User Interface Modelling Using UML for a library System

BY:

Zahra Ben-Jammaha Abu-Trife

Supervisor: Dr. Mohammed A. Khlaif

A thesis submitted to the Department of Computer Science
In partial fulfillment of the requirements for the degree of
Master of Science

Al-Tahaddi University, Faculty of science
Department of Computer science
Sirit, G. S. P. L. A. J

Academic Year 2007/2008

بسم الله الرحمن الرحيم

﴿ لاَ يُكَلِّفُ اللهُ نَفْسًا إِلاَّ وُسْعَهَا لَهَا مَا كَسَبَتْ وَعَلَيْهَا مَا اكْتَسَبَتْ رَبَّنَا لاَ تُؤَاخِذْنَا إِن نَّسِينَا أَوْ أَخْطَأْنَا رَبَّنَا وَلاَ تَحْمِلْ عَلَيْنَا إِصْرًا كَمَا حَمَلْتَهُ عَلَى الَّذِينَ مِن قَبْلِنَا رَبَّنَا وَلاَ تُحَمِّلْنَا مَا لاَ طَاقَةَ لَنَا بِهِ وَاعْفُ عَنَّا وَاغْفِرْ لَنَا وَارْحَمْنَا أَنتَ مَوْلاَنَا فَانصُرْنَا عَلَى الْقَوْمِ الْكَافِرِينَ ﴾

صدق الله العظيم

(سورة البقرة)

Faculty of Science

Department of Computer Science

Title of Thesis

User Interface Modeling Using UML for A library System

By

Zahra Ben-Jammaha Abu-Trife

Approved by:

Dr. Mohammed A. Khlaif .... [illegible]
(Supervisor)

Dr. Faraj A. El-Mouadib .... [illegible]
(External examiner)

Dr. Idris S. El-Feghi .... [illegible]
(Internal examiner)

Countersigned by: [illegible]

Dr. Ahmed Farag Mhgoub

(Dean of faculty of science)

05
07
09

[illegible]

I. Dedication

I wish to express my gratitude to my Father for his encouragement during my investigation. Really, I can not forget all his efforts, so as not to give up, and being capable to continue my research.

To my husband, for his relief, remark and for all his concessions. Truly, I am so proud to be his wife.

And finally, most gratefully, to my supervisor, who was indispensable to my own personal research.

II. Acknowledgments

First, I would like to express my thankfulness to Dr. Mohammed A. Khlaif, my honorable and respectful advisor for his extensive support and assistances throughout this project. Truly, he squandered expensive moments in order to help me to finish this research.

I want to thank my family, my husband and my friends for their encouragements.

In conclusion, I promise my supervisor Dr. Mohammed A. Khlaif, that I will do my best so as to present better and more interesting these. So as to.

Zahra

III. Abstract

Although user interfaces represent an essential part of software systems, the Unified Modeling Language (UML) is a visual language for modeling software application. We can use standard UML to model important aspects of user interfaces.

This thesis presents two ideas, the first is user interface modelling using UML and the second is generating user interface prototypes from scenarios using UML. The case study of this thesis is Library System. This case study identifies a set of UML constructors that may be used to model and generate UIs, and identifies some aspects of UIs that cannot be modelled using UML notation.

IV. Table of Contents

V. List of Tables and Figures

VI. Abbreviations and Definitions

ActivatableUIFunctionality

An ActivatableUIFunctionality is a GUI element that can be activated.

ALGAE

A Language for Generating Asynchronous Event handlers

APM

<<*apm*>> stereotype identifies the Abstract Presentation Model classes.

CPM

<<*cpm*>> stereotype identifies the Concrete Presentation Model classes.

CTS

Collaboration diagram-To-State diagram transformation algorithm.

GB

graph Block

GT

Graph of Transition

GUI

Graphical User Interface

GUI layout

Sizing and positioning GUI elements to form a functional, visually attractive

IDE

Integrated Development Environment, a software development tool that includes at least an editor, a compiler and a debugger.2

JFC

Java Foundation Classes

JVM

Java Virtual Machine screen.

MB-UIDEs

Model-Based User Interface Development Environments.

MFC

Microsoft Foundation Classes

Mockup

A no- interactive, high-fidelity representation of a GUI

OCL

Object Constraint Language; used to specify constraints and operations in UML models

OMG

Object Management Group, http://www.omg.org

OOA

"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain."4

OOD

"Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design."5

OOP

"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represent an instance of some class, and whose classes are all members of a hierarchy of classes untited via inheritance relationships."6

SQL

Structured Query Language

StaticUIFunctionality

A StaticUIFunctionality displays a screen element without providing behavior or interaction.

Stereo Type

An extensibility mechanisms that can be used to extend UML to new domains.

UIB

User Interface Block .

UIMS

User Interface Management System

Chapter One

Introduction

Numerous tendencies have formed the UML to what it is today, improving on many fields to model all important aspects of software systems. These aspects are reflected in the diagrams UML provides for modeling. Judging from these, aspects covered are classes, use cases, components, deployment, internal states of the software, activities, timing, sequences, and collaboration. Obviously, presentation is not one of them. Does it mean presentation is not an important part of software? User interfaces are valued to the users of a software system; it is the only part of the system that is visible to the users. The UML simply fails to take this into account by not providing the methods for modelling user interfaces. How can a model be complete if it ignores an aspect as important as user interfaces?

User interfaces consist of their visual representation (the layout) and the interaction they permit. UML provides diagrams for modeling interaction, and they can be applied to the interaction within user interfaces nicely. But UML does not provide a diagram for modeling the layout of user interfaces, which is especially important for graphical user interfaces (GUIs), because their graphical nature allows for more diversified designs. UML provides extensibility mechanisms that can be used to extend UML to new domains.

This thesis presents two ideas, the first is user interface modelling using UML and the second is generating user interface prototypes from scenarios using UML. The case study of this thesis is *Library System*. This case study identifies a set of UML constructors that may be used to model and generate UIs, and identifies some aspects of UIs that cannot be modelled using UML notation.

The remainder of this thesis is structured as follows: *In chapter one*, we will introduce some basic facts for reasons for choosing research area, what is a UML and why do we use UML? What is a user interface? User interface programming methods and type of users.

After the basics, *in chapter two*, we will perform a detailed analysis of the case study (library system), see how it is structured into UML diagrams, and which diagrams are used for domain model, task model, abstract presentation model, concrete presentation model and lastly, how to put all these diagrams together in one model?

In chapter three, generating user interface prototype from scenarios, we show how we suggest an approach for requirements engineering supporting the Unified Modeling Language (UML). The approach provides a five activities process for deriving a prototype of the UI from scenarios and generating a formal specification of the application. Scenarios are acquired in the form of UML collaboration diagrams and enriched with UI information. These diagrams are transformed into the UML State chart specifications of all the objects involved. The prototype is embedded in a UI builder environment for further refinement.

Finally, chapter four, we summarized our results and give outlooks of future work in this area.

1.1 Reasons for choosing research area:

- Programmers are often encouraged not to do user interfaces. In large software companies, interface designs are often done by specialists, usually user interface designers or graphic designers.
- User interface design is not part of most computer science curricula, nor is it a prominent topic in most programmers' magazines.
- Many books about user interface design are too academic to be useful, and focused on the theoretical, not the practical.

1.2 The aim

- To present a summarized description of a comprehensive UI modelling case study using UML.
- This case study has the purpose of identifying common UI modelling problems when using UML.
- Identifying a set of UML constructors and diagrams that may be used by application developers to design UIs.

1.3 Related work

Many proposals have been made for models that support the design of UI elements, using several different notations. For instance, there is research concerning the design of user tasks, as in Kirwan and Ainsworth [18] and in Johnson [17]. Moreover, there are several proposals for designing UIs using declarative models, as described in Griffiths [8] and Szekely [30]. Therefore, it would be best not to have to invent new modelling constructs for the UI if existing ones, which can be used effectively.

A number of methods have been suggested for deriving the UI from specifications of the application domain. Typically, data attributes serve as input for the selection of interaction objects according to rules based on style guidelines such as *CUA* (Common User Access) [14]. Such methods include the *Genius*, *Janus*, and *TRIDENT* approaches. In *Genius* [16], the application domain is captured in data models that are extended entity-relationship models.

1.4 What is UML?

In the field of software engineering, the **Unified Modeling Language** (UML) is a standardized specification language for object modeling. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a *UML model*. UML is not restricted to modeling software. UML is also used for business process modeling, systems engineering modeling and representing organizational structures. UML is extensible, offering the following mechanisms for customization: profiles and stereotype. [25]

1.5 Why UML?

UML supports a rich set of graphical notation elements. It describes the notation for classes, components, nodes, activities, work flow, logical, use-cases, objects, states and how to model relationships between these elements. UML also supports the notion of custom extensions through stereotyped elements. The UML provides significant benefits to software engineers and organizations by helping to build rigorous, traceable and maintainable models, which support the full software development lifecycle. [25]

1.6 What is a user interface?

In the past, applications were supplied with their own proprietary user interfaces built in some low level programming language with a unique "look and feel". While this is still the case, system integrators and end user have begun to use high level, object-oriented development environments (GUI's) such as Visual Basic and PowerBuilder to customize application interfaces and to provide a more common look and feel. These high volume programming environments also will make it much easier to create multiapplication interfaces for specific classes of users. [20]

1.7 Types of Users

Because of the user's skill level, is a primary factor determining how the user will interact with our program, understanding users' skill level plays a significant role in our user interface design. This section explores the different types of users, how their different skill levels result in different interface designs. [20]

- ***Beginning Users***

  Beginning users are determined largely by how much they know about Windows alone. And beginning users don't know much. It means that certain user interface features are probably inappropriate for them. Luckily for user interface designers, while all users start out as beginners, few of them stay for long as beginners, with more and more experience with Windows, beginning users quickly become intermediate users.

- ***Intermediate Users***

  Intermediate users understand how to use the standard features of Windows fairly well. They understand more subtle details, but they do not understand all of them. Users often become advanced users.

- ***Advanced Users***

  Like intermediate users, advanced ones understand almost all of the standard Windows user interface features. Advanced users understand most of the functionality of your program and they want to get their work done as quickly as possible.

1.8 User interface programming methods

Originally people programmed computers in binary machine code. Later assembly language was a big revolution. People could write programs using mnemonics instead of strings of zeros and ones. Soon after programming languages came and compilers, scripting languages and interpreters, visual language and visual builders and finally markup languages and reindeers. High-level programming languages gave programmers more time to think about other aspects of software development, such as the user interface; scripting languages allowed greater portability and flexibility in software; visual languages removed the need to memorize the language vocabulary; and finally, markup languages reduced the expertise needed to develop user interfaces and preserve information. The next section looks at the advantages and disadvantages of each method.

1.8.1 Low-level Programming

Low-level interface programming was the first method programmers used to create interfaces. Most or all of the code is written in assembly and used the instructional machines. The problem with assembly programming is that it is platform-specific and porting to new platforms requires a complete redesign of the applications, because each platform has its own instructional machine set. On the plus side, assembly programs are extremely fast and compact and do not require a compiler. Currently, low-level interface programming is mainly used for highly interactive applications, such as games, where response time is more important than portability.

1.8.2 High-Level Programming

One of the most popular ways to build user interfaces for applications is with a high-level language or with a visual designer. High-level programming is powerful and provides the programmer with a lot of control over details in the design and it requires significant programming experience. The most popular high-level languages are C/C++, Java, and Visual Basic.

1.8.3 Scripting Languages

Markup languages are not "functionally complete." They lack some important programming features such as conditional statements, loops and functions. Scripting languages provide a nice complement to markup languages and the combination of the two provides the functionality needed so as to build most applications. However, the power and sophistication of scripting languages has improved dramatically in recent years. The tremendousness increases in both computer speed and memory storage makes it possible to use them for a much broader range of applications than was previously possible. Scripting languages are also easier for non-computer persons to learn. [28]

1.8.4 Toolkit Programming

Toolkit programming uses object-oriented techniques to raise the level of abstraction in building user interfaces. Programmers build the interface in a high-level programming or scripting language by using widgets from a particular toolkit set. Widgets are high-level objects (e.g. buttons). The major advantage of toolkit programming is the ability to hide all the low-level details (e.g., drawing the widget on the screen) and handling low-level events (e.g., keyboard interrupts) from the programmer. Some of the most popular toolkits are Microsoft Foundation Classes (MFC) used in MS-Windows and the Motif toolkit used in X-Windows. When Sun designed the Java language, it also designed a new toolkit (Java Foundation Classes, or JFC). Each toolkit is trying to solve a different problem: portability, easy of use, looks, more features and so on. [22]

1.8.5 Visual Programming

Visual programming is like toolkit programming, but instead of writing code the programmer uses direct manipulation to design the interface. Visual builders allow the programmer to drag-and-drop widgets into a design area and then specify the events by writing code in some high-level programming or scripting language. For simple interfaces, visual programming is faster than toolkit programming; it is used for simple interfaces and quick prototyping.

1.8.6 Markup Languages

A markup description is higher than toolkit programming in the user interface abstraction. With the advent of the Web, markup languages are now used to describe and preserve user interfaces. They provide high degrees of portability and this allows cross platform user interfaces to be distributed over the Internet. Markup languages require little programming experience and they are usable by novice programmers. An example of markup languages for user interfaces is eXtensible Markup Language (XML). [9]

1.8.7 Hybrid Programming

Java Applets / HTML

An applet is a special program written in the Java programming language that can be included in an HTML page.

Perl / CGI / HTML

The combination of the three (HTML is for the user interface, Common Gateway Interface (CGI) for the communication, and Perl-scripts for the backend) is a popular combination for Web-based applications.

VBScript / ASP / HTML

Active Server Pages (ASP) is a server based on scripting language that is used to build database driven Websites, where the browser may have no scripting at all.

1.9 Specification Formats

Programming user interfaces at the toolkit level is quite difficult [21]. One way to make the user interface production process easier is with high-level tools. These tools aid the user interface production at various stages. At design time the tool lets the user interface designer creates the interface. This can be done with a graphical editor that can lay out the interface or a compiler that can process a textual specification. At run-time the tool manages the user interface and monitors the interaction with the end-user (the term "User interface Management System" or UIMS is also used for this kind of tools). This usual contains a toolkit, but may also include other software that measures the performance of the interface or other administrative work. Finally, at after-run-time the tool can help with the evaluation and debugging of the interface. Due to the lack of good user interface metrics, few tools provide support for after-run-time help.

There are several ways to classify high-level user interface tools. One way is by how the interface designer specifies what the interface should be [21]. As Figure 1.1 shows, some tools require the programmer to program in a special purpose language, some provide an application framework to guide the programming, some automatically generate the interface from a high-level model or specification, and others allow the interface to be designed interactively. Following is a more detail description of each category.

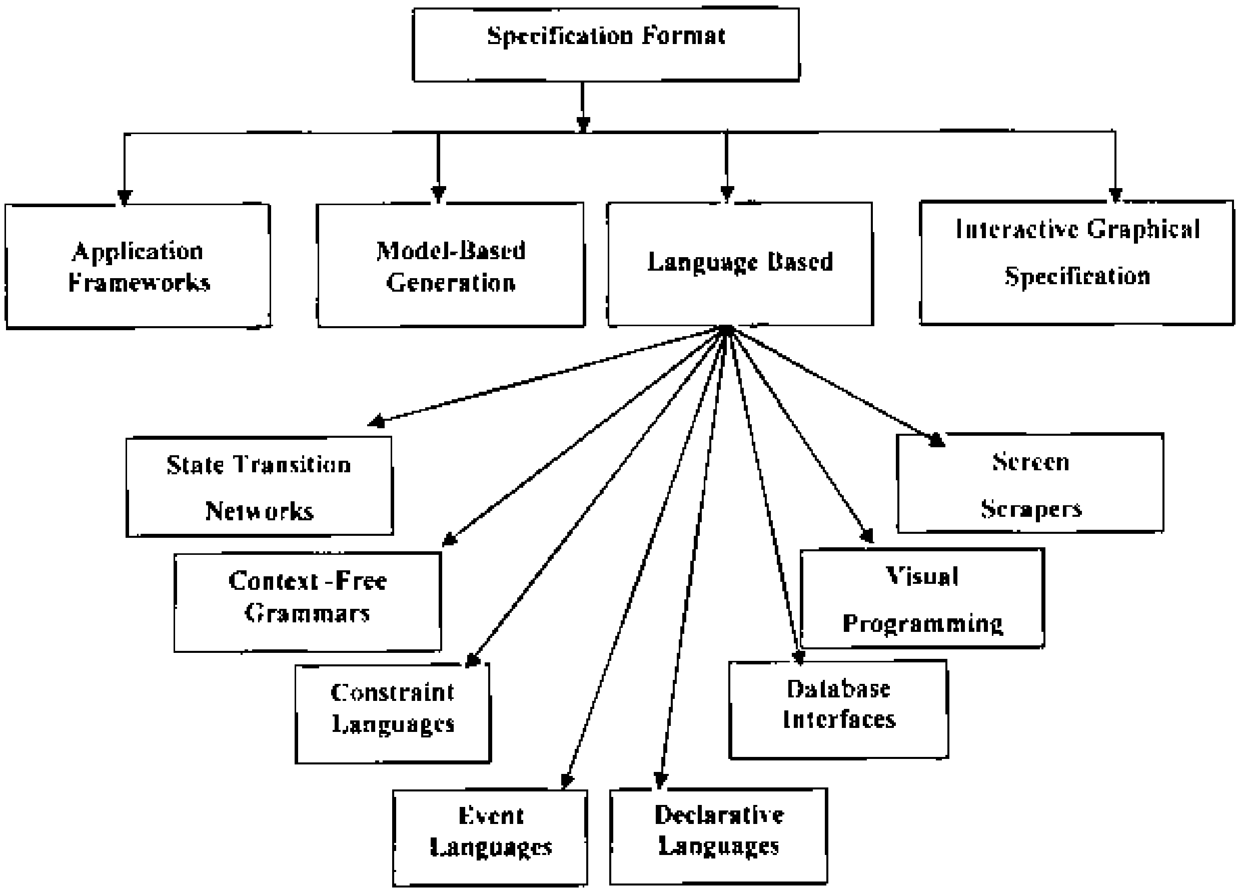


Figure 1.1: User Interface Specification Formats

1.9.1 Application Frameworks

Most windowing systems provide a low-level toolkit for building powerful and sophisticated user interfaces (e.g., XLib for X-Windows) [12]. These toolkits provide routines for controlling line-drawing, pixel coloring, cursor movement, and other low-level operations. Although necessary for some classes of interfaces, programming at this level is very difficult and requires knowledge of the underlying platform. Also, building interfaces that conform to the platform style guidelines (i.e., look-and-feel) is also difficult. Today there are many frameworks to help with the development of user interfaces. The Microsoft Foundation Classes (MFC) for Windows and the CodeWarrior PowerPlant for the Macintosh are some examples. Some frameworks span multiple platforms providing a way to enforce the same look-and-feel on multiple platforms. For example, the Java Foundation Classes (JFC) provides that same look-and-feel on any platform that has a Java Virtual Machine (JVM) implementation. JFC goes one step further in that it provides a way to separate the look-and-feel from the implementation. Accordingly, you can create a custom look-and-feel and enforce it for all applications on all platforms.

1.9.2 Model-Based Generation

All interface generation tools are faced with a trade off between giving designers control over an interface design and providing a high level of automation [31]. On one hand, giving extensive control forces designers to program by hand all the details of the design. In this case, the designer must be an expert in interface design and the interface is costly to build. Automating significant portions of the interface design. On the other hand, removes the power from the designers, allowing them to control only a few details. This is preferred for applications where few resources are available for building and maintaining the interface code (e.g., one person job). Automation can generate cheap yet complete and consistent user interfaces.

1.9.3 Interactive Specification

Creating a good user interface requires good artistic skills. The problem is that graphic designers and user interface specialists (the people who should be designing user interfaces) are not generally good programmers. Interactive specification (also called direct manipulation) programming enables users to graphically manipulate the user interface parts (and their properties) by placing objects on the screen and organize them using a pointing device. The system then generates the appropriate code thus limiting the amount of programming required.

Direct manipulation tools can be subdivided into four categories:[12]

1. Prototyping tools,
2. Wizard (sequence of cards) tools,
3. Interface builders, and
4. Graphical editors.

The ***prototyping tools*** allow the designer to quickly mock up how the interface looks for certain scenarios but cannot create the real user interface. These tools are different from "rapid prototyping" tools that can create workable user interfaces.

The ***wizard tools*** are tools for developing user interfaces that exhibit sequential behavior. The user traverses a sequence of screens (also known as cards, frame, or forms) and the final screen shows the result. Each screen contains a set of widgets, which can be static (fixed set of widgets) or dynamic (set of widgets depends on previous responses from the user). The wizard tools usually allow the designer to create both static screens (each screen individually) and dynamic screens (using a template with embedded scripts).

Interface builders allow the designer to build the interface using direct manipulation. The user selects a widget from the list of available widgets (associated with a particular toolkit) and places them on a drawing area using a pointing device. The system then generates code that is compiled with the rest of the application. An example of an interface builder is (Visual Studio) from Microsoft, which provides a graphical tool to generate a user interface and then compile it with the actual application (written in C++, Visual Basic, or Java).

In the end, ***graphical editors*** are specialized tools for data visualization applications. Although similar to interface builders, they include custom widgets for sophisticated operations (such as simulations, process control, system monitoring, network management, and data analysis).

1.9.4 User Interface Language Based Specification

From the beginning most user interface tools provided a special-purpose language for the designer to specify the user interface. Many different types of languages were developed with each language taking a different form, such as:

- **State Transition Networks**
- **Context-Free Grammars**
- **Constraints**
- **Event-Based**
- **Database Queries**
- **Screen Scrapers**
- **Visual Programming**
- **Declarative Languages**

For detailed information concerned every kind, go back to references [12], [27].

Chapter Two

User Interface Modelling With UML

In this chapter, the case study clarifies user interface modeling by using UML. Numerous tendencies have formed the UML to what it is today, improving on many fields to model all important aspects of software systems. These aspects are reflected in the diagrams UML provides for modeling. Judging from these, aspects covered are classes, use cases, components, activities, sequences, collaboration and finally the package. Presentation is not one of them, because UML does not provide a diagram for modeling the layout of user interfaces. UML provides extensibility mechanisms that can be used to extend UML to new domains. This will be explained in this chapter.

2.1 Case Study: The Library System

In this section, we explain a simple case study of a library system to illustrate the problems that faced during the modeling of user interface.

In the *Unified Modeling Language* (UML), one of the key tools for behavior modeling is the *Use Case* model. The key concepts associated with the use case model are *actors* and *use cases*. The users and any other systems that may interact with the system are represented as actors. The required behavior of the system is specified by one or more uses cases, wishes are defined according to the needs of the actors. Each use case specifies some behavior, possibly including variants that the system can perform in collaboration with one or more actors.

Use case model is intended to be used in early stages of the system analysis in order to specify the system functionality, as an external view of the system.

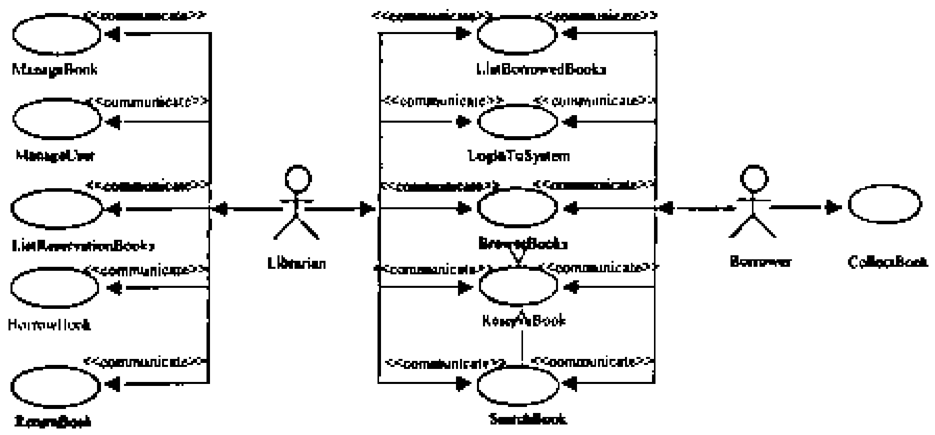


Figure 2.1: the use case diagram

We noticed from use case diagram in Figure 2.1, that there are two actors, they are *Librarian* and *Borrower*. When the *Librarian* succeeds to inter the system, it can perform the following functions:

- Manage Book (add, update and remove book records).
- Manage User (add, update and remove user records).
- Borrow Book.
- Return Book.
- ListReservationBooks.

Also from the diagram in Figure 2.1, there are some functions are performed by both *Librarian* and *Borrower:*

- List the books borrowed by a library user.
- Search Book.
- Browse Book.
- Check the availability of a book.

The use case *CollectBook*, associated with <<actor>> *Borrower*, is modeled to represent a task performed by Borrowers, although it is not implemented in the Library System. For this reason, the use case CollectBook does not have a <<communicates>> stereotype attached to it.

The Library System must ensure that *Borrower* does special functions of borrowers and *Librarian* does only special functions of librarian.

2.2 Domain Modelling

Classes and objects modeling the entities of a system are elements of the domain. Therefore the domain models describe the properties of classes and objects of the domain. From the use case diagram in Figure 2.1 and the system specification not entirely described in this research, we obtained the design of the domain model represented by the class diagram shown in Figure 2.2. This class diagram is composed of the following *<<entity>>* classes: *Library, User, Librarian, Borrower, Book* and *copy.*

- The ***Library class***: has a list of all the books and all the users of a library. It also provides find operations for the *User, Book,* and *Copy* classes.
- The ***User class***: represents people who lend copies of books from the library, and therefore they have a list of borrowed copies and a list of copies that are waiting to be picked up by the user (reserved copies).
- The ***Book class***: represents the data that is common to all copies of the book like title and author… etc. Instances of this class also provide a list of all copies of the represented book and a list of users that are waiting for a copy of this book to become available (association "reservation list").
- Each instance of the ***Copy class:*** represents a single copy of a book that is available in the library and thus there is a simple association to the Book class.

The borrowBook and returnBook methods of class Library both expect a user and a copy as arguments and are used to indicate that the given user borrows or returns the given copy.

The <<entity>> stereotype, <<control>> stereotype and <<boundary>> stereotype are used throughout this research, they were introduced by Jabcobson in his Object-Oriented Software Engineering [15] and incorporated by UML:

- The ***<<entity>>*** stereotype identifies classes and class instances that model things or objects that exist in their own right.
- The ***<<control>>*** stereotype identifies classes and class instances that perform system behavior.
- The ***<<boundary>>*** stereotype identifies classes and class instances that handle the interaction between system users and systems.

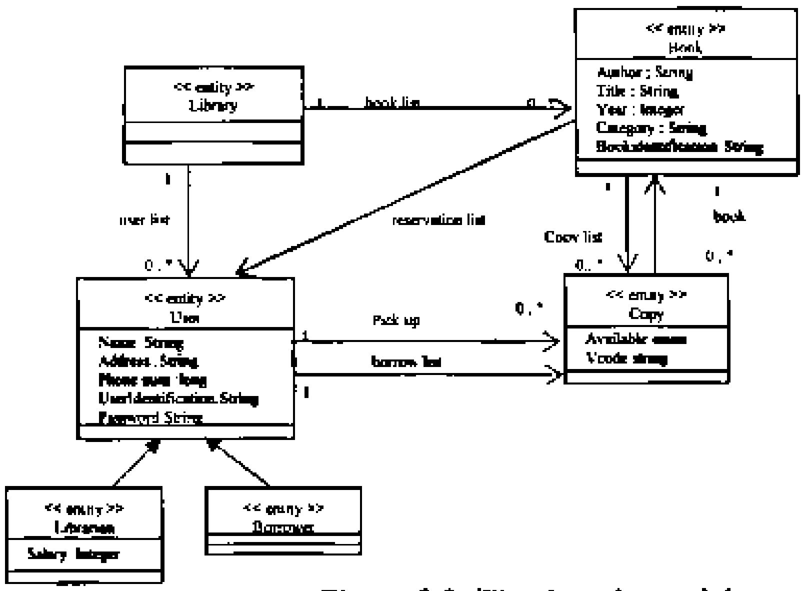


Figure 2.2: The domain model

2.3 Task Modelling

Indeed, both of use case and activities in UML represent the task's notation. We can elicit user interface functionalities required to allow users achieve their goals by using the use cases and their scenarios. In addition, to be able to identify possible ways to perform actions that support the functionalities elicited using use cases, we can use the using activities. Therefore, mapping use cases into top-level activities can help describe a set of interface functionalities similar to that described by task models.

The *BorrowBook*, *ReturnBook*, *SearchBook*, *ManageUser* and *ManageBook* use cases in Figure 2.1 shows that the *Librarian* can perform borrowing, returning, managing the books and managing users. The same use case diagram shows that the *SearchBook* and *BrowseBook* use cases shows that both of the *Librarian* and *Borrower* can search and browse a book. However, both of the librarians and borrowers must be logged to perform the functions for them. Using UML terminology this means that the actor *Librarian* use the *BorrowBook*, *ReturnBook*, *ManageUser*, *ManageBook*, *SearchBook* and *BrowseBooks* use cases, the actor *Borrower* use the *SearchBook* and *BrwoseBooks* use cases, if they previously used the *LoginToSystem* use case. In fact both of the librarian and the borrower are used *LoginToSystem*. With out it they can not enter to the system. The same use case diagram shows that, the *ReservationBook* use case extends the *SearchBook* and *BrowseBooks* use cases, but it does not explain how this extension happens.

The use case model is intended to be used in early stages of the system analysis in order to specify the system functionality, as an external view of the system, but they do not provide control flow information related to tasks and do not provide some features often associated with user requirements, like goals, pre-condition and post-condition, which may help the design stage. Therefore the activity diagram is required. The activity diagram in Figure 2.3 shows how a *Librarian* can interact with the user interface of the Library System. The activity diagram shows that after logging into the system, a *Librarian* needs to select one of the following options: *borrowing book, returning book, managing user, managing book, searching for a book or quit* the interaction with the application.

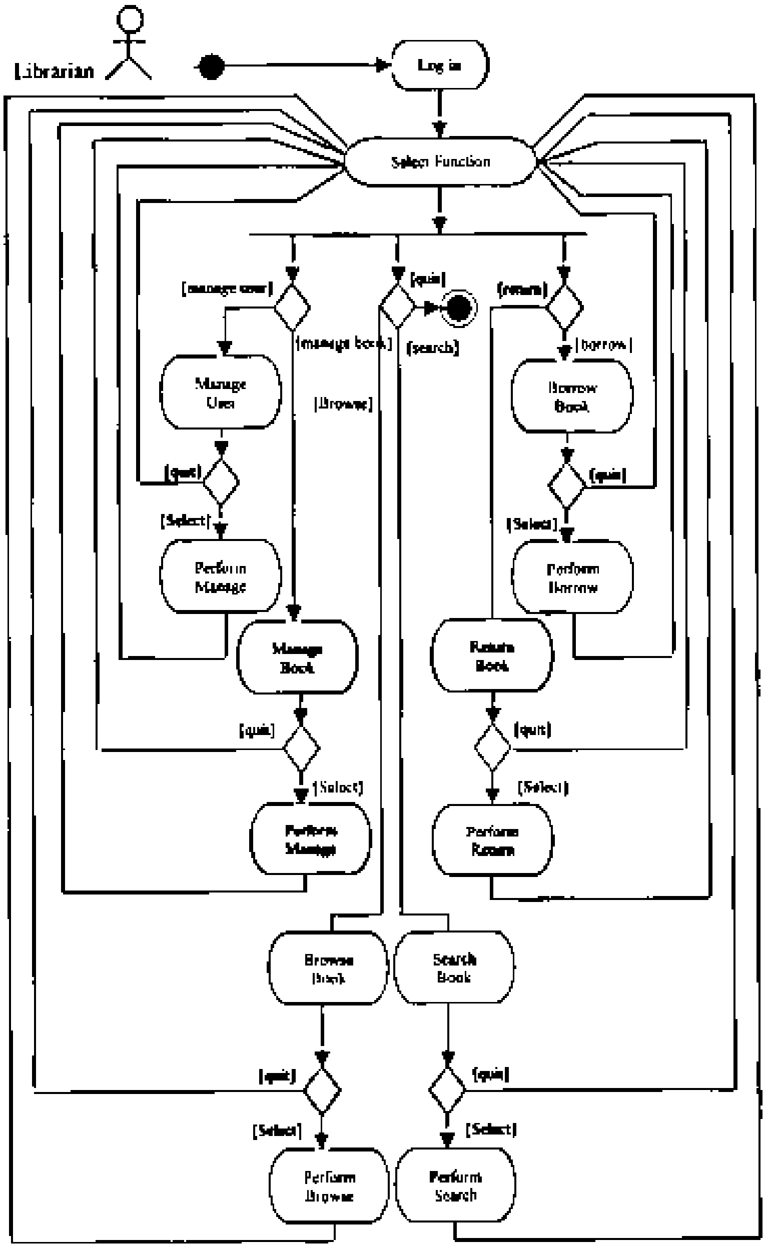


Figure 2.3: A view of the task model for Librarian.

While the activity diagram in Figure 2.4 shows the *Borrower* after logging into the system, he needs to select one of the following options: Searching for a book, Browsing the books, Display a list of the borrowed books or Quitting the interaction with the application.

Furthermore, the borrowing operation performed through Search or Browse operation, then we know if the book is available, after that ReservationBook operation is executed this is a requisition of borrowing.

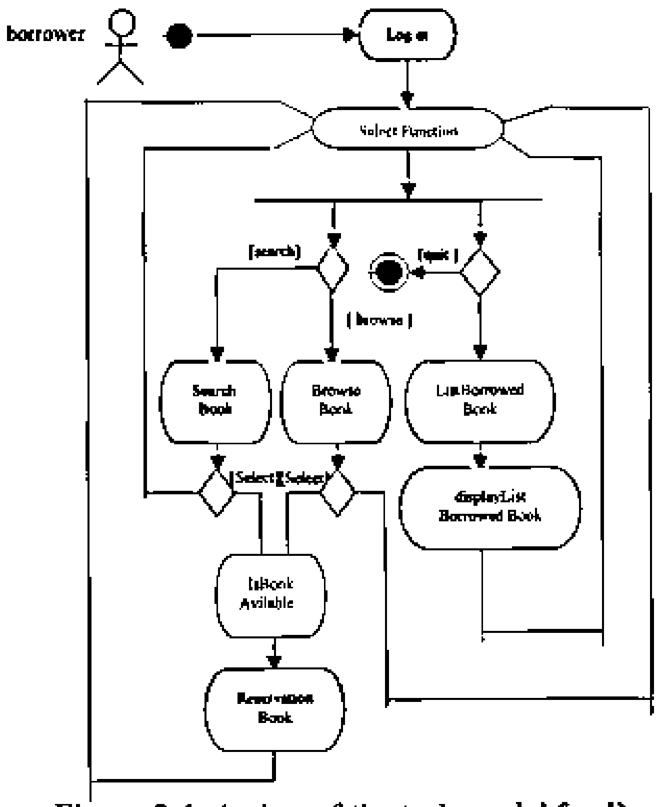


Figure 2.4: A view of the task model for Borrower.

Activity diagrams show the procedural flow of control between two or more class objects while processing an activity. Activity diagrams can be used to model higher-level process and also to model low-level internal process. For that we can decompose for some activities to give more details about the behavior of system. Therefore in my research, the activities *BorrowBook* and *Manage User* of Figure 2.3 can be explained more precisely by an additional activity diagrams as shown in Figure 2.5, 2.6(a) and 2.6(b) respectively. And so the activity *Browse book* of Figure 2.4 can be explained more precisely by an additional activity diagram as shown in Figure 2.7.

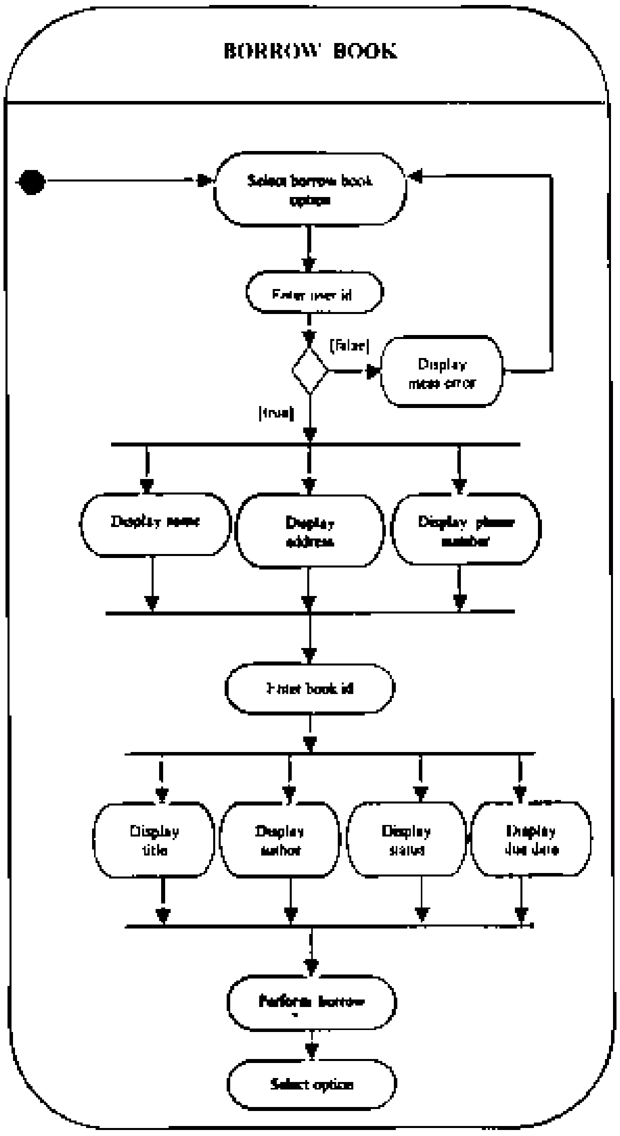


Figure 2.5: The borrow book is a partial view of the task model for Librarian.

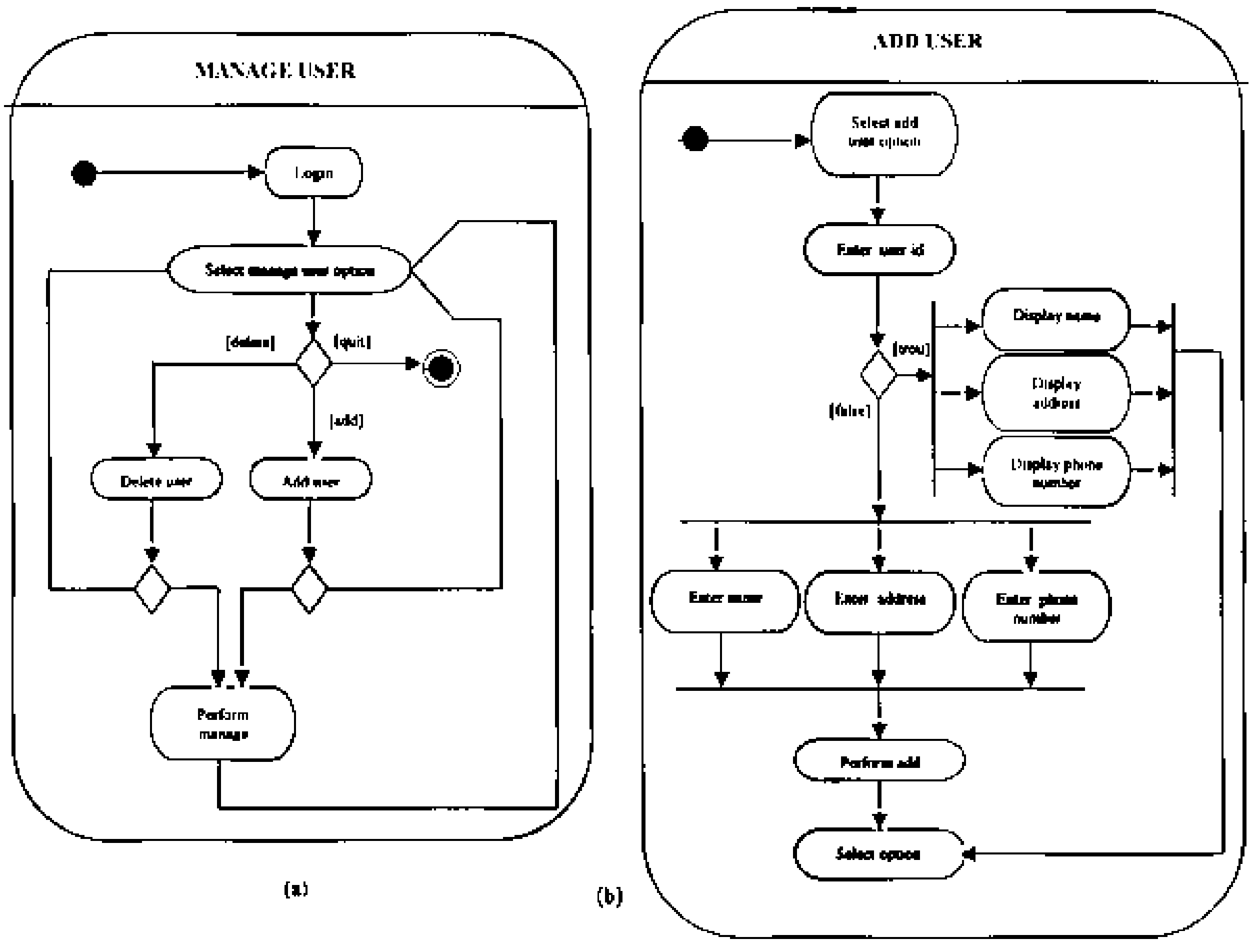


Figure 2.6: (a) The manage user is a partial view of the task model for Librarian. (b) The add user is a partial view of the task model for manage user.

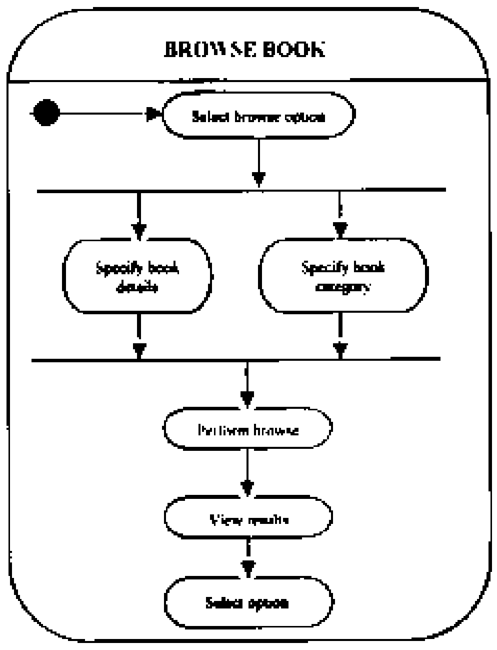


Figure 2.7: The browse book is a partial view of the task model for Borrower.

2.4 Modelling interaction objects

A user interacts with a system through interaction objects. Interaction objects are commonly classified as either abstract or concrete [19]. We can represent important features of both concrete and abstract presentations using standard UML class diagrams. Additionally, we can describe interaction objects' associated behavior using standard UML sequence diagram. However, UML provides special visualizations for abstract presentation models. UML interface diagrams are essentially UML class diagrams that clarify the purpose of individual abstract components and the containment relationships between different components.

2.5 Relationships between models in UML

We use object flows in activity diagrams to describe how to use class instances to perform actions in actionable states. In fact, by using object flows, we can incorporate the notion of state into sequence diagrams that are primarily used for modelling behavior. In UML, we can also use object flows to describe how to use interaction class instances. UML specifies categories of object flow states specific to interaction objects [2]:

- The **<<interacts>>** object flows relate primitive interaction objects to action states. They indicate that associated action states are responsible for interactions in which users invoke object operations or visualize the results of object operations.
- The **<<presents>>** object flows relate FreeContainers to activities and specify that the associated FreeContainers should be visible while the activities are active.
- The **<<confirms>>** object flows relate ActionInvokers to selection states and specify that selection states have finished normally.
- The **<<cancels>>** object flows relate ActionInvokers to composite activities or selection states and specify that activities or selection states have not finished normally and that the application flow of control should be rerouted to a previous state.
- The **<<activates>>** object flows relate ActionInvokers to other activities, thereby triggering the associated activities that start when an event occurs.

2.6 Modeling user interfaces in UML

Because we can model abstract and concrete interaction objects using class diagrams, no particular need seems to exist to extend UML's representational facilities to describe interface components. However, class diagrams don't necessarily provide an intuitive interface representation. UML provides an alternative diagram notation for describing abstract interaction objects. UML's user interface diagram consists of five constructors [7]:

- The ***FreeContainers*** or ***AbstractForm*** is a top-level interaction class that no other interaction class can include.
- The ***Containers*** is a mechanism that groups interaction classes other than *FreeContainers*.
- The ***StaticDisplay*** category is related to those components that just provide some visual information, such as labels.
- The ***ActionInvoker*** category is related to those components that can receive system events that are propagated as system operations, such as buttons. *ActionInvokers* receives direct instructions from users
- The ***InteractionControl*** category is related to those components that can receive system events that normally model user options concerning navigation through the UI. such as menus.
  - ***Inputters*** receive information from users.
  - ***Editors*** facilitate provide two-ways to exchange the information.
  - ***Displayers*** send information to the users.

Bodart and Vanderdonckt [2] provide a more precise discussion of the categorization of abstract components.

2.7 Abstract Presentation Modelling

Presentation models Classes and objects responsible for the visual appearance of user interfaces are structural elements, interaction objects are usually called *widgets*. Presentation models are structural models describing properties of widgets and their classes. Interaction objects can be concrete interaction objects (CIOs) and abstract interaction objects (AIOs). The CIOs are the widgets that compose the UI. The AIOs are abstractions of these widgets that describe if interaction objects are used for data input (*inputter*), data output (*displayer*) or both (*editor*) in presentation models.

In the abstract presentation model, we do not need a detailed model of the UI presentation, but only [5]:

- To know what kind of components compose the UI.
- How many components there are.
- How may they be grouped?
- Know which operations these UI elements should have. Therefore, we need an *abstract presentation model.*

The modeling of a user successfully logging into the Library System, regular borrow book, successfully search book, regular browse books and regular manage user into the Library System, can be used to exemplify the use of an abstract presentation model.

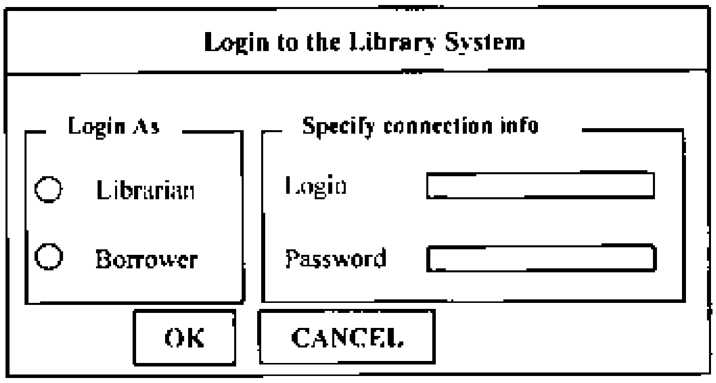


Figure 2.8: the display of the *LoginUI*

The ***LoginUI*** object presents to the user a login user interface, requesting a login name and a password. This user interface can be something like the form shown in Figure 2.8, which is not a UML diagram. However, it would be good to have a notation that allows designers to specify widgets and their layout or to abstract over such details, should they choose to do so.

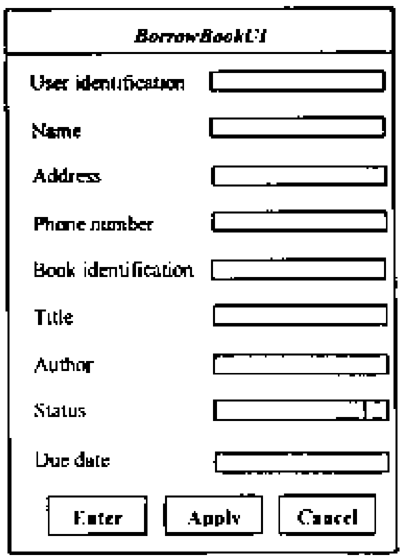


Figure 2.9: The display of the *BorrowBookUI*

The *BorrowBookUI* object presents to the user a loan user interface as the form in the Figure 2.9, this form request a user identification and document identification, then displaying a name, an address, a phone number, a title, an author, a status and a due date.

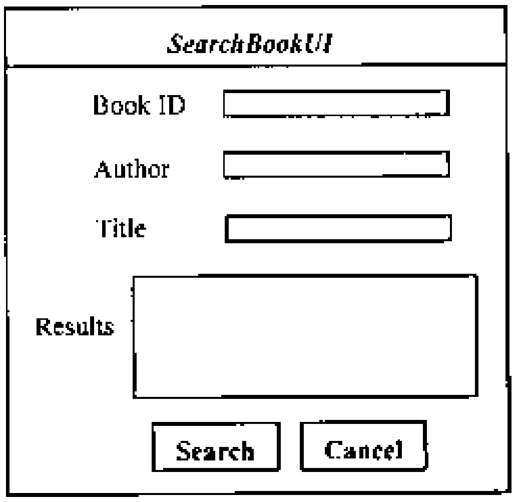


Figure 2.10: The display of the *SearchBookUI*

The *SearchBookUI* object presents to the user a search user interface as in Figure 2.10, which requesting book identification, author, title or a combination of these, and then displaying the result.

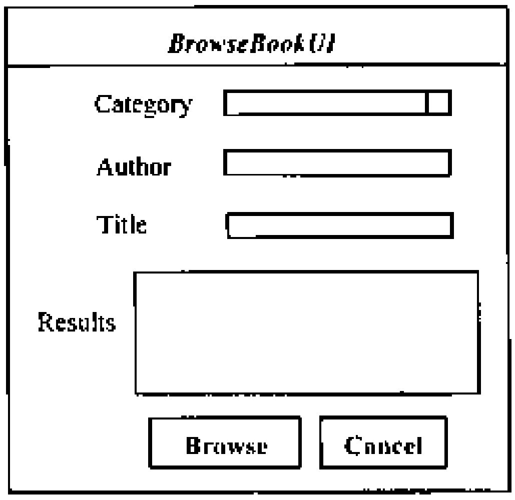


Figure 2.11: The display of the *BrowseBookUI*

The *BrwoseBookUI* object presents to the user a browse book user interface as the form in the Figure 2.11, this form requesting specific the category of the book, author, title or a combination of these, or without them all, and then displaying a result.

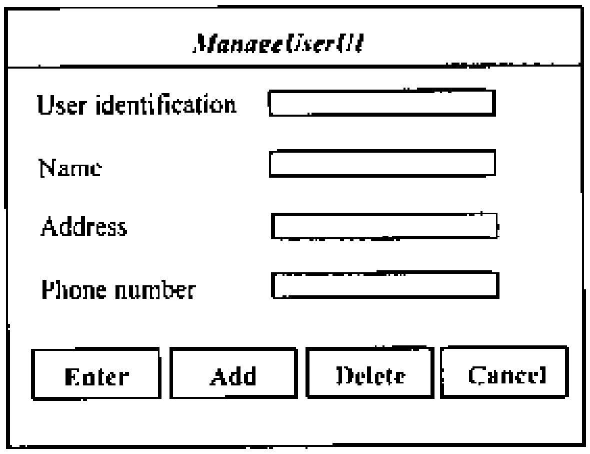


Figure 2.12: The display of the *ManageUserUI*

The *ManageUserUI* object presents to the user the additional or the removed user interface, which requesting a user identification, a name, an address and a phone number, this is an additional case. But in the deleting case, user identification will be requested .This user interface can be something like the form shown in Figure 2.12.

Figures 2.8, 2.9, 2.10, 2.11, 2.12 are not a UML diagram since UML does not specify any notation for designing presentations. In fact, we are not claiming that UML should have a UI mock up notation that can lead to an early commitment in terms of UI layout and component selection. However, we argue that UML needs a notation that can describe better the structure of abstract user interfaces than class and object diagrams. In fact, such notation could be used early in the UI design even to support the task design using activity diagrams.

We can describe interaction objects' associated behavior using standard UML sequence diagram, description of how UML constructs can be used to model UI presentations is presented in the next section.

2.7.1 *Abstract Presentation Structure*

The abstract presentation model (APM) in Figure 2.13 provides a generic description of classes and their relationships used to represent abstract widgets. There the APM has a top-level container: which are the *<<apm>> FreeContainer* (*AbstractForm*) and the Container are defined in section 2.5. All the structural elements of the UI presentation are represented by the abstract component *InteractionClass*. The *ActionInvoker* sub-category of *InteractionClass*. The *PrimitiveInteractionClass* sub-category of *InteractionClass* can be further specialized into *Displayer*, *Inputter* and *Editor*.

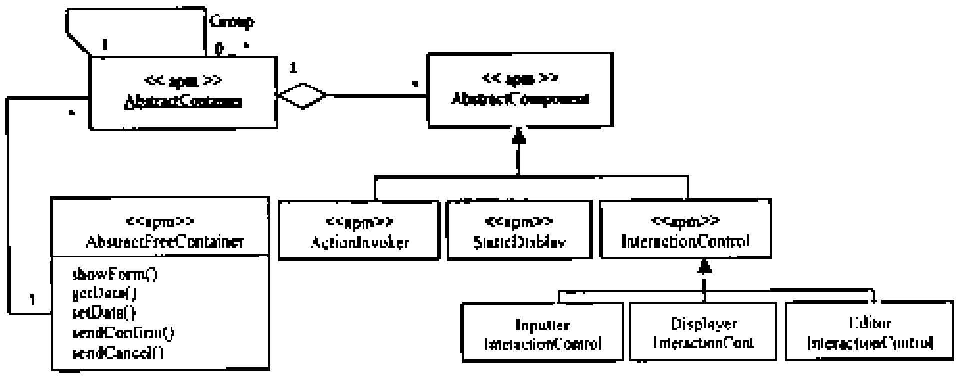


Figure 2.13: The abstract presentation model.

An object diagram of the class diagram in Figure 2.13 of the user interface *loginUI* is described by the model shown in Figure 2.14. The ***compose*** is the name between *AbstractComponents* and *AbstractContainers*, while the ***integrate*** is representing the links between two instances of *AbstractContainers*.

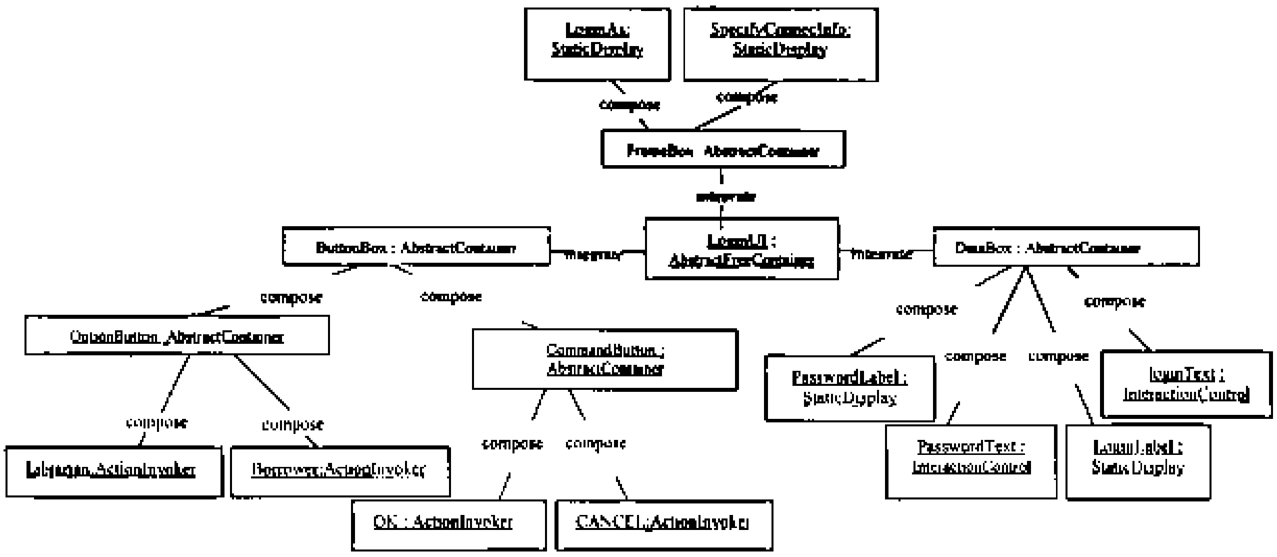

LoginAs: StaticDisplay
SpecifyConnectInfo: StaticDisplay
compose
compose
FrameBox : AbstractContainer
integrate
LoginUI : AbstractFreeContainer
ButtonBox : AbstractContainer
integrate
DataBox : AbstractContainer
integrate
compose
compose
OptionButton : AbstractContainer
CommandButton : AbstractContainer
compose
compose
compose
compose
Librarian : ActionInvoker
Borrower : ActionInvoker
OK : ActionInvoker
CANCEL : ActionInvoker
PasswordLabel : StaticDisplay
PasswordText : InteractionControl
LoginLabel : StaticDisplay
loginText : InteractionControl

Figure 2.14: The abstract model of the *LoginUI*

An object diagram of the class diagram in Figure 2.13 of the user interface *BorrowBookUI* is described by the model shown in Figure 2.15.

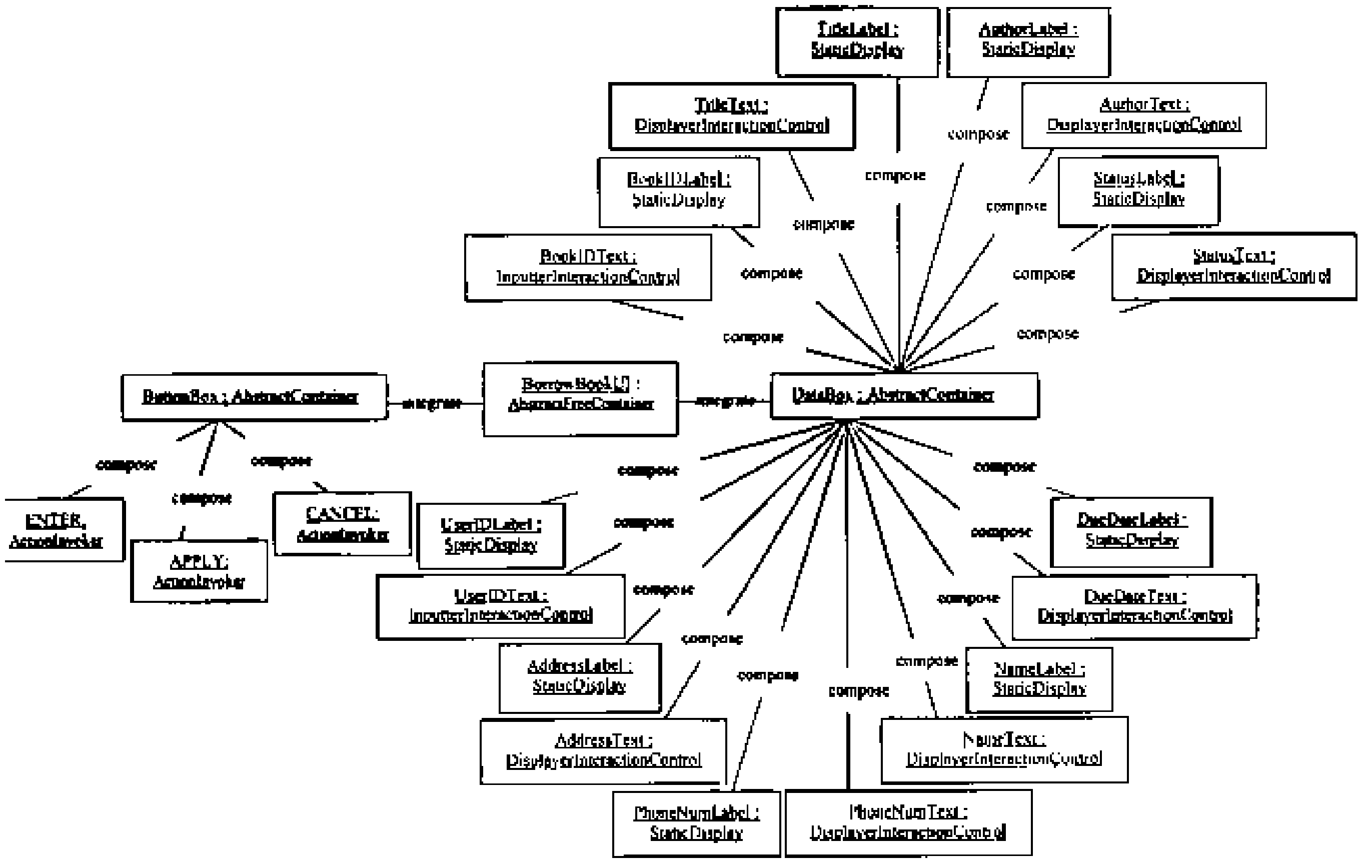

TitleLabel : StaticDisplay
AuthorLabel : StaticDisplay
TitleText : DisplayerInteractionControl
AuthorText : DisplayerInteractionControl
BookIDLabel : StaticDisplay
StatusLabel : StaticDisplay
BookIDText : InputterInteractionControl
StatusText : DisplayerInteractionControl
compose
ButtonBox : AbstractContainer
integrate
BorrowBookUI : AbstractFreeContainer
integrate
DataBox : AbstractContainer
ENTER : ActionInvoker
APPLY : ActionInvoker
CANCEL : ActionInvoker
UserIDLabel : StaticDisplay
UserIDText : InputterInteractionControl
AddressLabel : StaticDisplay
AddressText : DisplayerInteractionControl
PhoneNumLabel : StaticDisplay
PhoneNumText : DisplayerInteractionControl
NameText : DisplayerInteractionControl
NameLabel : StaticDisplay
DueDateText : DisplayerInteractionControl
DueDateLabel : StaticDisplay

Figure 2.15: The abstract model of the *BorrowBookUI*

An object diagram of the class diagram in Figure 2.13 concerned the user interface *SearchBookUI* is described by the model shown in Figure 2.16.

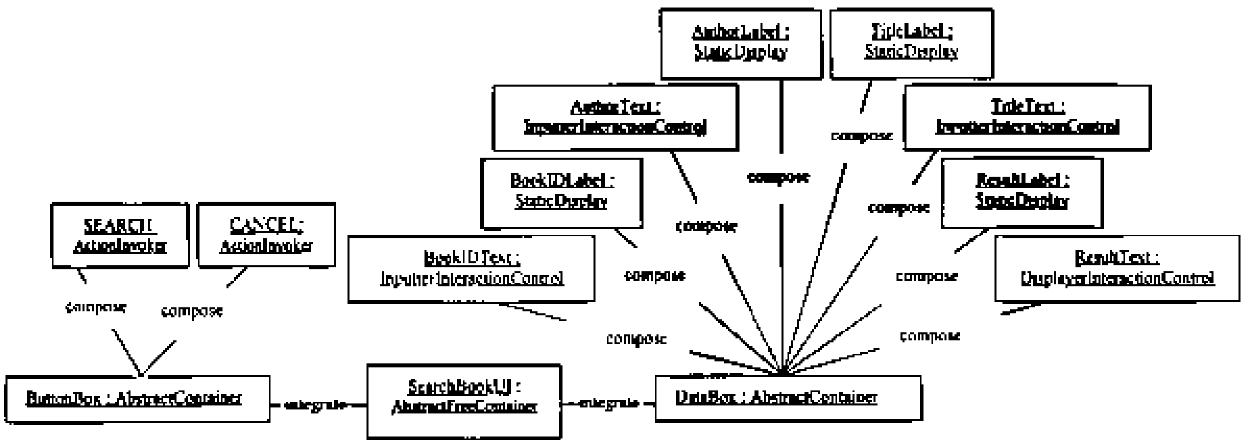

AuthorLabel : StaticDisplay
TitleLabel : StaticDisplay
AuthorText : InputterInteractionControl
TitleText : InputterInteractionControl
BookIDLabel : StaticDisplay
ResultLabel : StaticDisplay
SEARCH : ActionInvoker
CANCEL : ActionInvoker
BookIDText : InputterInteractionControl
ResultText : DisplayerInteractionControl
compose
ButtonBox : AbstractContainer
integrate
SearchBookUI : AbstractFreeContainer
DataBox : AbstractContainer

Figure 2.16: The abstract model of the *SearchBookUI*

An object diagram of the class diagram in Figure 2.13 of the user interface *BrowseBooksUI* is described by the model shown in Figure 2.17.

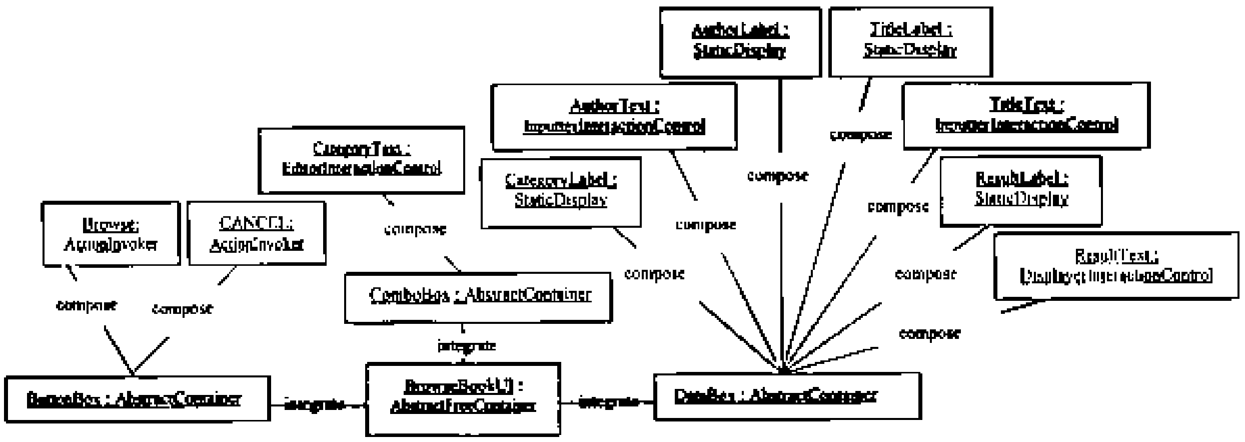

AuthorLabel : StaticDisplay
TitleLabel : StaticDisplay
AuthorText : InputterInteractionControl
TitleText : InputterInteractionControl
CategoryText : EditorInteractionControl
CategoryLabel : StaticDisplay
ResultLabel : StaticDisplay
Browse : ActionInvoker
CANCEL : ActionInvoker
ResultText : DisplayerInteractionControl
ComboBox : AbstractContainer
compose
integrate
ButtonBox : AbstractContainer
BrowseBookUI : AbstractFreeContainer
DataBox : AbstractContainer

Figure 2.17: The abstract model of the *BrowseBookUI*

An object diagram of the class diagram in Figure 2.13 of the user interface *ManageUserUI* is described by the model shown in Figure 2.18.

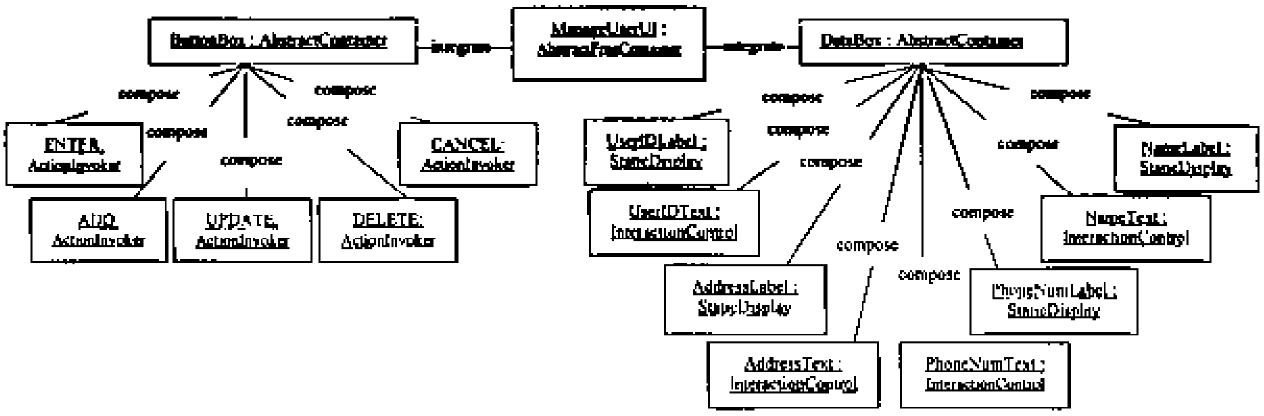

ButtonBox : AbstractContainer
integrate
ManageUserUI : AbstractFreeContainer
integrate
DataBox : AbstractContainer
compose
ENTER: ActionInvoker
ADD: ActionInvoker
UPDATE: ActionInvoker
DELETE: ActionInvoker
CANCEL: ActionInvoker
UserIDLabel : StaticDisplay
UserIDText : InteractionControl
AddressLabel : StaticDisplay
AddressText : InteractionControl
PhoneNumLabel : StaticDisplay
PhoneNumText : InteractionControl
NameLabel : StaticDisplay
NameText : InteractionControl

Figure 2.18: The abstract model of the *ManageUserUI*

2.7.2 Abstract Presentation Behaviour

A five operations are defined in the APM in Figure (2.13): *showForm(), getData(), setData(), sendConfirm()* and *sendCancel()*. These are the abstract operations of UI presentation elements that should be implemented by (widgets) *<<boundary>>* objects.

- ***showForm()*** specify that the FreeContainer is a presentation unit. This means that the widgets directly contained by UI must be instantiated and must be made visible when the UI is activity.
- ***getData()*** operation informs ***<<boundary>>*** objects relate *InteractionControl* that collects information provided by the user after an interaction, doing any required transformation on the information provided into suitable parameters for system operations.
- ***setData()*** operation informs ***<<boundary>>*** objects relate *InteractionContro*, gets information provided by the system operations, then displayed them on the out put device.
- ***sendConfirms()*** operation informs ***<<boundary>>*** objects relate *ActionInvoker* that the system's user is submitting information to the system,
- ***sendCancels()*** operation specifies that the Cancel ActionInvoker is active and can finish the Connect activity any time when the control flow is there.

2.7.3 Using the Abstract Presentation Model

In the Figure 2.19, the actor (*Library* or *Brower)* make interaction with the *LoginUI* like Figure 2.8, when the interaction is happened, it could pick up the Name and the Password to the library system through *Login()* function, the object *LibrarySystem* verify the name and the password of the user by *verifyUser()* operation where the *LibrarySystem* Passing the name and the password as a parameters to the Data Base. Data Base executes *Search()* operation about the name and the password. If the name and the password are correct, then create the *MainUI.*

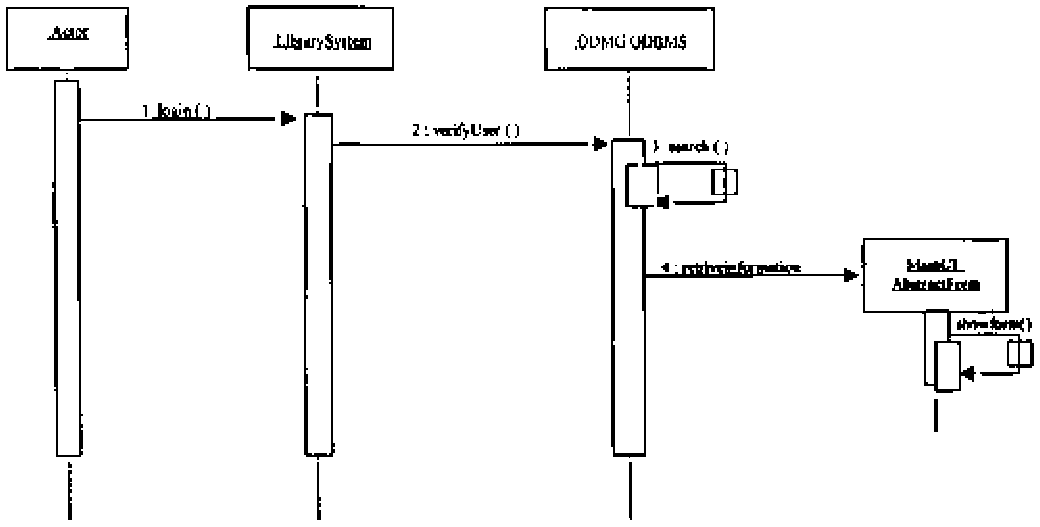


Figure 2.19: A sequence diagram for the *LoginToSystem* use case.

From the ***RegularBorrowBook*** sequence diagram in Figure 2.20, the *LibrarySystem* creates the *<<boundary>> BorrowBookUI* object of class *AbstractForm* (*AbstractFreeContainer*), which executes the *showForm()* method. This method draws the *BorrowBookUI* form that presented to the user. Interacting with the UI the user sends a *sendConfirmation()* message to the *BorrowBookUI* object. The *sendConfirmation()* message can be an event associated with the Enter button shown in Figure 2.9, but this is not specified during the abstract presentation modelling. The *BorrowBookUI* object performs a *getData()* operation that picks up the data provided by the user. After collecting the data, the *BorrowBookUI* object sends a system operation message *checkUser()* to the *<<control>> BorrowBookController* object, passing the user identification as parameter.

The *BorrowBookController* object prepares a query that is submitted to a database management system. If there are objects of class *Person* with the provided user identification in the database, the database instantiates *Person*. Then, the *<<control>> BorrowBookController* object sends a message to the *<<entity>> Person* object checking the provided user identification. If the user identification is correct, the *<<control>> BorrowBookController* object sends a *getbackData()* message to the *<<entity>> Person* object. The *getbackData()* message recovered the information (name, address and phone number) for the user, who has the user identification. After collecting the data, the *<<control>> BorrowBookController* object sends a system operation message *display ()* to the *BorrowBookUI* form object which executes the *showData()* method. This method shows the data (user information) on the *BorrowBookUI* form that presented to the user.

For the next time, the user interact with the *BorrowBookUI*, the user sends a *sendConfirmation()* message to the *BorrowBookUI* object. The *sendConfirmation()* message can be an event associated with the Enter button shown in Figure 2.9. The *BorrowBookUI* object performs a *getData()* operation that picks up the data provided by the user. After collecting the data, the *BorrowBookUI* object sends a system operation message *checkBook()* to the *<<control>> BorrowBookController* object, passing the book identification as parameter. The *BorrowBookController* object prepares a query that is submitted to a database management system. If there are objects of class *Book* with the provided book identification in the database, the database instantiates *Book*. Then, the *<<control>> BorrowBookController* object sends a message to the *<<entity>> Book* object checking the provided book identification. If the book identification is correct, the *<<control>> BorrowBookController* object sends a *getbackData()* message to the *<<entity>> Book* object. The *getbackData()* message recovered the information (title, author, status and due date) for the book, who has the book identification. After collecting the data, the *<<control>> BorrowBookController* object sends a system operation message *display ()* to the *BorrowBookUI* form object which executes the *showData()* method. This method shows the book information (book identification, title, author, status and due date) on the *BorrowBookUI* form that presented to the user.

For the thired time, the user interact with the *BorrowBookUI*, the user sends a *sendConfirmation()* message to the *BorrowBookUI* object. The *sendConfirmation()*

message can be an event associated with the Apply button shown in Figure 2.9. The *BorrowBookUI* object performs a *getData()* operation that picks up the data provided from the last operation. After collecting the data, the *BorrowBookUI* object sends a system operation message *checkBook()* to the *<<control>> BorrowBookController* object, passing the book identification and book status as parameter. The *BorrowBookController* object prepares a query that is submitted to a database management system. To make sure that the book status, is not borrowed in the database, the database instantiates *Book*. Then, the *<<control>> BorrowBookController* object sends a message to the *<<entity>> Book* object checking the provided book identification and book status. If the book identification and the book status are correct, the *<<control>> BorrowBookController* object sends a *loanBook()* message, which perform *loanBook()* operation, to the database management system, that instantiates *Loan* object. When the *loanBook()* operation is finished, the *BorrowBookController* object creates a *NewBorrowBookUI* object and destroys the last *BorrowBookUI*.

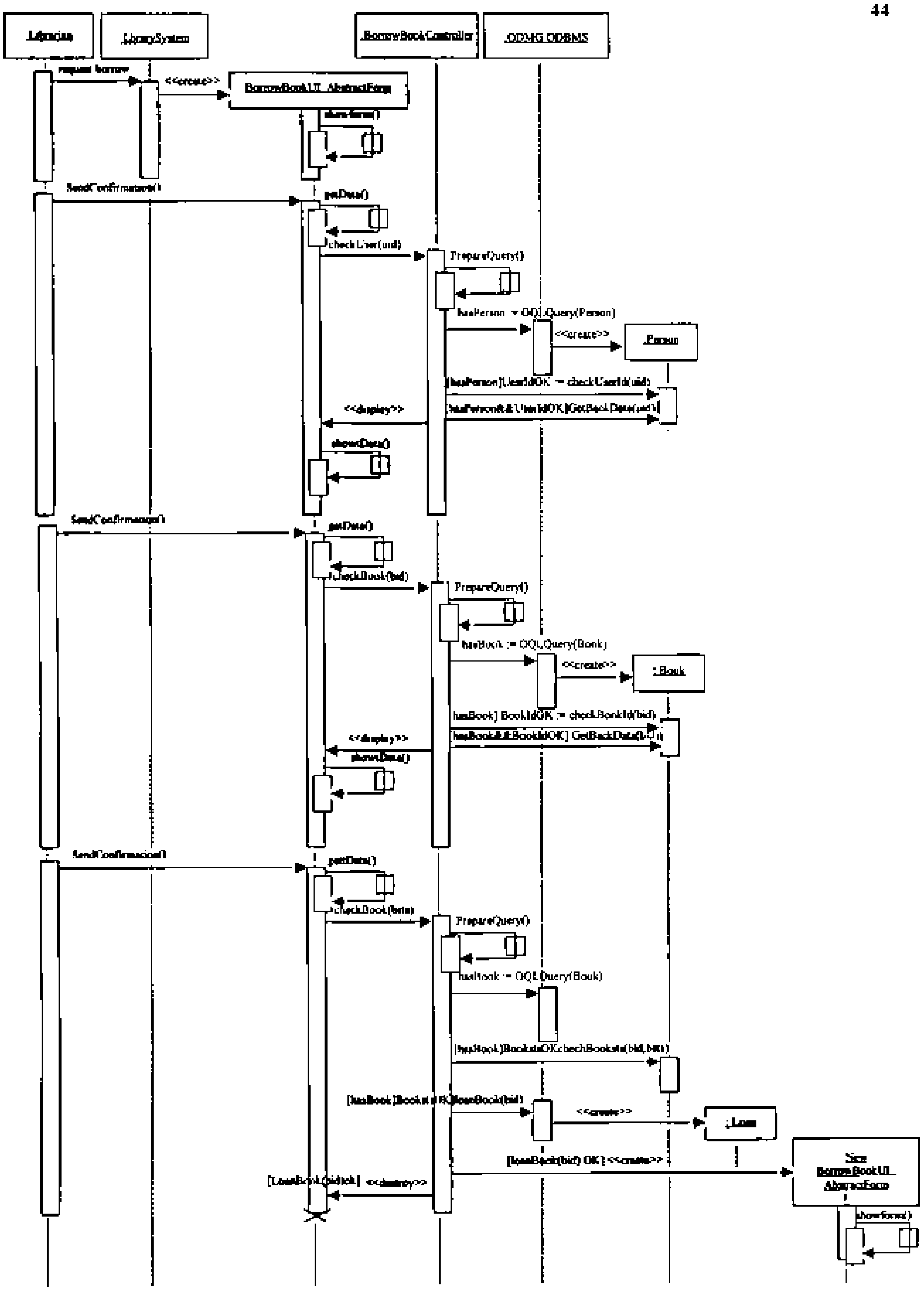

Librarian
LibrarySystem
BorrowBookController
ODMG ODBMS
request borrow
<<create>>
BorrowBookUI_AbstractForm
showform()
SendConfirmation()
getData()
checkUser(uid)
PrepareQuery()
hasPerson := OQL.Query(Person)
<<create>>
:Person
[hasPerson]UserIdOK := checkUserId(uid)
[hasPerson&&UserIdOK]GetBackData(uid)
<<display>>
showData()
SendConfirmation()
getData()
checkBook(bid)
PrepareQuery()
hasBook := OQL.Query(Book)
<<create>>
:Book
hasBook] BookIdOK := checkBookId(bid)
[hasBook&&BookIdOK] GetBackData(bid)
<<display>>
showData()
SendConfirmation()
getData()
checkBook(bid)
PrepareQuery()
hasBook := OQL.Query(Book)
[hasBook]BookstatOK:checkBookstat(bid, bst)
[hasBook]BookstatOK]loanBook(bid)
<<create>>
:Loan
[loanBook(bid) OK] <<create>>
New BorrowBookUI_AbstractForm
[LoanBook(bid)ok] <<destroy>>
showform()

Figure 2.20: A sequence diagram for the *BorrowBook* use case.

The ***SearchBook*** sequence diagram in Figure 2.21, the *Library System* creates the *<<boundary>> SearchBookUI* object of class *AbstractForm* (*AbstractFreeContainer*), which executes the *showForm()* method. This method draws the *SearchBookUI* form that presented to the user. Interacting with the UI, the user sends a *sendConfirmation()* message to the *SearchBookUI* object. The *sendConfirmation()* message can be an event associated with the Search button shown in Figure 2.10, but this is not specified during the abstract presentation modelling. The *SearchBookUI* object performs a *getData()* operation that picks up the data provided by the user. After collecting the data, the *SearchBookUI* object sends a system operation message *checkBook()* to the *<<control>> SearchBookController* object, passing the book identification, author, title, or a combination of them as parameters. The *SearchBookController* object prepares a query that is submitted to a database management system. If there are objects of class *Book* with the provided (book identification, author or title) in the database, the database instantiates *Book*. Then, the *<<control>> SearchBookController* object sends a message to the *<<entity>> Book* object checking the provided book identification, author or title. If there at least one item from the last items is correct, the *SearchBookController* object performs a *getbackData()* operation that get back the data provided by the database management system. After collecting the data, the *<<control>> SearchBookController* object sends a system operation message *display()* to the *SearchBookUI* object, which executes the *showData()* method. This method shows the data (result of searching) on the *SearchBookUI* form that presented to the user.

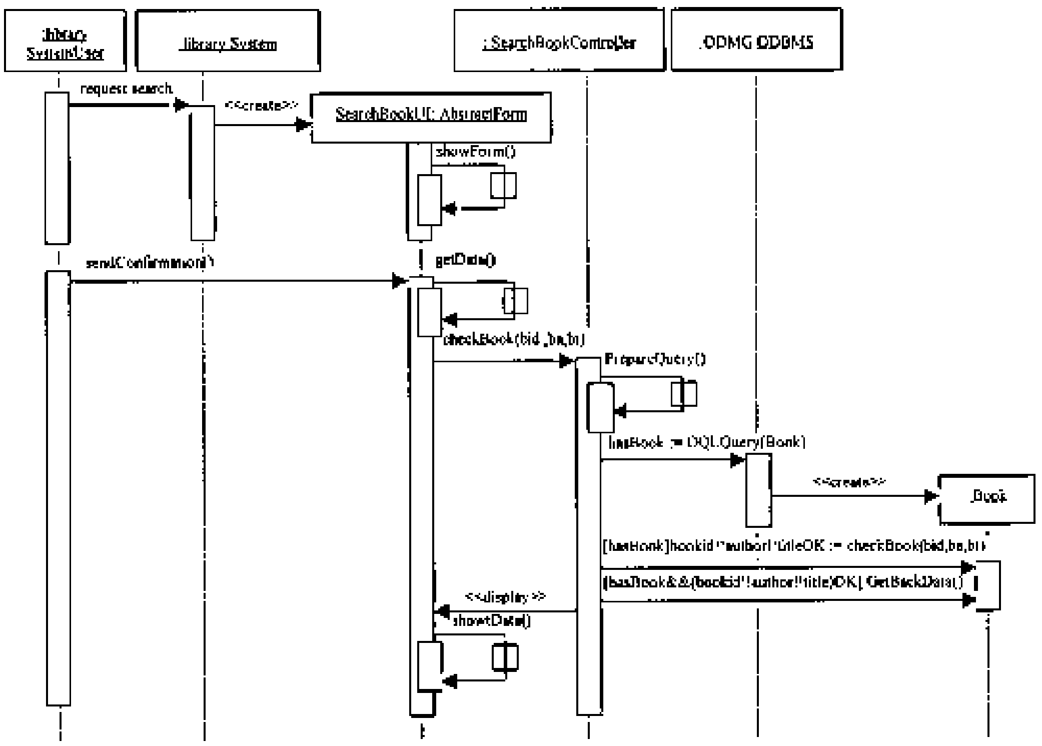


Figure 2.21: A sequence diagram for the *SearchBook* use case.

Returning to the ***BrowseBooks*** sequence diagram in Figure 2.22, the *Library System* creates the *<<boundary>> BrowseBooksUI* object of class *AbstractForm*, which executes the *showForm()* method. This method draws the *BrowseBooksUI* form that presented to the user. Interacting with the UI, the user sends a *sendConfirmation()* message to the *BrowseBooksUI* object. The *sendConfirmation()* message can be an event associated with the Browse button shown in Figure 2.11. The *BrowseBooksUI* object performs a *getData()* operation that picks up the data provided by the user. After collecting the data, the *BrowseBooksUI* object sends a system operation message *checkBook()* to the *<<control>> BrowseBooksController* object, passing the book category, author or title, as parameters. The *BrowseBooksController* object prepares a query that is submitted to a database management system. If there are objects of class *Book* with the provided (book category, author or title) in the database, the database instantiates *Book*. Then, the *<<control>> BrowseBooksController* object sends a message to the *<<entity>> Book* object checking the provided book category, author or title. If there at least one item from the last items is correct or with out them, the *BrowseBooksController* object performs a *getbackData()* operation that get back the data provided by database management system. After collecting the data, the *<<control>> BrowseBooksController* object sends a system operation message *display()* to the *BrowseBooksUI* object . which executes the *showData()* method. This method shows the data on the *BrowseBooksUI* form that presented to the user.

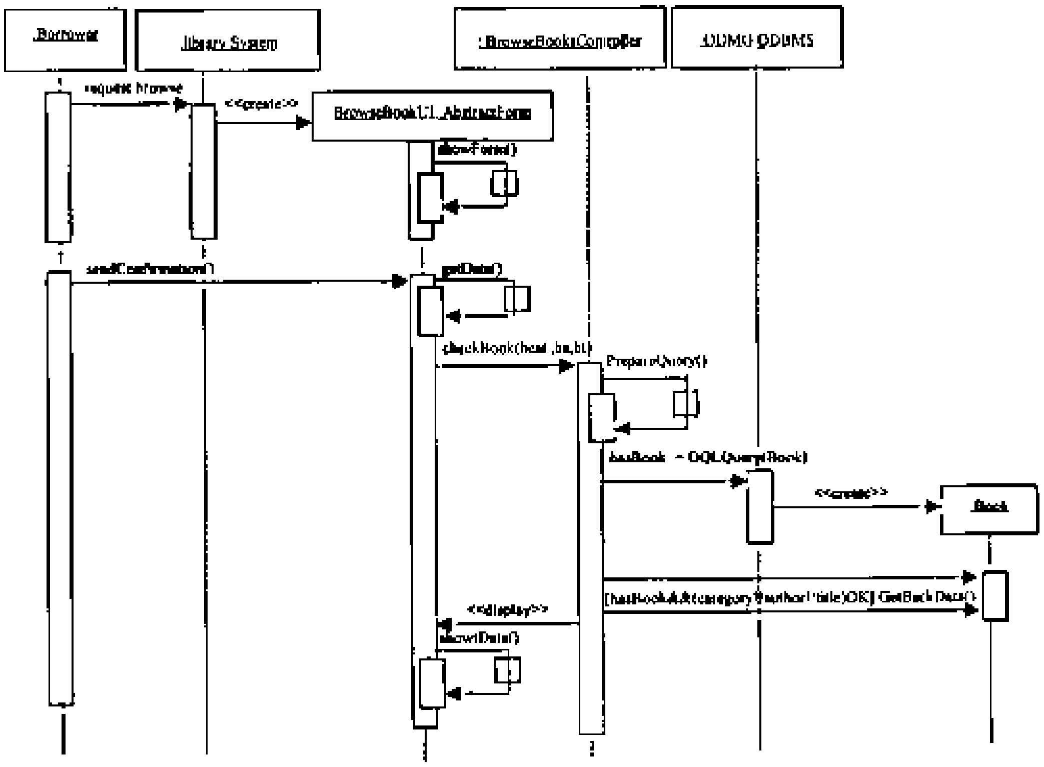


Figure 2.22: A sequence diagram for the *BrowseBook* use case.

From the *AddUser* sequence diagram in Figure 2.23(a), the *Library System* creates the *<<boundary>> ManageUserUI* object of class *AbstractForm*, which executes the *showForm()* method. This method draws the *ManageUserUI* form that is presented to the user. Interacting with the UI, the user sends a *sendConfirmation()* message to the *ManageUserUI* object. The *sendConfirmation()* message can be an event associated with the Enter button shown in Figure 2.12. The *ManageUserUI* object performs a *getData()* operation that takes up the data provided by the user. After collecting the data, the *ManageUserUI* object sends a system operation message *checkUser()* to the *<<control>> ManageUserController* object, passing the user identification as parameter. The *ManageUserController* object prepares a query that is submitted to a database management system. If there are not objects of class *Person* with the provided user identification in the database, the database does not instantiates *Person*. Then, the *<<control>> ManageUserController* object sends an *enable()* message to the *ManageUserUI* object. The *enable()* message makes the other fields active to user information entered.

For the next time the user Interacting with the UI, the user sends a *sendConfirmation()* message to the *ManageUserUI* object. The *sendConfirmation()* message can be an event associated with the Add button shown in Figure 2.12. The *ManageUserUI* object performs a *getData()* operation which for the second time take the data provided by the user. After collecting the data, the *ManageUserUI* object sends a system operation message *checkUser()* to the *<<control>> ManageUserController* object, passing the user identification, name, address and phone number as parameters. The *ManageUserController* object sends an *addUser()* message that is submitted to a database management system. For create new objects of class *Person* with the provided user information in the database, the database instantiates *Person*. Then, the *<<control>> ManageUserController* object sends a *addUser()* message to the *<<entity>> Person* object, that add a new user to the *<<entity>> Person* object. When *addUser()* operation is finished, the *ManageUserController* object creates a new *ManageUserUI* object and destroys the last *ManageUserUI*.

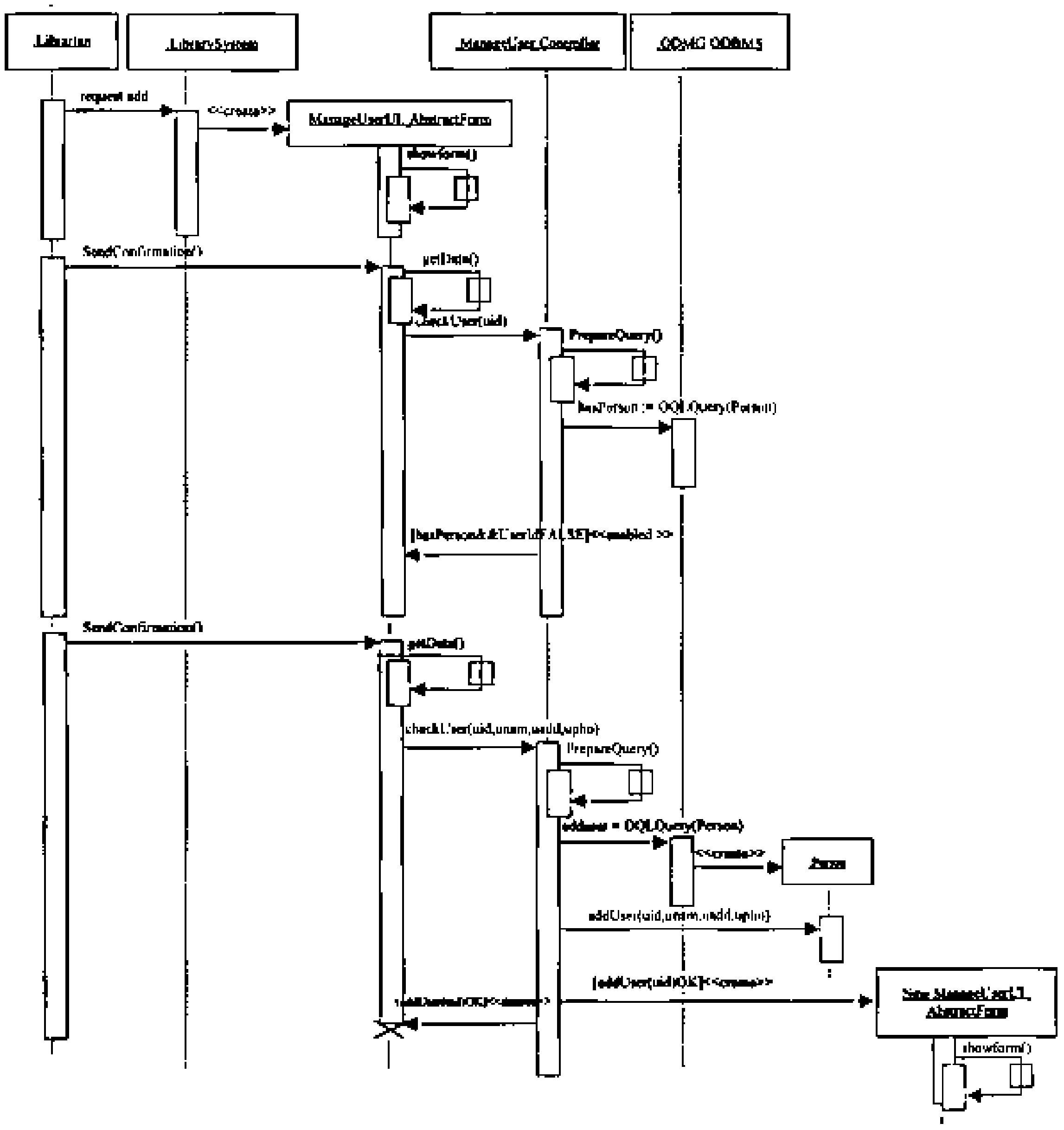
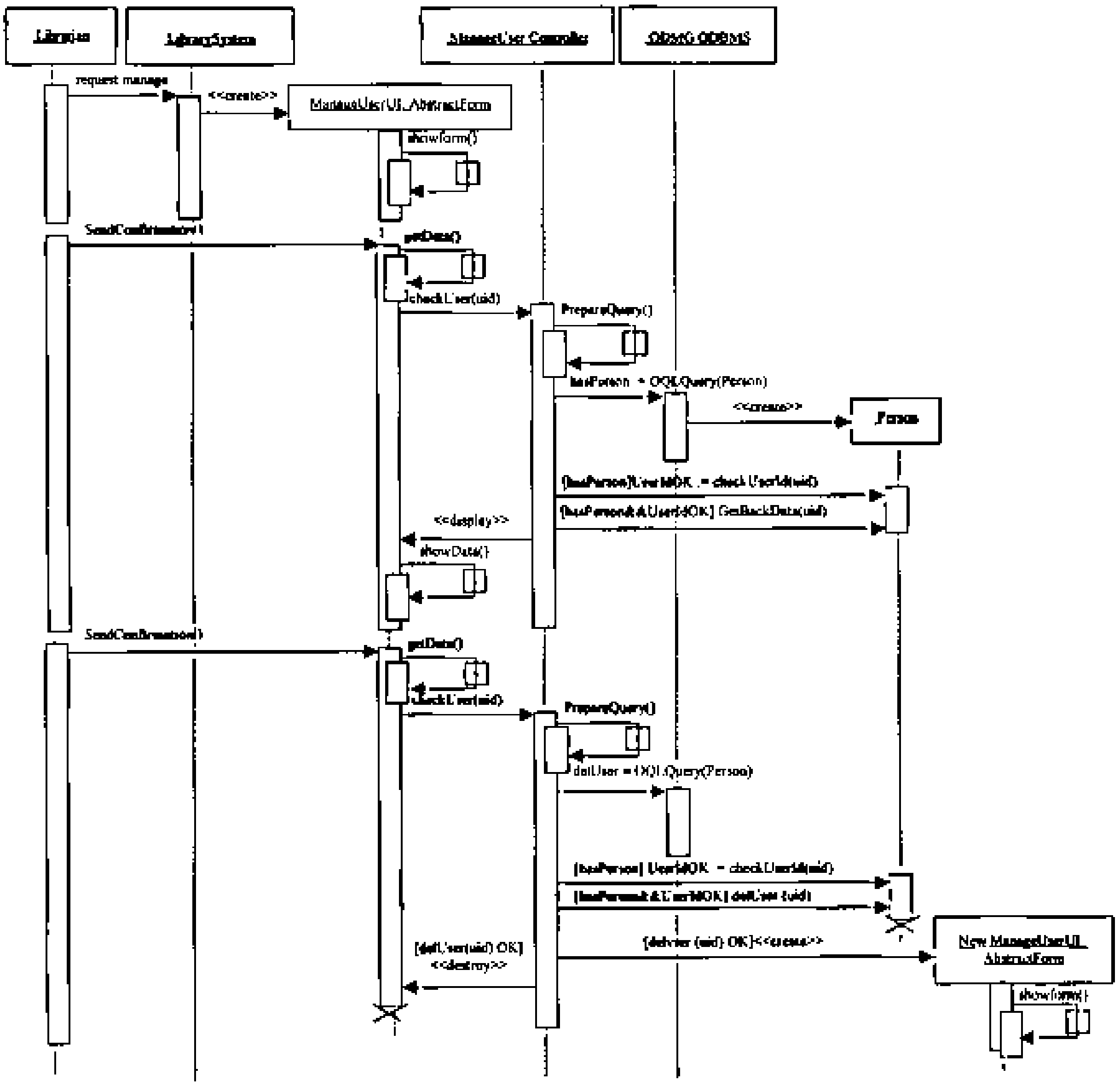

Librarian
LibrarySystem
ManageUser Controller
ODMG ODBMS
request add
<<create>>
ManageUserUI_AbstractForm
showform()
SendConfirmation()
getData()
checkUser(uid)
PrepareQuery()
hasPerson := OQLQuery(Person)
[hasPerson&&UserId!FALSE]<<enabled>>
SendConfirmation()
getData()
checkUser(uid,unam,uadd,upho)
PrepareQuery()
addUser = OQLQuery(Person)
<<create>>
Person
addUser(uid,unam,uadd,upho)
[addUser(uid)OK]<<create>>
New ManageUserUI_AbstractForm
showform()

Figure 2.23(a): A sequence diagram for the *(AddUser)ManageUser* use case.

Returning to the ***DeleteUser*** sequence diagram in Figure 2.23(b), the *Library System* creates the *<<boundary>> ManageUserUI* object of class *AbstractForm*, which executes the *showForm()* method. This method draws the *ManageUserUI* form that is presented to the user. Interacting with the UI the user sends a *sendConfirmation()* message to the *ManageUserUI* object. The *sendConfirmation()* message can be an event associated with the Enter button shown in Figure 2.12. The *ManageUserUI* object performs a *getData()* operation that picks up the data provided by the user. After collecting the data, the *ManageUserUI* object sends a system operation message *checkUser()* to the *<<control>> ManageUserController* object, passing the user identification as parameter. The *ManageUserController* object prepares a query that is submitted to a database management system. If there are objects of class *Person* with the provided user identification in the database, the database instantiates *Person*. Then, the *<<control>> ManageUserController* object sends a message to the *<<entity>> Person* object checking the provided user identification. If the user identification is correct, the *ManageUserController* object sends a *getbacktData()* message that get back the data provided by the *<<entity>> Person* object. After collecting the data, the *<<control>> ManageUserController* object sends a system operation message *display()* to the *ManageUserUI* form object which executes the *showData()* method. This method shows the data (user information) on the *ManageUserUI* form that presented to the user.

For the next time the user Interacting with the UI, the user sends a *sendConfirmation()* message to the *ManageUserUI* object. The *sendConfirmation()* message can be an event associated with the Delete button shown in Figure 2.12. The *ManageUserUI* object performs a *getData()* operation that picks up the data provided From previous operation.. After collecting the data, the *ManageUserUI* object sends a system operation message *checkUser()* to the *<<control>> ManageUserController* object, passing the user identification, name, address and phone number as parameters. The *ManageUserController* object prepares a query that is submitted to a database management system. To restrict object of class *Person* with the provided user information (user identification, name, address and phone number) in the database, the database instantiates *Person*. Then, the *<<control>> ManageUserController* object sends a message to the *<<entity>> Person* object checking the provided user informations. If the user information are correct, the

ManageUserController object performs a *delUser()* operation that remove the user from database. When *delUser()* operation is finished, the *ManageUserController* object creates a new *ManageUserUI* object and destroys the last *ManageUserUI*.



Librarian
LibrarySystem
ManageUser Controller
ODMG ODBMS
request manage
<<create>>
ManageUserUI_AbstractForm
showForm()
SendConfirmation()
getData()
checkUser(uid)
PrepareQuery()
hasPerson = OQLQuery(Person)
<<create>>
Person
[hasPerson]UserIdOK = checkUserId(uid)
[hasPerson&&UserIdOK] GetBackData(uid)
<<display>>
showData()
SendConfirmation()
getData()
checkUser(uid)
PrepareQuery()
delUser = OQLQuery(Person)
[hasPerson] UserIdOK = checkUserId(uid)
[hasPerson&&UserIdOK] delUser (uid)
[delUser(uid) OK]
<<destroy>>
[delUser (uid) OK]<<create>>
New ManageUserUI_AbstractForm
showForm()

Figure 2.23(b): A sequence diagram for the *(DeleteUser) ManageUser* use case.

The presented sequence diagrams are restricted to the scenario where the user successfully in logs into the system, regular Loan, successfully search book, successfully browse books and regular manage user. Unsuccessful attempts to log into the system, irregular borrow book scenario, Unsuccessful search book and Unsuccessful Browse books scenario can be modelled as described in Section 2.8.2.

Activities, as presented in Section 2.3, abstract presentation models are weakly connected by the flow objects in the activity diagrams. Indeed, *AbstractComponents* should be used in activity diagrams to explain the data flow between the UI and the underlying application. However, we believe that a well-defined relationship between activities and instances of *AbstractForms* can facilitate the design of tasks and abstract presentation. For instance, activities that involve user interactions should be supported by *<<boundary>>* objects. However, it is difficult to identify *<<boundary>>* objects from an activity or to identify activities from <<boundary>> objects.

2.8 Concrete Presentation Modelling

A user interacts with a system through interaction objects. Interaction objects are commonly classified as either abstract or concrete. But:

- The abstract presentation model does not give much of a feel for the functionality or organization of the event for components user interface associated with operation of *<<control>>* classes.
- The abstract presentation model can not give any description of layout, and can not describe what is component that formed for *<<boundary>>* class.

Because the abstract presentation model can not cover these aspects, so we should use *Concrete presentation model.*

2.8.1 Concrete Presentation Structure and Layout

A concrete interaction object, any widget, is a physical implementation of an abstract interaction object. Most interface builders (such as Microsoft Visual C++) provide facilities for interactively selecting and placing concrete interaction objects during interface development.

Figure 2.13 shows an abstract presentation model, in this model <<apm>>AbstractComponent Class represents different functions that typify those supported by concrete interaction objects in widely available widget sets, and <<apm>>AbstractContainer represents the grouping of components and containers in an interface. We can represent a corresponding concrete presentation model as UML classes. We might reverse-engineer a UML class diagram from an existing object-oriented widget set, such as Java Swing [6], as a practical option.

We could then allocate concrete widgets to support abstract components with a UML framework, as described in Figure 2.24. In Figure 2.24, the class diagram in Figure 2.13 is the specification of the pattern for the UI presentation model. This pattern called ***PresentationFramework*** collaboration. We can bind concrete classes to abstract ones using this *PresentationFramework*, where the *<<cpm>> Frame* is bound to the *<<apm>> AbstractForm*, the *<<cpm>> Container* is bound to the *<<apm>> AbstractContainer*, the *<<cpm>> Label* is bound to the *<<apm>> StaticDisplay*, the *<<cpm>> TextField* is bound to the *<<apm>> InteractionControl*, the *<<cpm>> Button* is bound to the *<<apm>> ActionInvoker* and the *<<cpm>> Combo Box* is bound to the *<<apm>> ActionInvoker*.

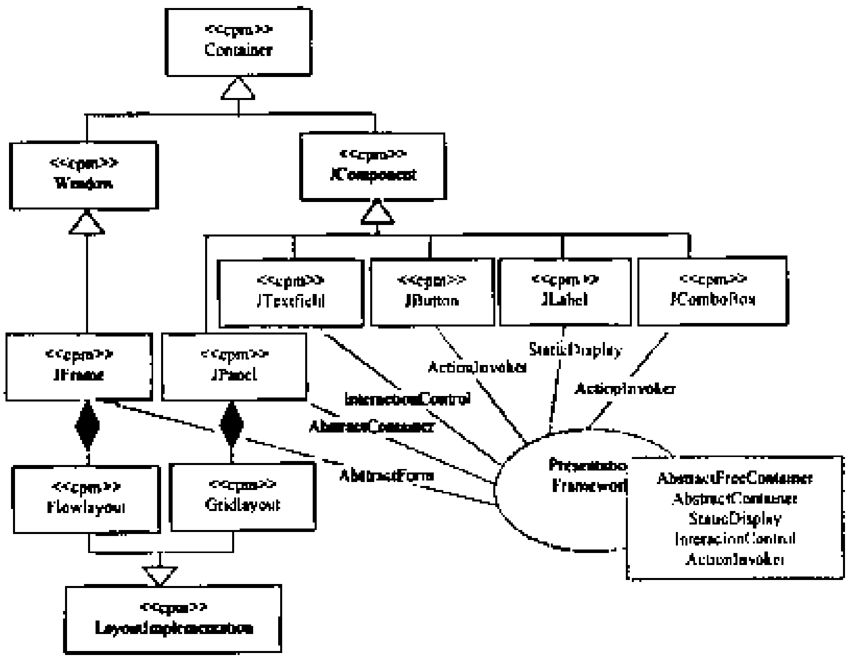


Figure 2.24: The concrete presentation model.

We notice from the Figure 2.24:

- What is required to model the case study?
- Give us description of how the layout will be.
- Every class dealed as a *Container* so it must be an instance of *LayoutImplementation*.

UML also provides specialized visualizations for abstract presentation models. However, UML interface diagrams are essentially UML class diagrams that clarify the purpose of individual abstract components and the containment relationships between different components.

However, specifying specific concrete interaction object placement is very much an implementation activity. And specifying an interface using concrete interaction objects risks premature commitment to a specific look and feel.

Figure 2.25 presents the concrete presentation model for the *LoginUI*, presented in Figures 2.8. The model is an object diagram where the links are: the **compose** and **integrate** links introduced in Section 2.7.1, and the **organise** link that relates instances of *Frame* (playing the role of the *AbstractContainer*) with their respective instances of *LayoutImplementation*. This link is mandatory for each instance of *Frame*.

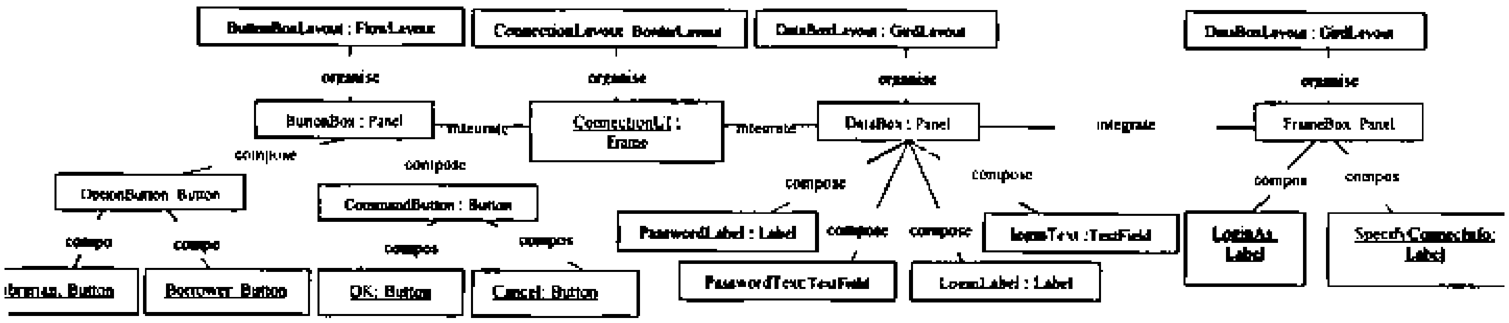


Figure 2.25: The concrete presentation model of the *loginUI*

Figures 2.26 is an object diagram of the concrete presentation models for the *BorrowBookUI* presented in Figures 2.9.

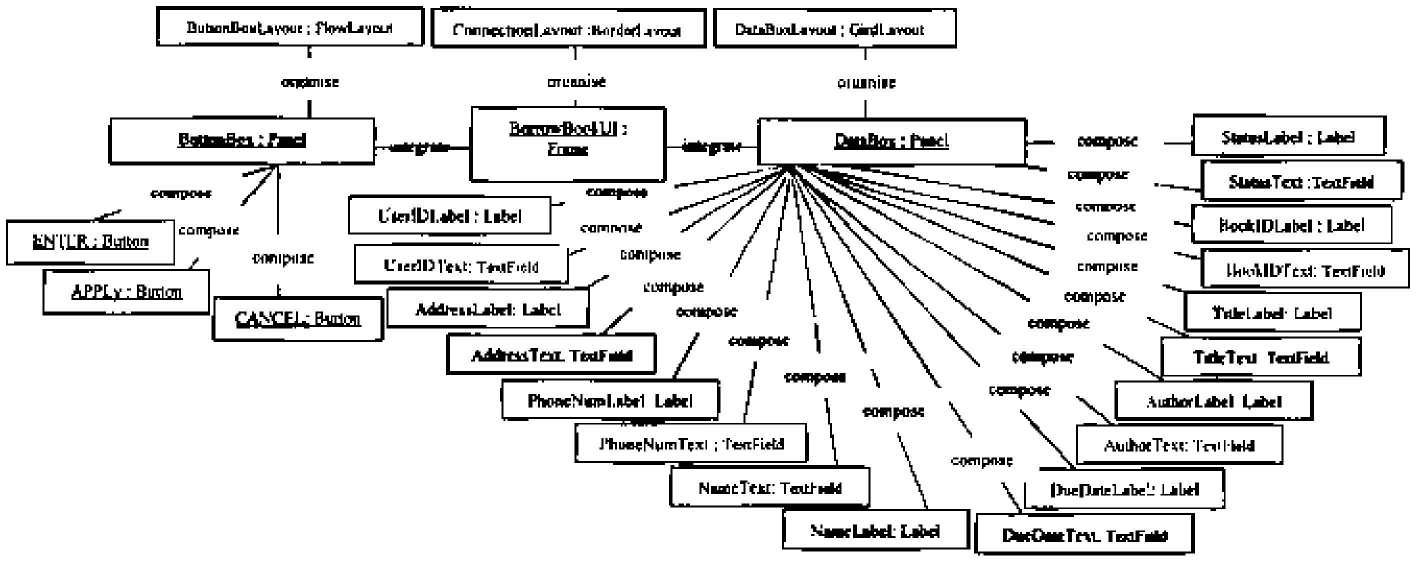


Figure 2.26: The concrete presentation model of the *BorrowBookUI*

Figures 2.27 is an object diagram of the concrete presentation models for the *SearchBookUI* presented in Figures 2.10.

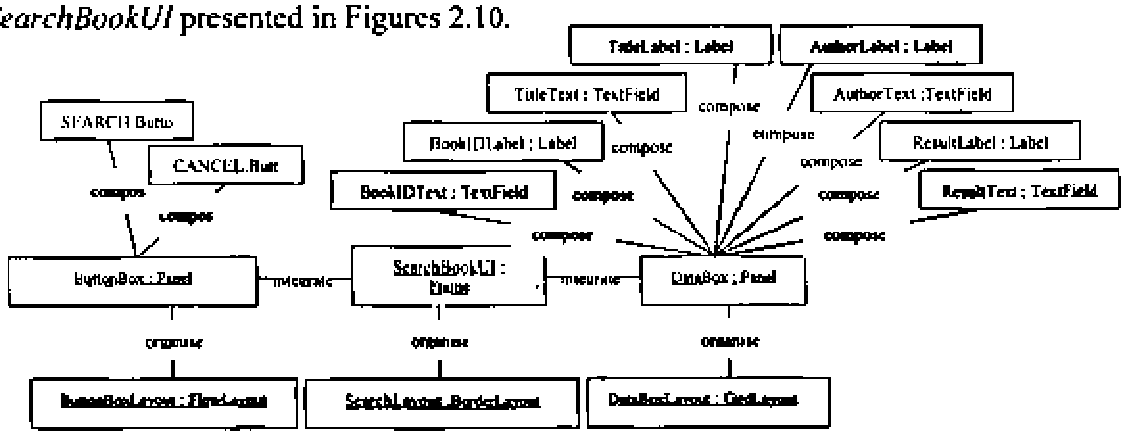


Figure 2.27: The concrete presentation model of the *SearchUserUI*

Figures 2.28 is an object diagram of the concrete presentation models for the *ManageUserUI* presented in Figures 2.12.

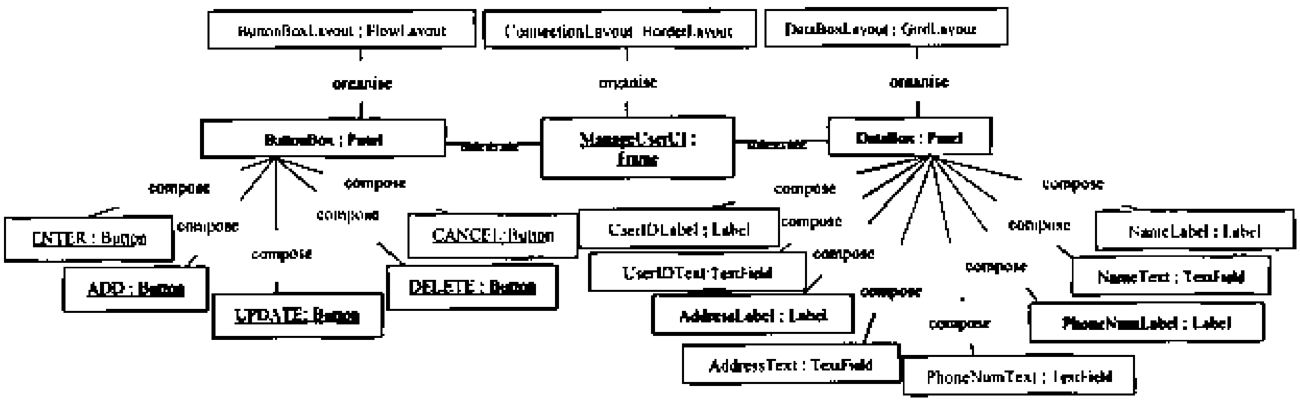


Figure 2.28: The concrete presentation model of the *ManageUserUI*

Further, Figures 2.25, 2.26, 2.27, 2.28 shows that *Panels* are being used instead of *Containers* to model non top-level containers. This is possible, since the subclasses of the bound to classes can also be considered as part of the concrete presentation model.

2.8.2 Concrete Presentation Behaviour

We can represent important features of both concrete and abstract presentations using standard UML class diagrams. Additionally, we can describe interaction objects' associated behavior using standard UML sequence or activity diagrams.

In section 2.7.3 presented the sequence diagrams for the *ConnectToSystem, BorrowBook, SearchBook, BrowseBook* and *ManageUser* use cases are restricted to the scenario where the user successfully in logs into the system, regular Loan, successfully search book.

successfully browse books and regular manage user. The scenario for these use cases where the CANCEL button is pressed are going to clarify in this section by using *Concrete Presentation Behaviour*.

The Figures 2.25, 2.26, 2.27, 2.28 we have noticed that, these Figures have a number of components, which have a special behaviour. to discuss the behaviour of concrete presentation model, we have to know each event associated with each component (<<boundary>> object). The idea here is, every *<<boundary>>* object (component) is an instance of *Frame* and has operations that have events associated with them, in which, every message sent by an actor to an *<<boundary>>* object represents an event associated with an UI component.

Figure 2.29[1] shows the sequence diagram for the *BorrowBook* use case where the *CANCEL* button is a <<boundary>> object (component) in the *BorrowBookUI: Frame* that is a *Concrete Presentation Model*. There for, this button has an event associated with it, represented as *CancelPressed* message, that triggered when the *Actor* press the button CANCEL. *CancelPressed* message is execute the operation *sendcancellation()* between the library system and the user.

And it was advised to deal with the other use cases for modelling a scenario where the *CANCEL* button is pressed, with a difference in the frame name.

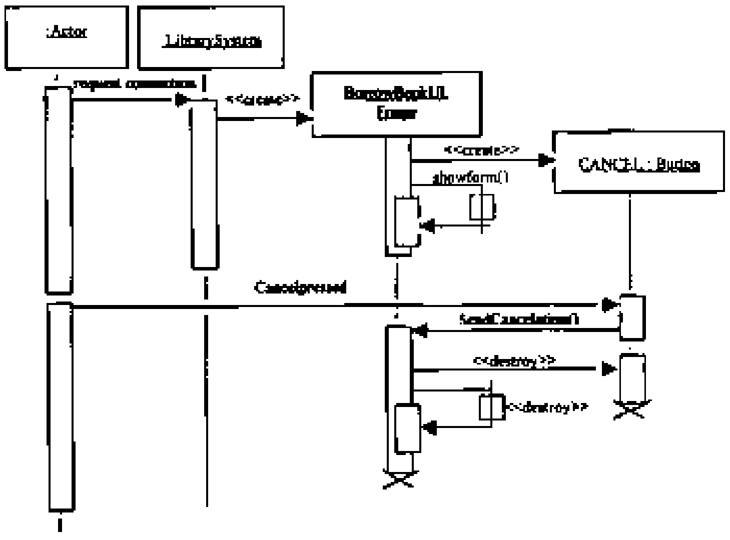


Figure 2.29: A second sequence diagram for the *BorrowBook* use case.

[1] The Figure is borrowed from [4], with some changes.

2.9 Packaging the Application

The package diagram shows dependencies between various components of the system. Figure 2.30 shows a package diagram that presents an overview to the whole system.

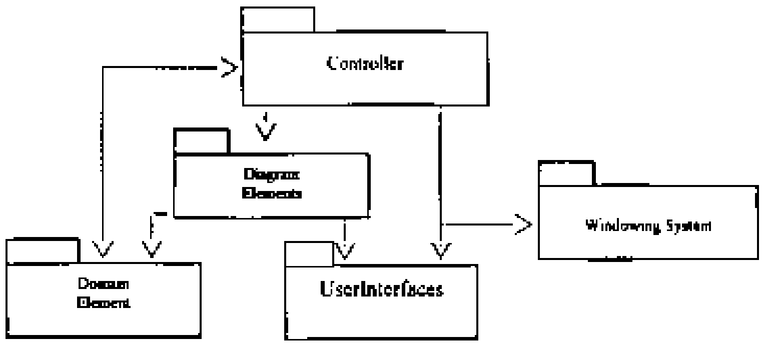


Figure 2.30: The package diagram of the Library System.

The classes and class instances are grouped into five packages, as follows:-

- The ***Windowing System*** composed of those classes used to build the user interface. The environment could be an object-oriented programming language. We have used java language as the environment of this research work.
- The ***User Interface*** package composed of <<boundary>> classes and objects.
- The ***Domain Model*** package composed of <<entity>> classes. The class diagram of these classes forms the domain model in Figure 2.2.
- The ***Control*** package composed of <<control>> classes. These classes are presented in sequence diagrams that shown in Figures 2.19, 2.20, 2.21, 2.22, 2.23(a), and 2.23(b).
- The ***Element Diagram*** package composed of <<*apm*>> classes as shown in Figure 2.13, and the <<*cpm*>> classes of the *Presentation Framework* pattern.

Chapter Three

Generating User Interface Prototypes from Scenarios

Over the past years, scenarios have received significant attention and have been used for different purposes such as human computer interaction analysis [23], specification generation [1], Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) [3, 15, 24], and requirements engineering [13]. A typical process for requirements engineering based on scenarios [13] has two main tasks. The ***first task*** consists of generating from scenarios specifications that describe system behavior. The ***second task*** concerns scenario validation with users by simulation and prototyping.

For the purpose of validation in early development stages, prototyping tools are commonly and widely used. Recently, many advances have been made in User Interface (UI) prototyping tools like UI builders and UI management systems. Yet, the development of UIs is still time-consuming, since every UI object has to be created and laid out explicitly. Also, specifications of dialogue controls must be added by programming (for UI builders) or via a specialized language (for UI management systems).

In this chapter, we suggest an approach for requirements engineering supporting the Unified Modeling Language (UML). The approach provides a five activities process with limited manual intervention for deriving a prototype of the UI from scenarios and generating a formal specification of the application. Scenarios are acquired in the form of UML collaboration diagrams and enriched with UI information. These diagrams are automatically transformed, based on previous work [8] and [13] into the UML Statechart specifications of all the objects involved. The prototype is embedded in a UI builder environment for further refinement.

Section 1 of this chapter gives a brief overview of the UML diagrams relevant for our work. Section 2 presents the five activities of our approach. Section 3 describes in detail the fifth of these activities.

3.1 Unified Modeling Language

The UML provides a syntactic notation to describe all major views of a system using different kinds of diagrams. In this section, we introduce the UML diagrams that are relevant for our approach: Collaboration diagram (CollD), and Statechart diagram (StateD).

3.1.1 Collaboration diagram (CollD)

A scenario shows a particular series of interactions among objects in a single execution of a use case of a system. Scenarios can be viewed in two different ways: through sequence diagrams (SequenceDs) or CollDs. Both types of diagrams rely on the same underlying semantics, and conversion from one to the other is possible. For this work, we chose to use CollDs because the UML documentation defines them more precisely than sequenceDs. For a complete definition of CollDs refer to [12].

Figures 3.1(a), 3.1(b), and 3.1(c) depict three scenarios (CollDs) of the use case *ConnectToSystem*. Figure 3.1(a) represents the scenario where a user successfully logging into the library system, Figure 3.1(b) represents the case where the *LoginToSystem* is canceled, and Figure 3.1(c) shows the scenario where the user is not registered yet in the system.

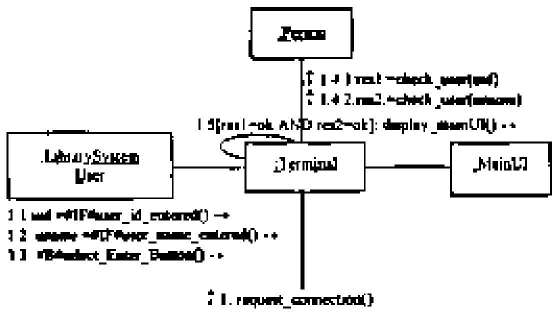

:Person
1.4.1 res1=check_user(uid)
1.4.2 res2=check_user(uname)
1.5[res1=ok AND res2=ok]: display_mainUI()
:LibrarySystem User
:Terminal
:MainUI
1.1 uid=#IF#user_id_entered()
1.2 uname=#IF#user_name_entered()
1.3 #B#select_Enter_Button()
1. request_connection()

Figure 3.1(a): Scenario
successfullyLoginToSystem

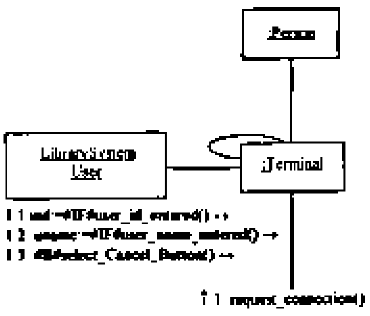

:Person
:LibrarySystem User
:Terminal
1.1 uid=#IF#user_id_entered()
1.2 uname=#IF#user_name_entered()
1.3 #B#select_Cancel_Button()
1 request_connection()

Figure 3.1(b): Scenario
CancelLoginToSystem

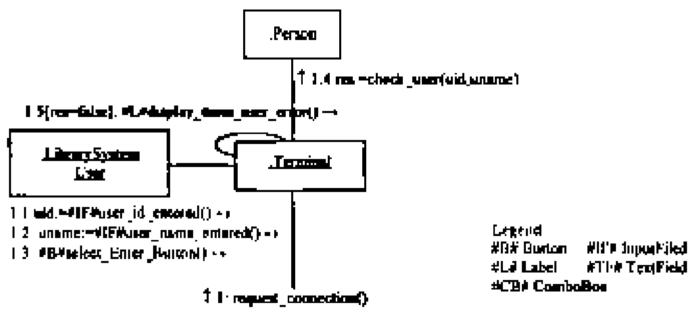

.Person
↑ 1.4 res:=check_user(uid,uname)
1.5[res=false]: #L#display_error_user_error() →
Library System User
.Terminal
1.1 uid:=#IF#user_id_entered() →
1.2 uname:=#IF#user_name_entered() →
1.3 #B#select_Enter_Button() →
↑ 1: request_connection()
Legend
#B# Button
#IF# InputFiled
#L# Label
#TF# TextField
#CB# ComboBox

Figure 3.1(c): Scenario *errorLoginToSystem*

Figures 3.2(a), 3.2(b), and 3.2(c) depict three scenarios (CollDs) of the use case ***BorrowBook***. Figure 3.2(a) represents the scenario where the borrow book is correctly registered, Figure 3.2(b) represents the case where the borrow book is canceled, and Figure 3.2(c) shows the scenario where the user is not registered yet in the system.

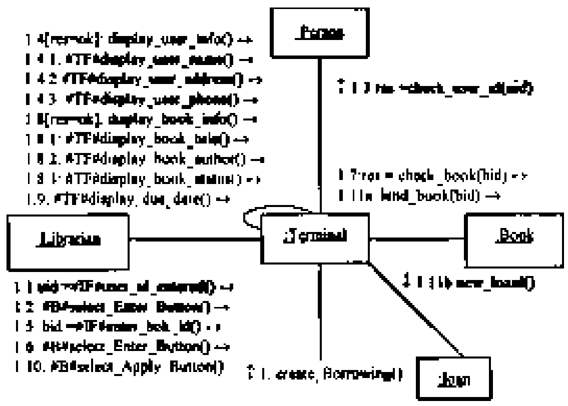
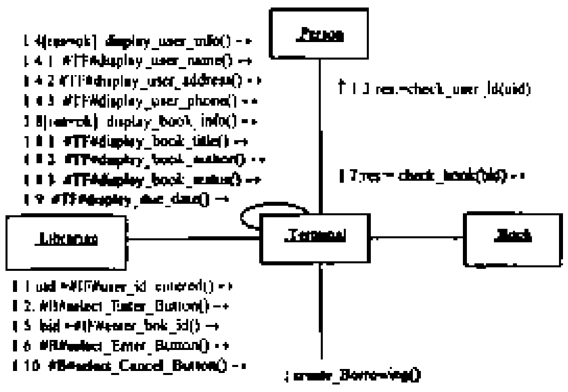

Person
1.4[res=ok]: display_user_info() →
1.4.1. #TF#display_user_name() →
1.4.2 #TF#display_user_address() →
1.4.3 #TF#display_user_phone() →
1.8[res=ok]: display_book_info() →
1.8.1: #TF#display_book_title() →
1.8.2. #TF#display_book_author() →
1.8.3: #TF#display_book_status() →
1.9. #TF#display_due_date() →
↑ 1.3 res:=check_user_id(uid)
1.7 res := check_book(bid) →
1.11a: bind_book(bid) →
Librarian
:Terminal
Book
1.1 uid:=#IF#user_id_entered() →
1.2 #B#select_Enter_Button() →
1.5 bid:=#IF#enter_bok_id() →
1.6 #B#select_Enter_Button() →
1.10. #B#select_Apply_Button()
↑ 1. create_Borrowing()
↓ 1.11b: new_loan()
:loan

Figure 3.2(a): Scenario *RegularBorrowBook*


Person
1.4[res=ok]: display_user_info() →
1.4.1 #TF#display_user_name() →
1.4.2 #TF#display_user_address() →
1.4.3 #TF#display_user_phone() →
1.8[res=ok]: display_book_info() →
1.8.1 #TF#display_book_title() →
1.8.2 #TF#display_book_author() →
1.8.3 #TF#display_book_status() →
1.9 #TF#display_due_date() →
↑ 1.3 res:=check_user_id(uid)
1.7 res:= check_book(bid) →
Librarian
Terminal
Book
1.1 uid:=#IF#user_id_entered() →
1.2. #B#select_Enter_Button() →
1.5 bid:=#IF#enter_bok_id() →
1.6 #B#select_Enter_Button() →
1.10 #B#select_Cancel_Button() →
create_Borrowing()

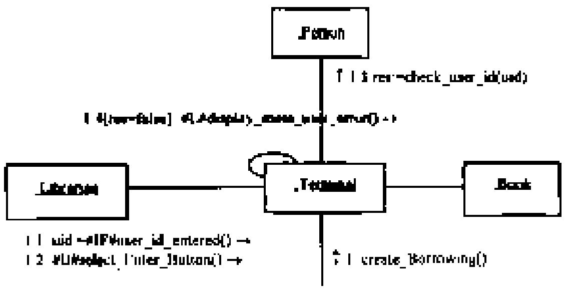

Person
↑ 1.3 res:=check_user_id(uid)
1.4[res=false] #L#display_error_user_error() →
Librarian
.Terminal
Book
1.1 uid:=#IF#user_id_entered() →
1.2 #B#select_Enter_Button() →
↑ 1: create_Borrowing()

Figure 3.2(b): Scenario *CancelBorrowBook*

Figure 3.2(c):Scenario *errorBorrowBook.*

Figures 3.3(a), 3.3(b) and 3.3(c) depict three scenarios (CollDs) of the use case ***ManageUser (Add User)***. Figure 3.3(a) represents the scenario where an add user is correctly registered, Figure 3.3(b) represents the case where the add user is canceled , and Figure 3.3(c) shows the scenario where the add user is error.

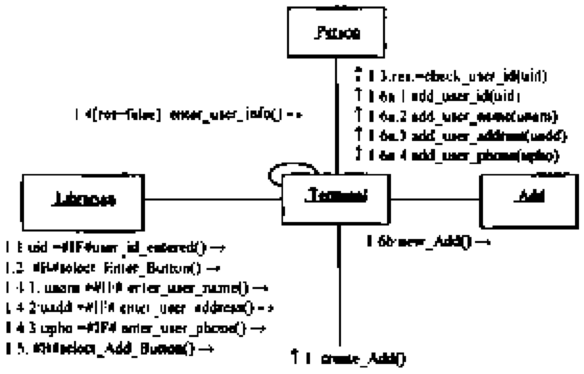

Person
1.3.res:=check_user_id(uid)
1.6a.1 add_user_id(uid)
1.6a.2 add_user_name(unam)
1.6a.3 add_user_address(uadd)
1.6a.4 add_user_phone(upho)
1.4[res=false] enter_user_info() →
Librarian
Terminal
Add
1.6b new_Add() →
1.1 uid =#IF#user_id_entered() →
1.2 #B#select_Enter_Button() →
1.4.1. unam =#IF# enter_user_name() →
1.4.2.uadd =#IF# enter_user_address() →
1.4.3 upho =#IF# enter_user_phone() →
1.5. #B#select_Add_Button() →
1. create_Add()

Figure 3.3(a): Scenario *RegulaAddUser*

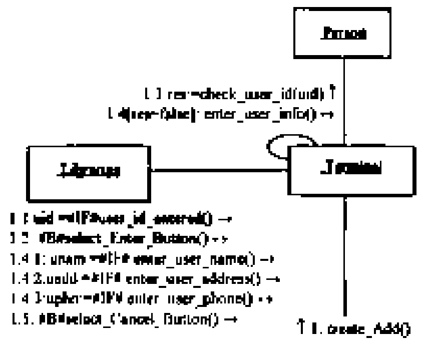

Person
1.3 res:=check_user_id(uid)
1.4[res=false]: enter_user_info() →
Librarian
Terminal
1.1 uid =#IF#user_id_entered() →
1.2 #B#select_Enter_Button() →
1.4.1: unam =#IF# enter_user_name() →
1.4.2.uadd =#IF# enter_user_address() →
1.4.3 upho =#IF# enter_user_phone() →
1.5. #B#select_Cancel_Button() →
1. create_Add()

Figure 3.3(b): Scenario *CancelAddUser*

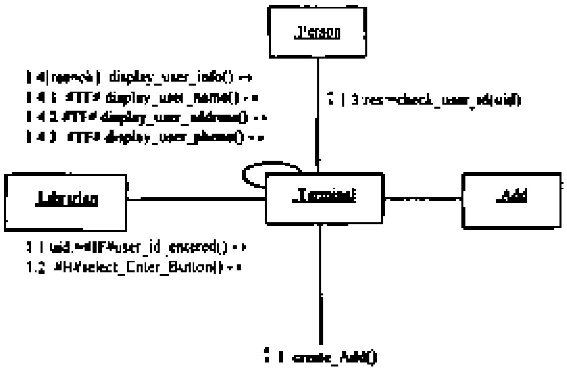

Person
1.4[res=ok] display_user_info() →
1.4.1 #TF# display_user_name() →
1.4.2 #TF# display_user_address() →
1.4.3 #TF# display_user_phone() →
1.3 res:=check_user_id(uid)
Librarian
Terminal
Add
1.1 uid:=#IF#user_id_entered() →
1.2 #B#select_Enter_Button() →
1 create_Add()

Figure 3.3(c): Scenario *ErrorAddUser*

Figures 3.4(a), 3.4(b) and 3.4(c) depict three scenarios (CollDs) of the use case ***ManageUser (DeleteUser)***. Figure 3.4(a) represents the scenario where a delete user is correctly. Figure 3.4(b) represents the case where the delete user is canceled. and Figure 3.4(c) shows the scenario where the delete user is error.

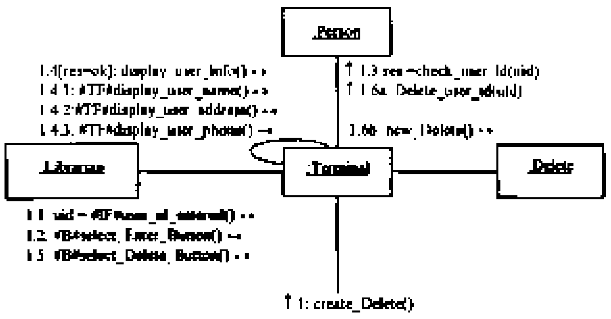


Figure 3.4(a): Scenario *RegularDeleteUser*

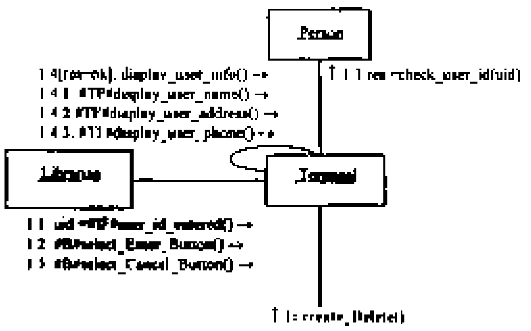


Figure 3.4(b): Scenario *CancelDeleteUser*

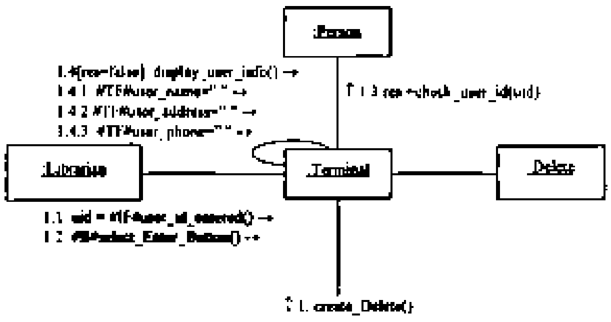


Figure 3.4(c): Scenario *errorDeleteUser*

Figures 3.5(a), 3.5(b), and 3.5(c) depict three scenarios (CollDs) of the use case ***SearchBook***. Figure 3.5(a) represents the scenario where the search book is successfully, Figure 3.5(b) represents the case where the search book is canceled, and Figure 3.5(c) shows the scenario where the search book is failer.

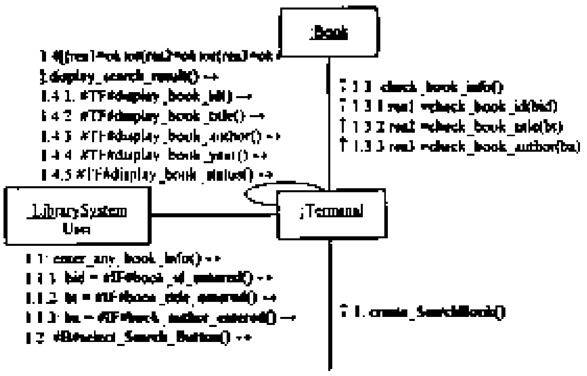


Figure 3.5(a): Scenario *successfullySearchBook*

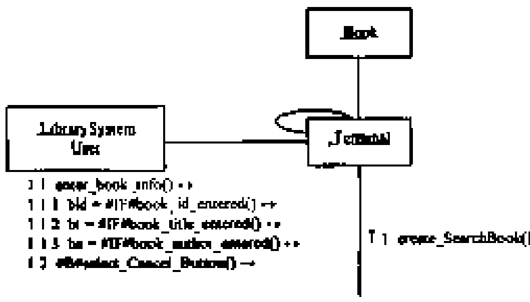


Figure3.5(b): Scenario *CancelSearchBook*

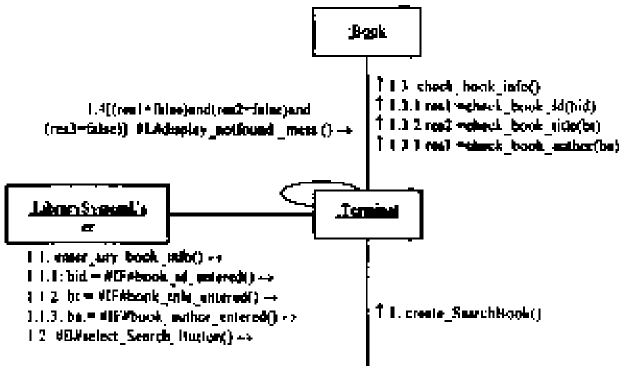

:Book
1.3. check_book_info()
1.3.1 res1 =check_book_id(bid)
1.3.2 res2 =check_book_title(bt)
1.3.3 res3 =check_book_author(ba)
1.4[(res1=false)and(res2=false)and (res3=false)] #LA#display_notfound_mess () →
LibrarySystemUser
:Terminal
1.1. enter_any_book_info() →
1.1.1: bid = #TF#book_id_entered() →
1.1.2. bt = #TF#book_title_entered() →
1.1.3. ba = #TF#book_author_entered() →
1.2. #B#select_Search_Button() →
1. create_SearchBook()

Figure 3.5(c): Scenario *failerSearchBook*

Figures 3.6(a), 3.6(b) and 3.6(c) depict tree scenarios (CollDs) of the use case ***BrowseBook***. Figure 3.6(a) represents the scenario where the browse book is successfully, Figure 3.6(b) represents the case where the browse book is canceled, and Figure 3.6(c) shows the scenario where the browse book is failer.

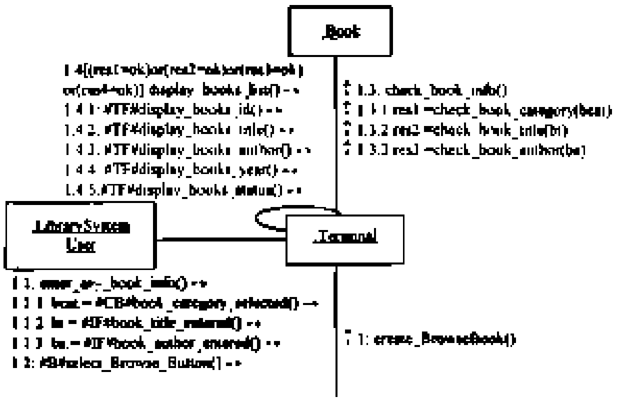

Book
1.4[(res1=ok)or(res2=ok)or(res3=ok) or(res4=ok)] display_books_list() →
1.4.1. #TF#display_books_id() →
1.4.2. #TF#display_books_title() →
1.4.3. #TF#display_books_author() →
1.4.4. #TF#display_books_year() →
1.4.5. #TF#display_books_status() →
1.3. check_book_info()
1.3.1 res1 =check_book_category(bcat)
1.3.2 res2 =check_book_title(bt)
1.3.3 res3 =check_book_author(ba)
LibrarySystem User
:Terminal
1.1. enter_any_book_info() →
1.1.1 bcat = #CB#book_category_selected() →
1.1.2 bt = #TF#book_title_entered() →
1.1.3 ba = #TF#book_author_entered() →
1.2: #B#select_Browse_Button() →
1: create_BrowseBook()

Figure 3.6(a): Scenario *succefullyBrowseBooks*

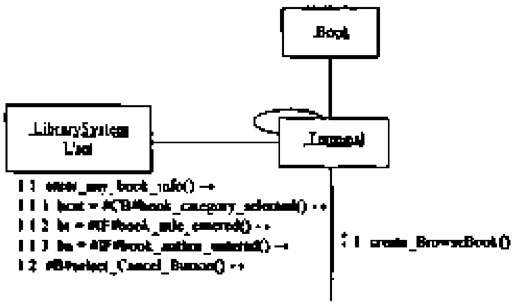

Book
LibrarySystem User
:Terminal
1.1 enter_any_book_info() →
1.1.1 bcat = #CB#book_category_selected() →
1.1.2 bt = #TF#book_title_entered() →
1.1.3 ba = #TF#book_author_entered() →
1.2 #B#select_Cancel_Button() →
1 create_BrowseBook()

Figure3.6(b):Scenario *CancelBrowseBooks*

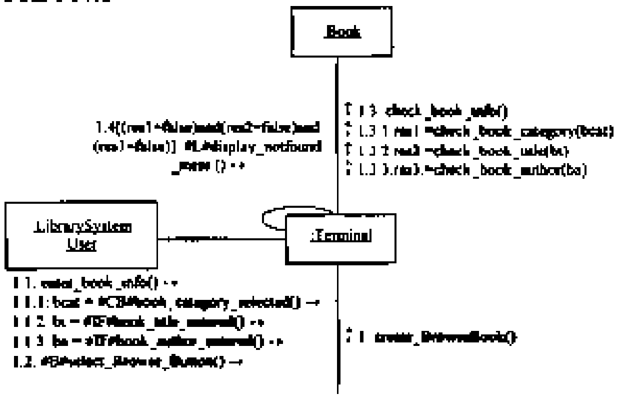

Book
1.3. check_book_info()
1.3.1 res1 =check_book_category(bcat)
1.3.2 res2 =check_book_title(bt)
1.3.3 res3 =check_book_author(ba)
1.4[(res1=false)and(res2=false)and (res3=false)] #L#display_notfound _mess () →
LibrarySystem User
:Terminal
1.1. enter_book_info() →
1.1.1: bcat = #CB#book_category_selected() →
1.1.2. bt = #TF#book_title_entered() →
1.1.3. ba = #TF#book_author_entered() →
1.2. #B#select_Browse_Button() →
1 create_BrowseBook()

Figure 3.6(c): Scenario *failerBrowseBooks*

3.1.2 Statechart diagram (StateD)

A StateD shows the sequence of states that an object goes through during its life cycle in response to stimuli. Generally, a StateD may be attached to a class of objects with an interesting dynamic behavior.

The formalism (notation and semantics) used in StateDs is derived from Statecharts as defined by Harel [10]. Any state in a StateD can be recursively decomposed into exclusive states (*or-state*) or concurrent states *(and-state)*. When a transition in a Statechart is triggered ''event receive and guard condition tested'', the object leaves its current state, initiates the action(s) for that transition and enters a new state. Transitions between concurrent states are not allowed, but synchronization and information exchange are possible through events. As an illustration, Figure 3.9 depicts the StateD of the object *Terminal*. The state *waitingForApplyOrCancel*, is an *and-state* composed of two concurrent sub states separated by a dashed line.

3.2 Description of the Approach

In this section, we describe the overall approach to derive a UI prototype from scenarios using the UML artifacts. We aim to provide a process that bridges two iterative software processes: the formal specification process as illustrated at the top of figure 3.7, and the UI prototyping process at the bottom of the figure 3.7.[1]

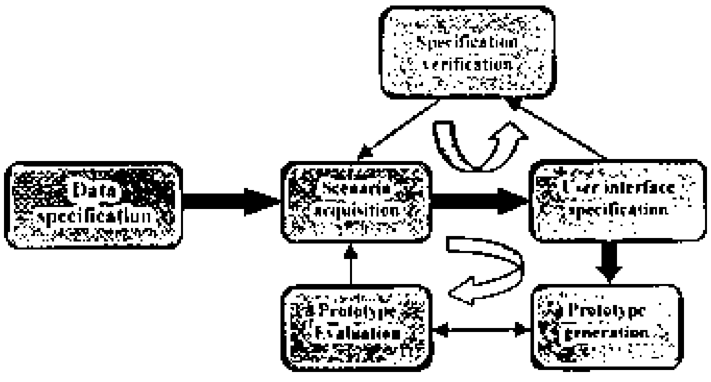


Figure 3.7: View of the overall process combining formal specification and UI prototyping

[1] The figure is borrowed from [11]

Data specification (see Figure 3.7) are captured in a detailed ClassD which shows structural relationships between classes, and specifies class attributes and method together with pre-and postconditions. This information is used for ***scenario acquisition*** via CollDs, and for ***prototype generation*** to enhance the visual aspect of the generated prototypes. ***User interface specifications*** are derived from scenario descriptions, and are used for both generation of UI prototypes and for ***specification verification*** (verifying coherence and completeness of the UI specification). The generated prototypes are ***evaluated*** with end users to validate the users' needs.

In this chapter, we focus on the UI prototyping process, essentially on the transformations represented by the bold arrows in Figure 3.7. This process can be decomposed into five activities (see Figure 3.8) which are detailed below:

- Requirements acquisition (section 3.2.1)
- Generation of partial specifications from scenarios (section 3.2.2)
- Analysis of partial specifications (section 3.2.3)
- sIntegration of partial specifications (section 3.2.4)
- User interface prototype generation (section 3.2.5).

3.2.1 Requirements acquisition

Scenario modeling is the key technique mostly used in this activity. It is used in object-oriented methodologies [3, 15 and 24] as an approach to requirements engineering. The UML proposes a suitable framework for scenario acquisition using UsecaseDs for capturing system functionalities and CollDs for describing scenarios.

In this activity, the analyst first elaborates the UsecaseD of the system (see Figure 2.1). Then, we acquires scenarios as CollDs for each use case in the UsecaseD. for instance Figures 3.1(a), 3.1(b), and 3.1(c) show for example, the three sample CollDs corresponding to the use case *LoginToSystem* of the library system.

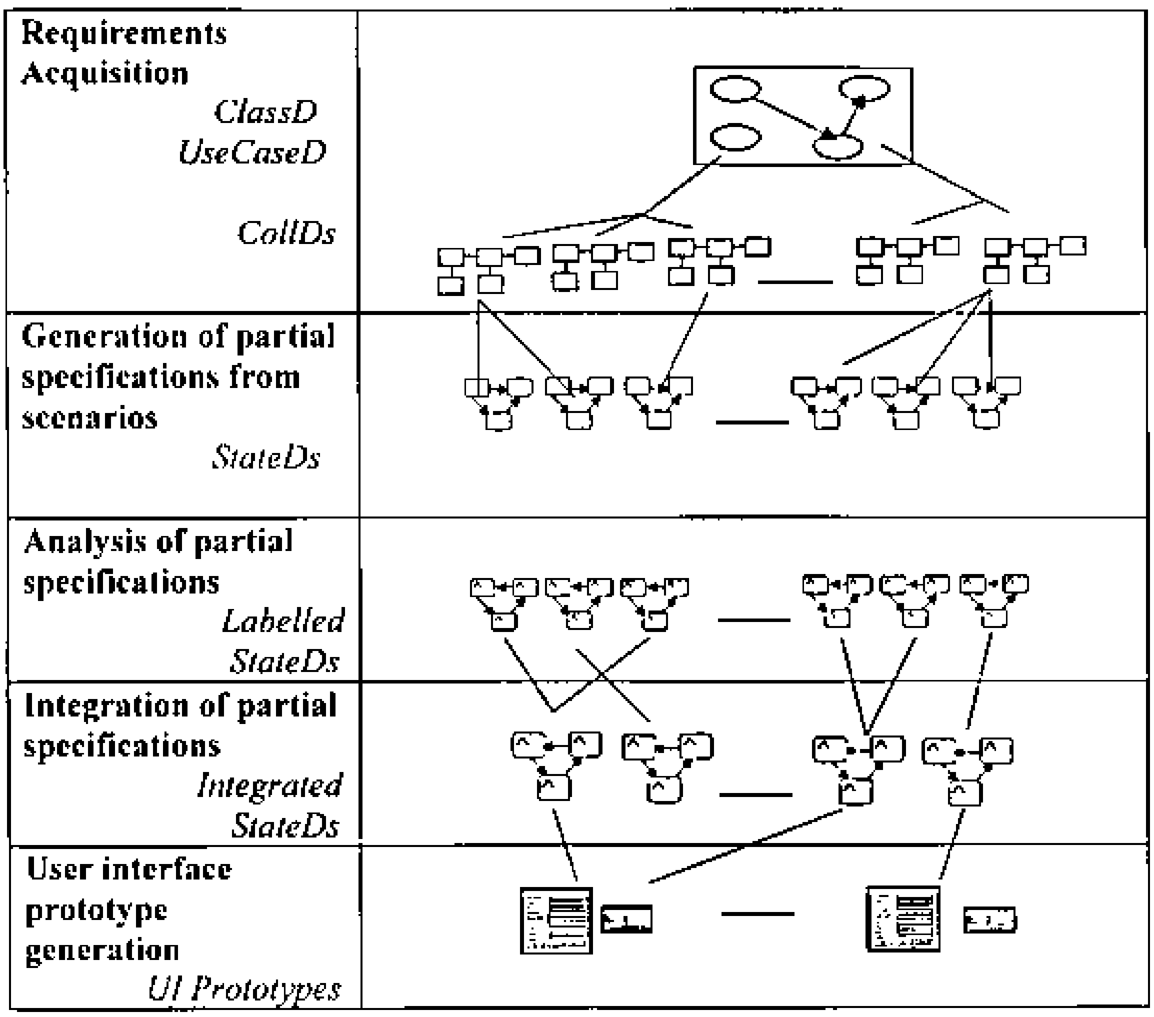

Requirements Acquisition
ClassD
UseCaseD
CollDs
Generation of partial specifications from scenarios
StateDs
Analysis of partial specifications
Labelled StateDs
Integration of partial specifications
Integrated StateDs
User interface prototype generation
UI Prototypes

Figure 3.8: The five activities of the UI prototyping process.

Scenarios of a given use cases are classified by type and ordered by frequency of use. We have considered two types of scenarios: normal scenarios, which are executed in normal situations, and scenarios of exception executed in case of errors and abnormal situations. The frequency of use of a scenario is a number between 1 and 10 assigned by the analyst to indicate how often a given scenario is likely to occur [15]. In our examples, the use case ***LoginToSystem*** has one normal scenario (scenario *successfullyLoginSystem* with frequency 5) and two scenarios of exception (scenario *cancelLoginToSystem* with frequency 3 and scenario *errorLoginToSystem* with frequency 5). The use case ***BorrowBook*** has one normal scenario (scenario *regularBorrowBook* with frequency 10) and two scenarios of exception (scenario *cancelBorrowBook* with frequency 3 and scenario *errorUserBorrowBook* with frequency 5).The use case ***ManageUser*** has one normal scenario (scenario *regularAddUser* with frequency 6) and one scenarios of exception (scenario *cancelAddUser* with frequency 5), and ***ManageUser*** use case has one normal scenario (scenario *regularDeleteUser* with frequency 6) and one scenarios of exception (scenario *cancelDeleteUser* with frequency 5). The use case ***SearchBook*** has one normal scenario (scenario *regularSearchBook* with frequency 4) and two scenarios of exception (scenario *cancelSearchBook* with frequency 3 and scenario *failerSearchBook* with frequency 4). The use case ***BrowseBooks*** has one normal scenario (scenario *BrowseBook* with frequency 4) and one scenarios of exception (scenario *cancelBrowseBook* with frequency 2). This classification is used for the composition of UI blocks (see section 3.3.4).

In our examples, the object *Terminal* is a generalization of the following objects, *LoginUI, BorrowBookUI, ManageUserUI, SearchBookUI* and *BrowseBookUI*. The object *Terminal* is a special object called *interface object*. An *interface object* is defined as an object through which the user interacts with the system to enter input data and receive results. An *interactive message* is defined as a message in a CollD that is sent to an interactive object. For UI generation purposes, messages corresponding to user interactions, which are marked in the CollDs with the type of interaction objects (i.e., widgets) that the analyst wants to find in the resulting UI. For instance in Figure 3.2(a), the mark *#B#* at the beginning of the name of message *1.2* means that this message corresponds to a user interaction with the *Button* widget. Note that *#TF#* stands for *Text Field* as in message *1.4.1*, *#IF#* for *Input Field* as in message *1.1*, and *#L#* for *Label* as in message *1.4* in Figure 3.2(c), and #CB# for *Compo Box* as in message *1.1.1* in Figure 3.6(a).

3.2.2 Generation of partial specifications from scenarios

In this activity, we repeatedly apply on each CollD the CollD-To-StateD transformation algorithm (CTS) described in [26]. In order to generate Partial specifications for all the objects participating in the input scenario.

Transforming one CollD into StateDs is, according to the CTS algorithm, a process of five steps. ***Step 1*** creats a StateD for every distinct class implied by the objects in the CollD. ***Step 2*** introduces as state variables all varibles that are not attributes of the objects of the CollD. ***Step 3*** creats transitions for the objects from which messages are sent. *Step 4* creates transitions for the objects to which messages are sent. Finally, *step 5* brings for all StateDs the set of generated transitions into correct sequences, connecting them by states.

After applying the CTS algorithm to the scenarios *regularBorrowBook*, *ErrorBorrowBook*, we obtain for the object Terminal the partial StateDs shown in figure 3.9(a), figure 3.9(b), *respictively*.

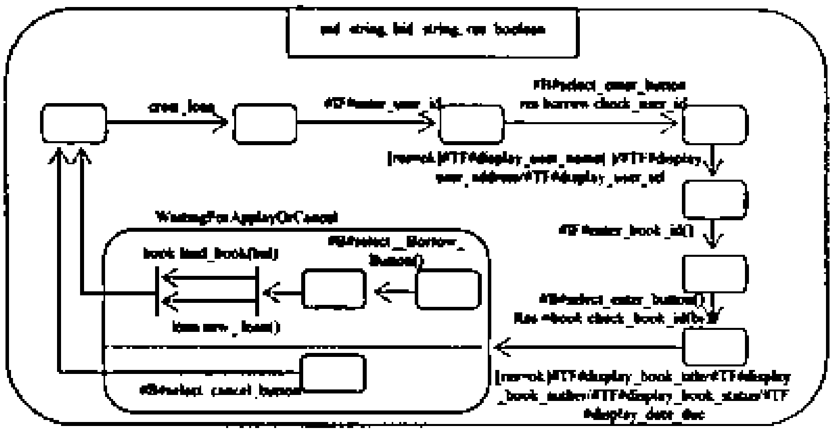


Figure 3.9(a): StateD for the object terminal generated by CTS algorithm on the scenario *regularBorrowBook*

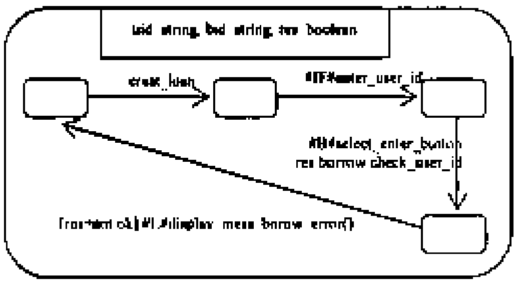


Figure 3.9(b): StateD for the object terminal generated by CTS algorithm on the scenario *ErrorBorrowBook*

Applying the CTS algorithm to the scenarios *regularSearchBook and ErrorSearchBook*, we obtain for the object Terminal the partial StateDs shown in figure 3.10(a) and figure 3.10(b), *respictively.*

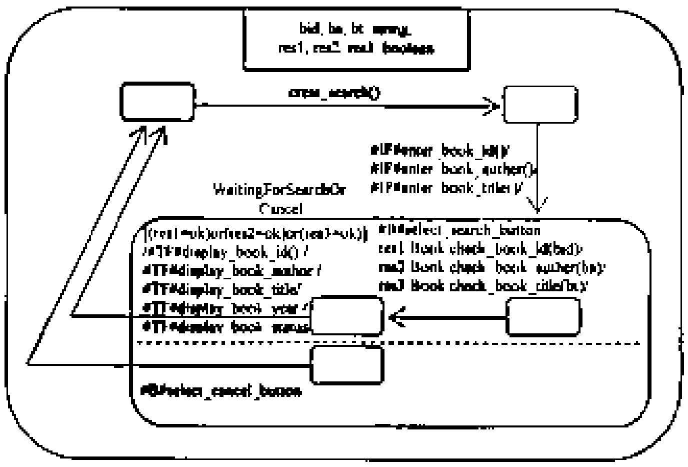


Figure 3.10(a): StateD for the object terminal generated by CTS algorithm on the scenario *succefullySearchBook*

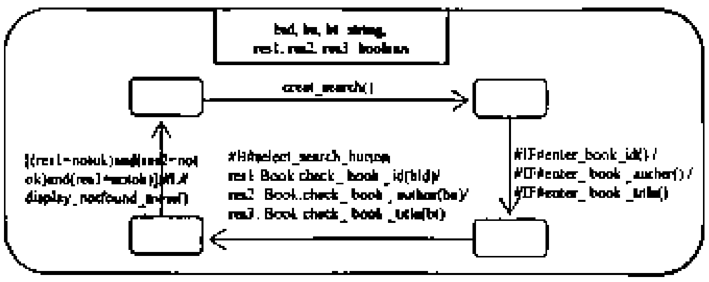


Figure 3.10(b): StateD for the object terminal generated by CTS algorithm on the scenario *failerSearchBook*

The applying of the CTS algorithm to the other scenarios (*regularLoginToSystem, ErrorLoginToSystem, regularBrowseBooks, ErrorBrowseBooks, regularManageUser* and *errorManageUser*), see the appendix A.1.

3.2.3 Analysis of partial specifications

The partial StateDs generated in the previous activity are unlabeled, i.e., their states do not carry names. However, the scenario integration algorithm (see Section 3.2.4 below) is state based, requiring labled StateDs as input. To obtain labled StateDs, our approach uses the pre- and postconditions to add state names. The analyst must identify equivalent states and give them common state names. Unique states are labeled with unique state names.

Applying this algorithm to the StateDs for ***BorrowBook*** of figures 3.9(a) and 3.9(b), we obtain the StateD shown in figure 3.11(a) and 3.11(b), respectively.

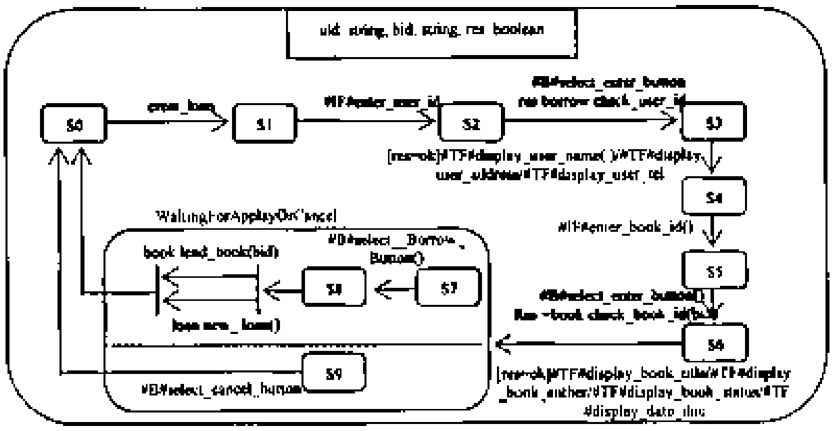


Figure 3.11(a): The labeled StateD obtained from the StateD of Figure 3.9(a)

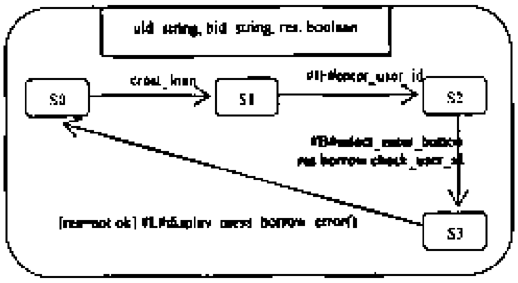


Figure 3.11(b): The labeled StateD obtained from the StateD of Figure 3.9(b)

Applying this algorithm to the StateDs for ***SearchBook*** of figures 3.10(a) and 3.10(b), we obtain the StateD shown in figure 3.12(a) and 3.12(b), respectively.

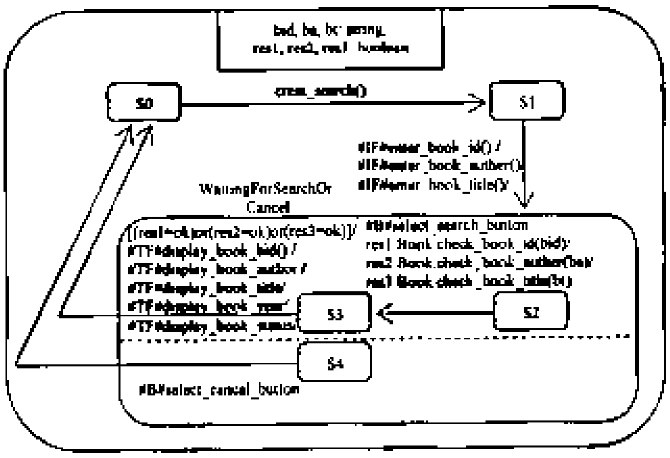


Figure 3.12(a): The labeled StateD obtained from the StateD of Figure 3.10(a)

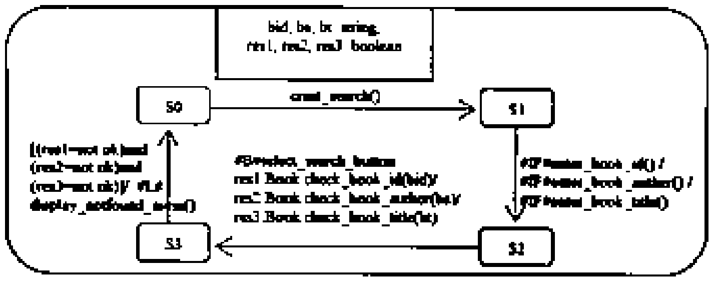

bid, ba, bt string
res1, res2, res3 boolean
S0
S1
S2
S3
start_search()
[(res1=not ok)and (res2=not ok)and (res3=not ok)]/ #L# display_notfound_mess()
#B#select_search_button
res1 Book check_book_id(bid)/
res2 Book check_book_author(ba)/
res3 Book check_book_title(bt)

Figure 3.12(b): The labeled StateD obtained from the StateD of Figure 3.10(b)

Note that no new state lable is generated for this particular StateD. The applying of this algorithm to the other scenarios (*regularLoginToSystem*, *ErrorLoginToSystem*, *regularBrowseBooks*, *ErrorBrowseBooks*, *regularManageUser* and *errorManageUser*), see the appendix A.2.

3.2.4 Integration of partial specifications

The objective of this activity is to integrate for each object and each use case in which it participates all its partial StateDs into one single StateD per use case [8]. For instance Figure 3.13 shows the resultant StateD of the *Terminal* object after the integration of the three scenarios of use case *BorrowBook*.

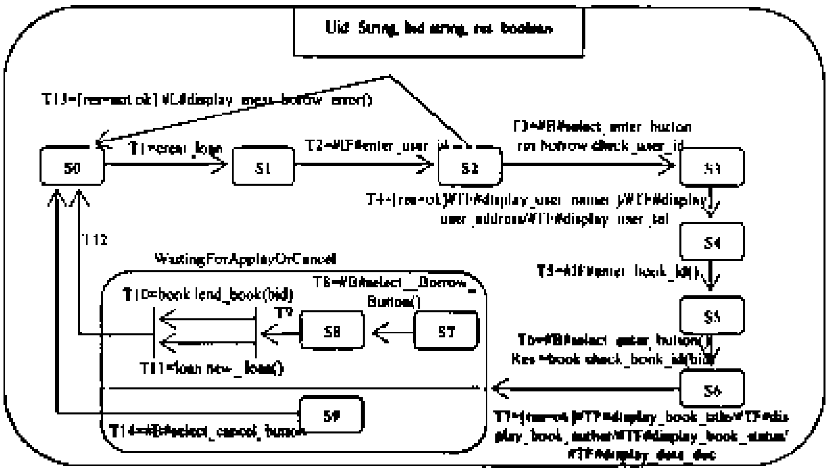

Uid String, bid string, res boolean
T13=[res=not ok] #L#display_mess_borrow_error()
T1=creat_loan
T2=#LF#enter_user_id
T3=#B#select_enter_button
res borrow check_user_id
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
T12
WaitingForApplyOrCancel
T10=book lend_book(bid)
T11=loan new_loan()
T9
T8=#B#select_Borrow_Button()
T14=#B#select_cancel_button
T5=#LF#enter_book_id()
T6=#B#select_enter_button()
Res =book check_book_id(bid)

Figure 3.13: The resultant StateD for the Terminal object after integration of the three scenario of the use case *BorrowBook*

Figure 3.14 shows the resultant StateD of the *Terminal* object after the integration of the three scenarios of use case *SearchBook*.

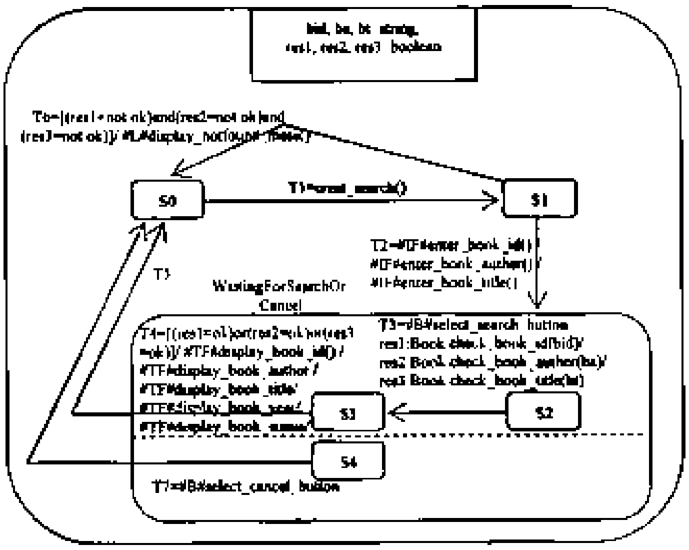

S0
S1
S2
S3
S4
T1=start_search()
T2=#TF#enter_book_id() / #TF#enter_book_author() / #TF#enter_book_title()
T3
WaitingForSearchOr Cancel
T7=#B#select_cancel_button

Figure 3.14: The resultant StateD for the Terminal object after integration of the three scenario of the use case *SearchBook*

The resultant StateDs of the *Terminal* object after the integration of the scenarios of other use cases (*regularLoginToSystem*, *ErrorLoginToSystem*, *regularBrowseBooks*, *ErrorBrowseBooks*, *regularManageUser* and *errorManageUser*), see the appendix A.3.

3.2.5 User interface prototype generation

In this activity, we derive UI prototypes for the interface objects found in the system. Both the static and the dynamic aspects of the UI prototypes are generated from the StateDs of the underlying interface objects. For each interface object, we generate from its StateDs, as found in the various use cases, a standalone prototype. This prototype comprises a menu to switch between the different use cases. The different screens of the prototype visualize the static aspect of the object; the dynamic aspect of the object maps into the dialog control of the prototype. In our current implementation, prototypes are Java applications comprising each a number of frames and navigation functionality (see Figures 3.15 and 3.24). The details of prototype generation are described in the next section.

3.3 Algorithm for User Interface Prototype Generation

In this section, we detail the process of prototype generation from interface object behavior specifications. This process can be summarized in the following algorithm [32].

```
Let IO be the set of interface objects in the system,
Let UC={uc1, uc2,..., ucn} be the set of use cases of the system,
For each io in IO
        For each uci in UC
                If   io usedInUsecase(uci) then
                        sd = io getStateDforUsecase(uci)
                        sd generatePrototype()
                End If
        End For
        io generateCompletePrototype()
End For
```

The operation *usedInUsecase(uci)*, applied to the object *io*, checks if the object *io* participates or not in one or more of the CollDs associated with use case *uci*. If the operation returns *true*, the operation *getStateDforUsecase(uci)* is called, which retrieves *sd*, the StateD capturing the behavior of object *io* that is related to this use case. From StateD *sd*, a UI prototype is generated using the operation *generatePrototype()*.

The operation *generateCompletePrototype()* integrates the prototypes generated for the various use cases into one single application. This application comprises a menu (see Figure 3.15) providing as options the different use cases in which object *io* participates.

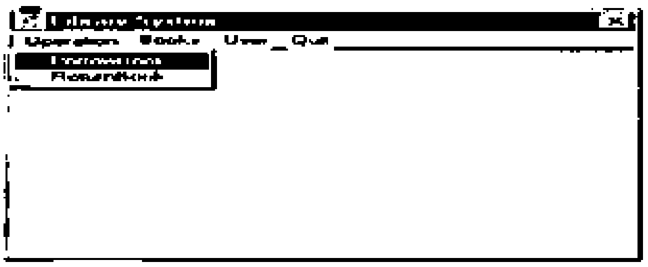


Figure 3.15: Menu generated for the interface object

The operation of prototype generation (*generatePrototype()*) is composed of five operations, which are described in the sections below:

- generating graph of transitions
- masking non-interactive transitions
- identifying user interface blocks
- composing user interface blocks
- generating the user interface from composed blocks.

3.3.1 Generating graph of transitions

This operation consists of deriving a directed graph of transitions (GT) from the StateD of an interface object *io* related to a use case *uci*. Transitions of the StateD will represent the nodes of the GT. Edges will indicate the precedence of execution between transitions. If transition *t1* precedes transition *t2* in execution, we will have an edge between the nodes representing *t1* and *t2*.

A GT has a list of nodes *nodeList*, a list of edges *edgeList*, and a list of initial nodes *initialNodeList* (entry nodes for the graph). The *nodeList* of a GT is easily obtained, since it corresponds to the transition list of the StateD at hand. The *edgeList* of a GT is obtained by identifying for each transition *t* all the transitions that enter the state from which *t* can be triggered. All these transitions precede the transition *t* and hence define each an edge to node *t*.

In the library system, given the StateD of *Terminal* for the use cases ***BorrowBook*** (see Figure 3.13) and ***searchBook*** (see Figure 3.14), the graph of transition GT generated shown in Figures 3.16(a), 3.17(a), respictively. The star character (*) is used to mark initial nodes in the graph. We mention that the graph transition GT for the StateD for *Terminal* to the scenarios of other use cases (*LoginToSystem*, *BrowseBooks* and *ManageBook*), had illustrated in the appendix A.4.

The algorithm details how to get *nodeList*, *edgeList*, and *initialNodeList* of the GT from a given StateD *sd*, see the appendix B.

3.3.2 Masking non-interactive transitions

This operation consists of removing all transitions that do not directly affect the UI (i.e., that do not carry widgets). These transitions are called *non-interactive* transitions. All such transitions are removed from the list of nodes *nodeList* and from the list of initial nodes *initialNodeList*, and all edges defined by those transitions are removed from *edgeList*. When a transition *t* is removed from *nodeList*, we remove all edges where *t* takes part, and we add new edges in order to "bridge" the removed transition nodes. If the *initialNodeList*

list of initial transitions contains any non-interactive transitions, they are replaced by their successor nodes.

The result of this operation on the graph of Figures 3.16(a) and 3.17(a) for ***BorrowBook*** and ***SearchBook*** use cases, is given in Figures 3.16(b) and 3.17(b), respectively.

Applying this operation on the scenarios of other use cases (*LoginToSystem*. *BrowseBooks* and *ManageBook*, had illustrated in the appendix A.4. The pseudocode details this operation, see the appendix B.

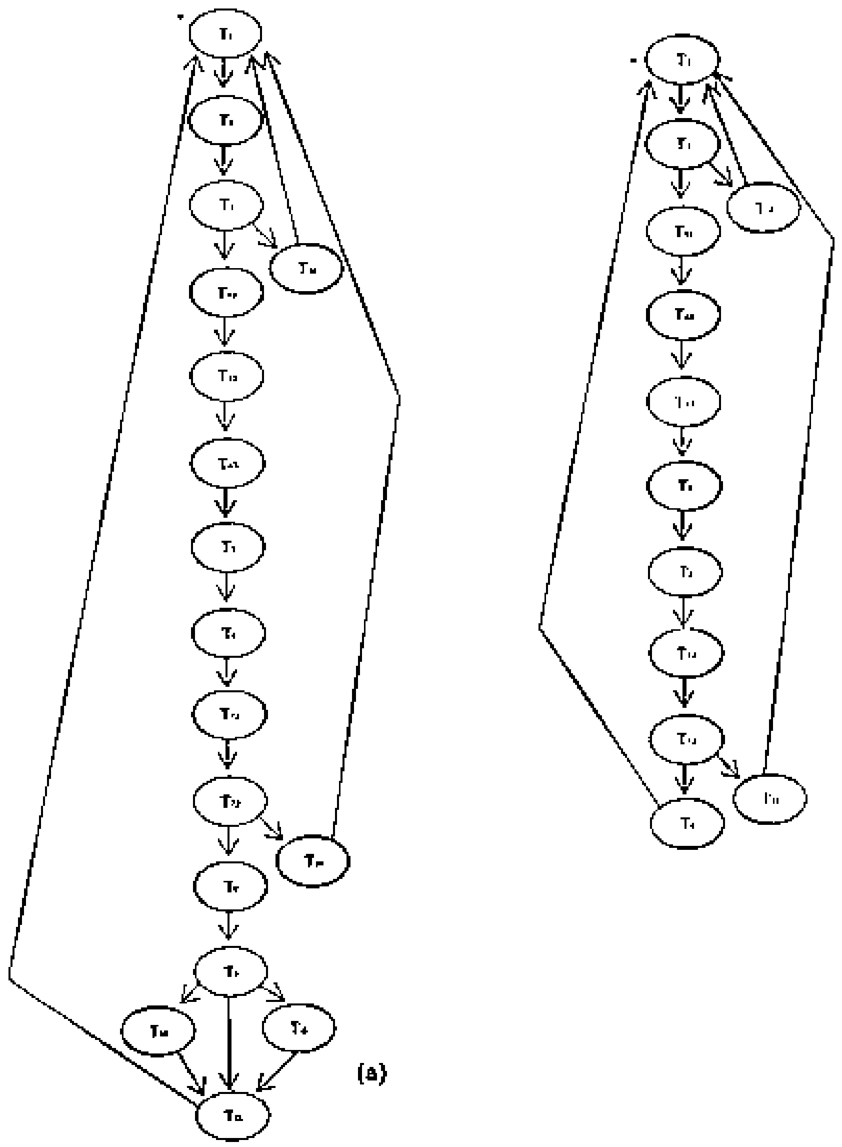


Figure 3.16: (a) Transition graph for the object Terminal and the use case *BorrowBook* (GT). (b) Transition graph after masking non-interactive transitions (GT').

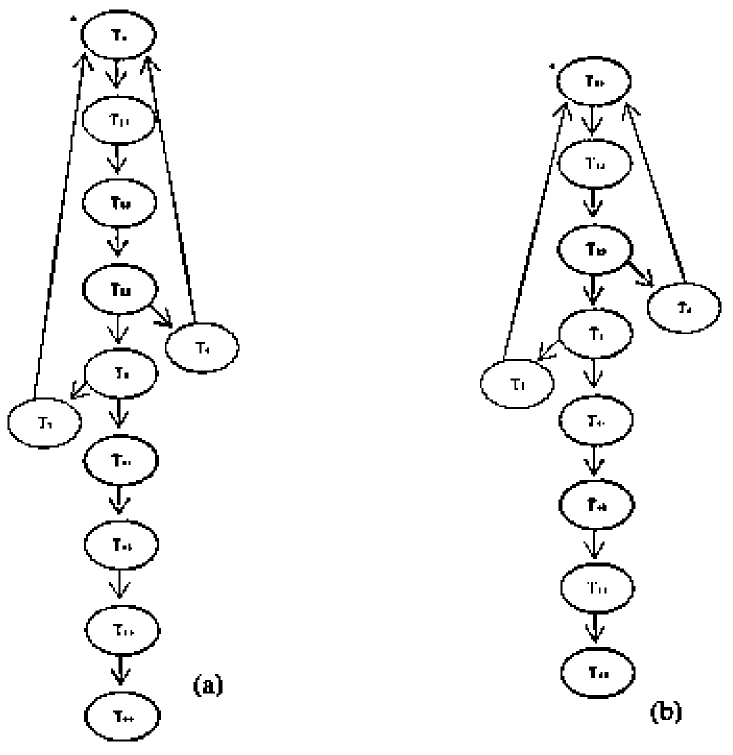


Figure 3.17: (a) Transition graph for the object Terminal and the use case *SearchBook* (GT). (b) Transition graph after masking non-interactive transitions (GT').

3.3.3 Identifying user interface blocks

This operation consists of constructing a directed graph where nodes represent *User Interface Blocks* (UIB). A UIB is a subgraph of GT' consisting of a sequence of transition nodes that is characterized by a single input and a single output edge. The beginning and the end of each UIB is identified from the graph GT' based on the following rules [16]:

(Rule 1) An initial node of GT' is the beginning of a UIB.

(Rule 2) A node that has more than one input edge is the beginning of a UIB.

(Rule 3) A successor of a node that has more than one output edge is the beginning of a UIB.

(Rule 4) A predecessor of a node that has more than one input edge ends a UIB.

(Rule 5) A node that has more than one output edge ends a UIB.

(Rule 6) A node that has an output edge to an initial node ends a UIB.

Applying these rules to the graph of Figures 3.16(b) and 3.17(b), we obtain the graph block GB shown in Figures 3.18 and 3.19, respictively. In figures 3.18, Rule 1 determines the beginning of *B1 (T2)* and Rule 5 the end of *B1 (T3)*. Rules 3 and 5 determine the UIB *B2*. The UIBs *B3, B4* and *B5* are generated by applying Rule 3 or Rule 6.

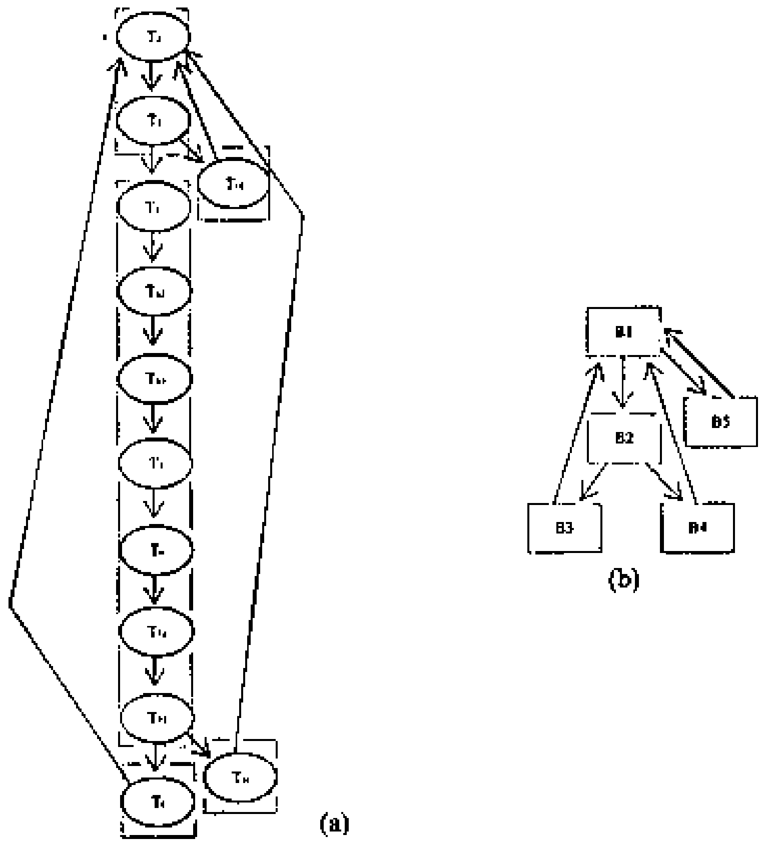


Figure 3.18: Graph GB resulting from UIB identification on the graph GT' of Figure 3.16(b)

In figures 3.19, Rule 1 determines the beginning of *B1 (T2.1)* and Rule 5 the end of *B1 (T2.3)*. Rule 3 determine the UIB *B2*. The UIBs *B3* and *B4* are generated by applying Rule 3 or Rule 6.

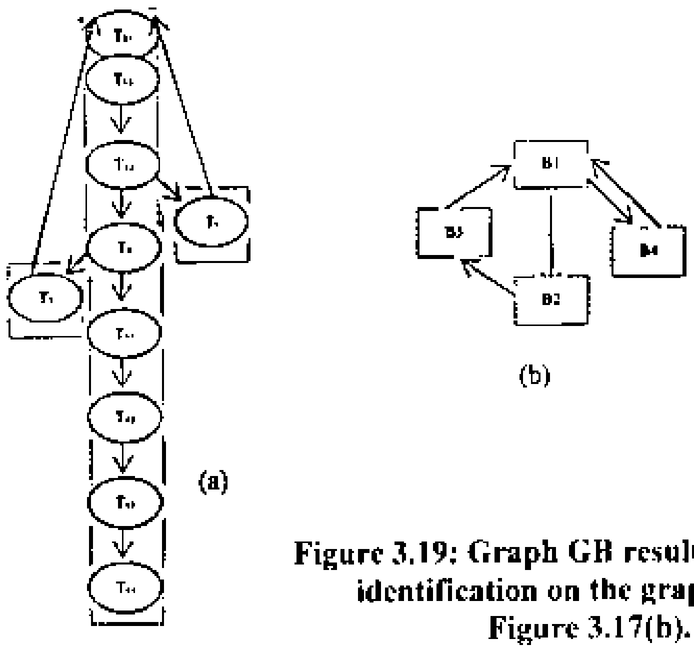


Figure 3.19: Graph GB resulting from UIB identification on the graph GT' of Figure 3.17(b).

Applying these rules on the other scenarios of use cases (*LoginToSystem*, *BrowseBooks* and *ManageUser*, see the appendix A.5.

3.3.4 Composing user interface blocks

Generally, the UI blocks obtained from the previous operation contain only few widgets and represent only small parts of the overall use case functionality. Our approach supports the combination of UIBs in order to have more interesting blocks which can be transformed into suitable graphic windows. We use the following rules to merge the UIBs of a use case [15]:

(Rule 6) Adjacent UIBs belonging to the same scenario are merged (scenario membership).

(Rule 7) The operation of composition begins with scenarios having the highest frequency (scenario classification, see Section 3.2.1).

(Rule 8) Two UIBs can only be grouped if the total of their widgets is less than 20 (ergonomic criterion).

Applying these rules to the GB of Figures 3.18 and 3.19, results in the graph GB' of UIBs shown in Figures 3.20 and 3.21, respictively.

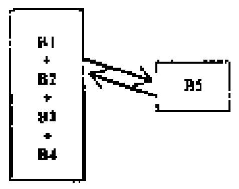


Figure 3.20

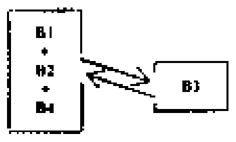


Figure 3.21

Figures 3.20, 3.21: Graph GB' resulting from user interface block composition on the graph GB of Figures 3.18, 3.19, respictively.

Applying these rules to the GB of on the other scenarios of use cases (*LoginToSystem*, *BrowseBooks* and *ManageBook*, results in the graph GB' of UIBs see the appendix A.6.

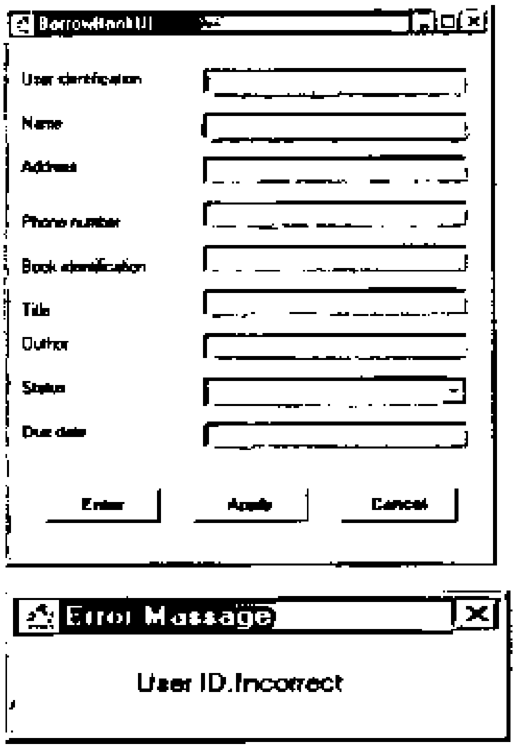
3.3.5 Generating the user interface from composed blocks

In this operation, we generate for each UIB of GB' a graphic frame. The generated frame contains the widgets of all the transitions belonging to the concerned UIB. Edges between UIBs in GB' are transformed to call functions in the appropriate frame classes. In our current implementation, Java code is generated that is compatible with the interface builder [29].

The two frames derived from each composed blocks of the graph GB' of Figures 3.20 and 3.21, are shown in Figures 3.22 and 3.24, respictively. The dynamic aspect of the UI is controlled by the behavior specification (StateD) of the underlying interface object. The prototype responds to all user interaction events captured in GT', and ignores all other event.

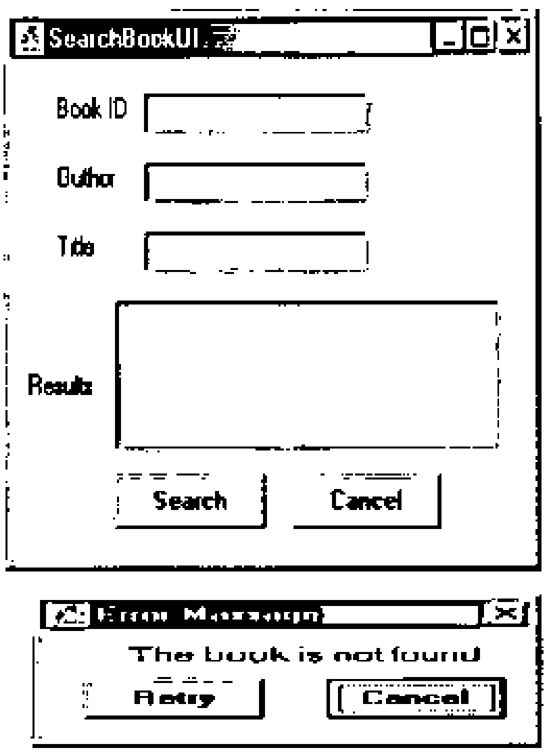
To support prototype execution. For example, after selecting the use case *BorrowBook* from the *UseCases* menu (see Figure 3.23, top window), a frame is displayed in the simulation window that confirms the use case selection and prompts the user to input the user identification and to click the ***Enter*** button. When execution reaches a node in GT' from which several continuation paths are possible. In the example of Figure 3.24, the lower frame corresponds to the scenario *errorBorrowBook*, and the upper frame one to the scenarios *regularBorrowBook* and *cancelBorrowBook.*

To support prototype execution. For example, after selecting the use case *Search Book* from the *UseCases* menu (see Figure 3.25, top window), a frame is displayed in the simulation window that confirms the use case selection and prompts the user to input the book information and to click the ***Search*** button. When execution reaches a node in GT' from which several continuation paths are possible. In the example of Figure 3.25, the lower frame corresponds to the scenario *failerSearchBook*, and the upper frame one to the scenarios *regularSearchBook* and *cancelSearch*. Once a path has been selected.



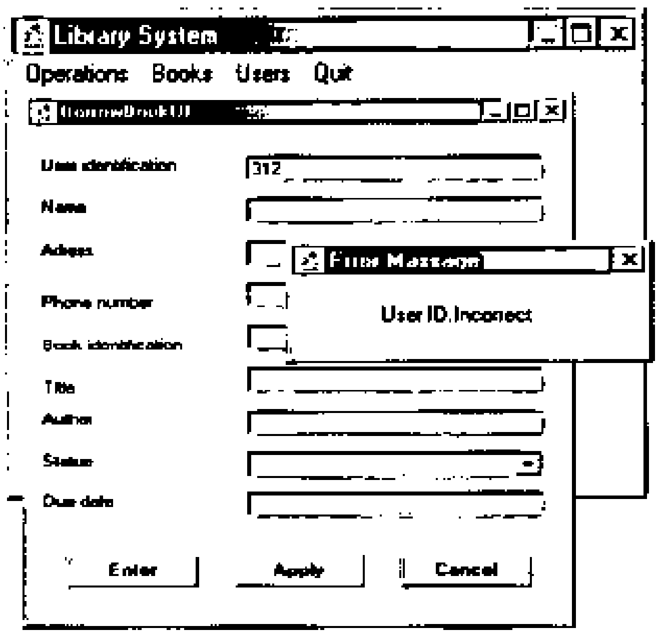
User ID.Incorrect

Figure3.22 : Frames generated for the use case *BorrowBook.*



The book is not found

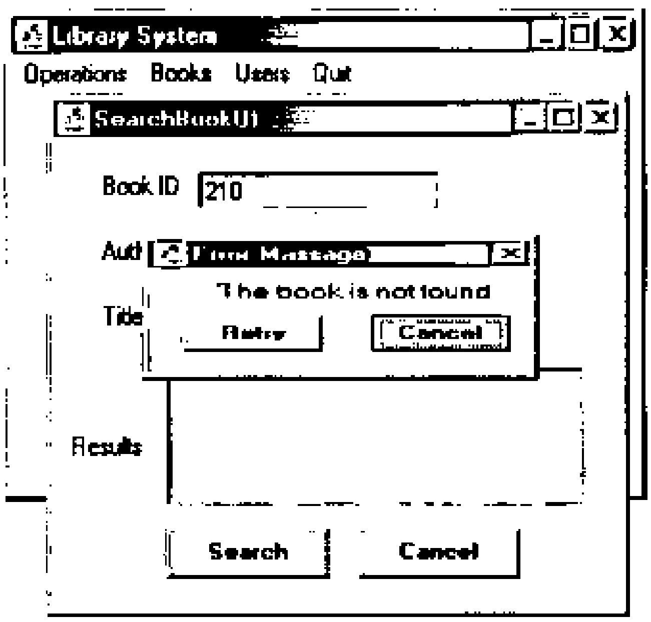
Figure3.23 : Frames generated for the use case *SearchBook.*



312

User ID.Incorrect

Figure 3.24: Prototype execution for BorrowBook.



210

The book is not found

Figure 3.25: Prototype execution for SearchBook.

Chapter Four

Results

4.1 Conclusions

This thesis discussed user interface modeling, a new approach to the generation of UI prototypes from scenarios using a Library System case study. The case study was modelled using the Unified Modeling Language that has proved for modeling and generating user interfaces.

On one hand, from the case study we can elicit and define some problems that faced during modeling and generating UIs. from the modelling problems we can identify some aspects of UIs that are not covered by the UML, such as:

- Use cases do not provide some aspects of user requirements like goals, pre-conditions and post conditions that may help the design of activity diagrams.
- UML does not describe clearly the relationship between use cases and activities.
- UML does not have a notation to describe abstract presentations.
- UML does not provide a relationship between classes providing an abstract presentation and activities. In fact, it is difficult to identify which UI is related to each activity that involves user interaction.

On the other hand, from the set of constructors we can identify the aspects of UIs that are covered by the UML, where we found that the UML has a rich set of constructors complete enough to model the architectural aspects of form-based user interfaces.

Additionally, from the approach of generating user interface prototypes from scenarios, some notes that can be obtained:

- The analyst has the *manual* task of eliciting scenarios of the system and of labeling the generated partial StateDs.
- Our approach may be applied to windows and widgets interfaces, yet fails to support in its current form alternative UI paradigms.
- Verification of characteristics such as coherence, completeness, etc.

4.2 Outlook

For modelling the user interface, the case study provides illustrative examples of the use of many UML constructors, in terms of diagrams. The UML diagrams used in this thesis are class diagram, activity diagram and class diagram with design patterns, interaction (sequence) diagram object diagram, these UML diagrams had used to model Domain Model, Task Model and Presentation Model (abstract and concrete) as user interface element, respectively.

There are also some lessons that can be learned from the modelling of the Library System:

- The design of an user interface is a complex process since it requires complete comprehension of the elements that compose the user interface. Indeed, UIs in general have many elements that are not clearly required from the beginning of the design.
- The elements of the user interface have many dependencies among them. Therefore, the design process should consider UI modelling as integral.

And also, from the approach of generating user interface prototypes from scenarios, the scope of this approach is three fold:

- It proposes a process for requirements engineering compliant with the UML.
- It provides automatic support for building object specifications.
- It supports UI prototyping.

APPENDIX A

Figures Related to Chapter 3

A.1. Generation of partial specifications from scenarios

After applying the CTS algorithm to the senarios *regularConnectToSystem*, *ErrorConnectToSystem*, we obtain for the object Terminal the partial StateDs shown in figure A.1.1(a), figure A.1.1(b), *respictively*.

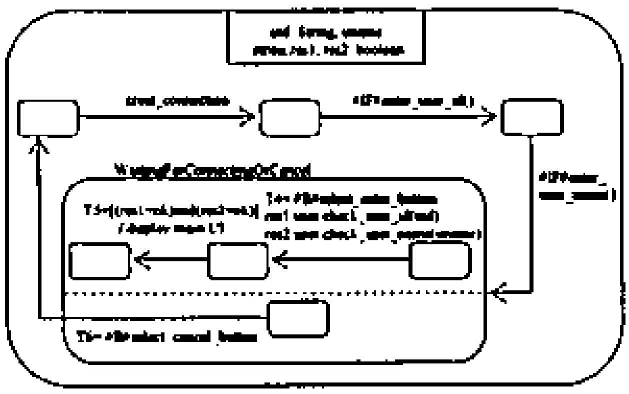


Figure A.1.1(a): StateD for the object termenal generated by CTS algorithm on the scenario *successfullyConnectToSystem*

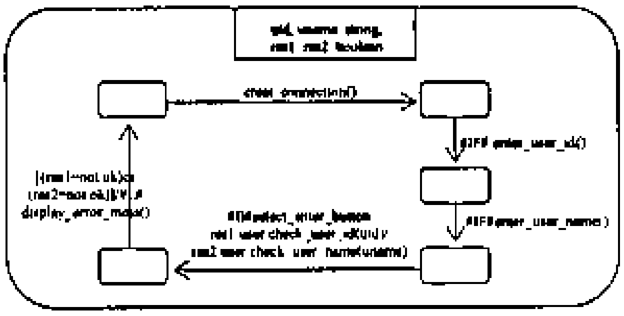


Figure A.1.1(b): StateD for the object termenal generated by CTS algorithm on the scenario *errorConnectToSystem*

Applying the CTS algorithm to the senarios *regularBrowseBook*, *ErrorBrowseBook*, we obtain for the object Terminal the partial StateDs shown in figure A.1.2(a), figure A.1.2(b), *respictively*.

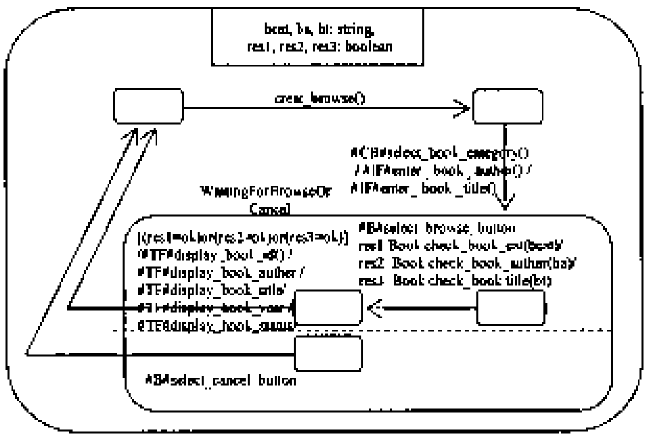

bcat, ba, bt: string, res1, res2, res3: boolean
creat_browse()
WaitingForBrowseOr Cancel
#BA#select_cancel_button

Figure A.1.2(a): StateD for the object termenal generated by CTS algorithm on the scenario *succefullyBrowseBook*

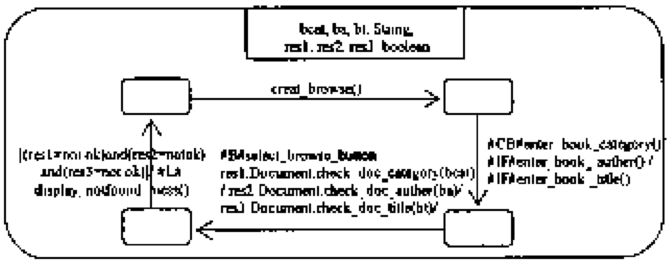


Figure A.1.2(b): StateD for the object termenal generated by CTS algorithm on the scenario *failerBrowseBook*

Applying the CTS algorithm to the senarios, *regularManageUser* (*Add* User) and *errorManageUser* (*Add* User), we obtain for the object Terminal the partial StateDs shown in figure A.1.3(a) , figure A.1.3(b), *respictively*.

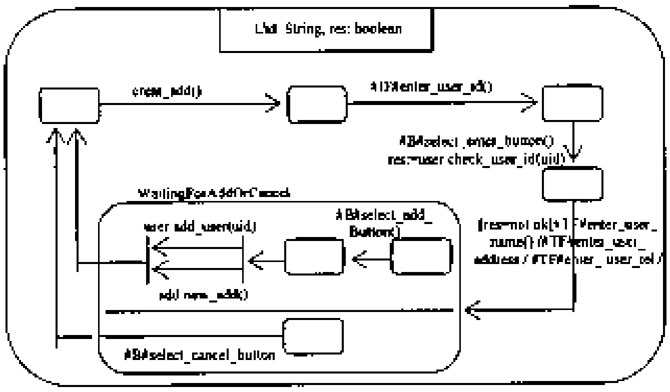


Figure A.1.3(a): StateDfor the object termenal generated by CTS algorithm on the scenario *regularAddUser*

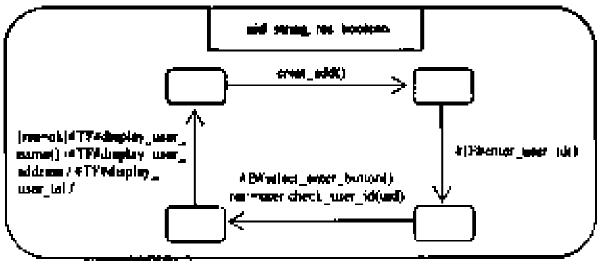


Figure A.1.3(b): StateDfor the object termenal generated by CTS algorithm on the scenario *errorAddUser*

Applying the CTS algorithm to the senarios, *regularManageUser* (*Delete User*) and *errorManageUser* (*Delete User*), we obtain for the object Terminal the partial StateDs shown in figure A.1.4(a) *and* figure A.1.4(b), *respictively.*

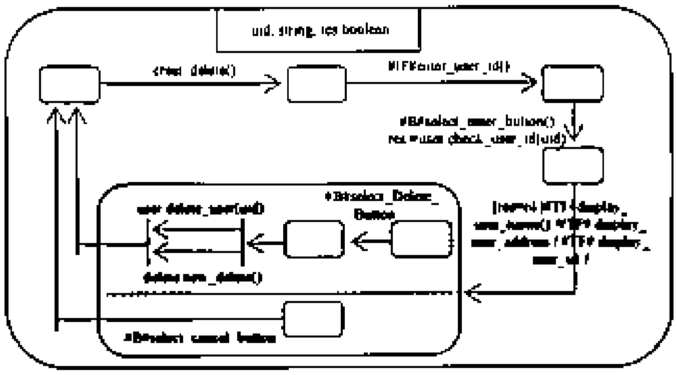


Figure A.1.4(a): StateDfor the object termenal generated by CTS algorithm on the scenario *regulareDeleteUser*

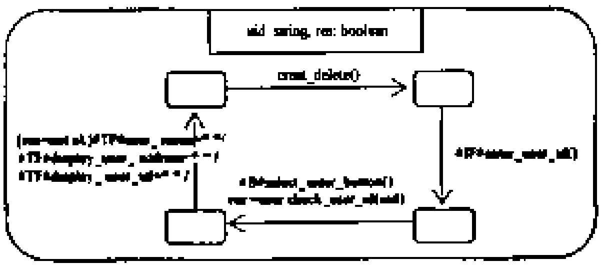


Figure A.1.4(b): StateDfor the object termenal generated by CTS algorithm on the scenario *errorDeleteUser*

A.2. Analysis of partial specifications

Applying the analysis of partial specifications algorithm to the StateDs of figures A.1.1(a) and A.1.1(b), we obtain the StateD shown in figure A.2.1(a) and A.2.1(b), respectively.

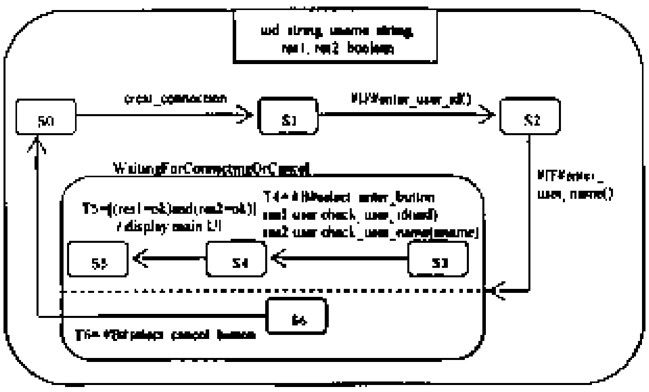

uid string, uname string,
res1, res2 boolean
S0
creat_connection
S1
#LF#enter_user_id()
S2
#TF#enter_
user_name()
WaitingForConnectionOrCancel
T5=[(res1=ok)and(res2=ok)]
/ display main UI
T4=#B#select enter_button
res1 user check_user_id(uid)
res2 user check_user_name(uname)
S5
S4
S3
S6
T6=#B#select cancel button

Figure A.2.1(a): The labeled StateD obtained from the StateD of Figure A.1.1(a)

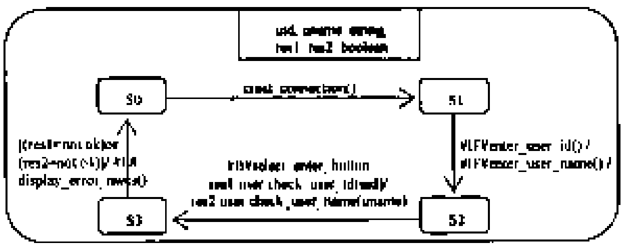


Figure A.2.1(b): The labeled StateD obtained from the StateD of Figure A.1.1(b)

Applying the analysis of partial specifications algorithm to the StateDs of figures A.1.2(a) and A.1.2(b), we obtain the StateD shown in figure A.2.2(a) and A.2.2(b), respectively.

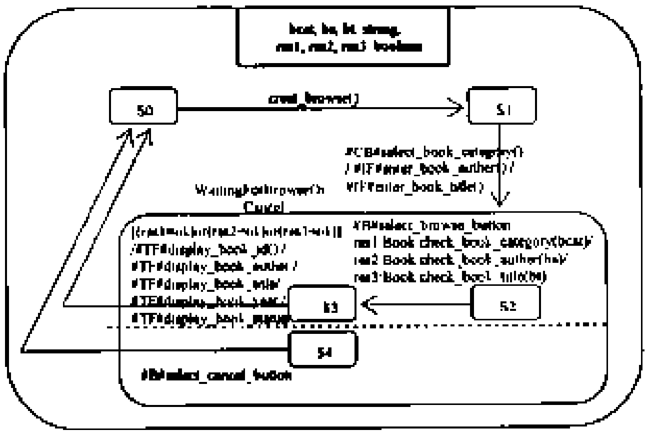


Figure A.2.2(a): The labeled StateD obtained from the StateD of Figure A.1.2(a)

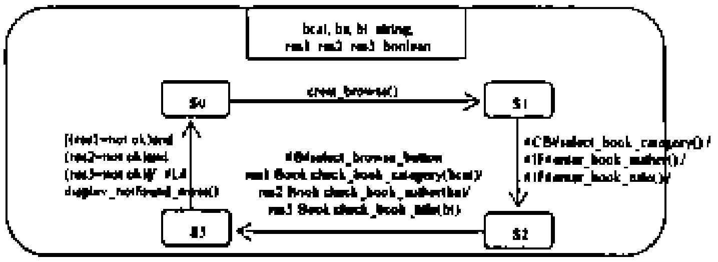


Figure A.2.2(b): The labeled StateD obtained from the StateD of Figure A.1.2(b)

Applying the analysis of partial specifications algorithm to the StateDs of figures A.1.3(a) and A.1.3(b), we obtain the StateD shown in figure A.2.3(a) and A.2.3(b), respectively.

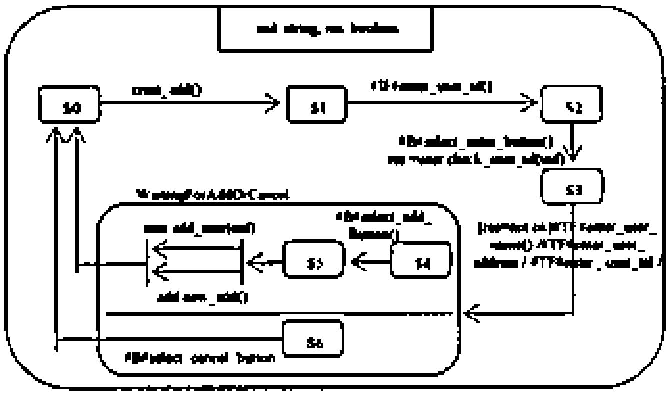


Figure A.2.3 (a): The labeled StateD obtained from the StateD of Figure A.1.3(a)

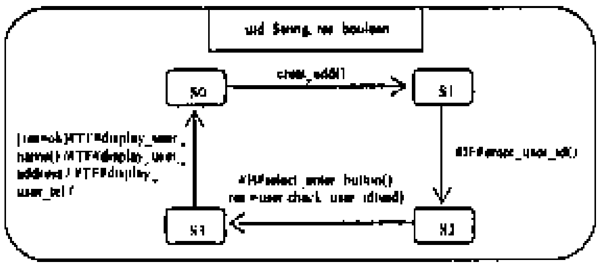


Figure A.2.3(b): The labeled StateD obtained from the StateD of Figure A.1.3(b)

Applying the analysis of partial specifications algorithm to the StateDs of figures A.1.4(a) and A.1.4(b), we obtain the StateD shown in figure A.2.4(a) and A.2.4(b), respectively.

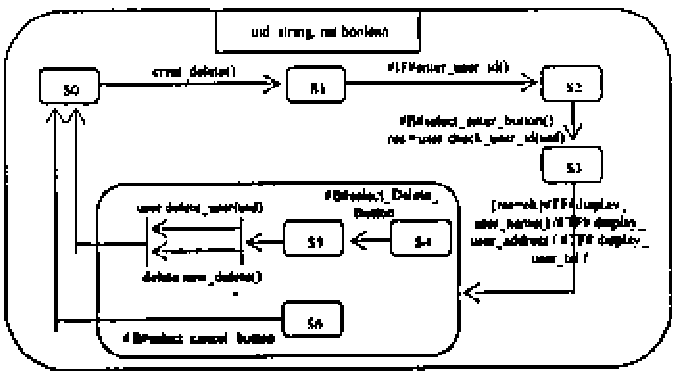


Figure A.2.4(a): The labeled StateD obtained from the StateD of Figure A.1.4(a)

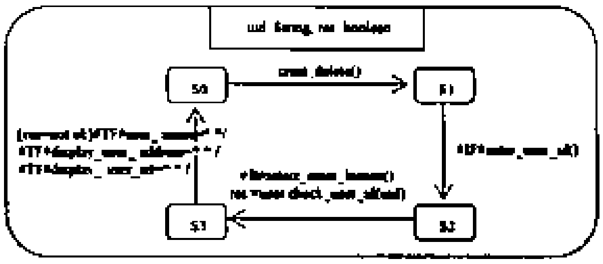


Figure A.2.4(b): The labeled StateD obtained from the StateD of Figure A.1.4(b)

A.3. Integration of partial specifications

Figure A.3.1 shows the resultant StateD of the *Terminal* object after the integration of partial specifications of the three scenarios of use case *ConnectToSystem*.

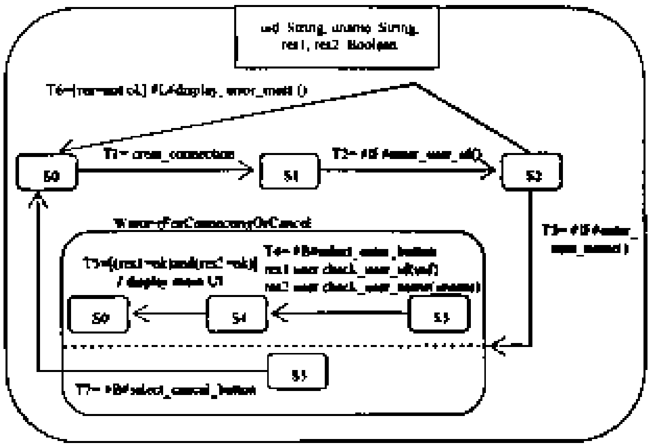


Figure A.3.1: Resultant StateD for the *Connect To System*

Figure A.3.2 shows the resultant StateD of the *Terminal* object after the integration of partial specifications of the three scenarios of use case *BrowseBook*.

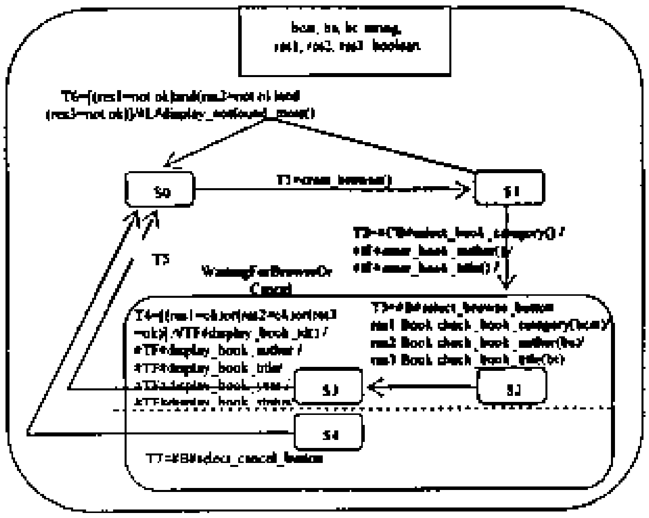


Figure A.3.2: stateD for the *BrowseBooks*

Figures A.3.3 and A.3.4 shows the resultant StateD of the *Terminal* object after the integration of partial specifications of the three scenarios of use case *ManageUser* (*AddUser* and *DeleteUser*), *respictively*.

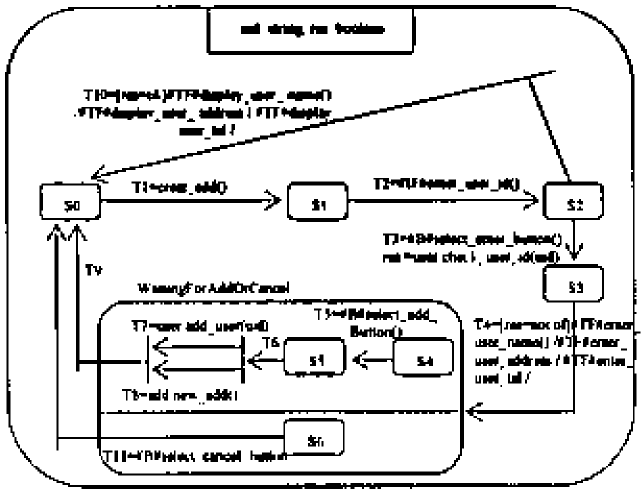


Figure A.3.3: stateD for *AddUser*

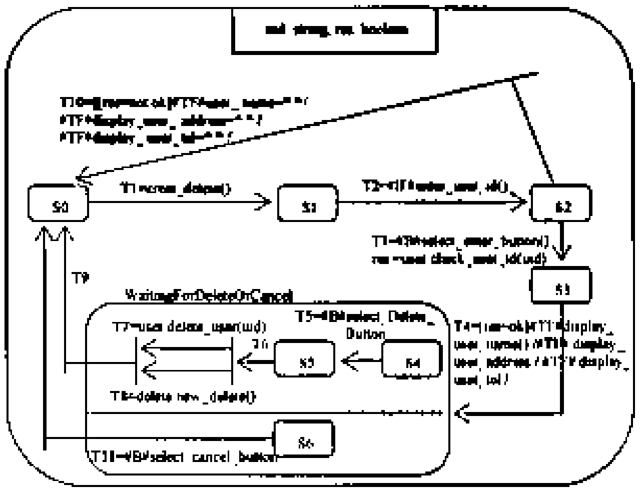


Figure A.3.4: stateD for *DeleteUser*

A.4. Generating graph of transitions and Masking non-interactive transitions

Given the StateD of *Terminal* for the use cases *ConnectToSystem* (see Figure A.3.1), the GT generated shown in Figures A.4.1(a).

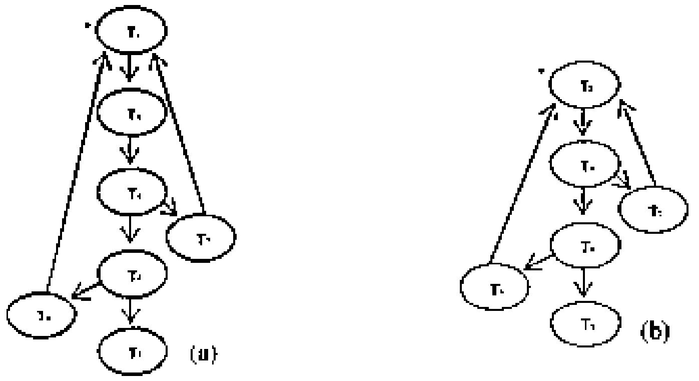


Figure A.4.1: (a) Transition graph for the object Terminal and the use case *ConnectToSystem* (GT). (b) Transition graph after masking non-interactive transitions (GT').

Given the StateD of *Terminal* for the use cases *BrowseBook* (see Figure A.3.2), the GT generated shown in Figures A.4.2(a).

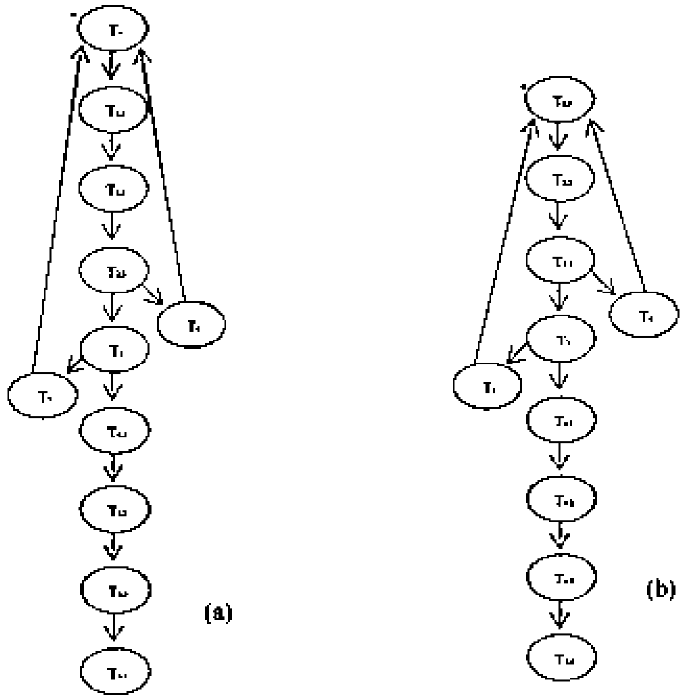


Figure A.4.2: (a) Transition graph for the object Terminal and the use case *BrowseBook* (GT). (b) Transition graph after masking non-interactive transitions (GT').

Given the StateD of *Terminal* for the use cases *ManageBook* (*AddUser* and *DeleteUser*) (see Figures A.3.3, A.3.4 respectively), the GT generated shown in Figures A.4.3(a).

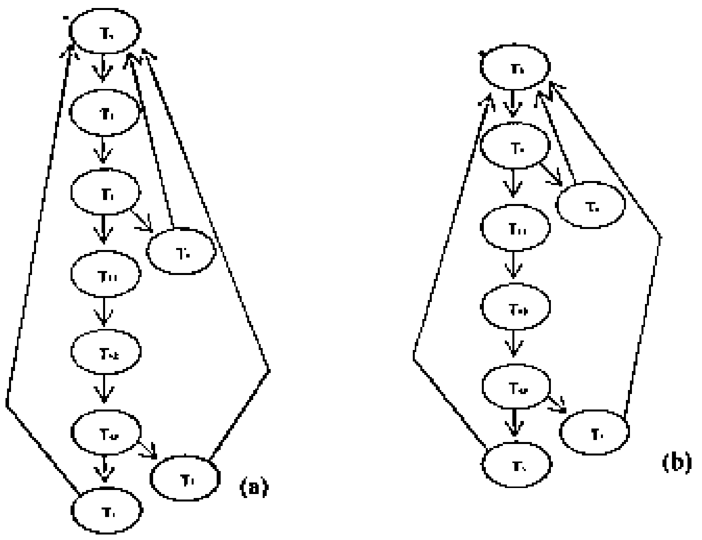


Figure A.4.3: (a) Transition graph for the object Terminal and the use case *ManageUser* (GT). (b) Transition graph after masking non-interactive transitions (GT').

A.5. Identifying user interface blocks

In figure A.5.1, Rule 1 determines the beginning of *B1 (T2)* and Rule 5 the end of *B1 (T3)*. Rule 3 determine the UIB *B2(T4)*. The UIBs *B3* and *B4* are generated by applying Rule 3 or Rule 6.

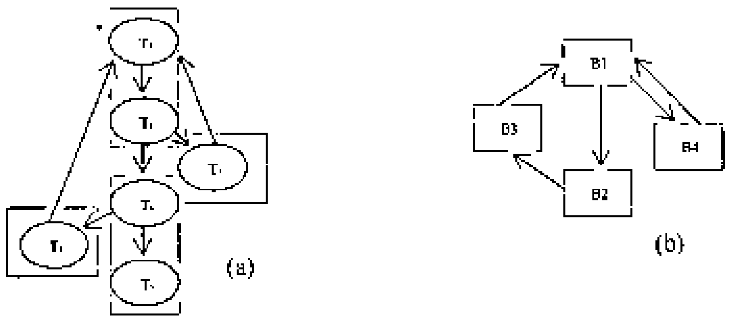


Figure A.5.1: Graph GB resulting from UIB Identification on the graph GT' of Figure A.4.1:(b)

In figure A.5.2, Rule 1 determines the beginning of *B1 (T2.1)* and Rule 5 the end of *B1 (T2.3)*. Rule 3 determine the UIB *B2*. The UIBs *B3* and *B4* are generated by applying Rule 3 or Rule 6.

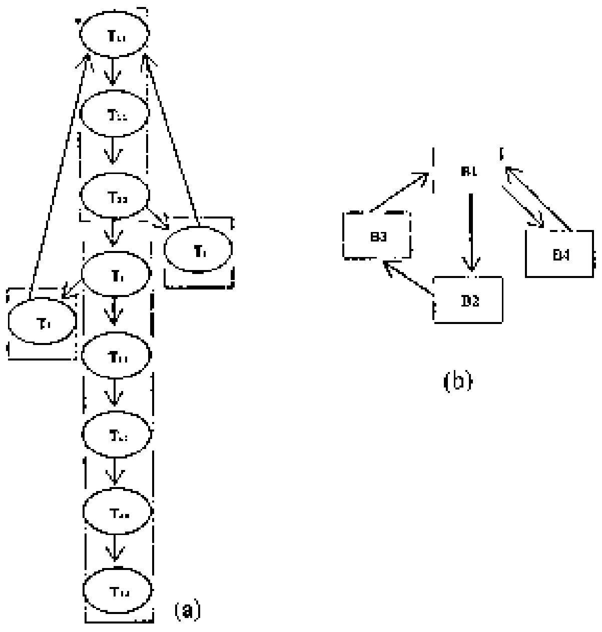
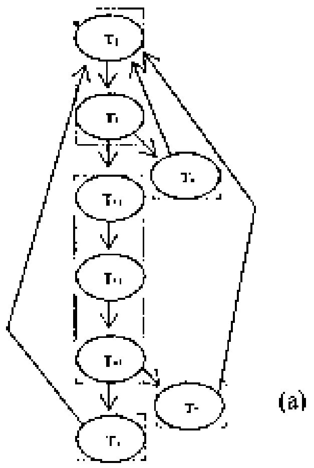


Figure A.5.2: Graph GB resulting from UIB identification on the graph GT' of Figure A.4.2:(b)

In figure A.5.3, Rule 1 determines the beginning of *B1 (T2.1)* and Rule 5 the end of *B1 (T2.3)*. Rule 3 determine the UIB *B2*. The UIBs *B3* and *B4* are generated by applying Rule 3 or Rule 6.



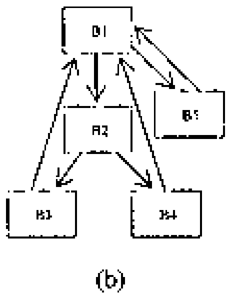


Figure A.5.3: Graph GB resulting from UIB identification on the graph GT' of Figure A.4.3:(b)

A.6. Composing user interface blocks

Applying the rules to the GB of Figures A.5.1, A.5.2 and A.5.3, results in the graph GB' of UIBs shown in Figures A.6.1, A.6.2 and A.6.3, respictively.

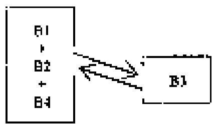


Figure A.6.1

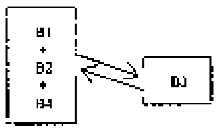


Figure A.6.2

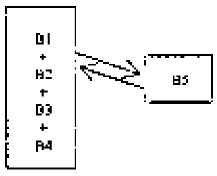


Figure A.6.3

Figures A.6.1, A.6.2, A.6.3: Graph GB' resulting from user interface block composition on the graph GB of Figures A.5.1, A.5.2, A.5.3, respictively.

Appendix B

Algorithms for User Interface Prototype Generation

B.1 The following algorithm details how to get *nodeList*, *edgeList*, and *initialNodeList* of the GT from a given StateD *sd*.

```
// returns the list of transitions in StateD sd
nodeList = sd.TransitionList()

// edgeList computation
edgeList = ∅
For each t ∈ nodeList

        s = sd.FromState(t)
        // returns the state from which
        // the transition t is originating
        List = sd.inputTransitions(s)
        // returns the list of transitions that
        // enter the state s
        For t_i ∈ List edgeList.addEdge(t_i,t)
        // case where s is an initial state of sd
        If s.type == initialState
                s_i = s.superState()
                // returns the parent state of s
                List = sd.inputTransitions(s_i)
                For t_i ∈ list
                        edgeList.addEdge(t_i,t)
                End If
                // case where s is a composite state
                // (and-state, or-state):
                If (s.type==andState) or (s.type==orState)
                List = s.transitionsInside()
                // returns the list of transitions inside
                // the composite state s
                For t_i ∈ List edgeList.addEdge(t_i,t)
        End If
End For
// initialNodeList computation:
initialNodeList = ∅
LIS=sd.initialStates()
For each s ∈ LIS
        OT = s.outputTransitions()
        // returns the list of transitions that
        // fan out from s
        initialNodeList = initialNodeList ∪ OT
End For
```

B.2 The following pseudocode details the Masking non-interactive transitions operation (the update of *initialNodeList* is not shown):

```
For each t ∈ nodeList
        If t.widget()="" then
                nodeList.delete(t)
                ITL=edgeList.inputEdge(t)
                // returns the list of transition $t_i$
                // with ($t_i$,t) ∈ edgeList
                OTL=edgeList.outputEdge(t)
                // returns the list of transitions $t_o$
                // with (t,$t_o$) ∈ edgeList
                For each $t_i$ ∈ ITL
                        For each $t_o$ ∈ OTL
                                edgeList.addEdge($t_i$,$t_o$)
                                edgeList.deleteEdge($t_i$,t)
                                edgeList.deleteEdge(t,$t_o$)
                        End For
                End For
        End If
End For
```

Appendix C

References

[1] J. S. Anderson, and B. Durney, "Using Scenarios in Deficiencydriven Requirements Engineering", *Requirements Engineering'93*, IEEE Computer Society Press, 1993, pp. 134-141.

[2] F. Bodart and J. Vanderdonckt. Widget standardisation through abstract interaction objects. In Advances in Applied Ergonomics, pages 300{305, Istanbul - West Lafayette, May 1996. USA Publishing.

[3] G. Booch. *Object Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[6] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison-Wesley, Reading, MA, 1996.

[7] T. Griffiths er.al., "A Model-Based User Interface Development Environment for Object Databases," Interacting with Computers, vol. 14, no. 1, Dec. 2001, pp. 31-68.

[8] T. Griffiths, J. McKirdy, G. Forrester, N. Paton, J. Kennedy, P. Barclay, R. Cooper, C. Goble, and P. Gray. Exploiting model-based techniques for user interfaces to database. In Proceedings of VDB-4, pages 21{46, Italy, May 1998.

[9] *Extensible Markup Language (XML) Version 1.0*. World Wide Web Consortium Recommendation 10-February-1998. http://www.w3.org/TR/REC-xml

[10] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8, June 1987, pp. 231-274.

[11] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, (16)4, April 1990, pp. 403-414.

[12] Hartson, H. R., D. Hix. Human-Computer Interface Development: Concepts and Systems for Its Management. In *ACM Computing Surveys*, 21, 1, 1989, pp. 5-92.

[13] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen, "Formal approach to scenario analysis", *IEEE Software*, (11)2, March 1994, pp. 33-41.

[14] IBM. *Systems Application Architecture: Common User Access - Guide to User Interface Design - Advanced Interface Design Reference*, IBM, 1991.

[15] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley, Reading, MA, 1992.

[16] C. Janssen, A. Weisbecker, and U. Ziegler, "Generating User Interfaces from Data Models and Dialogue Net Specifications", *Proc. Of the Conference on Human Factors in Computing Systems (CHI'93)*, Amsterdam, The Netherlands, April 1993, pp. 418-423.

[17] P. Johnson. Human computer interaction: psychology, task analysis and software engineering. McGraw-Hill, Maidenhead, UK, 1992.

[18] B. Kirwan and L. Ainsworth. A Guide to Task Analysis. Taylor & Francis, London, UK, 1992.

[19] P. Markopoulos and P. Marijnissen, " UML as a Representation for Interaction Design," Proc. Australian Conf. Computer-Human Interaction, CHISIG, 2000, pp. 240-249.

[20] N. McKay, Developing User Interfaces for MS Windows, pp 295-315. 1993

[21] Myers, B. A. User Interface Software Tools. In *ACM Transactions on Computer-Human Interaction*, vol.2, no. 1, March, 1995, pp. 64-103.

[22] Myers. B. A. and M. Rosson. Survey on User Interface Programming. In *Human Factors in Computing Systems*. Proceedings of SIGCHI'92, Monterey, CA, May 1992, pp. 195-202.

[23] B. A. Nardi, "The Use Of Scenarios In Design", *SIGCHI Bulletin*, 24(4), October 1992.

[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented Modeling and Design*, Prentice-Hall, Inc., 1991.

[25] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, Inc., 1999.

[26] S. Schnberger, R. K. Keller and I. Khriss, *Algorithmic Support for Transformations in Object-Oriented Software Development*, Technical Rep. GELO-83, Univ. de Montréal, Montréal, Qc, Canada, April 1998.

[27] Schulert, A. J., G. T. Rogers, and J. A. Hamilton. ADM-A Dialogue Manager. In *Human Factors in Computing Systems*. Proceedings SIGCHI'85, San Francisco, CA, April, 1985. pp. 177-183.

[28] Ousterhout, J. K. Scripting: Higher-level Programming for the 21st Century. In *IEEE Computer*, March 1998, pp. 23-30.

[29] Symantec, Inc, *Visual Café for Java: User Guide*, Symantec, Inc.,1997.

[30] P.A. Szekely, "Retrospective and Challenges for Model-Based Interface Development," Computer-Aided Design of User Interfaces, F. Bodart and J. Vanderdonckt, eds., Namur Univ. Press, pp. xxi-xliv.

[31] Luo, P., P. Szekely, and R. Neches. Management of Interface Design in Humanoid. In *Proceedings of INTERCHI'93*, April 24-29, 1993, pp. 107-114.

[32] Wiecha, C., W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: A Tool for Rapidly Developing Interactive Applications. In *ACM Transactions on Information Systems*, vol. 8, no. 3, July 1990, pp. 204-236.

ملخص الرسالة

بما أن شاشة المستخدم أصبحت جزءا أساسيا في نظم البرمجة (Software Systems)، لذلك فان لغة النمذجة الموحدة (Unified Modelling Language) هي اللغة المرشحة لنمذجة شاشة المستخدم (User Interface Modelling)، لاعتبارها الصيغة القياسية لنمذجة الأشياء المترابطة (Object Oriented Modelling) في التطبيقات البرمجية.

هذه الرسالة تناولت موضوع نمذجة شاشة المستخدم باستخدام لغة النمذجة الموحدة (UML) لنظام المكتبة، وكيفية توليد نموذج لشاشة المستخدم من السيناريو باستخدام لغة النمذجة الموحدة. حيث تم استخدام أدوات لغة النمذجة الموحدة في تعريف نظام المكتبة، وحاولنا من خلاله اظهار نقاط الضعف والقوة في لغة النمذجة الموحدة.

و تحتوي هذه الرسالة على الفصول التالية:

__الفصل الأول:__ عبارة عن مقدمة للرسالة، حيث تناول أسباب اختيار هذا العمل، والهدف منه، تعريف بلغة النمذجة الموحدة (Unified Modelling Language)، و سبب استخدامها، تعريف بأنواع المستخدمين، طرق برمجة شاشة المستخدم (User Interface Methods).

__الفصل الثاني:__ في هذا الفصل تم تحليل نظام المكتبة باستخدام لغة النمذجة الموحدة (UML)، حيث تم تعريف ماهي الأشكال(Diagrams) التي سيتم استخدامها لنمذجة نظام المكتبة، ابتداء من نمذجة حدود النظام (Domain Model)، نمذجة مهام النظام (Task Model)، وصولا الى كيفية ضم كل هذه الأشكال في شكل واحد هو (Package Diagram).

__الفصل الثالث:__ تناول هذا الفصل توليد نموذج شاشة المستخدم من السيناريو (User Interface Generation From Scenario)، حيث يتم الحصول على السيناريو من هندسة المتطلبات(Requirement Engineering) وتتم صياغته في شكل (UML Collaboration Diagrams) التي يتم تحويلها إلى (UML State Chart Diagram)، ثم يتم توضيح كيفية اتباع خوارزمية توليد نموذج شاشة المستخدم (Algorithm For User Interface Prototype Generation).

__الفصل الرابع:__ هو خاتمة الرسالة، ويعطي ملخص للنتائج التي تم التوصل اليها.

الجماهيرية العربية الليبية الشعبية الاشتراكية العظمى

جامعة التحدي

سرت

إن الدراسة ليست غاية في حد ذاتها وإنما الغاية هي خلق الإنسان النموذجي الجديد

التاريخ : 2009-2-5

الموافق :

الرقم الإشاري : ك ع/1423/8.0

كلية العلوم

قسم الحاسوب

عنوان البحث

نمذجة واجهة المستخدم باستخدام UML لنظام مكتبة

مقدمة من الطالبة

زهرة بن جماعة سالم

* لجنة المناقشة :

1 – د. محمد أحمد اخليف .... [illegible]

(مشرفاً)

2 – د. ادريس سامي الفقيه .... [illegible]

(ممتحناً داخلياً)

3 – د. فرج عبد القادر المؤدب .... [illegible]

(ممتحناً خارجياً)

يعتمد [illegible]

د. احمد فرج المحجوب

أمين اللجنة الشعبية لكلية العلوم

جامعة التحدي- كلية العلوم

نمذجة شاشة المستخدم لنظام المكتبة باستخدام الـ يو ام ال

رسالة مقدمة لقسم الحاسوب

كمتطلب جزئي للحصول على درجة الماجستير فى علوم الحاسب

مقدمة من الطالبة:

زهرة بن جماعة صالح ابوطريف

إشراف:د. محمد اخليف

العلم الجامعي 2007/ 2008