National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

INCORPORATING USE CASE TESTING INTO A DESIGN TOOL

Afaf Tabach

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

January 1996

© Afaf Tabach, 1996

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10903-8

Canada

Abstract

Incorporating Use Case Testing Into a Design Tool

Afaf Tabach

Many methods for object oriented analysis and design have appeared during recent years and several of them have gained popularity in the software industry. Not surprisingly, a number of design tools have been developed to provide support for these development methods. Although most of the existing tools provide facilities for describing and checking *static* design models, only a small number of them have the capability of describing and checking *dynamic* models.

In this thesis, we describe a design tool that checks the dynamic behaviour of a design based on use cases. The concept of *use case* was introduced by Ivar Jacobson; a use case is a user-initiated, behaviourally-related sequence of transactions that the system can perform. In Jacobson's work, a use case is an informal design aid. Our design tool uses the system design to check the feasibility of use cases automatically. The tool enables system designers to check both the static and dynamic integrity of their designs.

Acknowledgments

I would like to thank my supervisor Dr. Peter Grogono for his valuable input to this work and for his continuous guidance and support.

I would like also to thank my husband Ibrahim for his support and encouragement.

Finally, I would like to dedicate this work to my parents with great love and gratefulness.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Computer aided software engineering tools 2

1.2 The objective of the thesis 3

1.3 Outline of the thesis 5

2 Overview of Object-Oriented Methods 6

2.1 OMT methodology 6

2.2 OOSE/use case driven approach 8

2.3 Booch method 13

2.4 Responsibility-Driven Design 17

2.5 Shlaer and Mellor Method 20

2.6 A brief Comparison Between The Different Methodologies 22

3 Concepts of Dynamic Behaviour 23

3.1 Use Cases 23

3.2 Timethreads 26

3.3 State Charts 29

3.4 A Practical Application of Use Cases 31

4 The Original Tool 33

4.1 Design Notations and Display Forms 33

4.1.1 The Design of The Tool 39

4.1.2 Implementation Issues 43

5 The Extended Tool 47

5.1 The Testing Process 47

5.2 Design of The Tool 48

5.2.1 Example System 49

5.2.2 Design Model and Representation 56

5.2.3 User-Interface 59

5.2.4 Data structure 63

5.2.5 Parsers 65

5.2.6 Major Functional Module 65

6 Implementation of The Tool **67**

6.1 The Implementation Language 67

6.2 Abstract Syntax Tree (AST) 67

6.3 Parsers 68

6.3.1 Multiple Parsers 71

6.4 User Interface 73

6.4.1 The Major functional Module — Use Case Tester 76

7 Assessment of The Tool **80**

7.1 Advantages of The Tool 80

7.2 Limitations of The Tool 81

8 Conclusion **83**

A The Abstract Data Structure **90**

B Design Texts **95**

List of Figures

1 Different Models and Phases of OOSE 9

2 Example of a Timethread 27

3 The yacc grammar for a system 36

4 The yacc grammar for a system cont'd 37

5 Textual Form of The Heating System Design 38

6 ToolMainWindow 40

7 The Design—Edit Window 41

8 The CheckWindow 42

9 The UNIX Library Hierarchy 44

10 System, class and method C structures from AST 46

11 The design model for the recycling machine 53

12 Interaction diagram for returning items use case 54

13 The yacc grammar of the *use case files* 58

14 The use case "Returning Items" 59

15 The yacc grammar of the *interaction files* 60

16 Textual form of interaction diagram "Returning Items" 61

17 ToolMainWindow 61

18 UseCaseTesting Window 62

19 The useDialog_popup Window 63

20 Final output of *ReturningItems* use case testing 64

21 Error message displayed when consistency check fails 66

22 C structures of the system, use case and interaction diagram 69

23 The original grammar rule for **method** 70

24 The modified grammar rule for **method** 70

25 An example of the recommended style of comments 71

26 The Motif widget structure of the ToolMainWindow 76

27 The Motif widget structure of the UseCaseTesting Window 77

28 The Motif widgets structure of the UseCaseWindow—Feasibility Checking Output 77

Chapter 1

Introduction

Since the early days of computing, back in the 1940's, there has been interest in what we call now programming.

Programming languages evolved gradually from machine code, in which the language was highly dependent on the architecture of the computers, to high level languages, in which the separation between the two becomes clearer and stronger. Consequently, the availability of tools to help the programmer was very limited. At first, there were no tools at all. Then assemblers were introduced, and as the programming languages moved to higher levels, tools like compilers, editors, and debuggers become available. As applications became more complex, software development became more difficult, and improving the field of programming was not enough anymore. What was needed is the classic engineering approach: define the problem clearly, and try to find appropriate tools and techniques to solve it [8].

Since the 1960's software engineering has made clear progress. However, as the complexity of the applications continued to increase, the traditional software development was not enough, and a new software development paradigm started to emerge: Object Oriented Paradigm.

Since 1980's the object-oriented approach has been getting more and more popular,

because of its many advantages, such as reusing of software components, modularity and maintainability. In addition, this paradigm is closer to the real world and offers more powerful abstraction capabilities than the traditional approach.

Similar to the history of traditional programming, the object oriented technology started with the emphasis on developing object oriented languages, such as Simula [16] and Smalltalk [17]. Later, emphasis shifted towards developing object oriented methods, such as *Object Modelling Technique* (OMT), *Object-Oriented Software Engineering* (OOSE) and the method of Booch. These object oriented development methods have introduced many new concepts and techniques to be handled by the developer. Therefore, software engineers need now more than ever to be equipped with adequate tools to assist them and to enable them to overcome the complexity of the development task.

1.1 Computer aided software engineering tools

Computer aided software engineering (CASE) tools became popular in the 1980's along with object-oriented programming. Since 1990, the state of the software tools industry has evolved significantly. Since expectations from software tools continue to get higher and higher, tools must continuously improve and broaden in scope. For example, the following are features provided by recent tools, and which were considered earlier a gap in coverage:

- program understanding and reverse engineering
- system level analysis and design
- performance assessment and simulation
- graphical user interface.

This is in addition to the improvement of traditional areas covered by tools, such as code generation, configuration management, testing and maintenance [7].

Currently, there is a growing number of tools that support object-oriented methods. The following are examples from [9].

- System architect case tool (from Popkin Software and Systems Inc.) runs under Microsoft Windows and supports many diagramming techniques, such as ER (Entity-Relationship), state transition and data flow diagrams. It also supports diagramming techniques based on Booch's method. Coad/Yourdon method is also supported.
- Object-Maker is a meta-CASE tool (a tool to build case tools). It supports many conventional and object oriented methods, such as OMT, ADM3, and the methods of Booch, Shlaer and Mellor and others. It runs on many platforms.
- Virtual Software Factory (VSF) is also a meta-Case tool which uses Cantor, a Prolog-like formal description language, to describe methodological and structural rules. Syntactic check and semantic checks are allowed. Traceability between analysis, design and coding stages is also supported, helping with reverse engineering approaches [9].

1.2 The objective of the thesis

The tools mentioned in Section 1.1 and others studied during this research perform the tasks of code generation, consistency checking, document generation, and support graphical editors for the different kinds of diagrams used in many development methods [4].

The type of design checking performed by these tools is mostly consistency checking, in other words, they check the **static** aspect of the design, while the **dynamic behaviour** is not checked.

Therefore, the objective of this thesis is to develop an extension of an existing design tool, developed by Hanwei Ding [5], which provides facilities for creating, examining, and modifying an object-oriented design and performs consistency checking on that design.

This extension includes expressing and adding the feature of checking the dynamic behaviour of a system by means of *use cases.* Consequently, the user will be able to check if a certain behaviour can be realized by an existing design. If not—which means some checking rules are violated, such as a method does not exist or a class cannot call another class— the tool will provide the designer with useful guidelines to modify the current design in order to realize the tested behaviour. The new feature also facilitates and provides traceability between requirements and design.

Not many existing tools check the dynamic behaviour of a design; however, the following is a brief account that deals with this problem.

A study made by M. Hedlund et al. [12] aimed at showing how to use LOTOS (formal specification language) with the *objectory* development process, developed by Jacobson. LOTOS was used as a verification tool for use case designers, which means checking the completeness and consistency of use case specifications, which is similar to what we have accomplished. The use case descriptions, (use cases interaction diagrams) were translated into LOTOS templates. Then, those templates were completed by adding more information about the operation functionality and state change. Finally, the whole LOTOS specification was collected into one specification on which the verification is performed. The problem with this method is that it is done manually, so it is expensive.

Jacobson mentions a tool called Oryse [14], and states that this tool supports the OOSE/use case driven approach. We have not been able to obtain any further information about this tool.

1.3 Outline of the thesis

The rest of this thesis is structured as follows: **Chapter 2** provides an overview of some popular object-oriented development methods including OOSE/use case driven approach, which is the method used in the new extended tool. **Chapter 3** includes a description of some important concepts of dynamic behaviour and their importance in the system development. **Chapter 4** provides a description of the existing tool, which is the starting point of the extended tool. **Chapter 5** includes the design of the tool, and **Chapter 6** includes the implementation details and problems encountered during the development of the tool. **Chapter 7** provides an assessment of the tool. Finally, **Chapter 8** provides a conclusion of what was achieved and why the work is important. In addition, pointers for further research are given.

Chapter 2

Overview of Object-Oriented Methods

As mentioned in Chapter 1, many Object-Oriented methods have emerged in the recent years. However, many of these methods are still in the paper stage, some are not very well documented, and some are limited to specific applications. The following is an overview of some of the most popular methods currently used in software development.

2.1 OMT methodology

OMT methodology, developed by J. Rumbaugh and his colleagues, supports the entire software life cycle [21]. It consists of three different phases: Analysis, System Design, and Object Design.

The **Analysis phase** is responsible for presenting a clean and understandable model of the real world to describe the problem domain. The output from the analysis phase is a clear statement of what the system should do, and this provides a basis for the work on design and implementation.

The next phase is the **system design.** In this phase the designer try to find the way to solve the problem identified in the analysis. This is done by identifying the architecture framework, the topology and the subsystems of the system. It describes each subsystem, and specifies its interface with the other subsystems, in addition to various decisions, such as identifying concurrency, handling global resources, choosing software control implementation, and studying trade-off priorities.

The third phase is the **object design.** The object design determines the internal design of each class, its associations, and its methods. Data structures and algorithms are also chosen during this phase.

An important advantage of OMT is that the same notation is used during each of the three phases. Each phase is described using three models: the object, dynamic, and functional models.

Object Model: This model captures the static aspects of the system. It identifies the objects in the system, which mostly represents the real world entities. It describes their identities, relationships to other objects, attributes, and operations. The object model is represented graphically using the *object diagram.*

Dynamic Model: The dynamic model captures a different aspect of the system. It deals with time, sequence, and control of events. It describes the states that each object goes into during the execution of the system, determines the events that each object can receive and the object's response to them. It also specifies the possible scenarios that might occur between the system and its users. The dynamic model is represented graphically using *state diagrams* and *event traces.*

Functional Model: The functional model deals with aspects other than those captured by the previous two models. So far the object model identified what are the objects of the system. The dynamic model described the sequence of operations performed on these objects. Now the functional model will determine how these operations are performed. It shows the internal computation of the

input values and the resulted output values.

The functional model is represented by many *data flow diagrams* that show the input values, the data stores, the operations they go through, and the resulted output values.

The OMT methodology covers the entire life cycle span of the system development and it puts an emphasis on the higher levels of the system development —analysis and design— rather than implementation. It also uses notations and terminologies that are not dependent on any programming language.

2.2 OOSE/use case driven approach

The OOSE/use case driven approach is a method for object-oriented analysis and design [15]. OOSE is one of the earliest object-oriented methods. It was developed by Ivar Jacobson, one of the oldest practitioners of software engineering.

OOSE is a powerful method that addresses the full system life cycle. Its emphasis is on meeting the requirements of the system users, especially when the system is large. OOSE includes three development phases: analysis, construction, and testing.

- The analysis phase provides a complete understanding of the system. The output from this phase is two models, *requirement model* and *analysis model.*
- The construction phase tries to design and implement the requirement specified in the analysis model.
- The testing process verify the system according to the requirements.

According to Jacobson, system development is model building, thus OOSE tries to handle the complexity of large systems by working with different models, each of

Model/Phase	Analysis	Construction	Testing
Requirement	x		
Analysis	x		
Design		x	
Implementation		x	
Test			x

Figure 1: Different Models and Phases of OOSE

which has a different aspect of the system and belong to one of the three development phases. These models are shown in Figure 1.

In the **requirement model**, all the functionality of the system is described. This is done by using *use cases* developed in the *use case model* which is part of the requirement model. A use case model consists of *actors* and *use cases*. *Actors* represent entities that exist outside and interact with the system. An actor is different than a user. The actor implies certain behaviour, and the user can perform this behaviour in addition to many other behaviours, which means that one user can represent many actors.

Use cases represent what should be performed by the system: they describe its intended usage. Use cases can be defined as a behaviourally related sequence of transactions, performed by the users when they use the system.

The use case is a key concept in this method. It is not just another terminology, it is a way of thinking about the system development. In addition, the use case has a main role in achieving one of the most important characteristics of OOSE, which is *traceability*.

The analysis model is the first model that focuses on the real structure of the system. In this model objects are identified. There are three different types of objects: interface objects, entity objects, and control objects.

- Interface objects represent the interface between the user and the system.
- Entity objects models the information handled by the system and which must be stored and kept even after a use case is over.
- Control objects are more abstract. When a behaviour does not belong to either the interface or entity objects, like the process of distributing tasks among objects, this functionality is allocated to a control object. Often there is a one-to-one mapping between use cases and control objects. In other words, each use case is handled by one control object.

The analysis model also includes dividing the system into subsystems by grouping related objects together. These groups are called *subsystems.*

In the **Design model**, use cases are designed. The analysis objects are mapped into design blocks. The environment constraints are represented by as few objects as possible. After that the communication between the design blocks are determined, and represented by *interaction diagrams* which describes a set of objects and how they communicate with each other. For each use case there is a corresponding interaction diagram which describes how the use case is realized by the communicating objects.

The *Interaction diagram* is a type of diagram used for a long time in the world of telecommunications and it describes the communication between different blocks. Each block is represented by a bar which is drawn as a vertical line. Blocks communicate by sending stimuli to each others. Each stimulus is represented by an arrow from the sender to the receiver.

Interactions diagrams are similar to *timing diagrams* of Booch's method, which express global dynamics, and to the event traces in *OMT*, which describe the the flow of events for one scenario.

Having all the *interaction diagrams* of one system, the operations belonging to each block can be identified by extracting its interface from all the interaction diagrams it

participates in. This can be done using an automated tool.

In the **implementation model**, the blocks designed in the design model are translated into the target programming language. For example if the target language is an object-oriented language like C++, blocks are translated into classes or files, attributes that are identified in the analysis as well as the inheritance associations are directly mapped into attribute and inheritance relationship between classes. In addition, internal or private supporting classes for the design blocks are identified and implemented.

The test model aims at verifying the system. There are three levels of testing:

1. Unit testing tests each block separately.
2. Integration testing verifies that all the units are working together properly. Use cases are very useful tool for integration testing, since it involves interactions between different blocks.
3. System testing, in which the entire system is tested by performing the end-user actions on the system.

After discussing the different models and phases of OOSE we can see how use cases can achieve traceability to a great extent, and how OOSE is indeed use case driven. This observation can be verified as follows.

The functionality specified in the use cases leads to the identification of objects and the allocation of functions and roles to these objects in the analysis model. In the design model, these objects are mapped into design blocks. These blocks communicate with each others using stimuli. These stimuli are described in the interaction diagram. Thus for each use case there will be a corresponding interaction diagram that describes the use case behaviour by specifying the objects that perform this behaviour and the stimuli sent between them and in what order.

It should be noted that the concept of use cases is getting more and more attention from other methodologies. For instance, Runmbaugh (OMT), and Booch and many others are trying to incorporate it into their methods [14]. Jacobson mentions that he is happy that his work is being recognised by other practitioners, but he emphasises that the use case concept cannot just be added to another object-oriented method. Use cases are more than a notation or a concept: they are a way of thinking and understanding the system. Use cases deeply affect the way the developer is going to design the system. In addition Jacobson provides a list of advantages of use cases:

" Use cases serve several important purposes. Among other things, use cases are the basis for:

- Defining functional requirements
- Deriving objects
- Allocating functionality to objects
- Defining object interactions and object interface
- Designing the user interface
- Performing integration testing
- Defining test cases
- determining developments increments
- Composing user documentation and manuals

They also help define the system and control development by serving as the vehicle for:

- Capturing and tracing requirement
- Envisioning an application

- Communicating with end-users and customers
- Delimiting a system
- Estimating project size and required resources
- defining database-access patterns
- Dimensioning processor capabilities "

In short he says, "use cases are the *dynamics, black box view of the system, and the driver of development activities*" [14].

Jacobson also says that use cases are not the "magic solution" for software development: many other concepts are as important to get a successful software development object modelling for example.

One weakness of OOSE is that it assumes that all sequences of actions that might be performed by the users of the system can be expressed using the interaction diagrams. However, this can be difficult to achieve with complex systems, since there may be a large number of possible sequences of transactions.

2.3 Booch method

According to Booch, in order for a system design to be completely captured it should be studied from different views. In his method [1], he defines six diagrams that capture the logical and the physical view of the design. These are the:

1. Class diagram
2. Object diagram
3. Module Diagram

4. Process diagram
5. Timing diagram
6. Transition state diagram

The first two diagrams describe the logical view of the system which means defining the main classes and objects of the system and their semantics. The third and fourth diagrams describe how the classes and objects are distributed in the physical implementation of the system. And the last two diagrams describe the dynamics of the system. Each one of these diagrams is described briefly as follows:

Class diagram: This diagram captures the existence and the relationships, between the classes of the system. There are four main notations used in a class diagram.

- Classes are represented by the shape of amorphous blob or "cloud". Its borders are dashed lines to indicate that operations are performed on instance of that class.
- Class relationships are represented by different types of lines drawn between classes. Each type represent one kind of relationship. These relationships are, *uses, instantiates, inherits, metaclass* and *undefined* (which indicates the existence of some relation between two classes but the type of this relation is not precise yet).
- Class utility represents one or more free subprograms. It is represented by an icon that is very similar to the class icon but distinguished by a shadow around the borders. *Uses* and *instantiates* are the most frequent relationships used between class utilities.
- Class category: Sometimes a system has so many classes that they can not be put all together in one class diagram. Thus, it is logically divided into different chunks, which are called *class categories.*

Each class can also be described by a *class template*. A class template is a textual form used to describe the class, its visibility, cardinality, its hierarchy, its operations, and many other things. A *class utility* also has a template.

State diagram : This diagram captures or describes the dynamic behaviour of each instance class of the system. For each class, the state diagram describes the different states the class goes through, and the events that cause the transition from one state to another. There are two notations used in the state diagram:

- State: Each single state is represented by a circle, which contains the state's name.

- State transition: This is the only type of relationship between different states. It is represented by a directed line from one state to another or to itself. A state transition diagram has a template which list the different events it receives, its documentation and the *action*.

Object diagram: This diagram shows the object structure of the system—the objects existence and their relationships. An object diagram describes the dynamic semantics of the operations, since it shows what objects are created and destroyed during a snapshot of execution, unlike the class diagram that shows the static relationships between classes. However, those two diagrams are strongly related, since an object represent one or more instance of a class, and the two diagrams should have consistency in the use of methods and operations.

Usually, one system is described by more than one object diagram. There are two important concepts in an object diagram:

- objects are represented by an icon very similar to the class icon but with solid lines. The object icon can contain the object name, in addition to some properties like concurrency and persistence.

- Objects relationships are drawn between objects with no arrow to indicate the possibility of bidirectional messages.

An object also has a template which includes the object name, its class and its persistence property.

Timing diagram: This diagram is used to express what cannot be expressed by the object diagram alone. It describes the flow of control, and the sequence of events. Timing diagrams have two axis, one for the time and one for the objects. One object diagram may have zero to many timing diagrams.

Module diagram: This diagram is used to document the physical design. Module diagrams show the allocation of classes and objects to modules in the physical design of a system. Module architecture can be expressed by one or more module diagram. Module diagrams consist of modules which are represented by different types of icons.

Process diagram: This diagram describes the process architecture, by sharing the allocation of processes to different processors, describing the use of different devices and processes and showing their connection.

All these diagrams constitutes only the *notation of the method* and not the process. *The process* proposed by Booch is incremental and iterative. It is neither top-down nor bottom-up, but " Round-trip Gestalt design".

In every iteration, the steps of the design method are applied on a low level of abstraction until the design and implementation of the whole system are complete. The following are the steps of the process:

1. Identification of objects and classes.
2. Assigning the meaning of each class and object, which includes updating the class and object diagrams.
3. Identifying the relationships among classes and objects which means a possible completion and refinement of object and class diagrams.

4. This step is concerned with the representation of the objects and classes, which are already identified. It also includes the allocation of classes and objects to modules including the refinement of most of the templates, the module diagrams, and the process diagrams if needed.

The fourth step may not be the last one. As mentioned earlier, this is an iterative process, thus if necessary, another iteration will take place but on a lower level of abstraction.

The difference between Booch's method and OMT is that, unlike OMT, Booch puts more emphasis on design and less emphasis on analysis. However, it must be noted that the two methodologies have recently been combined.

Some of Booch's method advantages are prototyping, and the continuous refinement of object model, due to the different iterations during the system development.

2.4 Responsibility-Driven Design

The responsibility-driven approach is a design method proposed by Wirfs-Brock [23]. Some terminology used in the design process should be explained in order to understand the process.

- Abstract classes are different from concrete classes because they are not identified to produce instances of themselves, but to capture a common behaviour that can be inherited by other classes.
- Class responsibility contains all the operations that each class is responsible for performing and the data that it should store and know about.
- Collaborations are a list of communicating objects; each collaboration includes the names of one object calling another object.

- Clients and servers are specific roles that can be played by objects. An object is a client if it is calling another object and the server is the object being called to perform certain operations.
- Contract is included in the server object and it is a list of the only operations that the client object is allowed to request from the server objects.

The design process of the responsibility-driven approach includes the following steps.

1. **Finding classes.** The method gives many guidelines to find candidate classes. For example, identify all the noun phrases from the requirement specifications, model physical entities and known interfaces, and many others.

   After identifying and refining the classes, the method suggests recording these classes by using index cards. Each class has a card and on each card there is a short description of the class. This is followed by identifying abstract classes and superclasses.

2. **Identifying the responsibility of each class of objects.** The responsibility of a class includes the knowledge that it should maintain and the operations that it should perform. Again, the method suggests many guidelines to identify responsibilities. For example, extract verbs from requirement specifications and associate it with responsibilities using common sense, perform a walk-through of the system by inquiring different scenarios of the system, which is similar to the use case concept, and what are the necessary actions required to meet these scenarios, etc.. Then record responsibilities of each class on the index cards created in the first step.

3. **Identifying collaborations between classes.** This may be identified by checking the responsibilities of each class for dependencies. If a class is responsible for an action but does not have all the necessary information to perform it, this implies that it needs to collaborate with another class.

4. This step is concerned with **analysing the design created** so far to benefit the most from object-oriented technology. This is done by examining the structure of the inheritance hierarchy which can be achieved by using many tools:

   - Hierarchy graph: It represents graphically the inheritance relationship between classes.
   - Venn Diagram: This diagram considers classes as sets of responsibilities and it draws those sets in such a way that common responsibilities can be factored out.
   - Contracts: after the examining class hierarchy and factoring responsibilities, the contract of each object can be specified, which is a good way to state explicitly a group of responsibilities offered by one class. One class can have many contracts. Subsequently, each class card is modified to reflect the final design and to include the contracts and their responsibilities.

5. **Identifying subsystems** by grouping together the classes that collaborate closely together to perform a part of the whole system functionality. A subsystem has a contract which is the collection of all the responsibilities of the subsystem classes and which are requested from the outside of the subsystem. Each subsystem has a subsystem card similar to the class card.

6. **Identifying protocols**. After refining the classes and the subsystems, contracts can be formalised into protocols, which are the set of the signatures of methods in the contract.

Wirfs-Brock's approach is useful and effective for small systems but, because of its manual and informal nature, probably unsuitable for large systems.

2.5 Shlaer and Mellor Method

Shlaer and Mellor's object analysis method was first introduced in 1988 [22]. But at that time, the method was not regarded as object-oriented for many reasons. For example, there was a complete absence of the inheritance concept. However, in 1991 Shlaer and Mellor introduced a new version of their method where they introduced inheritance (entity subtyping) and the idea of modelling the entities life cycle with STDs (state transition diagrams) [9] .

Shlaer and Mellor claim that their method is mainly for the analysis phase of system development. The first step in this method is developing an information model. The idea of the information model is to focus on the real world by abstracting what is in the problem domain, and organising this information into a formal structure. This information is abstracted in the form of objects, their attributes, and their relationships. The method provides some guidelines for identifying this information.

Different relationships between objects are described using entity-relationship (ER) notation.

The information model describes only the static aspect of the system. In order to describe its dynamic behaviour, the method uses state transition diagrams to describe the life cycle of each object in the information model.

State transition diagrams (STDs) consist of *states*, *events*, and *actions*. Transition from one state to another is triggered by an event and in each state an action is performed. Furthermore, the actions described in the state model, are further explained as processes in the data flow diagrams.

Finally, in order to summarise the whole dynamics of the system, an object communication model (OCM) is introduced to show the source and target of each event. Each state will be represented by an oval box, and the events which they exchange are represented as arrows [20].

According to Shlaer and Mellor, there is more than one right software development process. Consequently, whatever process is used, the information model developed will affect and influence all the stages of development. There are many key rules that affect the semantics of the different models developed using this method [20]. These rules include:

1. The method assumes objects work concurrently and execute in syncronization.
2. Objects communicate by events. An object responds to an event according to its current state and usually an event causes a state transition and an execution of an action.
3. An object can execute only one action at a time.
4. One object can receive many events/requests simultaneously, and because of the previous rule, it can only respond to one event at a time. Thus, the method introduces a monitor object that handles the multiple requests, and serves one of them according to some rules embodied in the object.
5. When an event causes the execution of one action, this action can cause the execution of other actions. This is called triggering a thread of control.
6. Sometimes a state action requires the knowledge of some data attributes of other objects. In order not to violate the concept of data encapsulation of object, there is an accessor object, which handles the requests of accessing internal data.
7. Not all the objects handled by one method are supposed to be active. Some objects are passive, i.e. they do not have a lifecycle, such as objects which abstract data information.
8. The method is applicable mainly to real time applications with great consideration for the time factor.

2.6 A brief Comparison Between The Different Methodologies

By looking at this overview, it becomes obvious that the leading development methodologies have many similar concepts and techniques. And it becomes clearer why these methods are merging and reaching a consensus [14].

An example of this, is the recent merging of three of the most popular object-oriented methods, **OMT**, the **method of Booch**, and **OOSE** .

This implies that the object-oriented approach is maturing. And one of the object-oriented concepts that is having a great consensus from many methods is the *use case* concept.

This converging between the methodologies is very encouraging, and it might eliminate the overlapping work, and encourage instead the exchange of important knowledge and experiences of many practitioners, in order to elaborate deeper on more advanced and complicated issues facing the development of object-oriented techniques.

Chapter 3

Concepts of Dynamic Behaviour

Whatever method is used to develop a system, it is very important to be able to describe the way in which it will behave, especially for testing purposes [19]. The software development methods mentioned in Chapter 2 describe the usage and behaviour of a system in different ways. Each method has different concepts, for example, OMT uses *scenarios*, OOA uses *threads of execution*, and OOSE uses *use cases.*

The following sections contain descriptions and analysis of these concepts, in addition to an introduction to our work and how it is influenced by them.

3.1 Use Cases

Use cases were introduced by Jacobson [15]. They offer advantages in all stages of software development from requirements definition to integration testing. Use cases have been widely accepted and have been incorporated into several object oriented design methodologies.

Use cases have two principal roles. First, they focus on the clients' requirements and the actions of end users. Second, they have a major influence on building the different

object models of the system, thus providing *Traceability* [19].

Use cases are used by many groups of users, from **outside** and **inside** the system [13].
Users from outside the system are :

- End users of the system, who need to verify that the use cases really specify all their needs from the system.
- User-interface analysts, who need to specify user-interface support of each use case.
- Technical writers, who need to write the user's manual at an early stage.
- Function managers and project leaders, who need to supervise the system at different levels.

Users from inside the system are:

- Designers of the object model, who need to harmonise the design and resolve conflicts among use case instances in the analysis and design phases.
- Testers, who need use cases for use case testing.

A workshop held during OOPSLA'92 [19], software designers compared their experiences with use cases and interaction diagrams.

D. Bennet focuses on the behaviour allocation in use cases. He analyses the behaviour allocation decisions made in the use case model. One of his techniques to evaluate the behaviour allocation is to collect the interfaces of each object and show it on separate diagrams, which enables the designer to study and evaluate the cohesiveness and consistency of each object.

Betz et al. discuss the use of interaction diagrams for use cases. They believe that interaction diagrams enhance the understanding of object-oriented systems; however, they have many drawbacks :

- Loops and conditions are hard to show.
- Messages to **self** are hard to show.
- Anything that isn't a message is hard to show. In addition, creation and construction methods don't have convenient receivers.
- Return values are also problematic. If they are shown as tokens under a message, they will be appearing too early. On the other hand, if they are not shown, it will be difficult to discuss subsequent messages.

In addition, Betz et al. found that the *act of developing* an interaction diagram was more valuable than the interaction diagram itself. Moreover, they found that, once interaction diagrams had been completed, it was difficult to maintain consistency between the interaction diagrams and the code.

Stephan Wallin, discusses the usage of use cases within the design of large real-time applications. He believes that use cases or similar concepts are necessary for a complete understanding of the system, and that documentation of the objects alone is not enough. He also believes that dividing a large system into layers facilitates its understanding. Each layer is assigned to a different group of the development organisation, thus there will be more than one person responsible for each use case. Consequently, Wallin believes that it is very important for behaviour description to express use cases at the level of subsystems rather than at the level of classes.

A major problem encountered by Wallin was to decide the level of granularity of events that should be used in the description of use cases. Moreover, they found that it is not feasible to use interaction diagrams to describe all of the events of a system. Instead, they decided that an interaction diagram should be used only when it adds

clarification to the problem and when the communication between objects is very complicated.

As a conclusion to his study, Wallin found both advantages and disadvantages with use cases. The disadvantages are :

- Interaction diagrams cannot cover all sequences.
- There is a risk that the system becomes functionally structured instead of object structured.
- Some behaviours, such as screen updating, must be deduced from the requirements.

The advantages of use cases include:

- The system will meet the user's requirements.
- Use case testing can limit the extent of an object model

It is clear from the forgoing discussion that the use cases have both advantages and disadvantages. Since use cases have been widely accepted by the software development industry, however, we can safely assume that their strengths outweigh their weaknesses.

3.2 Timethreads

Timethreads are a new design concept introduced by R.J.A. Buhr [3] to help in capturing the overall behaviour of a system design. Timethreads are represented by a curve traversing the system components. Triggered by certain stimulus, the timethread

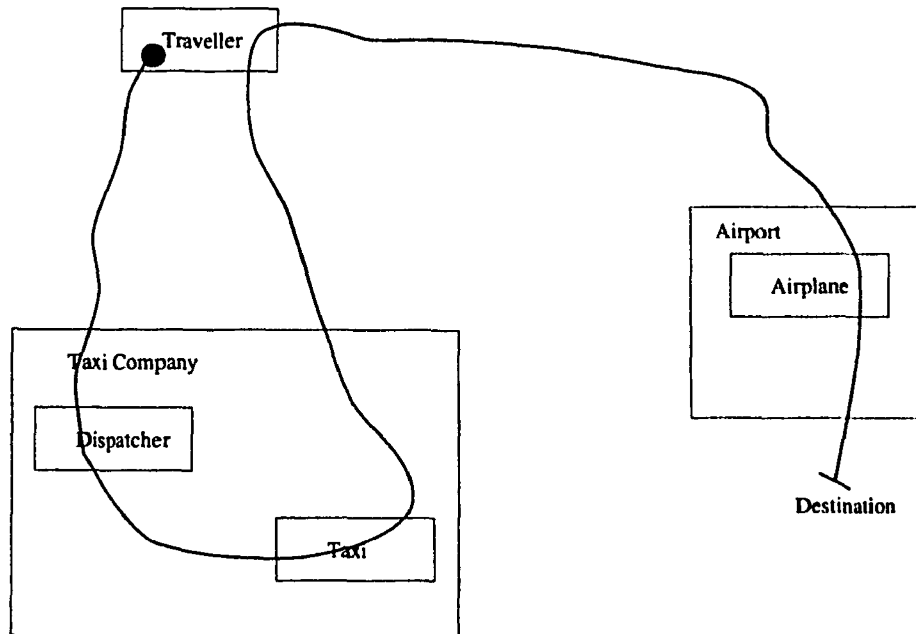

Figure 2: Example of a Timethread

starts at one component and continues through the system until the response action to the stimulus is terminated.

Timethreads describe the *causality flow* of the system. Buhr claims that causality flow is a higher abstraction than *data flow* and *control flow*. It describes the "big picture" of the system behaviour, from the starting point to the end point, without getting into the details. This is very useful for large systems where control is distributed over different objects.

Figure 2 shows an example of a timethread. This example is taken from [3]. The timethread describes the causality flow scenario of a traveller how calls a taxi company to send him a taxi to take him to the airport in order to catch his airplane.

Usually, in order for the developer to express the overall behaviour of the system, he has to understand the behaviour of each of the individual components. This is not very practical, especially when the designer is still in the early stages of reasoning about the behaviour of the system.

Timethreads overcome this problem by allowing the designer to decide **which** components will realize the functionality without knowing **how** they are going to do it.

Buhr claims that timethreads are not just for design understanding and design analysis, but also for **design discovery**. Timethreads not only document the different scenarios of the system, but they help the designer in identifying the components of the system and in allocating their behaviour.

One of the main advantages of timethreads is **informality**. This feature gives the designer less restrictions in expressing the behaviour of the system. However, they cannot be considered as complete specifications, and as a result, they cannot be used for code generation.

To give a clear idea of how timethreads are represented, here are some basic **timethreads notations:**

- The start of a timethread begins with a filled circle and ends with a line segment across the timethread curve.
- Between the start and the end points, there is a continuous curve which represents the timethread body.
- When the curve cuts across one component, this implies that those components have a certain responsibility and functionality in responding to the cause of the timethread.
- The component that touches the start circle is the one responsible for accepting the stimulus and starting the timethread.

Finally, there are many hidden things in the timethread which are not shown by the notations, such as data flow, states, and other details which are pushed to a lower level of abstraction. Only a very high level of abstraction is shown to provide an optimum, understandable and clear picture of the system behaviour.

3.3 State Charts

State charts, developed by David Harel [11], provide a graphical descriptions of the behaviour of large reactive systems.

Unlike transformational systems, reactive systems are event driven. The description of their behaviour requires a listing of all the possible *stimuli* and *events* the system can have. State charts were developed to provide a solution to such problem by describing the behaviour of large reactive systems in a clear, realistic, yet formal way.

The development of state charts is based on improving and modifying the *finite state machine (FSM)* and its state diagrams to better suit large and complex systems. Many developers of such system have avoided the use of FSM and their state diagrams because FSMs have a number of drawbacks:

1. State diagrams do not allow the representation of either depth, modularity, or hierarchy concepts.

2. State diagrams are uneconomical when it comes to transitions. If an event causes the same transition to many states, the same transition has to be shown attached to each of these states.

3. It is not feasible to describe concurrent systems with state diagrams. A the number of possible events of a concurrent system grows linearly, the number of states grows exponentially. Representing a large system be a state diagram becomes impossible.

Therefore, Harel developed a *state chart language* as an extension to state diagrams with many features that offer solutions to the drawbacks listed above. Harel describes state charts as follows :
" State charts = State diagrams + depth + orthogonality + broadcast communication " [11].

Depth is provided by allowing states to be included in other states, thus showing the hierarchy and the modularity, which solves the first problem. In addition, this feature allows common transitions to be shared between the states, which solves the second problem.

Orthogonality is achieved by allowing one state to have an AND decomposition. For example, one state A can be decomposed into two sets of states B and C which are called *orthogonal components*. Entering state A implies being in states B and C simultaneously. Such decomposition can factor out common states and reduce their number which solves the third problem.

Broadcast communication allows an external event to cause a transition, not only in one state, but in all orthogonal components to which it is relevant.

These are the basic features of state charts. Additional features include **"not-sure"** transitions and states. The "not-sure" states means that there is identified state triggered by an identified transition but the resulting state is not identified yet, then it is allowed to call this state "not-sure".

In conclusion, state charts are important because they describe the **behaviour of the system** in terms of its interactions with the "outside". It should be noted, however, that this description does not describe the activities inside the system following each transition from one state to another, but only describes the flow of control.

State charts are important because they form a visual formalism that describes computer-related systems and their behaviour. **Visual** because it can be analysed and comprehended by humans and **Formal** because it can be interpreted by computers.

3.4 A Practical Application of Use Cases

The work described in this thesis is based on an existing design tool for object-oriented development [5]. The objectives defined for the original tool were:

- Provide automated assistance for the object-oriented program development, with a strong emphasis on the design phase.
- Help the designer of object-oriented programs to maintain the system, by keeping all the development phases consistent during the system evolution.
- Facilitate the work of the designer, by making his/her job easier and quicker through many automated design functions.

The tool provides the following facilities:

- Read, write, modify, and display the design.
- View the design in different display forms.
- Perform consistency and completeness checking of the design.
- Generate high-quality printable reports, documenting the design.

The tool is organised around the *object model.* It considers the system as a collection of objects and the design is represented as a collection of class interfaces.

The consistency and completeness checking performed by the tool validate the **static design** of the system. Furthermore, the *functional* model can be deduced from the object model by examining the description of the methods in each class. However, the *dynamic model* cannot be extracted from the object model. Thus, a natural extension to this tool is to provide it with **support for dynamic modelling.** Therefore, the objective of this thesis is to modify the existing tool by providing it with the ability to

express, represent and test the dynamic behaviour of the system by means of *use cases*.

The new tool supports the OOSE/use case driven approach development method and it provides the following features:

- Read, write, and display use cases.
- Read, write and display interaction diagrams in a specified textual form.
- Test use cases, by checking if the existing design can execute the sequence of behaviour specified by the tested use case.
- Generate a printable report containing the output of the use case test.
- Provide traceability between design and requirements, by providing useful information to modify the existing design upon its failure in a use case test.

In the next three chapters, we describe the original tool and the extensions that we have added to it.

Chapter 4

The Original Tool

In this chapter, we describe the design tool constructed by Hanwei Ding [5]. This tool provided a starting point for our tool, which we describe in Chapter 5 and 6.

The tool is intended to be used for creating, modifying and checking an **Object-Oriented design**. Its objectives and facilities are listed in Section 3.4. This section includes a description of its design structure and some important implementation issues.

4.1 Design Notations and Display Forms

The design notation used in this tool is not based on a particular design methodology; however, it is biased towards the **Responsibility-driven design method** developed by Wirfs-Brock [5]. The design is built around the **object model**. Each system design consists of a list of class descriptions. A class can be described at different levels of detail. The minimal description of a class includes just its name, while the full description includes, in addition to its name, its superclasses, subclasses, variables, and methods.

The tool assumes the following properties of a design:

- The design consists of a number of classes. Each class contains a number of methods.
- Classes have two types of relations: *uses* which is a client/server relation and *inherits* which is a superclass/subclass relation.
- In a client/server relation, the client class uses one or more methods provided by the server class.
- Cycles are allowed in the *uses* relation. For instance, if class A uses class B, B may use A.
- In the *inherits* relation, the subclass provides all the methods inherited from its in addition to a number of its own methods.
- Cycles are not allowed in the *inherits* relation.
- There are three types of method:
    1. *Constructor*, which creates a new object of its class.
    2. *Observer*, which returns an attribute of the current object.
    3. *Mutator*, which modifies an attribute of the current object.
- A class is called mutable if at least one of its methods is a *mutator*, otherwise it is called immutable.

The tool provides two forms of design display: The **textual** and the **tabular** forms.

The Textual Form is the input form and the main and default display form used in the tool. The textual form is represented by a concrete grammar which contains many syntax rules that the textual form has to follow. This grammar enables the parser, provided by the tool, to scan, parse, and collect all the data provided by the design. Those syntax rules are:

- The keywords of the design are: **system, class, inherits, uses, var, method,** and **end.** Keywords must be written in lower-case characters.
- Each system design has a name list of classes containing at least one class.
- Each class has a name, an optional list of superclasses, an optional list of subclasses, and an optional list of variables.
- Each class contains zero or more methods. Each method has a name and an optional list of arguments which follows the method name and is enclosed in parentheses. Each argument has the form of ⟨name⟩ : ⟨type⟩ . It also has an optional return type separated from the arguments list by a colon.
- Each method has an optional list of used methods which begins with the keyword **uses** followed by a ⟨name⟩ :: ⟨type⟩ pair. The name indicates the method being used and the type indicates the class that provides the method.
- Comments in the design begin with two dashes "-- " and continue to the end of the line. Consecutive lines of comments are considered as one comment. Comments are allowed after the system, class, variable, and method names.

To provide a clearer idea of the textual form we provide the yacc grammar for a system in Figures 3 and 4. Figure 5 is an example of a design in textual form. It is a design for a program that simulates a heating system. For details see [5, pages 24 - 28 and 30 - 32].

The **tabular form** is the second type of display forms provided by the tool. Tables are displayed upon request from the users. They are considered as a useful tool to describe the system. They are efficiently processed by computers and they enable the user to have a clear, quick, and understandable view of the system design.

There are two types of tables. Each one provide a different level of detail about the design:

```
system
     :  SYSTEM IDENTIFIER
        opt_comment
        class_list
     ;

  class
     :  CLASS IDENTIFIER
        opt_comment
        opt_inherits
        opt_class_use
        opt_var
        opt_method
        END IDENTIFIER
     ;


  class_list
     :  class
     |  class_list class
     ;

  opt_inherits
     :  /* nothing */
     |  INHERITS name_list
     ;

  name_list
     :  IDENTIFIER
     |  name_list opt_comma IDENTIFIER
     ;

  opt_class_use
     :  /* nothing */
     |  USES name_list
     ;

  var
     :  VAR IDENTIFIER
        COLON IDENTIFIER
        opt_comment
     ;

   opt_var
     :  /* nothing */
     |  var_list
     ;

   var_list
     :  var
     |  var_list var
     ;

   opt_method
     :  /* nothing */
     |  method_list
     ;

   method_list
     :  method
     |  method_list method
     ;
```

Figure 3: The yacc grammar for a system

```
method
  :  METHOD IDENTIFIER
     opt_para_list
     opt_type
     comment_list
     opt_uses
  ;

opt_para_list      /* argument list */
  :  /* nothing */
  |  LBRACE para_list RBRACE
  ;

para
  :  IDENTIFIER
     COLON
     IDENTIFIER
  ;

para_list
  :  para
  |  para_list opt_comma para
  ;

opt_type
  :  /* nothing */
  |  COLON IDENTIFIER
  ;

opt_uses
  :  /* nothing */
  |  uses_list
  ;

uses_list
  :  uses
  |  uses_list uses
  ;

uses
  :  USES IDENTIFIER
     DOUBLECOLON IDENTIFIER
  ;

opt_comma
  :  /* nothing */
  |  COMMA
  ;

opt_comment
  :  /* nothing */
  |  comment_list
  ;

comment_list
  :  COMMENT /*
  |  comment_list COMMENT /* multiple line comment */

  ;
```

Figure 4: The yacc grammar for a system cont'd

```
system HeatingSystem
-- an OO design for a domestic heating system

class View
-- An instance of a view class can display chaging values of
-- several floating-point variables. This class, View, is
-- abstract: it defines the minimal protocol for a class that
-- provides views. A view has several channels, each corresponding
-- to a particular variable, channels are initialized and updated
-- independently in any sequence. The display is updated as a unit.
method init (channel:Int value: Float)
-- initialize the given channel with the given value.
method set-val (channel:Int value: Float)
-- Update the given channel with the given value.
method set-title (channel: Int title:String)
-- Provide the given channel with the given title.
method calibrate (value:Float str-value:String)
-- A calibration point is used to label the axis of graph or to
-- perform a similar service for another display mode. this method
-- uses the given value to position the calibration mark
-- and writes the string at that position.
method update
-- Update the display of all channels.
method message (txt:String)
-- Display a message other than channel data
method close
-- Null method to close display. Descendant classes which require
-- a closing action should redefine this method.
end View

......
```

Figure 5: Textual Form of The Heating System Design

System-level tables contain design information about the whole system. This includes the list of its classes. Each class has an entry which contains the class name, the list of its superclasses (*inherits* list), list of its subclasses (*uses* list), and its description. In addition, the system name and its description are displayed next to the table.

Class-level tables contain information about one class. This includes the list of its methods. Each method has an entry, which contains the method name, its return type, parameters, used method and description. In addition the name of the class and its description is displayed next to the table.

4.1.1 The Design of The Tool

This section contains the design structure of the tool: the representation of data, the way it is handled and processed; and the major functions provided by the tool. The design has four major parts:

1. The Data Structure, which is an abstract syntax tree (AST)
2. Parser module
3. User-interface
4. Major Functional modules

The **abstract syntax tree (AST)** is the central data structure of the tool. Since the design is a list of classes description, the AST is designed to hold an arbitrary number of classes. Each class may have an arbitrary number of variables of methods.

AST is the center of communication between all the parts of the tool. It is generated by the parser using the input file and all of the functional modules of the tool use it to perform their actions and generate their outputs.

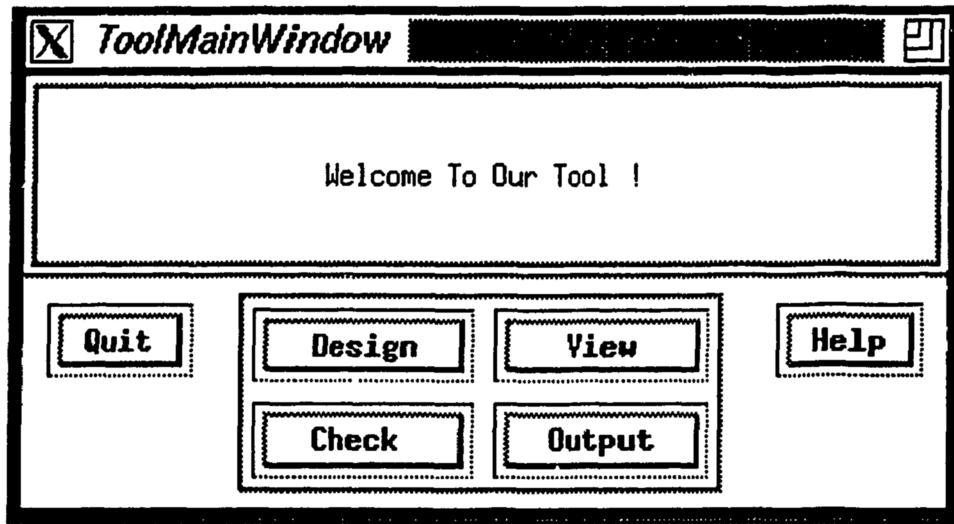

Figure 6: ToolMainWindow

The **parser** module consists of three parts:

- The *scanner*, which scans the input and generate tokens for the syntax checker.
- The *parser* which checks the syntax of the input file.
- The *constructor* which is a module for obtaining the data and constructing the AST.

User-interface module: the user-interface of the tool is user-friendly, and its design follows many important principles, such as ease of use, consistency, and simplicity. The user-interface consists of different windows:

- The **ToolMainWindow** is the first window displayed upon invoking the tool. It has a message part to welcome the user to the tool, and a selection area where each one of the tool main actions is represented by a push button : *Design, View, Check*, and *Output*, where *Output* means generating a high-quality printable document about the design. In addition, there is a push-button for help and another one to quit the tool. This window is shown in Figure 6.

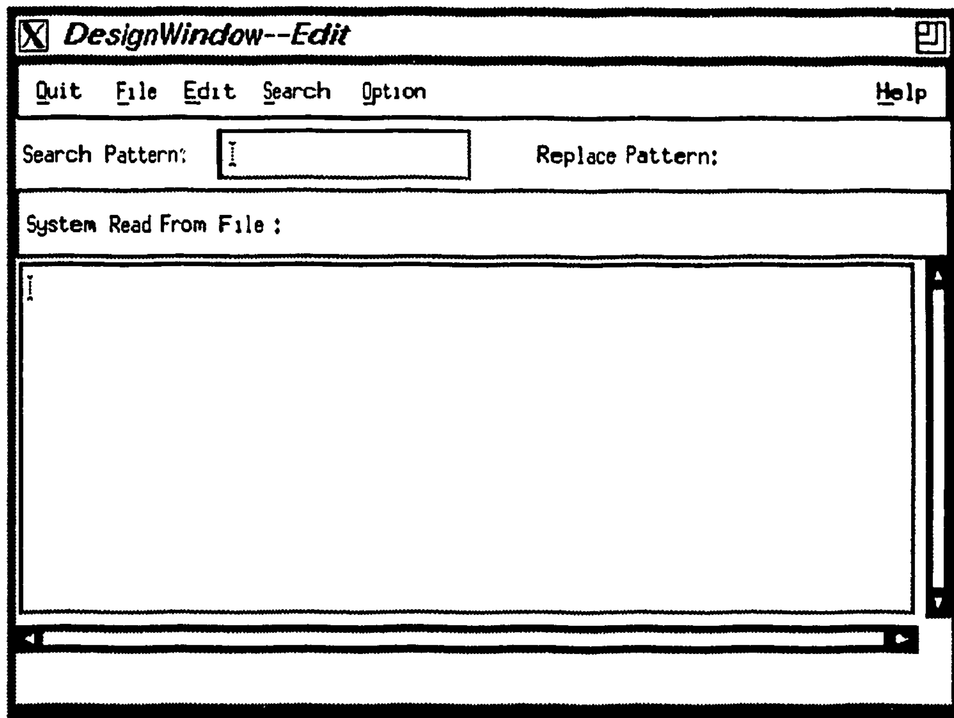

Figure 7: The Design—Edit Window

- The **DesignWindow** pops up upon choosing the Design button from the ToolMainWindow. This window has two active push buttons: *Edit* and *Quit*, and two inactive buttons: *Query* and *Help*. The *Edit button* invokes **DesignWindow–Edit**. This window provides the user with text editing facilities. It has six pull down menus: **Quit, File, Edit, Search, Option and Help.** Each pull down menu has different actions that the user can invoke. The file menu has two options **save** and **open** which allow the user to open an existing or a new file for a system, edit it and save it. This window is shown in Figure 7.

- The **ViewWindow** pops up upon invoking the push button *View* in the ToolMainWindow. This window has two active push buttons: *Quit* and *Table*. The *Table* button invokes the **ViewWindow—Table**. This window has three pull down menus: **Quit, TableView, and Options.** The **Options** menu has one option for clearing the screen. The **TableView** menu has two options: **System** and **Class.** The system option displays a

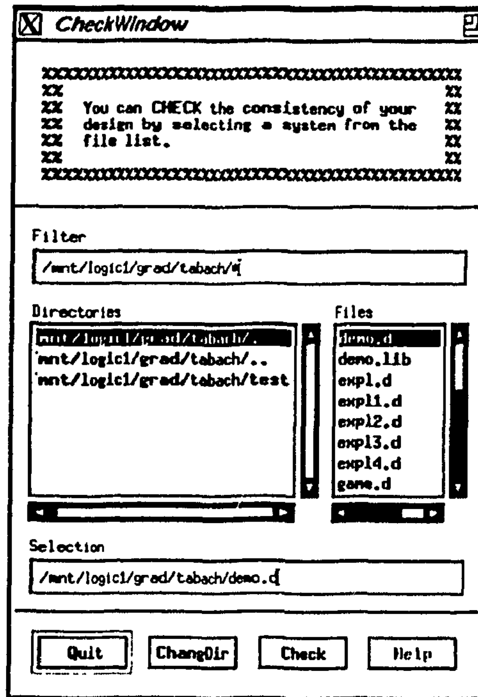

Figure 8: The CheckWindow

system-level table and the class option provides a class-level table. Both tables are discussed in Section 4.1.

- The **CheckWindow** pops up upon invoking the push button *Check* in the ToolMainWindow. This window has a file selection part where the users can select the system to be checked. It also has three active push buttons: *Quit, ChangeDir,* and *Check.*

  The *Check* button invokes a module for consistency checking. The system selected is checked and if its consistency check is successful, a message will be displayed to inform the users. Otherwise, another window will pop up containing information about the errors reported by the consistency check. Figure 8 shows the **CheckWindow** appearance.

- The **PrintWindow** pops up upon invoking the push button *Output* in the ToolMainWindow. This window has a file selection area where the users can select a system to generate its output. The output is a high-quality printable document containing the design of the system selected.

- The **HelpWindow** pops up upon invoking the push button *Help* in the ToolMainWindow. This window provides help for the user by providing information about the tool and the different options in the ToolMainWindow.

The **Major Functional modules** of the tool are : **the Text Editor**, the **Viewer**, the **Checker**, and the **Printer**.

The **Text Editor** module provides the basic facilities to edit a file, and save it. Files of the system design have an extension **'.d'**.

The **Viewer** module provides the tabular display form explained in Section 4.1. It provides two levels of tables the system-level and the class level. It uses AST to get information about the classes and methods to be displayed in the table.

The **Checker** module checks the consistency and completeness of the design. Consistency and completeness means that every class and method used in the design is defined in the system or in the standard library which has the same name of the system but with an extension **'.lib'**. The checker uses the AST which is generated by the parser using the the input file selected by the user, to perform the consistency check.

The **Printer** module converts the design file into LaTeX code to generate a high quality readable document.

4.1.2 Implementation Issues

Some of the implementation details about the existing tool are described in this section, because they are crucial to all parts of the existing tool and they affect our extension to the tool. In particular, the implementation of the AST and the parser are discussed, in addition to the implementation language.

Implementation Language and environment

C is the implementation language of the tool and Motif is the toolkit used for building the user-interface.

Motif is a collection of user-interface objects called *widgets* [2]. The Motif widget set includes objects that the user expect to find in a graphical user interface, such as pull-down menus, dialog boxes, and scroll bars.

Motif is a part of a UINX library hierarchy which has four layers as shown in Figure 9.

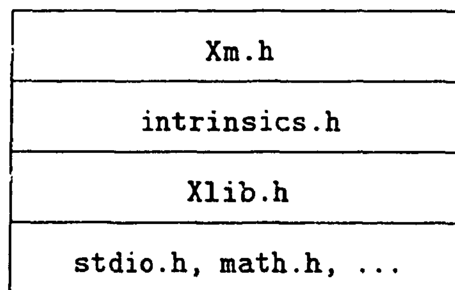

Figure 9: The UNIX Library Hierarchy

stdio.h and math.h. are the Unix libraries; XLib.h is the X library; intrinsics.h is the X toolkit library, and Xm.h is the Motif library. On top of UNIX sits the X window system, which is accessed through XLib.h. On top of X sits the X toolkit accessed through intrinsics.h. And finally, comes Motif which is accessed through Xm.h.

Advantages of Motif include:

- Motif facilitates the job of the programmer enormously, because using X windows directly to create windows and draw in them can be a very cumbersome job. Instead, Motif provides a set of preconstructed user-interface objects which can be placed on the screen by an application program.
- Motif provides consistency in the user-interface appearance.

Abstract Syntax Tree

The Abstract Syntax Tree (AST) is the central data structure of the tool. Although AST is generated by the parser, it is not a parse tree. It is made up of a list of data structures representing classes.

- The system is represented by a defined structure with the fields: the system name and its description, and a pointer to a list of classes.
- Each class is represented by a defined structure with the fields: the class name and its description, a pointer to the list of the inherited classes, a pointer to the list of used classes, a pointer to the list of variables, and a pointer to the list of class methods.
- Each method is represented by a defined structure with the fields: the method name and its description, its return type, a list of arguments $\langle$name$\rangle$: $\langle$type$\rangle$ pair, a list of uses $\langle$method$\rangle$ :: $\langle$class$\rangle$ pair and a pointer to the next method definition.

The definition of the system, class, and method structures are given in Figure 10

Parser

As mentioned in Section 4.1.1, the parser module consists of three parts: the **scanner** for generating tokens; a parser which uses the tokens to check the syntax of the input according to some grammar rules; and a constructor which obtain the data returned by the parser and construct the AST.

The **scanner** is responsible for the lexical analysis. **Lexical analysis** means dividing an input stream into meaningful units [18]. The scanner converts the input streams into tokens and passes them to the parser module.

Lex is a tool for building lexical analysers or lexers. *Lex* takes a set of descriptions of possible tokens which is called lex specifications and produces a C routine called

```
typedef struct TMethod {
     TLabel          label;
     TTOPair         typeOrig;
     TNTPairDict     *paraList;
     TNTPairDict     *usesList;
     struct TMethod *next;
   } TMethod;

typedef struct TClass {
     TLabel          label;
     TInherits       *inherits;
     TStringDict     *usesList;
     TVar            *varList;
     TMethod         *methodList;
     struct TClass   *next;
   } TClass;

typedef struct TSystem {
     TLabel          label;
     TClass          *classList;
     TUsecase        *usecaseList;
     struct TSystem *next;
   } TSystem;
```

Figure 10: System, class and method C structures from AST

lexer. A lexer takes an arbitrary input and tokenizes it. The scanner in this tool use a hand-coded lexer written in c, instead of using *Lex*.

The **parser** takes the tokens generated by the scanner and checks if the relationship among these tokens follow certain syntax rules.

The parser is generated by yacc [18]. yacc is a tool provided by UNIX. It takes a grammar specified by the user of yacc and writes a parser that recognises valid "sentences" in that grammar and invokes corresponding actions.

The parser uses the scanner by calling it repeatedly until the input stream is exhausted.

The **constructor** is the set of actions invoked when the parser find a set of tokens that matches one of the syntax rules.

This will conclude our description of the existing tool.

Chapter 5

The Extended Tool

The extended tool differs from the original tool in several ways, of which the most important is that it can test the behaviour of the design by means of use cases. The tool tests a use case by checking if the design of the system is capable of providing the sequence of transactions specified by the use case. This chapter includes a description of the testing process and the design. The implementation of the tool is discussed in Chapter 6.

5.1 The Testing Process

Before describing the testing process, we outline a set of criteria for the tool.

- The tool should support the OOSE method—the analysis and design of the system should be done using the OOSE method.
- According to the OOSE method, discussed in Section 2.2, each use case in the analysis phase has a corresponding interaction diagram in the construction phase. Therefore, the tool should be able to accept, process and store use cases and interaction diagrams.

- The textual form of the design, explained in Section 4.1, remains unchanged and it is used in the tool. The only modification is making the comment after the method declaration mandatory. The significance of this condition will be explained later in this chapter.

The steps of the test process are:

- When a use case is tested by the tool, the tool traverses the corresponding interaction diagram. An interaction diagram describes a use case as a sequence of operations calls. Upon reading each operation call in the interaction diagram, the tool extracts its corresponding comment.

- When the interaction diagram is exhausted, the tool will have a sequence of comments in a format similar to the use case. This sequence is called the *generated use case.*

- The tool displays the *generated use case* next to the original use case being tested.

- The user reads both use cases and compare their contents to see if they provide the same sequence of transactions. Note that the comparison cannot be done automatically because both the use cases and the method comments consist of informal text.

5.2 Design of The Tool

The tool accepts three types of documents as its **input**. These documents are parsed by the **parser** which obtains the information in the documents and stores it in the data structure (AST). The **user interface** of the tool allows the user to select a design and test the feasibility of its use cases. The tool uses the data stored in the

AST to obtain the use cases and the interaction diagrams of the selected design and to perform the **testing algorithms**.

Accordingly, the tool design has five main parts: The **system design forms and documents** which represent the input of the tool, the **user interface**, the **data structure**, the **parsers**, and the **major functional module**. In this section the design of these parts will be discussed while important details about their implementations are discussed in section 5.3. But first a description of an example system that will serve as a running example throughout the rest of the thesis is provided.

5.2.1 Example System

The example considered in this paper is a recycling machine system taken from [15]. This specific application is chosen because it can have many use cases, which is essential to demonstrate and test the new developed features of the tool.

The system is developed using OOSE/se case driven approach. We do not provide a detailed description of the system development process here, however, because Jacobson has described it in [15]. We discuss the various models of the system will be briefly, and we present a block design for two classes of the system.

Description of The System

The machine allows the users to return three kinds of items: bottles, cans and crates. The machine has to check the type of each returned item and it can be accessed simultaneously by many users.

After depositing an item, the user presses the return button to obtain a receipt which contains what the user has deposited, the total value of each returned item, and the total sum paid to him.

The other user of the machine is the operator who can perform the following tasks:

1. Obtain a report about the total amount of items returned;
2. Change the settings of the machine.

When something goes wrong the operator is notified by a special alarm.

Requirement Model

Building the requirement model consists of three activities: building the use case model, defining the interface of the system, and identifying the problem domain objects.

Building the use case model includes specifying the actors and use cases of the system using the description of the system.

Actors:

There are two actors in this application, *Customer* and *Operator*.

Use cases:

The functionality of the system is identified by specifying the use cases performed by the actors. In this system there are four possible use cases :

1. Returning items;
2. Item stuck (extends the first use case);
3. Generate daily report;
4. Change item.

The first two use cases are performed by the *Customer* while the other two are performed by the *Operator*.

The following is the *Returning Items* use case:

"*When the customer deposits an item, it is measured by the system. The measurements are used to determine what kind of can, bottle or crate has been deposited. If accepted the customer total is incremented, as is the daily total for that specific item type. If the item is not accepted, 'NOT VALID' is highlighted on the panel. When the customer presses the receipt button, the printer prints the date, The customer total is calculated and the following information printed on the receipt for each item type:*

name
number returned
deposit value
total for this type
Finally, the sum that the customer should receive is printed on the receipt. "

The interface of the system: The Customer interfaces are: the Customer panel (including buttons, holes, and alarm devices) and the receipt layout .

The Problem domain objects represent real entities in the application environment. they facilitate the description of the use cases since they represent a common terminology understood by the users and by the system. Some of the problem domain objects are: *Returnable item*, *receipt*, *bottle*, and *can*.

Analysis Model

After defining the functionality and interfaces of the system, the next step in OOSE method is to create the structure of the system and to distribute the different functionalities specified by the uses cases among the analysis objects.

There are three types of analysis objects: interface objects, entity objects, and control objects, each of which represents a different aspect of the system.

The **interface objects** of the recycling machine are:

- *Customer panel*, which manages the sensors in the deposit slots, receipt and start buttons.
- *Receipt printer*, which prints the information on a paper roll. When the paper roll is almost finished, it informs the operator through the alarm device.
- *Operator panel*, which is the operator's interface to the system where he/she can access the information inside the system and generate daily reports.
- *Alarm device*, which Controls a signal device and has a reset button for the alarm.

The **entity objects** are

- *Can, Bottle, and Crate*, which store the size of the entity. Each one has different size attributes.
- *Deposit item*, which is an abstract entity object inherited by can, bottle and crate object. It holds the deposit value of the item, name, and day total.
- *Receipt basis*, which keeps track of all the items deposited by the customer.

The control objects, which are usually identified by assigning one control object for each use case such that in each use case the functionality that is not handled by either an interface or an entity object is assigned to the control object. In the recycling machine system these objects are:

- *Deposit item receiver*, which handles the coupling of interface and entity objects in the returning items use case.
- *Report generator*, which controls the report generation in generate daily report use case.

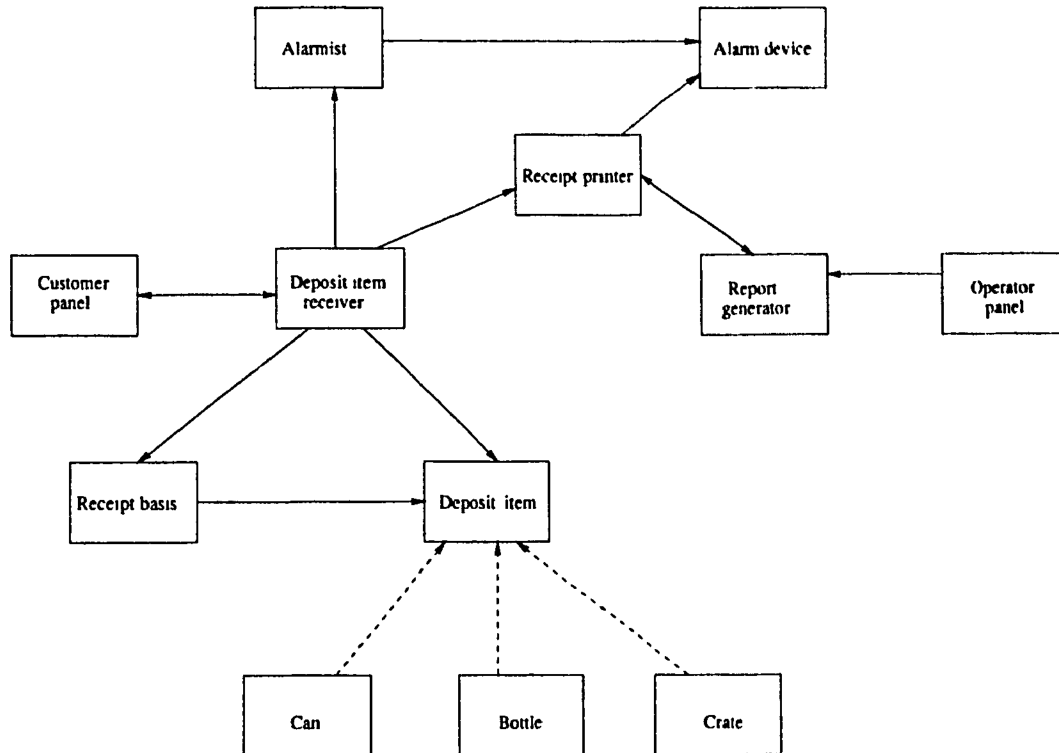

Figure 11: The design model for the recycling machine

- *Alarmist*[1], which controls the alarm device in item stuck use case.

The Design Model

Since the implementation environment for the recycling machine is made simple, defining the design objects, which are called blocks, is a direct mapping from the analysis objects. The design model is shown in Figure 11.

The next step is to design the use cases described in the previous models. This is done by showing how the different blocks communicate and how each use case is realized by these communicating blocks.

The interaction diagram of *Returning Items* use case is shown in Figure 12.

[1]This word is used in [15] to stand for an object that controls an alarm device

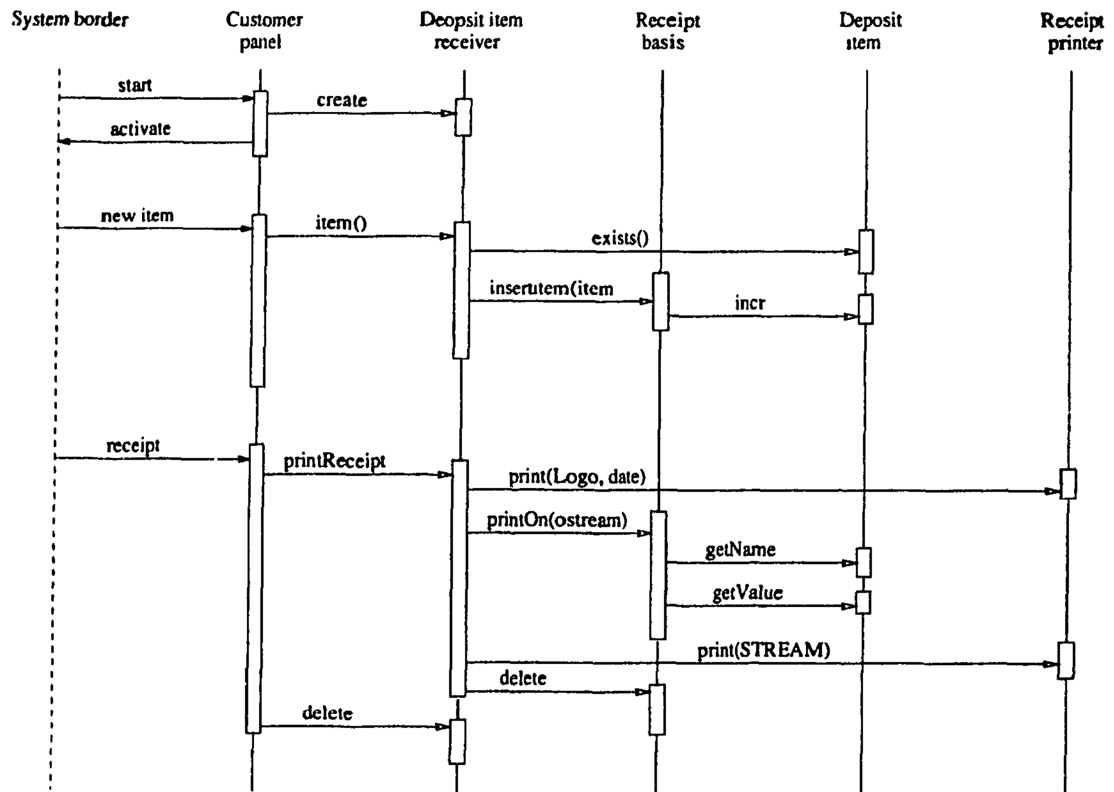

Figure 12: Interaction diagram for returning items use case

Block design

Designing a block starts by identifying its interface. This may be done by going through all the use cases that the block participates in. Therefore, after designing the use case *Returning items*, it becomes possible to do the block design for the *receipt basis* object since it participates in this use case only.

Class ReceiptBasis;

Operations;

create;

insertItem(DepositItem);

printOn(stream);

delete;

Attributes:

itemList: listOf ReturnedItem;

sum: ECU;

end Class

The block design for another object *the deposit item* can also be shown. This object participates in two other Use cases besides returning items, these are *generate daily report* and *change item*. Although these last two use cases are not shown here, the interface operations of *deposit item* object are *getTotal* and *changeValue*.

Class DepositItem;

Operations;

exists();

incr;

getName;

getValue;

getTotal;

changeValue;

Attributes;

total : ECU;

value : ECU;

name : String;

endClass

The complete design text of the recycling machine system is given in Appendix B.

5.2.2 Design Model and Representation

The current tool supports the OOSE as a system development method. Accordingly, the design model used by the tool consis‘s of three separate documents:

1. The *system design document*, which consists of a list of classes. This design document is used in the existing tool as described in [5, pages 22 and 23] and is represented by a file called the *system file*.
2. *Use cases design document*, which consists of a list of use cases. Each use case includes a text which describes a related sequence of transactions. This document is represented by a separate file called the *use case file*.
3. *Interaction diagrams design document*, which includes a list of interaction diagrams. Each interaction diagram should correspond to one use case. The interaction diagrams describe how use cases are realized by means of operation calls between the system classes. This document is represented by a separate file called the *interaction diagram file*.

Use cases and **interaction diagrams** are represented by a textual form and follow some syntax rules.

The *use case files* have the following syntax:

- The file can have an unlimited number of use cases.
- Each use case has a name and a body.
- The body of a use case consists of a numbered sequence of sentences, which are syntax-free text.

Figure 13 shows the yacc grammar of the use case files.

Based on this syntax, the use case *"Returning Items"* described in the Example system, will be written as shown in Figure 14.

The *interaction diagram files* have the following syntax:

- Each interaction diagram has a name, and a body.
- The body of an interaction diagram consists of a numbered sequence of expressions with fixed syntax. There are three types of expressions:
  1. One class can call another class using a method name.
  2. One class can send a signal to the interface of the system using a signal name.
  3. The interface of the system can send a signal to one class—usually a user-event— using a signal name

Figure 15 shows the yacc grammar of the *interaction diagram files*.

Based on this syntax, the interaction diagram *"Returning Items"* shown in Figure 12 is translated into the textual form shown in Figure 16.

```
usecase_list :
                usecase | usecase_list usecase ;

 usecase :
          USECASE IDENTIFIER {
               settingUsecaseName($2);
                             }
          opt_comment {
                addComm(aUSECASE);
                      }
          bodytext_list
          END IDENTIFIER {
               addUsecaseToSys();
                          }
         ;

bodytext_list :
          BODYTEXT {
               settingUsecaseBody($1);
                  }
          | bodytext_list BODYTEXT {
               concatUsecaseBody($2);
                                   }
         ;


opt_comment
      :  /* nothing */
      |  comment_list
      ;

comment_list
      :  UCOMMENT {  /* one-line comment */
            settingComm($1);
          }
      |  comment_list UCOMMENT { /* multiple line comment */
            concatComm($2);
          }
      ;
```

Figure 13: The yacc grammar of the *use case files*

```
usecase ReturningItems

1- After the customer returns the deposit item, it is measured by the
>  system to determine what kind of can, bottle or crate has been deposited.

2- If the item is accepted , the customer total of the item type is
>  incremented.

3- The daily total for that item is incremented.

4- If the item is not accepted, 'Not valid' is highligted on the panel.

5- When the customer presses the receipt button, the printer prints the
>  date, the customer total is calulated, and the following info are printed
>  on the receipt for each item type : name, number returned, deposit value,
>  and total for this type. Finally the amount that the customer should receive
>  is also printed.

end ReturningItems
```

Figure 14: The use case "Returning Items"

5.2.3 User-Interface

The user-interface is built to be compatible with the interface of the existing tool. The major windows of the tool are:

ToolMainWindow

The ToolMainWindow is a modified version of the ToolMainWindow in the original tool discussed in Section 4.1.1. It is the first window displayed upon running the tool. This window has five main options: **Design, View, Check, Output** and **Use Case Testing**. The first four options are discussed earlier in Section 4.1.1. The **Use Case Testing** is the new option provided by the tool. The ToolMainWindow is shown in Figure 17.

UseCaseTesting Window

When the user chooses the use case testing option from the ToolMainWindow the UseCaseTesting window pops up. This window contains the list of *system files* from which the user can choose to perform the **use case testing**. This window is shown

```
interactiondiagram  :
                    INTDIAGRAM IDENTIFIER {
                           settingIdName($2);
                                           }
                    opt_comment{
                           addComm(aID);
                                  }

                    steplist
                    END IDENTIFIER {
                        addIdToUsecase();
                                      }
                    ;

steplist :
                    step | steplist step
                    ;

step :
                    SEQUENCE CLASS IDENTIFIER
                    CALLS CLASS IDENTIFIER
                    METHOD IDENTIFIER {
                         addStep($1, $3, $6, $8);
                                          }
                    |

                    SEQUENCE SYSTEM CALLS CLASS IDENTIFIER
                    SIGNAL IDENTIFIER {
                         addSignal($1, $5, $7);
                                          }
                    |

                    SEQUENCE CLASS IDENTIFIER CALLS SYSTEM
                     SIGNAL IDENTIFIER { }

                    ;

opt_comment :
                    /* nothing */
                    |  comment_list
                    ;
comment_list :
                    IDGCOMMENT {  /* one-line comment */
                         settingComm($1);
                                }
                    |
                    comment_list IDGCOMMENT { /* multiple line comment */
                         concatComm($2);
                                             }
                    ;
```

Figure 15: The yacc grammar of the *interaction files*

```
interactiondiagram ReturningItems

1- system calls class CustomerPanel signal start
2- class CustomerPanel calls class DepositItemReceiver method create
3- class CustomerPanel calls system signal activate
4- system calls class CustomerPanel signal newItem
5- class CustomerPanel calls class DepositItemReceiver method item
6- class DepositItemReceiver calls class DepositItem method exists
7- class DepositItemReceiver calls class ReceiptBasis method insertItem
8- class ReceiptBasis calls class DepositItem method incr
9- system calls class CustomerPanel signal receipt
10- class CustomerPanel calls class DepositItemReceiver method printReceipt
11- class DepositItemReceiver calls class ReceiptPrinter method print
12- class DepositItemReceiver calls class ReceiptBasis method printOn
13- class ReceiptBasis calls class DepositItem method getName
14- class ReceiptBasis calls class DepositItem method getValue
15- class DepositItemReceiver calls class ReceiptPrinter method printStr
16- class DepositItemReceiver calls class ReceiptBasis method delete
17- class CustomerPanel calls class DepositItemReceiver method delete

end ReturningItems
```

Figure 16: Textual form of interaction diagram "Returning Items"

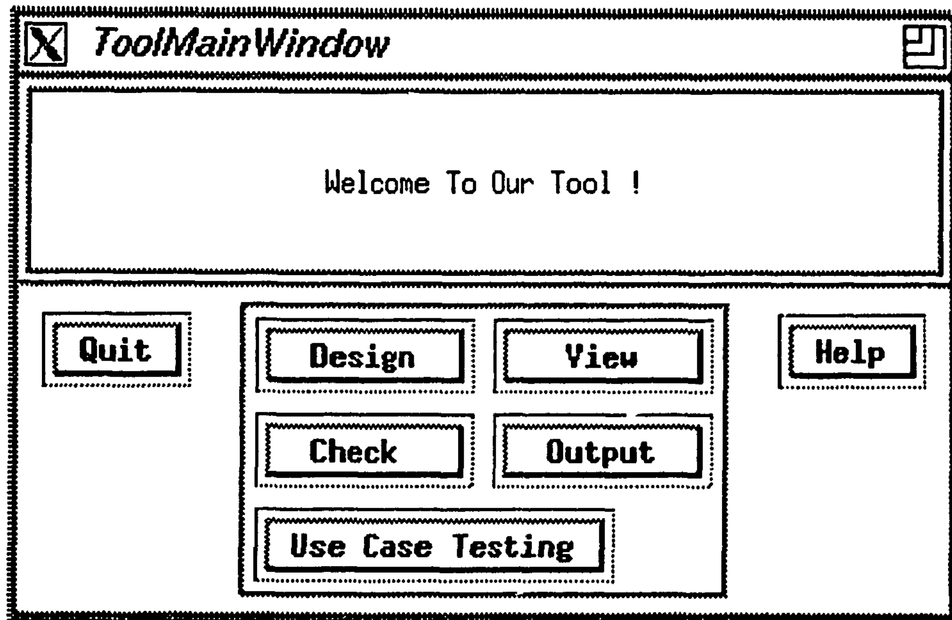

Figure 17: ToolMainWindow

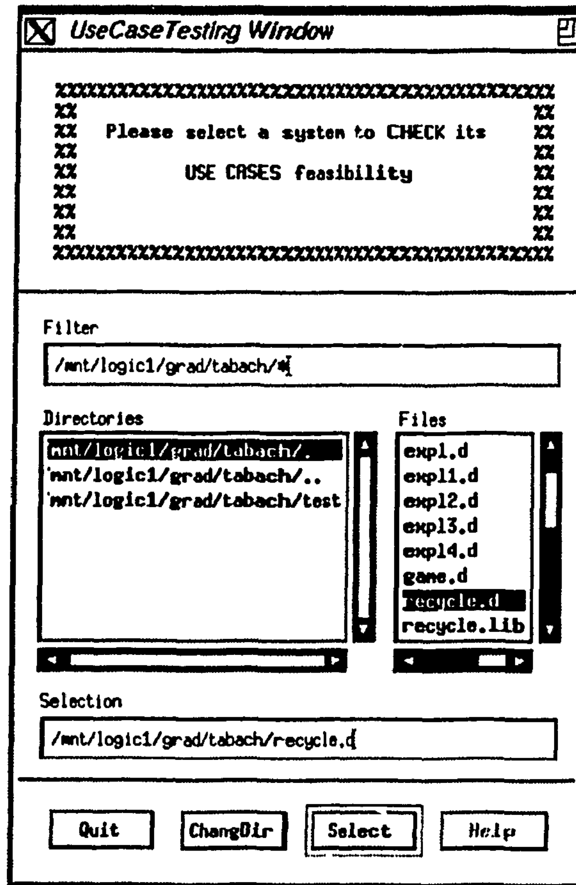

Figure 18: UseCaseTesting Window

in Figure 18.

useDialog_popup Window

This window pops up after selecting a file from the UseCaseTesting window. This window has the list of use cases of the system. The user can select one use case to be tested. This window is shown in Figure 19.

UseCaseWindow—Feasibility Checking Output

After selecting one use case from the *useDialog_popup* window, the use case is tested by the tool, if the test is successful, the *UseCaseWindow--Feasibility Checking Output* will appear.

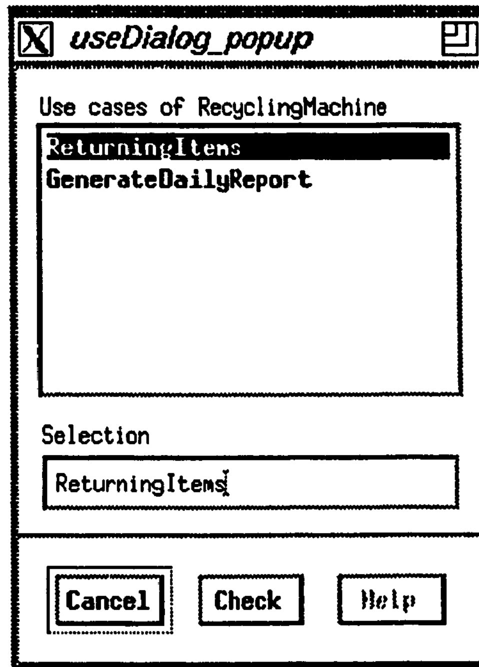

Figure 19: The useDialog_popup Window

1. The first part contains the original text of the use case being tested.
2. The second part contains the new generated use case which is a sequence of comments separated by blank lines.

Both texts can be scrolled and the entire output can be printed on an output file. This window is shown in Figure 20 showing the test output of the *Returning Items* use case.

5.2.4 Data structure

The abstract syntax tree (AST) used in the original tool is extended to hold the data of the *use cases* and *interaction diagrams* available in the *use case file* and the *interaction diagram files*.

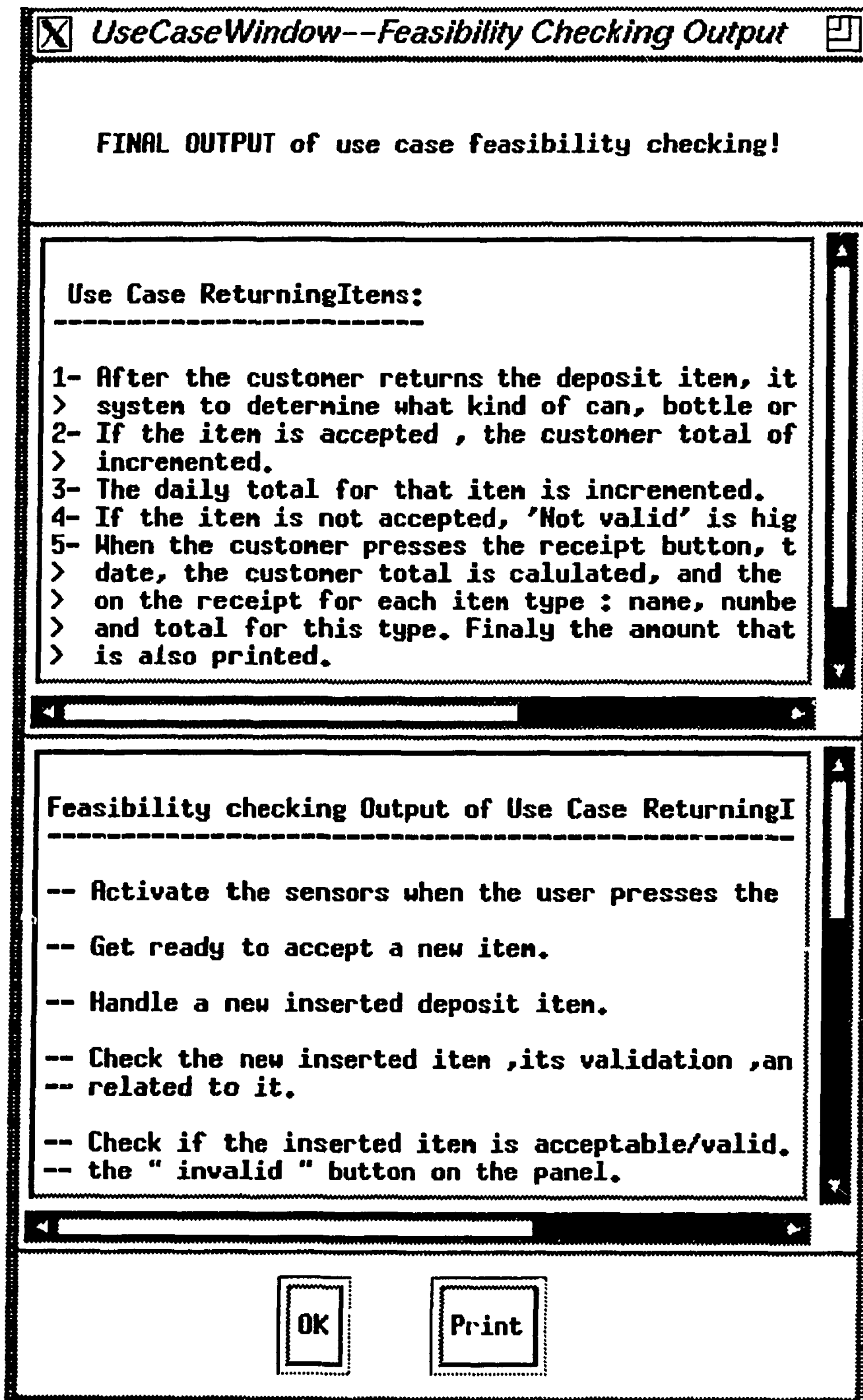

Figure 20: Final output of *ReturningItems* use case testing

5.2.5 Parsers

There are two new parsers modules developed in the extended tool. One for the *use case file* and the other is for the *interaction diagram file*. Each parser module has a **scanner** for lexical analysis, a **parser**, and a **constructor** which is a set of actions to construct the AST. The tool uses the parser developed in the original tool to parse the *system file*. The three parsers obtain data from three different files and store it into one AST.

5.2.6 Major Functional Module

There is one main functional module in the tool, the **Use Case Tester**. This module is responsible for the important function of the tool—testing use cases.

When the user selects a system file, which has an extension '**.d**', The module searches for its corresponding *use case file*, and *interaction diagram file*, which both have the same name as the system file but different extensions. The extensions of *use case files* and *interaction diagram files* are '**.use**' and '**.id**', respectively.

The files are then parsed and the AST is constructed. Next, the list of all the use cases of the current system are displayed and the user can select one of them. When a use case is selected, the module searches for its corresponding interaction diagram from the AST, and starts to execute the testing process explained in Section 5.1.

The followings are the validation checks performed in this module:

- Each file with the extension '**.d**' selected for use case testing should have a corresponding '**.use**' and '**.id**' files.
- The three file, the '**.d**', '**.use**', and '**.id**', should be parsed successfully.
- The *Consistency check* provided by the original tool should be passed before

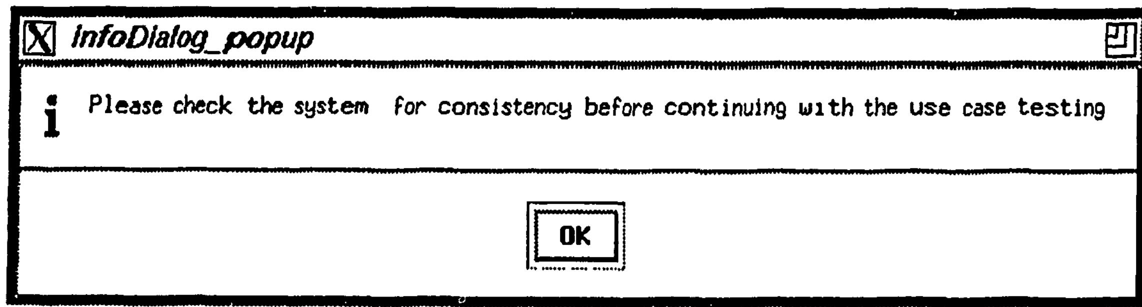

Figure 21: Error message displayed when consistency check fails

continuing with use case testing. This message is shown in Figure 21.

- Use cases in the *use case file* should not be multiply defined.
- Each use case in the *use case file* should have a corresponding interaction diagram in the *interaction diagram file.*
- The syntax of one of interaction diagram expressions looks as follows:
  P.S. The keywords are in **bold** and the names are in *italic.*

  class C1 **calls class** C2 **method** M1

  the following checks should be done:

  - C1 should be defined in the system file
  - C2 should be defined in the system file
  - M1 should be defined in the system file and included in C2. If M1 is not defined, the error message displayed includes the name of the class it should be included in. In this case it is class C2.
  - C2 should be defined in the uses list of C1, in the *system file.*

Chapter 6

Implementation of The Tool

6.1 The Implementation Language

C is the implementation language of the tool. It was chosen because the original tool, which is the starting point of our tool, is implemented in C. Moreover, the interfaces to Motif, which is used to build the user interface of the tool, are specified as C prototypes. The tool runs under Unix and X windows.

6.2 Abstract Syntax Tree (AST)

The system is represented by a set of classes, and can have a set of use cases. Each use case can point to one interaction diagram, and each interaction diagram can point to a number of classes connected in a way to show the operations calls between them. The C structure representing the *system* is modified and four new C structures are added to the AST in order to represent this information.

- The system is represented by a defined structure which is modified from the original structure to hold a list of use cases. The structure has the following

fields: system name and its description, a pointer to a list of classes, a pointer to a list of use cases, and a pointer to the next system.

- Each use case is represented by a defined structure with the following fields: the use case name and its description, the text of the use case, and a pointer to its interaction diagram.
- Each interaction diagram is represented by a defined structure with the following fields: the interaction diagram and its description, and a pointer to a C structure called *IdNode*.
- *IdNode* structure represents one class in the interaction diagram which is calling other classes and it has the following fields: its name, a pointer to a C structure called *RefNode*, and a pointer to the next class initiating an operation call.
- The C structure RefNode represents a list of classes which receive an operation call from The *IdNode* class and have the following fields: The class name, the method name used in the call, and its sequence in the interaction diagram.

Figure 22 shows part of the AST containing the C structures listed above

6.3 Parsers

There are three parser modules, one for the *system files* '.d', one for the *use case files* '.use', and one for the *interaction diagram files* '.id'. Each module contains three programs, the lexer, the parser and the constructor.

The **lexer** is a hand-written program coded in C. The **parser** uses **yacc**, and the **constructor** contains all the functions called by the parsers to insert data into the AST.

The parser module for the system design files '.d' is the same one used in the original tool except for one modification in in its yacc grammar rules—the comment after the

```c
typedef struct RefNode {
      TString          classname;
      TString          methodname;
      TString              count;
      struct RefNode *next;
    } RefNode;

typedef struct IdNode {
      TString         clname;
      RefNode         *reflist;
      struct IdNode *next;
    } IdNode;

typedef struct TId {
      TLabel    idlabel;
      IdNode     *idiagram;
    } TId;

   /* New structure added to represent the different */
   /* use cases of the system                         */

typedef struct TUsecase {
     TLabel            label;
     TString           usetext;
     TId               *id;
     struct TUsecase *next;
    } TUsecase;


typedef struct TSystem {
     TLabel           label;
     TClass           *classList;
     TUsecase         *usecaseList;
     struct TSystem *next;
    } TSystem;
```

Figure 22: C structures of the system, use case and interaction diagram

```
method
        :   METHOD IDENTIFIER {
               settingMethodName($2);
             }
            opt_para_list
            opt_type
            opt_comment {
               addComm(aMETHOD);
            }
            opt_uses {
               addMethodToClass();
             }
        ;
```

Figure 23: The original grammar rule for **method**

```
method
        :   METHOD IDENTIFIER {
               settingMethodName($2);
             }
            opt_para_list
            opt_type
            comment_list {
               addComm(aMETHOD);
            }
            opt_uses {
               addMethodToClass();
             }
        ;
```

Figure 24: The modified grammar rule for **method**

method is made mandatory. Figures 23 and 24 show the original and the modified grammar rule, respectively.

The comment after each *method* is made mandatory because when the program traverse an interaction diagram, it reads the *comment* of each method contained in the diagram. Later it uses all this list of *comments* to write the *generated use case* described in Section 5.1. Therefore, if a method in the interaction diagram does not have a comment, it means that the *generated use case* is not complete. Consequently, the result from the use case testing will not be correct.

```
...
method exists
-- Check if the inserted item is acceptable/valid. If not, highlight
-- the " invalid " button on the panel.

method incr
-- Increment the total of the deposit item type.

method getName
-- Get the name of the deposit item.

method getValue
-- Get the deposit value of the deposit item.

method getTotal
-- Get the daily total for a deposit item type.

method changeValue
-- Change the deposit value of an item.

...
```

Figure 25: An example of the recommended style of comments

Even though the comment is syntax-free, it is recommended to use an imperative mode which makes the comment looks like a request from the system. The better the user writes his comments, the better the output will be. Figure 25 shows an example of the recommended style of comments. This example is taken from the *system file* of the recycling machine system.

6.3.1 Multiple Parsers

One problem encountered during the implementation of the tool is the use of **multiple parsers** in the same application.

The parser generated by **yacc** calls the lexer *yylex()*—which is the main function in the lexer—whenever it needs a token from the input.

yacc takes the grammar provided by the user and creates a file called **y.tab.c**, which is the C language parser. **yacc** also creates the file **y.tab.h** by defining the token

names in the parser as C preprocessors names in **y.tab.h** [18].

In our case, we needed to use three separate parsers. But **yacc** does not make this easy because every parser it generates has the same entry point *yyparse()* and calls the same lexer *yylex()* which uses the same token value *yylval.* In addition, most versions of **yacc**, including the one we are using (sun release 4.1), put the parse table and the parser stack in global variables with names like *yyact* and *yyv.* If we translate the three grammars, compile, and link the resulting files(renaming at least two of them into something other than **y.tab.c**) we still got a long list of multiply defined symbols.

One solution to this problem is to change the names that **yacc** uses for its functions and variables. An easy way to do this is to use the command-line switch **-p** to change the prefix used on the names in the parser generated by **yacc**. For example the command line:

```
yacc -p nn grammar.y
```

produces a parser using grammar.y with the entry point *nnparse()* instead of *yyparse()*, which calls the lexer *nnlex()* instead of *yylex()* and uses the value *nnlval*, and variable *nnchar*, and so on.

Another command-line switch **-b** is used to change the prefix of the files generated by **yacc**. For example:

```
yacc -p nn -b pref grammar.y
```

This produces **pref.tab.c** and **pref.tab.h** files instead of **y.tab.c** and **y.tab.h**.

Unfortunately, this solution did not work with the currently available version of **yacc** because it is relatively old and does not provide the '**-p**' and '**-b**' switches. Therefore, **yacc** was replaced by **bison** which worked properly.

bison is a **yacc**-compatible parser generator. The **yacc** grammar works with **bison** without any modification. Like **yacc**, **bison** converts a grammar into a C parser which recognises valid "sentences" based on that grammar [6].

The following commands were used in the *make file* to change the default prefix of all the symbols generated by **bison**:

bison -d -p u e uparse.n

where 'uparse.n' is the file containing the grammar for the '.use' files. The same command was used for the parser of the interaction diagram files 'iparse.n'.

The main function in the lexer of the *use case files* is named uselex(). The same thing was done for the '.id' files.

The default prefix of the files generated by **bison** are not 'y.tab.c' and 'y.tab.h'. Instead, they are 'filename.tab.c' and 'filename.tab.h' where 'filename' is the name of the file containing the **yacc/bison** grammar. Since there are three different files containing the three different grammars, the prefix of the generated files didn't need special handling. For example the C parser file generated by the above command line is uparse.n.tab.c.

6.4 User Interface

The user interface is built using Motif. Motif is briefly described in Section 4.1.2. Motif facilitates greatly the work of the programmer in building the user interface by providing preconstructed user interface objects- widgets. The user can use any widgets in his application program to get the user interface objects needed.

This section includes a brief description of the major Motif widgets used in the tool implementation. However before proceeding, two important terms related to the

Motif widgets should be pointed out:

Resources: Every widget has a set of associated resources. Resources are similar to variables but they are accessed in a more complicated way. The widget's resources control its appearance and behaviour.

Callbacks are a way to handle user-events on the widgets. In other words, when a certain widget allows the user to manipulate it on the screen, the widget needs a way to inform the program about the user's input. This is done by using callback functions.

A callback function is a C function that performs certain action. The function is registered as a callback function for a specific widget by using **'Xtaddcallback'** . When the widget receives a user event, it calls its registered callback function to perform the action requested by the user.

Top level shell

This is the first widget created in a Motif program. It is the main application window that provides the standard window appearance: the title bar, the maximise and minimise buttons, the resize area, and so on.

Form widgets

The Form widget is a manager widget. A manager widget handles the placement of many different widgets in a single window. A form widget holds many widgets and provides them with **automatic resizing and reposition.**

RowColumn Widgets

The RowColumn widget is a manager widget similar to the form widget. The difference between the two widgets is that in the form widget the user has to specify the placement of the child widgets by attaching them to the Form widget, while in the RowColumn widget the child widgets are arranged automatically which is especially useful when the number of child widgets is large.

Paned Window Widgets

The paned window holds other widgets and passes to them some of its resources. This window allows the user to resize different panes of a window using a draggable control called a sash. The programmers usually puts the manager widgets, such as **Form** and **RowColumn**, inside the paned window then, puts multiple widgets in these manager widgets as described above.

Frame Widgets

The Frame widget is a very simple widget. It puts a frame around its child widget if it lacks one, such as the **label widget** and **toggle button**.

Label Widgets

The label widget can display strings of characters. It is usually used to display help information, labels, greeting messages, and so on.

Push Button Widgets

The Push button widget allows the user to issue a command by clicking on it. The Push button has a label and it inherits all the resources of the **Label widget**.

File Selection Box

The file selection box allows the users to select a file form a list of files in the current directory. It is a very powerful widget because it provides a number of resources that contains important information, such as the current directory and its list of files, the filter string, the types of files to be displayed, and many others.

Text/Scrolled Text Widgets

The text widget is very powerful and more complicated than the other widgets. It allows the users to enter characters and display them on the screen. The user can delete characters using the backspace.

The scrolled text widget is a variation from it. It adds a scroll bar to the window and allows the user to scroll the text in all directions.

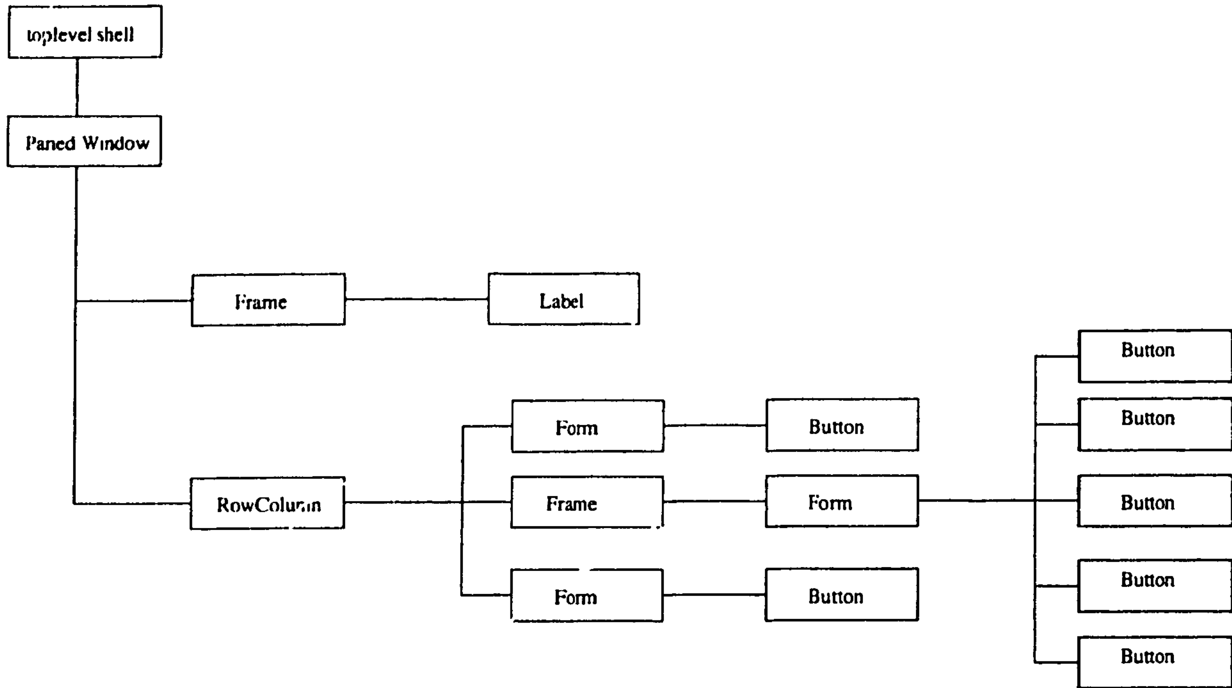

Figure 26: The Motif widget structure of the ToolMainWindow

Figure 26, 27, and 28 show the Motif widgets structure of the **major windows of the tool's user interface.**

6.4.1 The Major functional Module — Use Case Tester

This module consists of many important functions that perform the use case testing.

useCaseCb Function

useCseCb is a callback function invoked when the users select the **use case testing** option from the *ToolMainWindow*. Since this function must consider many possibilities, we describe its actions by means of pseudocode:

Create the *UseCaseTesting* window (shown in Figure 18) where only the files with extension '**.d**' are displayed in the files list;
Obtain the 'file name' and validate it;

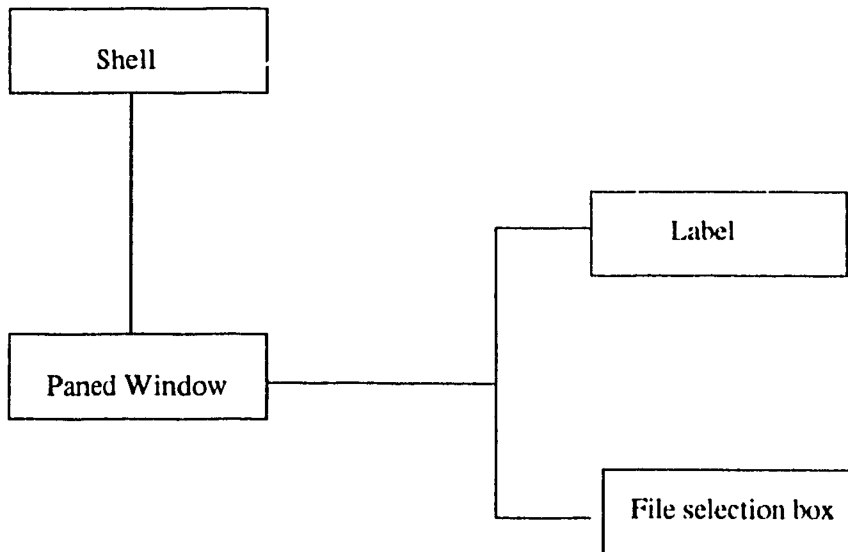

Figure 27: The Motif widget structure of the UseCaseTesting Window

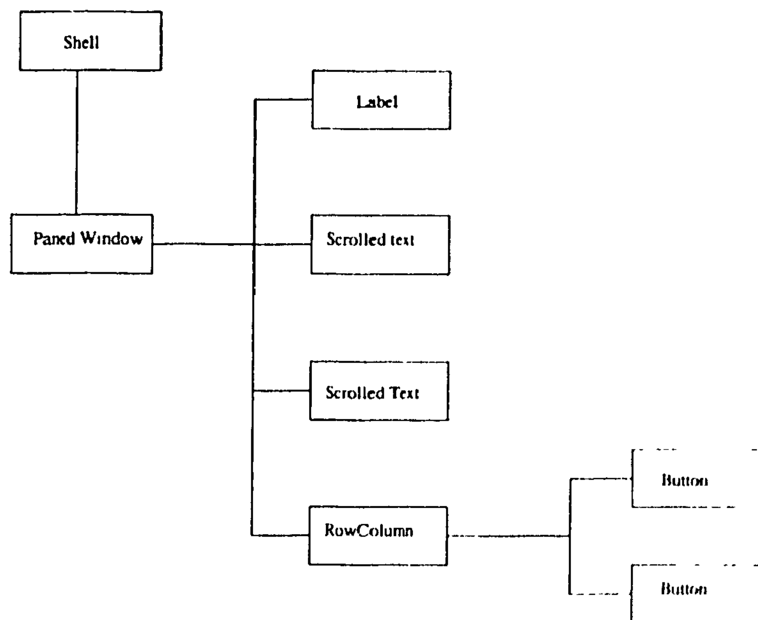

Figure 28: The Motif widgets structure of the UseCaseWindow Feasibility Checking Output

```
If 'Not Valid' Then
  Return;
Else
  Call the system parser to parse 'filename.d';
  If parse fail Then
    Display an error message and return;
  Else
    Call use case parser to parse 'filename.use';
    If parse fail Then
      Display an error message and return;
    Else
      Call the interaction diagram parser to parse 'filename.id';
      If parse fail Then
        Display an error message and return;
      Else
        Check if there exists a library file 'filename.lib';
        If exist Then
          call the system parser to parse it;
        Call for consistency Checking (provided by the original tool);
        If Consistency check fails Then
          Display the error message shown in Figure 21;
        Else
          Display the useDialog_popup Window (shown in Figure 19) and create the use cases
          selection list by reading the use case list of the current system from the AST;
```

checkuseCb function

This is a callback function invoked when the users select a use case to test it. Since this function must consider many possibilities, we describe its actions by means of pseudocode:

```
Search for use case in the AST;
If (not found) Then
   Display an error message and return;
If use case does not have an interaction diagram Then
   Display an error message and return;
Else
   Start Traversing the interaction diagram;
   While (not the end of interaction diagram) Do
           If class1 is undefined Then
              Display an error message and return;
           If class2 is undefined Then
              Display an error message and return;
           If method is undefined in class2 Then
              Display an error message containing the name of class2 and return;
           If class2 is not defined in the uses list of class1 Then
              Display an error message: " class1 cannot call class2 ";
           Else
              Read the comment of the method and store it;
   End While
   Display all the comments stored while traversing the interaction diagram on the output screen
   Display the text of the use case on the output screen;
```

In addition to these major functions, there are other important key functions that are not described here, such as the functions which obtain the data from the *use case files* and the *interaction diagram files* and construct the AST.

Chapter 7

Assessment of The Tool

This chapter includes an assessment of the extended tool, developed during this project—Why the tool is useful, what kind of help it provides for the designers, and what are the limitations.

7.1 Advantages of The Tool

The extended tool has many good properties and useful features:

- Jacobson introduced use cases in the OOSE method in [15] as an informal text representing a behaviourally related sequence of transactions which drive the system development and help the designer in testing the design. The extended tool incorporates this **informal concept** of use cases into a design tool and provides **semi-automatic use case testing** for the design and the resulting behaviour of the system. We do not know of any other tools, either by Jacobson or others, that provide automatic use case testing.
- The tool can automatically check if a use case is feasible by generating the possible sequence of events that will happen upon executing the use case using

the current design. The tool however, cannot automatically determine if the output sequence of events is correct. The designer has to compare manually the text of the original use case and the output sequence of events (which is called the new generated use case) and decide if they provide the same behaviour.

- In case there is a gap between the design and the requirements, the extended tool will likely detect the error and provide a useful feedback to help the designer in identifying the missing classes, methods, and operation calls.

- The tool provides the designer with an early feedback—at the design stage – instead of waiting until the implementation stage to know if the system meets the users requirements.

- The tool enables the maintainers to use the tool when there are modifications in the requirements, so they can change the design, then change the code accordingly. This helps in keeping the requirements, design, and code consistent during the system evolution.

- The tool is reliable and robust due to the numerous validation checks.

- The extended tool provides the feature of testing the dynamic behaviour of the system in addition to the feature of testing the static model of the design, provided by the original tool. As a result, the extended is more than just a design tool which provides facilities for creating a design and generating a printable design documents. It helps the designer to generate a higher quality design and improves the speed of the system development.

- Although the tool provides many important features, its user interface is very simple and clear which makes its use intuitive and pleasant.

7.2 Limitations of The Tool

The tool has some limitations and drawbacks:

- The tool supports only one software development method, the OOSE method. Therefore the designer cannot develop a system using OMT for example, then test its design directly using the tool. However, since all the development methods are getting closer and have many similar concepts, it will not be difficult to transfer a design developed in another development method into the OOSE method, especially, since the use case concept is gaining more popularity among the other development methods, such as OMT, Responsibility-driven approach, Booch's method, and so on.

- There is a small overhead work required initially from the designer to transform the interaction diagrams from their graphical form to the textual form specified by the tool and to write a meaningful comment for each method in the design.

- Currently, the tool runs only under Unix operating systems.

Finally, we are confident that the tool is both usable and useful. We have used it to validate both designs from Jacobson's book [15] and designs prepared by independent testers.

Chapter 8

Conclusion

In this thesis, we describe an extension to an existing design tool. The existing tool provides assistance for the designers and maintainers of a system by supporting automation in capturing and altering the design. Its important features are: editing, viewing, checking, and generating a printable report of a design.

The extended tool, developed during this project, adds to the original tool a new feature which is **use case testing**. The purpose of this feature is to test the behaviour of the system at the design stage.

The extended tool enables the designer to check the dynamic behaviour of the design by checking if a certain behaviour, extracted from the requirements and represented by the use case, is feasible using the existing design.

This test process is not completely automated by the tool. The tool will automatically generate a sequence of events based on the existing design and the input, which is the use case to be tested and its interaction diagram. However, it the responsibility of the designer to compare the generated sequence of events with the use case and decide if they provide the same behaviour.

The **use case testing** feature, helps in verifying whether the design meets the requirements. If not, it helps in identifying the necessary changes to meet the new requirements.

This feature, makes the design more reliable and more flexible which is one of the main objective of developing the first version of the tool. " The design must be "light weight" - easy to change... " [10]. In addition, this enables the developers of the system to save a lot of effort and time by getting an early feedback at the design stage and before writing the code.

A detailed description of the extended tool, including its functionality. the testing process, the design formats, and the important design and implementation details are given in this thesis. One of the implementation issues discussed is the problem of using multiple parsers in the same application. This problem is described in detail and a solution is presented.

A review of the original tool, including its functionality and design is also provided.

An overview of the existing and most popular software development methods is given, including a discussion about their differences, commonalities, advantages and disadvantages.

Currently, the extended tool runs under Unix and X Windows environment, and has the following capabilities:

- Edit a design
- View a design
- Check the consistency of a design
- Generate a printable report of a design
- Perform use case testing on a design

Many possible features can be incorporated into the design tool. One important suggestion for future work is:

- Incorporate the feature of **behaviour allocation in use cases** into the design tool. This idea is proposed by D. Bennet [19] and described in Section 3.1. For each use case, the designer collects the interface of each object participating in the use case (the incoming and outgoing messages), studies all the interfaces and evaluates their cohesiveness and consistency. For example, if an object sends most of its messages to one receiver, the designer can join the two objects. The process of collecting all the objects interfaces can be automated by the tool as well as checking their cohesiveness and consistency.

In the conclusion of her thesis, Ding [5] made the following suggestions, which we endorse:

- "Some new functions can be added to the tool to enlarge the tool's functionality. This may include the ability to provide graphical display (diagram) for the system design and the classes of the system; the ability to respond to the user's queries. Some typical queries are: which classes uses service C::S? Which services does class C use?, etc.." Using the existing AST, implementing those queries is straightforward.
- " The tool could be extended with generators for various target languages. For example, it could generate class specifications in C++ or Eiffel, leaving only the bodies of the methods to be completed by the implementors."
- " An experiment could be conducted to compare this tool with other tools."

In addition to the above suggestions, Ding suggested a future version of the tool that can take into account the dynamic model of the system, which is partly what we have achieved in this thesis.

Finally, the project significance is that it benefits from the importance of *use case* concept introduced by Jacobson [15]. This concept has many advantages in capturing the requirements, driving the system development, providing traceability, assisting integration testing, and many other advantages which are listed in Section 3.1. The project resulted in incorporating this concept into a design tool which provides an automatic testing of the behaviour of a design by means of *use cases*.

Most existing tool provide only consistency checking of the **static model** of the design. What is new about our tool is that, in addition, it tests the **dynamic behaviour** of the design.

Bibliography

[1] Grady Booch. *Object Oriented Design With Applications.* Benjamin/Cummings, 1991.

[2] Marshall Brain. *Motif Programmming: The esssentials....and More.* Digital Press, 1991.

[3] R.J.A. Buhr and R. S. Casselman. Designing with timethreads. Technical report, Carleton University, September 1993.

[4] Terry Church and Philip Mathewa. An evaluation of object and case tools:the Newbridge experience. In Hausi A. Muller and Ronald J. Norman, editors, *Proceedings of the Seventh International Conference on Computer-Aided Software Engineering*, pages 4–9, July 1995.

[5] Hanwei Ding. A design tool for object oriented development. Master's thesis, Department of Computer Science, Concordia University, November 1994.

[6] Charles Donnely and Richard Stallman. *Bison.* Free Software Foundation, December 1992.

[7] Gene Forte. Tools fair: out of the lab, onto the shelf. *IEEE Software*, pages 70–79, May 1992.

[8] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[9] Ian Graham. *Object Oriented Methods.* Addison-Wesley, 1994.

[10] Peter Grogono. Designing for change. In *CASCON'94*, November 1994. Invited workshop presentation.

[11] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[12] M. Hedlund. A representation of object behavior in LOTOS. In *OOPSLA '92*, pages 125–126, October 1992.

[13] Ivar Jacobson. Formalizing use-case modeling. *Journal of Object-Oriented Programming*, pages 10–14, March 1995.

[14] Ivar Jacobson and Magnus Christerson. A growing consensus on use cases. *Journal of Object-Oriented Programming*, March 1995.

[15] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering*. Addison Wesley, 1992.

[16] Bjorn Kirkerud. *Object-oriented programming with SIMULA*. Addison-Wesley, 1989.

[17] Wilf R. Lalonde and Jhon R. Pugh. *Inside Smalltalk*. Prentice Hall, 1990.

[18] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc, 1995.

[19] Fredrick Lindström. Experiences of use case and similar concepts. In *OOPSLA '92*, pages 123–130. Acm Press, October 1992.

[20] Libero Nigro. A real time architecture based on Shlaer-Mellor object lifecycles. *Journal of Object-Oriented Programming*, page 20, March 1995.

[21] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[22] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1988.

[23] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

Appendix A

The Abstract Data Structure

```
#ifndef TOOLSTRUCT_H
#define TOOLSTRUCT_H

#include <stdlib.h>
#include <stdio.h>
#include <strings.h>

#define NIL   NULL

#define TRUE     1
#define FALSE    0

#define NOFILENAME   0
#define CANNOTOPEN   1
#define PARSEFAIL    2
#define PARSESUCCEED 3
```

```
typedef char* TString;

    /* data structure used for holding multi-line string in display */
typedef  struct TMultiStr {
        char  *str;
        int   len;
 } TMultiStr;

typedef enum
      { aSYSTEM, aCLASS, aUSECASE, aID, aMETHOD, aVAR, aUSES, anINHERITS } TAtt

typedef struct TLabel {
     TString   name;
     TString   comment;
   } TLabel;

typedef struct TStringDict {
     TString              name;
     struct TStringDict *next;
   } TStringDict;

typedef struct TNTPairDict {
     TString        name;
     TString        type;
     struct TNTPairDict *next;
   } TNTPairDict;

typedef struct TTOPair {
     TString   type;
     TString   origin;
   } TTOPair;
```

```
typedef struct TVar {
    TLabel      label;
    TTOPair     typeOrig;
    struct TVar *next;
  } TVar;

typedef struct TMethod {
    TLabel          label;
    TTOPair         typeOrig;
    TNTPairDict     *paraList;
    TNTPairDict     *usesList;
    struct TMethod *next;
  } TMethod;

struct  TClass;
typedef struct TInherits {
    TString          name;
    struct TClass    *inhNodePtr;
    struct TInherits *next;
  } TInherits;

typedef struct TClass {
    TLabel          label;
    TInherits       *inherits;
    TStringDict     *usesList;
    TVar            *varList;
    TMethod         *methodList;
    struct TClass   *next;
  } TClass;
```

```
/* Three NEW structures added to the old version      */
/* of the tool to represent the interaction diagrams */
typedef struct RefNode {
     TString        classname;
     TString        methodname;
     TString            count;
     struct RefNode *next;
  } RefNode;

typedef struct IdNode {
     TString       clname;
     RefNode       *reflist;
     struct IdNode *next;
  } IdNode;

typedef struct TId {
     TLabel   idlabel;
     IdNode    *idiagram;
  } TId;

 /* New structure added to represent the different */
 /* use cases of the system                         */

typedef struct TUsecase {
    TLabel          label;
    TString         usetext;
    TId             *id;
    struct TUsecase *next;
  } TUsecase;
```

```
typedef struct TSystem {
     TLabel          label;
     TClass          *classList;
     TUsecase        *usecaseList;
     struct TSystem *next;
   } TSystem;

#endif
```

Appendix B

Design Texts

This appendix includes the design files for two example systems which were used to test the tool. For each example thre are three design files: the **system design**, the **use case** and the **interaction diagram** files.

The Recycling Machine Example:

The system design file 'recycle.d':

```
system RecyclingMachine
-- The system controls a recycling machine for returnable bottles, crates
-- and cans.

class ReceiptBasis

inherits StdErrors Program
uses DepositItem

method create
-- Initialize the calculation of the of the information to be printed.
```

```
method insertItem ( DepositItem : String)
-- Increment the daily total of the deposit item type

method printOn(Stream : Buffer)
-- Put together the information about all the items inserted by the customer
-- and store it in the stream.

method delete
-- Delete the customer of receipt basis to get ready for the next customer.

end ReceiptBasis

class DepositItem

var total : ECU
var value : ECU
var name : String

method exists
-- Check if the inserted item is acceptable/valid. If not, highlight
-- the " invalid " button on the panel.

method incr
-- Increment the total of the deposit item type.

method getName
-- Get the name of the deposit item.

method getValue
-- Get the deposit value of the deposit item.
```

```
method getTotal
-- Get the daily total for a deposit item type.

method changeValue
-- Change the deposit value of an item.

method delete
-- Delete the total of the deposit item type

end DepositItem

class CustomerPanel

uses DepositItemReceiver

method start
-- Activate the sensors when the user presses the start button.

method newItem
-- Handle a new inserted deposit item.

method receipt
-- The receipt button is pressed, start issuing a receipt.

end CustomerPanel

class DepositItemReceiver

uses DepositItem ReceiptBasis ReceiptPrinter
method create
-- Get ready to accept a new item.
```

```
method item
-- Check the new inserted item ,its validation ,and update the information
-- related to it.

method printReceipt
-- Print all inforamtion related to the transcations done by one customer.

method delete
-- Delete all the information about the transcation done by last customer

end DepositItemReceiver

class ReceiptPrinter

method print(logo: String, date: Date)
-- Print the logo, date on the paper roll. If paper roll is empty, set
-- the alarm device.

method printStr(stream: Buffer)
-- Print the stream on the paper roll.

end ReceiptPrinter

class ReportGenerator

uses DepositItem ReceiptPrinter
method generate
-- Prepare a printout for the daily total of all deposit items returned
-- during the day, by getting the total of each item, then adding it up
-- and put it all on the print stream.
```

```
method delete
-- Delete the daily total to prepare for a new daily report.
end ReportGenerator

class OperatorPanel

uses ReportGenerator

method printReport
-- Handle the print button on the operator panel when it is pressed.

end OperatorPanel
```

The use case file 'recycle.use':

```
usecase ReturningItems

1- After the customer returns the deposit item, it is measured by the
>  system to determine what kind of can, bottle or crate has been deposited.

2- If the item is accepted , the customer total of the item type is
>  incremented.

3- The daily total for that item is incremented.

4- If the item is not accepted, 'Not valid' is highlited on the panel.

5- When the customer presses the receipt button, the printer prints the
>  date, the customer total is calulated, and the following info are printed
>  on the receipt for each item type : name, number returned, deposit value,
```

```
> and total for this type. Finaly the amount that the customer should receive
> is also printed.

end ReturningItems

usecase GenerateDailyReport

1- The operator asks for a daily report which includes the total values of
> all the items returned during the day, in addition to the sum of all these
> total, by pressing on the print button on the operator panel.

2- The daily total of an item type updated when the customer returns an
> item of its type, is printed on the report

3- Then all the totals are added up and also printed on the report.

4- The total numbers are reset to start a new daily report, next day.

end GenerateDailyReport
```

The interaction diagram file 'recycle.id':

```
interactiondiagram ReturningItems

1- system calls class CustomerPanel signal start
2- class CustomerPanel calls class DepositItemReceiver method create
3- class CustomerPanel calls system signal activate
4- system calls class CustomerPanel signal newItem
5- class CustomerPanel calls class DepositItemReceiver method item
6- class DepositItemReceiver calls class DepositItem method exists
7- class DepositItemReceiver calls class ReceiptBasis method insertItem
```

```
8- class ReceiptBasis calls class DepositItem method incr
9- system calls class CustomerPanel signal receipt
10- class CustomerPanel calls class DepositItemReceiver method printReceipt
11- class DepositItemReceiver calls class ReceiptPrinter method print
12- class DepositItemReceiver calls class ReceiptBasis method printOn
13- class ReceiptBasis calls class DepositItem method getName
14- class ReceiptBasis calls class DepositItem method getValue
15- class DepositItemReceiver calls class ReceiptPrinter method printStr
16- class DepositItemReceiver calls class ReceiptBasis method delete
17- class CustomerPanel calls class DepositItemReceiver method delete

end ReturningItems


interactiondiagram GenerateDailyReport

1- system calls class OperatorPanel signal printReport
2- class OperatorPanel calls system signal activate
3- class OperatorPanel calls class ReportGenerator method generate
4- class ReportGenerator calls class ReceiptPrinter method print
5- class ReportGenerator calls class DepositItem method getName
6- class ReportGenerator calls class DepositItem method getTotal
7- class ReportGenerator calls class ReceiptPrinter method printStr
8- class ReportGenerator calls class DepositItem method delete
9- class OperatorPanel calls class ReportGenerator method delete

end GenerateDailyReport
```

The Game example

The system design file 'game.d':

```
system GamePlayer

class UserInterface
uses Analyser Controller
  method DoPlay
    -- Establish conditions for making a move.
  method DoAnalyse
    -- Request an analysis of the current situation.
  method Display
    -- Display the given data on the screen.
end UserInterface

class Controller
uses Game ScoreBoard
  method Play
    -- Request execution of a move.
    -- Request display of updated scores.
end Controller

class Game
uses Environment ScoreBoard
  method Move
    -- Obtain current status from environment.
    -- Execute the move.
    -- Inform scorer of scores and penalties.
    -- Update environment status.
  method Show
    -- Issue a report of the status of the game.
end Game

class ScoreBoard
```

```
uses UserInterface
  method Update
    -- Alter scores to reflect last move.
  method Show
    -- Issue report of scores for the players.
end ScoreBoard

class Environment
  method Status
    -- Compute the current status of the playing environment.
  method Update
    -- Update the environment to reflect the last move.
  method Show
    -- Issue report of the status of the environment.
end Environment

class Analyser
uses Game Environment ReportGenerator
  method Analyse
    -- Request a report of the game status.
    -- Request a report of the environment status.
    -- Evaluate the current situation.
    -- Issue a report of the current situation.
end Analyser

class ReportGenerator
uses UserInterface
  method Generate
   -- Format data and issue a comprehensive report.
end ReportGenerator
```

The use case file 'game.use':

```
usecase Analyse

1- The player initiates an analysis.
2- The controller passes the analysis request to the analyser.
3- The analyser obtains game information.
4- The analyser obtains environment information.
5- The analyser request a report.
6- The report generator sends a report to the user.

end Analyse

usecase PlayMove

1- The player initiates a move.
2- The controller passes the move request to the game simulator.
3- The game simulator requests the current status of the environment.
4- The game simulator executes the move.
5- The game simulator computes scores and penalties.
6- The controller displays the updated scores.

end PlayMove
```

the interaction diagram File 'game.id':

```
interactiondiagram Analyse
```

```
1- system calls class UserInterface signal DoAnalyse
2- class  UserInterface   calls class Analyser        method Analyse
3- class  Analyser        calls class Game            method Show
4- class  Analyser        calls class Environment     method Show
5- class  Analyser        calls class ReportGenerator method Generate
6- class  ReportGenerator calls class UserInterface   method Display
end Analyse

interactiondiagram PlayMove

1- system calls class UserInterface signal DoPlay
2- class  UserInterface calls class Controller    method Play
3- class  Controller    calls class Game          method Move
4- class  Game          calls class Environment   method Status
5- class  Game          calls class ScoreBoard    method Update
6- class  Game          calls class Environment   method Update
7- class  Controller    calls class ScoreBoard    method Show
8- class  ScoreBoard    calls class UserInterface method Display
end PlayMove
```