the Description field. Figure 5-54 shows the Data Dictionary after entering a key word and selecting a category. Figure 5-55 shows the Data Dictionary window with the search result.

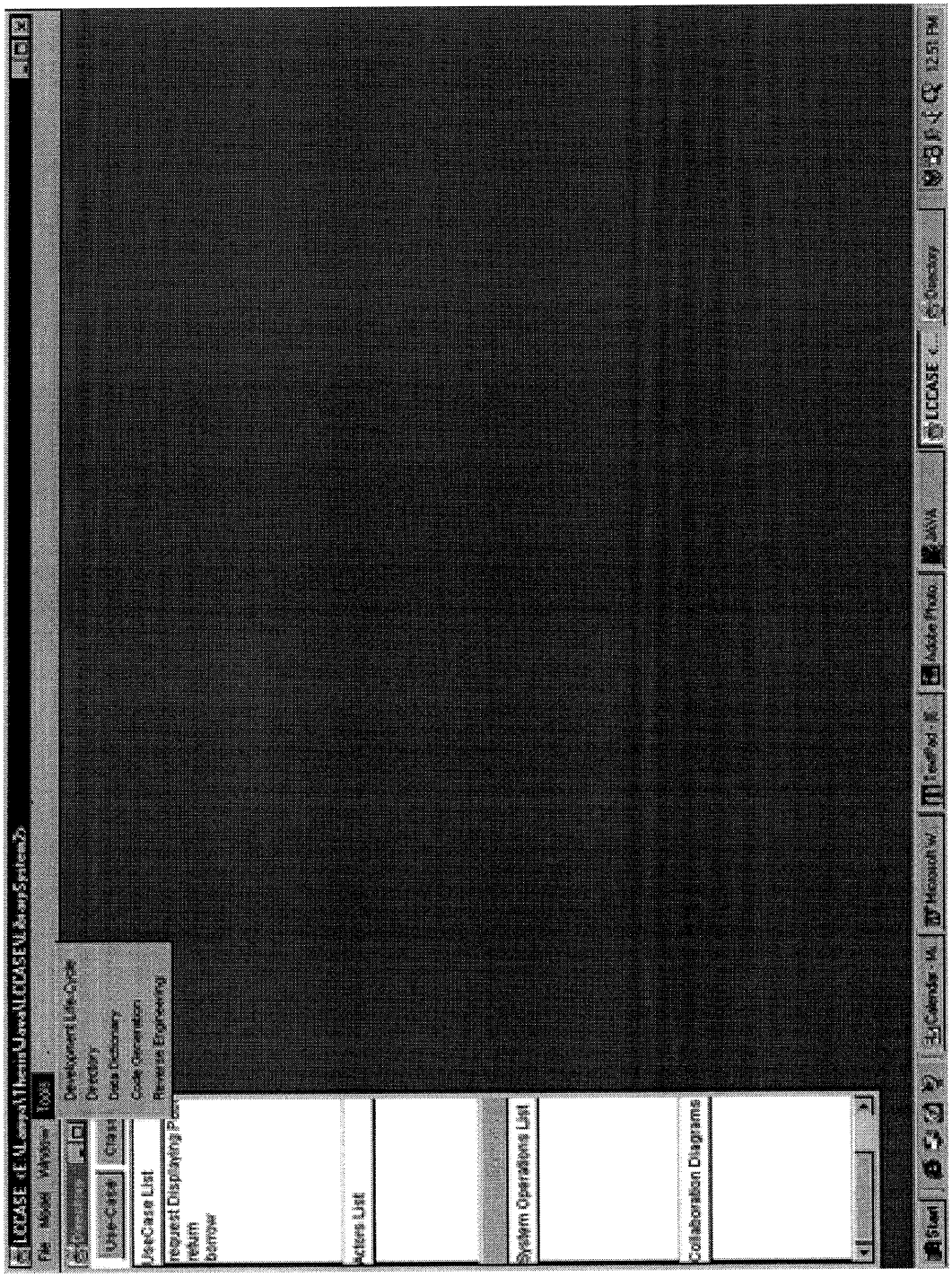


Figure 5-53 The Tools Menu

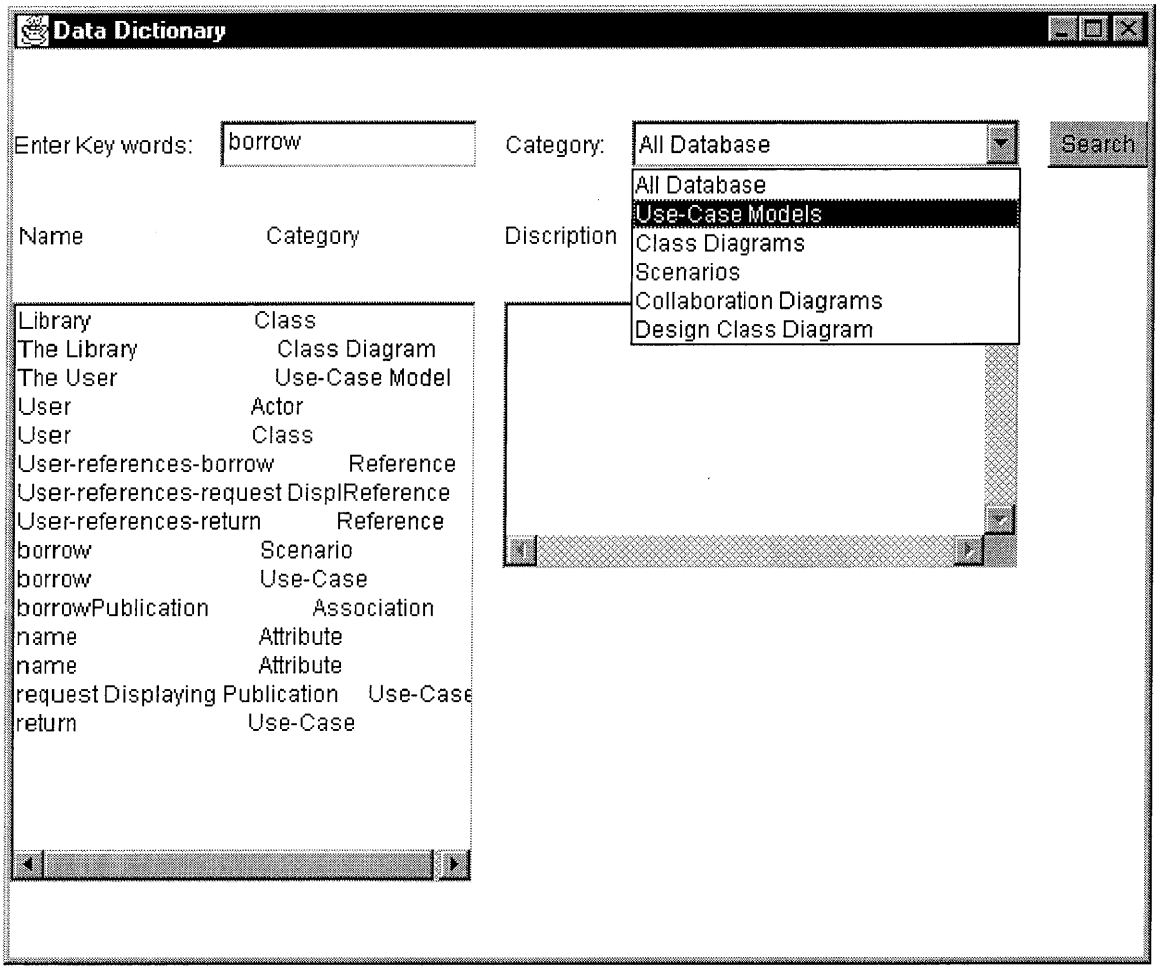


Figure 5-54 The Data Dictionary

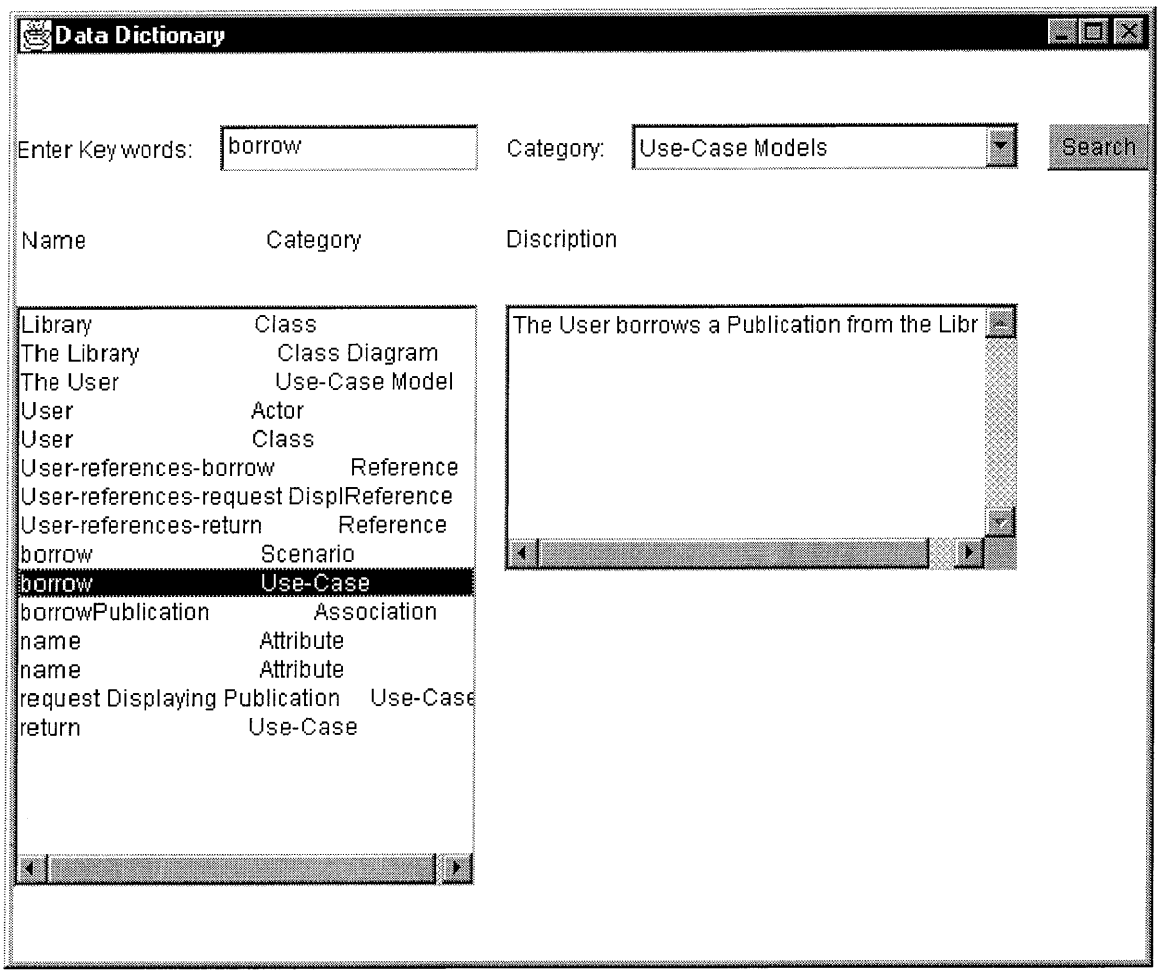


Figure 5-55 The Data Dictionary Displaying the Search Result

5.2 Chapter Summary

The chapter discussed how the user of the LCCASE tool could develop a project of an object-oriented software system. The project development included several operations demonstrating the functionality and highlighting the important characteristics of the LCCASE tool.

The chapter explained how to develop a Use-Case Model, Class Diagram, Scenario, and Contract using the LCCASE tool. In addition, it was shown the roles of the components of LCCASE tool and how they co-operate in developing the models.

Next, an example that shows the LCCASE tool support to the iterative life cycle style was demonstrated.

Finally, the chapter discussed the role of the Data Dictionary during the development of the project.

6 Implementation

The implementation of the LCCASE tool follows a defined strategy. The LCCASE tool has been developed by carefully applying object-oriented principles and analysis and design techniques. The Next Generation of the Fusion method, which has been explained in chapter 2 and Java programming language have been used to develop the LCCASE tool. In addition, the iterative and incremental life cycle style, explained in chapter 2, has been followed.

The design and development of the LCCASE tool were the most challenging tasks during my research. My aim during the implementation phase was the production of a CASE tool that would be useable by the experienced developers and learners of software engineering. The LCCASE tool is intended to function as a powerful environment, which supports the development of successful object-oriented software that any software developer would want to use. In addition, it is meant to assist learners of a course on object-oriented software engineering in developing software project and provide them with practical and user-friendly environment.

Moreover, the design and implementation of the LCCASE tool addresses the deficiencies with current CASE tools. One such deficiency is that they have little knowledge of the semantics and rules of the method they implement. In addition, these tools do not ensure the consistency between the models, support requirements tracing within the models of the method, or provide intelligent guidance through the software development.

Another goal was to use the LCCASE tool as a CASE tool for developing itself. This chapter discusses of the development of the models constructed within the LCCASE tool life cycle and examples of models developed by the LCCASE tool itself are given.

The implementation phase started with the analysis phase. The first step in the analysis phase was gathering the requirements of the LCCASE tool, which are discussed in the next section. These requirements are expressed in terms of use-cases. The use-cases describe the different ways in which the user acts on the LCCASE tool. The Use-Case Models are discussed in section 6.2.

Developing the Class Diagrams was the next step in the analysis phase. The requirements of the LCCASE tool and the Use-Case Models were of great help during the process of recognising the classes, their attributes, and the relationships with other classes in the Class Diagrams. During the development of Class Diagrams special attention was given to determine the inheritance relationships between the classes to minimise the code and maintain good standard of OO structures and techniques, find the aggregation relationships between the classes so that a complex class is composed of collection of small classes to make the Class Diagrams easy to understand and manageable, and establish association relationships between the classes to maintain well-defined visibility between the classes. The Class Diagrams are discussed in section 6.3.

A Scenario is developed for each use-case in the Use-Case Models. Each Scenario shows the events and system operations that are passed between the LCCASE tool and its user. The Scenarios which have been developed during the life cycle of the LCCASE tool are discussed in section 6.4 and included in Appendix A.

The last step in the analysis phase was the development of the Contracts, which are discussed in section 6.5 and is included in Appendix B. Contracts have been developed for most system operations in the Scenarios.

The design phase started by developing the Collaboration Diagrams discussed in section 6.6 and is included in Appendix C. Collaboration Diagrams have been developed for Contracts containing postconditions. They show the message sequence between a collection of objects as a result of a method call. This method call is the system operation described in the Contract, which corresponds to the Collaboration Diagram. The messages are methods, which are included in the postcondition close. In addition, frequent checks have been made on the developed Class Diagrams to ensure the consistency between the models about the information of the classes and objects.

Developing Design Class Diagrams was the next step in the design phase. Section 6.7 includes a discussion regarding the Deign Class Diagrams and they are included in Appendix D. Many important OO design techniques have been implemented during this phase such as abstraction, inheritance, polymorphism, and cohesion.

The implementation phase in the LCCASE tool life cycle started with writing the code. The Design Class Diagrams and Collaboration Diagrams are mapped to code. Section 6.8 describes the code phase. Several prototypes have been produced during the iterations of the LCCASE tool life cycle until the optimum version has been achieved.

Section 6.9 describes the complete life cycle of the development of a class in the Class Diagram. The implementation of the linkage of information between the models is discussed in section 6.10.

6.1 Requirements

The LCCASE tool is intended to provide four kinds of support: graphical, logical, navigational, and code generation. Graphical support allows the user to draw all the diagrams of the models and be able to edit, move, and delete their components. Logical support is responsible for understanding the semantics of the models, making consistency checks, providing intelligent guidance, and linking the information between the models. Navigational support provides easy access to the models and their components. Code generation produces code from the developed models.

The user of the LCCASE tool should be able to draw the diagrams, store the information about the models, receive intelligent guidance and feedback about the consistency between the information in the models, navigate between the models, trace the requirements implementation through out the development of the models, and easily view the use-cases, classes, and the relationships.

6.2 The Use-Case Models

The Use-Case Models consist of an actor, *Developer*, and many use-cases that describe the operations of the actor, *Developer*, to perform a process using the system, LCCASE tool. The actor, *Developer*, represents the user of the LCCASE tool, which is described in the previous section.

The Use-Case Models illustrate the requirements of the system by showing the interaction between the *Developer* and the use-cases. In order to keep the Use-Case Models simple and clear to understand, each Use-Case Model describes the

operations of a specific function of the LCCASE tool that is performed by one of its subsystems. The developed Use-Case Models include:

- Model Use-Case Model.
- Directory Use-Case Model.
- Development Life Cycle Use-Case Model.
- Data Dictionary Use-Case Model.
- File Use-Case Model.
- Window Use-Case Model.
- Code Generation and Reverse Engineering Use-Case Model.

6.2.1 Model Use-Case Model

The Model Use-Case Model consists of the actor, *Developer*, and the use-cases, *create*, *browse*, and *delete*. These use-cases describe the high-level operations that occur during the development process of a model. Figure 6-1 shows the Model Use-Case Model, which has been developed using the LCCASE Tool itself.

The use-case, *create*, is responsible for creating a new model. The actor, *Developer*, specifies the type of the model and enters its name. The LCCASE tool displays the model window and becomes ready to execute the commands of the developer for creating, manipulating, and drawing the components of the model. In addition, it provides guidance and performs consistency checks. Each type of model such as Use-Case Model, ClassDiagram, Scenario, Collaboration Diagram, and Design Class Diagram has its own components, notations, functions, and diagram.

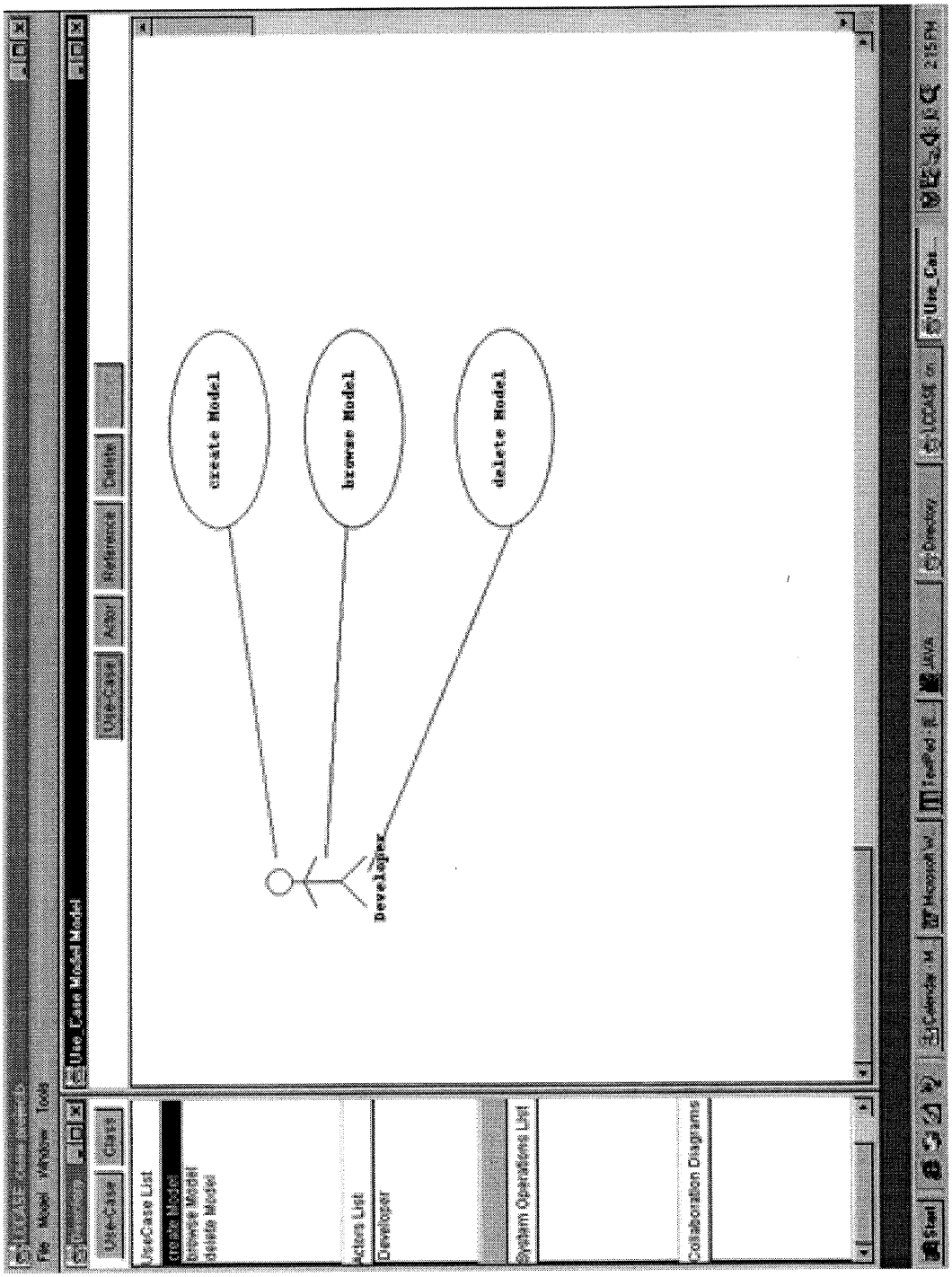


Figure 6-1 Model Use-Case Model developed by LCCASE Tool

The use-case, *browse*, allows the developer to work with an existing model. The actor, *Developer*, specifies the type and name of the model. The LCCASE tool displays the model window and provides him with all the facilities discussed above.

The use-case, *delete*, is responsible for deleting a model from the database. The actor, *Developer*, specifies the model and requests deleting it. The LCCASE tool performs consistency checks to make sure that this deletion does not affect other models in the project and provides the *Developer* with a feedback. The LCCASE tool deletes the model from the database but the essential components such as the use-cases and classes remain in the project and require explicit deletion request.

6.2.2 File Use-Case Model

In the case of File Use-Case Model (Figure 6-2), the actor, *Developer*, interacts with the LCCASE tool to start a new project, open an existing one, and save one in a file on disk.

The use-case, *Start New Project*, allows the *Developer* to work on a new project while he is using the LCCASE tool. The *Developer* requests starting a new project. The LCCASE tool reset all the subsystems such as the Database, Directory, Data Dictionary, Development Life Cycle, and Model and become ready to start the new project.

The use-case, *Open File*, loads a file from disk, containing a project that has previously been developed using the LCCASE tool. The LCCASE tool displays the name of the loaded project in the title bar and starts accepting the commands from the *Developer* to perform on the loaded project.

The use-case, *Save File*, is responsible for storing the project currently under development in a file.

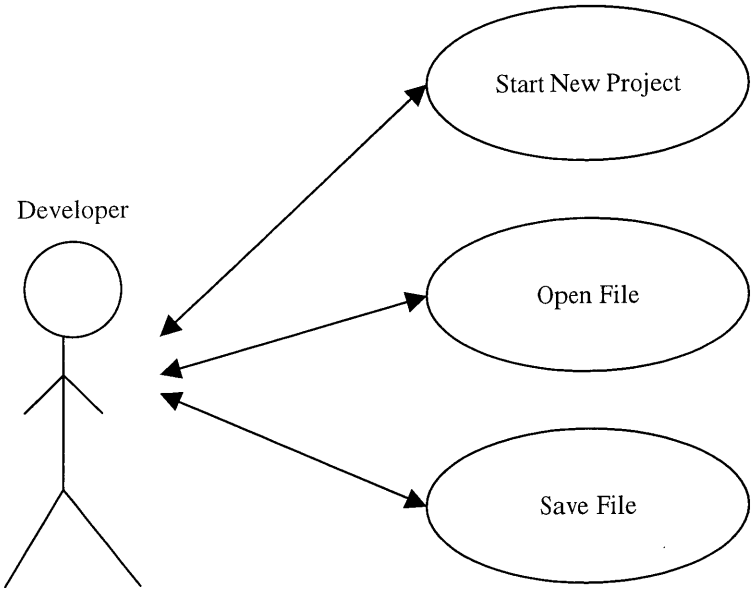


Figure 6-2 File Use-Case Model

6.2.3 Window Use-Case Model

The Window Use-Case Model (Figure 6-3) illustrates the window operations supported by the LCCASE tool. It consists of the actor, *Developer*, and the use-cases, *Tile Windows Across* and *Bring a Window to Front.*

Tile Windows Across is a use-case that is responsible for viewing multiple windows of the currently opened models on the screen. The LCCASE tool arranges and resizes the windows according to the number and type of the currently opened models.

The use-case, *Bring a Window to Front*, brings the window of a currently opened model to the front of the other opened windows. The LCCASE tool has a dynamic menu that displays the names of the currently opened models. The developer can request showing a model from the menu.

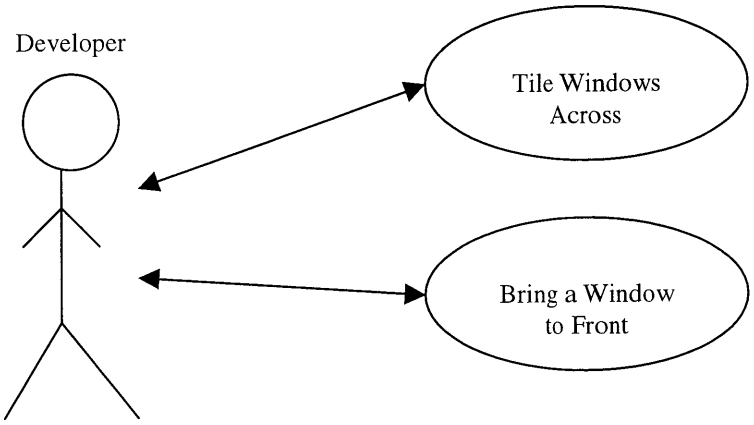


Figure 6-3 Window Use-Case Model

6.2.4 Development Life Cycle Use-Case Model

The Development Life Cycle Use-Case Model (Figure 6-4) contains the actor, *Developer*, and the use-cases, *Display Development Life Cycle* and *Check Models Development Status*. The use-cases show the interaction of the *Developer* with the LCCASE tool to benefit from the features and facilities of the Development Life Cycle subsystem, which are described in detail in chapter 4.

The use-case, *Display Development Life Cycle*, activates the Development Life Cycle subsystem and displays its window which shows a buttons that guides the *Developer* to the models that should be developed.

The use-case, *Checks Models Development Status*, shows the models that are developed and the models that should be developed of the type specified by the *Developer*.

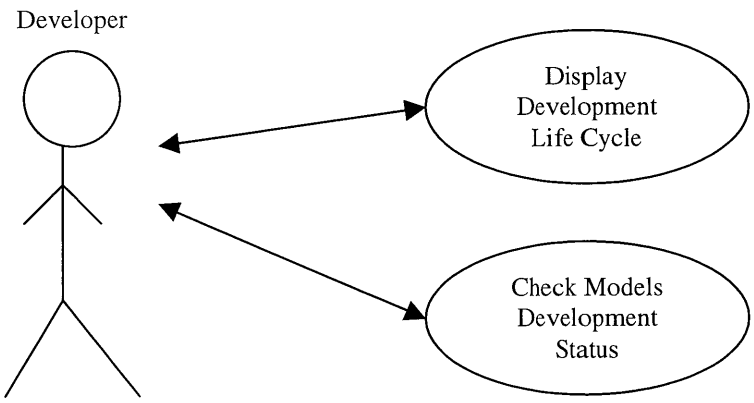


Figure 6-4 Development Life Cycle Use-Case Model

6.2.5 Directory Use-Case Model

The Directory Use-Case Model (Figure 6-5) shows the interaction between the *Developer* and the Directory subsystem. The *Developer* can display the Directory, select the Use-Case panel, select use-case, delete use-case, select Class Diagram panel, select class, and delete class.

The use-case, *Display Directory*, shows the window of the Directory on the screen.

Select Use-case Panel and *Select Class Panel* are use-cases that are responsible for showing the use-case tab view or the class tab view. The Directory has two views, use-case and class.

The use-case, *Select Use-Case*, allows the Developer to select a use-case from the Use-Case list in the Directory. The LCCASE tool displays the names of the actors that are connected to the selected use-case via a reference in the Actors list. It activates the Scenario Button if a scenario is created in the project that corresponds to the selected use-case. In addition, the LCCASE tool displays the names of the system operations in the System Operations list, the names of the Collaboration Diagrams in the Collaboration Diagram list, and shows the Use-Case Model containing the selected use-case.

The use-cases, *Delete Use-Case and Delete Class*, perform a global deletion to a selected use-case or class. The LCCASE tool checks if the selected use-case or class does not have relations with other components in the project or dependent model and provides the developer with the necessary feedback.

Select Class is a use-case that is responsible for allowing the Developer to select a class from the class view in the Directory. It displays in the association, inheritance, and aggregation relationships list, the names of the corresponding relationships that the selected class is participated in. In addition, it shows the window of the Class Diagram containing the selected class.

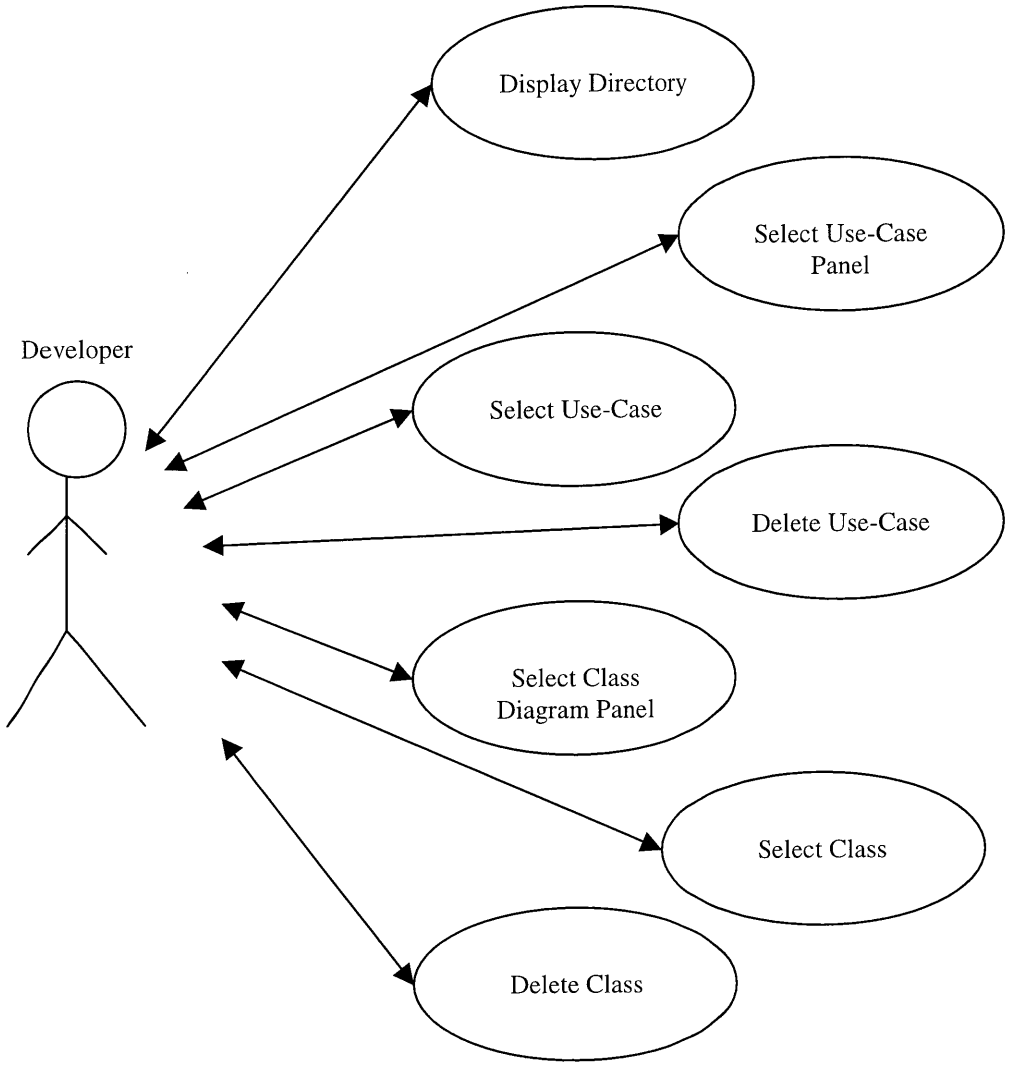


Figure 6-5 Directory Use-Case Model

6.2.6 Data Dictionary Use-Case Model

The Data Dictionary Use-Case Model (Figure 6-6) illustrates the interaction between the *Developer* and the Data Dictionary subsystem. The *Developer* can display the Data Dictionary Window, search for an item, or select an item.

The use-case, *Display Data Dictionary*, activates the Data Dictionary subsystem and shows its window.

The *Developer* enters a keyword. The LCCASE tool finds the keyword in a list that contains all the components in the project and displays its description. This operation is performed by the use-case, *Search for Item.*

The use-case, *Select Item*, allows the Developer to select an item from the list containing all the components in the project. The list contains other useful information such as the type.

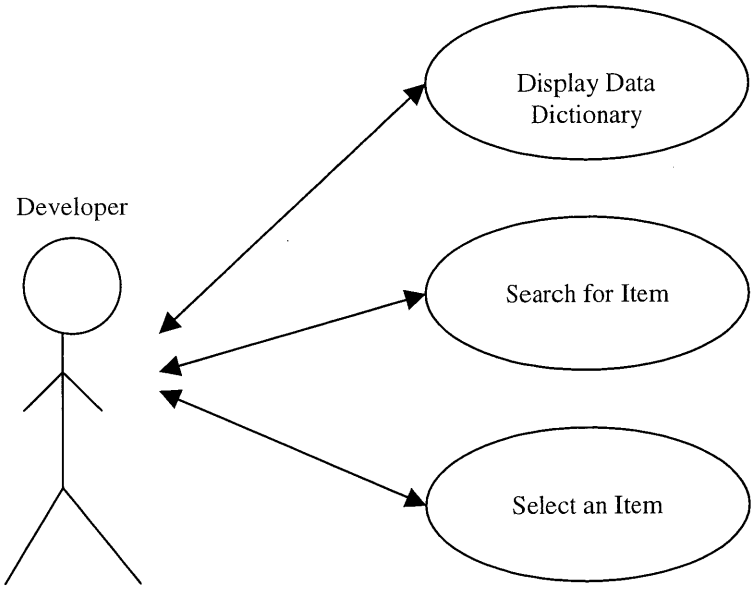


Figure 6-6 Data Dictionary Use-Case Model

6.2.7 Code Generation Use-Case Model

The Code Generation Use-Case Model (Figure 6-7) contains the actor, *Developer*, and the use-case, *Generate Code*.

The use-case, *Generate Code*, provides code in Java programming language for the project. It utilises the information in the Design Class Diagrams and Collaboration Diagrams to generate code.

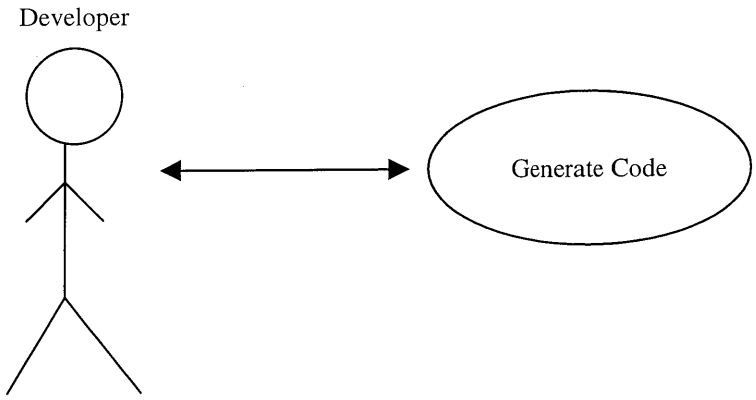


Figure 6-7 Code Generation Use-Case Model

6.3 Class Diagrams

The development of the Class Diagrams involves decomposing the problem into individual concepts or classes. The Class Diagrams show the classes, relationships between classes such as association, inheritance, and aggregation, and attributes of the problem domain. They aid in clarifying the vocabulary and terminology of the problem domain.

The development process of large software systems such as the LCCASE tool is very complex. To deal with this complexity, my primary task during the analysis phase and especially during the development of Class Diagrams is to decompose the problem domain into manageable units. Within the object-oriented environment, the decomposition process driven by the concepts in the problem domain. Thus, my goal is to create Class Diagrams that contain important and meaningful concepts of the problem domain.

Developing the Class Diagrams within the iterations of the software life cycle involves many activities, which include:

- finding and listing the candidate classes with the help of the information and the noun phrases in the requirement statement and the Use-Case Models;
- drawing the Classes;
- adding the association, inheritance, and aggregation relationships to the diagrams and adding the attributes to the classes.

Once the development of scenarios and system operations has been completed in the first iteration of the software life cylce, methods can be added to the classes in the second iteration.

The following subsections describe the Class Diagrams that have been developed during the life cycle of the development of the LCCASE tool and discuss the important issues that are raised during this phase. These Class Diagrams have undergone several refinement processes throughout the different iterations of the software life cycle. The subsections present the final version of the Class Diagrams and highlighting important issues in previous iterations.

6.3.1 AllModels Class Diagram

AllModels Class Diagram (Figure 6-8) implements the design concepts discussed in chapter 4. The LCCASE tool is composed of the subsystems, Model, Directory, Database, Data Dictionary, and Development Life cycle. Each subsystem performs a specific task. The subsystems co-operate together to support the LCCASE tool functionality.

The AllModels Class Diagram contains the class, *AllModels*, which allows the *Developer* to access the subsystems of the LCCASE tool and benefit from their facilities. This is achieved through the interaction with a main menu in AllModels class window. In addition, the Class Diagram contains the classes that represent the subsystems of the LCCASE tool. These classes include *Database*, *Directory*, *Data Dictionary*, and *Development Life Cycle*. It contains class, *ModelInfo*, which is responsible for accepting different data from the user and conveying it to the relevant subsystem.

There is an aggregation relationship between *AllModels* class and other classes in the diagram. The *AllModels* class contains the classes *Database*, *Directory*, *Data Dictionary*, *Development Life Cycle*, and *ModelInfo*. This aggregation relationship represents a logical containment, that is, the container class, *AllModels*, contains the subsystems of the tool and allows the component classes to be visible to it. The idea behind the compositional design in this Class Diagram is to build complex classes out of simpler ones to reduce its complexity and it is easier to manage small classes.

The *AllModels* class has the following attributes:

- *menu* which is the main menu of the tool. It allows the developer to request from the tool to perform the high level operations which are discussed in the Use-Case Models, such as create, browse, or delete a model.
- *fileDialog* is a dialog box that supports the file functions.
- *fileName* is a string that contains the file name that corresponds to a project.
- *directoryName* holds the name of the directory where the file that contains the project under development resides.

The following subsections describe the components of the *AllModel* class and their relations with other classes.

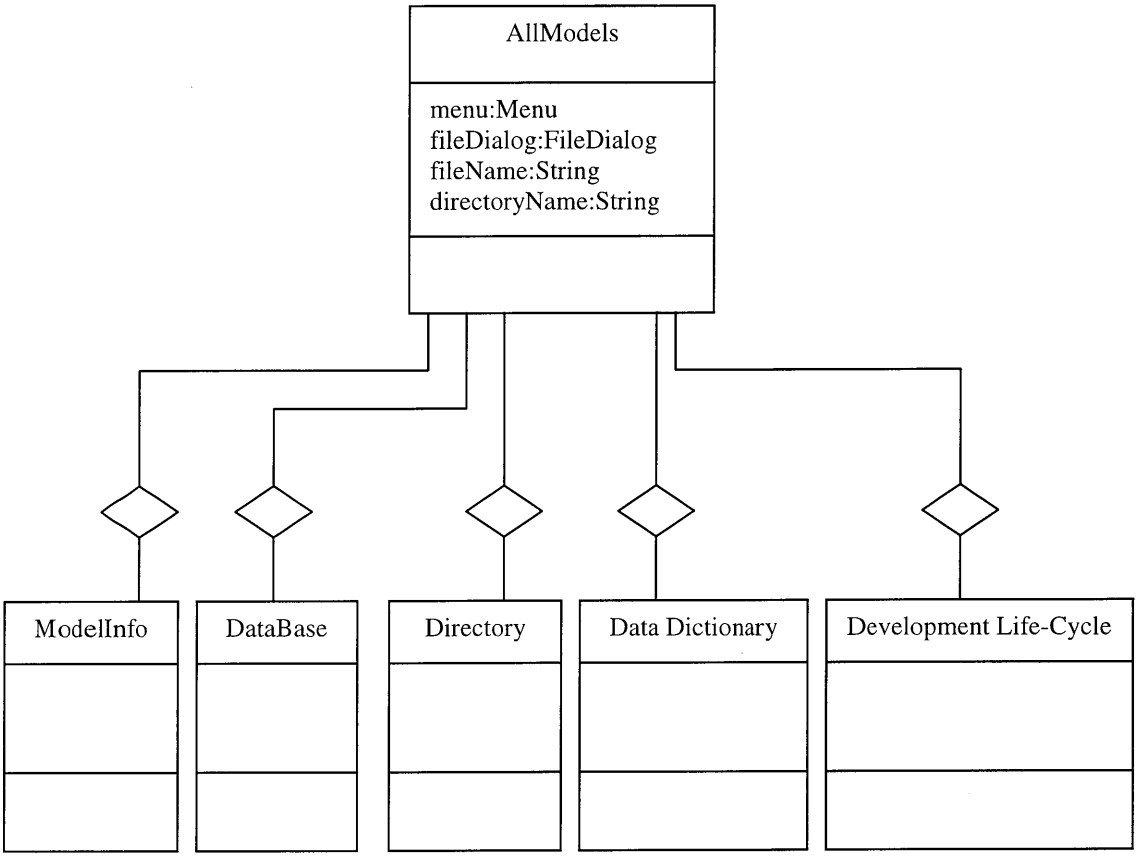


Figure 6-8 AllModels Class Diagram

6.3.2 Model Class Diagram

The Model Use-Case Model discussed in subsection 6.2.1 gave rise to the need for the classes *UseCaseModel*, *ClassDiagram*, *Scenario*, and *Collaboration Diagram*, which represent the models in the tool and handle the operations of the models specified in the Model Use-Case Model. These classes have many common features, such as they all have a name, description, window operations, accessed by the AllModels class to request creation, browsing, and deletion operations. This raises the need for the *Model* class, which is an abstract class that contains the common attributes among the models. Figure 6-9 shows Model Class Diagram. It shows the inheritance relationship between the *Model* class, parent, and the classes *UseCaseModel*, *ClassDiagram*, *Scenario*, *CollaborationDiagram* which represent the children. The inheritance relationship reduces the size of the code. The Model Class has two association relationships with the *AllModels* class. The first association relationship shows that the *AllModels* class creates one or more *Model* class. The second association relationship is *browses*.

In addition, the *Model* class has two association relationships, *updates*, with the *Database* and *Directory* classes. The *updates* association relationship between the *Model* class and *Database* class allows the *Model* class to store all the information about its components and structure in the *Database* class and retrieve the required information from the *Database* class.

The *updates* association relationship between the *Model* class and the *Directory* class allows the *Model* class to inform the *Directory* class about its new components.

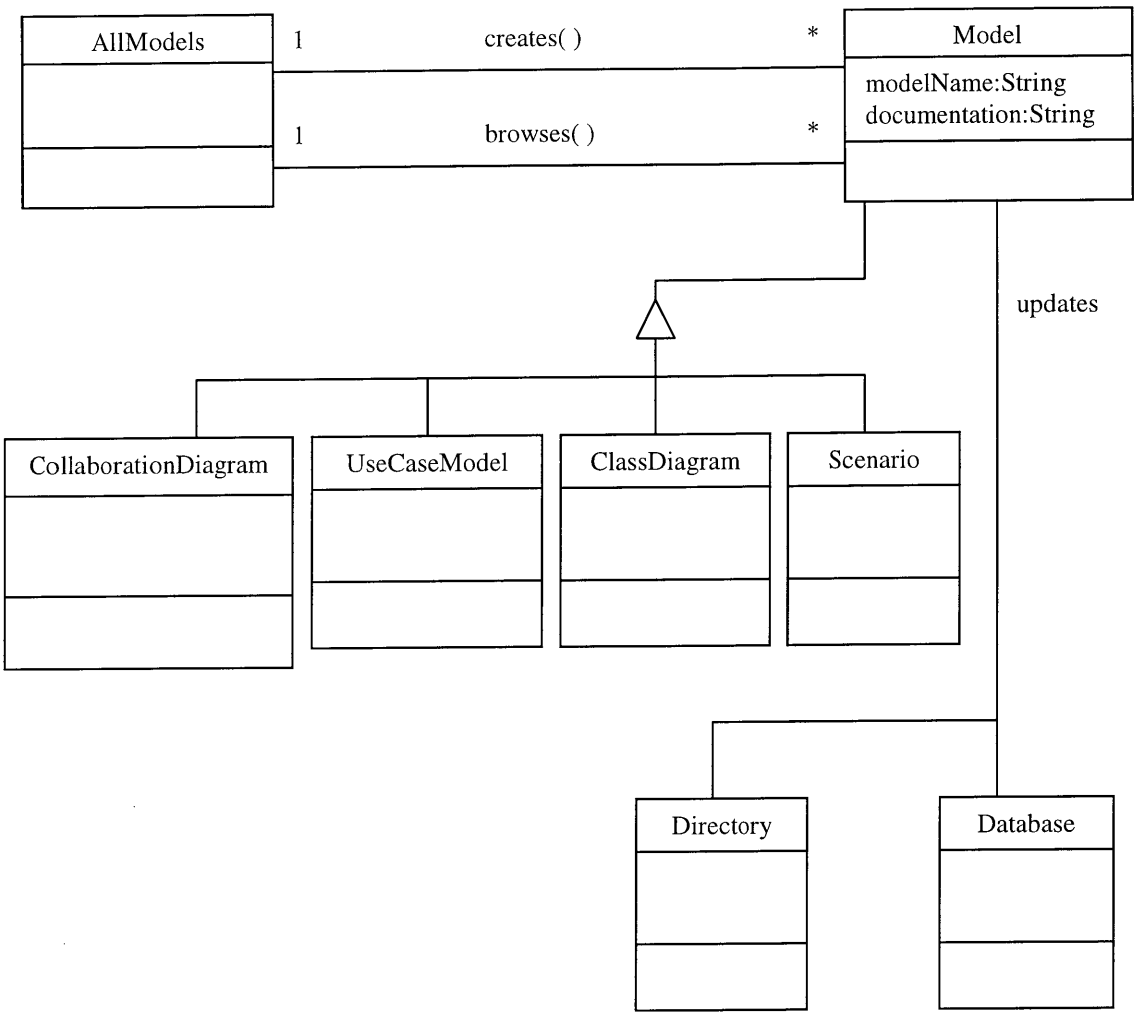


Figure 6-9 Model Class Diagram

6.3.3 Use-Case Model Class Diagram

The Use-Case Model Class Diagram shows the attributes of the *UseCaseModel* class and its relations with other classes. The *UseCaseModel* class has the attributes *usecases*, *actors*, and *references* which store pointers to the usecases, actors, and references in the UseCaseModel. The reason behind choosing hashtables to store the pointers of the

usecases and actors is that each use-case and actor has a unique name but the references do not have a unique name. Thus a vector was suitable to store references. The attribute, *choice,* accepts the user request such as create use-case, actor, or reference; delete component, or cancel and passes this information to the *UseCaseModelCanvas* class. There is an aggregation relationship between the *UseCaseModel* class, container, which is composed of the *UseCaseModelCanvas* class, component. The *UseCaseModel* class needs to have drawing capabilities to draw the Use-Case Model it represents. Its component, the *UseCaseModelCanvas* class, which is of type Canvas, is responsible for the drawing task.

The *UseCaseModelCanvas* class is also responsible for creating, modifying, and drawing the components of the Use-Case Model, which are the use-cases, actors, and references. The *UseCaseModelCanvas* class has association relationships, *creates* and *deletes*, with the *UseCase*, *Actor*, and *Reference* classes, which allow the *UseCaseModelCanvas* class to instantiate or destroy objects of the classes, *UseCase*, *Actor*, or *Reference*.

Since use-cases and actors can be included in more than one Use-Case Models and each occurrence can be in different location, their location in a diagram could not be stored in the use-case or actor classes. The *ObjectLocation* class is needed to hold the location of the use-cases and actors in the diagram. For each occurrence of actor or use-case in a diagram, there is an *objectLocation* object to hold its position. The *UseCaseModelCanvas* class creates and deletes the *ObjectLocation* class whenever it creates or deletes an actor or use-case in the diagram. The *ObjectLocation* class has the

attribute, *name*, which holds the name of the use-case or actor and the attribute, *location* to save the location of the use-case or actor in the diagram.

The *UseCaseModelCanvas* class has an association relationship, *updates*, with *Directory* class, which allows the *UseCaseModelCanvas* class to add or remove use-cases and actors to or from the Directory.

The *InfoDialog* class is responsible for conveying the data from the user to the LCCASE tool. The *UseCaseModelCanvas* class creates an object of type *InfoDialog* whenever it needs to get information from the developer.

The LCCASE tool was used to develop the Figures 6-10 to 6-13. The figures illustrated how the tool links the information in the project and provides it to the user when needed. Figure 6-10 shows the Use-Case Model Class Diagram. The Directory window in the diagram shows the relationships of the selected class, *UseCaseModel*, from the class list, which assists in developing the class diagram especially if the relationships are in different class diagram. Figure 6-11 shows the General View of the Class Dialog for the *UseCaseModel* class. The dialog displays the name of the parent of the *UseCaseModel* class, which is the *Model* class. Figure 6-12 shows the Attribute View of the Class Dialog for the *UseCaseModel* class. The dialog shows the inherited attributes and the attributes created in the class. Figure 6-13 shows the Association Dialog for the association relationship, creates. The LCCASE tool displayed the name of the two classes involved in the relationship.

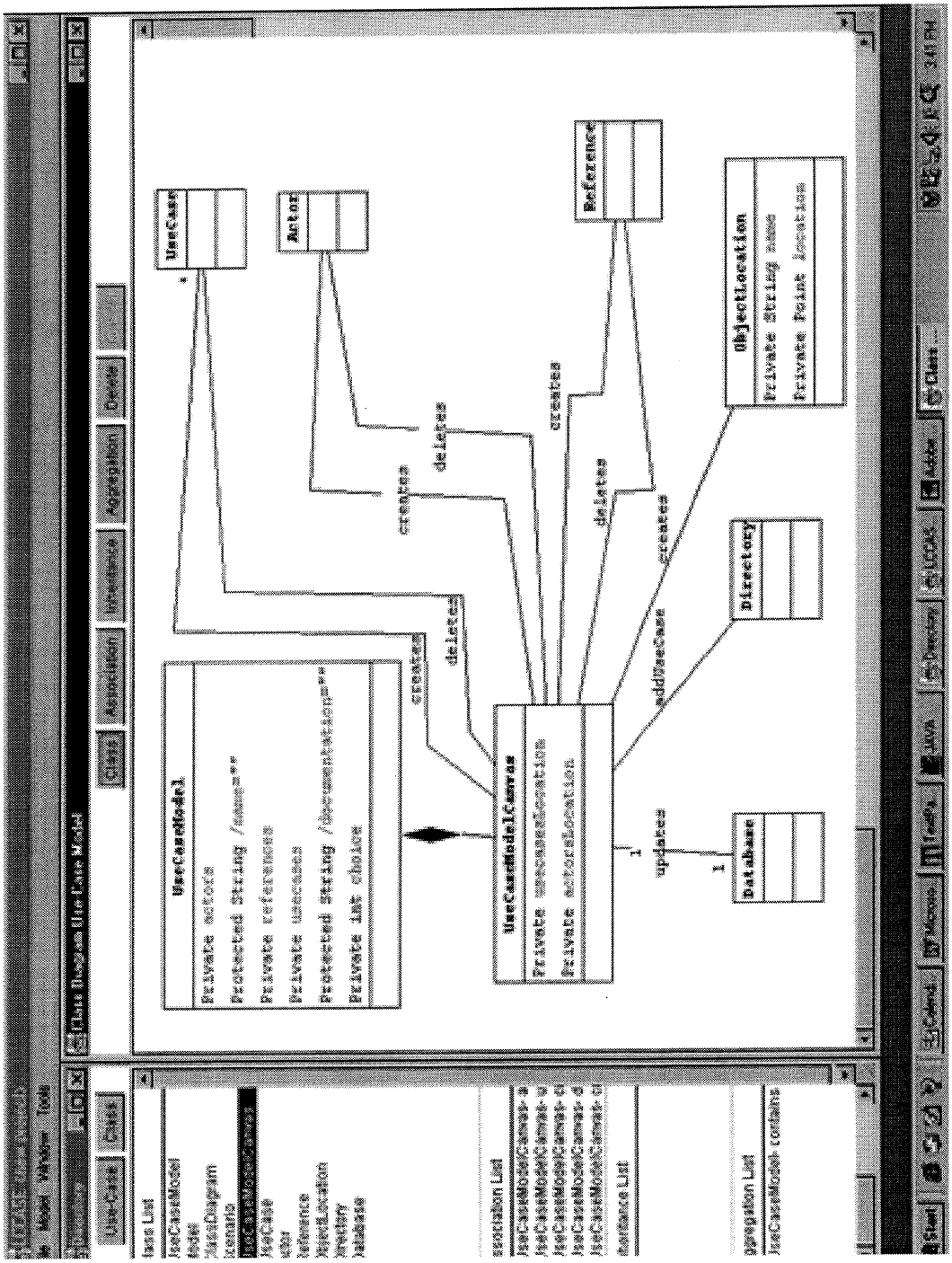


Figure 6-10 Use-Case Model Class Diagram

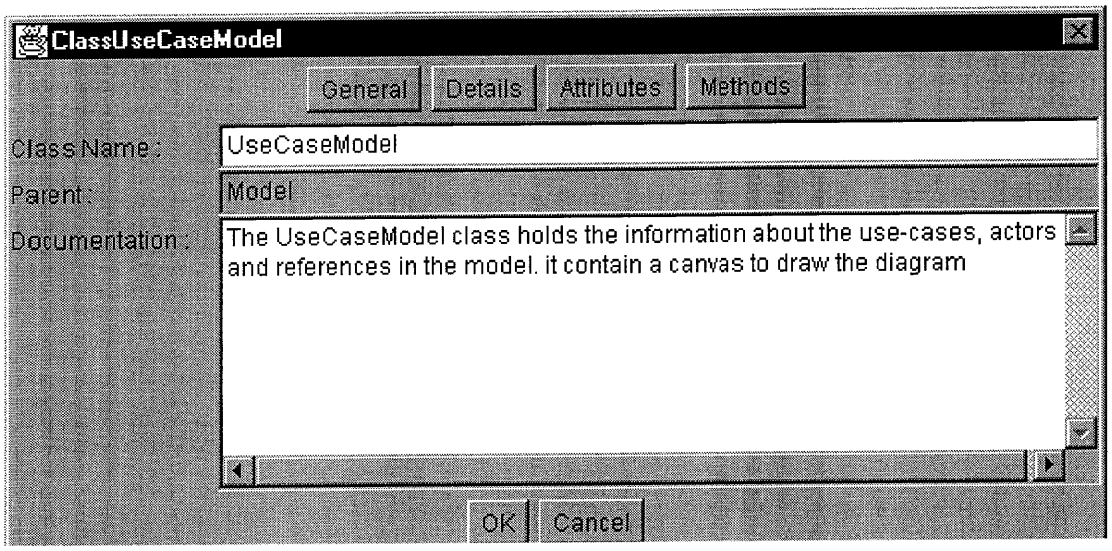


Figure 6-11 The General View of the Class Dialog for the UseCaseModel Class

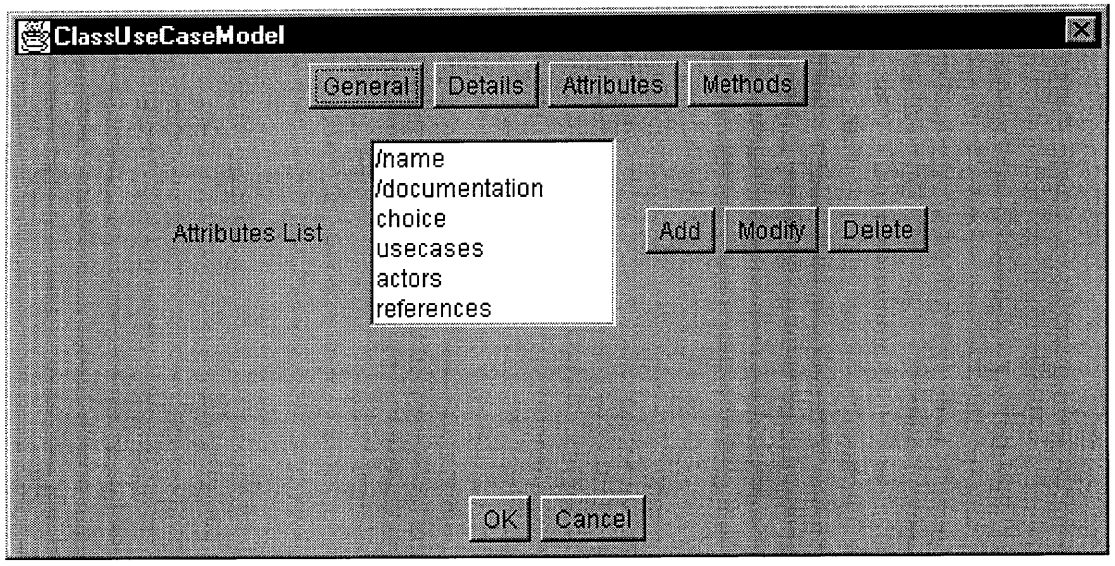


Figure 6-12 The Attribute View of the Class Dialog for the UseCaseModel Class

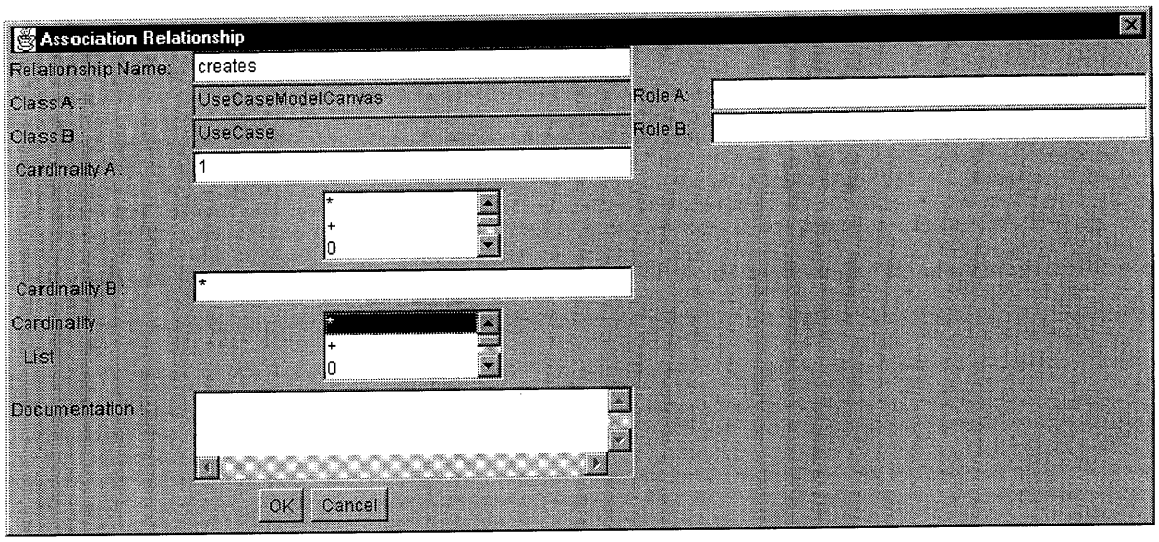


Figure 6-13 The Association Relationship Dialog

6.3.4 Use-Case Model Objects Class Diagram

The Use-Case Model Objects Class Diagram (Figure 6-14) shows that *Use-Case* class and the *Actor* class inherit from the abstract class, *UseCaseModelObjects* because they have common attributes such as *name*, *documentation*, *height*, and *width*. The attributes, *height* and *width,* store the dimension of the use-case and actor. Since the use-cases and actors have different shape, the abstract method *draw* in the *UseCaseModelObjects* class has been overridden in the classes, *Use-Case* and *Actor*. In addition, the *Use-Case* and Actor classes have the association relationships, *referenced-By*, with the class, *Reference*. The association relationships, *referenced-By*, allow the *Reference* class to know and be able to access the actor and use-case, which are linked via the reference.

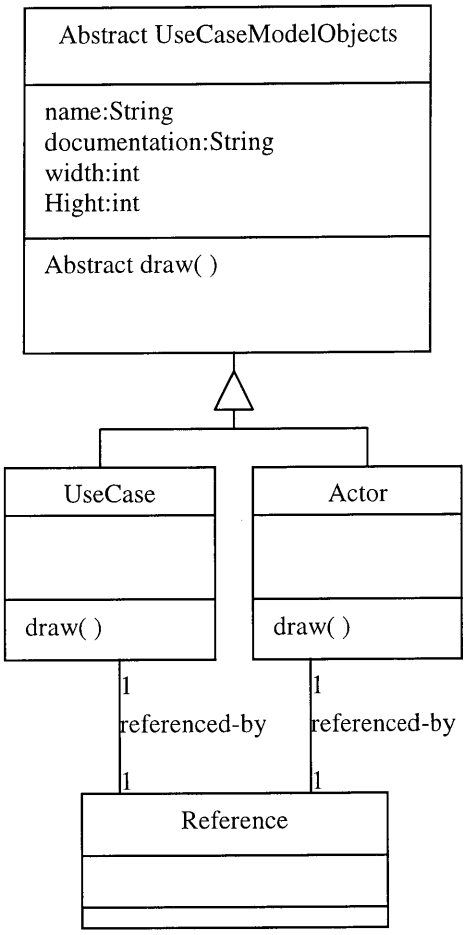


Figure 6-14 Use-Case Model Objects Class Diagram

6.3.5 The Relationships of the ClassDiagramCanvas Class Diagram

The Relationships of the ClassDigramCanvas Class Diagram (Figure 6-15) has a structure similar to the UseCaseModel Class Diagram, which reflects the uniform strategy used for developing the models.

The association relationships, *creates* and *deletes*, between the *ClassDiagramCanvas* class and the classes *ClassClass*, *Association*, *Inheritance*, *Aggregation*, and *ObjectLocation* allows the *ClassDiagramCanvas* class to instantiate or destroy objects of those types. The ClassDiagram class passes the value of the attribute, *choice*, which holds an integer number that are specified according to the request of the developer to create or delete class or relationship, to the ClassDiagramCanvas class and then it instantiates or destroys the object specified by *choice*.

The updates association relationship between the *ClassDiagramCanvas* class and the *Directory* class allows the *ClassDiagramCanvas* class to inform the *Directory* class about the new classes created in the diagram.

The *ObjectLocation* class used to hold the location of the classes in the diagram. It is used because of the same reason that *ObjectLocation* class needed in the Use-Case Model Diagram. Other classes in the diagram such as *ClassClass*, *Association*, *Inheritance*, and *Aggregation* are discussed in detail in the following Class Diagrams.

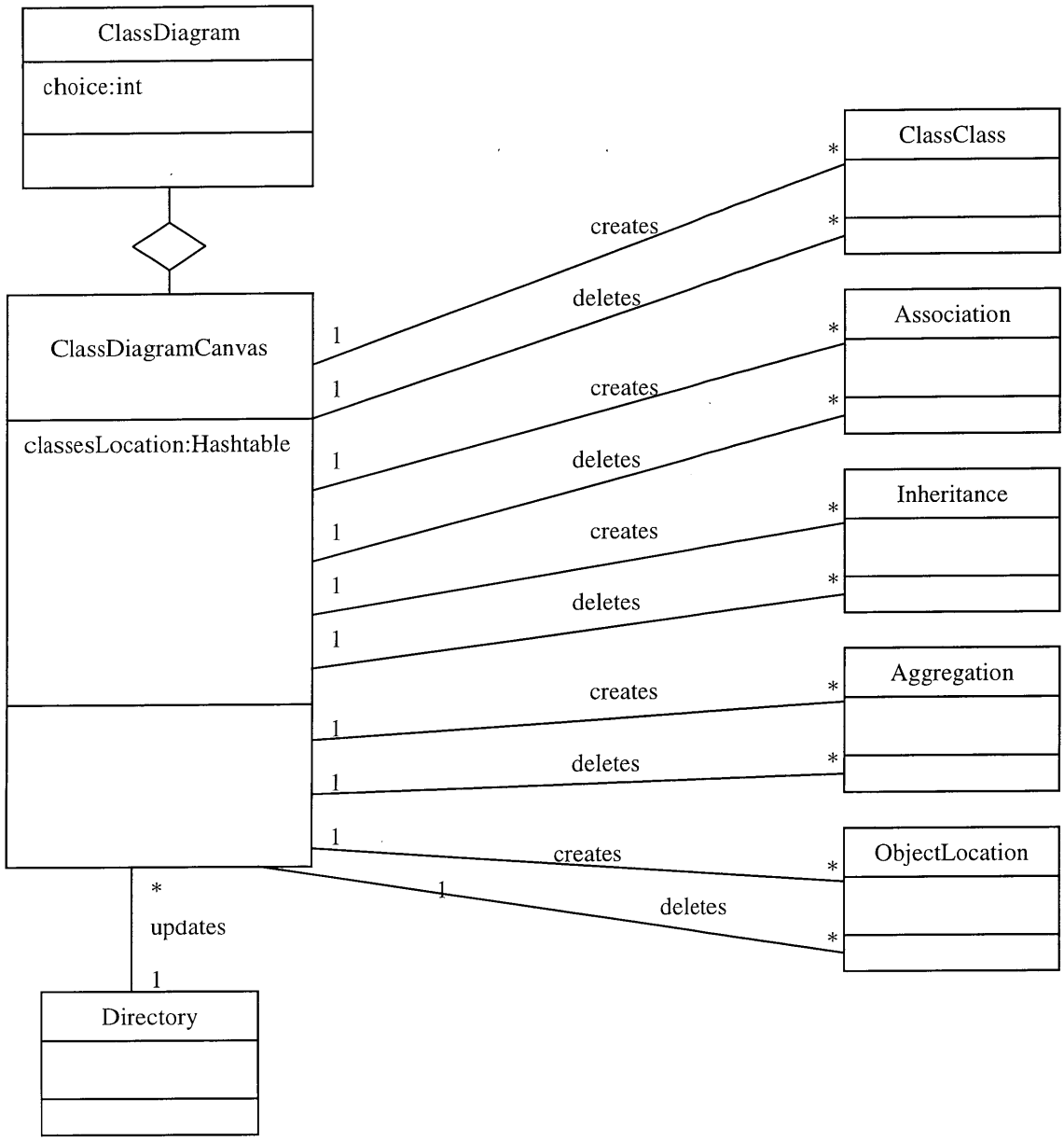


Figure 6-15 The Relationships of the ClassDiagramCanvas Class Diagram

6.3.6 ClassClass Class Diagram

Figure 6-16 shows the ClassClass Class Diagram. This diagram shows the components and attributes of the *ClassClass* class. The name *ClassClass* has been chosen because the word class is usually a reserved word in most programming languages and to avoid the confusion with other classes in the diagrams.

The *ClassClass* class has the attributes: *name* to hold the name of the class, *accessModifier* to hold the access modifier of the class such as public or private, *documentation* to save the description about the class, *cardinality*, and *width* and *height* to hold the dimensions of the class in the diagram.

The *ClassClass* class has many *getter* and *setter* methods to access and update its attributes. In addition, it has *draw* method to draw the class in a diagram.

The *Attribute* class is needed to hold all the information about an attribute of a class. In addition, the attributes with access modifier, protected, are inherited by the children classes in an inheritance relationship. This makes the use of a class to represent an attribute the most optimum choice so that the operations needed to handle the requirements of the inheritance relationship can easily applied to the attributes.

The *Method* class holds attributes, which represent its information and contains the *Parameter* Class, which holds the information about the parameters of a method.

The *ClassClass* class contains the classes *Attribute* and *Method*. In addition, it has a recursive aggregation relationship because it contains classes of the same type, which are the part and parent classes. If an aggregation relationship exists between two classes in

the Class Diagram, the container class needs to have a reference to the part class, which is useful for code generation. The existence of an inheritance relationship is a Class Diagram, requires that the child class inherits the attributes and methods of the parent class. This makes need of the availability of a reference in the child class to the parent class necessary to draw the diagram and to update the child information.

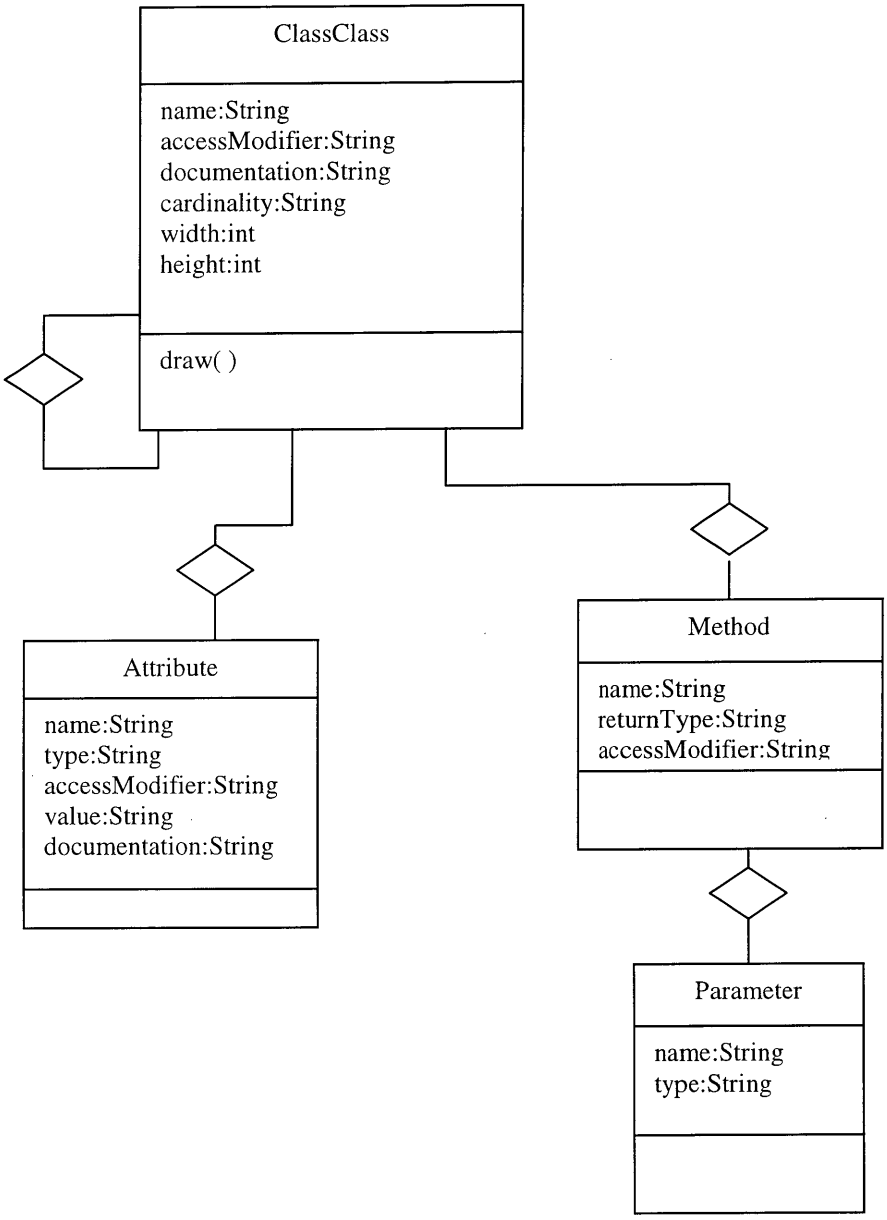


Figure 6-16 ClassClass Class Diagram

6.3.7 Relationship Class Diagram

The Relationship Class Diagram (Figure 6-17) is one of the most important Class Diagrams and involves many critical design decisions. It illustrates the relationships between the abstract class *Relationship* and other classes. The classes *Association*, *Inheritance*, and *Aggregation* inherit from the Relationship class because they have many common attributes and methods.

The *Relationship* class is an abstract class. It has the following attributes:

- *name* to hold the name of the relationship. An association relationship must have a name but the name of inheritance or aggregation relationships is optional.
- *cardinalityA* to hold the cardinality of class A (the first class associated with the relationship).
- *cardinality B* to hold the cardinality of class B (the second class associated with the relationship).
- *classesPoints* to hold the location of the classes participating in the relationship in the diagram.
- *points* which holds the points of the relationship line in the diagram. Since the line can be a polyline we needed a vector to hold the points.

The *Relationship* class has many methods such as the abstract method *draw*, *setPoints* which stores the points of the relationship line specified by the user in the diagram, *findTheta* determines the angle of the relationship's line from its points, *findSlope* determines the slope of the relationship's line, *findNamePosition* finds the location for displaying the relationship's name on the line, *findCardinAPosition* and

findCardinBPosition find the location of the relationship's cardinalities on the line, and equals determines if two relationships are the same.

The *ClassClass* class has three association relationships, *sends*, *receives*, and *sourceOf* with the Relationship class. The association relationship, *sends*, determines the first class specified by the user in the relationship. The association relationship, *receives*, determines the second class specified by the user in the relationship. The association relationship, *sourceOf*, specifies the original class source of the relationship which is important to know for drawing the diagram when the parent class is the one that participated in the relationship originally.

The children classes of the *Relationship* class override the *draw* method, which is responsible for drawing the relationship, since the relationships have different shapes.

The *Association* class inherits all the attributes and methods of the *Relationship* class. In addition, it has its own attributes and methods, which are specific for the association relationship. The private attributes of the *Association* class are *roleA* and *roleB* to store the role of classA and classB in the relationship. Other methods in the Association class are *findRoleAPosition* and *findRoleBPosition* which determine the location of roleA and roleB under the relationship's line.

Like the *Association* class the *Inheritance* class inherits all the attributes and methods of the *Relationship* Class. In addition, it has the following methods:

- *setChildAttributes* adds the protected attributes in the parent class to the child class.
- *setChildMethods* adds the public and protected methods in the parent class to the child class.

- *setChildAssociations* allows the child class to participate in an association relationship that the parent class participated in.
- *setChildAggregation* allows the child class to participate in an aggregation relationship that a parent class participated in.

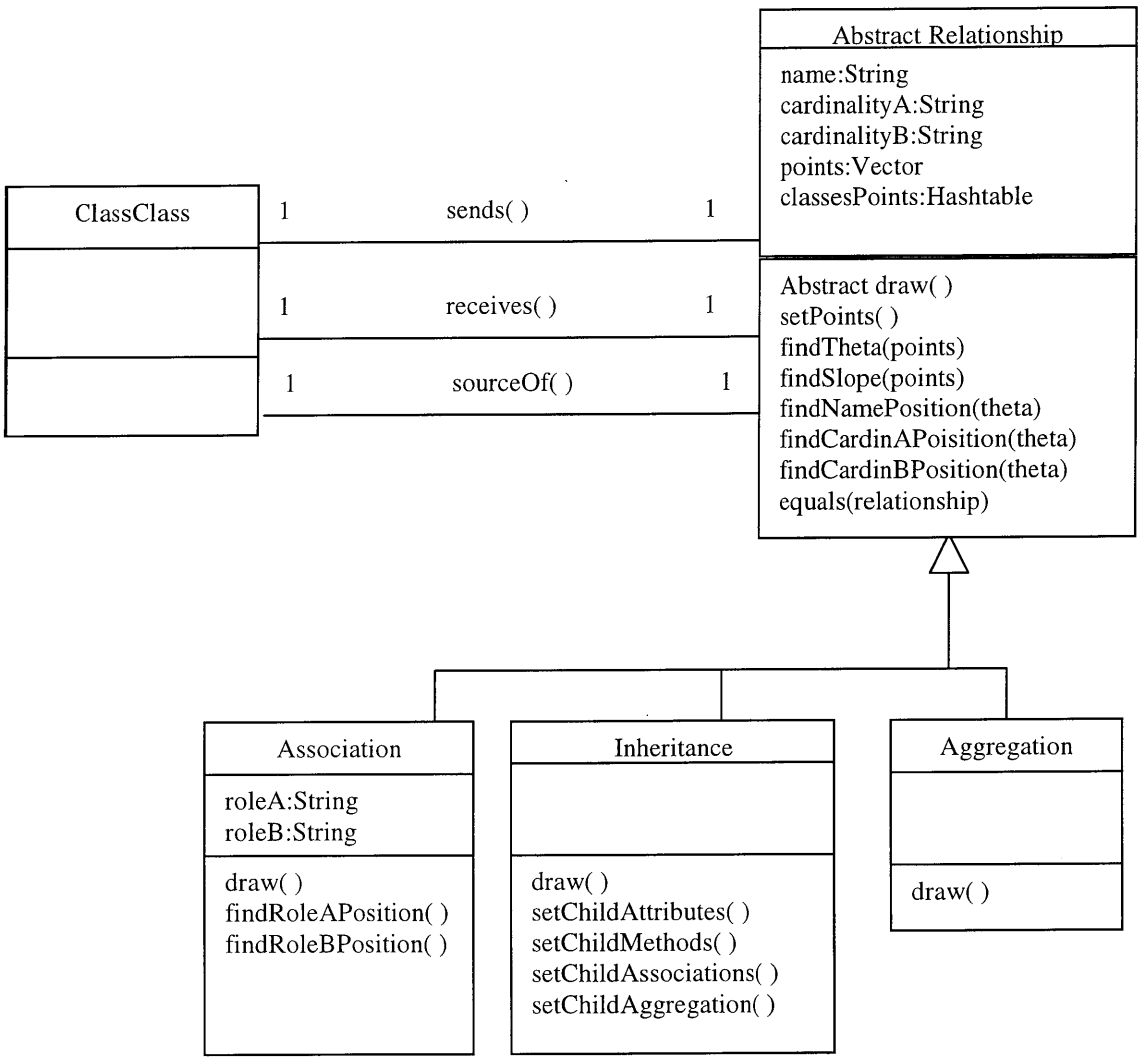


Figure 6-17 Relationship Class Diagram

6.3.8 Scenario Class Diagram

The structure of the Scenario Class Diagram reflects important objectives of the LCCASE tool. These objectives include the ability of the tool to link and reuse information in different models and maintain a consistency between the models. Figure 6-18 shows the Scenario Class Diagram. It contains a *Scenario* class, which inherits from the *Model* class as shown in Figure 6-9. In addition to the attributes and relationships inherited from the *Model* class, the *Scenario* class participates in an association relationship, *scenario-Of* with the *UseCase* class. The association relationship, *scenario-Of*, allows the *Scenario* class to access the use-case that corresponds to the scenario. This visibility allows the *Scenario* class to be aware of the actors of the use-case and the Use-Case Models contain the use-case.

The *Scenario* class, container, has an aggregation relationship with the *ScenarioCanvas* class. The *ScenarioCanvas* class has the association relationship *creates* and *deletes* with the *SystemOperation* and *ScenarioActorLocation* classes. These association relationships allow the *ScenarioCanvas* class to add or remove a system operation or actor to the scenario when the developer requests that. The association relationship *addsActor* with the *UseCaseModel* class allows the *Scenario* class to add an actor to a Use-Case Model when the developer requested adding an actor to the scenario that was not created in the project before. The association relationship *addsreference* with the *UseCaseModel* class allows the *Scenario* class to create a reference in the Use-Case Model that contains the use-case corresponding to the scenario when the developer added an actor to the scenario that is available in the same

Use-Case Model containing the use-case corresponding to the scenario but they are not linked via a reference. The *Scenario* class creates a reference between the use-case that corresponds to the scenario and the actor in the Use-Case Model that corresponds to the actor in the scenario. The two association relationships *addsActor* and *addsReference* are necessary to keep the information consistent in the project.

SystemOperation class inherits from the *Method* class because the system operations are methods in the Design Class Diagram, which will be developed during the design phase of the software life cycle. This will make the process of incorporating system operations in the Design Class Diagram easier since they have the attributes and methods of the *Method* Class beside their own attributes and methods. The attributes of the *SystemOperation* class reflects the fields of the contracts in the UML notation since this class is used to construct the contracts. The attribute, *time*, stores the position of the system operation in the sequence of the system operations in the scenario.

The *ScenarioActorLocation* class inherits from the *ObjectLocation* class. It saves the information about the location of the actor in the class and contains a method to draw the actor in the scenario since the actor in the scenario has different shape than the actor in the Use-Case Model diagram. The *ScenarioActorLocation* class has association relationships, *sends* and *receives*, with the *SystemOperation* class since the actor initiates or calls a system operation.

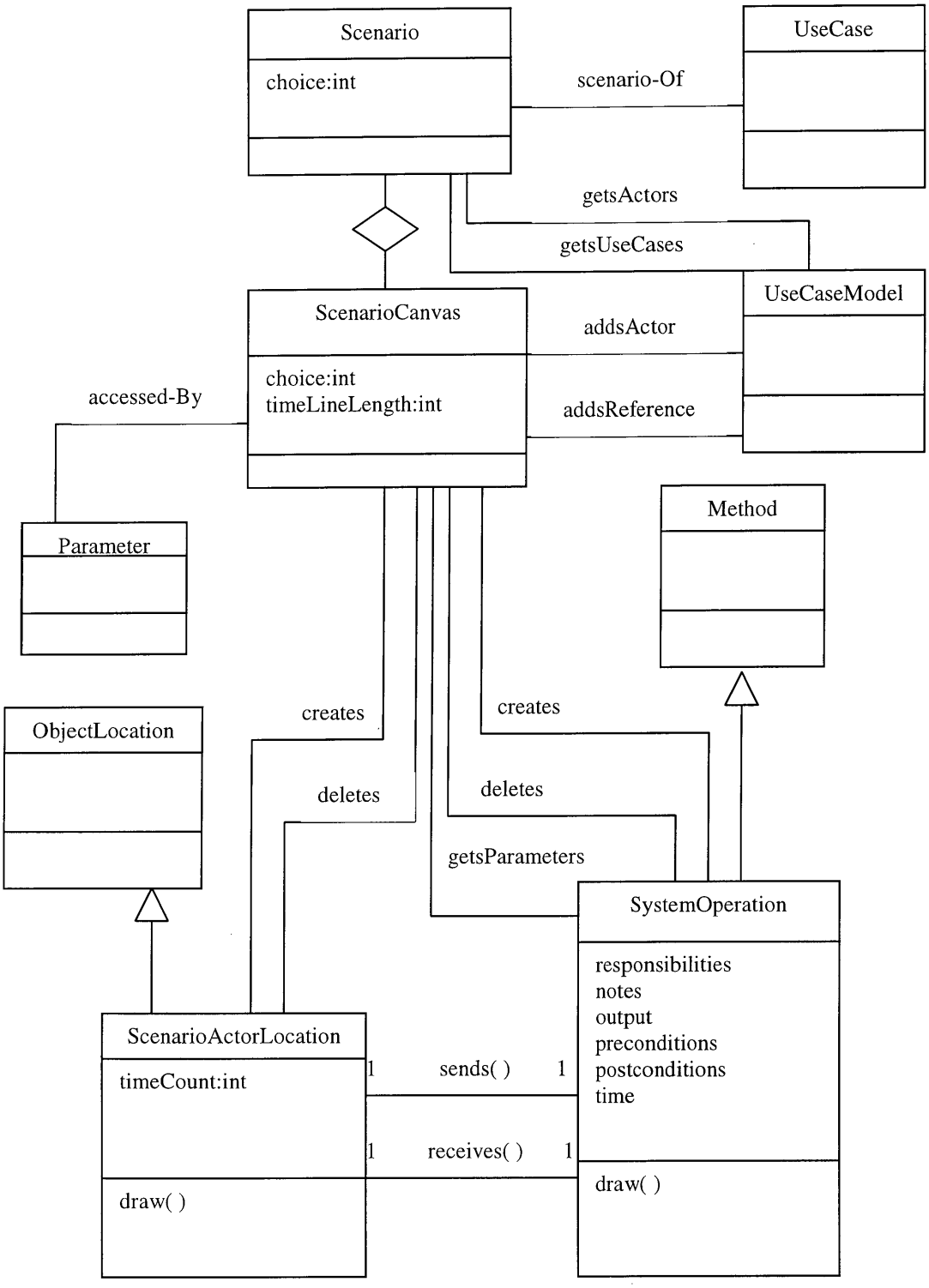


Figure 6-18 Scenario Class Diagram

6.3.9 Directory Class Diagram

The Directory Class Diagram (Figure 6-19) shows the *Directory* class and its relations with other classes. The *Directory* class is responsible for the operations of the Directory Subsystem discussed in chapter 4. It manages the activities performed within the Directory window.

The *Directory* class holds the attributes, *cards* and *tabs*. They are needed to manage the two views of the Directory window, which are the use-case and class views.

The *Directory* class has an association relationship, *updates*, with the *Database* class, which allows the *Directory* class to get up-to-date information from the *Database* class about the use-cases, classes, and other objects in the project and provides it with the means to allow the developer to delete the components of the project and inform the *Database* class of the change.

There are two aggregation relationships between the *Directory* class, container, which contains the classes *UseCaseDirectory* and *ClassDirectory*, components. The *UseCaseDirectory* class manages the operations of the directory within the use-case view and will be discussed in Figure 6-20.

The *Class*Directory class manages the operations of the directory within the class view. It has the attributes *classViewer*, *associationViewer*, *inheritanceViewer*, and *aggregationViwer*. Each one of them is responsible for displaying the names of the classes ands association, inheritance, and aggregation relationships in the directory and manages the operations of the developer.

The association relationships, *getsSender* and *getsReciever* allow the *ClassDirectory* class to know the classes participating in the relationships, association, inheritance, and aggregation.

6.3.10 AllDirectories Class Diagram

AllDirectories Class Diagram (Figure 6-20) shows that the classes *UseCaseDirectory* and *ClassDirectory* inherit from the *AllDirectories* class. This inheritance relationship allows the inclusion of the common attributes and methods in the parent class and it facilitate any future extension to the directory. The *AllDirectories* class has the attribute, *popup*. This attribute is a popup menu that allows the developer to delete an element in the directory from the database. The *AllDirectories* class has an association relationship with the *Database* class, updates.

The *UseCaseDirectory* class has the attributes *useCasesViewer*, *actorsViewer*, *scenarioViewer*, *systemOperationsViewer*, and *collaborationDiagramViewer* to display the name of the use-cases, actors, scenarios, system operations, and collaboration diagrams on the use-case view of the directory and manage the users activities.

The association relationship, *displays*, between the *UseCaseDirectory* and the *Scenario*, *SystemOperation*, *CollaborationDiagram* classes allows the Directory to display the windows of those classes. In addition, there is an association relationship, *deletes*, between the *UseCaseDirectory* class and the *UseCase* class to delete a use-case from the Database.

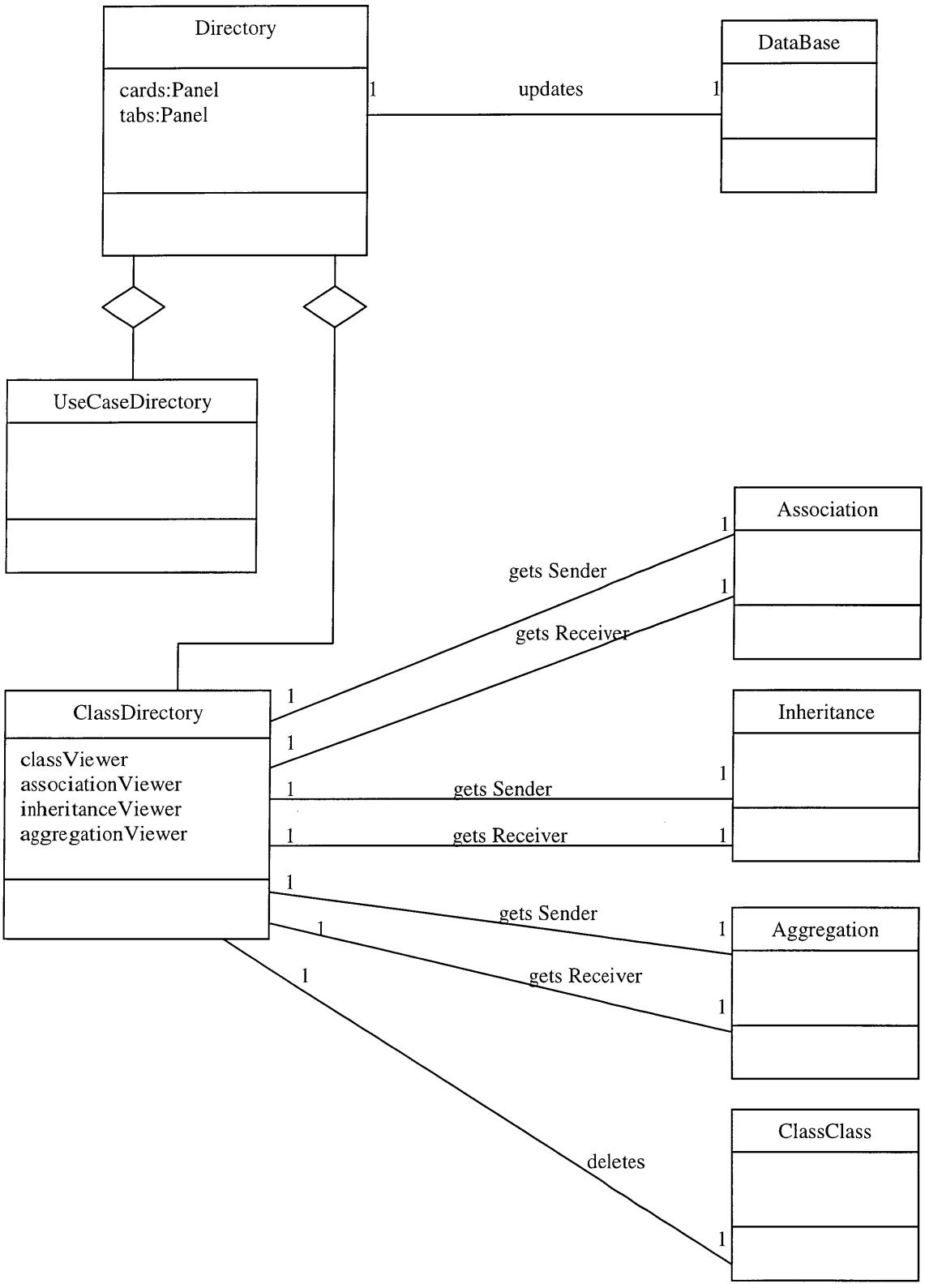


Figure 6-19 Directory Class Diagram

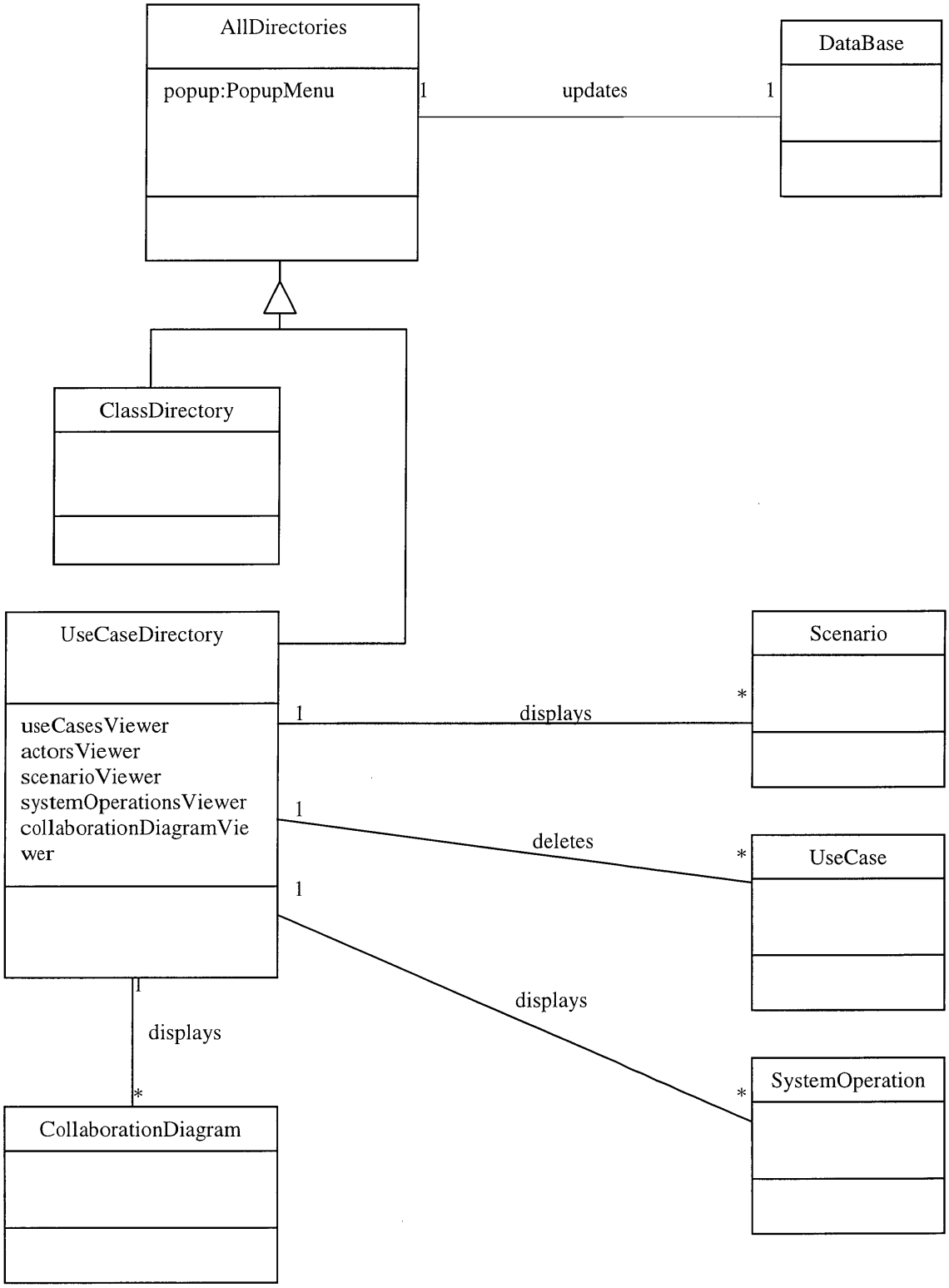


Figure 6-20 AllDirectories Class Diagram

6.3.11 Window Organiser Class Diagram

The Window Organiser Class Diagram (Figure 6-21) shows the *WindowOrganizer* class, which is responsible for organising the displaying of the models' windows. The association relationship, *getsInformation*, allows the *WindowOrganizer* class to get up-to-date information about the models in the project.

The association relationships, *organizesDisplaying* and *shows*, between the *WindowOrganizer* class and *Model* class allows the *WindowOrganizer* class to access the *Model* class and change the dimensions of its window so that multiple windows can be viewed at the same time.

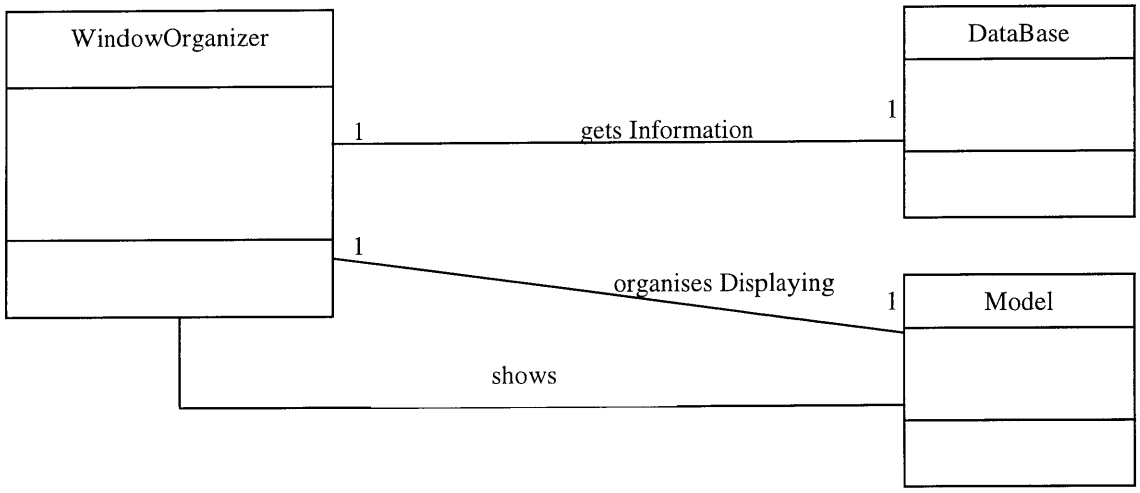


Figure 6-21 WindowOrganizer Class Diagram

6.3.12 Development Life Cycle Class Diagram

The Development Life Cycle Class Diagram (Figure 6-22) shows the *DevelopmentLifeCycle* class and its associations. The *DevelopmentLifeCycle* class is responsible for performing and managing the operations done by the Development Life Cycle subsystem described in chapter 4. The association relationships, *getsInformation*, allows the *DevelopmentLifeCycle* class to get the latest information about the project from the *Database* class. This is necessary to make the *DevelopmentlifeCycle* class able to generate the guidance to the developer about the models that are developed in the project and the models that should be developed. The association relationship, *getsSystemOperations*, allows the *DevelopmentLifeCycle* class to access the system operations in each scenario in the project. The association relationship, *creates*, instantiates an object of type *DevelopmentLifeCycleDialog* class when the developer requests any guidance.

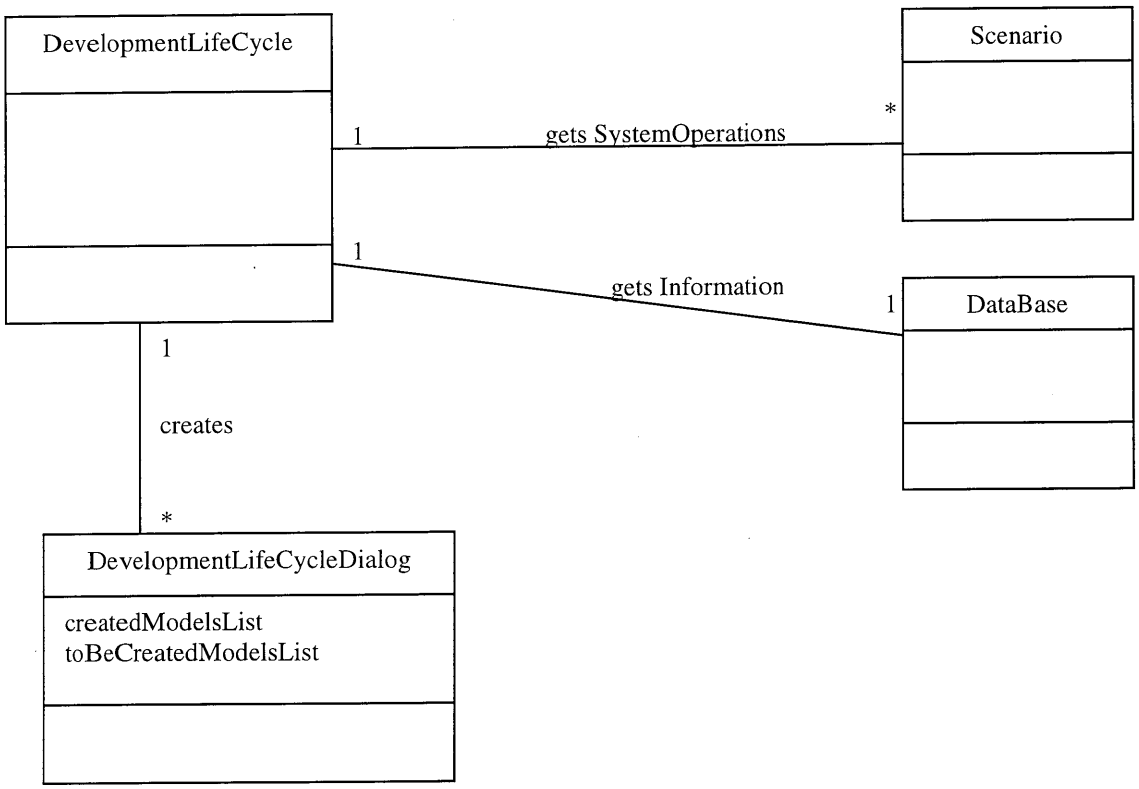


Figure 6-22 DevelopmentLifeCycle Class Diagram

6.3.13 Database Class Diagram

The Database Class Diagram (Figure 6-23) shows the *Database* class and its parts. The *Database* class works as a central repository of all the information in the project. It contains the classes *UseCaseModel*, *Reference*, *UseCase*, *Actor*, *Scenario*, *ClassDiagram*, *Association*, *Inheritance*, *Aggregation*, *ClassClass*, *CollaborationDiagram*, *DesignClassDiagrams*.

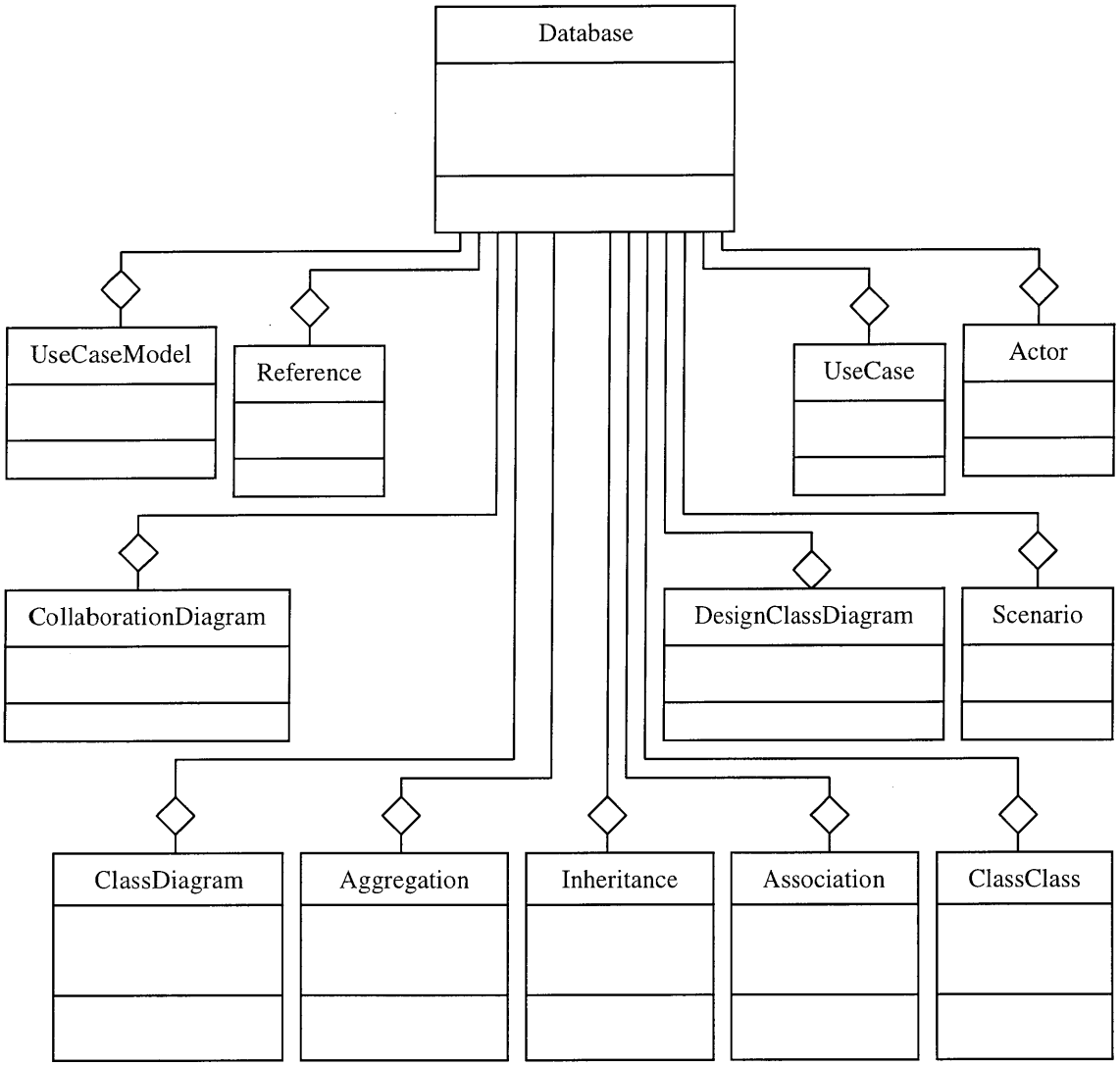


Figure 6-23 Database Class Diagram

6.4 Scenarios

This section shows and discusses the scenarios that I developed during the analysis phase of the LCCASE tool development life cycle. The scenario illustrates the events and system operations that occur between the actor, *Developer*, and the LCCASE tool. Their creation depends on the use-cases that are developed within the Use-Case Models. The use-cases specify how the actors interact with the LCCASE tool and the scenarios specify the LCCASE tool behaviour.

The nature of the LCCASE tool requires many interactions to occur between the Developer and the tool, which means the need for developing enormous number of scenarios. To simplify the presentation of the scenarios and make them more readable, the scenarios are divided into groups. Each group of scenarios corresponds to a Use-Case Model.

There is one actor in the scenarios, *Developer*, which is responsible for generating the system events. The names of the events and system operations in the scenarios are expressed with clear, simple, and meaningful words. They start with a verb to emphasise the operation between the actor and the LCCASE tool.

The following subsections discuss the nature of each group of scenarios. The scenarios are included in Appendix A.

6.4.1 Model Scenarios Group

The Model Scenarios group corresponds to the Model Use-Case Model. This group shows the operations that can be involved in creating or browsing a model.

Create New Model, Browse Model, and Delete Model scenarios correspond to the use-cases, *Create Model*, *Browse Model*, and *Delete Model* in the Model Use-Case Model. These scenarios involve many events and operations and by including them in a single scenario, the scenario will be difficult to understand and construct. Thus, separate scenarios are constructed for creating and editing the components of the models.

6.4.2 File Scenarios Group

The File Scenarios group corresponds to the File Use-Case Model. The scenarios in this group describe the operations that correspond to *Start new project*, *Open file*, and *Save file* use-cases.

6.4.3 Window Scenarios Group

The Window Scenarios group shows the scenarios of the use-cases in the Window Use-Case Model. The use-cases are *Tile windows across* and *bring a window to front*.

6.4.4 Development Life Cycle Scenarios Group

The Development Life Cycle Scenarios Group corresponds to the Development Life-Cycle Use-Case Model. The scenarios, Display the Development

Life Cycle and Checks a Model's Development Status, illustrate the operations of the use-cases, *Display Development Life Cycle* and *Check models' development status*.

6.4.5 Directory Scenarios Group

The Directory Scenarios group corresponds to the use-cases in the Directory Use-Case Model which are *display directory*, *select use-case panel*, *select use-case*, *delete use-case*, *select class diagram panel*, *select class*, and *delete class*.

6.4.6 Create UseCase Scenario

Figure 6-24 shows create UseCase Scenario, which is developed using the LCCASE tool. The Directory window displays the names of the actors that have references with the use-case corresponds to the currently developed scenario. This gives the developer an idea about the actors that should be added to the scenario.

Figure 6-25 shows the Scenario Dialog, which was displayed when the developer requested the creation of new Scenario. It displays the names of all use-cases in the project so the developer knows which scenarios should be created. Figure 6-26 shows the actor dialog, which was displayed when the developer requested adding an actor to the scenario. It displays the names of the actors that have references with the use-case.

The above discussion illustrated another example about the tool support during the software development.

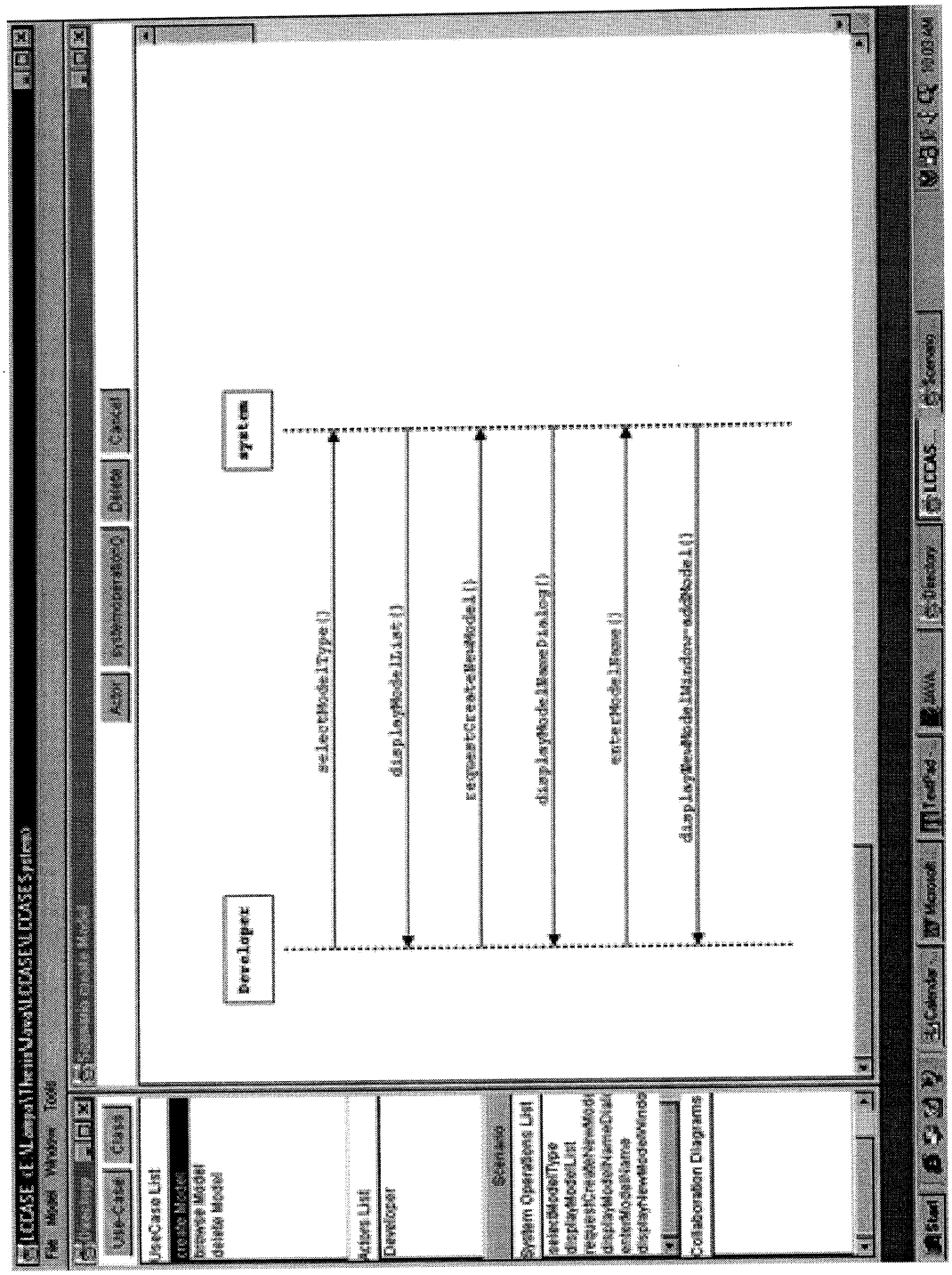


Figure 6-24 The Scenario, create Use-Case

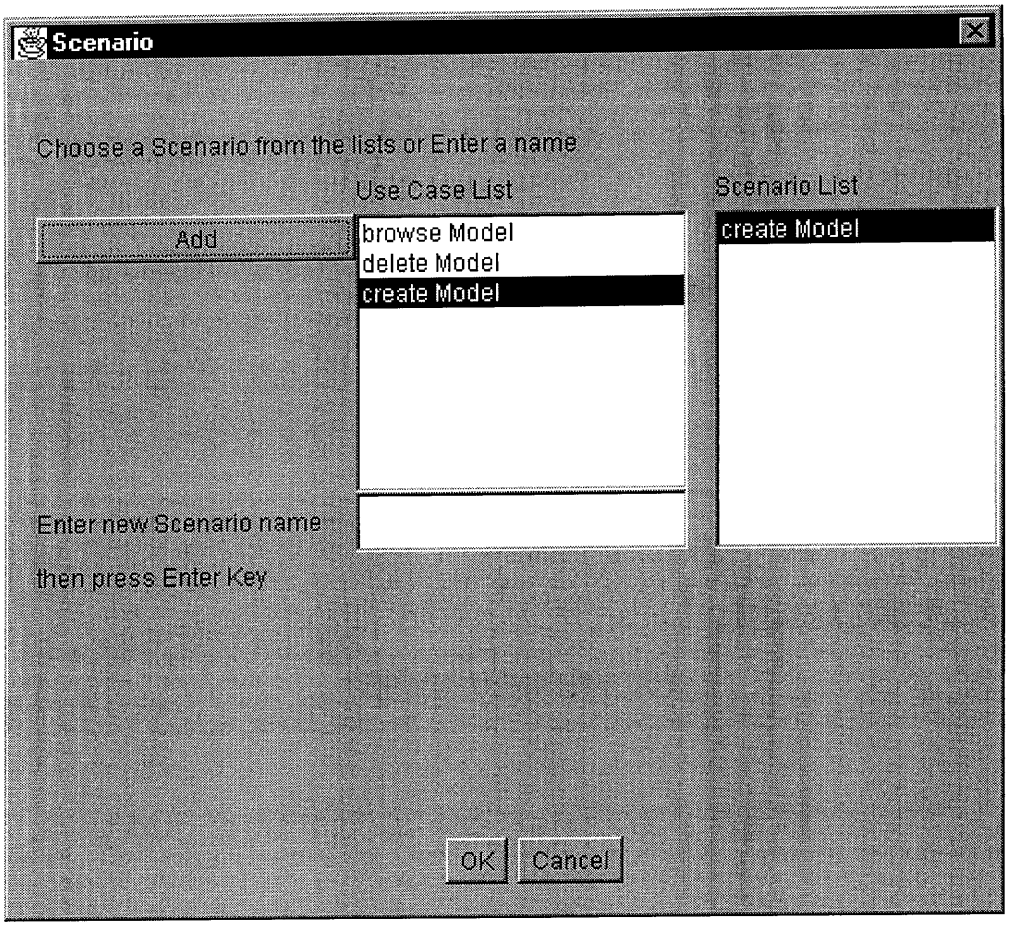


Figure 6-25 Scenario Dialog

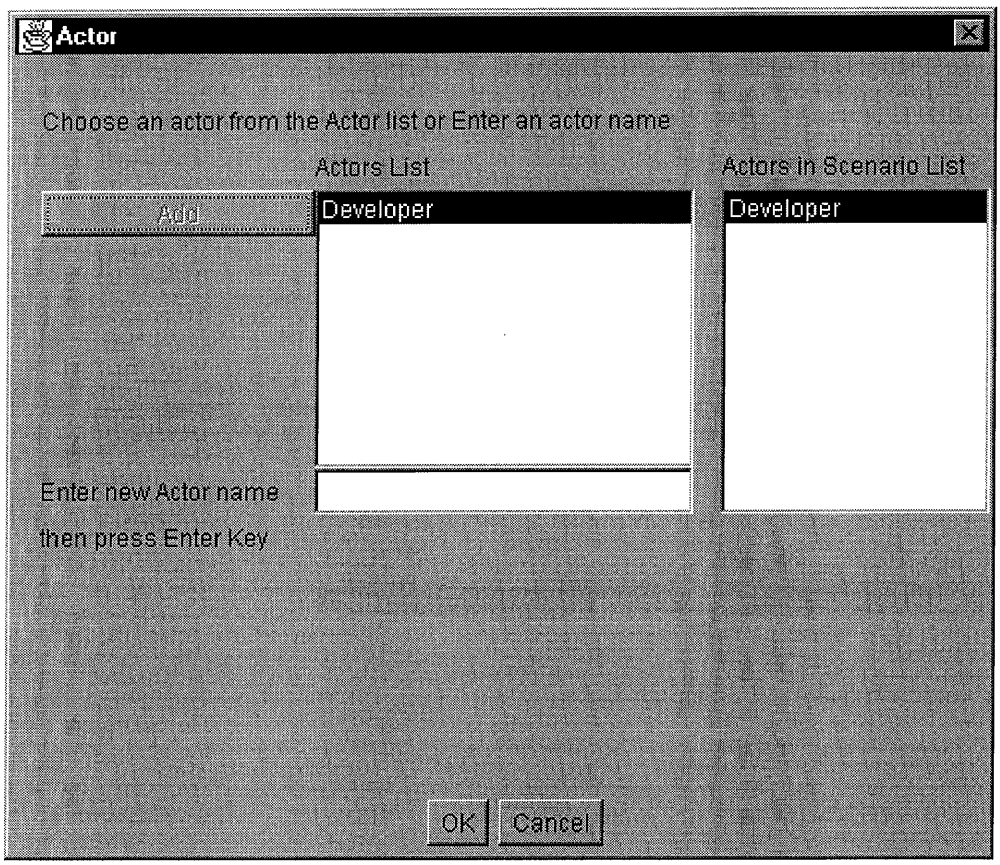


Figure 6-26 Add an Actor to the Scenario Dialog

6.5 Contracts

Contracts participate in defining the system behaviour. They discuss the effect of the operations on the system.

The contracts are documents created after adding the system operations to the scenarios. They provide a detailed description of the system operations and explain the changes in the state of the LCCASE tool when the system operation is processed. Each contract describes the system operation name, responsibility, the class contains the system operation or the method, cross-references, notes, outputs, preconditions and postconditions.

The contracts provide great help in writing the methods in the classes since the system operations are converted to methods in the design phase.

The Contracts in Appendix B are created to explain the system operations in the scenarios discussed in the previous section with a lot of care due to their importance in developing the design and implementation phases.

6.5.1 The Contract, addModel

Figure 6-27 shows the window of the Contract, addModel that is developed using the LCCASE tool. The name and cross-reference fields of the contract are updated automatically by the tool.

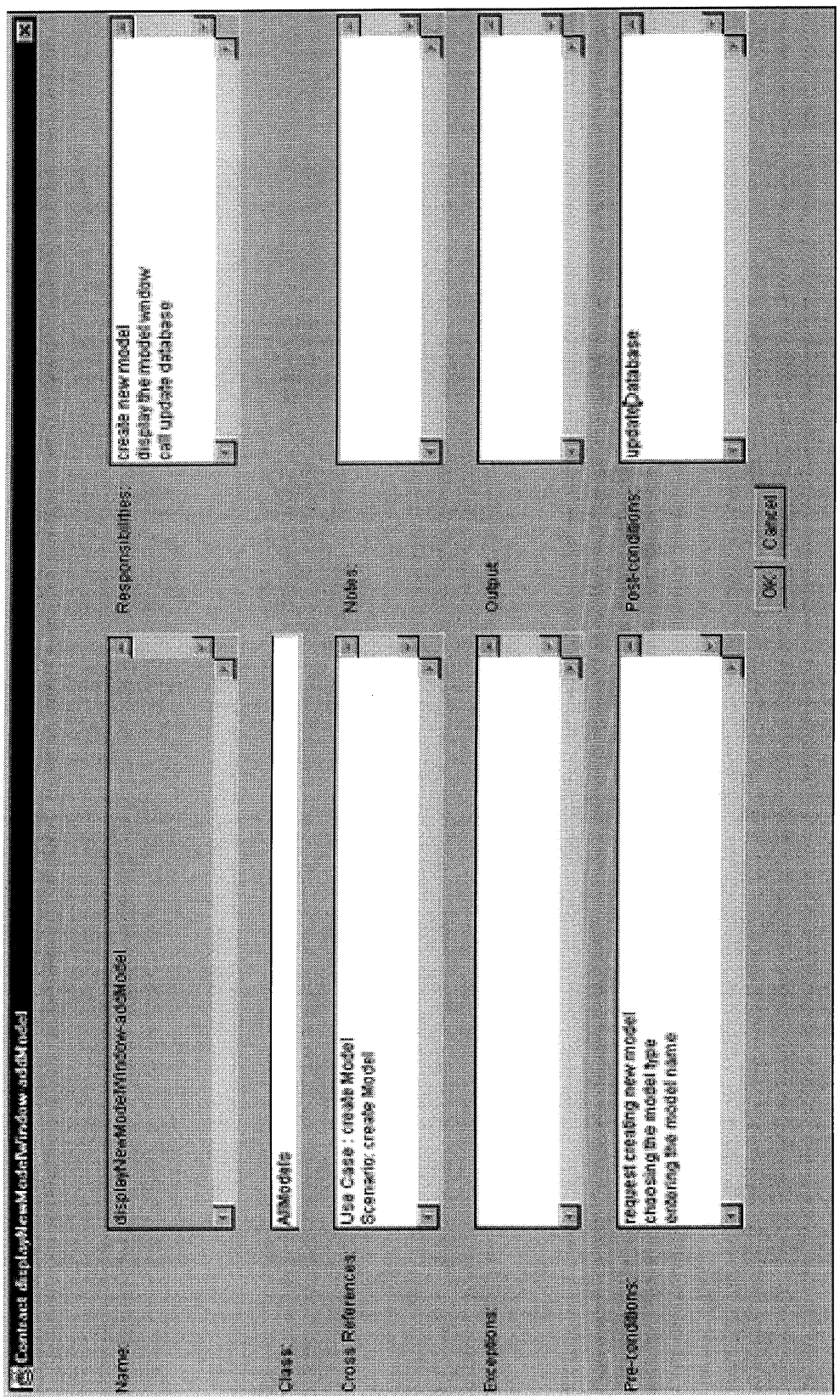


Figure 6-27 The Contract addModel

6.6 Collaboration Diagrams

Collaboration Diagrams are developed during the design phase of the LCCASE tool life cycle. Their creation depends on the class diagrams and system operations developed in the analysis phase.

The Collaboration Diagrams show the messages flow between the objects in the system to perform tasks. The first message in a collaboration diagram is a system operation that is described in a contract. The name of the collaboration diagram is the same as the name of the contract. The object that contains the system operation is called controller. The controller then sends messages to other objects, which called collaborators. The collaborators and the messages are determined from the postcondition entry in the contract of the system operation. The object name is the name of its class proceeded by colon and underlined. The messages are numbered according to their sequence of execution. A condition in a message is written between two square brackets. The collaboration diagrams contains high cohesion objects and maintain low coupling between the objects.

The development of the collaboration diagram consumed great amount of time during the LCCASE tool development. However, they provided great help during code development. The Collaboration Diagrams are included in Appendix C.

6.7 Design Class Diagrams

The Design Class Diagrams show the class diagrams, which are developed during the analysis phase, in the design phase context. The creation of the design class diagrams depends on the class diagrams and collaboration diagrams.

The Design Class Diagrams defines the visibility of the objects and their communication path. Moreover, it specifies the structure of the attributes and their interface, and the methods with their signature.

The Design Class Diagrams developed for the LCCASE tool are presented in Appendix D.

6.8 Coding

The code generation phase started after the development of the Design Class Diagrams. The process of generating code is dependent on the developed Collaboration Diagrams and Design Class Diagrams. They provided a great help during code generation. The Design Class Diagrams are translated to definitions of classes and the Collaboration Diagrams are mapped to methods' definitions.

Although that many important design decisions have been carried out during the analysis and design phases, which provided a significant support during code generation, the programming phase was not a trivial code generation. During the programming and testing changes have been made and some problems occurred and were resolved.

The iterative and incremental life cycle style encourages the development of several prototypes. The results of testing a prototype provided feedback to the next development cycle and next the analysis and design models are refined to fit the testing results. This

refinement process has been repeated until the final version of the LCCASE tool, which meets its objectives, has been achieved. The code has been written using the popular and efficient programming language, Java.

The following subsections explain problems occurred during the code and how they were resolved.

6.8.1 Flickering When Dragging an Object

The method, *paint( )*, in the classes of the models, draws the diagrams of the models. Flickering occurred when an object in the diagram was dragged. The reason behind that was that the canvas is cleared and the objects are redrawn every time the mouse moves to a new position during the dragging. To eliminate this problem, the complete diagram is first drawn in a background image and then the image is drawn in the canvas. Figure 6-28 shows the code for drawing a Class diagram.

```
public void paint(Graphics g)
{       // call update
        update(g);          }

public void update (Graphics g)
{       // find the width and height of the canvas
        int w = getBounds().width;
        int h = getBounds().height;
        if (backBuffer == null  ||  backBuffer.getWidth(null) != w
                                || backBuffer.getHeight(null) != h)
        //  backBuffer is an Image
        {       backBuffer = createImage(w, h);
                if (backBuffer != null)
                {       if (backG != null)
                        // backG is Graphics
                        {       backG.dispose(); }
                        backG = backBuffer.getGraphics(); }          }
        if (backBuffer != null)
        {       backG.setColor(getBackground());
                backG.fillRect(0, 0, w, h);
                backG.setColor(getForeground());
                // draw the association relationships in the background
                for (int n= 0; n < associations.size(); n++)
                {       Association ass = (Association) associations.elementAt(n);
                        // check if the association not inherited
                        if (ass.getRelationSource().equals(relationshipSource))
                        {       ass.drawassociation(backG);          }          }
                 // draw the inheritance relationships in the background
                for (int n= 0; n < inheritances.size(); n++)
                {       Inheritance inher = (Inheritance) inheritances.elementAt(n);
                        inher.drawInheritance(backG);       }
                // draw the aggregation relationships in the background
                for (int n= 0; n < aggregations.size(); n++)
                {       Aggregation agg = (Aggregation) aggregations.elementAt(n);
                        //check if the aggregation not inherited
                        if(agg.getRelationSource().equals(relationshipSource))
                                agg.drawAggregation(backG);       }
                // draw classes
                Enumeration e1 = classespoint.elements();
                // classespoint is a hashtable to store the locations of the classes in the diagram
               while (e1.hasMoreElements())
                {       ClassPoint up = (ClassPoint) e1.nextElement();
                        Point p = up.getClassPoint();
                        ClassClass u = (ClassClass) classes.get(up.getClassName());
                        u.drawClass(backG,p);     }
                // draw the diagram from the image
                g.drawImage(backBuffer, 0, 0, null);          }          }
```

Figure 6-28 The Method ClassDiagramCanvas.paint( )

6.8.2 Redraw the Inherited Relations

In theory, a child class in an inheritance relationship inherits protected attributes, protected and public methods, and association, inheritance, and aggregation relationships of its parent class. The tool implements this theory.

One of the design decisions of the tool is that the inherited attributes and methods are included in the child class in the diagram to make clear what the attributes and methods of a class are, especially if the parent class is included in another class diagram. However, drawing the inherited relationships makes the diagram complex. A condition is needed to differentiate the relationships that actually the class participating in and the relations inherited from the parent class. A class, *RelationshipSource* of type ClassClass included in and ClassClass when a relationship is created to point to the source of the relationship that is the same class or parent. Before drawing an association or aggregation relationship in the method, *paint ( )* shown in Figure 6-28, a condition is processed to check the *RelationshipSource* class.

The Inheritance class contains the methods, *setChildAssociations( )* and *setChildAggregations( )* which allows the child class to participate in an association or aggregation that its parent participated in. Figure 6-29 shows the methods setChildAssociations( ) and setChildAggregations( ).

```
public void setChildAssociations()
{       Vector associations = database.getallassociations();
        for (int i=0; i < associations.size(); i++)
        {       Association ass = (Association) associations.elementAt(i);
                //check if the parent involved in the above association as a receiver
                if (ass.getReciever().equals(receiver))
                // receiver in inheritance is the parent class
                {       //creates an association between the child and the sender of the association
                        Association childass = new Association(ass.getSender(),new Point(0,0),
                                                                new Hashtable(),receiver);
                        childass.setReciever(sender);
                        childass.setRelationshipName(ass.getRelationshipName());
                        associations.addElement(childass); }
                 //check if the parent involved in the association as a sender
                 else if (ass.getSender().equals(receiver))
                 {      //creates an association between the child and the receiver of the association
                        Association childass = new Association(sender,new Point(0,0),
                                                                new Hashtable(),receiver);
                        childass.setReciever(ass.getReciever());
                        //childass.setRelationSource(receiver);
                        childass.setRelationshipName(ass.getRelationshipName());
                        associations.addElement(childass); }        }       }
////////////////////////////////////////////
public void setChildAggregations()
{       Vector aggregations = database.getallaggregations();
        for (int i=0; i < aggregations.size(); i++)
        {       Aggregation agg = (Aggregation) aggregations.elementAt(i);
                // check if the parent class involved in an aggregation as a container
                if (agg.getReciever().equals(receiver))
                {       // creates an aggregation between the child, container and the component
                        Aggregation childagg = new Aggregation(agg.getSender(),new Point(0,0),
                                                                new Hashtable(),receiver);
                        childagg.setReciever(sender);
                        childagg.setRelationSource(receiver);
                        childagg.setRelationshipName(agg.getRelationshipName());
                        aggregations.addElement(childagg);      }
                 // check if the parent involved in an aggregation as a component
                 else if (agg.getSender().equals(receiver))
                 {      // create an aggregation between the child, component and the container
                        Aggregation childagg = new Aggregation(reciever,new Point(0,0),
                                                                new Hashtable(),receiver);
                        childagg.setReciever(agg.getReciever());
                        childagg.setRelationSource(receiver);
                        childagg.setRelationshipName(agg.getRelationshipName());
                        aggregations.addElement(childagg); }    }       }
```

Figure 6-29 Setting the Child Class Relations

6.8.3 Saving Images on Disk

The LCCASE tool supports storing the project in a file. In Java programming language, saving and retrieving of objects in a file can be done with the use of the Serializable interface. Thus the models in a project can be stored in a file in the form of objects. However, the images which are created in the method, *paint( )*, which is shown in Figure 6-28, of the models can not be saved in a file by this method. To resolve this problem, the images are set to null before storing them on disk and they are recreated after loading the project. Loops are used to set the images in all the models in the project to null. Figure 6-30 shows the code for setting the images to null.

```
else if(arg.equals("Save"))
{       FileDialog filedialog = new FileDialog (this, "Save", FileDialog.SAVE);
        filedialog.setFile("New Project");
        filedialog.setDirectory("E:\\LCCASE");
        filedialog.show();
        directoryname =filedialog.getDirectory();
        filename= filedialog.getFile();
        String filename1 = directoryname+filedialog.getFile();
        Hashtable classDiagrams =database.getclassdiagrams();
        Enumeration classDiagramsElements = classDiagrams.elements();
        while (classDiagramsElements.hasMoreElements())
        {       Class_Diagram cd= (Class_Diagram) classDiagramsElements .nextElement();
                cd.getClassDiagramCanvas().disposeImageGraphics();          }
                Hashtable usecaseModels =database.getusecasemodels();
                Enumeration useCaseModelsElements = usecaseModels.elements();
                while (useCaseModelsElements.hasMoreElements())
                {       UseCaseModel ucm= (UseCaseModel)
                                        useCaseModelsElements.nextElement();
                        ucm.getUseCaseModelCanvas().disposeImageGraphics();      }
                        Hashtable scenarios =database.getScenarios();
                        Enumeration scenariosElements = scenarios.elements();
                        while (eess.hasMoreElements())
                        {       Scenario ss= (Scenario) scenariosElements.nextElement();
                                ss.getSenarioCanvas().disposeImageGraphics();       }
                        try
                        {       ObjectOutputStream out = new ObjectOutputStream(
                                        new FileOutputStream(filename1));
                                out.writeObject(database);
                                out.close();
                                setTitle("LCCASE "+"<"+filename1+">");  }
                        catch(Exception e)
                        {       System.out.print("error: "+e);       }       }
```

Figure 6-30 Setting Images to Null Before Storing in File

6.9 The Implementation of the ClassClass Class

This section shows the development life cycle of the *ClassClass* class from the development of its Use-Case Model to writing the code. It shows how the models developed during the analysis and design phases are mapped to code.

In Figure 6-31, the Create Class Use-Case Model shows the actor, *Developer*, requests from the tool creating a class.

The Class Diagram of the ClassClass class discussed in the subsection 6.3.6. The use-case, *Create Class*, gave rise to the need of a class to hold the attributes and methods of a class. In addition, according to the Next Generation of the Fusion Method, a scenario should be developed for the *Create Class* use-case to describe the system operations and events passed between the *Developer* and the tool to create a class. Thus, the Scenario, *Create Class* has been developed. Figure 6-32 shows the Create Class Scenario.

Next, the Contract, createClass() has been developed. It shows important information and decisions regarding the method *createClass()*. A Contract in the Next Generation of the Fusion method corresponds to a method in the design phase. The importance of a contract comes from that it contains almost all the information needed to know about a method in a class. The inputs to the *createClass()* method are two integer numbers which represents the location of the class in the diagram since the *createClass()* method is called after the developer clicks on the Class Button to request creating class and then clicks in the diagram to specify the location of the class in the diagram which is specified in the precondition close. The postcondition close shows the methods that are called by the

createClass() method which can be in different classes to perform the tasks necessary to create class.

The creation of the Contract CreateClass was the last step in the analysis phase. Next, the design phase started by developing the Collaboration Diagram, createClass() which shows the calls of the method to other classes. The Collaboration Diagram, createClass() has been developed according to the information in the postconditions close of the Contract, createClass(). Figure 6-33 shows the createClass Collaboration Diagram.

The next step in the design phase is the development of the Design Class Diagram for the ClassClass class. Figure 6-34 shows the Design Class Diagram for the ClassClass class, which shows the declaration of the attributes in ClassClass. The ClassClass Design Class Diagram has been developed from the ClassClass Class Diagram in Figure 6-16 and many Collaboration Diagrams shown below.

The code for the ClassClass class has been developed during the implementation phase. The first step in writing the code for the ClassClass class was to define the attributes, which are included in ClassClass Design Class Diagram.

The next step was to write the definitions of the methods, which are determined from the Collaboration Diagrams and the ClassClass Design Class Diagram. The following are the methods that have been determined from the Collaboration Diagrams:

- The *ClassCanvas* class calls *creates( )* in the ClassClass class. This is determined from createClass( ) Collaboration Diagram which is shown in Figure 6-35. The Figure shows the constructor methods of the ClassClass class.

- The *Inheritance* class calls *setParentClass(class)* in *ClassClass* class which is determined from inheritance( ) Collaboration Diagram shown in Figure 6-36. The Figure includes the code of the method, *setParentClass(class).*
- The *ClassDiagramCanvas* class calls *addParentClass(Class)* in *ClassClass* class. This is determined from the aggregation( ) Collaboration Diagram shown in Figure 6-37, which includes the code of the method, *addParentClass(class).*
- The *ClassDiagramCanvas* class calls *getAttributes( ), getAttributeSource(name), removeAttribute(name), getMethods( ), getMethodSource(name), removeMethod(name), setParentClass(null)* in *ClassClass* class. This is determined from deleteInheritance( ) Collaboration Diagram in Figure 6-38, which includes the code of the methods.
- The *ClassDiagramCanvas* calls *removePartClass(name)* in *ClassClass* class. This is determined from deleteAggregation( ) Collaboration Diagram shown in Figure 6-39, which includes the code of the method, *removePartClass(name).*
- The *ClassDiagramCanvas* class calls *getClassWidth( )* and *getClassHeight( )* in *ClassClass* class. This is determined from *findClass( )* Collaboration Diagram shown in Figure 6-40 which includes the code for both methods.
- The classes, *Association*, *Inheritance*, and *Aggregation* call *getName( )* in *ClassClass* class. This is determined from the ClassDirectory.itemStateChanged( ) Collaboration Diagram shown in Figure 6-41.

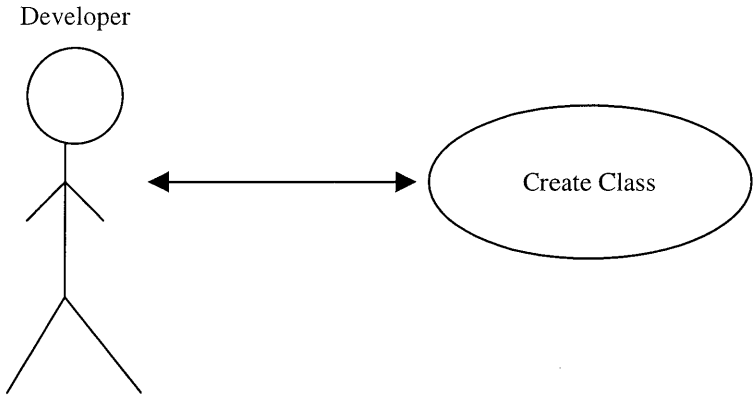


Figure 6-31 Create Class Use-Case Model

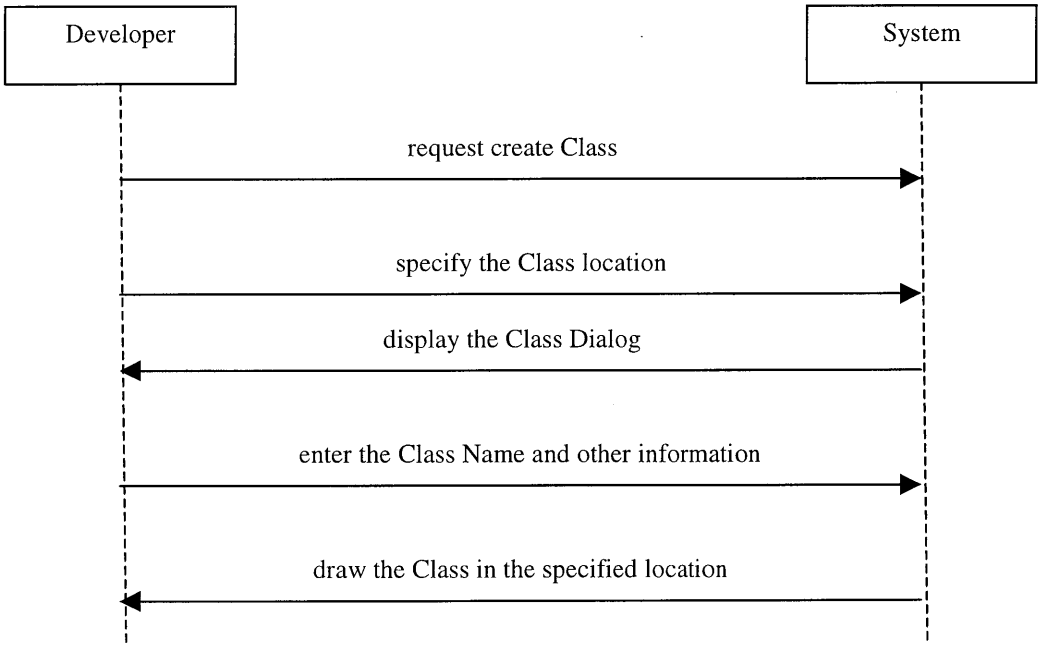


Figure 6-32 Create Class Scenario

Name:	createClass (x:int, y:int)
Responsibilities:	Create Class Draw Class
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = class AND mouseClicked to specify the location of the class
Postconditions:	new ClassClass new ClassDialog if Class does not exist in the database database.addClass(name,Class) new ClassLocation(x,y,name) directory.addClass(Class) else if Class does not exist in this ClassDiagram getClass(name) new ClassDialog new ClassLocation(x,y,name) else getClass(name) new ClassDialog paint( ) resetChoice( )

Figure 6-33 createClass( ) Contract

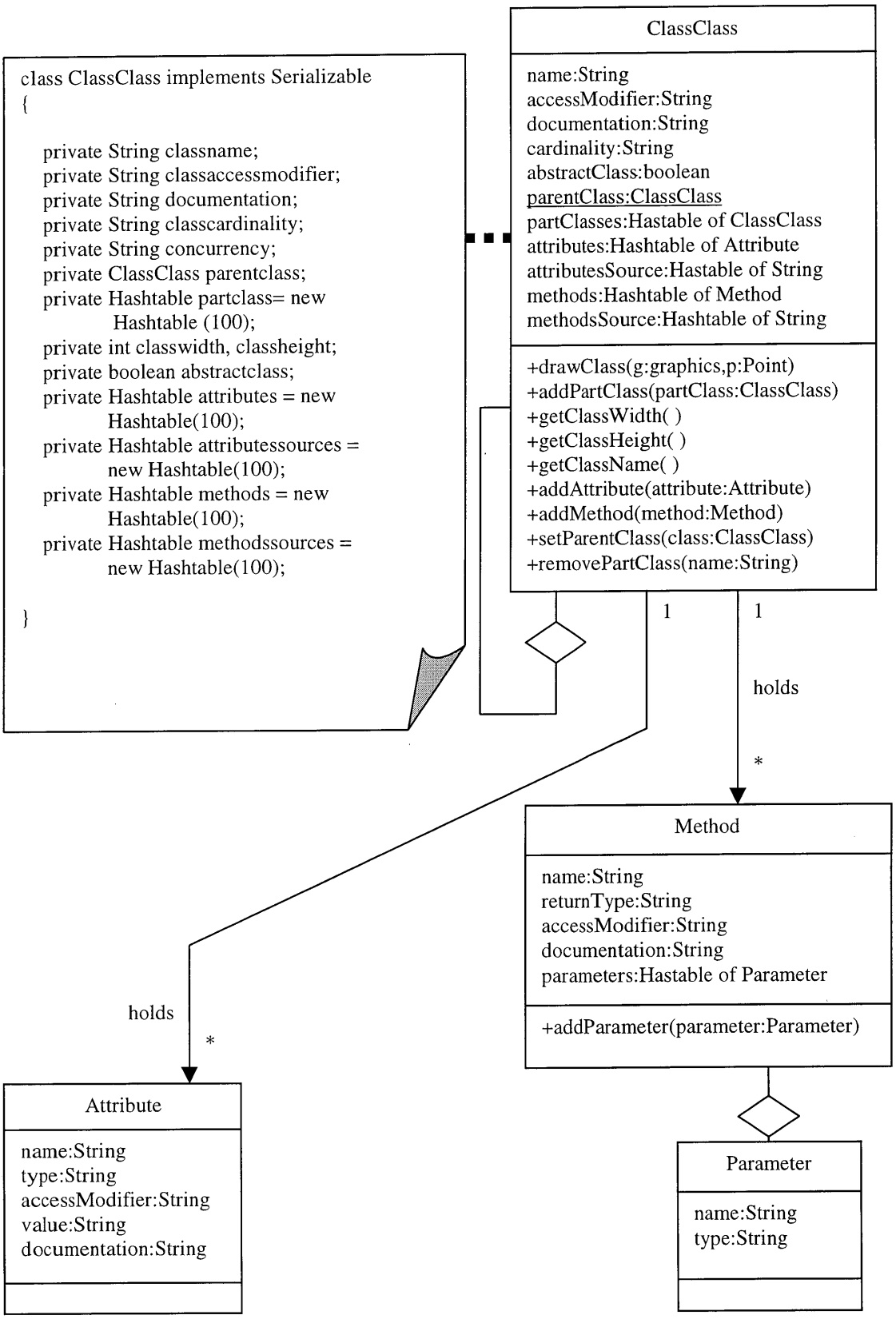


Figure 6-34 ClassClass Design Class Diagram

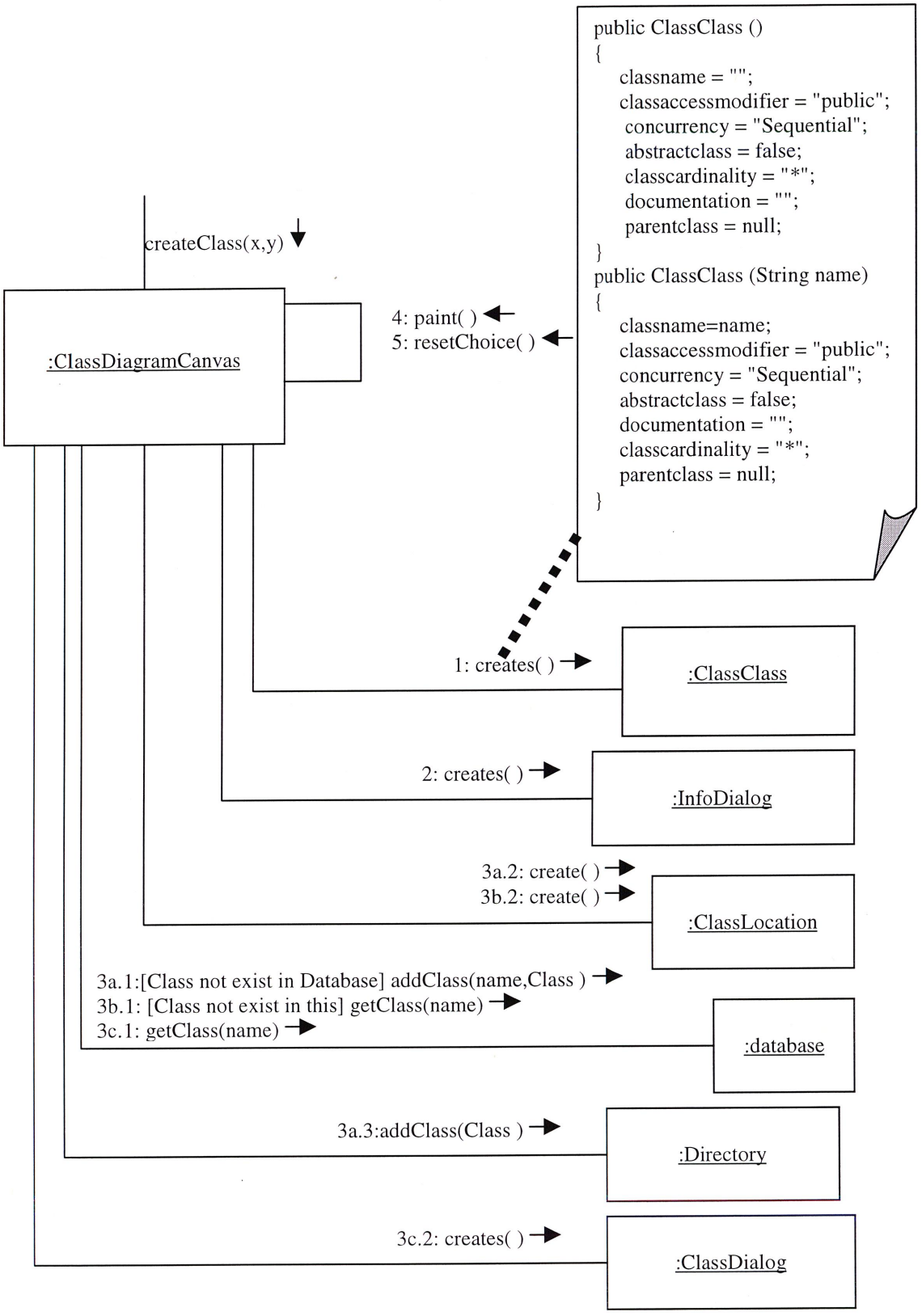


Figure 6-35 createClass( ) Collaboration Diagram

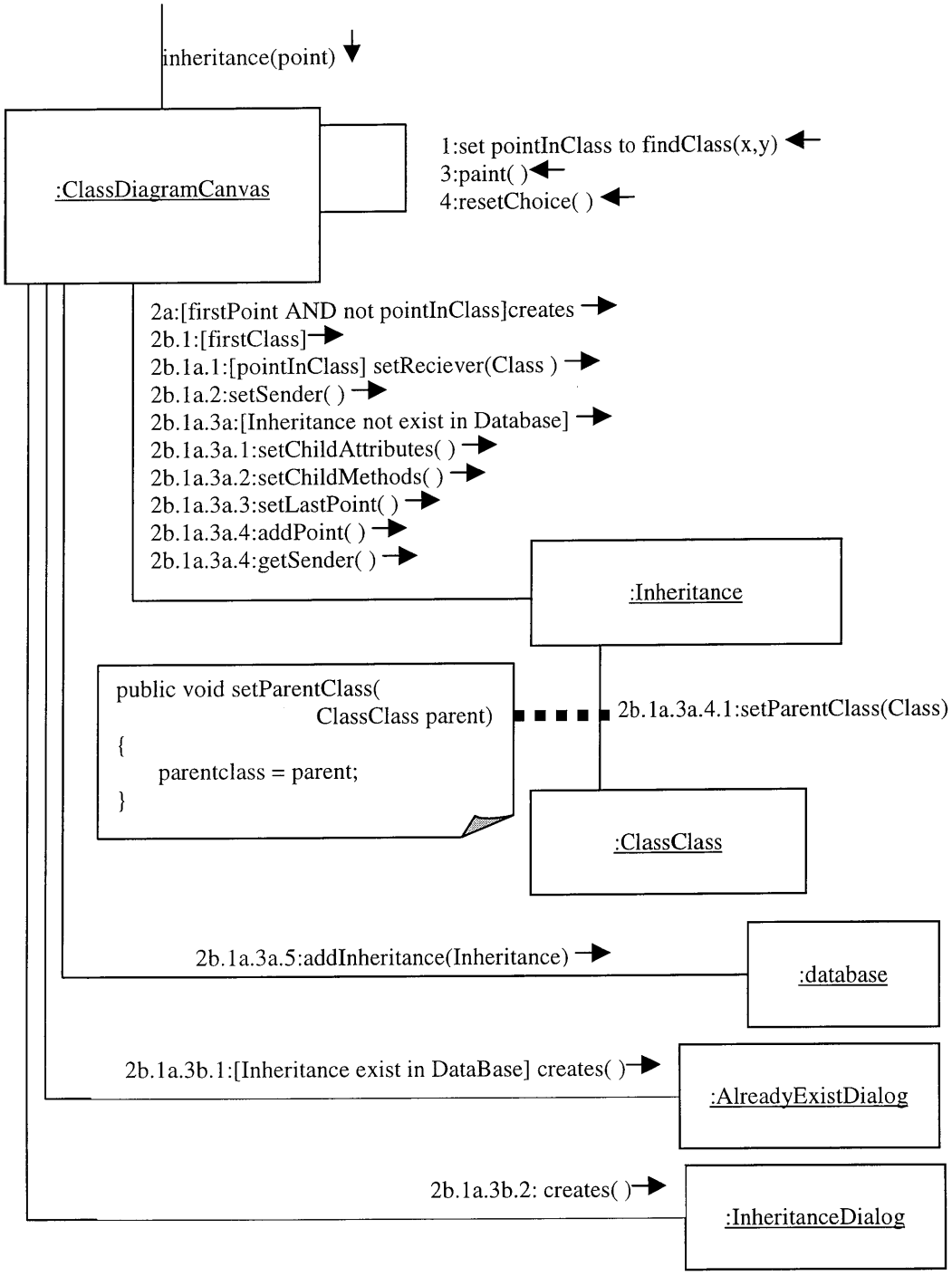


Figure 6-36 inheritance( ) Collaboration Diagram

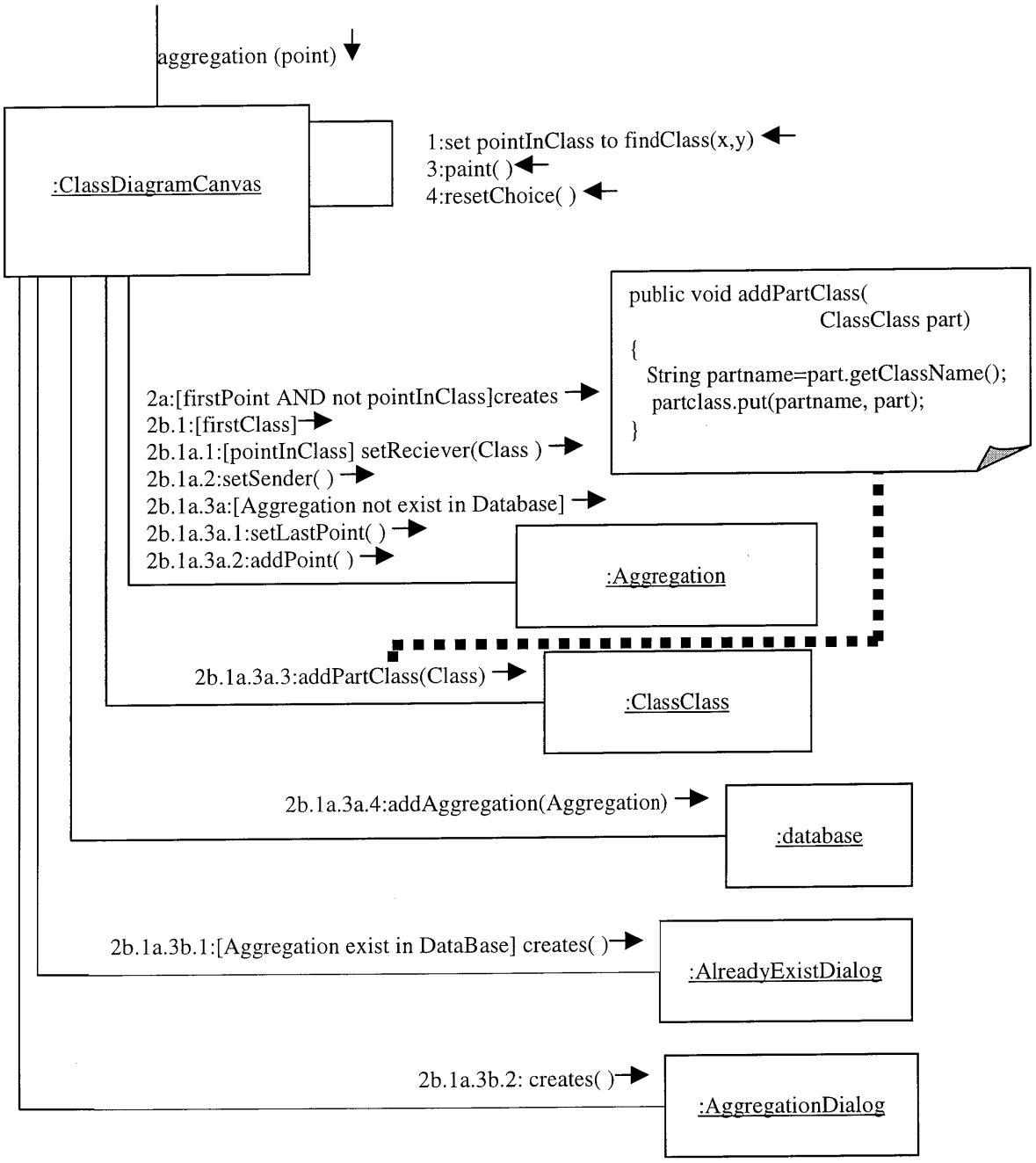


Figure 6-37 aggregation( ) Collaboration Diagram

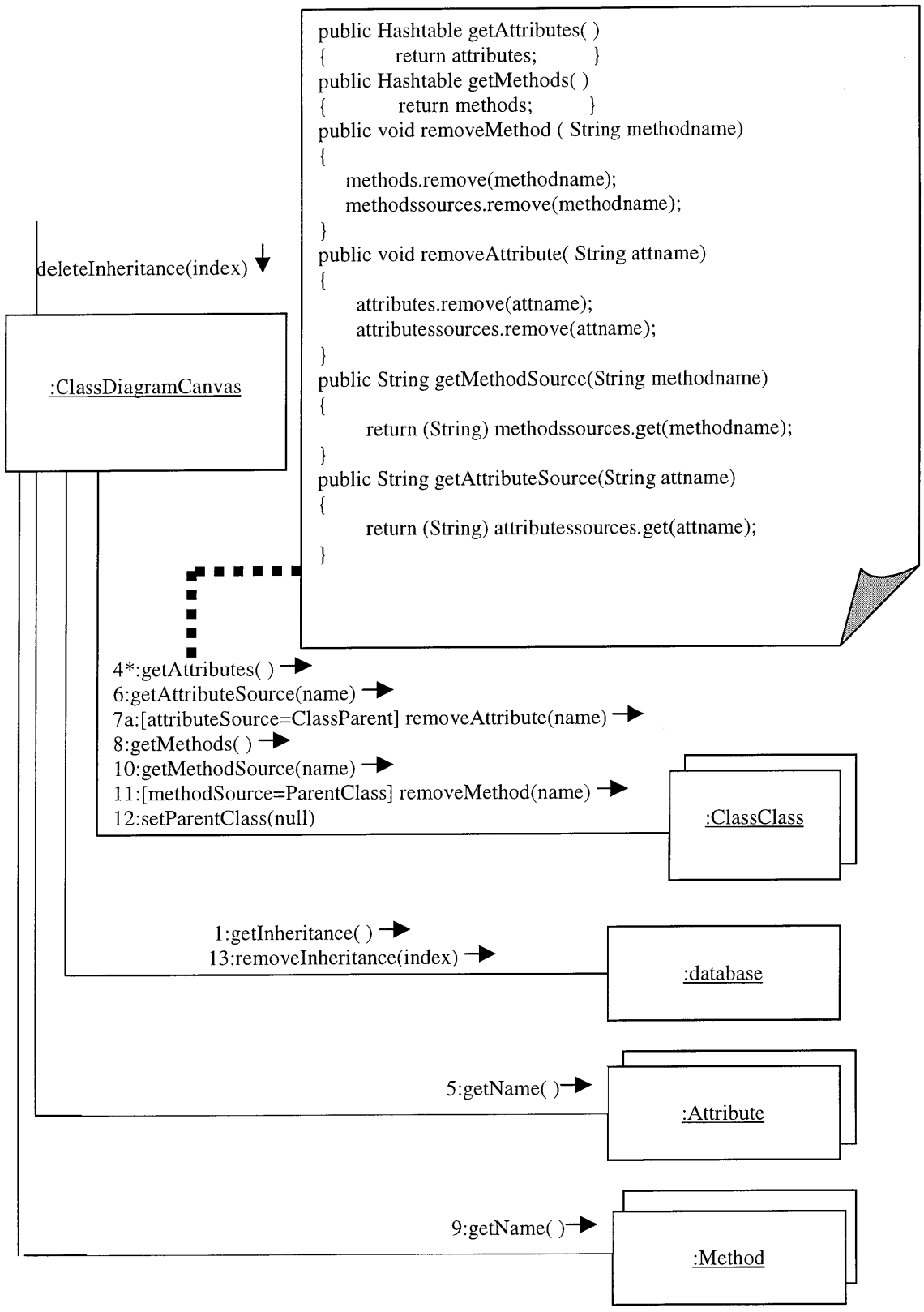


Figure 6-38 deleteInheritance( ) Collaboration Diagram

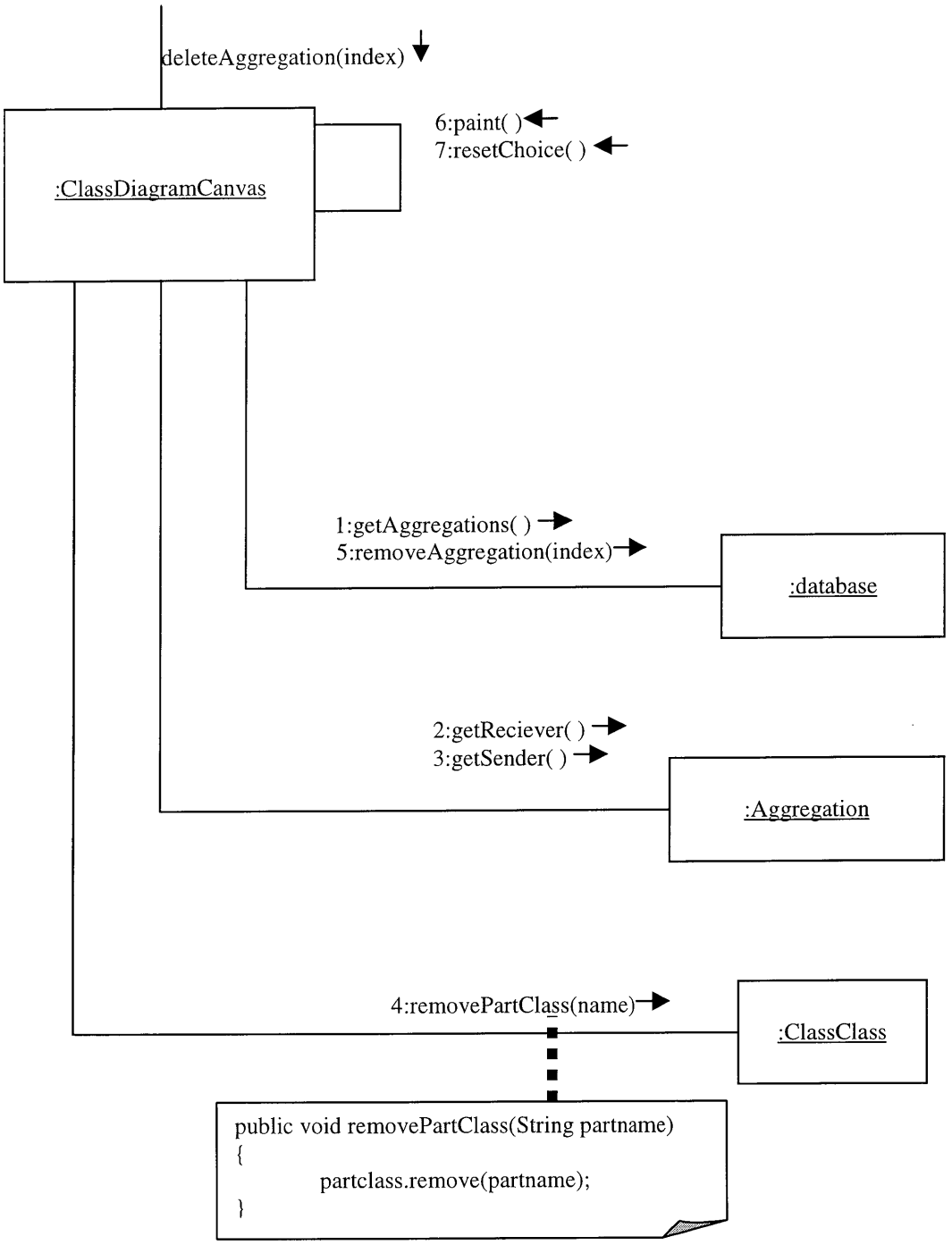


Figure 6-39 deleteAggregation( ) Collaboration Diagram

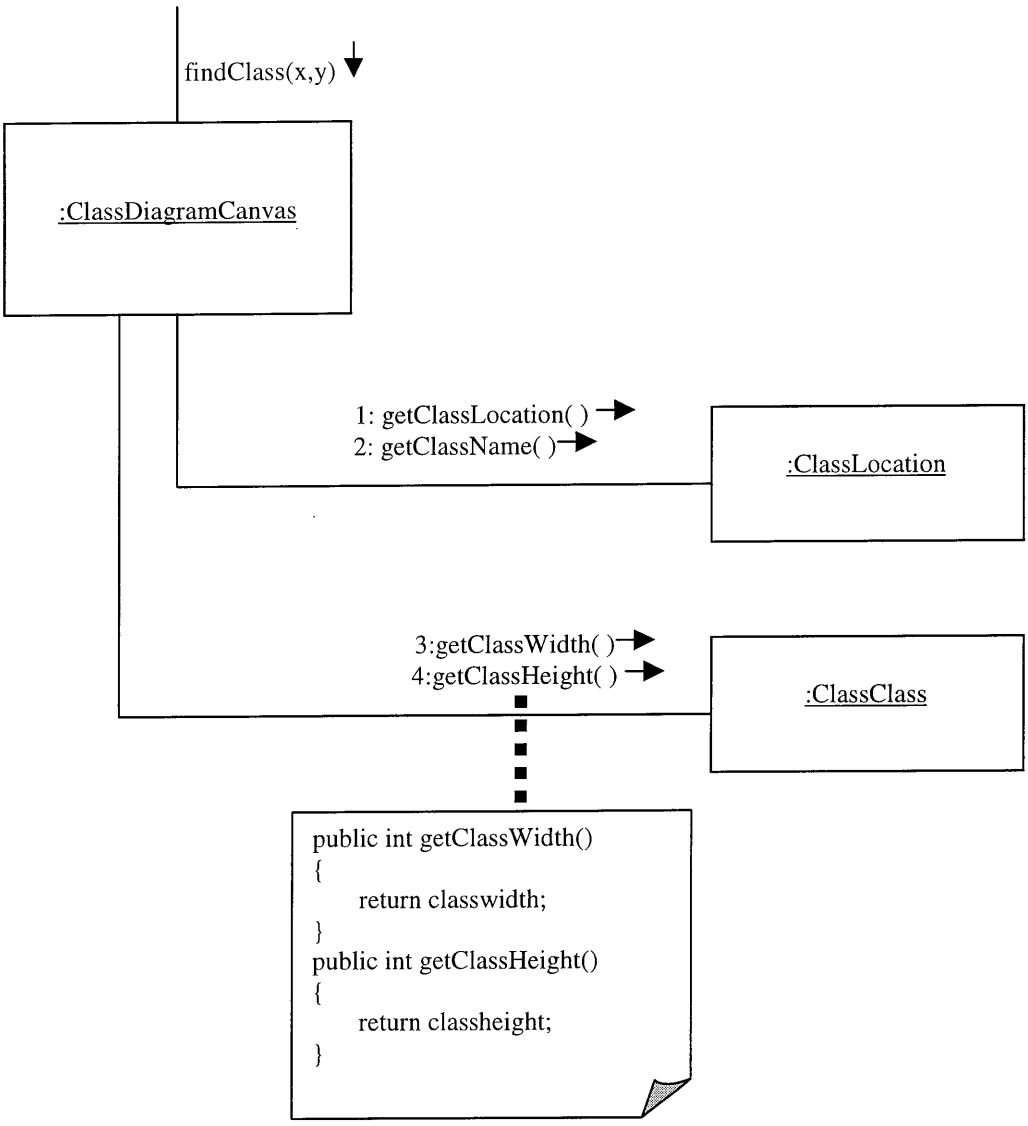


Figure 6-40 findClass( ) Collaboration Diagram

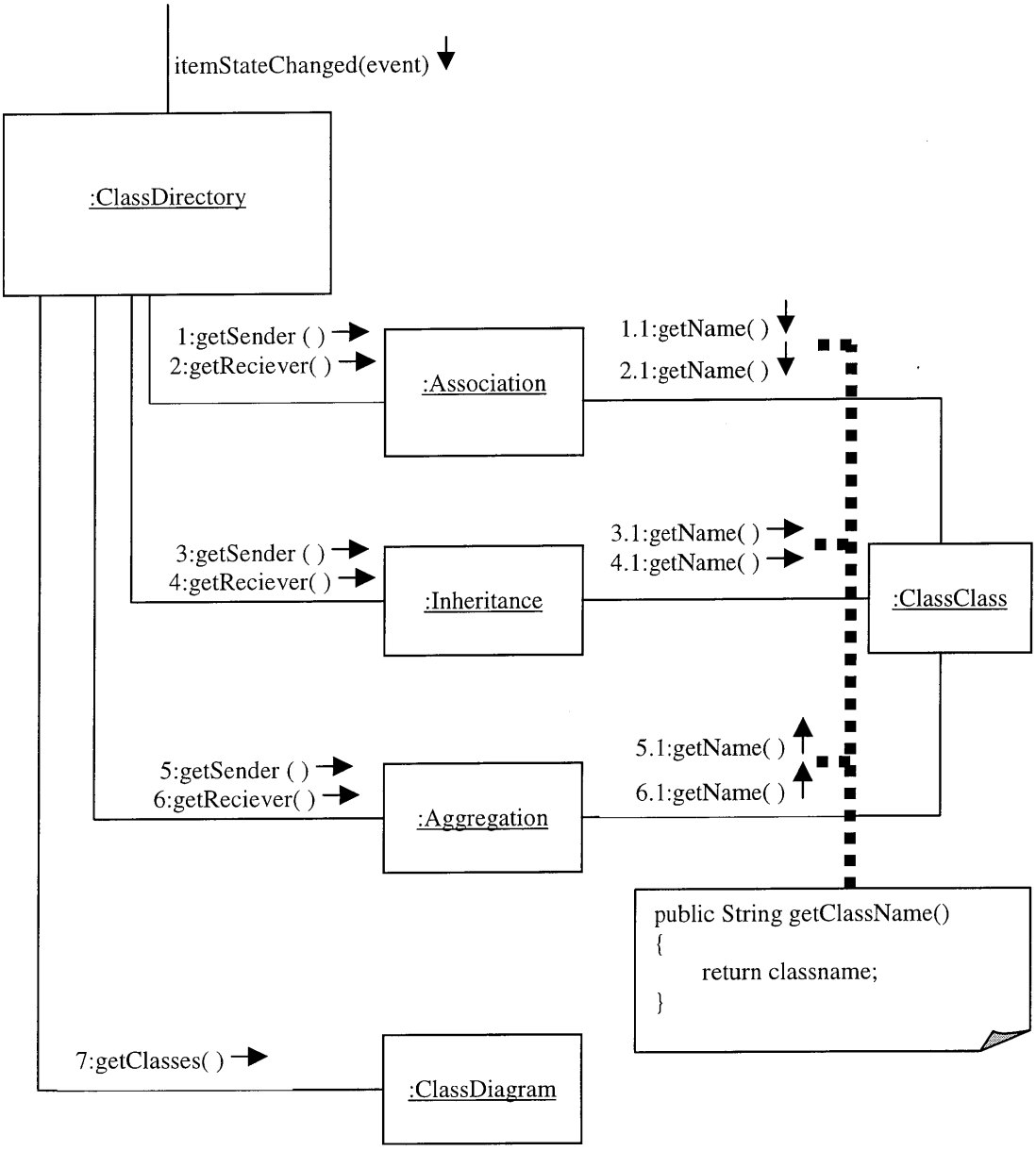


Figure 6-41 ClassDirectory.itemStateChanged( ) Collaboration Diagram

6.10 The Implementation of Linking the Information in the Models

The architectural design of the LCCASE tool includes a Database subsystem, which works as a central repository and provides other subsystems with up-to-date information in the project. The Database Subsystem facilitated the implementation of the link between the information in the project. Figure 4-10 shows the support of the LCCASE tool to the development of the models and shows the links between the models. These include:

- Line directed from a use-case to scenario. The LCCASE tool supports maintaining the correspondance between the use-cases and the scenarios. For each scenario in a project there should be a correspondant use-case. The implementation of this link is discussed in the sub-section 6.10.1.

- Line from the actors in a Use-Case Model to the actors in a Scenario. The actors in the Use-Case Model containing the use-case, which corresponds to the Scenario under development, are the nominated actors in the Scenario. Sub-section 6.10.2 discusses the implementation of this link.

- Line between the Class Diagram and Contract to check the consistency of the parameters and classes. The type of a parameter should be a type that is implemented in a programming language or a class that is defined in the project. If the type is not one of those mentioned before, the LCCASE tool reminds the developer of the conflict. The implementation of this link is discussed in section 6.10.2.

- Other links such as the ones between the Contract and Collaboration Diagram, Class Diagrams and Collaboration Diagrams, Class Diagram and Design Class Diagram, Contract and Design Class Diagram, and Collaboration Diagram and Design Class Diagram are going to be implemented within the future work.

The following sub-sections explain the implementation of the links between the information in the models. They describe the Class Diagrams, Scenarios, contracts, and Collaboration Diagrams, which are developed to implement the links.

6.10.1 Implementing the link between the Use-Cases and Scenarios

Figure 6-9 shows Model Class Diagram. In this diagram the *Scenario* Class inherits from *Model* class which means that there is an association relationship, *creates*, between *AllModels* class and *Scenario* class and *updates* between *UseCaseModel* class and *Database* class. The *updates* association relationship allows the *UseCaseModel* class to store use-cases in the database. The *creates* association relationship allows *AllModels* class to create a scenario. Since *AllModels* class contains *Database* class as shown in Figure 6-8, *AllModels* class obtains the information of the use-cases from the Database.

Figure 6-18 shows the Scenario Class Diagram. It contains a *Scenario* class, which inherits from the *Model* class as shown in figure 6-9. The *Scenario* class participates in an association relationship, *scenario-Of* with the *UseCase* class. The association relationship, *scenario-Of*, allows the *Scenario* class to access the use-case that corresponds to the scenario. This visibility allows the *Scenario* class to be aware of the actors of the use-case and the Use-Case Models contain the use-case.

In Figure 6-42 ModelListDialog Class Diagram, the *creates* association relationship between *AllModels* class and *ModelListDialog* class allows the *AllModels* class to create an object of the class, *ScenarioListDialog*, which has an inheritance relationship with *ModelListDialog* class.

Figure 6-43 shows *addModels ( )* Contract. The postconditions field specifies that if the request of the user is to create a scenario, then the LCCASE tool creates objects of the classes, *ScenarioListDialog* and *Scenario*, it then calls *addScenario( )* method. The sequence of the messages are shown in addModels Collaboration Diagram shown in Figure 6-44.

The ScenarioListDialog Contract, shown in Figure 6-45, explains the responsibilities of the scenarioListDialog( ) method. To fullfill those responsibilities, the method finds the use-cases that are created in the project and store them in a list, finds the scenarios that are created in the project and store them in a list, enables the developer to select a use-case to create a corresponding scenario or select a scenario to browse, and updates ModelInfo. The postconditions field shows that the ScenarioListDialog needs to access the database to obtain the use-cases and scenarios created in the project. the sequence of the messages in the postcondition fields are shown in ScenarioListDialog Collaboration Diagram in Figure 6-46.

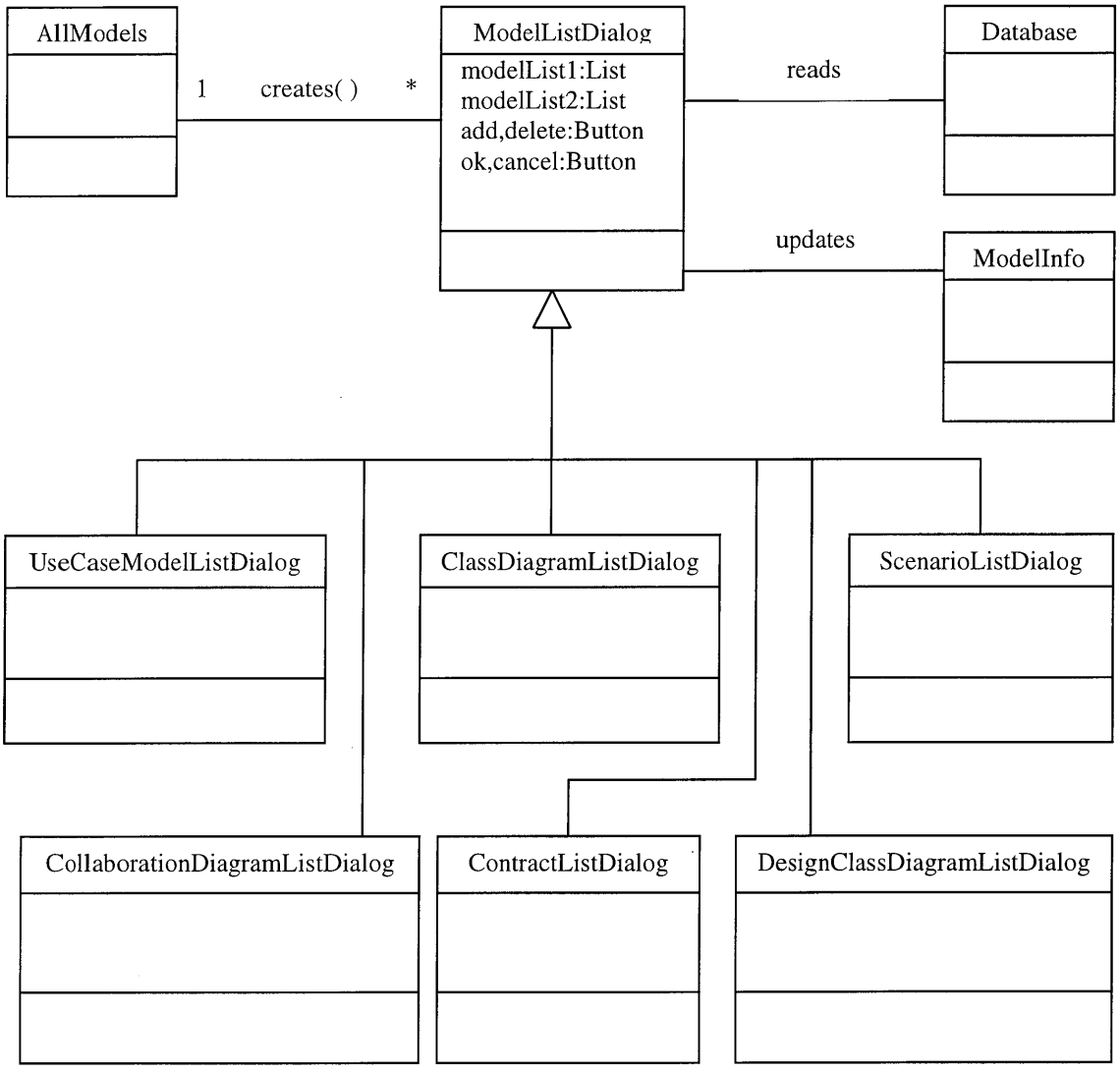


Figure 6-42 ModelListDialog Class Diagram

Name:	addModel (modelType:String, modelName:String)
Responsibilities:	Create the models and display them
Class:	AllModels
Cross Reference:	Model Use-Case Model CreateModel Scenario AllModel.actionPerformed() Contract
Notes	
Output:	Display the model frame
Preconditions:	if arg = Model AND addModel
Postconditions:	new ModelInfo If modelType = UseCaseModel new UseCaseModelListDialog(modelInfo, database) new UseCaseModel(name, database, allModels, directory) addUseCaseModel(useCaseModelName, UseCaseModel) else if modelType = ClassDiagram new ClassDiagramListDialog(modelInfo, database) new ClassDiagram(name, database, allModels, directory) addClassDiagram(classDiagramName, ClassDiagram) else if modelType = Scenario new ScenarioListDialog(modelInfo, database) new Scenario(name, database, allModels, directory) addScenario(scenarioName, Scenario) else if modelType = Contract new ContractListDialog(modelInfo, database) new ContractModel(name, database, allModels, directory) addContractModel(contractModelName, ContractModel) else if modelType = CollaborationDiagram new CollaborationDiagramListDialog(modelInfo, database)

Figure 6-43 addModel( ) Contract

Postconditions:	new CollaborationDiagramModel(name, database, allModels, directory) addCollaborationDiagramModel(collaborationDiagramModelName, CollaborationDiagramModel) else if modelType = DesignClassDiagram new DesignClassDiagramListDialog(modelInfo, database) new DesignClassDiagramModel(name, database, allModels, directory) addDesignClassDiagramModel(DesignClassDiagramModelName, DesignClassDiagramModel)

Figure 6-43 addModel( ) Contract (Continue)

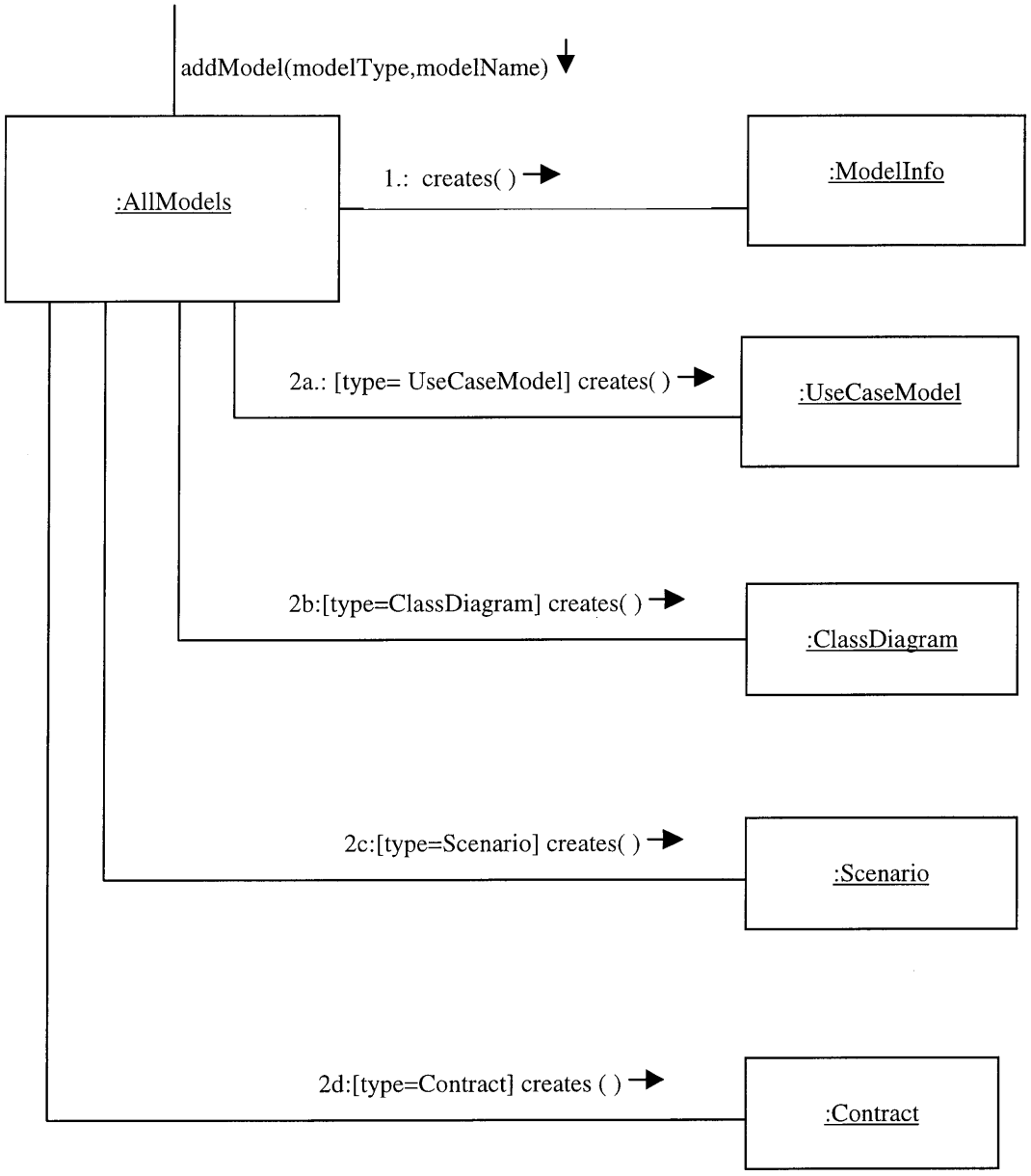


Figure 6-44 addModel( ) Collaboration Diagram

Name:	ScenarioListDialog (allModels:AllModels, modelInfo:ModelInfo, database:Database)
Responsibilities:	Displaying a list showing the use-case created in the project. Displaying a list showing the scenarios created in the project. Allows the developer to select a use-case to create a new scenario, or select a scenario to browse.
Class:	ScenarioList Dialog
Cross Reference:	AllModels.addModel() Contract
Notes **Output:**	Display a dialog displaying a list containing the names of the use-cases created in the project, list containing the names of the scenarios created in the project, a field to specify a scenario name.
Preconditions:	if arg = scenario
Postconditions:	database.getScenarios() database.getUseCaseModels()

Figure 6-45 scenarioListDialog( ) Contract

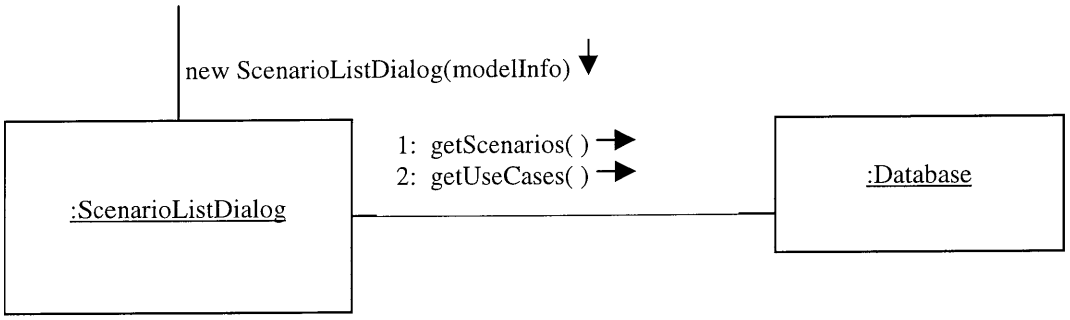


Figure 6-46 scenarioListDialog( ) Collaboration Diagram

6.10.2 Implementing the Link between the actors in a Use-Case Model and the actors in a Scenario

In Figure 6-9, which shows Model Class Diagram, the *updates* association relationship between the *Model* class and *Database* class allows the *Scenario* class, which is inherited from the *Model* class, to get the Use-Case Models from the database.

In Figure 6-18, The Scenario Class Diagram, the *getsUseCases* association relationship between the *Scenario* class and the *UseCaseModel* class allows the Scenario to search for the Use-Case Model, which contains the use-case corresponding to the scenario. The association relationship, *getsActors*, allows the *Scenario* class to obtain the actors included in the Use-Case Model containing the use-case corresponding to the Scenario.

In Figure 6-47, The ScenarioActor Class Diagram, the *creates* association relationships provide the *ScenarioCanvas* class with the ability to instantiate objects of the classes, *ModelInfo* and *ScenarioActorDialog*. The association relationship, *reads*, allows the *ScenarioActorDialog* class to get the references stored in the database. Once the *ScenarioActorDialog* has the references, it starts searching for the references containing the use-case corresponding to the scenario. The association relationship *getsUseCase*, obtains the use-case from the reference. The *ScenarioActorDialog* finds the actors, which are connected to the use-case corresponding to the scenario via a reference. The association relationship, *getsActor*, obtains an actor from a reference. In addition, the association relationship, *getsNames*, allows the *ScenarioActorDialog* to get the name of the use-cases and actors. The association relationship, *updates*, allows

the *ScenarioActorDialog* class to update the *ModelInfo* with the selected actor that should be added to the scenario.

The Use-Case Model Objects Class Diagram in Figure 6-14 shows that the *Use-Case* and *Actor* classes have the association relationships, *referenced-By*, with the class, *Reference*. The association relationships, *referenced-By*, allow the *Reference* class to know and be able to access the actor and use-case, which are linked via the reference.

Figure 6-48, The addActor Scenario, shows the interaction between the developer and the LCCASE tool to add an actor to a scenario.

The postconditions field in the Scenario Contract, shown in Figure 6-49, explains the calls to other methods occur during the creation of a scenario. These calls include:

- new Actor() to instantite a new actor which is the system.
- getUseCaseModels() which obtains the UseCaseModels from the database.
- getName() to get the name of a use-case.
- getActors() to obtain actors from a specified Use-CaseModel.
- new ScenarioCanvas() to instantiate an object of ScenarioCanvas class.

The scenarioActor( ) system operation occurs as a result of the developer request to add an actor to the scenario. Figure 6-50 shows the scenarioActor( ) contract. The *scenarioActor( )* is responsible for adding an actor to the scenario. It provides the developer with a list of actors that are connected to the use-case corresponding to the scenario. When the developer chooses an actor from the suggested list, it adds it to the scenario. However, when the developer requests adding an actor that is not conncted to

the use-case, it calls *addRefernce( )* method to add a reference to the Use-Case Model and then it adds the actor to the Scenario. If the developer requests adding an actor that is not included in the Use-Case Model, it adds first the actor and reference to the Use-Case Model and then it adds the actor to the scenario. The postconditions field illustrates the methods that should be called to perform the above responsibilities. Consistency checks have been made between the contract and the class diagrams to make sure that the information about the classes and their visibility satisfies the information about the methods. The Scenario Actor Collaboration Diagram in Figure 6-52 shows the sequence of the messages discussed above.

Figure 6-51 shows the scenarioActorDialog Contract. The *scenarioActorDialog( )* method is called by the *scenarioActor( )* method in the *ScenaarioCanvas* class. It is responsible for determining the actor that should be added to the scenario. It displays a list showing the actors that are in the Use-Case Model containing the use-case corresponds to the scenario and then asks the developer to select an actor from the list or enters a new actor name. Once the developer selects an actor from the list or enters an actor name, it updates the *ModelInfo* and return to the *scenarioActor( )*. Figure 6-52 shows the scenarioActor Collaboration Diagram.

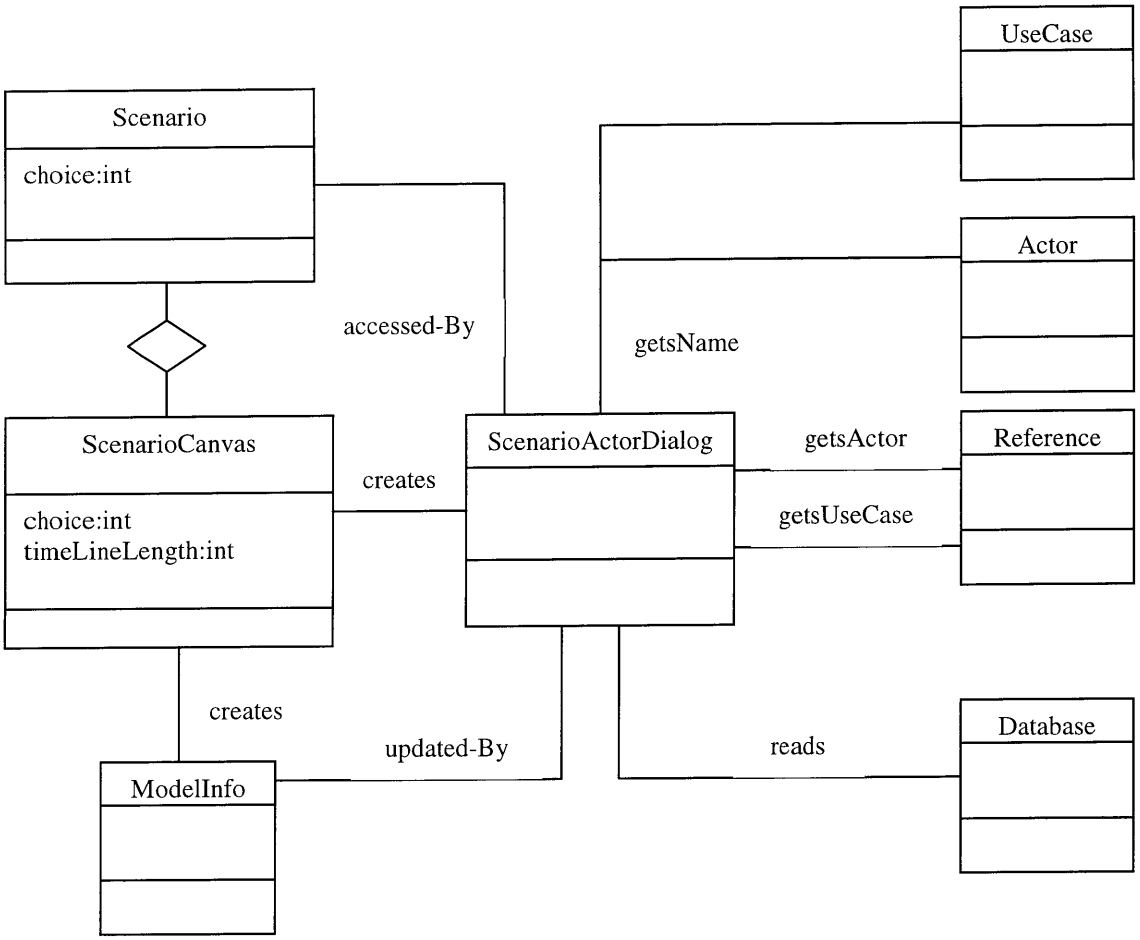


Figure 6-47 ScenarioActor Class Diagram

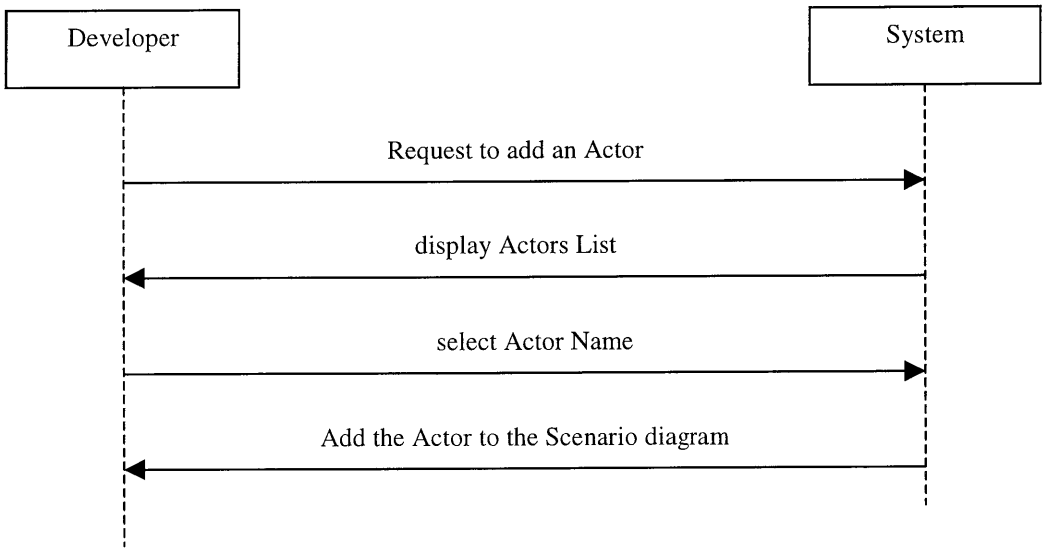


Figure 6-48 Add Actor to Scenario

Name:	scenario(name:String, database:Database, directory:Directory, allModels:AllModels)
Responsibilities:	Set the Scenario Window. Find the actors that are included in the Use-Case Model containing the use-case that corresponds to the scenario.
Class:	Scenario
Cross Reference:	
Notes	
Output:	Display scenario's window
Preconditions:	If arg = scenario
Postconditions:	new Actor() Database.getUseCaseModels() UseCaseModel.getUseCases() UseCase.getName() UseCaseModel.getActors() new ScenarioCanvas()

Figure 6-49 scenario( ) Contract

Name:	scenarioActor (x:int, y:int)
Responsibilities:	Add an Actor to the Scenario Add refernce to a Use-Case Model Add actor to a Use-Case Model
Class:	ScenarioCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Actor AND mouseClicked
Postconditions:	new ModelInfo( ) new ScenarioActorDialog( ) ModelInfo.getName( ) if no reference between actor and use-case UseCaseModel.addReference( ) else if actor not in UseCaseModel UseCaseModel.addActor( ) UseCaseModel.addReference( ) new ScenarioActorPoint paint( )

Figure 6-50 scenarioActor( ) Contract

Name:	ScenarioActorDialog (parent:Scenario, database:Database, modelInfo:ModelInfo
Responsibilities:	Determine the Actor that should be added to the Scenario
Class:	ScenarioActorDialog
Cross Reference:	
Notes	
Output:	
Preconditions:	
Postconditions:	Database.getReferences( ) Scenario.getName( ) Reference.getUseCase( ) UseCase.getName( ) Reference.getActor( ) Actor.getName( )

Figure 6-51 scenarioActorDialog( ) Contract

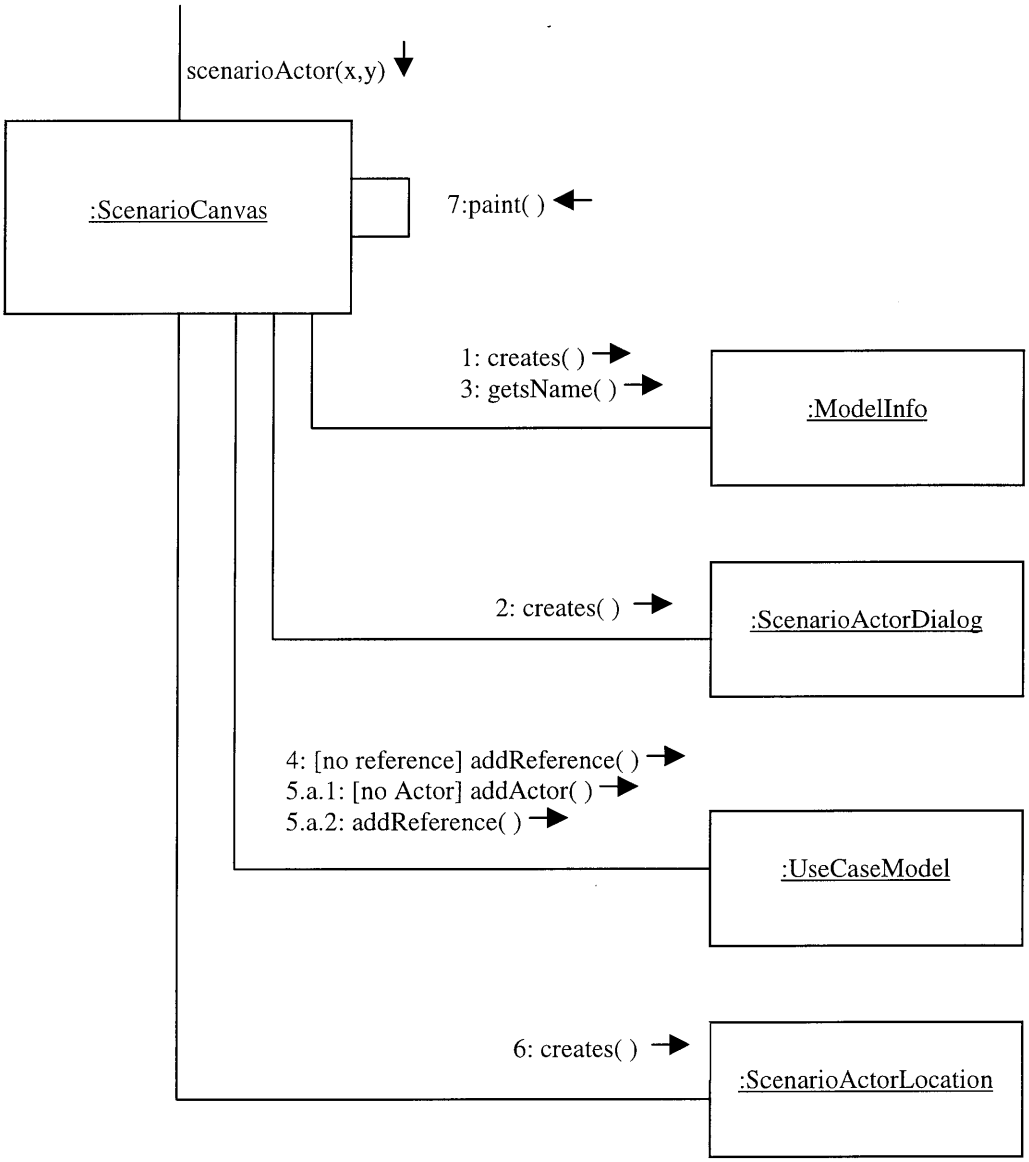


Figure 6-52 scenarioActor( ) Collaboration Diagram

6.10.3 Implementing the Link between the Parameters of a System Operation and the Classes

The association relationship, *getsParameters*, allows the *ScenarioCanvas* to obtain the parameters of a system operation. Once the *ScenarioCanvas* obtains the parameters, it checks their types across the classes created in the project to determine their consistency. The association relationship, *getsType*, allows the *ScenarioCanvas* to find the type of a parameter. The association relationship, *updates*, with the database allows the *ScenarioCanvas* to determine the classes created in the project.

The mouseClicked Contract shown in Figure 6-53 illustrates how the *ScenarioCanvas* performs consistency checks for the parameters of a system operation. Once the developer enters the parameters through the *MethodDialog*, the *ScenarioCanvas* calls:

- *getParameter( )* method in the system operation to obtain its parameters.
- *getType( )* method in the *Parameter* class to determine the type of a parameter in a system operation.
- *getClasses( )* in *Database* class to find the classes created in the project.

Name:	MouseClicked (mouseEvent:MouseEvent)
Responsibilities:	Respond to the mouse clicks in the Scenario window by calling the appropriate methods to add actors and system operations to the scenario or displaying the method dialog.
Class:	ScenarioCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	Mouse event fired
Postconditions:	If click on Actor ScenarioActor(x,y) Else if click on System Operation AND selection = = 3 DeleteSystemOperation() If clicksCount >= 2 AND click on System Operation New MethodDialog() SystemOperation.getParameters() Parameter.getType() Database.getClasses()

Figure 6-53 ScenarioCanvas.mouseClicked() Contract

6.11 Summary

This chapter described the implementation phase of the LCCASE tool. The implementation phase has been split into distinct phases, analysis, design, and implementation. Each phase focused on the development of the models that support its purpose. By utilising the Next Generation of the Fusion Method to develop the life cycle phases and models and producing several prototypes through the life cycle iterations, the LCCASE tool has been produced and has met its objectives.

Developing a large and complex system such as the LCCASE tool was a challenging task. Since the LCCASE tool has many goals and large scope, many subsystems, use-cases, scenarios, classes, relationships, and collaboration diagrams, a huge amount of code was needed to be developed during the implementation phase. The analysis and design phases for the complete LCCASE tool system has been produced. The code has produced for supporting the development of the models: Use-Case Model, Class Diagram, Scenario, and Contracts, the subsystems: Directory, Development Life Cycle, and Data Dictionary, and many features of the tool such as providing guidance, performing consistency checks, and linking the information between the models. Due to the time constraint and the development of such large system solely by the author, no code was written for the models of the design phase and code generation.

Having completed the implementation phase, which resulted in successful object-oriented software, I highly appreciate the OO techniques, method, and programming language. They reduce the complexity of my work. The use of the

inheritance, abstraction, polymorphism, and dynamic binding reduces the size of code and potential work. Other features such as encapsulation and loosely coupled classes and methods supported solving the problems raised during the code phase.

7 Conclusion

This thesis has presented the philosophy, design, and development work I have carried in producing the LCCASE tool for supporting the development and learning processes of object-oriented software engineering. These are discussed below and the contributions of the work are highlighted.

7.1 The LCCASE Tool

The development of large and complex software demands tremendous amounts of time, effort, and money. Such developments have suffered from many difficulties. These difficulties include failure to meet the user requirements, being developed and delivered behind schedule, and being erroneous. The importance of a methodology combined with a CASE tool to support the development of large and complex software is noticed by the rationales in software engineering field. However, current CASE tools suffer from many deficiencies, which make them inadequate for developing software with complex nature. For these reasons current CASE tools do not necessarily improve the productivity of the developer. Moreover, most of such CASE tools are designed to support the needs of the experienced developer and lacking proper and effective support for the special needs of the learners.

The LCCASE tool presented in this thesis incorporates novel aspects for supporting the development of large and complex software systems. These aspects include applying the rules and maintaining the semantics of the *Next Generation of the Fusion method.* This method has many powerful and effective features. It implements important and useful object-oriented principles and techniques. In addition, it implies a certain methodological

process, which are step-by-step activities that include many consistency checks to ensure that the information does not conflict. When these activities have been performed, a desired result is achieved and produced. This outcome is efficient software that satisfies the requirements of the user. Learners of software engineering certainly need a clear pattern to follow. This is especially true for those who lack the experience in software development that qualifies them to decide that their development work is complete and consistent. On the other hand, the experienced developer can benefit from the rich and effective features of the method. The *Next Generation of the Fusion method* supports the objectives of the LCCASE tool, which made it the most suitable candidate for automation.

Unlike other CASE tools, the LCCASE tool has the ability to link the information between the models developed during the different phases of the software life cycle. The links, discussed in chapter 4, start as early as the use-cases, which represent the needs of the users and continue until the code. Linking the use-cases to most models within the software life cycle maintains clear and precise view of the system requirements through out the software development process. This has great impact on achieving a high degree of software efficiency and confirming that the software goals have been satisfied.

The LCCASE tool offers innovative opportunities to overcome the difficulties that arise during the software development process. It provides the developer with intelligent guidance throughout the software life cycle. This guidance steers the attention of the developer to the important issues, misconceptions, or conflicts within the software design, which increases the software reliability.

The LCCASE tool has closed the gap of previous tools in assisting the learners. It provides new powerful means of supporting the software development in a learner centred approach. Such means aim at adapting the needs of the learner, improving the learning of object-oriented software engineering, and helping the learners in developing software. In addition, it supports the needs of the experienced developer to construct large, complex, and successful software.

Moreover, the LCCASE tool strikes an excellent balance between a user-friendly CASE tool and a highly comprehensive and integrated tool. These kinds of support are usually missing in previous CASE tools.

Furthermore, the LCCASE tool has been used to develop itself, which is a piece of reliable and maintainable software. It was easy to use and provided effective support during the development of Use-Case Models, Class Diagrams, Scenarios, and Contracts, shown and discussed in chapter 6. The use of the LCCASE tool was satisfactory and enjoyable. The development of diagrams using the LCCASE tool was far easier and powerful than other CASE tools.

7.1.1 The Characteristics of the LCCASE Tool

The main objective of the LCCASE tool is to provide the developer with a fully integrated CASE tool that assists him in the development of large software systems. It offsets the drawbacks of the development and learning processes of software engineering. In addition, the LCCASE tool supports the whole development life cycle of a system by automating the *Next Generation of the Fusion Method.*

The main characteristics of the LCCASE tool include providing intelligent guidance, navigational assistance, graphical assistance, code generation facility, Data Dictionary, and support for the maintenance phase.

The intelligent guidance feature of the LCCASE tool provides support to the software developer. The LCCASE tool realises the connection between the models and provides this information to the developer as required. This decreases the complexity of the software development process, draws the attention of the developer to the important concepts, allows him to link between different concepts during the software development process, and helps him to interpret this relevant information correctly.

In addition, the LCCASE tool has the ability to make many consistency checks between the information in the models and proposes the models that should be created by the developer based on the information in previously developed models.

Furthermore, the LCCASE tool maintains multiple views of the models, advises the developer to reuse resources and has the ability of automatic reuse of resources. In addition, the LCCASE tool provides a facility to track the requirements specified in the Use-Case Model of the *Next Generation of the Fusion method* and maps these requirements in other models, which supports producing end-users software.

Moreover, the LCCASE tool has the ability to analyse the models and provide the developer with a feedback. The LCCASE tool encourages but does not force the developer to maintain all the requirements of the method it support. While the LCCASE tool guides the developer through the software life cycle, the developer controls the development process.

The ability of the LCCASE tool to associate links between the models facilitates easy and smooth navigation between the models. The Directory subsystem of the LCCASE tool allows the developer to navigate through the models and the use-cases and the classes hierarchies. In addition, the LCCASE tool main menu with its dynamic structure enables the developer to easily access the models. The navigational capability provides the developer with cohesive views of the design and means to trace the relations and the dependency between the components of the project, which assist the developer in deciding the new components that should be developed and the necessary changes to the existing ones.

The LCCASE tool simplifies the task of drawing and creating diagrams of the different models. It assists the developer in creating flawless diagrams. These diagrams are expressive to convey all the information within the method, but simple enough to be easily drawn and understood. The provision of a tool that allows the developer to draw the diagrams of the method very easily, assists him to concentrate on the actual development process.

The design of the LCCASE tool has the ability to support code generation in the future. When the developer creates a *Design Class Diagram*, Java source code can be generated. The code generation feature helps the developer to produce several rapid prototypes during the software life cycle and implement the iterative life cycle style. In addition, it supports the learner.

The Data Dictionary of the LCCASE tool provides the developer with the ability to document important information about the models in a project effortlessly. The LCCASE tool creates, maintains, and updates a Data Dictionary automatically. The Data Dictionary

helps the developer to better understand concepts and ideas during the development process and supports maintaining and extending the software.

Through the Data Dictionary, the LCCASE tool supports the maintenance phase of the software life cycle. In addition, the Use-Case Models, which represent the needs of the users, are linked to most other models, such as, the Scenarios, Contracts, and Collaboration Diagrams. When a change occurs to the needs of the user that affect the Use-Case Model, the LCCASE tool informs the developer with the alterations in other connected models.

7.1.2 The Design of the LCCASE Tool

The design of the LCCASE tool has been carefully executed to achieve the above mentioned objectives and characteristics. In addition, more special consideration is given to extensibility and portability. During the design, the following aims have been adopted and satisfied:

- Designing the architecture of the system. This involves dividing the system into co-operating subsystems, thus reducing dependencies between subsystems and facilitating easier future extensibility.
- Designing the interface between the subsystems by determining how they communicate together.
- Identifying the components of each subsystem, their services, and their interaction.
- Enforcing object-oriented technology to reduce the complexities involved in the design.

- Maintaining a central repository, which is implemented by Database Subsystem. The Database Subsystem stores all the data of the developed project and controls accessing the shared data by all subsystems.
- Choosing the JAVA programming language for its object-oriented structure, relative ease with graphics, mechanisms for an event driven system, and modular characteristics to support the portability of the system.

Extra care was taken in designing the LCCASE tool to pay attention to implementing the links between the different models, generating intelligent guidance, and flexibility of the system that achieves the objectives of learner centred approach.

The user interface strategies of the LCCASE tool were selected to:

- provide the developer with the freedom to navigate through the models and their components using different techniques such as the Directory subsystem and the main menu with its dynamic structure;
- allow the developer to view the relations between the components of the developed project;
- enable the incorporation of different views of the models;
- carefully place and choose the different information messages and dialogs;
- provide the developer with full control over the development of the models. Thus allow him to select the facilities of the tool according to his individual needs;
- provide the developer with fast and flexible interactions and useful feedback;
- maintain the consistency between the commands, menus, graphical representations, and the format of the models. This reduces time necessary for the developer to become familiar with the tool.

7.2 The Impact of the LCCASE Tool on the Design of New CASE Tools

The philosophy and design of the LCCASE tool can provide fundamental structure for constructing other CASE and design tools. This structure is systematic and facilitates extensibility through the integrated set of techniques and facilities presented in chapters 3 and 4, which play an important role in developing CASE tools for supporting successful projects. These techniques and facilities are based on object-oriented principles and provide a unified perspective to develop comprehensive CASE tools.

The ideas of the LCCASE tool can be used to influence the production of other CASE tools, which automate different method for software development. Especially, that the LCCASE tool implements the UML notation which facilitates reusing the diagrams and their components.

7.3 Conclusions Reached from Developing the LCCASE Tool

There are important conclusions, reached from the achievements and difficulties during the implementation of the LCCASE tool. These include:

- The object-oriented software engineering approach has been used successfully throughout the LCCASE tool development life cycle. Its powerful techniques and principles tremendously reduce the complexity of developing large software and render it a satisfying and enjoyable task.
- Choosing an appropriate object-oriented method for the problem domain of a project is essential to guide the development process. Moreover, this chosen method should be adapted to better suit the needs of the project.

- The analysis and design phases are essential for developing quality and successful software. Adequate time and effort should be allocated to these tasks before rushing to code.
- Producing a scenario for each use case and a contract for each system operation, provide an understandable and clear analysis documents and strong basis for testing, debugging, and maintaining the system.
- The iterative life cycle style is a necessity for developing successful software that meets all the requirements of its users.
- Well-designed object-oriented software was found to allow for smooth and straightforward alteration during the maintenance phase.

7.4 Future Work

One area that demands additional work is the development of the remaining models. Once this has been accomplished, I intend to evaluate the effectiveness of the LCCASE tool on real projects. I look forward to see the LCCASE tool used as a powerful CASE tool in developing and learning object-oriented software engineering.

The rich and challenging area of CASE tools has much possible future work. There is an opportunity for further work in the field of collaborative development. The LCCASE tool is designed to support teamwork, which will support extending it to maintain and co-ordinate the co-operative efforts between the developers to make the development process more effective. The LCCASE tool can play a significant role in accomplishing the collaborative development. This can be facilitated by using the interactive capabilities and the central Database subsystem of the LCCASE, which will ensure the consistency of the information in the project.

The Internet provides a very flexible way to distribute the working material and enables the developers to access them from different locations at any time. The Internet services combined with the LCCASE tool characteristics can be used to extend the LCCASE tool to support collaborative work of developers in different geographical locations.

The collaborative work should increase interactions among developers, and between developers and their group manager. This allows for more immediate feedback, from group manager to developers, regarding their development work and progress. In addition, group managers can monitor, enhance, and adapt the produced work. Furthermore, this collaborative work should have some intelligent guidance that suggests which group work could be combined and continued with others.

Further work is also possible to extend the intelligent guidance feature. The LCCASE tool currently can suggest the models that should be developed according to the information in the existing models. This can include the areas of development rate, style, and cognitive skills of the individual developers. The LCCASE tool can be extended to use this information to be able to formulate some inferences and hypotheses from the developed models to suggest the development of new classes or criticise the structure of the developed classes.

In future studies, I plan to continually explore the use of alternative mechanisms to support the development and learning processes of object-oriented software engineering.

References

[Abran 1996] Abran A., Teaching Software Engineering Using ISO Standards. *StandardView,* (September 1996), 4 (3) 139-145.

[Abu Al-Maati 1997] Abu Al-Maati S., Shaoff W., Integrating Literate Programming and Cleanroom Software Engineering to Teach Software Engineering Skills. *ACSE '97,* ACM, Melbourne, Australia (1997), 99-106.

[Anderson 1991] Anderson J., Sheffler J., Ward E., Manageable Object-Oriented Development: Abstraction, Decomposition, and Modeling. *Proceedings: Tri-Ada '91,* ACM, (1991), 119-212.

[Barclay 1997] Barclay K., Savage J., *Object-Oriented Design with C++,* Prentice Hall Europe, (1997).

[Bezivin 1992] Bezivin J., Roux O., Royer J., Aducators' Symposium- Teaching Object-Oriented Programming or Using the Object Model to Teach Software Engineering. *Addendum to the Proceedings of OOPSLA '92,* ACM, Vancouver, British Columbia, Canada (October 1992), 269-275.

[Biffl 1997] Biffl S., Grechenig, Teaching Software Engineering and Project Management to 300 Participants without Drain of Quality or Intensity. *ITiCSE '97,* ACM, Uppsala, Sweden (1997), 51-53.

[Blanc 1992] Blanc L., Korn W., A Structured Approach to the Evaluation and Selection of CASE Tools. *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing (vol. II): Technological Challenges of the 1990's,* ACM, Kansas City, MO, USA (March 1992), 1064-1069.

[Booch 1983] Booch G., *Software Engineering with Ada,* The Benjamin/Cummings Publishing Company. Inc., (1983).

[Booch 1994] Booch G., *Object-Oriented Analysis and Design with Applications,* The Benjamin/Cummings Publishing Company. Inc., (1994), 2nd.

[Booch 1998] Booch G., Rumbaugh J., Jacobson I., *The Unifed Modeling Language User Guide,* Addison-Wesley, (1998).

[Boyle 1994] Boyle T., Thomas G., Build Your Own Virtual Computer: A Computer Simulation using an Active Learning Approach. *Proceedings of the East-West Conference on Multimedia, Hypermedia and Virtual Reality, MHVR '94,* Moscow (September 1994).

[Broek 1992] Broek E., Verkoulen P., A Tool for Integrated Modelling of Static and Dynamic Aspects of Systems. *Next Generation CASE Tools,* IOS Press, (1992), 75-98.

[Carroll 1990] Carroll J., *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill,* MIT Press, Cambridge, Mass (1990).

[Coleman 1994] Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F., Jeremaes P., *Object-Oriented Development The Fusion Method,* Prentice-Hall International, Inc., (1994).

[Coleman 1997a] Coleman D., Cotton T., Dollin C., Jeremaes P., Johnson M., Malan R., Redmond-Pyle D., Fusion Update: Requirements and Analysis in the Next Generation of Fusion. *Fusion Newsletter,* Hewlett-Packard Laboratories, (May 1997), 5 (2) 1-5.

[Coleman 1997b] Coleman D., Cotton T., Dollin C., Jeremaes P., Johnson M., Malan R., Redmond-Pyle D., Yang H., Fusion Update Part II: Architecture and Design in the Next Generation of Fusion. *Fusion Newsletter,* Hewlett-Packard Laboratories, (September 1997), 5 (3) 1-8.

[Concepcion 1998] Concepcion A., Using an Object-Oriented Software Life-Cycle Model in the Software Engineering Course. *Proceedings of the twenty-ninth SIGCSE '98 Technical Symposium on Computer Science Education,* ACM, Atlanta, GA, USA (February 1998), 30-34.

[Cotton 1996] Cotton T., Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion. *Object-Oriented Development at Work: Fusion in the Real World,* Prentice-Hall, (1996), 179-202.

[DeMarco 1979] DeMarco T., *Structured Analysis and System Specification,* Yourdon Press, New York, NY (1979).

[Dodani 1992] Dodani M., Educators' Symposium- Teaching Practical Object-Oriented Software Engineering. *OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications,* ACM, Vancouver, Canada (October 1992), 251-256.

[Gay 1986] Gay G., Interaction of Learner Control and Prior Understanding in Computer-Assisted Video Instruction. *Journal of Educational Psychology,* (June 1986), 78 (3) 225-227.

[Gibson 1988] Gibson M., Implementing the Promise. *Datamation,* (1988), 35 (3) 65-67.

[Grimsdale 1997] Grimsdale R., Design of Large Software Systems. *Lecture notes,* (1997).

[Grimsdale 1998] Grimsdale R., Design of Large Software Systems. *Lecture notes,* (1998).

[Harel 1987] Harel D. , Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* (1987), 8, 231-274.

[Ho 1992] Ho Y., To What Extent will CASE Tools Assist Users in the Systems Development?- A Case Study in Academic Environment. *Proceedings of the 1992 ACM SIGCPR conference on Computer personnel research,* ACM, Cincinnati, OH, USA (April 1992), 93-96.

[Huang 1996] Huang R., Jarzabek s., PCS - A CASE Tool for Distributed Group Software Development. *IFIP 96 Advanced IT Tools,* (1996), (45) 402-410.

[Iivari 1996] Iivari J., Why are CASE Tools Not Used?. *Communications of the ACM,* ACM, (October 1996), 39 (10) 94-103.

[Jacobson 1992] Jacobson I. and others. *Object-Oriented: A Use-Case Driven Approach,* Addison- Wesley, Workingham (1992).

[Jarabek 1998] Jarabek S., Huang R., The Case for User-Centered CASE Tools. *Communications of the ACM,* ACM, (August 1998), 41 (8) 93-99.

[Jurik 1992] Jurik J., Schemenaur R., Experiences in Object Oriented Development. *Conference proceedings on TRI-Ada '92,* ACM, Orlando, FL, USA (November 1992), 189-197.

[Knudsen 1996] Knudsen J., Madsen O., Using Object-Orientation as a Common Basis for System Development Education. *ACM SIGPLAN Notices,* OOPSLA '96, (December 1996), 31 (12) 52-62.

[Larman 1998] Larman C., *Applying UML and Patterns An Introduction to Object-Oriented Analysis and Design,* Prentice-Hall International, Inc., (1998).

[Lyytinen 1992] Lyytinen K., Tahvanainen V., Introduction: Towards the Next Generation of Computer Aided Software Engineering (CASE). *Next Generation CASE Tools,* IOS Press, (1992), 1-7.

[Maibaum 1997] Maibaum T., What we Teach Software Engineers in the University: Do we Take Engineering Seriously? One Academic's View of Software Engineering Education. *Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT Symposium on Software Engineering,* ACM, Zurich, Switzerland (Sebtember 1997), 40-50.

[Malan 1995] Malan R., Coleman D., Letsinger R., Lessons from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard. *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications OOPSLA '95,* ACM, Austin, TX , USA (October 1995), 33-46.

[Norman 1989] Norman R., Nunamaker J., CASE Productivity Perceptions of Software Engineering Professionals. *Communications of the ACM,* ACM, (September 1989), 32 (9) 1102-1108.

[Pallatto 1989] Pallatto J., CASE Speeds Unum's Development Effort. *PC Week,* (6th February 1989), 39 & 47.

[Parsons 1994] Parsons D., *Object-Oriened Programming with C++,* DP Publications Ltd, London (1994), 1st.

[Patrick 1983] Patrick J., Evans P., Advance Organizers and Learner Control of Sequence in Recall of Topic Attributes. *Human Learning,* John Wiley & Sons, Ltd., (1983), 2, 269-277.

[Ram 1996] Ram B., Girdhar R., Simulation-Based Learning Tool for Facilities Design. *Proceedings of the conference on Winter simulation,* San Diego, CA, USA (December 1996), 1413-1420.

[Ramiller 1993] Ramiller N., Computer-Aided Software Engineering: Perceptions of Technology, Work, and Management among I.S. Personnel. *Proceedings of the 1993 Conference on Computer Personnel Research SIGCPR '93,* ACM, St Louis, MO, USA (April 1993), 436-449.

[Ross 1975] Ross D., Goodenough J., Irvine C., Software Engineering: Process, Principles, and Goals. *Computer,* IEEE Computer Society, (May 1975), 17-27.

[Rumbaugh 1991] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modelling and Design,* Prentice-Hall International Editions, (1991).

[Rumbaugh 1998] Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language Reference Manual,* Addison-Wesley, (1998).

[Steinberg 1985] Steinberg E., Computer Presented Organizational Memory Aids as Instruction for Solving Picofomi Problems. *Computer Based Instruction,* (Spring 1985), 12 (2) 44-49.

[Stobart 1996] Stobart S., Introduction to CASE. *CASE Home Page,* osiris.sund.ac.uk/sst-scripts/simon.html, (1996).

[Taylor 1987] Taylor R., Selecting Effective Courseware: Three Fundamental Instructional Factors. *Contemporary Educational Psychology,* (1987), 12, 231-243.

[Teichroew 1971] Teichroew D., Sayani H., Automation of Systems Building. *Datamation,* (August 1971), 15, 25-30.

[Teichroew 1977] Teichroew D., Hershey E., PSL/PSA: A Computer-Aided Tichnique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering,* IEEE, (January 1977), SE-3 (1) 41-48.

[Tewari 1995] Tewari R., Software Reuse and Object-Oriented Software Engineering in the Undergraduate Curriculum. *Papers of the 26th SISCSE Technical Symposium on Computer Science Education,* ACM, Nashville, TN, USA (March 1995), 253-257.

[Vessey 1992] Vessey I., Jarvenpaa S., Tractinsky N., Evaluation of Vendor Products: CASE Tools as Methodology Companions. *Communications of the ACM,* ACM, (April 1992), 33 (4) 90-102.

[Welsch 1997] Welsh C., Schalk A., Kramer S., Integrating Forward and Reverse Object-Oriented Software Engineering. *ICSE '97,* ACM, Boston, MA, USA (1997), 560-561.

[Williams 1996] Williams J., Managing Iteration in OO Projects. *Computer,* IEEE, (September 1996), 29 (9) 39-43.

[Yourdon 1986] Yourdon E., What Ever Happened to Structured Analysis?. *Datamation, (June 1986), 133-138.*

Appendices

Appendix A: Scenarios

Appendix A shows the scenarios developed during the analysis phase of the LCCASE tool development life cycle.

A.1 Model Scenarios Group

The Model Scenarios group corresponds to the Model Use-Case Model. This group shows all the operations that can be involved in creating or browsing a model. The following sub-sections show the scenarios in this group.

A.1.a Create New Model Scenario

Create New Model Scenario shows the interaction between the Developer and the System during the creation of a new model. The interaction corresponds to the system operations, which are represented with an arrow line from the sender of the system operation to the receiver. i.e. The Developer selects a Model Type then the System displays the Models list. Figure A-1 shows the Create New Model Scenario.

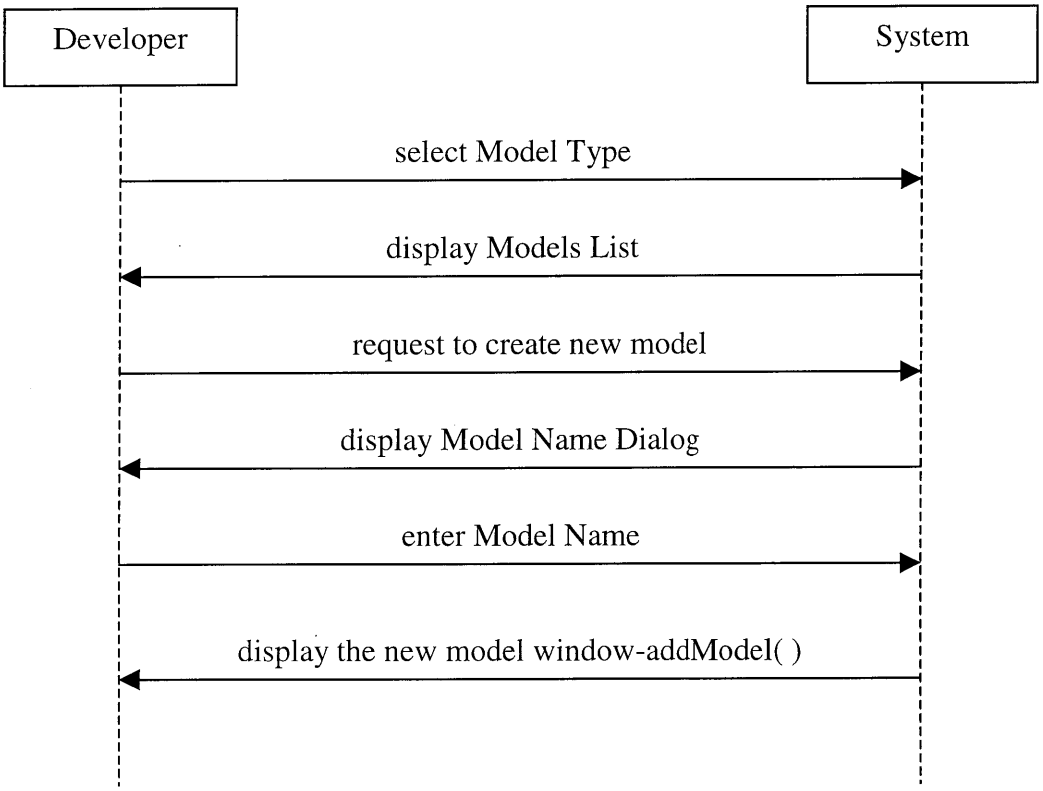


Figure A-1 Create New Model Scenario

A.1.b Browse Model Scenario

Figure A-2 shows the interaction between the Developer and the System to browse a model.

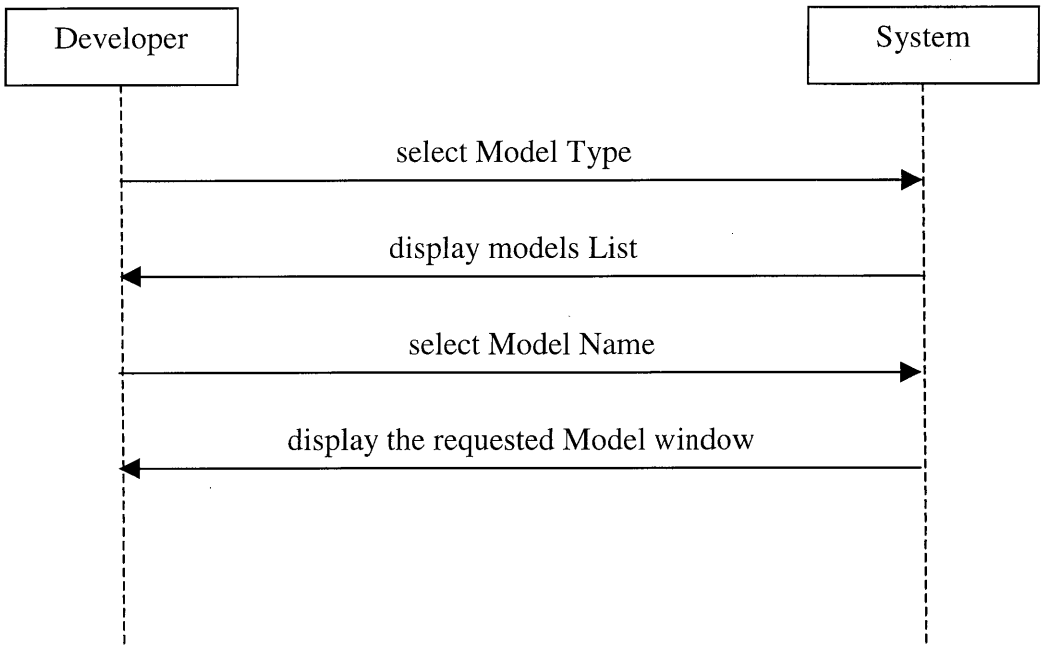


Figure A-2 Browse Model Scenario

A.1.c Create Use-Case Scenario

Figure A-3 shows the Create Use-Case Scenario. This scenario shows the operations that should be done in order to create a use-case.

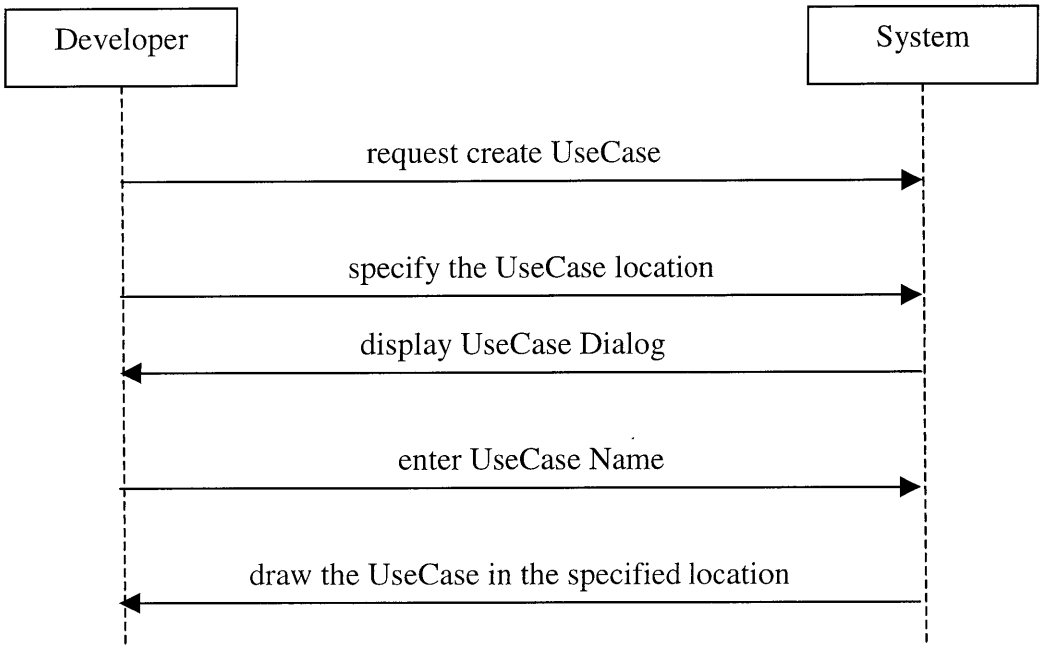


Figure A-3 Create Use-Case Scenario

A.1.d Create Actor Scenario

Figure A-4 shows the Create Actor Scenario.

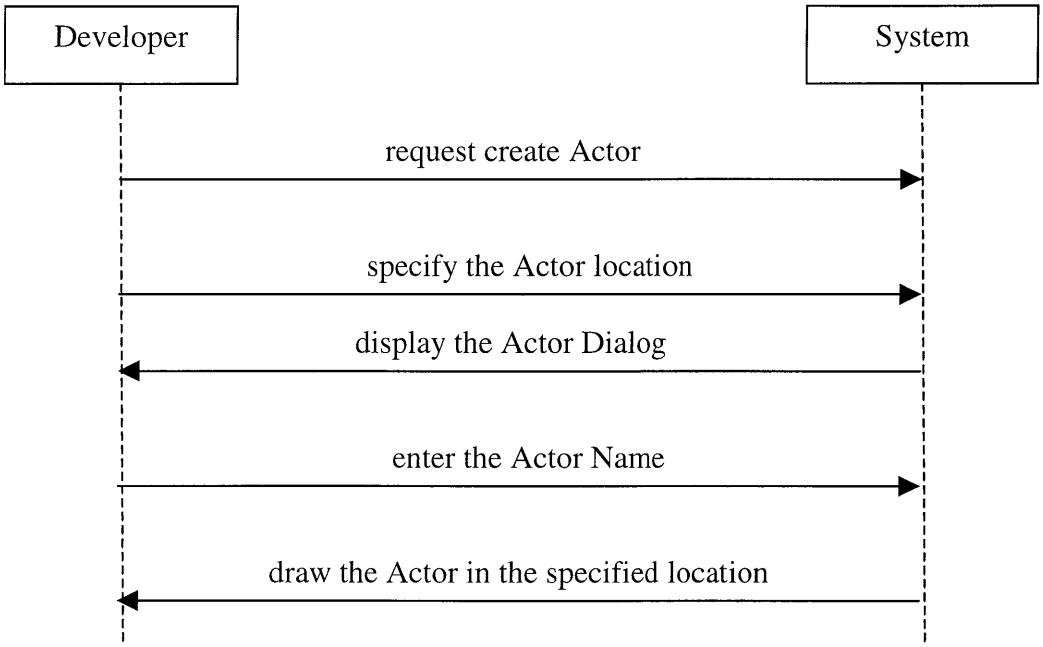


Figure A-4 CreateActor Scenario

A.1.e Create Reference Scenario

Figure A-5 shows the Create Reference Scenario.

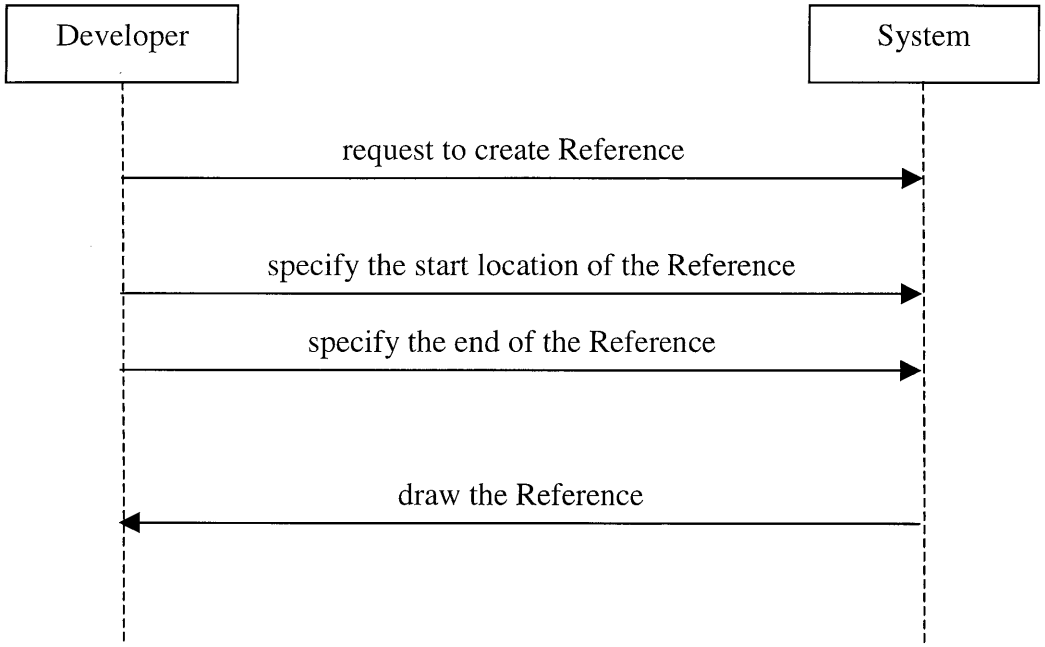


Figure A-5 CreateReference Scenario

A.1.f Delete Use-Case/Actor Scenario

Figure A-6 shows the Delete Use-Case Scenario.

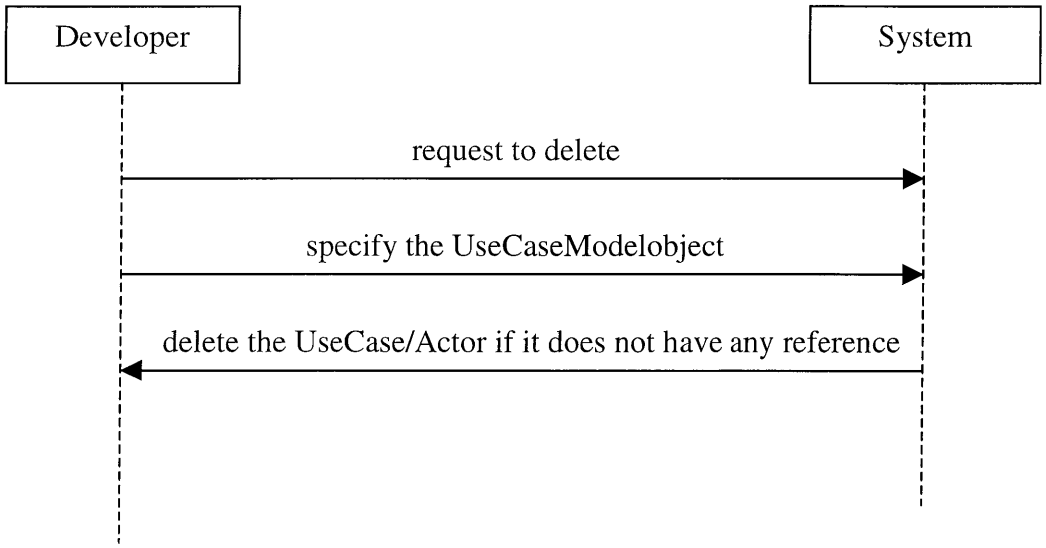


Figure A-6 Delete Use-Case Scenario

A.1.g Delete Reference Scenario

Figure A-7 shows the Delete Reference Scenario.

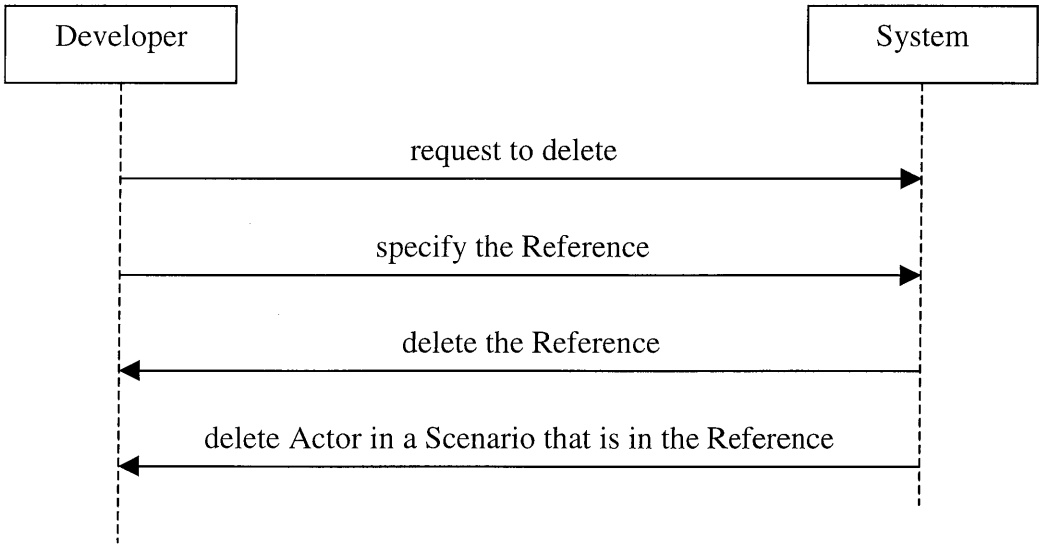


Figure A-7 Delete Reference Scenario

A.1.h Move Use-Case/Actor Scenario

Figure A-8 shows the Move Use-Case/Actor Scenario.

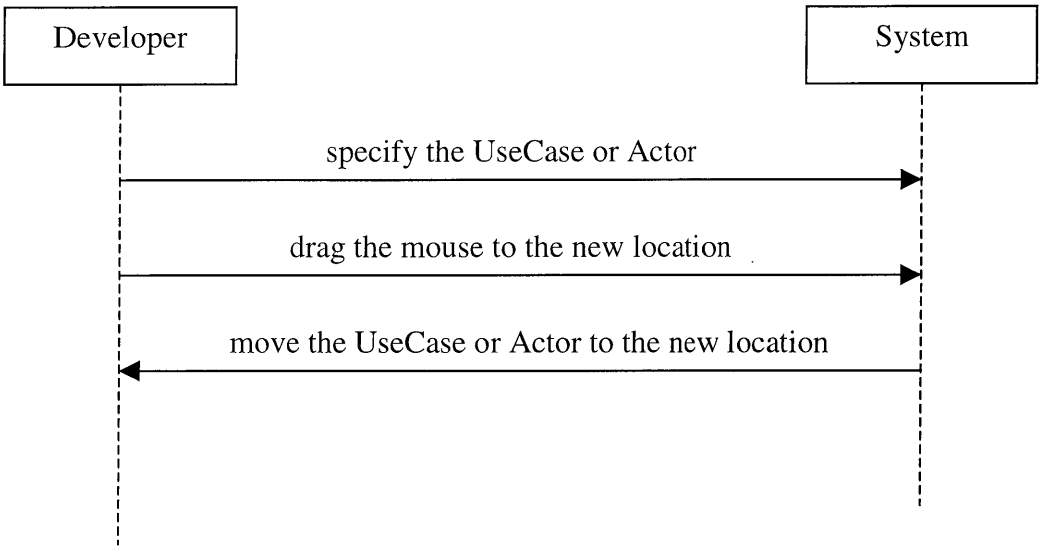


Figure A-8 Move Use-Case/Actor Scenario

A.1.i Create Class Scenario

Figure A-9 shows the Create Class Scenario.

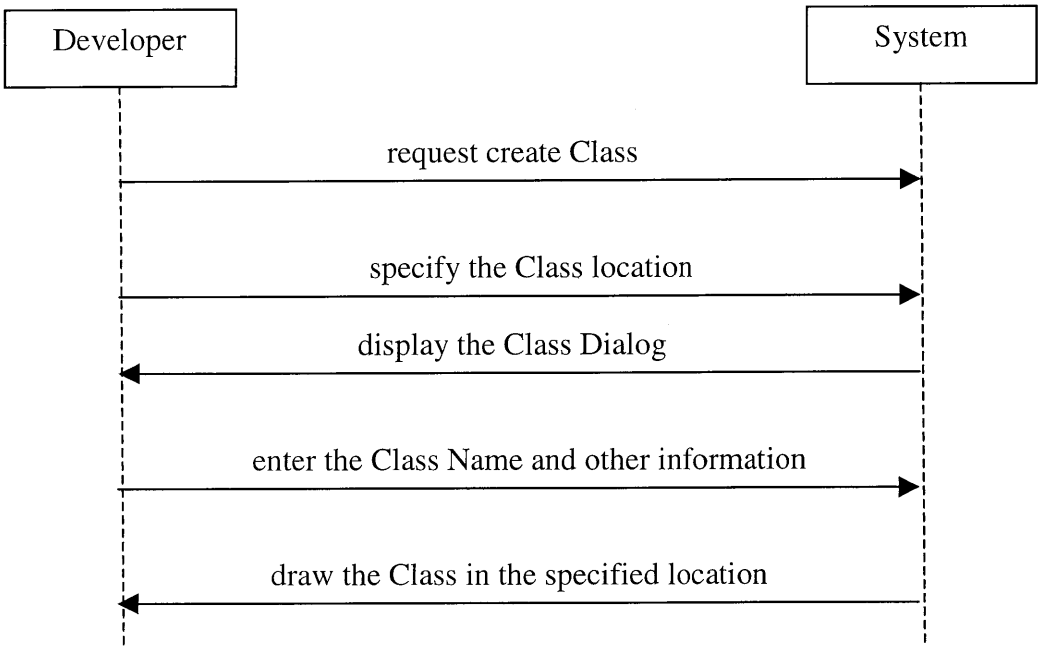


Figure A-9 Create Class Scenario

A.1.j Create Association Relationship Scenario

Figure A-10 shows the Create Association Relationship Scenario.

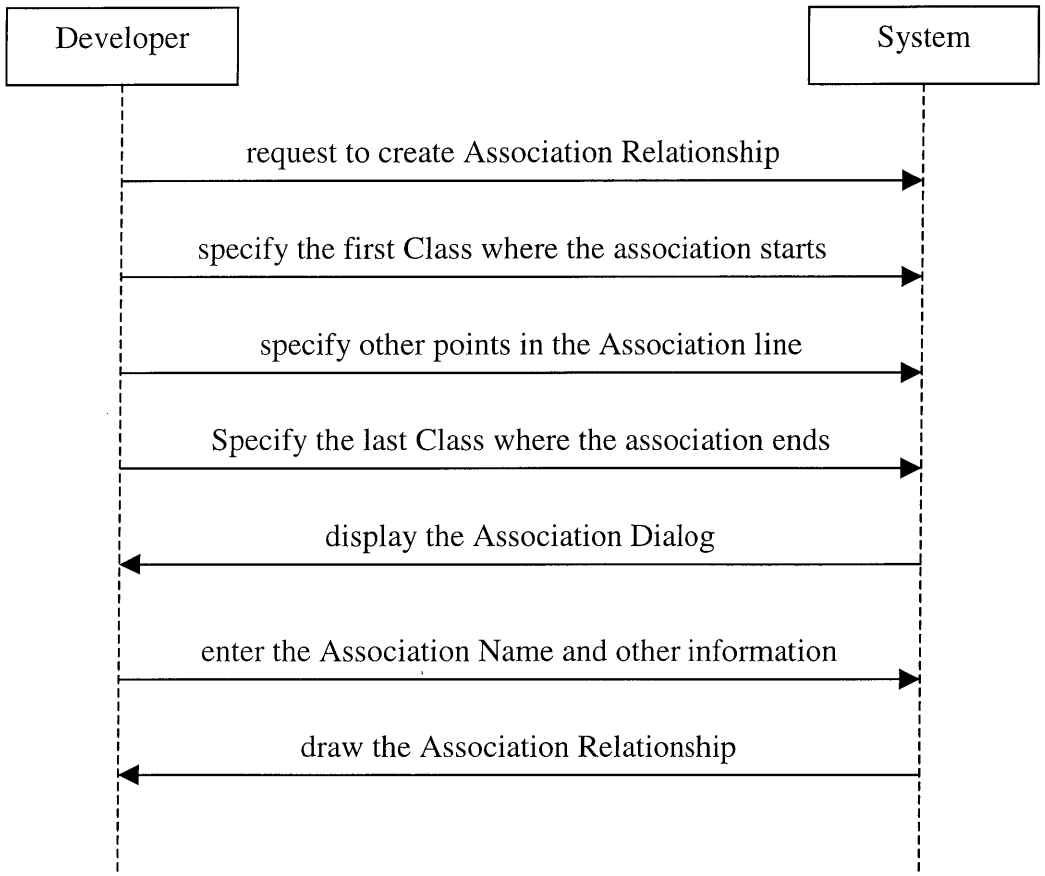


Figure A-10 Create Association Relationship Scenario

A.1.k Create Inheritance Relationship Scenario

Figure A-11 shows Create Inheritance Relationship Scenario.

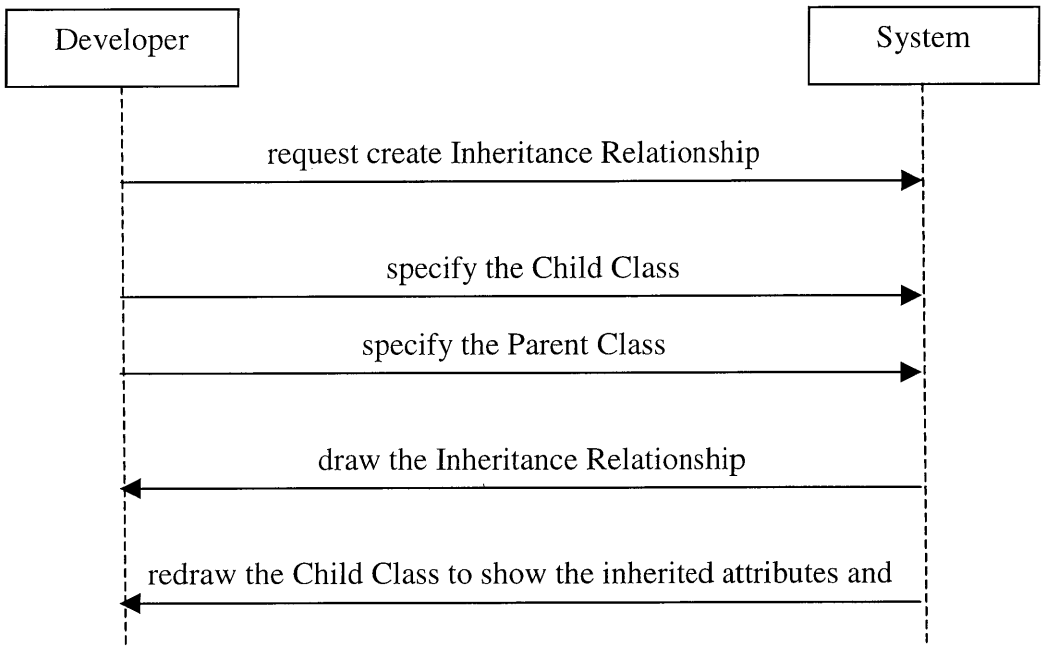


Figure A-11 Create Inheritance Relationship Scenario

A.1.I Create Aggregation Relationship Scenario

Figure A-12 shows the Create Aggregation Relationship Scenario.

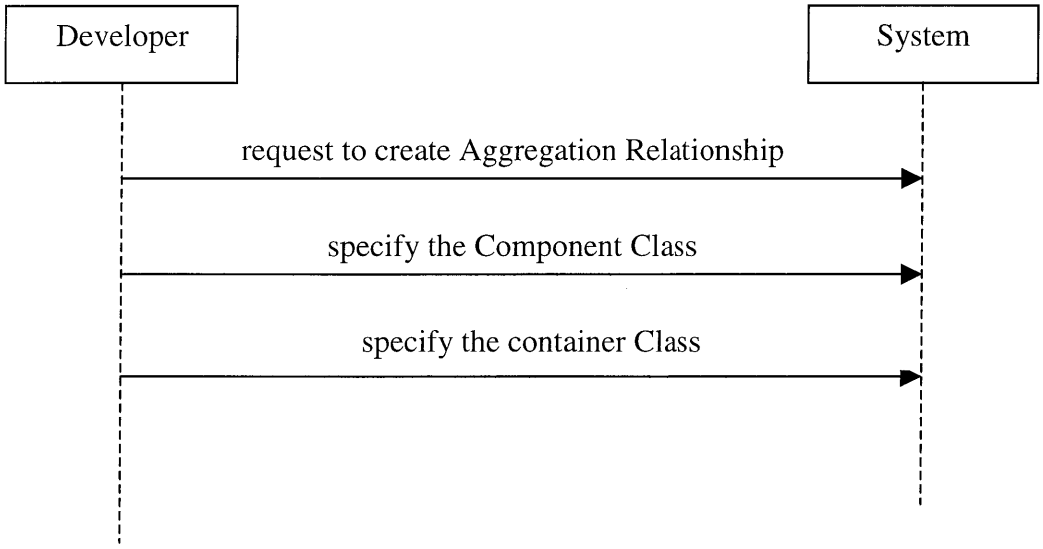


Figure A-12 Create Aggregation Relationship Scenario

A.1.m Delete Class Scenario

Figure A-13 illustrates Delete Class Scenario.

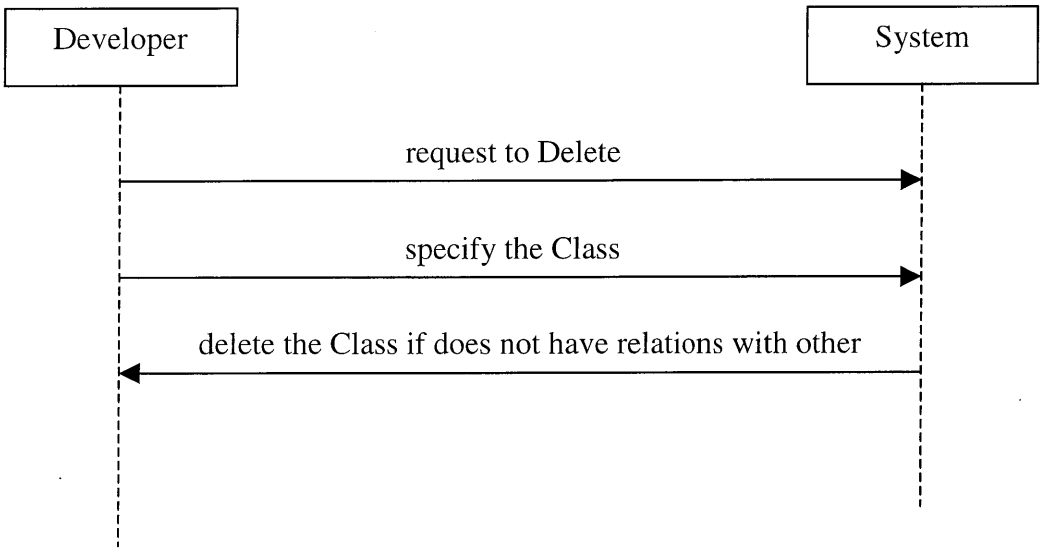


Figure A-13 Delete Class Scenario

A.1.n Delete Association Relationship Scenario

Figure A-14 shows Delete Association Relationship Scenario.

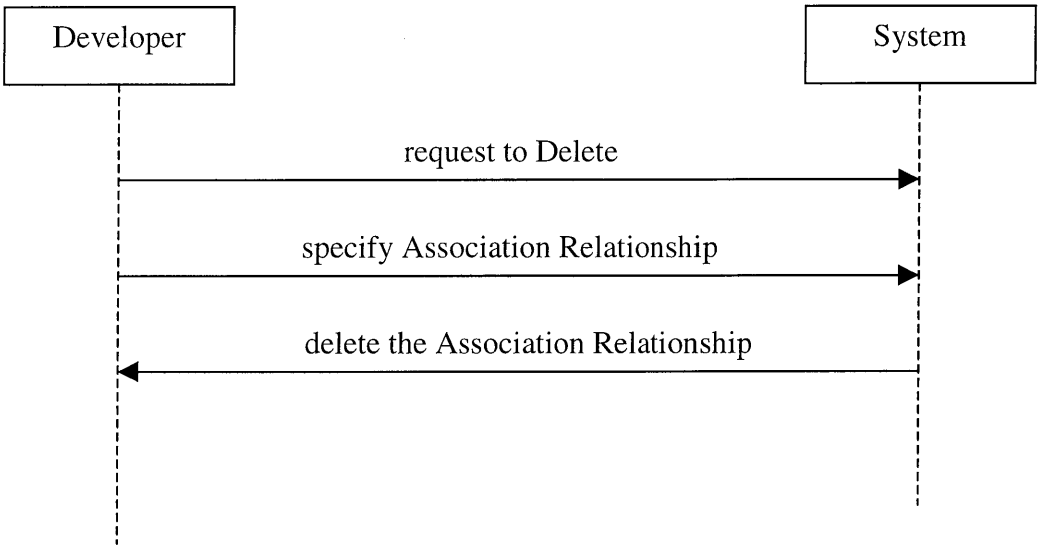


Figure A-14 Delete Association Relationship Scenario

A.1.o Delete Inheritance Relationship Scenario

Figure A-15 shows Delete Inheritance Relationship Scenario.

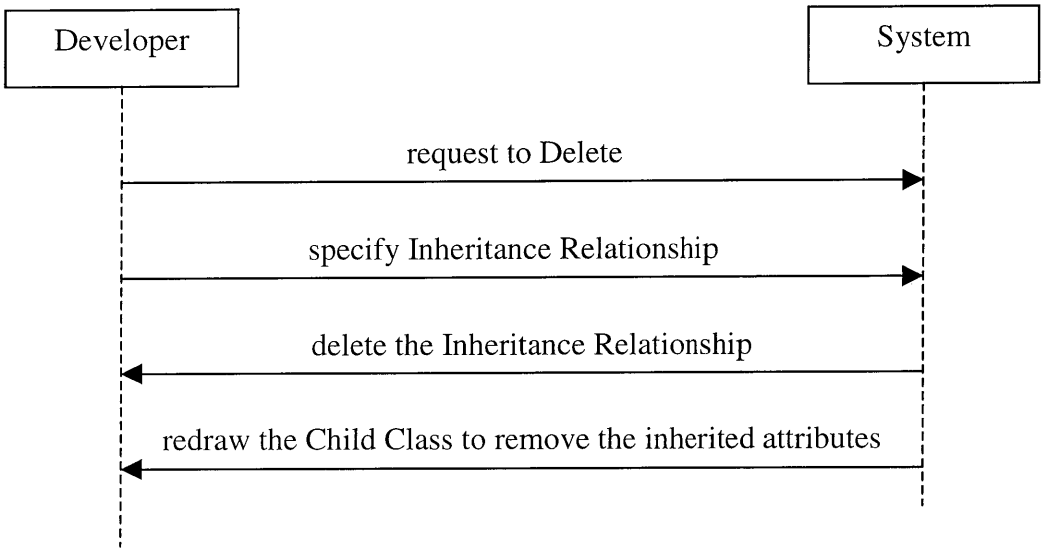


Figure A-15 Delete Inheritance Relationship Scenario

A.1.p Delete Aggregation Relationship Scenario

Figure A-16 shows Delete Aggregation Relationship Scenario.

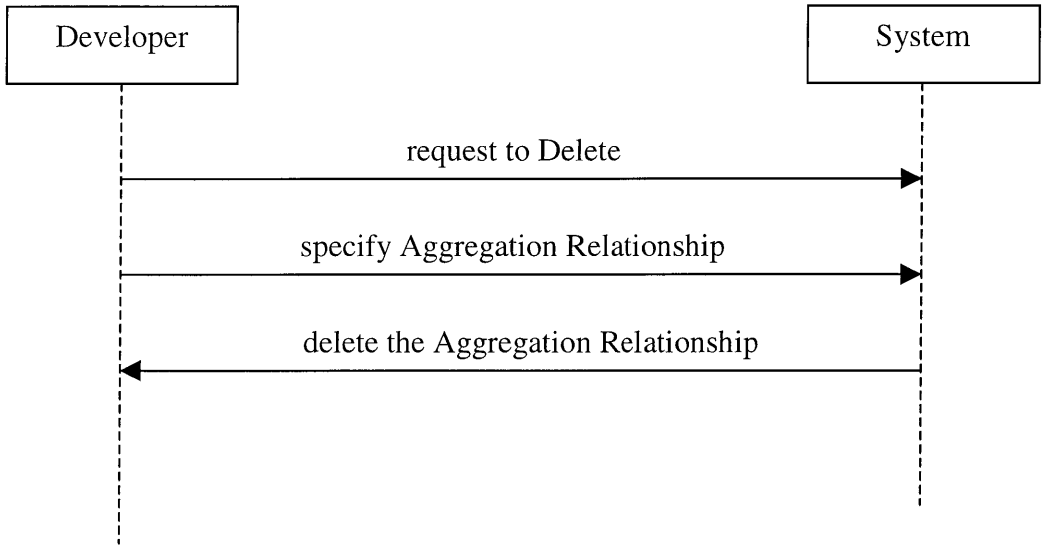


Figure A-16 Delete Aggregation Relationship Scenario

A.1.q Edit Class/Relationship Scenario

Figure A-17 shows Edit Class/Relationship Scenario.

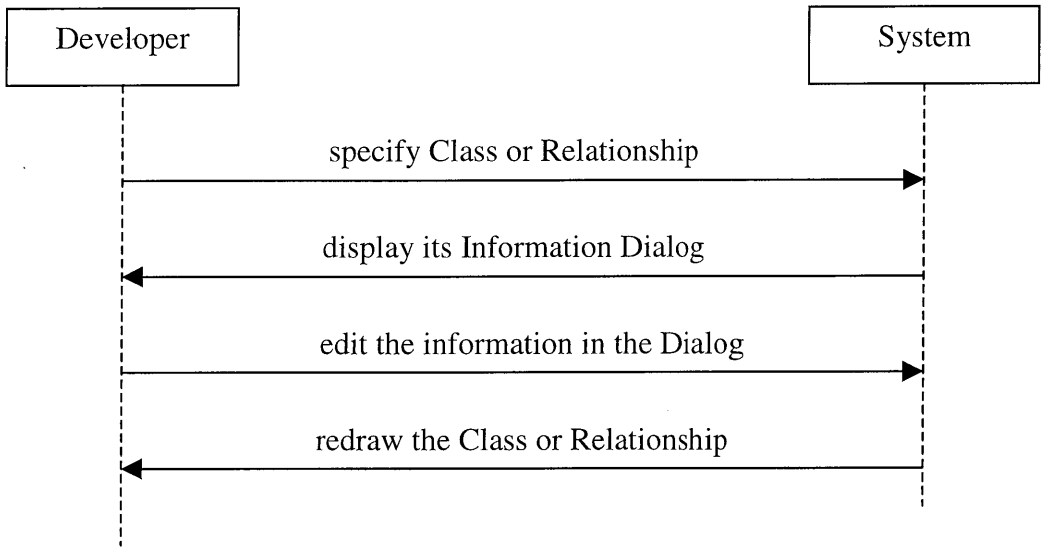


Figure A-17 Edit Class/Relationship Scenario

A.1.r Add Actor to the Scenario

Figure A-18 shows Add Actor Scenario.

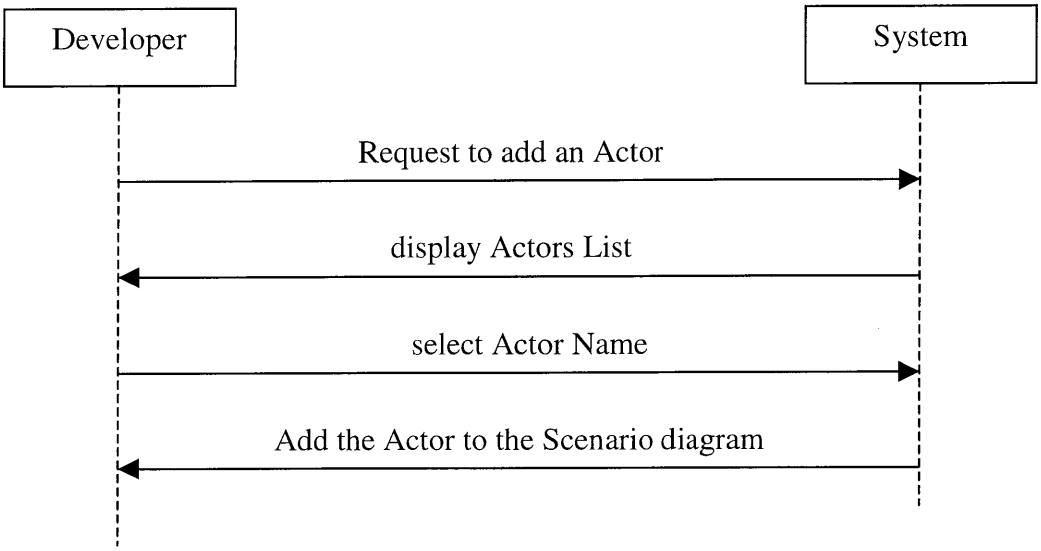


Figure A-18 Add Actor to Scenario

A.1.s Create System Operation Scenario

Figure A-19 shows Create System Operation Scenario.

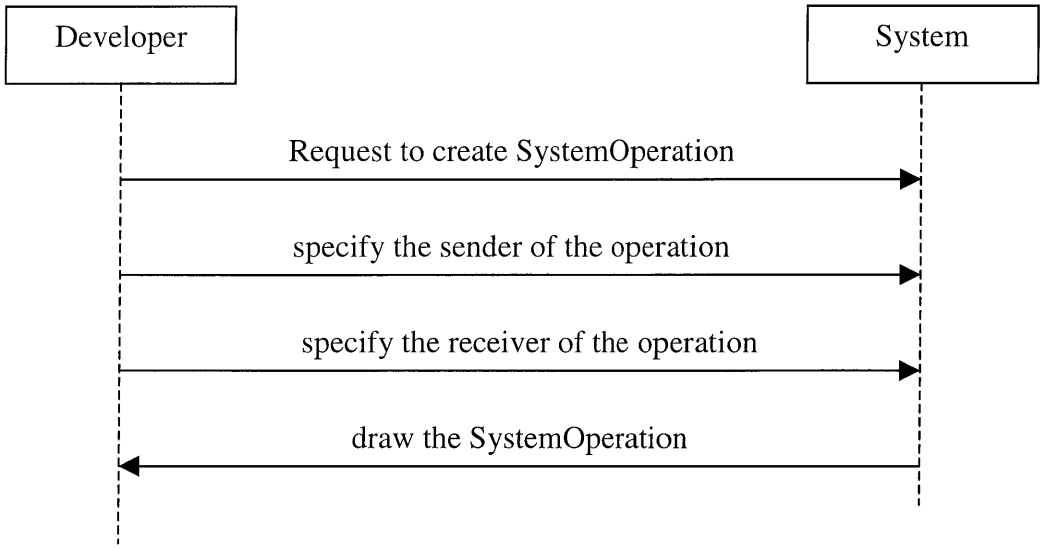


Figure A-19 Create System Operation Scenario

A.1.t Edit System Operation Scenario

Figure A-20 shows Edit System Operation Scenario.

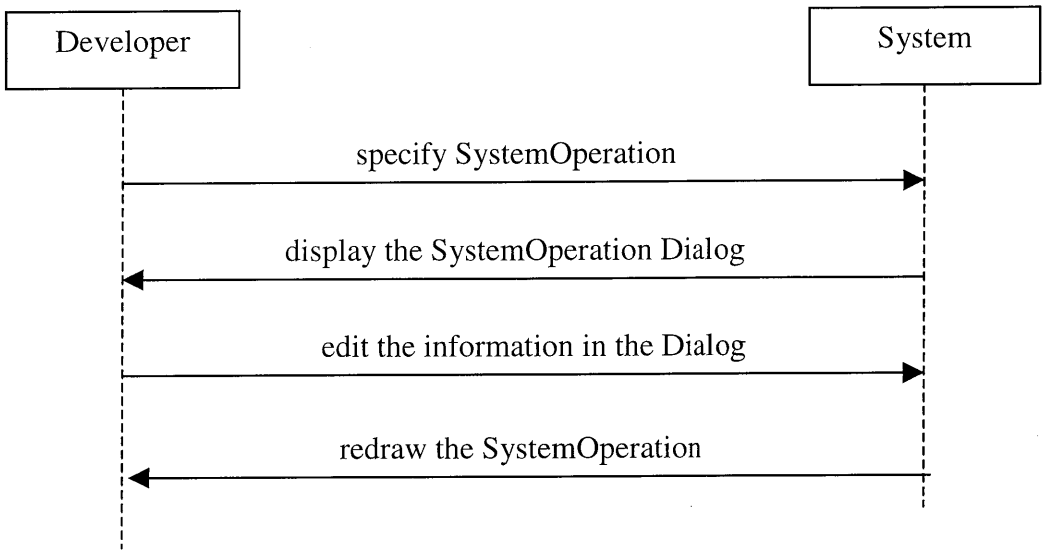


Figure A-20 Edit System Operation Scenario

A.1.u Delete System Operation Scenario

Figure A-21 shows Delete System Operation Scenario.

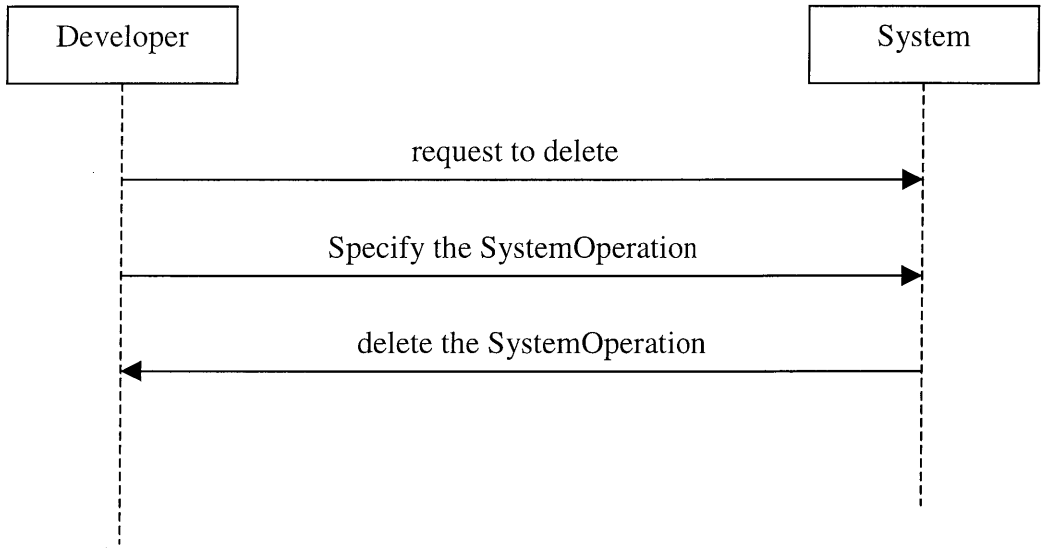


Figure A-21 Delete System Operation Scenario

A.1.v Edit Contract Scenario

Figure A-22 shows Edit Contract Scenario.

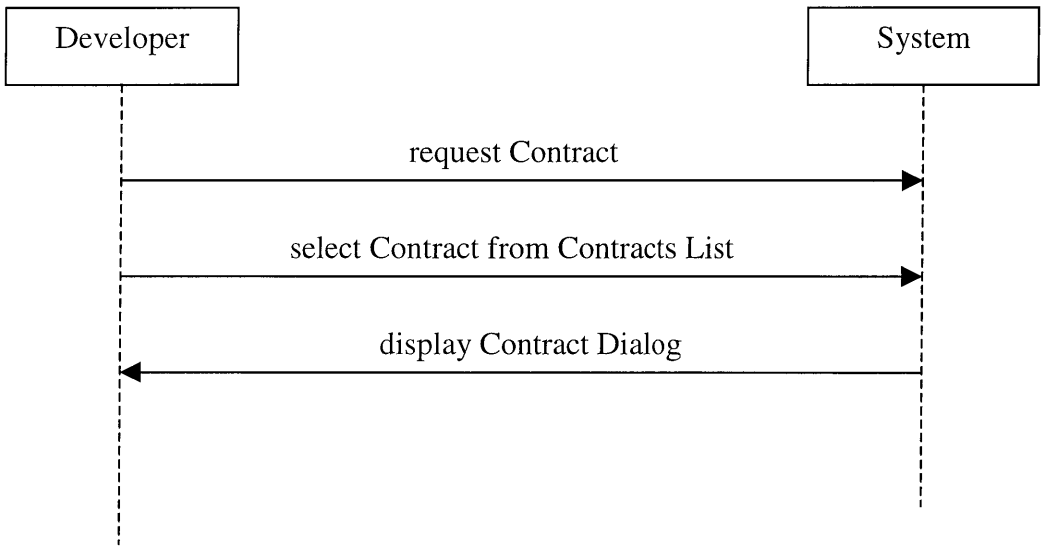


Figure A-22 Edit Contract Scenario

A.1.w Display Information Dialog Scenario

Figure A-23 shows the Display Information Dialog Scenario. This scenario represents the operations that should be done to display the information dialog of any object in the model.

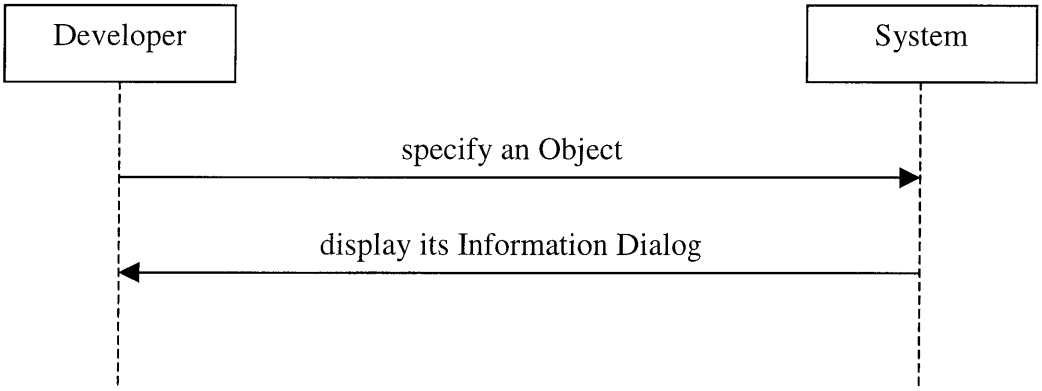


Figure A-23 Display Information Dialog Scenario

A.2 File Group Scenarios

The File group scenarios corresponds to the File Use-Case Model. The scenarios in this group describe the operations that occur in starting new project, open file, and save file use-cases.

A.2.a New Project Scenario

Figure A-24 shows New Project Scenario.

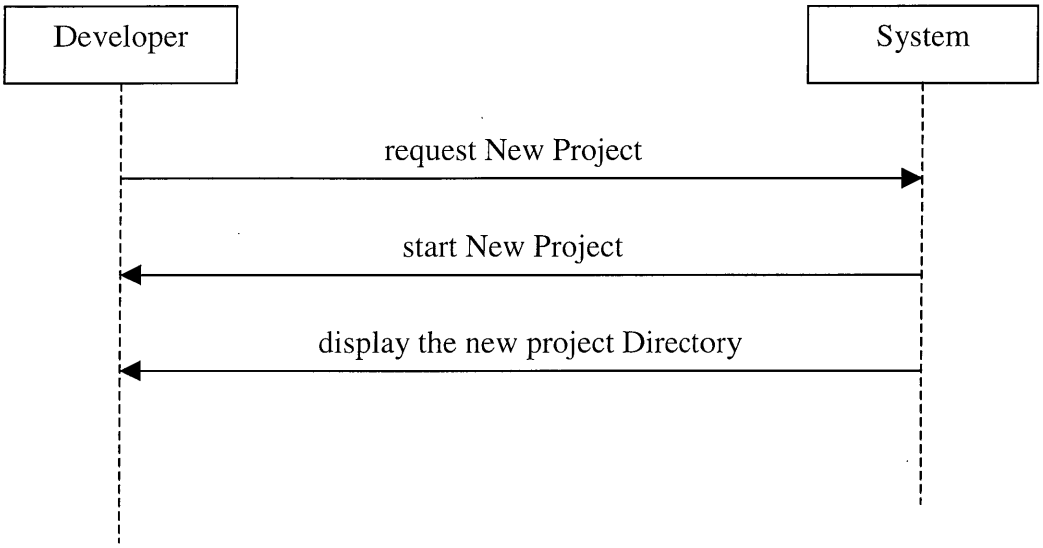


Figure A-24 NewProject Scenario

A.2.b Open File Scenario

Figure A-25 shows Open File Scenario.

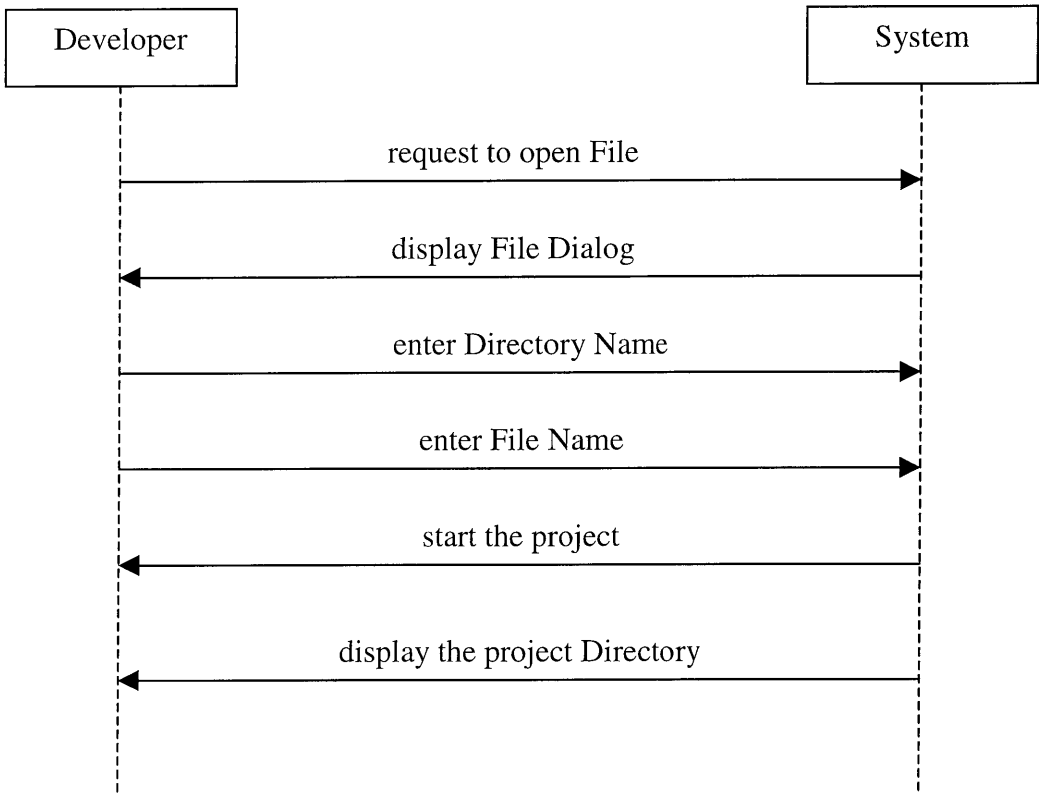


Figure A-25 Open File Scenario

A.2.c Save File Scenario

Figure A-26 shows Save File Scenario.

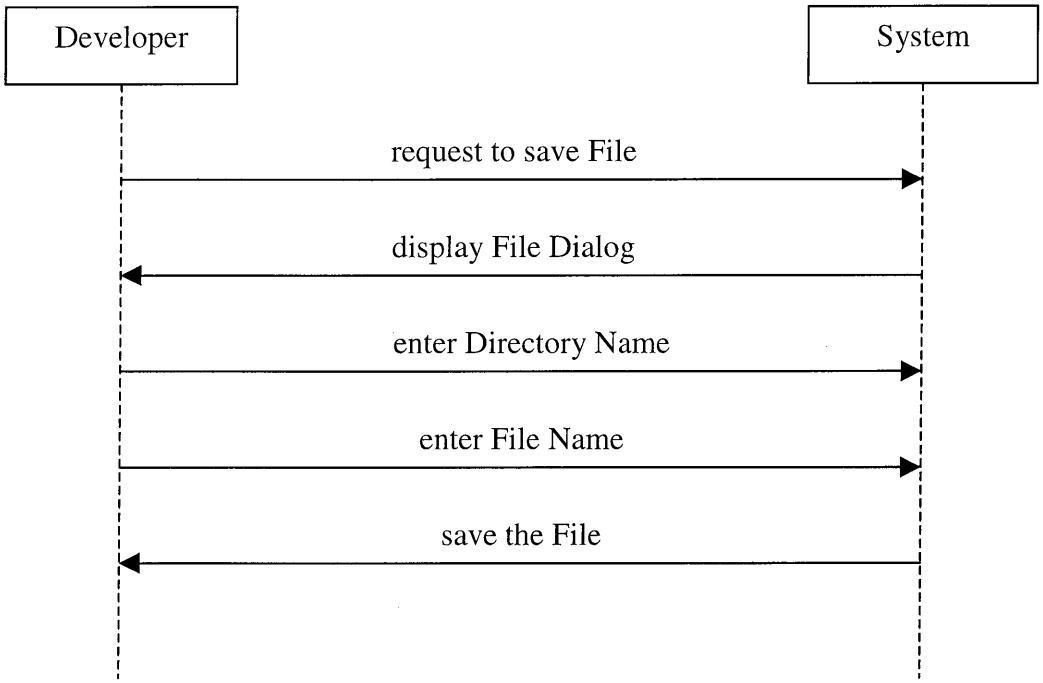


Figure A-26 Save File Scenario

A.3 Window Scenarios Group

The Window Scenarios Group corresponds to the use-cases in the Window Use-Case Model. It contains the scenarios, which show the operations involved in tile windows across.

A.3.a Tile Windows Across Scenario

Figure A-27 shows Tile Windows Across Scenario.

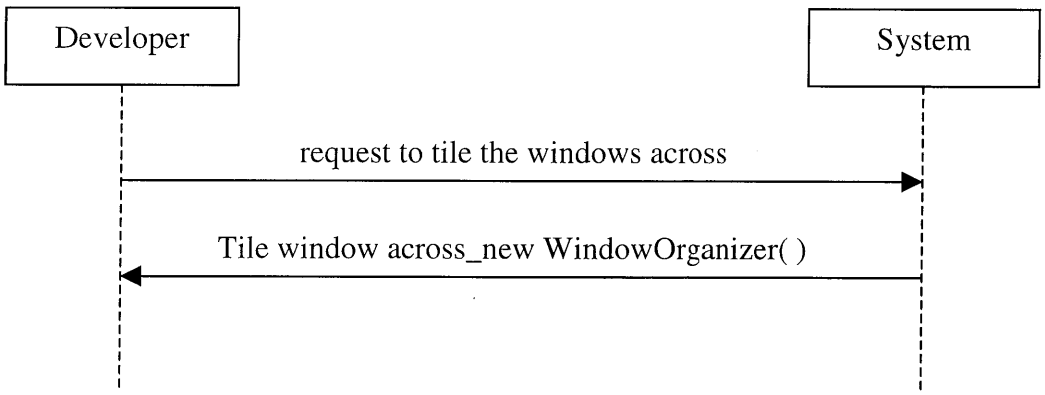


Figure A-27 Tile Windows Across Scenario

A.4 Development Life Cycle Scenarios Group

The Development Life-Cycle Scenarios Group corresponds to the use-cases in the Development Life Cycle Use-Case Model such as display the Development Life Cycle and check the development status of a model.

A.4.a Display the Development Life-Cycle Scenario

Figure A-28 illustrates Display the Development Life-Cycle Scenario.

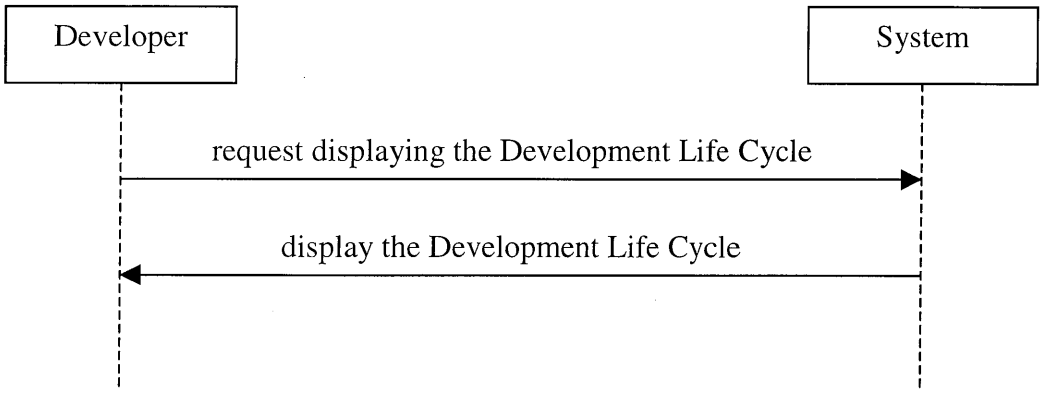


Figure A-28 Display Development Life-Cycle Scenario

A.4.b Checks a Model's Development Status

Figure A-29 shows Check a Model Development Status.

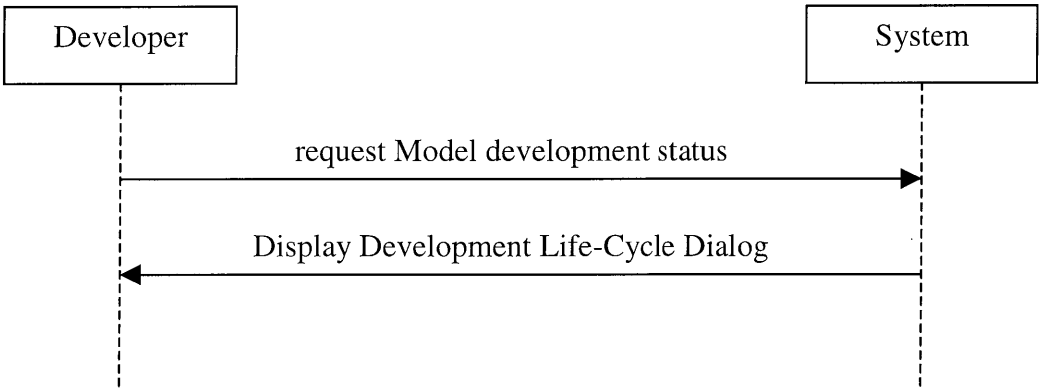


Figure A-29 Check Model Development Status Scenario

A.5 Directory Scenarios Group

The Directory Scenarios Group represents the scenarios for the use-cases in the Directory Use-Case Model.

A.5.a Display the Directory Scenario

Figure A-30 shows Display the Directory Scenario.

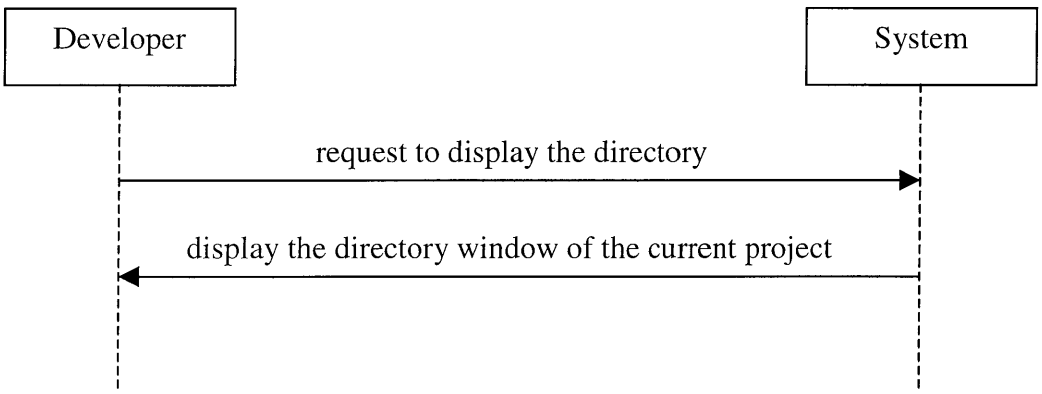


Figure A-30 Display Directory Scenario

A.5.b Select Use-Case Panel Scenario

Figure A-31 shows Select Use-Case Panel Scenario.

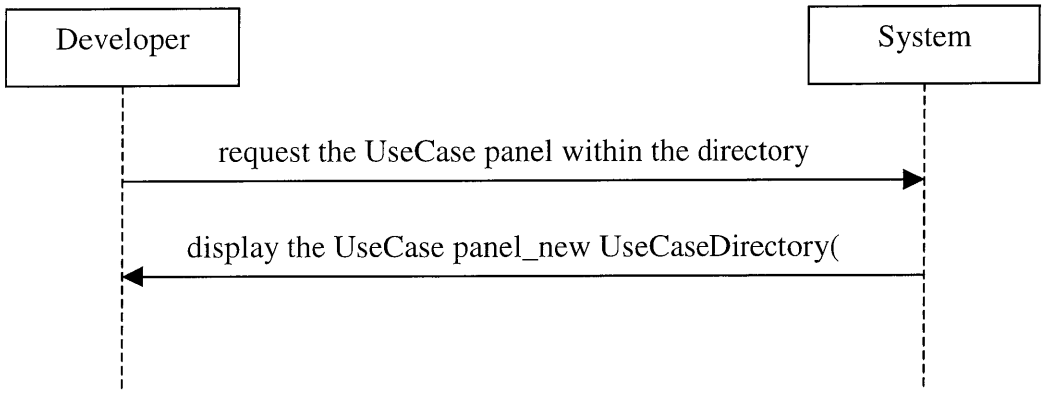


Figure A-31 Select Use-Case Panel from the Directory Scenario

A.5.c Select a Use-Case Scenario

Figure A-32 shows Select Use-Case Scenario.

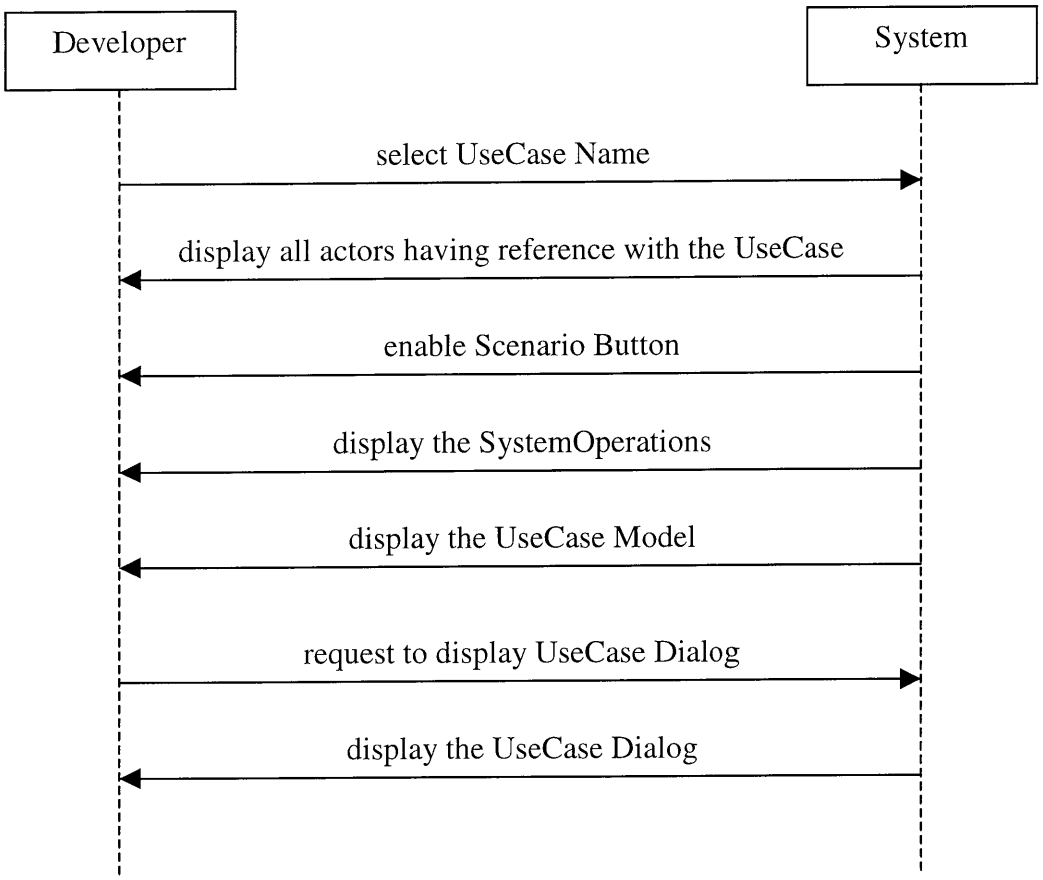


Figure A-32 Selecting Use-Case Scenario

A.5.d Delete Use-Case Scenario

Figure A-33 shows Delete Use-Case Scenario.

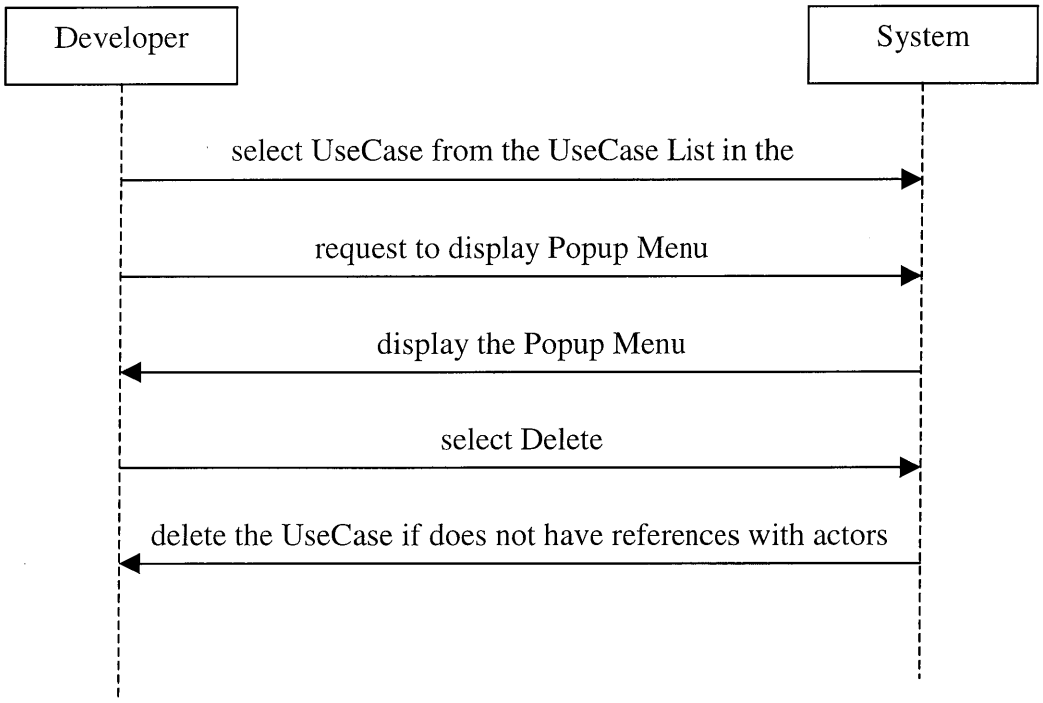


Figure A-33 Delete Use-Case Scenario

A.5.e Select Class Panel Scenario

Figure A-34 shows Select Class Panel Scenario.

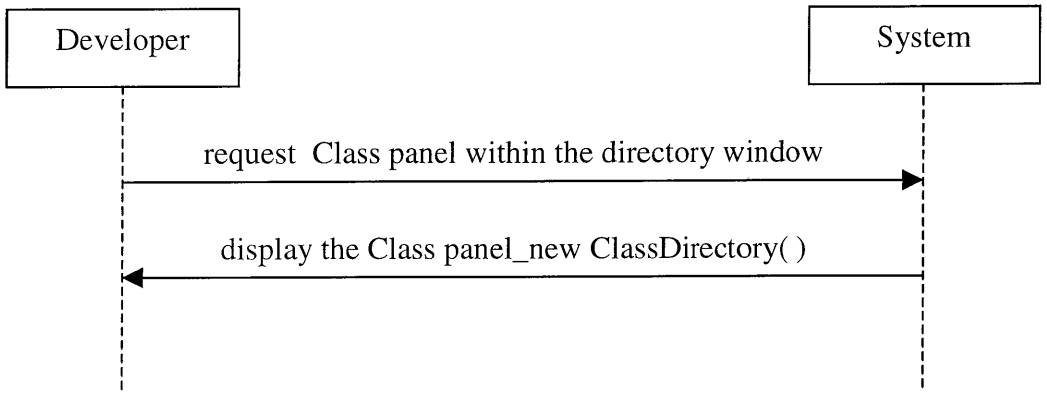


Figure A-34 Select Class Panel from the Directory Scenario

A.5.f Select Class Scenario

Figure A-35 shows Select Class Scenario.

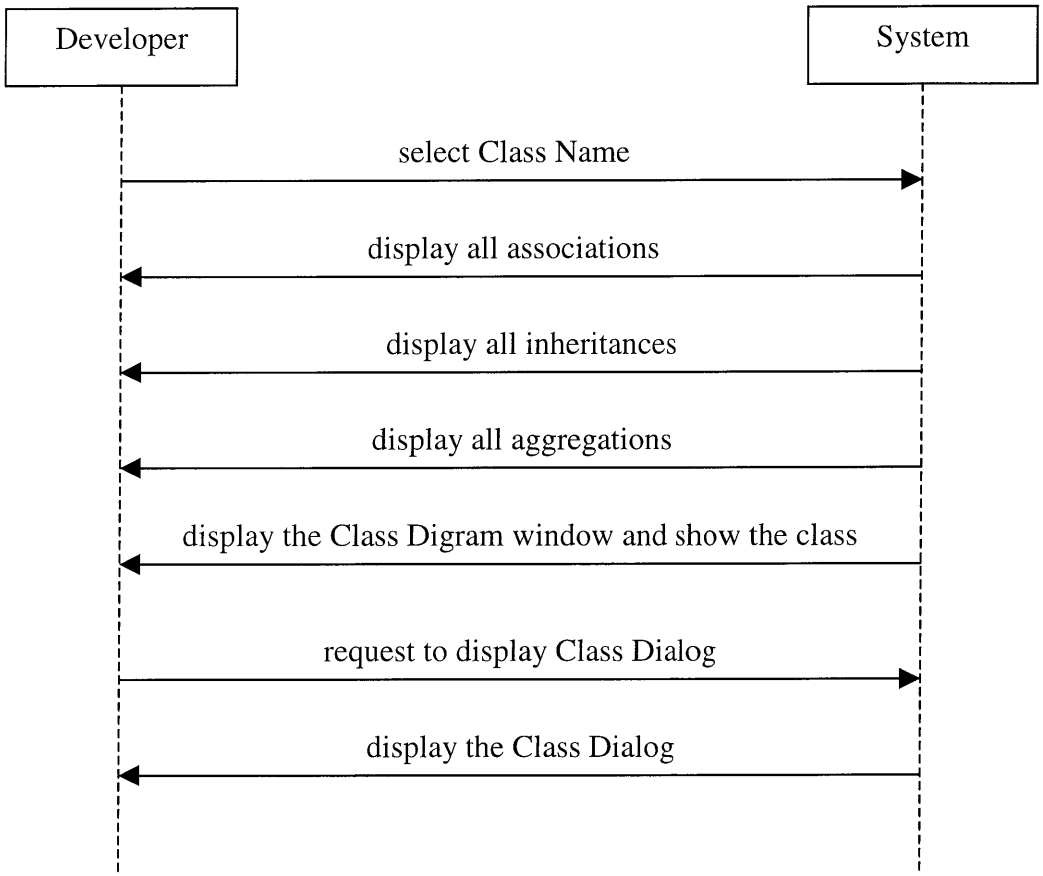


Figure A-35 Select Class Scenario

A.5.g Delete Class Scenario

Figure A-36 shows Delete Class Scenario.

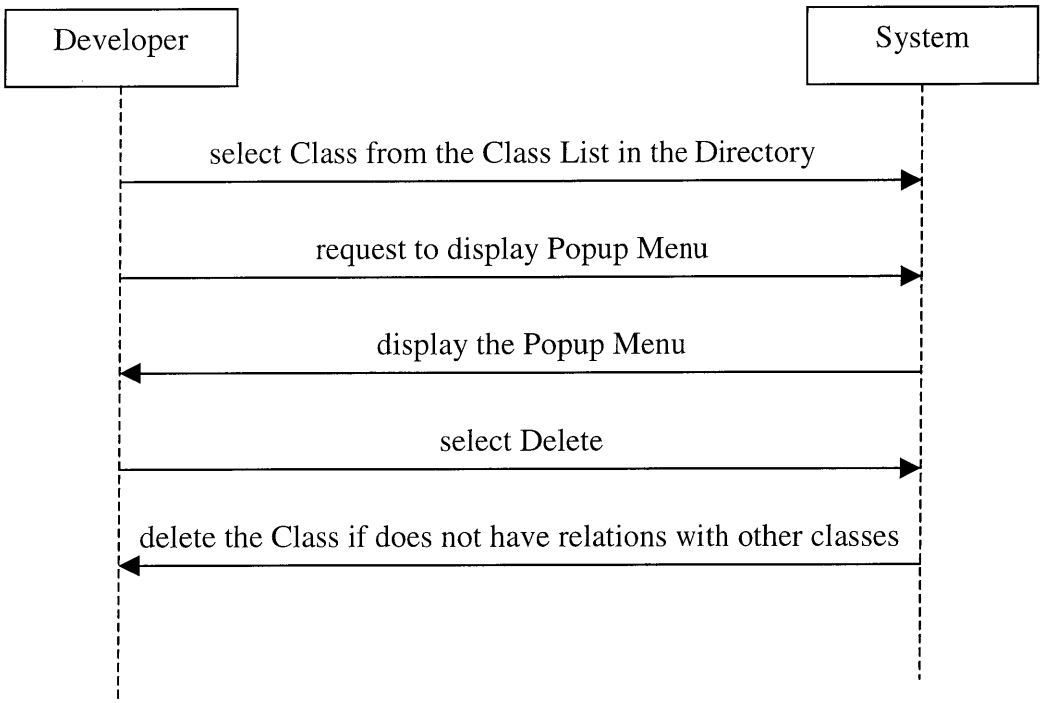


Figure A-36 Delete Class Scenario

Appendix B: Contracts

Appendix B explains the system operations in the scenarios discussed in Appendix A. Each contract describes the system operation name, responsibility, the class contains the system operation or the method, cross references, notes, outputs, preconditions and postconditions.

The next sections explain the contracts. The contract name is the name of the system operation described by it. When several system operations have the same name, the name of the contract is proceeded by the name of the class contains the system operation. In addition, the contract name start with small letter and ends with ( ) to give the feel of a method and indicate that the contract is a description of a method in the class.

B.1 AllModels.actionPerformed( ) Contract

Figure B-1 shows AllModel.actionPerformed( ) Contract

Name:	ActionPerformed (event : ActionEvent)
Responsibilities:	Accept the developer orders to create and edit models
Class:	AllModels
Cross Reference:	Model Use-Case Model CreateModel Scenario BrowseModel Scenario
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	if arg = Model new ModelListDialog if addModel selected if Model does not exist addModel(modelType,modelName) else new AreadyExist Dialog else browseModel(modelType,modelName) else if arg = Print print( ) else if arg = TileAcross new WindowOrganizer(database, arg) else if arg = Development Life-Cycle new DevelopmentLifeCycle(database) else if arg = Directory new Directory(database)

Figure B-1 AllModels.actionPerformed( ) Contract

B.2 addModel( ) Contract

Figure B-2 shows addModel( ) Contract.

Name:	AddModel (modelType:String, modelName:String)
Responsibilities:	Create the models and display them
Class:	AllModels
Cross Reference:	Model Use-Case Model CreateModel Scenario AllModel.actionPerformed() Contract
Notes	
Output:	Display the model frame
Preconditions:	if arg = Model AND addModel
Postconditions:	if modelType = UseCaseModel new UseCaseModel(name, database, allModels, directory) addUseCaseModel(useCaseModelName, UseCaseModel) else if modelType = ClassDiagram new ClassDiagram(name, database, allModels, directory) addClassDiagram(classDiagramName, ClassDiagram) else if modelType = Scenario new Scenario(name, database, allModels, directory) addScenario(scenarioName, Scenario) else if modelType = Contract new ContractModel(name, database, allModels, directory) addContractModel(contractModelName, ContractModel)

Figure B-2 addModel( ) Contract

B.3 browseModel( ) Contract

Figure B-3 shows browseModel( ) Contract.

Name:	browseModel (modelType:String, modelName:String)
Responsibilities:	Retrieve the model from the database and display it
Class:	AllModels
Cross Reference:	Model Use-Case Model BrowseModel Scenario AllModel.actionPerformed() Contract
Notes	
Output:	
Preconditions:	if arg = Model AND modelName selected
Postconditions:	if modelType = UseCaseModel getUseCaseModel(useCaseModelName) else if modelType = ClassDiagram getClassDiagram(classDiagramName) else if modelType = Scenario getScenario(scenarioName) else if modelType = Contract getContractModel(contractModelName)

Figure B-3 browseModel( ) Contract

B.4 UseCaseModel.actionPerformed( ) Contract

Figure B-4 UseCaseModel.actionPerformed( ) Contract.

Name:	actionPerformed (event : ActionEvent)
Responsibilities:	Accept the developer requests to create UseCase, Actor, Reference Accept the developer requests to delete UseCase, Actor, Reference
Class:	UseCaseModel
Cross Reference:	
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	setChoice(choice)

Figure B-4 UseCaseModel.actionPerformed( ) Contract

B.5 useCase( ) Contract

Figure B-5 shows the useCase( ) Contract.

Name:	useCase (x:int, y:int)
Responsibilities:	Create UseCase Draw the UseCase
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	Display AreadyExist Dialog
Preconditions:	If choice = UseCase AND mouseClicked to specify the location of the UseCase
Postconditions:	new UseCase new NameDialog if UseCase does not exist in this canvas new UseCaseLocation(x, y, useCaseName) if UseCase does not exist in allUseCaseModels database.addUseCase(useCaseName, UseCase) directory.adduseCase(UseCase) else new AlreadyExist Dialog paint( ) resetChoice( )

Figure B-5 useCase( ) Contract

B.6 actor( ) Contract

Figure B-6 shows actor( ) Contract.

Name:	actor (x:int, y:int)
Responsibilities:	Create an Actor Draw the Actor
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	Display AreadyExist Dialog
Preconditions:	If choice = Actor AND mouseClicked to specify the location of the Actor
Postconditions:	new Actor new NameDialog if Actor does not exist in this canvas new ActorLocation(x, y, actorName) if Actor does not exist in allUseCaseModels database.addActor(actorName, Actor) else new AlreadyExist Dialog paint( ) resetChoice( )

Figure B-6 actor( ) Contract

B.7 reference( ) Contract

Figure B-7 shows the reference( ) Contract.

Name:	reference (point:Point)
Responsibilities:	Create Reference Draw the Reference
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Reference
Postconditions:	set pointInUseCase to findUseCase(x,y) set pointInActor to findActor(x,y) if firstPoint if mouseClicked in UseCase new Reference(point, UseCase) else if mouseClicked in Actor new Reference(point,Actor) else if pointInUseCase not empty AND firstPoint In Actor setUseCase(UseCase) else if pointInActor not empty AND firstPoint in UseCase setActor(Actor) addPoint(point) if Reference exist in this model new AlreadyExist Dialog else if Reference does not exist in allModels addReference(Reference) paint( ) resetChoice( )

Figure B-7 reference( ) Contract

B.8 deleteUseCaseModelObject( ) Contract

Figure B-8 shows deleteUseCaseModelObject( ) Contract.

Name:	deleteUseCase (:UseCaseModelObject)
Responsibilities:	Delete UseCase or Actor
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	Display DoNotDelete Dialog
Preconditions:	If choice = Delete AND (UseCase OR Actor) is selected
Postconditions:	if object instance of UseCase if UseCase has reference with actor new DoNotDeleteDialog else paint( ) else if object instance of Actor if Actor has reference with usecase new DoNotDeleteDialog else paint( ) resetChoice( )

Figure B-8 deleteUseCaseModelObject( ) Contract

B.9 deleteReference( ) Contract

Figure B-9 shows deleteReference( ) Contract.

Name:	DeleteReference (index:int)
Responsibilities:	Delete Reference
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Delete AND Reference is selected
Postconditions:	getScenario(name) deleteActor(name) deleteReference( ) paint( ) resetChoice( )

Figure B-9 deleteReference( ) Contract

B.10 UseCaseModelCanvas.mouseClicked( ) Contract

Figure B-10 shows UseCaseModelCanvas.mouseCliked( ) Contract.

Name:	MouseClicked (event:MouseEvent)
Responsibilities:	Accepts the developer input of the location of UseCase, Actor, Reference
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	set pointInUseCase to findUseCase(x,y) set pointInActor to findActor(x,y) set poinInReference to findReference(x,y) if choice = UseCase AND pointInUseCase is empty AND pointInActor is empty useCase(x,y) else if choice = Actor AND pointInUseCase is empty AND pointInActor is empty actor(x,y) else if choice = delete AND pointInUseCase is not empty deleteUseCase(point) else if choice = delete AND pointInActor is not empty deleteActor(point) else if choice = delete AND pointInReference deleteReference(point) if click >= 2 AND pointInUseCase is not empty new UseCaseDialog else if click >=2 AND pointInActor is not empty new ActorDialog else if click>=2 AND pointInReference new ReferenceDialog

Figure B-10 mouseClicked( ) Contract

B.11 UseCaseModelCanvas.mousePressed( ) Contract

Figure B-11 shows UseCaseModelCanvas.mousePressed( ) Contract.

Name:	MousePressed (event:MouseEvent)
Responsibilities:	Accepts the developer input of the location of Reference
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	if choice = Reference reference(point)

Figure B-11 mousePressed( ) Contract

B.12 UseCaseModelCanvas.mouseMoved( ) Contract

Figure B-12 shows UseCaseModelCanvas.mouseMoved( ) Contract.

Name:	mouseMoved (event:MouseEvent)
Responsibilities:	Change the cursor shape according to the function
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	findUseCase(x,y) findActor(x,y)

Figure B-12 mouseMoved( ) Contract

B.13 findUseCase( ) Contract

Figure B-13 shows findUseCase( ) Contract.

Name:	findUseCase (x:int, y:int)
Responsibilities:	Ckeck if the mouse is pointing to a UseCase
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	
Postconditions:	

Figure B-13 findUseCase( ) Contract

B.14 findActor( ) Contract

Figure B-14 shows findActor( ) Contract.

Name:	findActor (x:int, y:int)
Responsibilities:	Ckeck if the mouse is pointing to an Actor
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	
Postconditions:	

Figure B-14 findActor( ) Contract

B.15 findReference( ) Contract

Figure B-15 shows findReference( ) Contract.

Name:	findReference (x:int, y:int)
Responsibilities:	Ckeck if the mouse is pointing to a Reference
Class:	UseCaseModelCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	
Postconditions:	

Figure B-15 findReference( ) Contract

B.16 ClassDiagram.actionPerformed( ) Contract

Figure B-16 shows ClassDiagram.actionPerformed( ) Contract.

Name:	actionPerformed (event : ActionEvent)
Responsibilities:	Accept the developer requests to create Class, Relationship Accept the developer requests to delete Class, relationship
Class:	ClassDiagram
Cross Reference:	addModelContract
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	setChoice(choice)

Figure B-16 ClassDiagram.actionPerformed( ) Contract

B.17 createClass( ) Contract

Figure B-17 shows createClass( ) Contract.

Name:	createClass (x:int, y:int)
Responsibilities:	Create Class Draw Class
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = class AND mouseClicked to specify the location of the class
Postconditions:	new ClassClass new ClassDialog if Class does not exist in the database database.addClass(name,Class) new ClassLocation(x,y,name) directory.addClass(Class) else if Class does not exist in this ClassDiagram getClass(name) new ClassDialog new ClassLocation(x,y,name) else getClass(name) new ClassDialog paint( ) resetChoice( )

Figure B-17 createClass( ) Contract

B.18 association( ) Contract

Figure B-18 shows association( ) Contract.

Name:	Association (point:Point)
Responsibilities:	Create Association Relationship Draw Association Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Association AND mousePressed
Postconditions:	findClass(x,y) if firstPoint AND Point in Class new Association(Class,point,ClassLocation) else if firstClass if Point in Class setReciever(Class) setLastPoint( ) addPoint(ClassPoint) else addPoint(point) if lastClass new AssociationDialog getAssociation( ) if Association exist in Database new AreadyExistDialog else assAssociation(Association) paint( ) resetChoice( )

Figure B-18 association( ) Contract

B.19 inheritance( ) Contract

Figure B-19 shows the inheritance( ) Contract.

Name:	inheritance (point:Point)
Responsibilities:	Create Inheritance Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Inheritance AND mousePressed
Postconditions:	findClass(x,y) if firstPoint AND Point in Class new Inheritance(Class,point,ClassLocation) else if firstClass if Point in Class setReciever(Class) getSender( ) if Inheritance does not exist in database setChildAttributes( ) setChildMethods( ) setLastPoint( ) addPoint( ) getSender( ).setParentClass(Class) for all inheritance relationships in database setChildAttributes( ) setChildMethods( ) else new AlreadyExistDialog new InheritanceDialog paint( ) resetChoice( )

Figure B-19 inheritance( ) Contract

B.20 aggregation( ) Contract

Figure B-20 shows the aggregation( ) Contract.

Name:	aggregation (point:Point)
Responsibilities:	Create Aggregation relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Aggregation AND mousePressed
Postconditions:	findClass(x,y) if firstPoint AND Point in Class new Aggregation(Class,point,ClassLocation) else if firstClass if Point in Class setReciever(Class) getSender( ) if Aggregation does not exist in Database setLastPoint( ) addPoint(point) addPartClass(sender) else new AreadyExistDialog new AggregationDialog paint( ) resetChoice( )

Figure B-20 aggregation( ) Contract

B.21 deleteClass( ) Contract

Figure B-21 shows deleteClass( ) Contract.

Name:	deleteClass (name:String)
Responsibilities:	Delete Class
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = delete AND mouseClicked in Class
Postconditions:	if Class has relations with other classes new DoNotDeleteDialog else paint( ) resetChoice( )

Figure B-21 deleteClass( ) Contract

B.22 deleteAssociation( ) Contract

Figure B-22 shows deleteAssociation( ) Contract.

Name:	deleteAssociation (index:int)
Responsibilities:	Delete an Association Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = delete AND mouseClicked in Association
Postconditions:	getAssociations() deleteAssociation(index) paint( ) resetChoice( )

Figure B-22 deleteAssociation( ) Contract

B.23 deleteInheritance( ) Contract

Figure B-23 shows deleteInheritance( ) Contract.

Name:	deleteInheritance (index:int)
Responsibilities:	Delete an Inheritance Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = delete AND mouseClicked in Inheritance
Postconditions:	getInheritance() getReciever( ) getSender( ) for all children getAttributes( ) for all attributes getName( ) getAttributeSource( ) if attributeSource = classParent removeAttribute(name) for all methods getName( ) getMethodSource( ) if (methodSource = classParent) removeMethod( ) setParentClass(null) deleteInheritance(index) paint( ) resetChoice( )

Figure B-23 deleteInheritance( ) Contract

B.24 deleteAggregation( ) Contract

Figure B-24 shows deleteAggregation( ) Contract.

Name:	deleteAggregation (index:int)
Responsibilities:	Delete an Aggregation Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = delete AND mouseClicked in Aggregation
Postconditions:	getAggregations() getReciever( ) getSender( ) deleteAggregation(index) paint( ) resetChoice( )

Figure B-24 deleteAggregation( ) Contract

B.25 ClassDiagramCanvas.mouseClicked( ) Contract

Figure B-25 shows ClassDiagramCanvas.mouseClicked( ) Contract.

Name:	mouseClicked (event:MouseEvent)
Responsibilities:	Accepts the developer input of the location of Class, Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	findClass(x,y) findAssociation(x,y) findInheritance(x,y) findAggregation(x,y) if choice = Class AND point not in class createClass(x,y) else if choice = delete AND point in class deleteClass(name) else if choice = delete AND point in Association deleteAssociation(index) else if choice = delete AND point in Inheritance deleteInheritance(index) else if choice = delete AND point in Aggregation deleteAggregation(index) if click >= 2 AND point in class new ClassDialog(name) else if click >= 2 AND point in Association new AssociationDialog(index) else if click >=2 AND point in Inheritance new InheritanceDialog(index) else if click>=2 AND point in Aggregation new AggregationDialog(index)

Figure B-25 ClassDisgramCanvas.mouseClicked( ) Contract

B.26 ClassDiagramCanvas.mousePressed( ) Contract

Figure B-26 shows ClassDiagramCanvas.mousePressed( ) Contract.

Name:	mousePressed (event:MouseEvent)
Responsibilities:	Accepts the developer input of the location of Relationship
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	If choice = Association AND point in class association(point) else if choice = Inheritance AND point in Class inheritance(point) else if choice = Aggregation AND point in Class aggregation(point)

Figure B-26 ClassDiagramCanvas.mousePressed( ) Contract

B.27 ClassDiagramCanvas.mouseMoved( ) Contract

Figure B-27 shows ClassDiagramCanvas.mouseMoved( ) Contract.

Name:	mouseMoved (event:MouseEvent)
Responsibilities:	Change the cursor shape according to the function
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	

Figure B-27 ClassDigramCanvas.mouseMoved( ) Contract

B.28 findClass( ) Contract

Figure B-28 shows findClass( ) Contract.

Name:	findClass (x:int, y:int)
Responsibilities:	Find the class where the point is located
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	getClassLocation( ) getClassName( ) getClassWidth( ) getClassHeight( )

Figure B-28 findClass( ) Contract

B.29findAssociation( ) Contract

Figure B-29 shows findAssociation( ) Contract.

Name:	findAssociation (x:int, y:int)
Responsibilities:	Find the Association where the point is located
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	getPoint( )

Figure B-29 findAssociation( ) Contract

B.30 findInheritance( ) Contract

Figure B-30 shows findInheritance( ) Contract.

Name:	FindInheritance (x:int, y:int)
Responsibilities:	Find the Inheritance where the point is located
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	GetPointsSize( ) getPoint( )

Figure B-30 findInheritance( ) Contract

B.31 findAggregation( ) Contract

Figure B-31 shows findAggregation( ) Contract.

Name:	findAggregation (x:int, y:int)
Responsibilities:	Find the Aggregation where the point is located
Class:	ClassDiagramCanvas
Cross Reference:	
Notes	
Output:	
Preconditions:	If mouseEvent is fired
Postconditions:	GetPointsSize( ) GetPoint( )

Figure B-31 findAggragation( ) Contract

B.32 systemOperation( ) Contract

Figure B-32 shows systemOperation( ) Contract.

Name:	systemOperation (point:Point)
Responsibilities:	Create a systemOperation
Class:	Scenario
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = systemOperation AND mouseClicked
Postconditions:	findSystem( ) findActor( ) if firstPoint if point in System new SystemOperation(systemoprationsSize+1, ScenarioSystem) else if point in Actor new SystemOperation(systemoprationsSize+1, ScenarioActorPoint) else if point in System AND firstActor setReciever(ScenarioSystem) else if point in Actor AND firstSystem setReciever(ScenarioActor) setTimeCount(timeCountLine) paint( ) resetChoice( )

Figure B-32 systemOperation( ) Contract

B.33 Scenario.actionPerformed( ) Contract

Figure B-33 shows Scenario.actionPerformed( ) Contract.

Name:	actionPerformed (event : ActionEvent)
Responsibilities:	Accept the developer requests to create Actor, SystemOperation Accept the developer requests to delete Actor, SystemOperation
Class:	Scenario
Cross Reference:	
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	setChoice(choice)

Figure B-33 Scenario.actionPerformed( ) Contract

B.34 scenarioActor( ) Contract

Figure B-34 shows scenarioActor( ) Contract.

Name:	scenarioActor (x:int, y:int)
Responsibilities:	Add an Actor to the Scenario
Class:	Scenario
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = Actor AND mouseClicked
Postconditions:	new ScenarioActorDialog if Actor does not exist in Scenario new ScenarioActorPoint paint( )

Figure B-34 scenarioActor( ) Contract

B.35 deleteSystemOperation( ) Contract

Figure B-35 shows deleteSystemOperation( ) Contract.

Name:	deleteSystemOperation (x:int, y:int)
Responsibilities:	Delete a systemOperation
Class:	Scenario
Cross Reference:	
Notes	
Output:	
Preconditions:	If choice = delete AND point in systemOperation
Postconditions:	SetTimeCount(systemoperationsSize+1) paint( )

Figure B-35 deleteSystemOperation( ) Contract

B.36 findsystemoperation( ) contract

Figure B-36 shows findsystemoperation( ) contract.

Name:	FindSystemOperation (x:int, y:int)
Responsibilities:	Find a systemOperation
Class:	Scenario
Cross Reference:	
Notes	
Output:	
Preconditions:	
Postconditions:	if SystemOperation.getReciever exist getTime( ) getSender( ).getActorPoint( ) getReciever( ).getActorPoint( )

Figure B-36 findSystemOperation( ) Contract

B.37 findActor( ) Contract

Figure B-37 shows findActor( ) Contract.

Name:	FindActor (x:int, y:int)
Responsibilities:	Find an Actor
Class:	Scenario
Cross Reference:	
Notes	
Output:	
Preconditions:	
Postconditions:	getActorName( ) getActorWidth( ) getActorHeight( )

Figure B-37 findActor( ) Contract

B.38 Directory.actionPerformed( ) Contract

Figure B-38 shows Directory.actionPerformed( ) Contract.

Name:	actionPerformed (event : ActionEvent)
Responsibilities:	Allows to switch between the Class and the UseCase panels
Class:	Directory
Cross Reference:	Directory UseCase Model
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	layout.show(cards,arg)

Figure B-38 Directory.actionPerformed( ) Contract

B.39 ClassDirectory.itemStateChanged( ) Contract

Figure B-39 shows ClassDirectory.itemStateChanged( ) Contract.

Name:	itemStateChanged (event : ItemEvent)
Responsibilities:	Select Class from Class List Show the selected Class relationships Show the ClassDiagram contains the selected Class
Class:	ClassDirectory
Cross Reference:	
Notes	
Output:	
Preconditions:	ItemEvent is fired
Postconditions:	Association.getSender( ).getClassName( ) Association.getReciever( ).getClassName( ) Inheritance.getSender( ) Inheritance.getReciever( ) Aggregation.getSender( ) Aggregation.getReciever( ) ClassDiagram.getClasses( )

Figure B-39 ClassDirectory.itemStateChanged( ) Contract

B.40 ClassDirectory.actionPerformed( ) Contract

Figure B-40 shows ClassDirectory.actionPerformed( ) Contract.

Name:	actionPerformed (event : ActionEvent)
Responsibilities:	Display ClassDialog
Class:	ClassDirectory
Cross Reference:	
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	if Class is selected new ClassDialog paint( ) else if delete new AreYouSureDialog if Sure getClass(name) if Class has relations new DoNotDeleteDialog else deleteClass(name) database.deleteClass(name)

Figure B-40 ClassDirectory.actionPerformed( ) Contract

B.41 UseCaseDirectory.itemStateChanged( ) Contract

Figure B-41 shows UseCaseDirectory.itemStateChanged( ) Contract.

Name:	itemStateChanged (event : ItemEvent)
Responsibilities:	Select UseCase from UseCase List Display all actors that are referenced by the selected UseCase Enable the Scenario Button Display all the SystemOperations
Class:	UseCaseDirectory
Cross Reference:	
Notes	
Output:	
Preconditions:	ItemEvent is fired
Postconditions:	Reference.getUseCase( ) Reference.getActor( ) Scenario.getSystemOperations( ) SystemOperation.getName( ) Database.getUseCaseModels( ) UseCaseModel.getUseCases( )

Figure B-41 UseCaseDirectory.itemStateChanged( ) contract

B.42 UseCaseDirectory.actionPerformed( ) Contract

Figure B-42 shows UseCaseDirectory.actionPerformed( ) Contract.

Name:	actionPerformed (event : ActionEvent)
Responsibilities:	Display UseCaseDialog Display Scenario Display SystemOperationDialog
Class:	UseCaseDirectory
Cross Reference:	
Notes	
Output:	
Preconditions:	ActionEvent is fired
Postconditions:	if UseCase is selected new UseCaseDialog else if SystemOperaion is selected Scenario.getModelName( ) SystemOperation.getName( ) New ContractDialog else if delete new AreYouSureDialog if Sure getUseCaseModels( ) if UseCase has references new DoNotDeleteDialog else deleteUseCase(name) database.deleteUseCase(name)

Figure B-42 UseCaseDirectory.actionPerformed( ) Contract

Appendix C: Collaboration Diagrams

Appendix C contains the Collaboration Diagrams developed during the design phase of the LCCASE tool life cycle.

C.1 AllModels.actionPreformed( ) Collaboration Diagram

AllModels.actionPreformed( ) Collaboration Diagram shows the sequence of messages that are called between the objects when the actionPerformed( ) method in the :AllModels object is called.

When the method actionPerformed( ) is called, the :AllModels object creates( ) the object, :ModelListDialog, if the condition arg = = Model is true. The rest of the messages will be sent in the same manner.

Figure C-1 shows AllModels.actionPerformed( ) Collaboration Diagram.

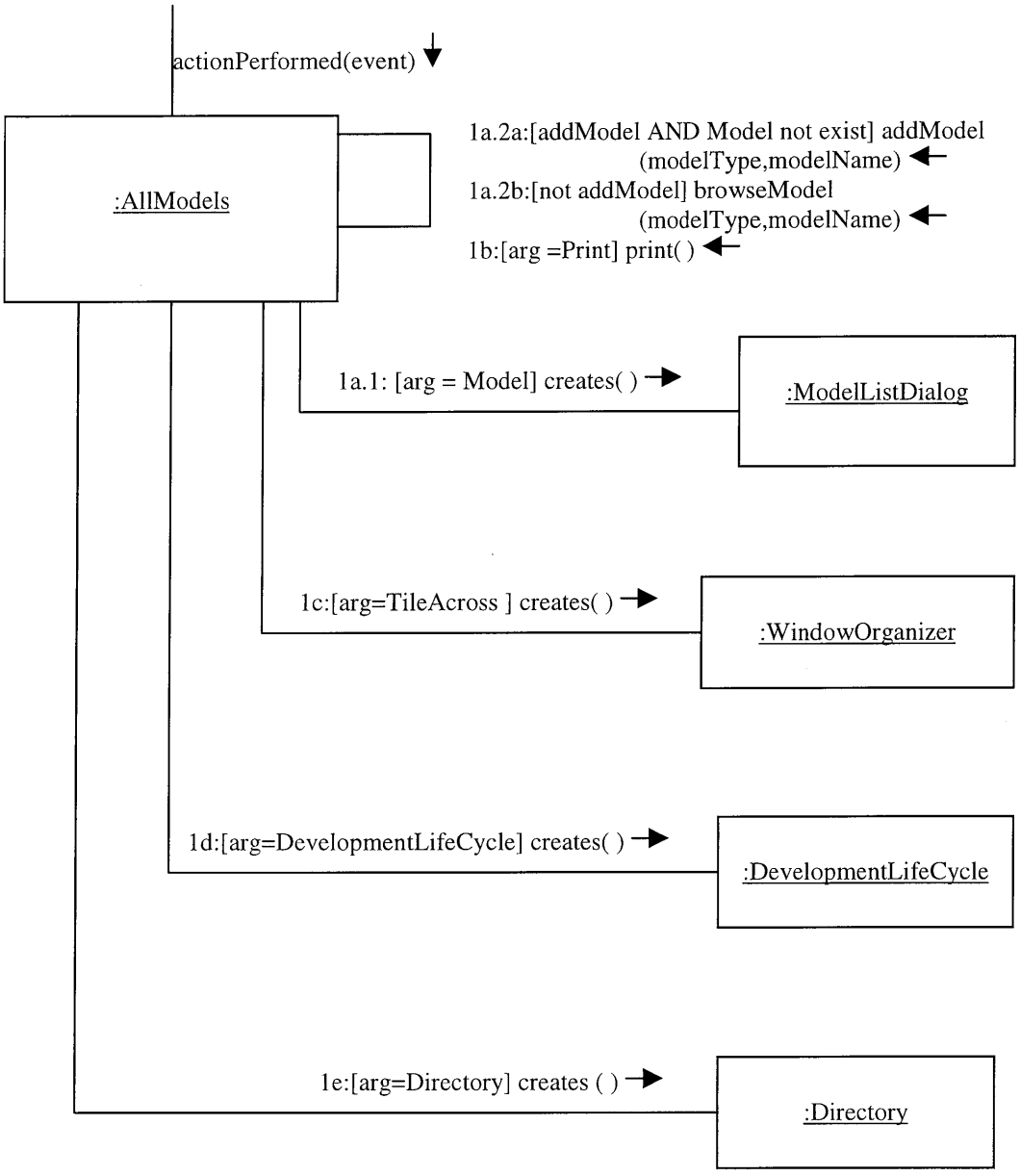


Figure C-1 AllModels.action_Performed( ) Collaboration Diagram

C.2 addModel( ) Collaboration Diagram

Figure C-2 shows addModel( ) Collaboration Diagram.

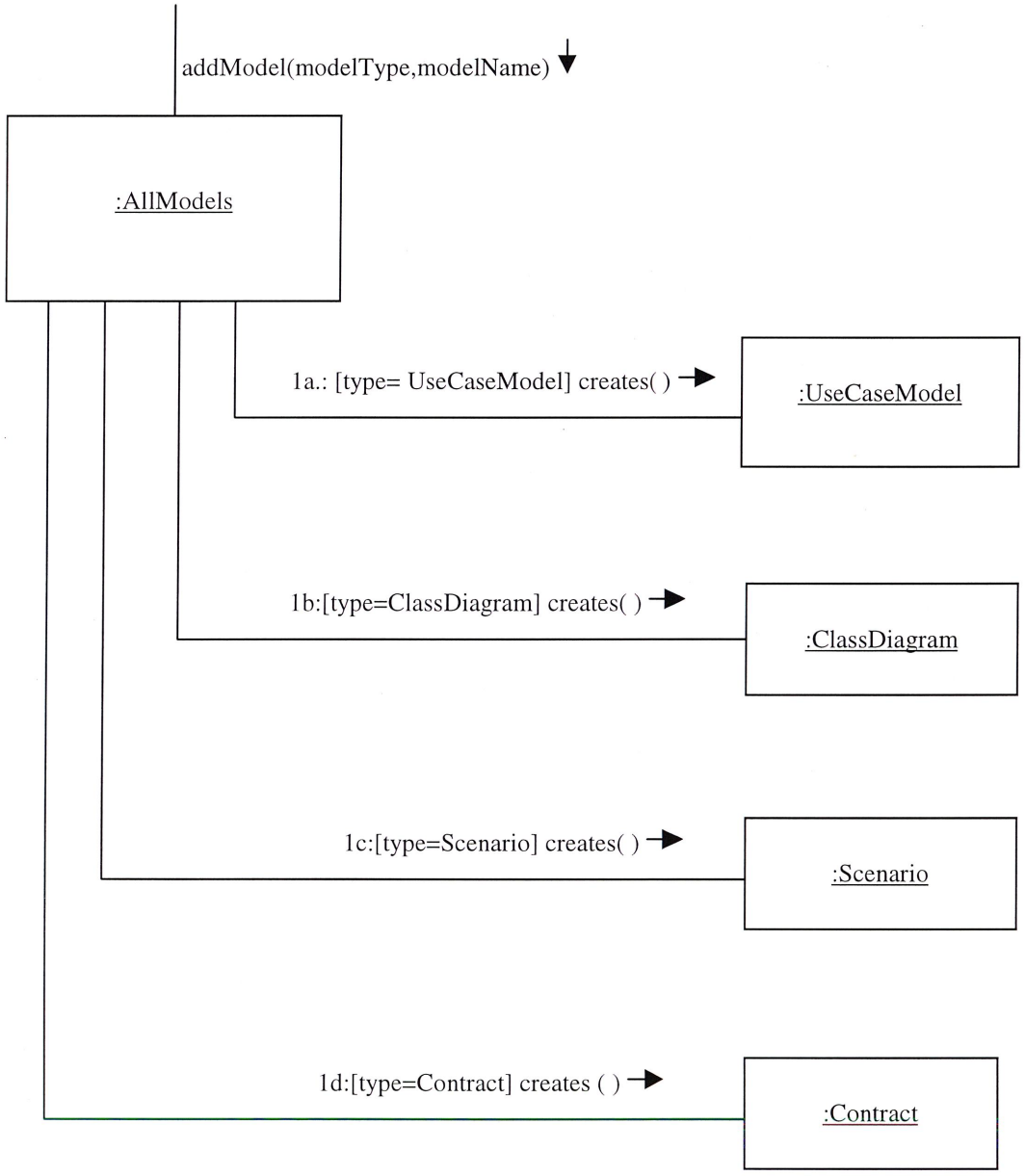


Figure C-2 addModel( ) Collaboration Diagram

C.3 browseModel( ) Collaboration Diagram

Figure C-3 shows browseModel( ) Collaboration Diagram.

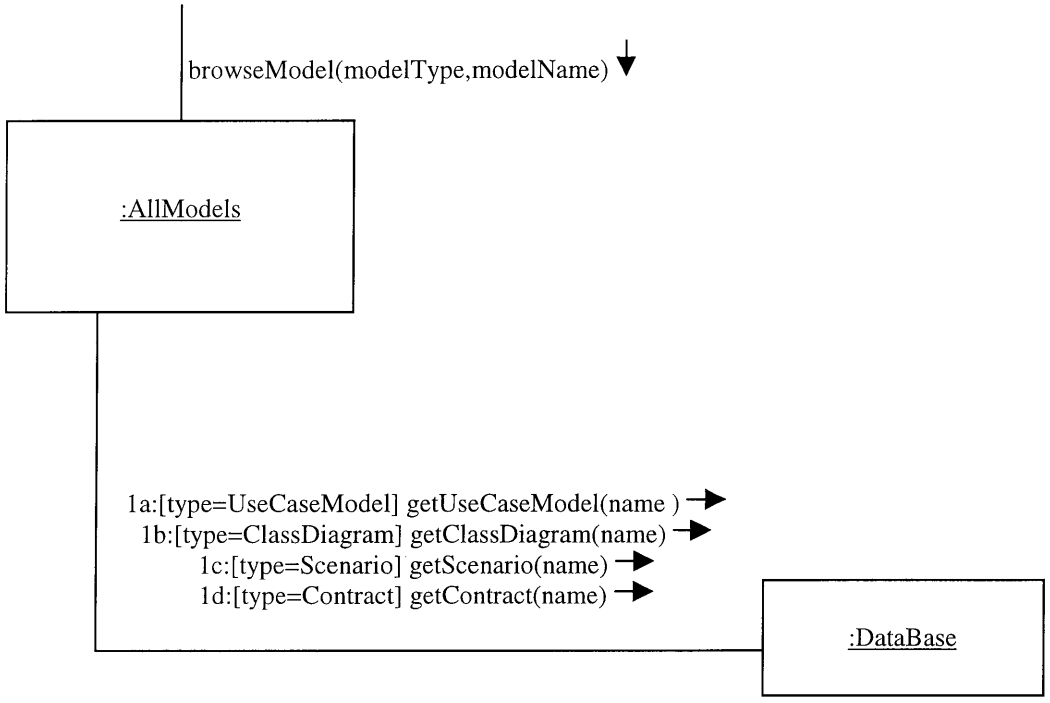


Figure C-3 browseModel( ) Collaboration Diagram

C.4 UseCaseModel.actionPerformed( ) Collaboration Diagram

Figure C-4 shows UseCaseModel.actionPerformed( ) Collaboration Diagram.

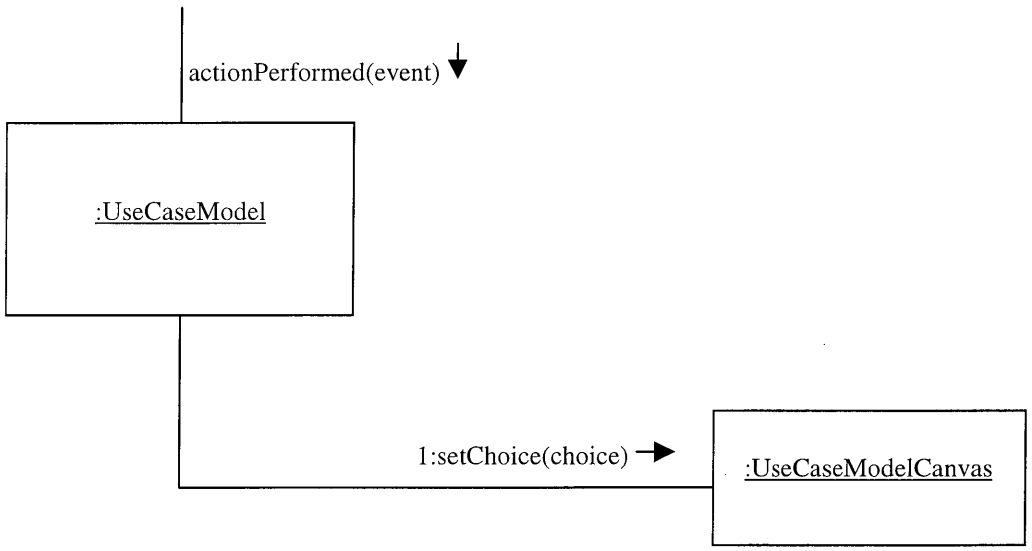


Figure C-4 UseCaseModel.actionPerformed( ) Collaboration Diagram

C.5 useCase( ) Collaboration Diagram

Figure C-5 shows useCase( ) Collaboration Diagram.

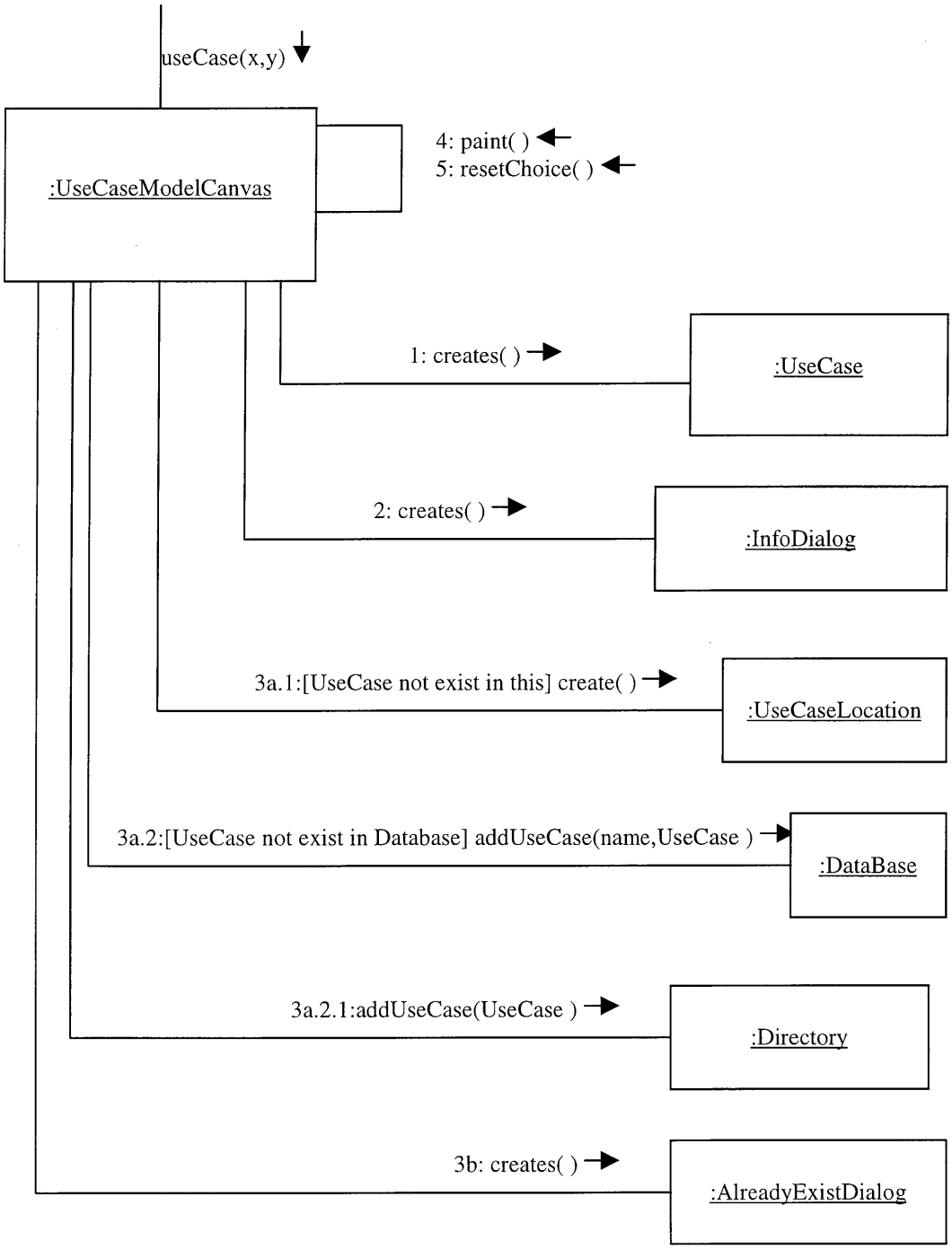


Figure C-5 useCase( ) Collaboration Diagram

C.6 actor( ) Collaboration Diagram

Figure C-6 shows actor( ) Collaboration Diagram.

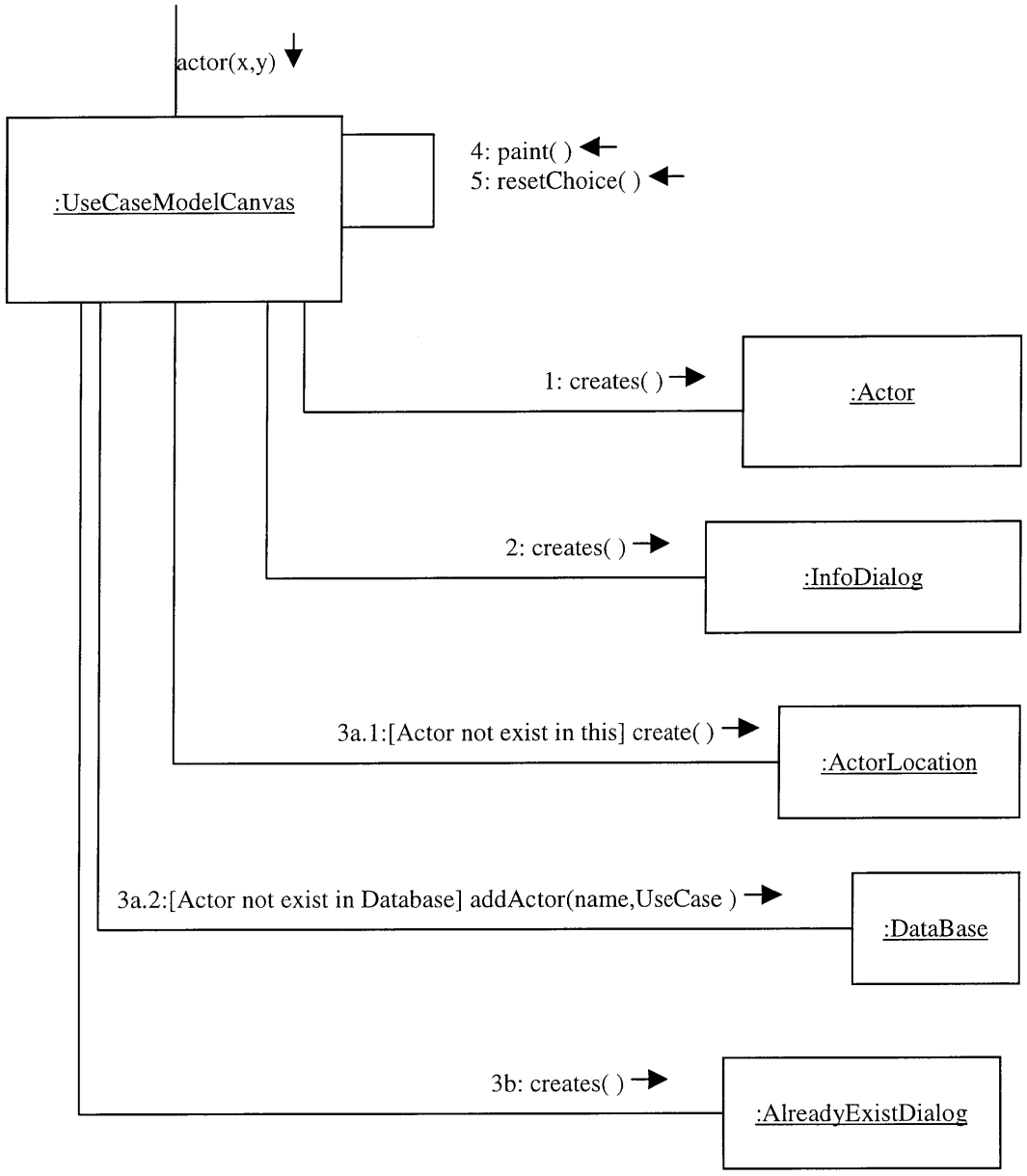


Figure C-6 actor( ) Collaboration Diagram

C.7 reference( ) Collaboration Diagram

Figure C-7 shows reference( ) Collaboration Diagram.

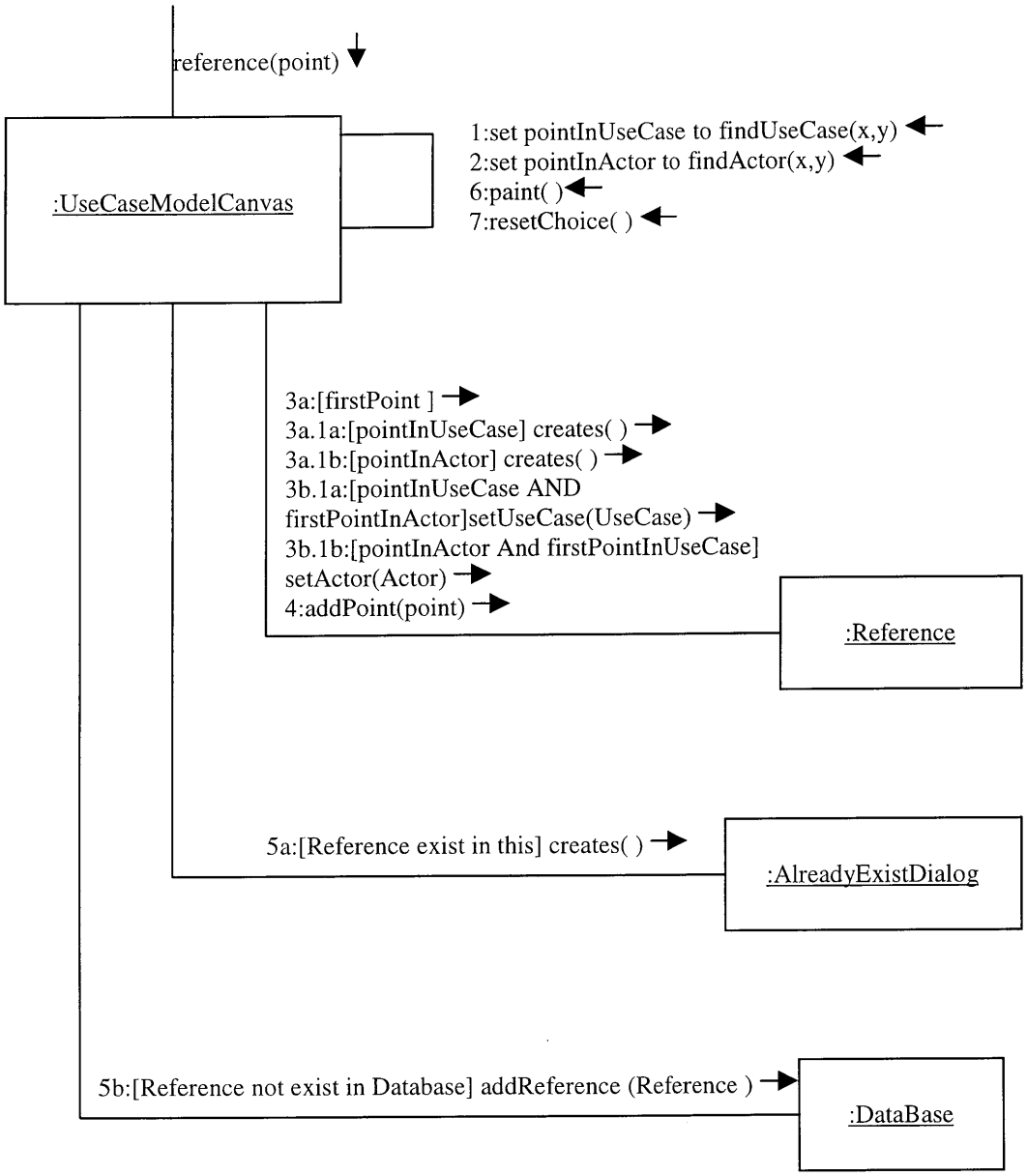


Figure C-7 reference( ) Collaboration Diagram

C.9 deleteReference( ) Collaboration Diagram ( )

Figure C-9 shows deleteReference( ) Collaboration Diagram ( ).

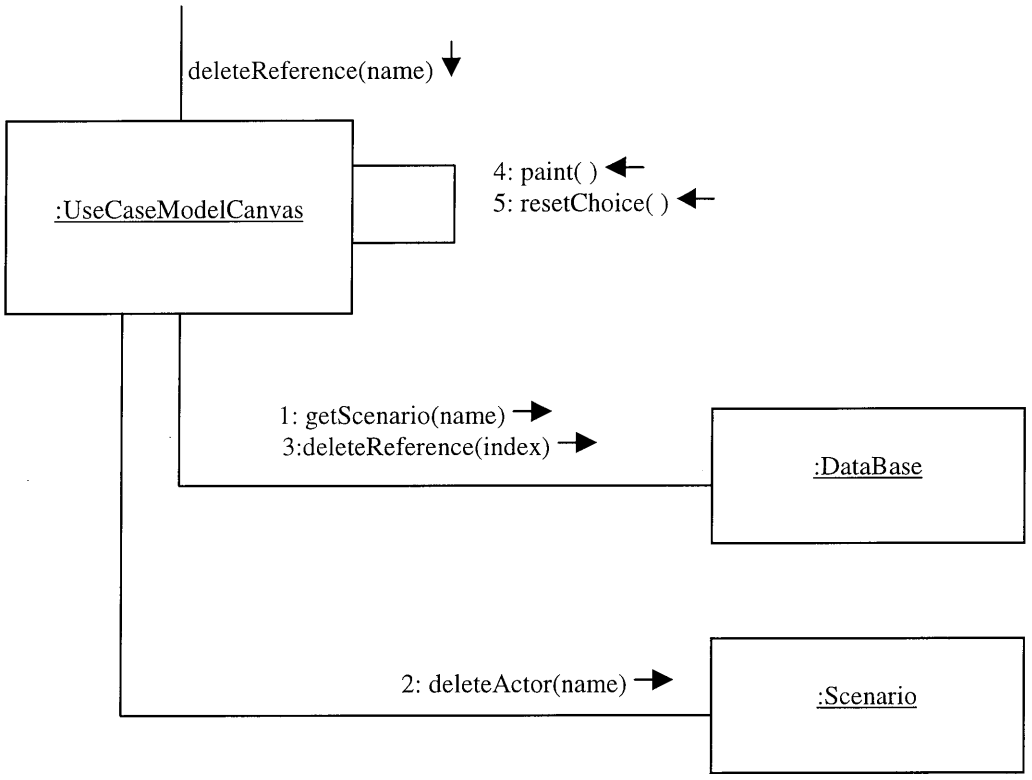


Figure C-9 deleteReference( ) Collaboration Diagram

C.10 UseCaseModelCanvas.mouseClicked( ) Collaboration Diagram

Figure C-10 shows UseCaseModelCanvas.mouseClicked( ) Collaboration Diagram.

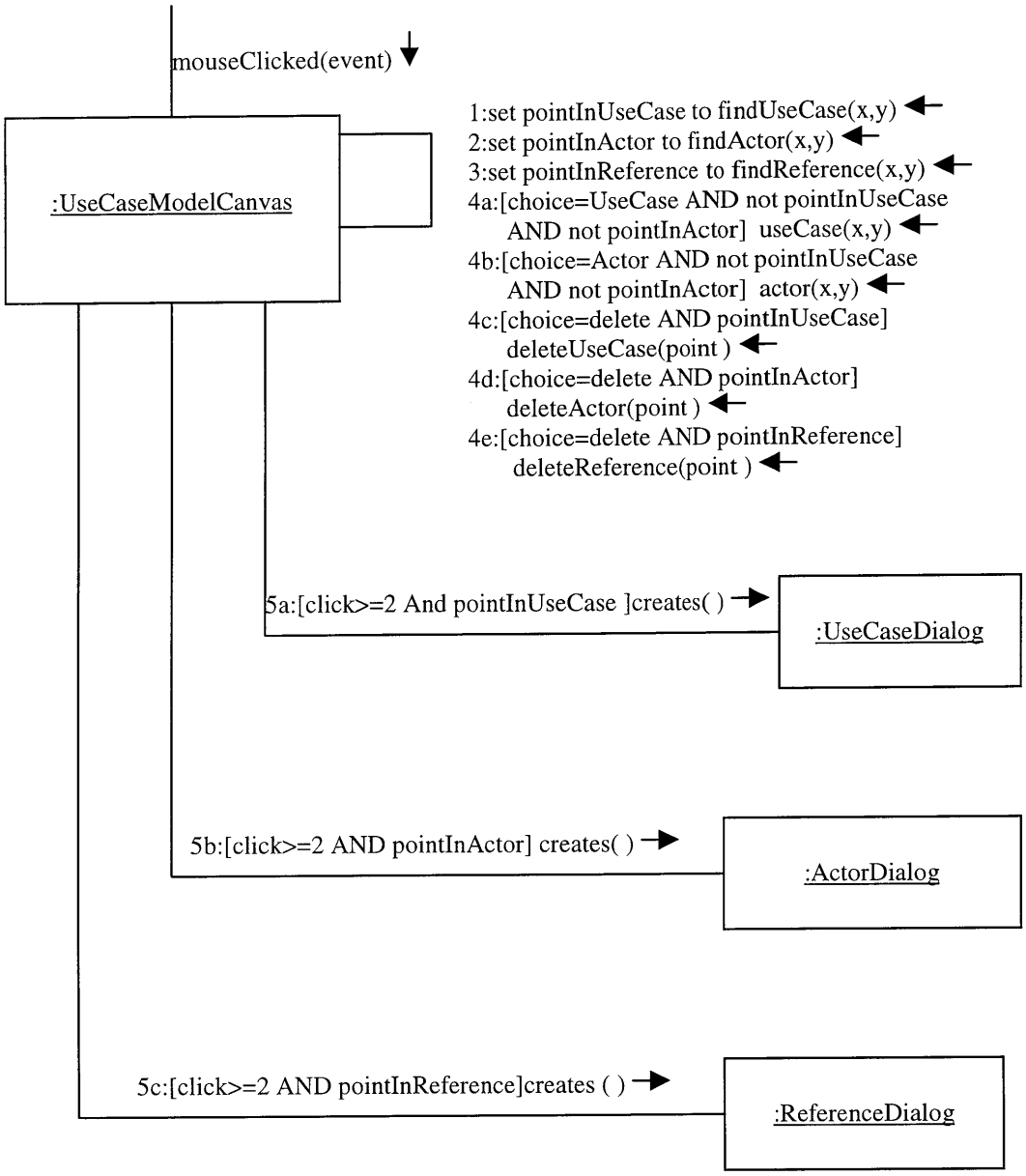


Figure C-10 UseCaseModelCanvas.mouseClicked( ) Collaboration Diagram

C.11 ClassDiagram.actionPerformed( ) Collaboration Diagram

Figure C-11 shows ClassDiagram.actionPerformed( ) Collaboration Diagram .

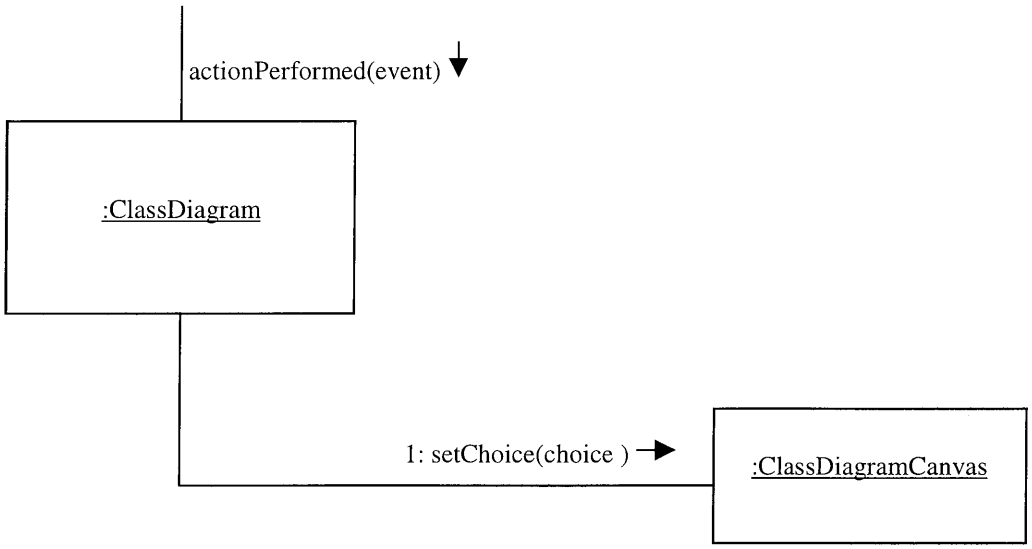


Figure C-11 ClassDiagram.actionPerformed( ) Collaboration Diagram

C.12 CreateClass Collaboration Diagram

Figure C-12 shows CreateClass Collaboration Diagram.

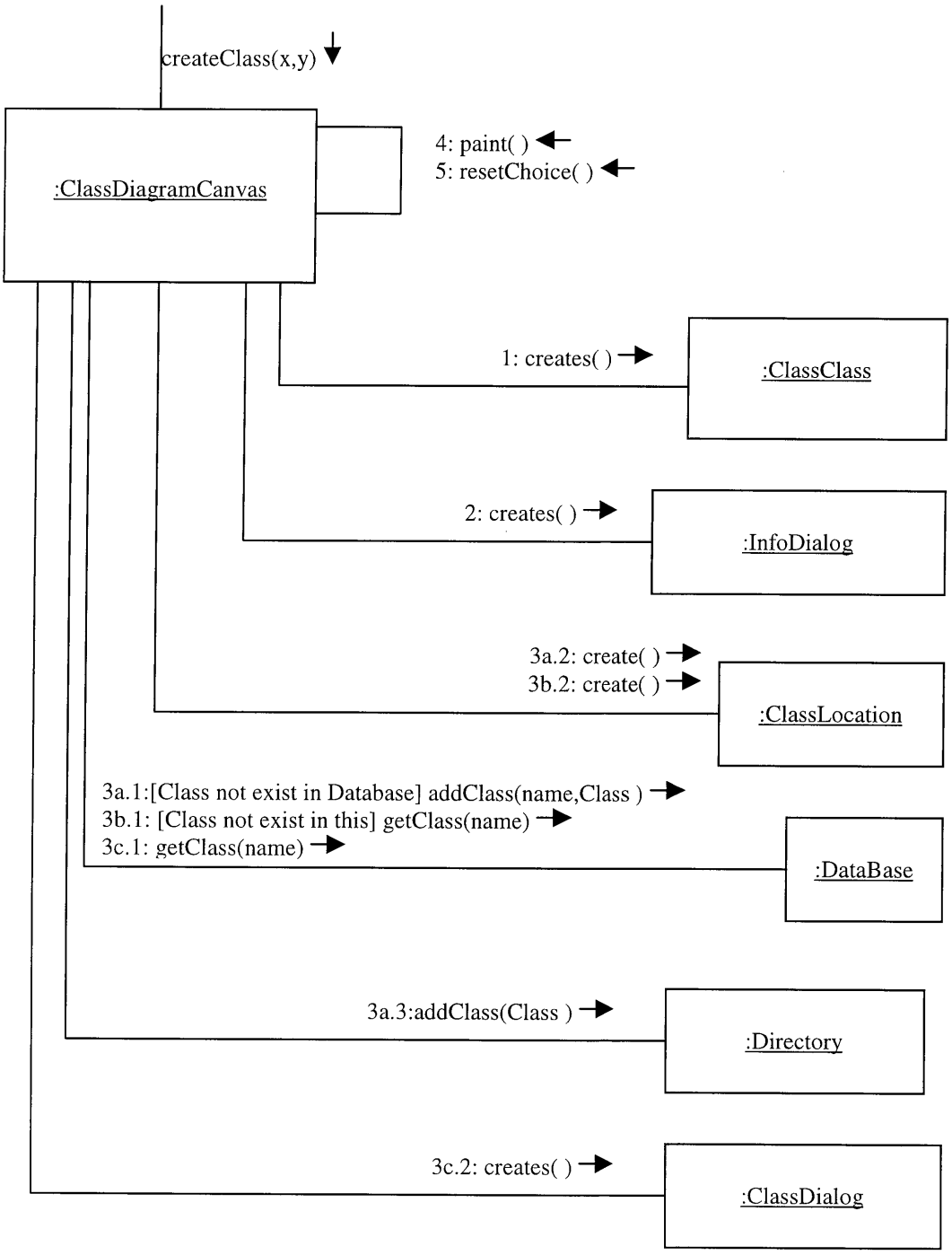


Figure C-12 createClass( ) Collaboration Diagram

C.13 Association( ) Collaboration Diagram

Figure C-13 shows Association( ) Collaboration Diagram.

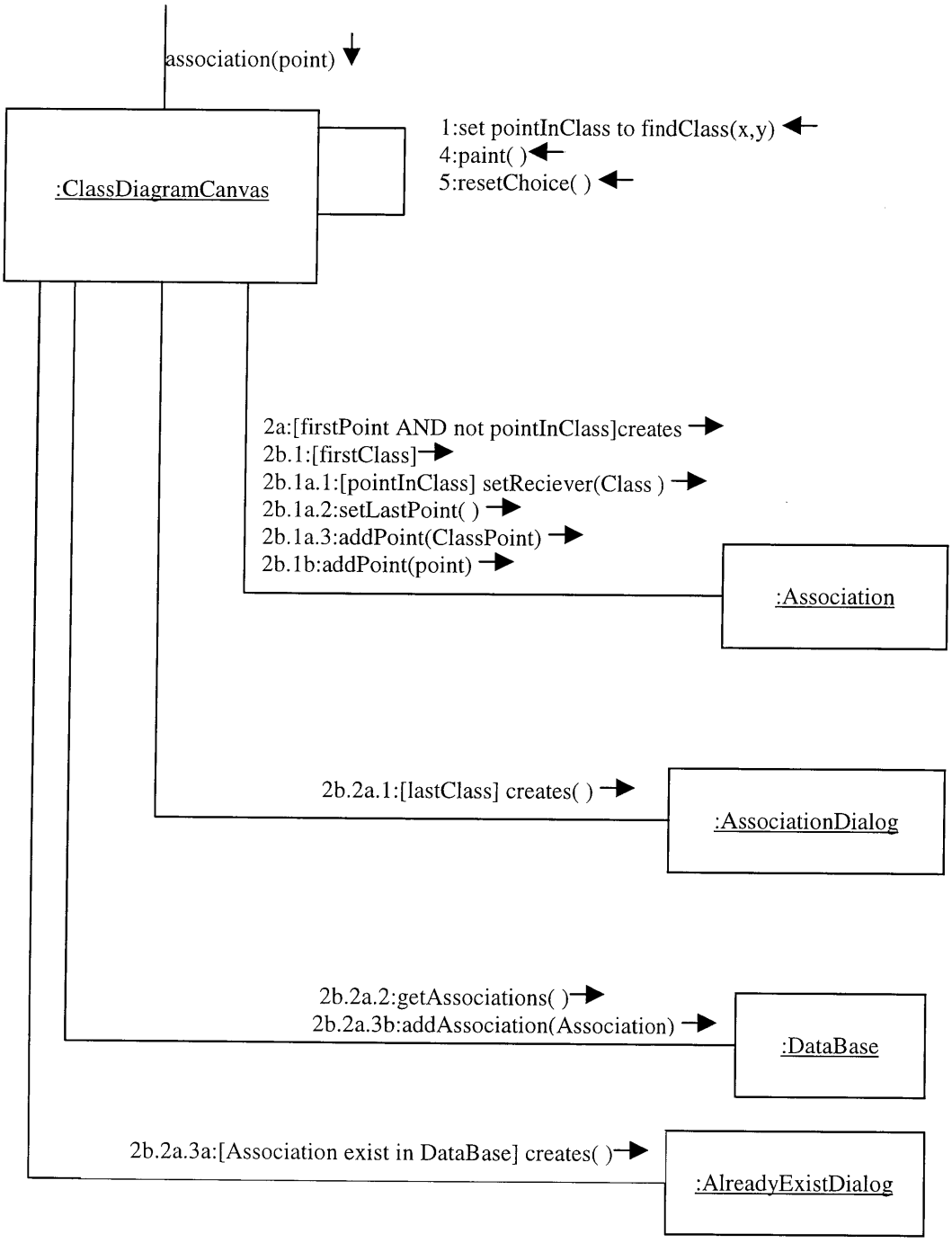


Figure C-13 association( ) Collaboration Diagram

C.14 Inheritance( ) Collaboration Diagram

Figure C-14 shows Inheritance( ) Collaboration Diagram.

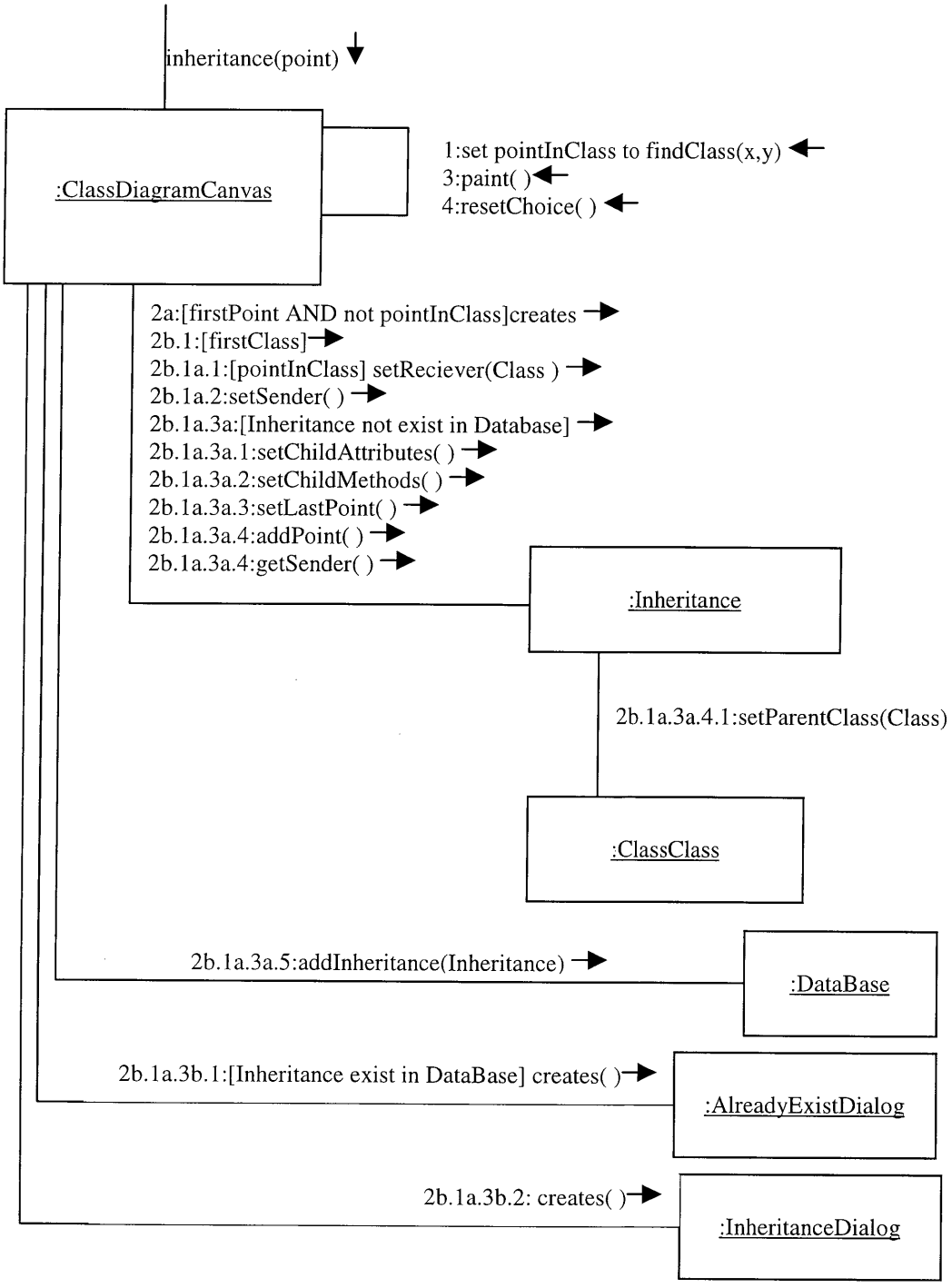


Figure C-14 association( ) Collaboration Diagram

C.15 Aggregation( ) Collaboration Diagram

Figure C-15 shows Aggregation( ) Collaboration Diagram.

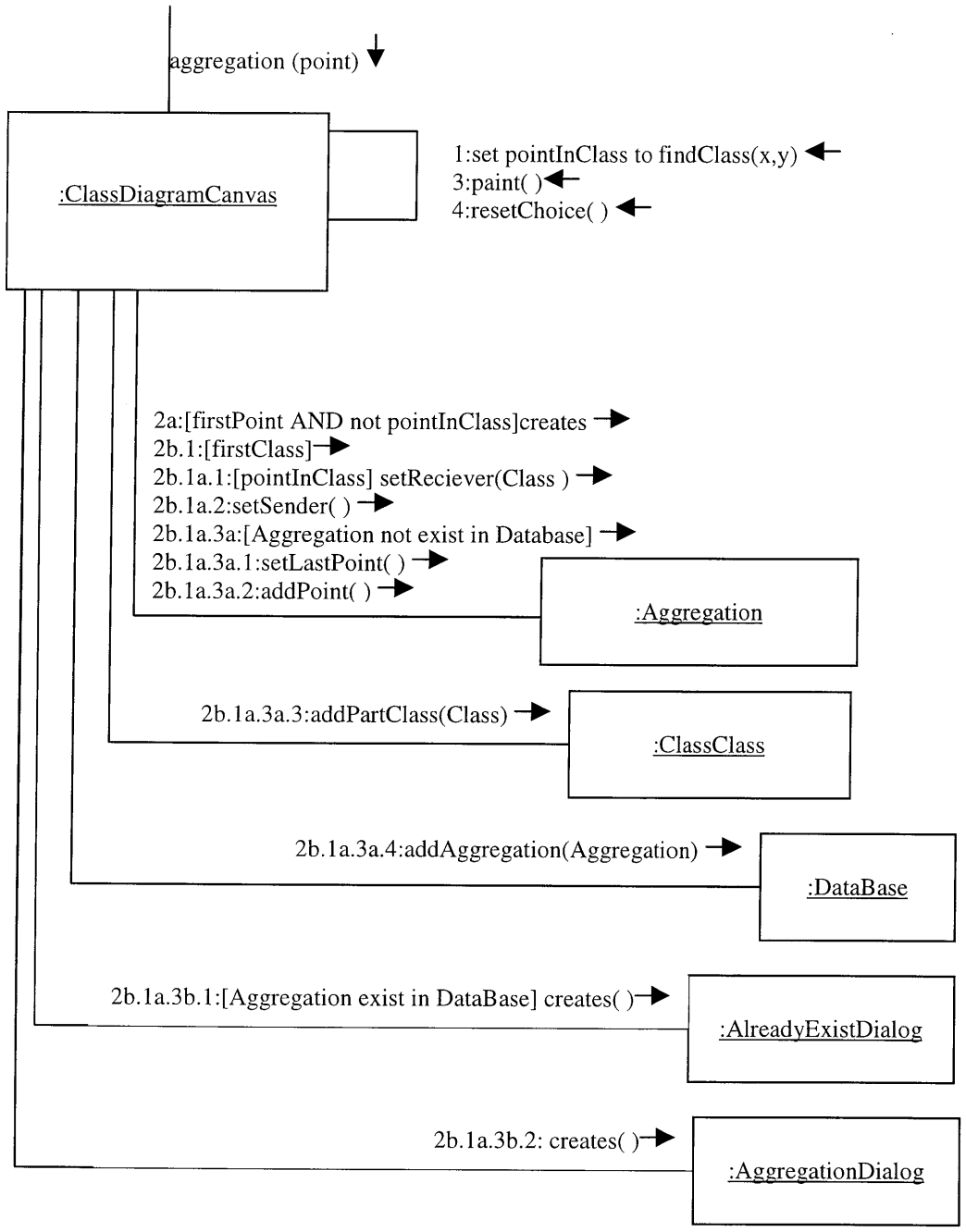


Figure C-15 aggregation( ) Collaboration Diagram

C.16 deleteClass( ) Collaboration Diagram

Figure C-16 shows deleteClass( ) Collaboration Diagram.

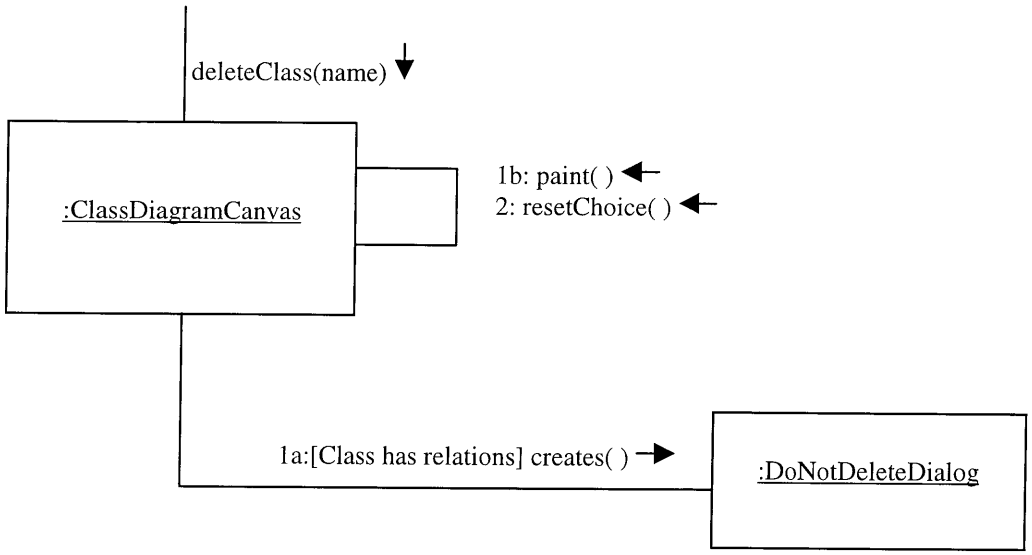


Figure C-16 deleteClass( ) Collaboration Diagram

C.17 deleteAssociation( ) Collaboration Diagram

Figure C-17 shows deleteAssociation( ) Collaboration Diagram.

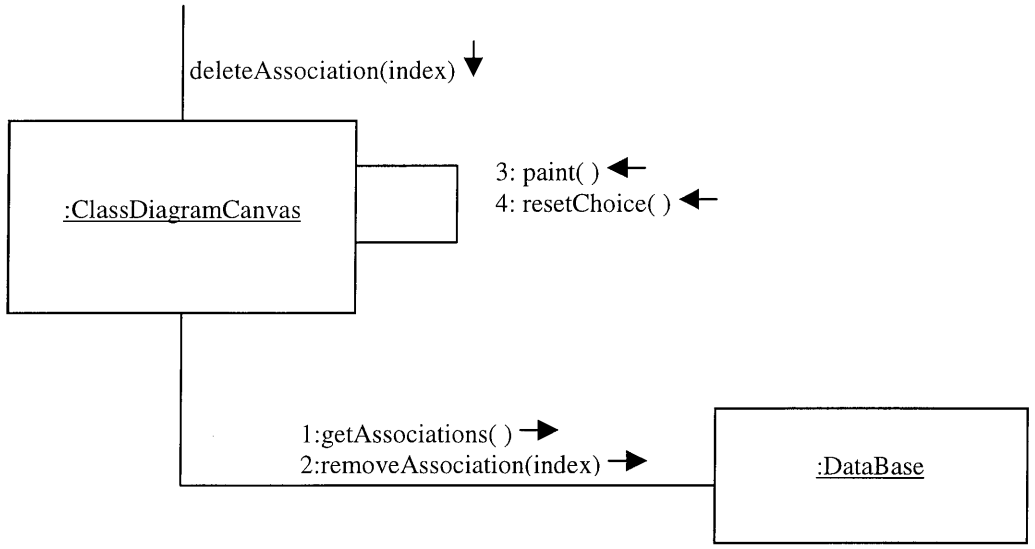


Figure C-17 deleteAssociation( ) Collaboration Diagram

C.18 deleteInheritance( ) Collaboration Diagram

Figure C-18 shows deleteInheritance( ) Collaboration Diagram.

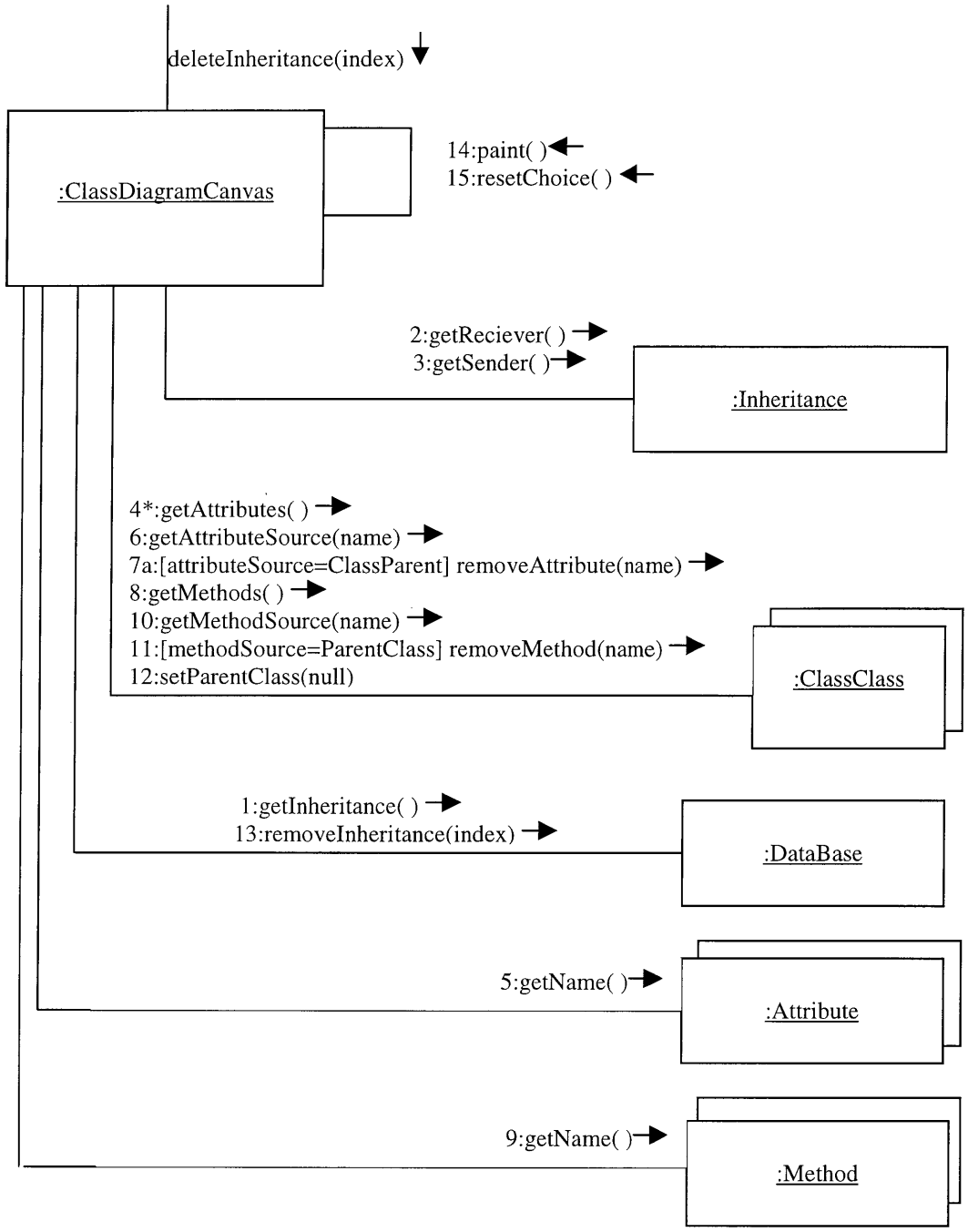


Figure C-18 deleteInheritance( ) Collaboration Diagram

C.19 deleteAggregation( ) Collaboration Diagram

Figure C-19 shows deleteAggregation( ) Collaboration Diagram.

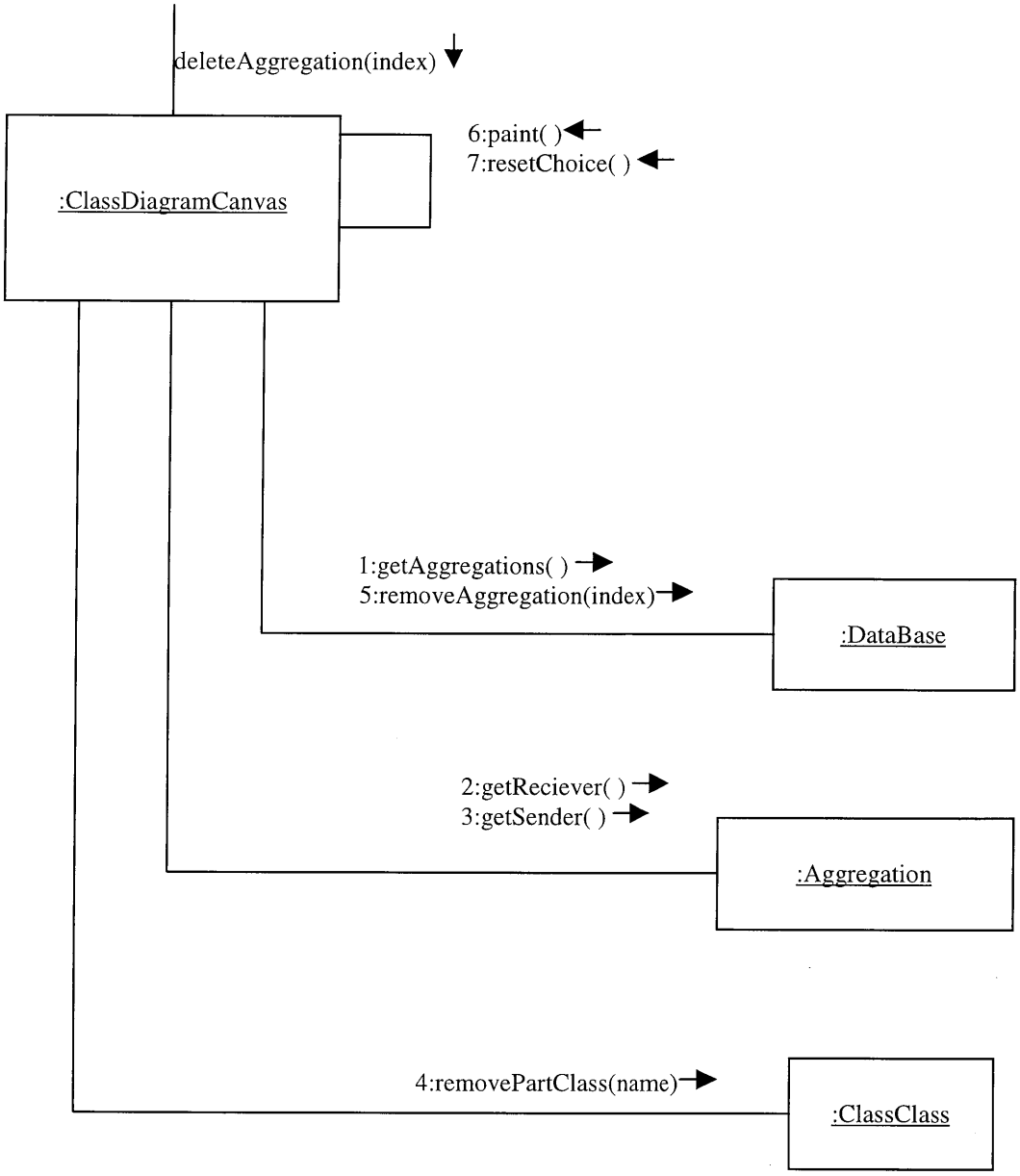


Figure C-19 deleteAggregation( ) Collaboration Diagram

C.20 ClassDiagramCanvas.mouseClicked( ) Collaboration Diagram

Figure C-20 shows ClassDiagramCanvas.mouseClicked( ) Collaboration Diagram.

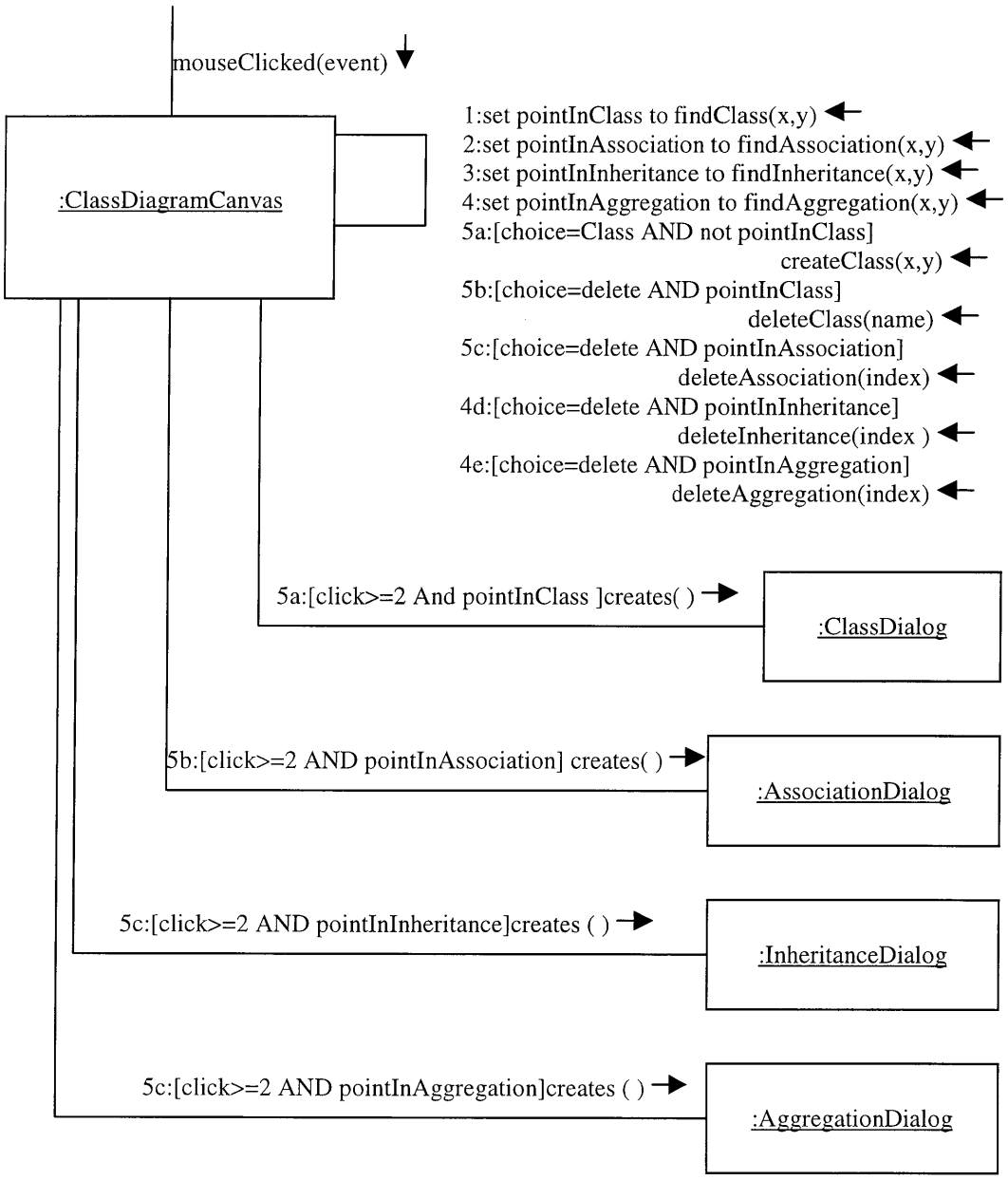


Figure C-20 ClassDiagramCanvas.mouseClicked( ) Collaboration Diagram

C.21 findClass( ) Collaboration Diagram

Figure C-21 shows findClass( ) Collaboration Diagram.

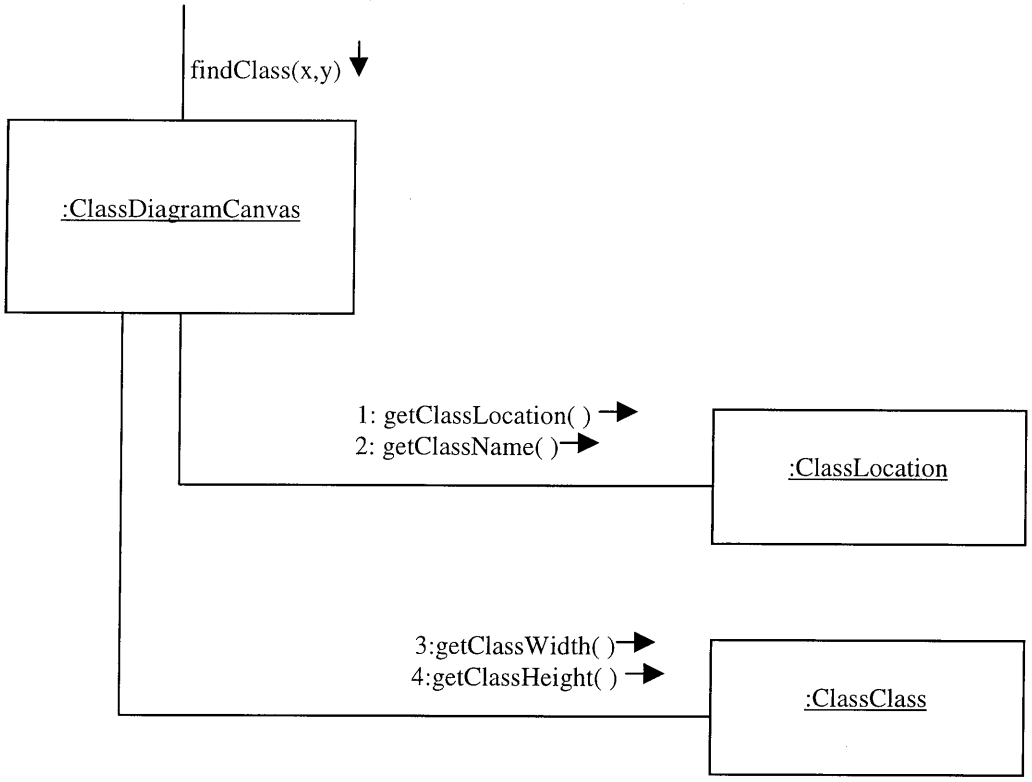


Figure C-21 findClass( ) Collaboration Diagram

C.22 findAssociation( ) Collaboration Diagram

Figure C-22 shows findAssociation( ) Collaboration Diagram.

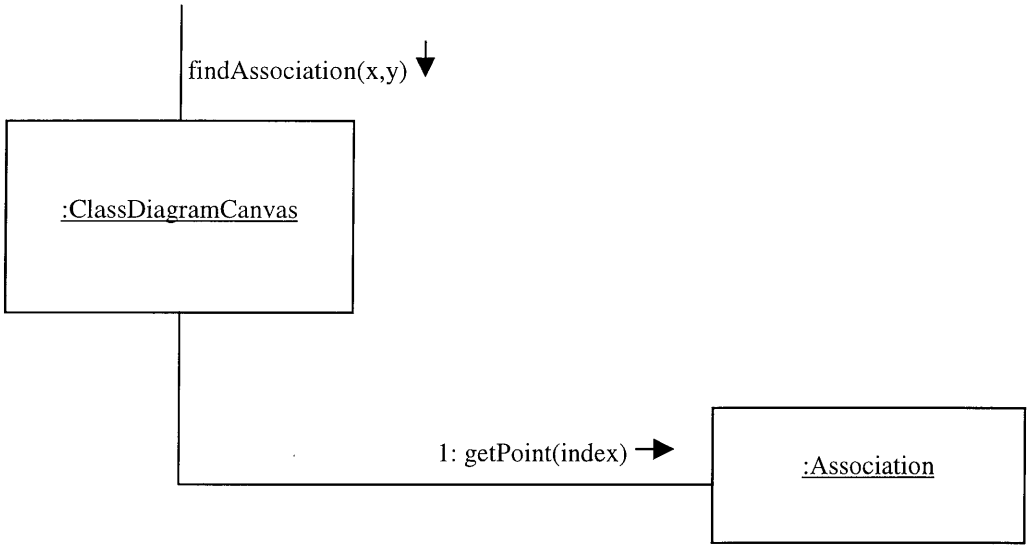


Figure C-22 findAssociation( ) Collaboration Diagram

C.23 findInheritance( ) Collaboration Diagram

Figure C-23 shows findInheritance( ) Collaboration Diagram .

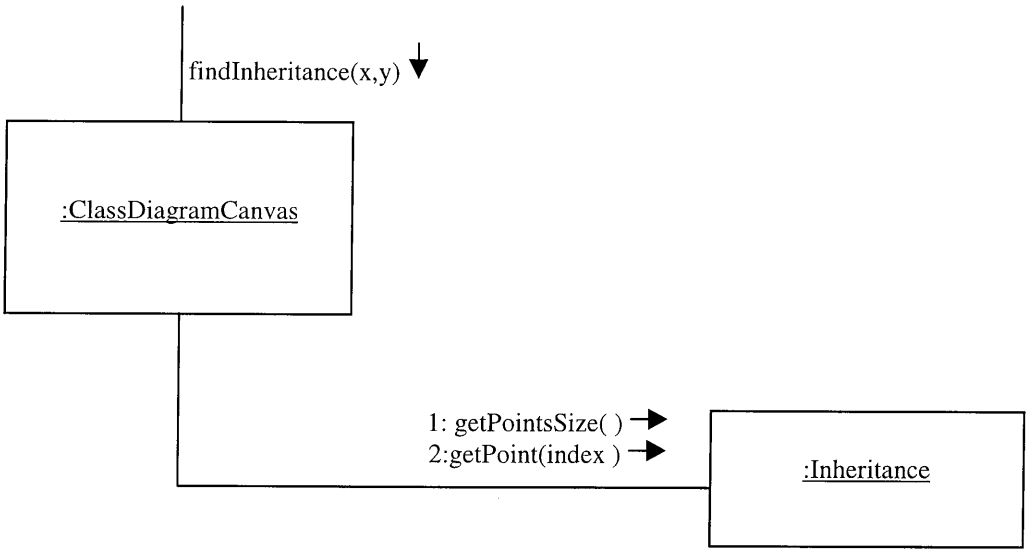


Figure C-23 findInheritance( ) Collaboration Diagram

C.24 findAggregation( ) Collaboration Diagram

Figure C-24 shows findAggregation( ) Collaboration Diagram .

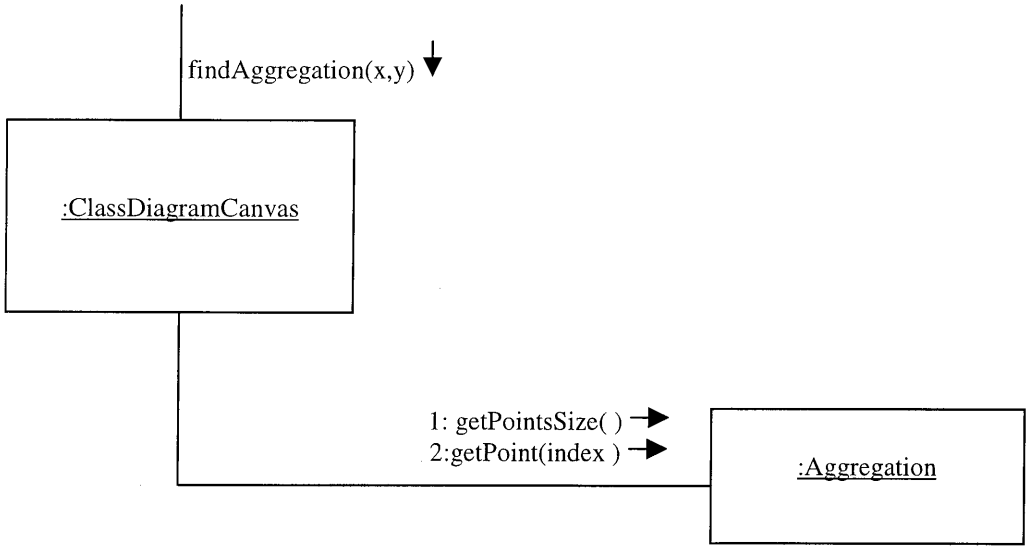


Figure C-24 findAggregation( ) Collaboration Diagram

C.25 Scenario.actionPerformed( ) Collaboration Diagram

Figure C-25 shows Scenario.actionPerformed( ) Collaboration Diagram.

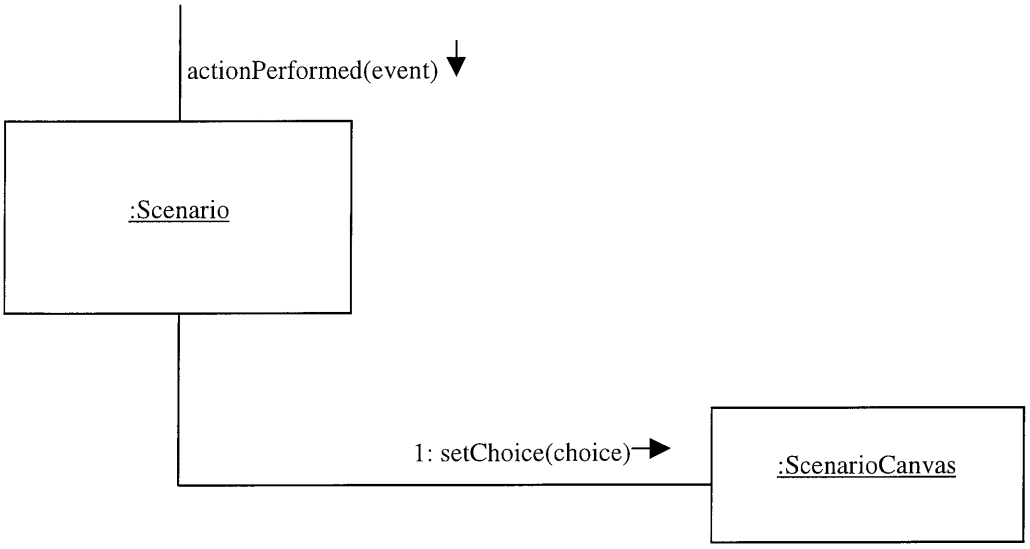


Figure C-25 Scenario.actionPerformed( ) Collaboration Diagram

C.26 scenarioActor( ) Collaboration Diagram

Figure C-26 shows scenarioActor( ) Collaboration Diagram.

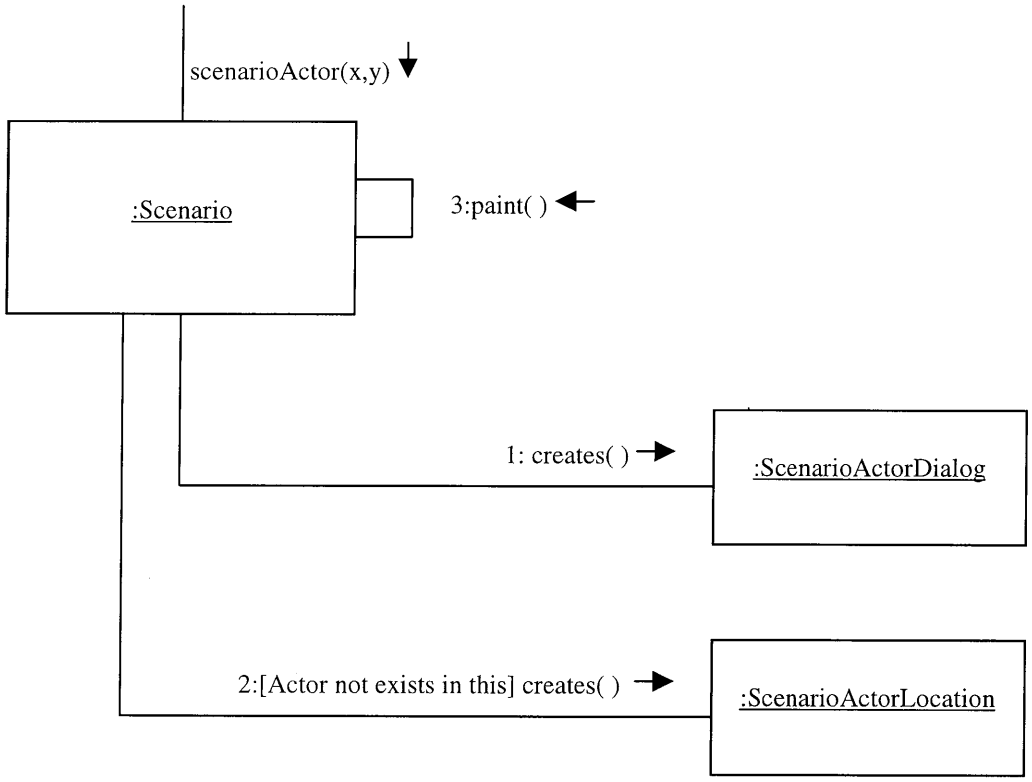


Figure C-26 scenarioActor( ) Collaboration Diagram

C.27 systemOperation( ) Collaboration Diagram

Figure C-27 shows systemOperation( ) Collaboration Diagram.

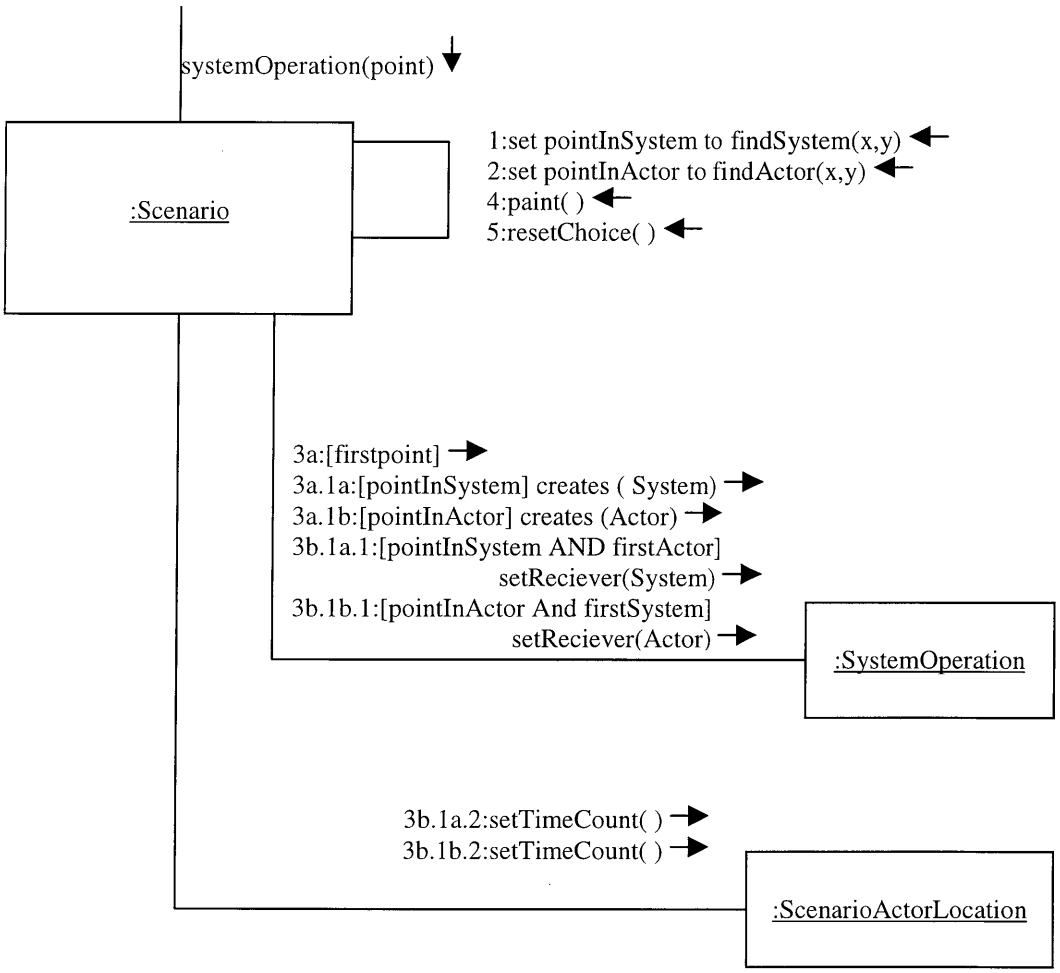


Figure C-27 systemOperation( ) Collaboration Diagram

C.28 deleteSystem( ) Collaboration Diagram

Figure C-28 shows deleteSystem( ) Collaboration Diagram.

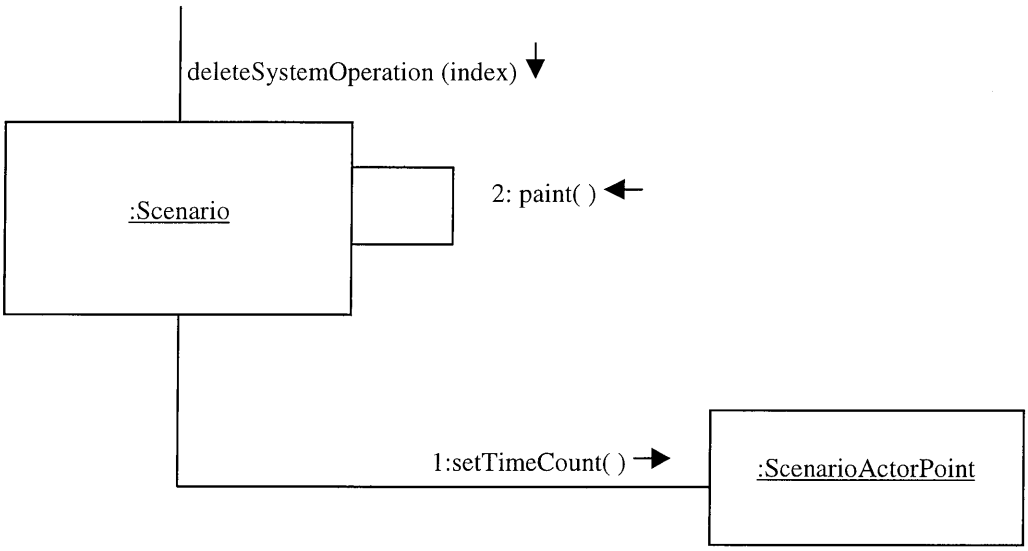


Figure C-28 deleteSystemOperation( ) Collaboration Diagram

C.29 findSystem Collaboration Diagram

Figure C-29 shows findSystem Collaboration Diagram.

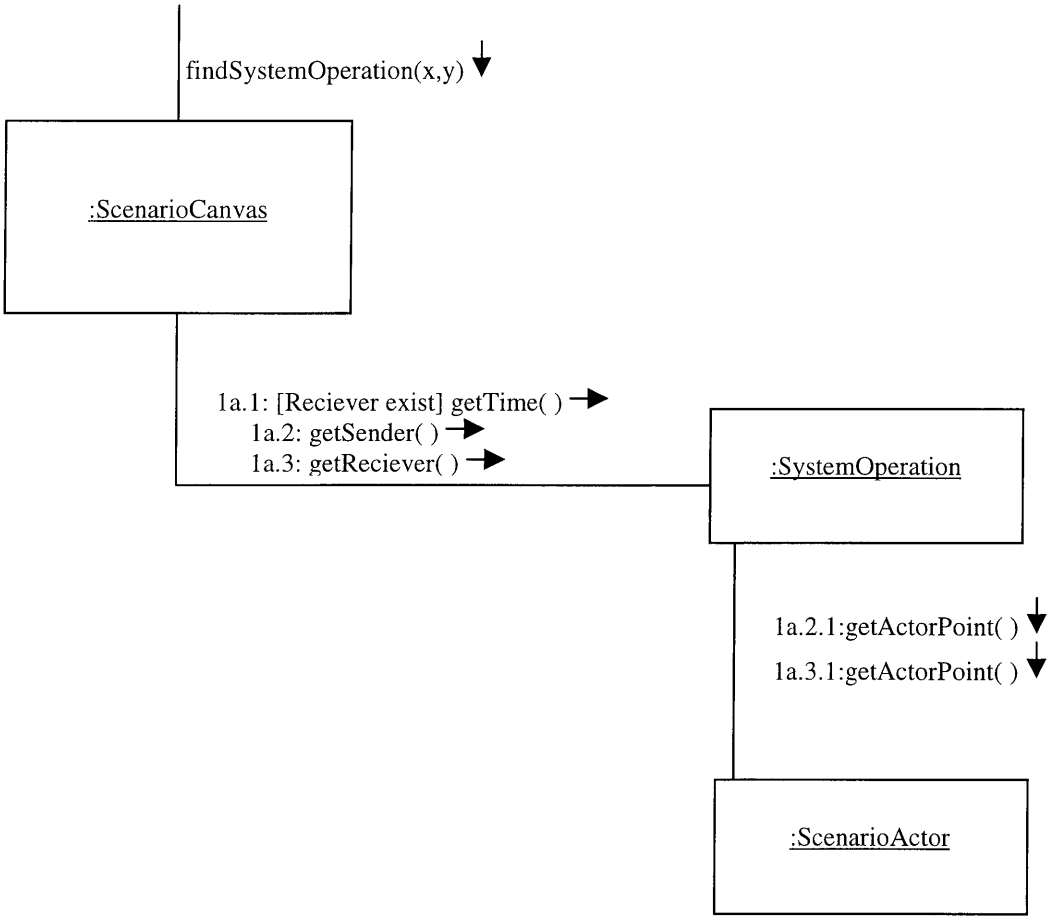


Figure C-29 findSystemOperation( ) Collaboration Diagram

C.30 findActor( ) Collaboration Diagram

Figure C-30 shows findActor( ) Collaboration Diagram.

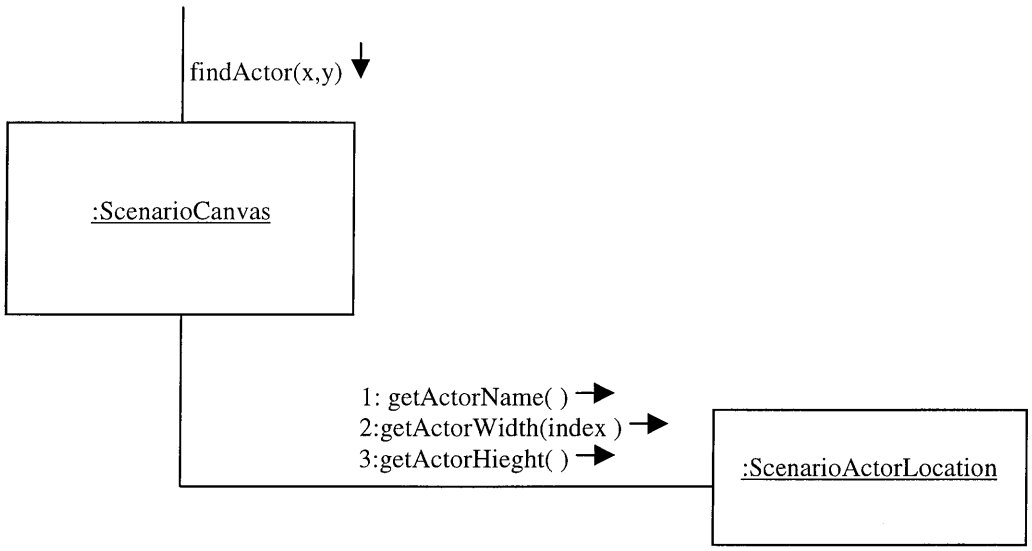


Figure C-30 findActor( ) Collaboration Diagram

C.31 ClassDirectory.itemStateChanged( ) Collaboration Diagram

Figure C-31 shows ClassDirectory.itemStateChanged( ) Collaboration Diagram.

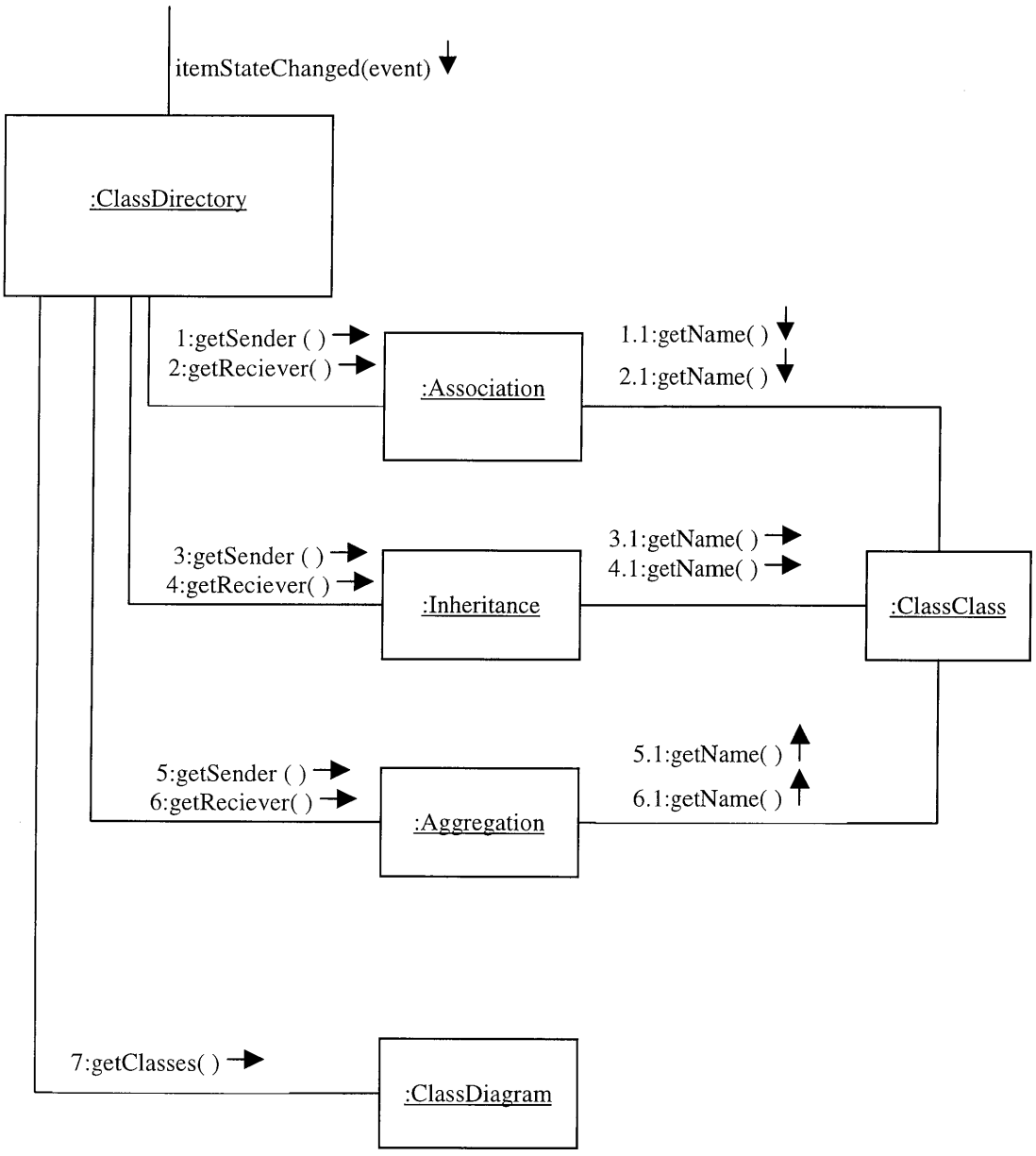


Figure C-31 ClassDirectory.itemStateChanged( ) Collaboration Diagram

C.32 ClassDirectory.actionPerformed( ) Collaboration Diagram

Figure C-32 shows ClassDirectory.actionPerformed( ) Collaboration Diagram.

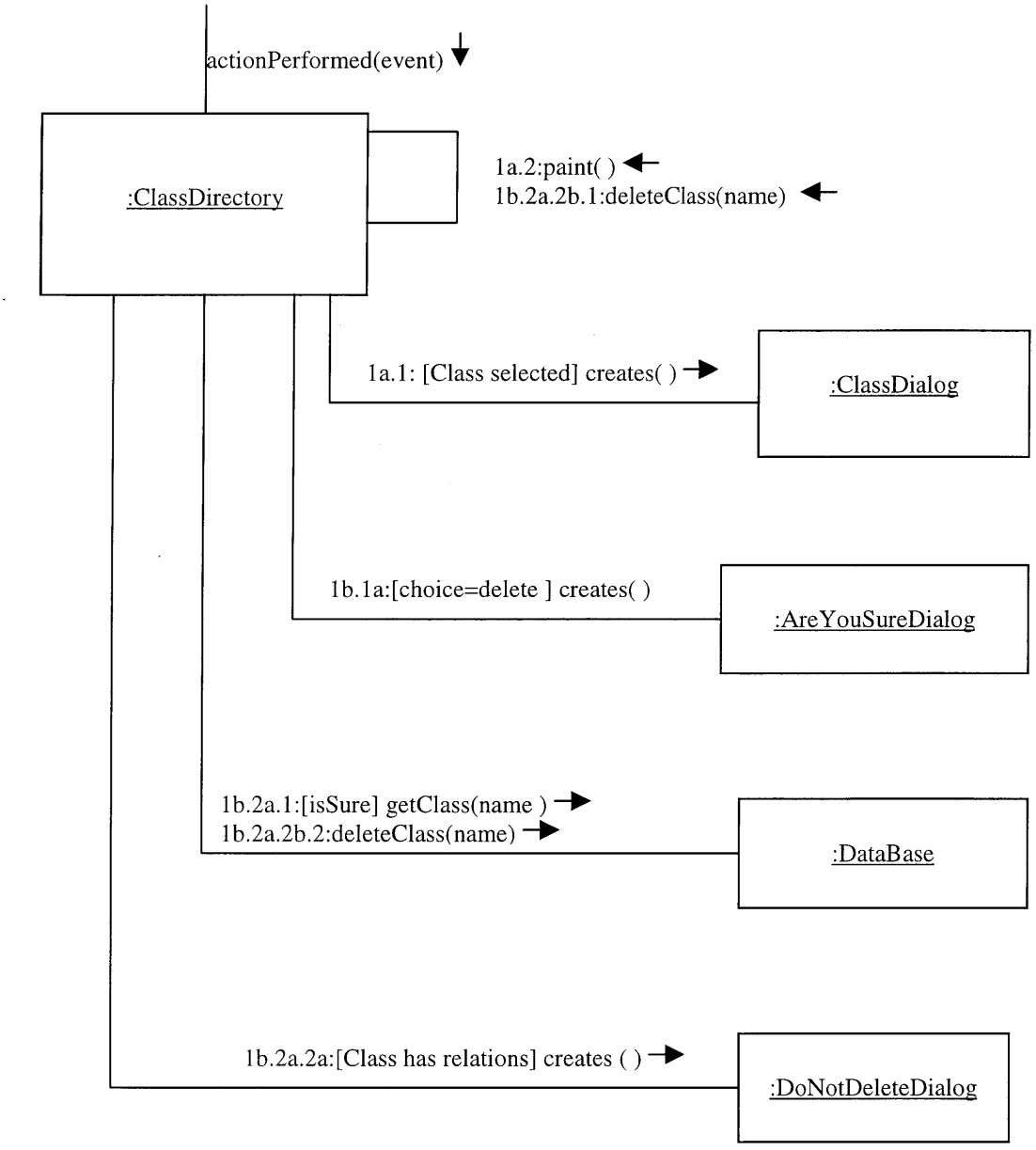


Figure C-32 ClassDirectory.action_Performed( ) Collaboration Diagram

C.33 UseCaseDirectory.itemStateChanged( ) Collaboration Diagram

Figure C-33 shows UseCaseDirectory.itemStateChanged( ) Collaboration Diagram.

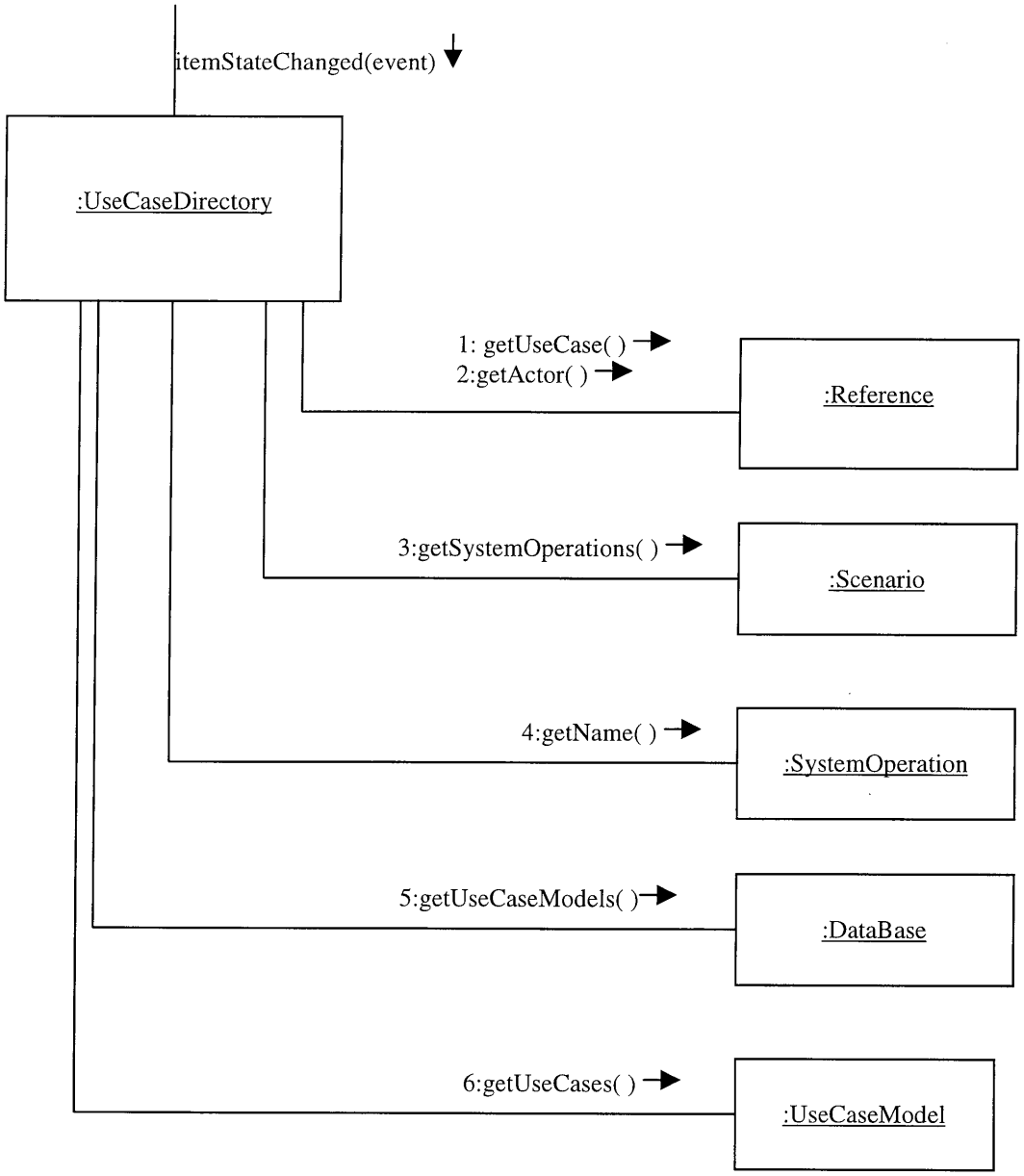


Figure C-33 UseCaseDirectory.itemStateChanged( ) Collaboration Diagram

C.34 UseCaseDirectory.actionPerformed( ) Collaboration Diagram

Figure C-34 shows UseCaseDirectory.actionPerformed( ) Collaboration Diagram.

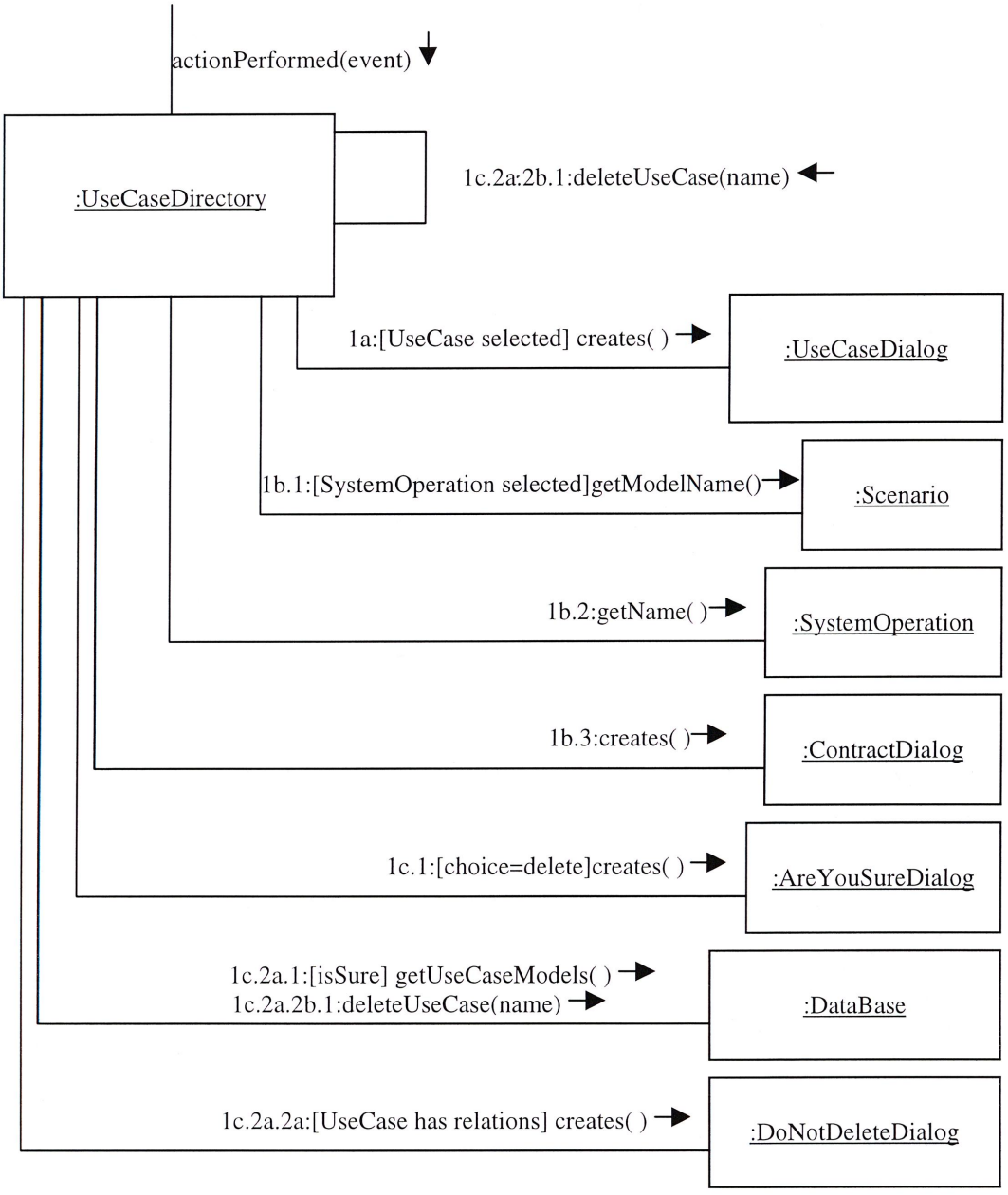


Figure C-34 UseCaseDirectory.actionPerformed( ) Collaboration Diagram

Appendix D: Design Class Diagrams

The Design Class Diagram looks like the Class Diagram except that an arrow represents the association relationship in the Design Class Diagram.

D.1 AllModels Design Class Diagram

The AllModels Design Class Diagram shows that AllModels class inherits from the CloseableFrame class, which inherits from the built-in class, Frame class, in Java. The association relationship between the AllModels class and other classes in the diagram make those classes accesseable to the AllModels class. Figure D-1 shows AllModels Design Class Diagram.

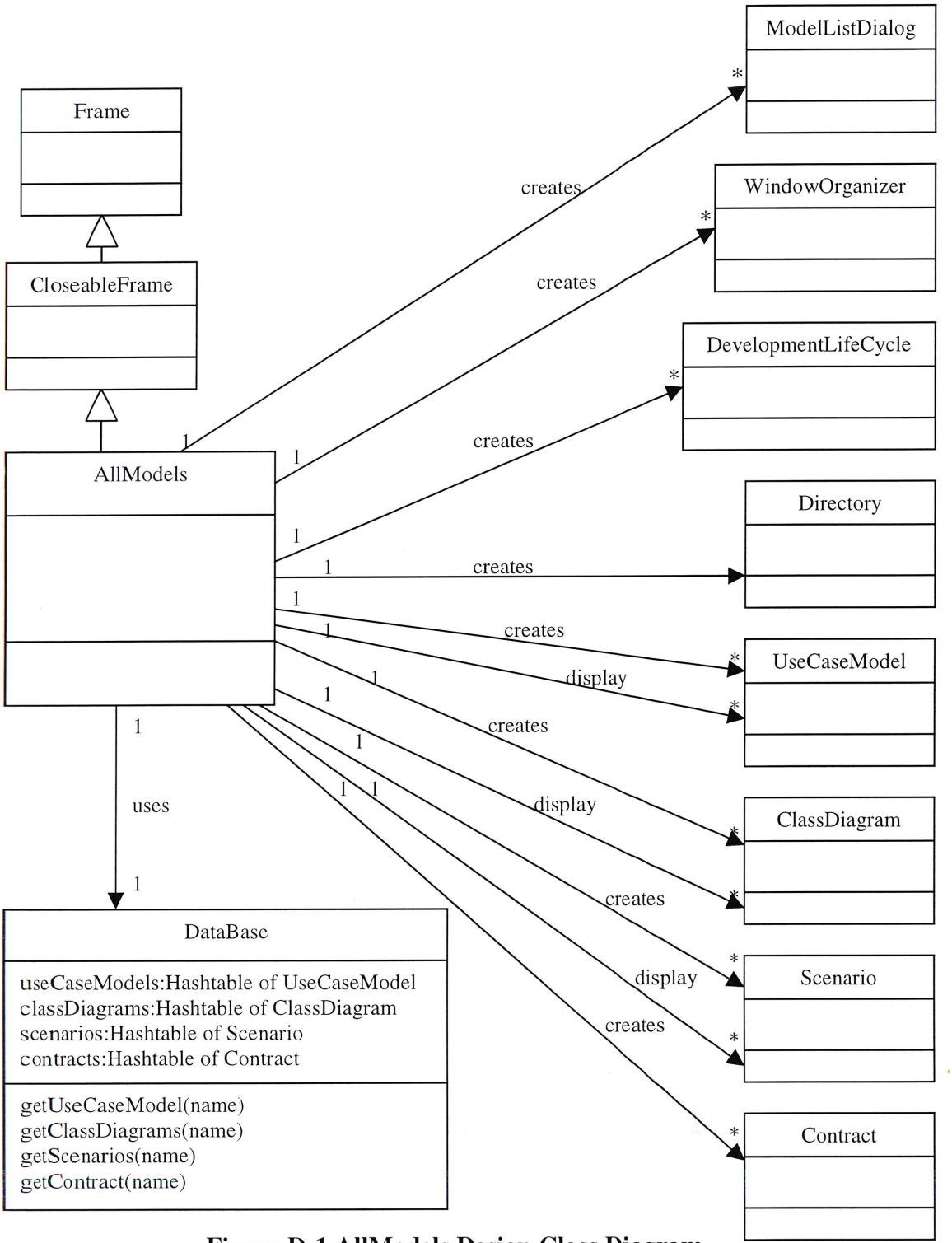


Figure D-1 AllModels Design Class Diagram

D.2 ClassDiagram Design Class Diagram

Figure D-2 shows ClassDiagram Design Class Diagram.

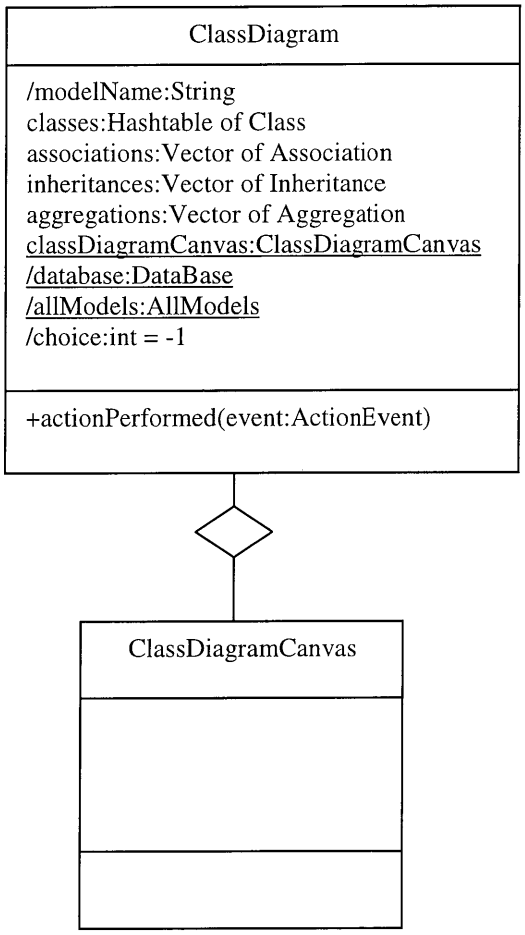


Figure D-2 ClassDiagram Design Class Diagram

D.3 ClassDiagramCanvas Design Class Diagram

Figure D-3 shows ClassDiagramCanvas Design Class Diagram.

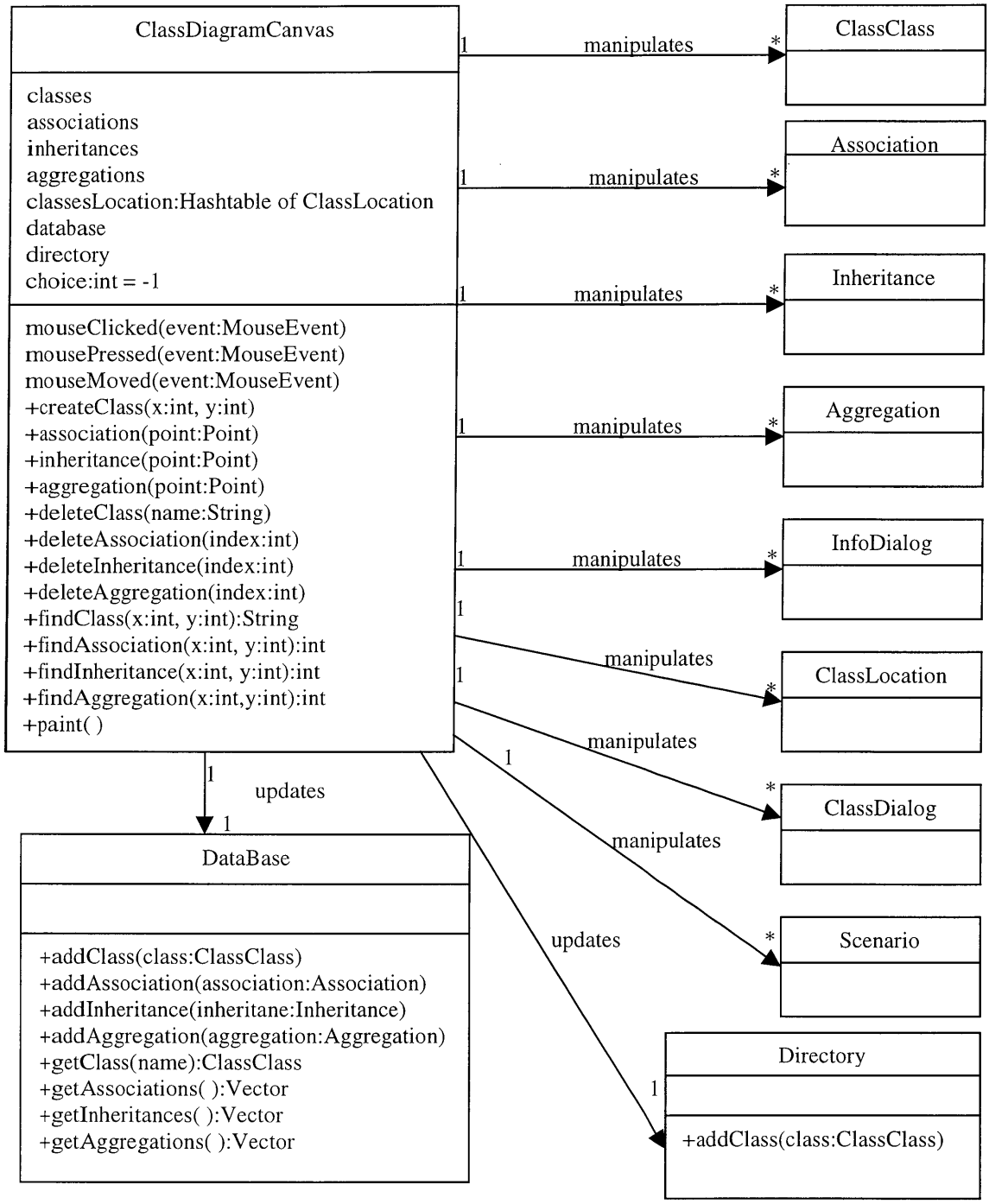


Figure D-3 ClassDiagramCanvas Design Class Diagram

D.4 ClassClass Design Class Diagram

Figure D-4 shows ClassClass Design Class Diagram.

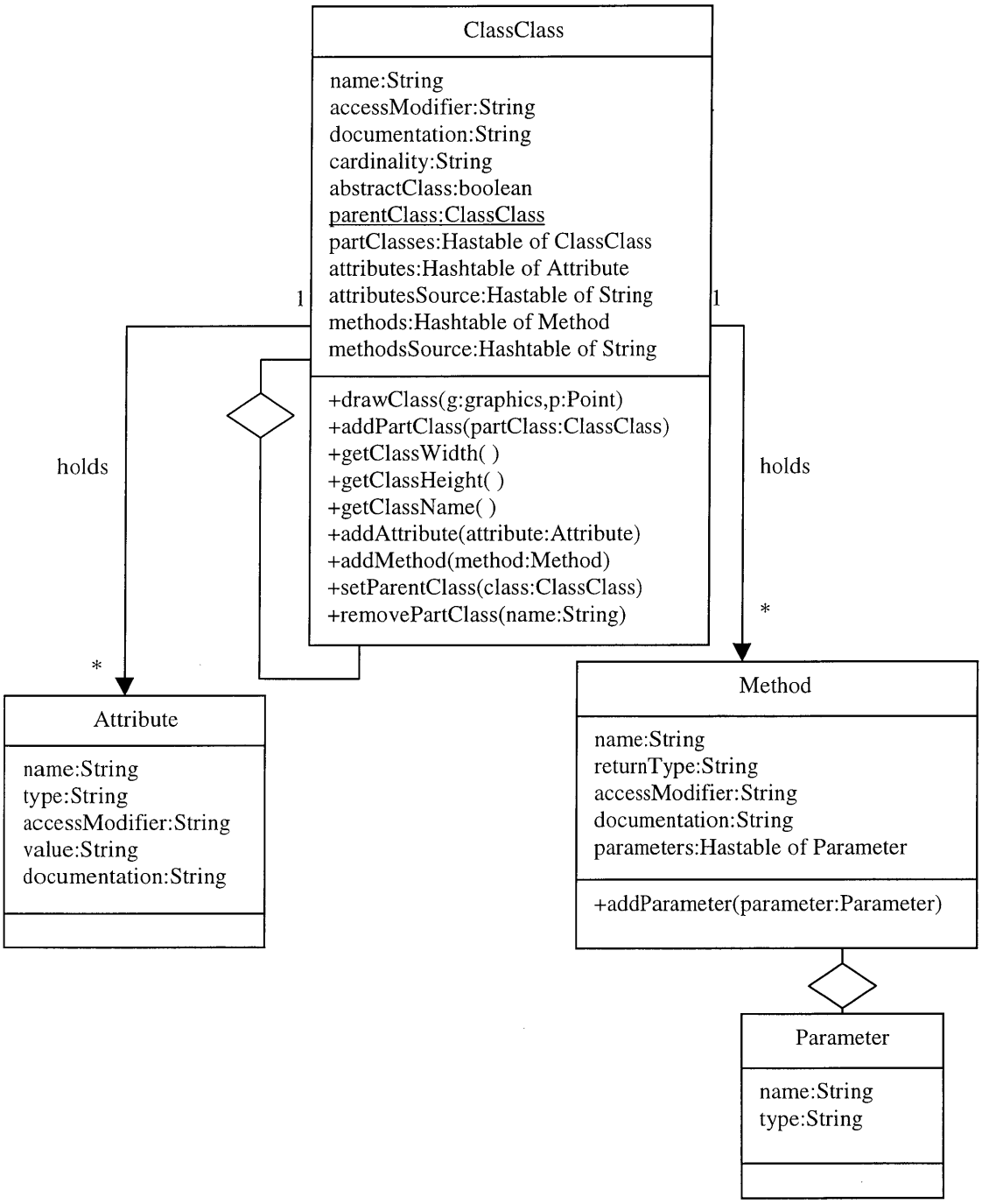


Figure D-4 ClassClass Design Class Diagram

D.5 Relationship Design Class Diagram

Figure D-5 shows Relationship Design Class Diagram.

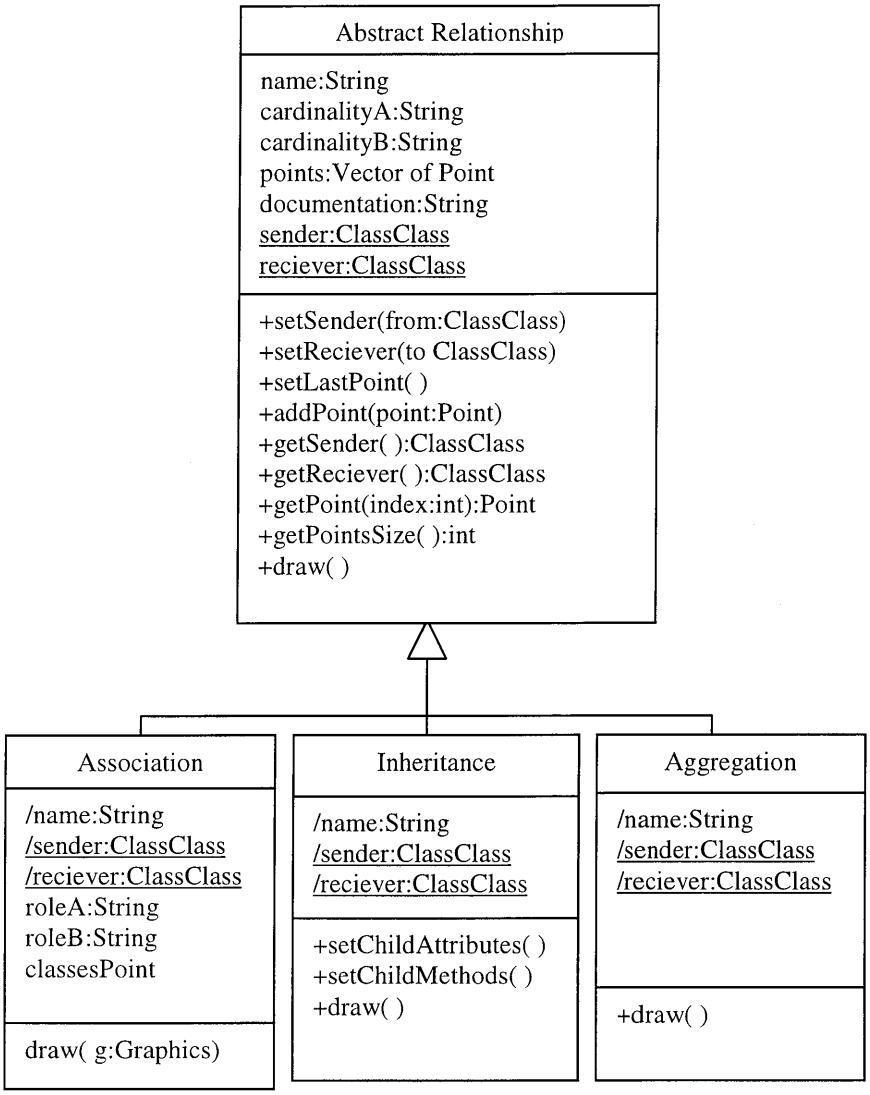


Figure D-5 Relationship Design Class Diagram

D.6 Model Design Class Diagram

Figure D-6 shows Model Design Class Diagram.

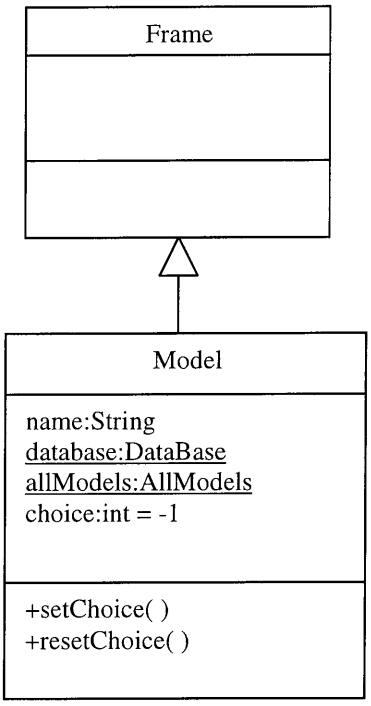


Figure D-6 Model Design Class Diagram

D.7 UseCaseModel Design Class diagram

Figure D-7 shows UseCaseModel Design Class diagram.

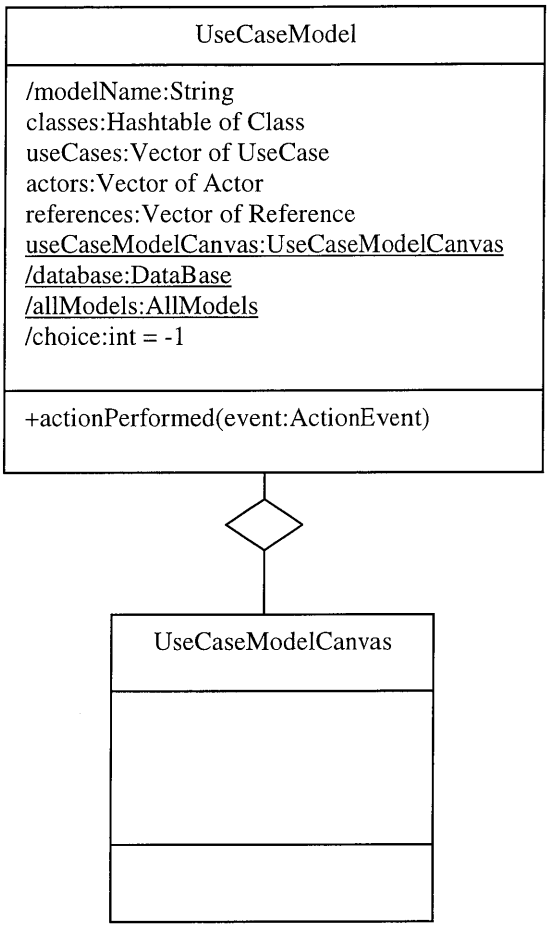


Figure D-7 UseCaseModel Design Class Diagram

D.8 UseCaseModelCanvas Design Class Diagram

Figure D-8 shows UseCaseModelCanvas Design Class Diagram.

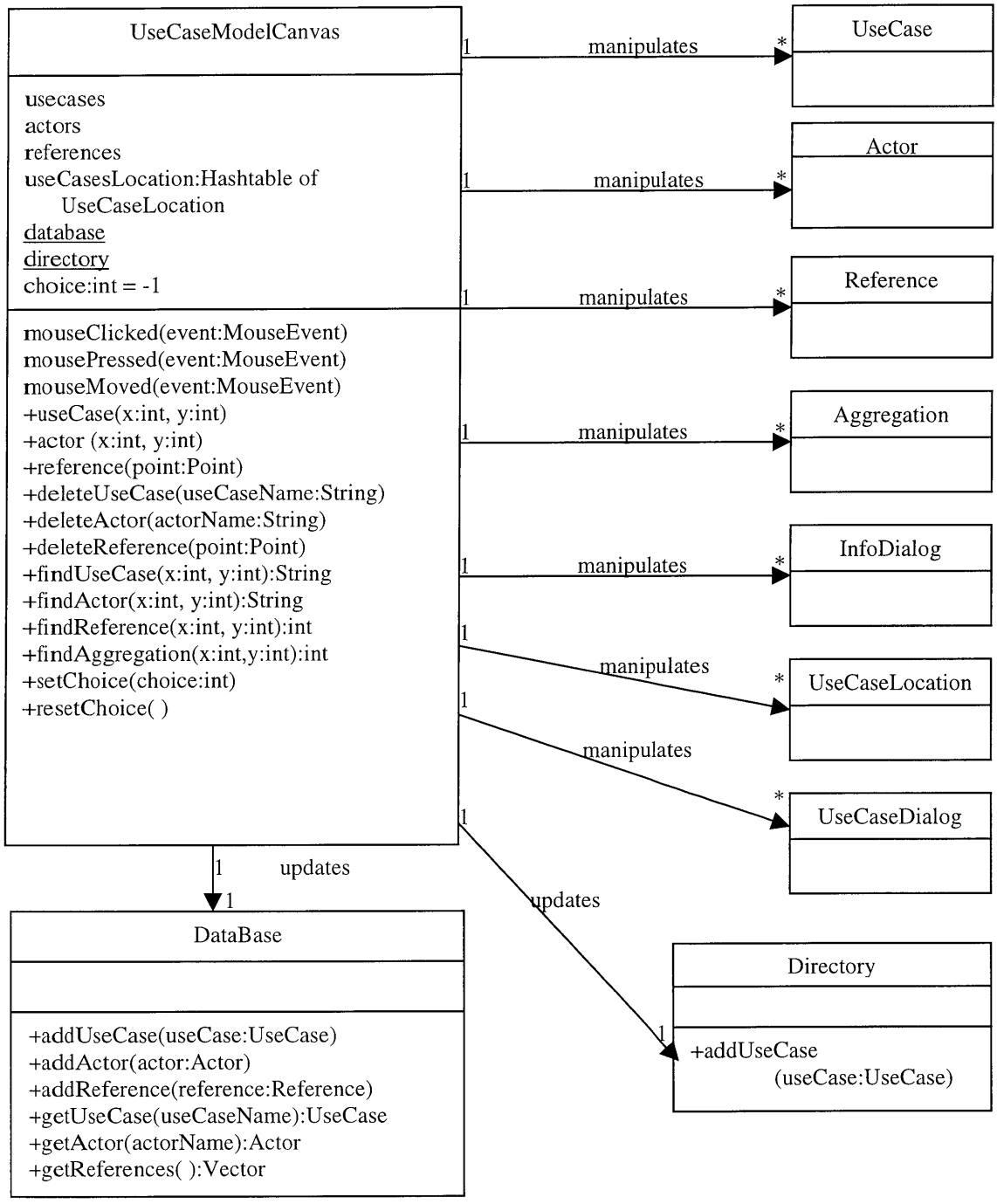


Figure D-8 UseCaseModelCanvas Design Class Diagram

D.9 UseCaseModelObjects Design Class Diagram

Figure D-9 shows UseCaseModelObjects Design Class Diagram.

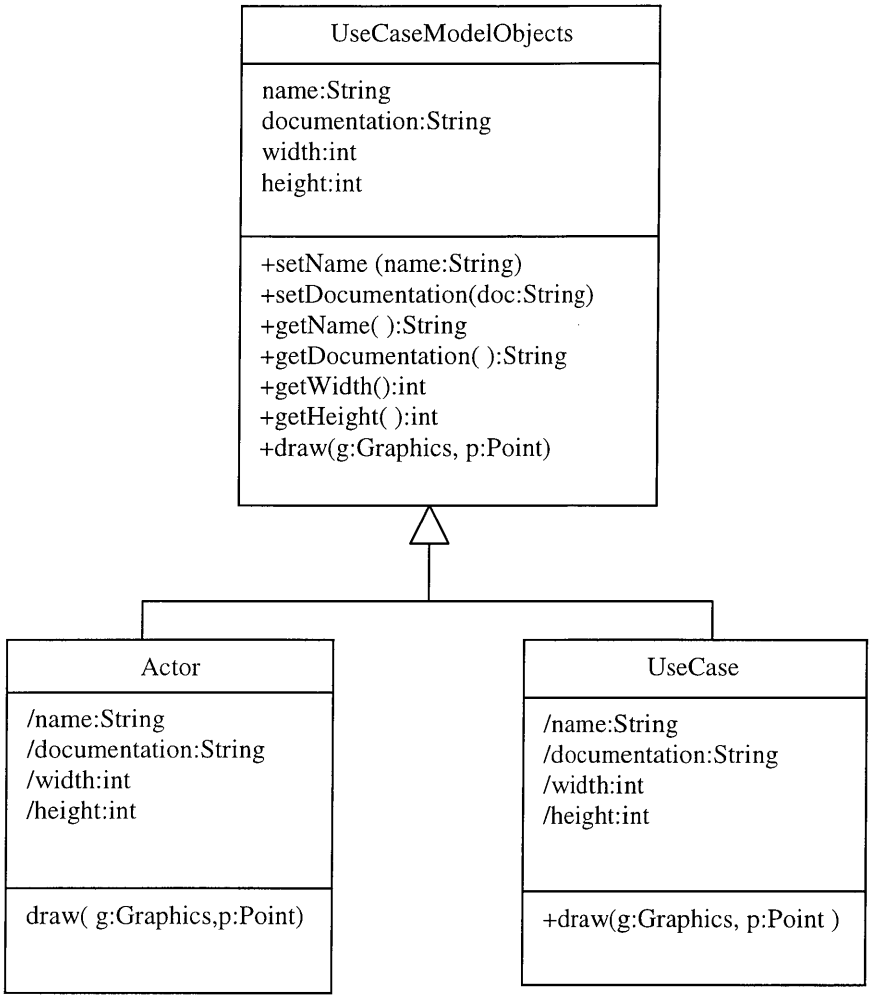


Figure D-9 UseCaseModelObjects Design Class Diagram

D.10 Scenario Design Class Diagram

Figure D-10 shows Scenario Design Class Diagram.

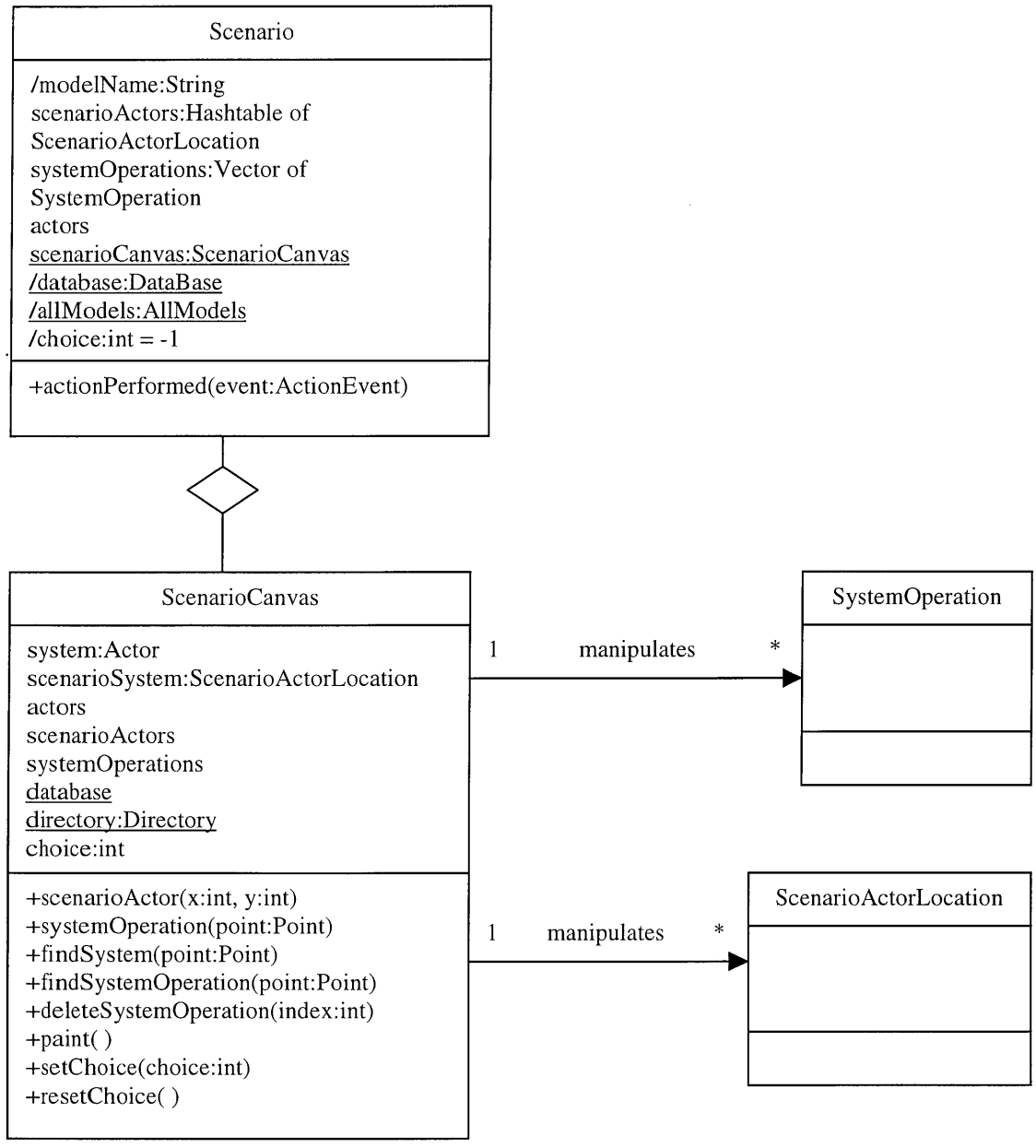


Figure D-10 Scenario Design Class Diagram

D.11 ClassDirectory Design Class Diagram

Figure D-11 shows ClassDirectory Design Class Diagram.

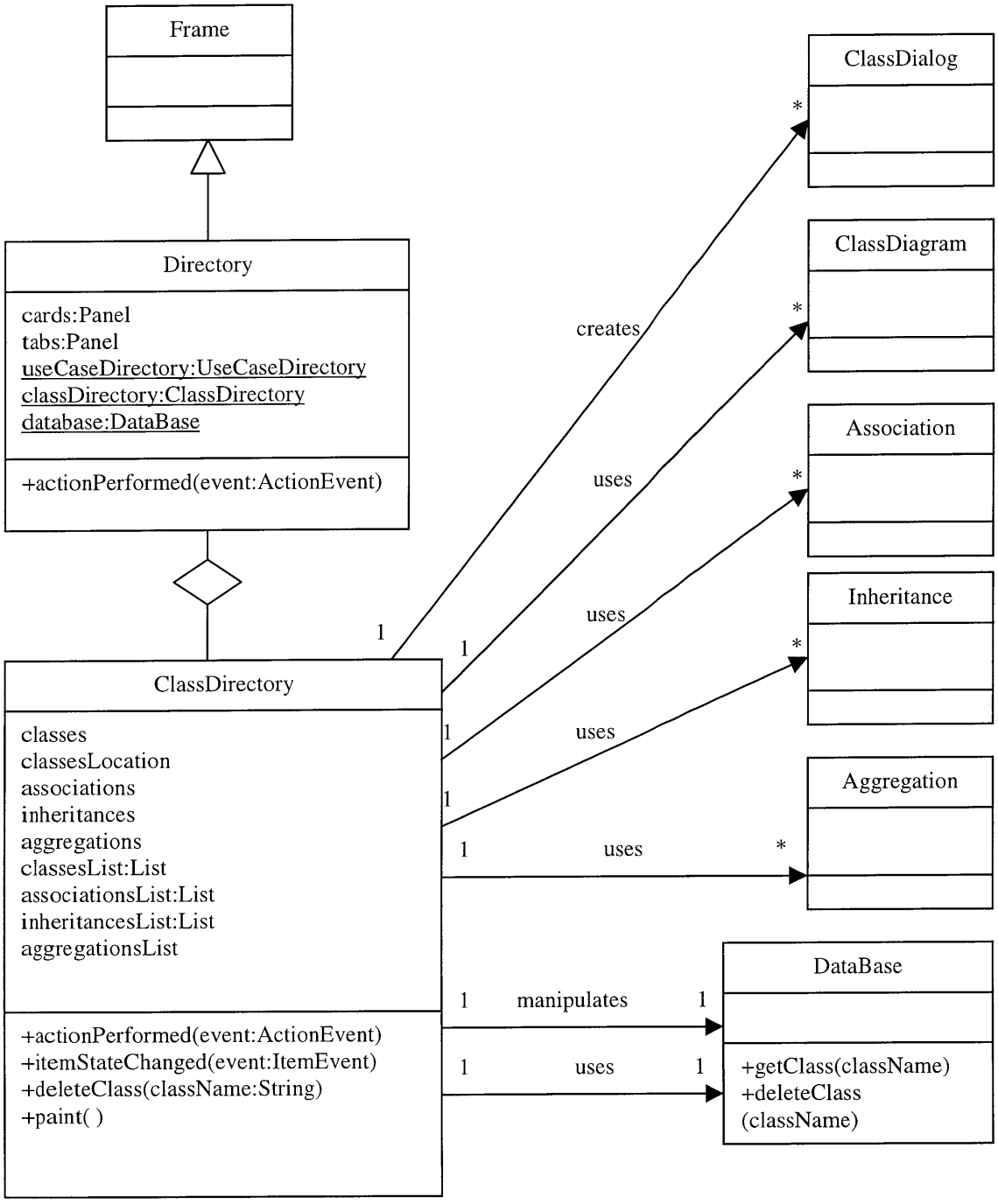


Figure D-11 ClassDirectory Design Class Diagram

D.12 UseCaseDirectory Design Class Diagram

Figure D-12 shows UseCaseDirectory Design Class Diagram.

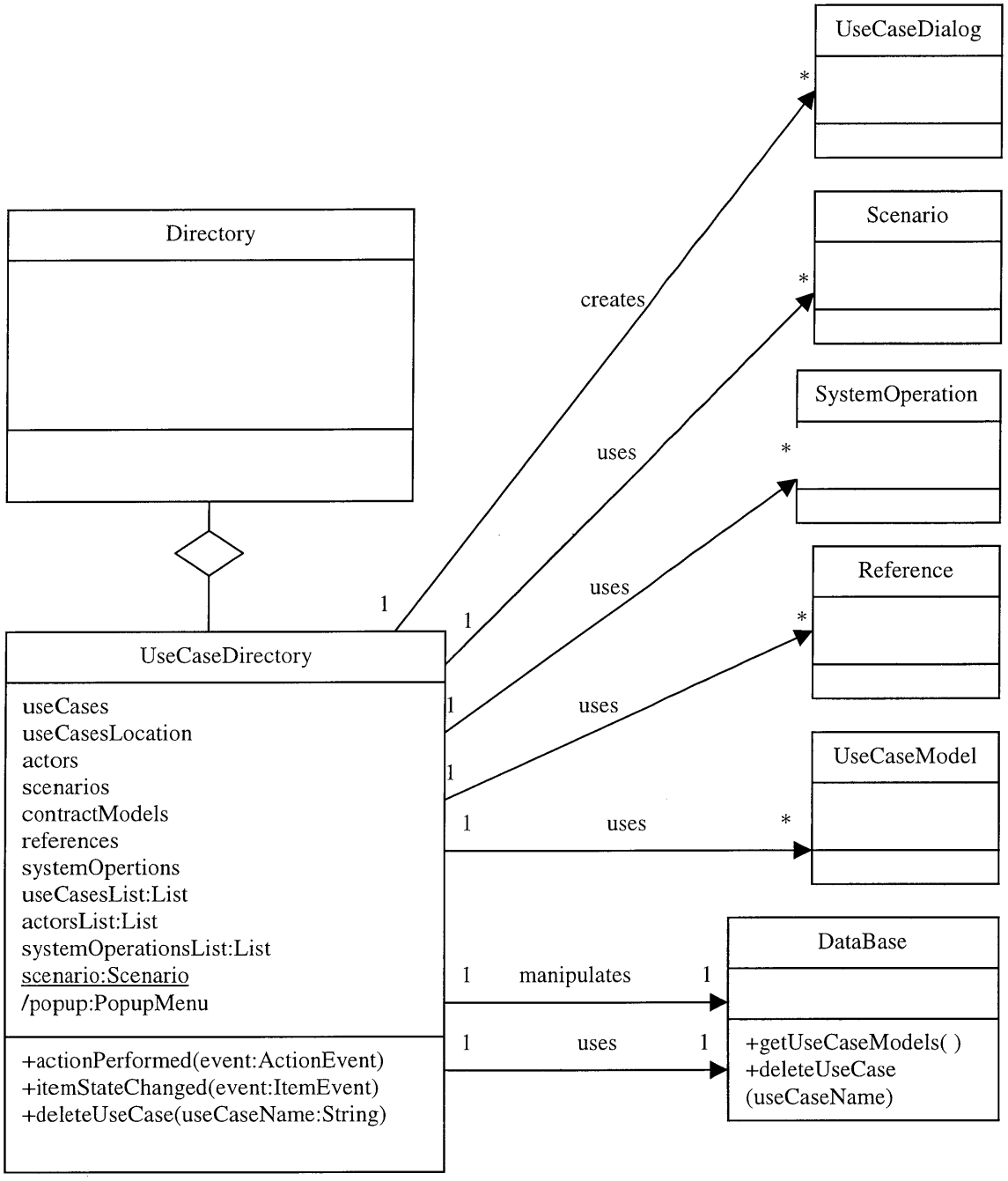


Figure D-12 UseCaseDirectory Design Class Diagram

D.13 WindowOrganizer Design Class Diagram

Figure D-13 shows WindowOrganizer Design Class Diagram.

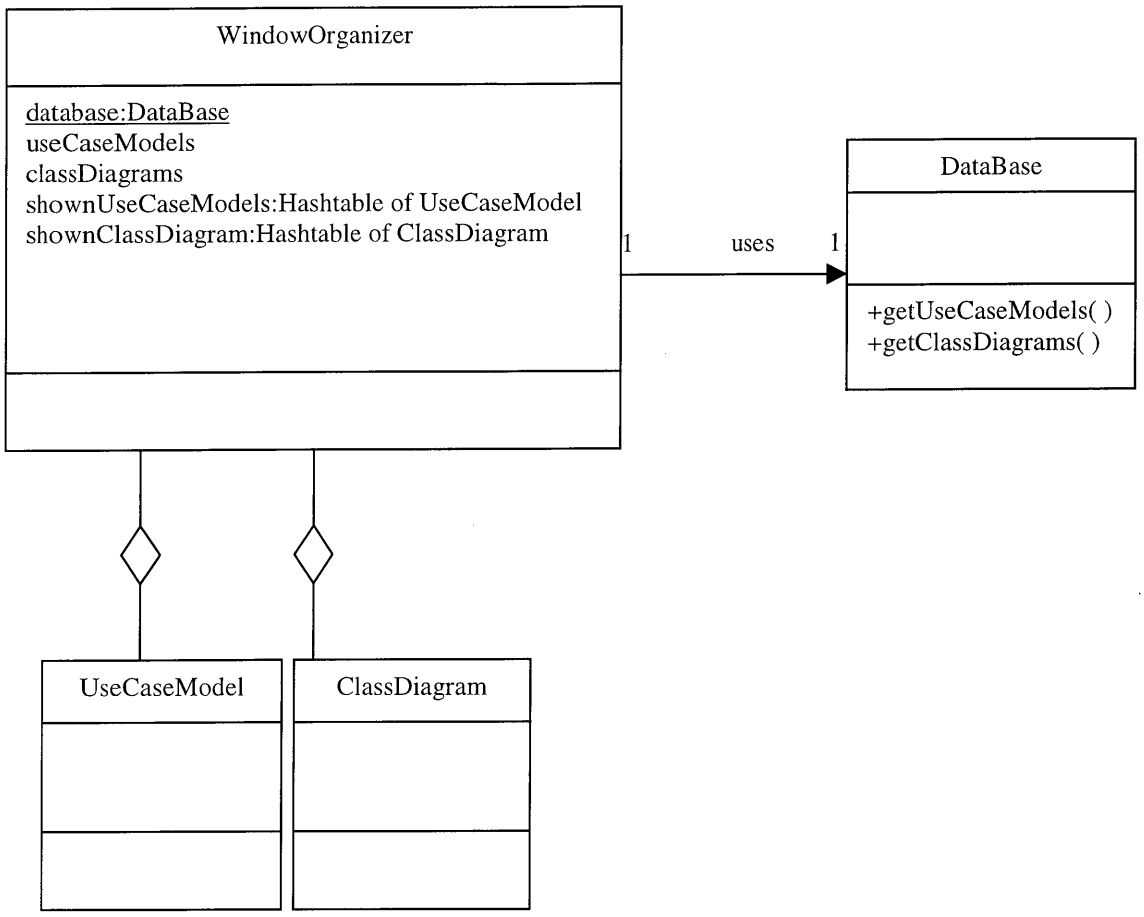


Figure D-13 WindowOrganizer Design Class Diagram

D.14 Development Life-Cycle Design Class Diagram

Figure D-14 shows Development Life-Cycle Design Class Diagram.

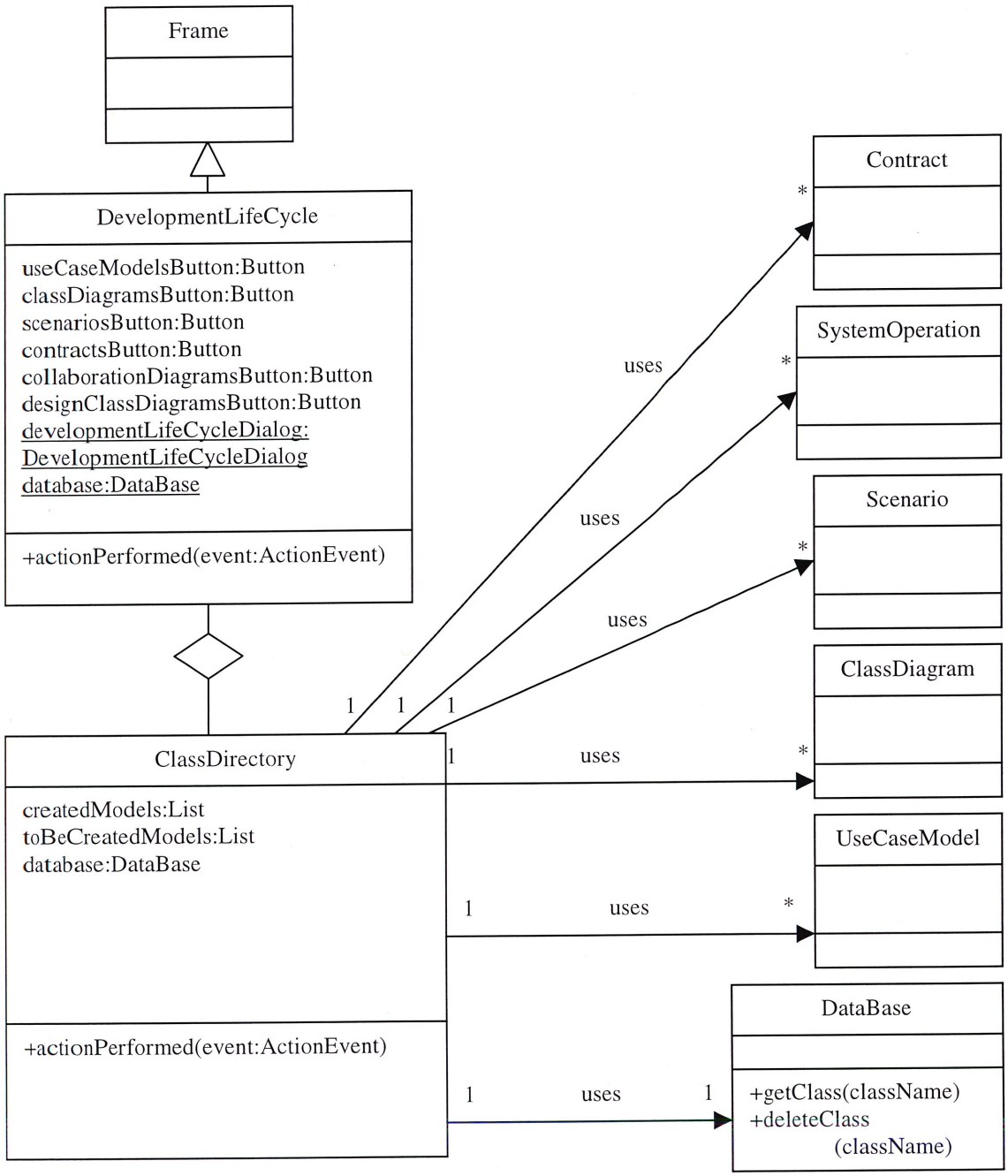


Figure D-14 ClassDirectory Design Class Diagram