Edith Cowan University
Research Online

Theses: Doctorates and Masters | Theses

1-1-1996

A search tool to enhance the selection and utilisation of reusable software modules within the object-oriented paradigm

Robert H. Cross
Edith Cowan University

Follow this and additional works at: https://ro.ecu.edu.au/theses

Part of the Computer Sciences Commons

Recommended Citation

Cross, R. H. (1996). *A search tool to enhance the selection and utilisation of reusable software modules within the object-oriented paradigm*. https://ro.ecu.edu.au/theses/949

This Thesis is posted at Research Online.
https://ro.ecu.edu.au/theses/949

Edith Cowan University
Research Online

Theses: Doctorates and Masters Theses

1996

A search tool to enhance the selection and utilisation of reusable software modules within the object-oriented paradigm

Robert H. Cross
Edith Cowan University

Recommended Citation
Cross, R. H. (1996). *A search tool to enhance the selection and utilisation of reusable software modules within the object-oriented paradigm* . Retrieved from http://ro.ecu.edu.au/theses/949

This Thesis is posted at Research Online.
http://ro.ecu.edu.au/theses/949

Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.

- A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author's moral rights contained in Part IX of the Copyright Act 1968 (Cth).

- Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

A Search Tool

to Enhance the Selection and Utilisation

of Reusable Software Modules

within the Object-Oriented Paradigm

by

Robert H. Cross B. Bus.

A dissertation submitted in partial fulfilment of the requirements for the Award of

Master of Science

at the

Faculty of Science, Technology and Engineering,

Edith Cowan University

Date of Submission: 8th February 1996

Abstract

The affinity for reuse within the object-oriented paradigm may enable high levels of productivity; however, gains will become realisable only if a systems developer has access to tools which aid in the selection of classes.

A method for object-oriented analysis and design is detailed and its process is assisted by an object-oriented search tool based on reference and corporate library technology. The search tool contributes to the determination of suitable existing inheritable classes and an explanation of its construction and use is included. A practical demonstration of the method, using the search tool, is elaborated.

The thesis demonstrates that text retrieval techniques used in modern libraries may be successfully applied to determine suitable classes for the object-oriented paradigm.

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Acknowledgments

I wish to thank my supervisors, Dr Thomas O'Neill and Dr James Millar for their valuable advice. Additional acknowledgment is extended to Dr Ken Mullin and Mr William Laidman for their proofing and comments. My wife, Ellie, has provided continual encouragement and support. I owe a debt of gratitude to Dr O'Neill for his unstinting time and enduring patience throughout the past year.

Table of Contents

		Page
Abstract		ii
Declaration		iii
Acknowledgments		iv
List of Figures		vi
Chapter		
1	Introduction	1
1.1	The Aim	1
1.2	The Problem Addressed	1
1.3	The Significance	3
1.4	The Structure	3
2	The Object-oriented Paradigm	6
2.1	Defining the Paradigm	7
2.2	Defining the Approach to Analysis and Design Methods	17
2.3	The Analysis and Design Process	22
2.4	Modelling the Process	29
2.5	Summary	41
3	Applying the Object-oriented Paradigm	42
3.1	The Example	42
3.2	The Analysis and Design	44
3.3	Summary	72
4	Achieving Reuse of Software Components	73
4.1	The Reuse of Software Components	73
4.2	Storing and Retrieving Reuse Components	77
4.3	Full and Free Text Retrieval	79
4.4	Summary	83
5	The Search Tool	84
5.1	Development of the Search Tool	84
5.2	Operating the Search Tool	104
5.3	Summary	115
6	The Demonstration System	116
6.1	Completion of the Analysis and Design	116
6.2	Description of the ATM System	121
6.3	Summary	136
7	Conclusion	137

Reference List 140

Appendices

A Code for the Class Find Search Tool 146
B Code for the Word Index Class from Digitalk 157
C Code for the ATM 159

List of Figures

Figure

1 The symbol for a class. 30
2 The symbol for a subsystem. 30
3 Depiction of a state. 31
4 An event causing a change in state. 31
5 A state in which an activity is initiated. 32
6 An event with an action. 32
7 The symbols for a *Functional Model.* 33
8 State and transition symbols for state net diagram. 34
9 Sending a message to another object. 34
10 An association between object groups. 35
11 Various representations of one to many relationships. 36
12 Many to many object group associations. 36
13 Aggregation of object groups. 37
14 One to many relationships with participation constraints. 37
15 Many to many relationships with participation constraints. 38
16 Aggregation symbolised by a triangle. 38
17 Synchronous object group interaction. 39
18 Asynchronous object group interaction. 39
19 Conveying attribute information. 40
20 Modelling inheritance. 40
21 The high-level *Object Model* for the ATM. 45
22 The ATM subsystem as an aggregation of classes. 46
23 High dependency between objects. 47
24 The initial detailed *Object Model* for the ATM. 48
25 The *Dynamic Model* for Card Reader. 50
26 The *Dynamic Model* for User Interface. 51
27 The validate model for User Interface. 52
28 The cancel model for User Interface. 53
29 The deposit model for User Interface. 54
30 The withdraw model for User Interface. 55

Figure

31 The transfer model for User Interface. 56
32 The query model for User Interface. 57
33 The *Dynamic Model* for Account. 58
34 The deposit model for Account. 59
35 The withdraw model for Account. 60
36 The transfer model for Account. 61
37 The *Dynamic Model* for Deposit Slot. 62
38 The *Dynamic Model* for Dispenser. 63
39 The *Dynamic Model* for Receipt Printer. 63
40 The *Dynamic Model* for Customer. 64
41 The transact model for Customer. 65
42 The deposit model for Customer. 66
43 The withdraw model for Customer. 67
44 The transfer model for Customer. 68
45 Dictionary of activities and actions for the ATM. 69
46 The ATM *Object Model* with associations. 70
47 The *Object Model* with Customer interaction. 71
48 Dictionary of attributes for the ATM. 72
49 Dewey classification. 77
50 Growth of new words with an increase in the text size. 80
51 An inverted list. 81
52 The initial *Object Model* for Class Find. 86
53 The *Dynamic Model* for Class Find. 87
54 The operate model for Class Find. 88
55 The initialise model for Class Find. 89
56 The locate-classes model for Class Find. 90
57 The lookup model for Class Find. 91
58 The transfer-output model for Class Find. 92
59 Dictionary of activities and actions for Class Find. 92
60 Associations for Class Find. 93
61 Dictionary of attributes for Class Find. 94
62 The final *Object Model* for Class Find. 95
63 A Venn diagram for the test environment. 97
64 Initiating the search tool. 105
65 The initialised search tool. 105
66 The search tool showing menu choices. 106
67 Entry of terms to the search tool. 107
68 The result of a search. 108
69 Selection of the result of a search. 108
70 Selecting initiators and implementors of a service method. 109
71 The initiators and implementors of a service method. 110
72 Search tool dictionary contents for the test environment. 111
73 Test results on modified code for the search tool. 112
74 Determining the number of classes. 113

Figure

75 Test results on unmodified code for the search tool. 114
76 Test results for erroneous input to the search tool. 114
77 The final *Object Model* for the ATM. 119
78 Initiating the demonstration ATM. 122
79 Entering an account identification number. 123
80 Entering account balances. 124
81 Account details. 124
82 Changing account details. 125
83 The ATM waiting for use. 126
84 Entering card details. 126
85 Card held by ATM. 127
86 An unreadable bank card inserted. 127
87 Selecting the transaction. 128
88 Selecting the account. 129
89 Entering the amount. 129
90 Request for insertion of deposit. 130
91 Completion of a deposit transaction. 130
92 Dispensing money for a withdrawal transaction. 131
93 Insufficient funds for the requested transaction. 132
94 Request for source of funds. 132
95 Request for destination of funds. 133
96 Completion of a transfer transaction. 133
97 Result of a query transaction. 134
98 Cancellation of a transaction. 135
99 Test results from operation of the ATM. 135

1 Introduction

1.1 The Aim

The object-oriented paradigm shows considerable promise in terms of faster development of quality computing systems. The advantages claimed depend largely on the affinity that the object-oriented paradigm has for reusing software components; however, it is difficult to determine a structure suitable for storing and retrieving components for reuse. Modern reference and corporate library organisations have made advances in retrieving information that is unstructured. The study aims to demonstrate that text retrieval techniques of a modern library organisation may be applied to determine suitable classes for the object-oriented paradigm.

1.2 The Problem Addressed

Common libraries of modules available for reuse today include:

- the library supplied with the Smalltalk environment and the extensions for Dashboard and FreeDrawing that are provided by Digitalk;
- ObjectWorks from ParcPlace and PARTS from Digitalk that include software modules for sliders, radio buttons and other user interface tools;
- the C++ libraries from Microsoft and Borland;
- the library supplied with Eiffel;
- CommonViews from Glockenspiel; and
- Gehani's ADA modules - supplied with *Ada: An Advanced Introduction* (1989).

It may be expected that the number of such modules will continue to increase beyond the collection representing the most elementary operations to the situation where a module may be developed for many required operations. Thus, to attain a

working knowledge of the existing modules and to keep abreast of new modules are demanding tasks for today's programmers.

It is recognised that tools must be provided to enable and encourage software developers to find relevant modules for reuse. As Meyer (1988, p. 28) states, "the best reusable components in the world are useless if nobody knows they exist, if it takes a long time to obtain them, or if they cost too much". This is supported by Frakes & Nejmeh (1988, p. 142) who say "a fundamental problem in software reuse is the lack of tools to locate potential code for reuse".

Without suitable tools, software developers may come to know only a limited number of modules that might be employed in a practical application, with the remaining development implemented as newly written code. This is evidenced by the fact that, even though libraries of common code have existed for many years, software development has not taken full advantage of them. This position is supported in part firstly by Frakes & Nejmeh (1988, p. 142) who cite DeMarco as estimating "that in the average software development environment only about five percent of code is reused"; and secondly, by Hooper & Chester (1991, p. 1) who affirm that software libraries of reusable components have been used for many years.

A more beneficial approach to the discovery and use of relevant software modules is required. Frakes & Nejmeh (1988, pp. 144 - 145) suggest an approach based on the practice in modern library organisations for searching free-form text, which matches search words with document contents. One advantage of this approach is that it removes the necessity of guessing future requirements of information searchers, because the task of association is performed at the time of search rather than at the time of storage.

This temporal matching technique forms a significant part of the study. More specifically, it will be incorporated in a tool which indicates to the object-oriented software developer those reusable library modules that match his/her search specifications.

1.3 The Significance

Meyer (1938, p. 27) points out that "reusability as a dream is not new . . . . There should be catalogues of software modules . . . [so that] we would write less software, and perhaps do a better job at that which we do get to develop". Complementary to this, Henderson-Sellers (1992, p. 51) states that "reusability is one of the major advantages that an object-oriented approach can provide". Thus, taking both together, the significance of this study is to realise the dream of reuse by applying the full-text search technique within the object-oriented paradigm.

1.4 The Structure

Chapter 2 encapsulates a literature review of contributors in the field of object-orientation. Details of the associated principles are ascertained from many sources, including devotees such as Rumbaugh, Blaha, Premerlani, Eddy & Lorensen (1991), Rumbaugh (1995), Embley, Kurtz & Woodfield (1992), Wirfs-Brock, Wilkerson & Weiner (1990), Wirfs-Brock & Johnson (1990), Booch (1994), Henderson-Sellers (1992), Tanzer (1995) and from organisations such as the Object Management Group (Soley, 1992) and Digitalk (1992). Furthermore, an approach to object-oriented systems design is drawn from the work of Rumbaugh et al. (1991), initially and Embley et al. (1992), latterly. Then, continuing with an example of the paradigm, Chapter 3 incorporates the object-oriented design of an application that is

taken from the field of automatic teller processing, described in Wirfs-Brock, Wilkerson & Weiner (1990).

Chapter 4 includes a literature review of reusable software components and the way in which the reuse may be achieved. In general, the discussion embraces the gains offered by reuse, various traditional methods provided by reuse (for example, Booch, 1987) and the advantages to reuse tendered by the object-oriented paradigm (Hooper & Chester, 1991). Then, as a means of achieving reuse, the chapter incorporates a description of text retrieval techniques for libraries, based upon the work of Cortez & Kazlauskas (1986) and Salton (1989). Finally, this description serves as the foundation for a detailed account of the innovative process (the search tool) for matching search criteria with software module content within the object-oriented paradigm.

Chapter 5, using the principles outlined in the earlier chapters, establishes the design, specification and implementation of the object-oriented tool to achieve the desired selection and utilisation of existing software modules. The methods espoused for specifying the automated teller system in Chapter 3 are employed in the design of the matching tool.

Chapter 6 illustrates the result of using the search tool within a test environment that implements a subset of the aforementioned automatic teller processing application. The subset is limited to a portion that may be conveniently demonstrated on a personal computer; however, this does not indicate a limitation of the search tool itself.

Chapter 7 presents conclusions about the design, specification, implementation and proven capability of the search tool. Then, a judgement is given of its ability to

enhance the selection and utilisation of reusable software modules within the object-oriented paradigm. Subsequently, future research directions to augment the search tool as a viable innovation are suggested.

Finally, the study includes a number of appendices that describe the environment and technicalities of the tool developed.

2 The Object-oriented Paradigm

Object-orientation is a relatively new approach for the development of computer-based applications. Kuhn (1962), in his treatise on scientific method, talks of a paradigm shift as a stage in the development of major new ideas. Henderson-Sellers (1992, pp. 15 - 16) contends that object-orientation is a major paradigm shift similar in significance to "the Copernican revolution in astronomy, in Darwinian evolution, and in the adoption of the underlying ideas of plate tectonics in geology". Booch (1994, 40) maintains that the object-oriented paradigm "will form the foundation for the next generation of software architectures in numerous domains".

The thought process for object-orientation is different to the established methods that precede it. In support, Jacobson (1991, 35 - 36) affirms that "with an object-oriented analysis technique, it is possible to avoid . . . [thinking] like machines . . . which is a sheer waste of human activity", whereas in the object-oriented paradigm "they can . . . be made closer to life and thus become more comprehensible".

To provide a comprehensive discussion on the object-oriented paradigm, this chapter addresses four areas: namely, defining the paradigm; defining the approach to analysis and design methods; the design steps; and modelling the design steps. A separate section treats each area, reflecting its individual significance.

In the first section, an understanding of the object-oriented paradigm is based on a study undertaken by Henderson-Sellers (1992, pp. 18 - 28). Then, the fundamental principles of the paradigm are defined in terms of the literature available. Specifically, the principles of object, class, encapsulation, inheritance and polymorphism are stated.

With the foundation principles established, the second section considers the processes of analysis and design. There are many methods available; however, the discussion is limited to those methods further "evolved" by more recent writers. A rationale for the determination of an evolved method is given.

The third section outlines the method described primarily by Wirfs-Brock, Wilkerson and Weiner (1990). It provides a holistic description of the object-oriented analysis and design process, with detailed advice on determining class, inheritance relationships and collaborations between classes.

The fourth section describes the analysis and design methods from an object-oriented modelling perspective. The discussion is based on work by Rumbaugh et al. (1991) and Embley et al. (1992), because they satisfy the aforementioned rationale.

2.1 Defining the Paradigm

The important issues for the object-oriented paradigm may be found in Exhibit 8 of *A Book of Object-Oriented Knowledge* (Henderson-Sellers, 1992, p. 18). In this exhibit, the writings of twelve works on object-orientation are examined for the importance of information hiding, encapsulation, objects, classification, classes, abstraction, inheritance, polymorphism, dynamic binding, persistence and composition. Henderson-Sellers (1992, pp. 19 - 28) draws the conclusion that, for a design and/or implementation to be considered object-oriented, the following properties must be present:

- encapsulation;
- the idea of abstract classes; and
- polymorphism and inheritance.

This is supported by Booch (1994, 37) who says that "object orientation involves data abstraction, encapsulation, and inheritance with polymorphism. If any of these elements [i.e. properties] are missing, you have something less than object orientation". These properties are defined in the general discussion below.

The basis of the paradigm is the object which, according to the Object Management Group (Soley, 1992, p. 42), has a "distinct identity, which is immutable, persists for as long as the object exists, and is independent of the object's properties or behaviour". An object is defined by de Champeaux & Faure (1992, 22) as "a conceptual entity" that:

- refers to a thing identifiable by the users of the target system - either a tangible system or a mental construct
- has features that span a local state space
- has operators that can change the status of the system locally while these operations may induce invocations of operations in peer objects.

The essence of the concept of an object is the definition provided by Henderson-Sellers (1992, p. 19), who says that "an object is essentially an encapsulation of data and functionality". Thus, objects consist of data and procedures that manipulate some or all of the data in response to requests from within and without the object.

In addition to the software services needed to manipulate an object, the paradigm provides a mechanism for "grouping software ideas into classes of things" (Henderson-Sellers, 1992, p. 19). As Rumbaugh et al. (1991, p. 24) states:

> Identifying and documenting individual objects and relationships among objects is useful, but very tedious and not powerful enough for documenting most systems. . . . . To manage this complexity we need some method of abstracting and grouping a large body of facts into smaller, more comprehensive units.

To address the complexity, the object-oriented paradigm abstracts behaviour to a class, defined by Soley (1992, p. 67) as "an implementation that can be instantiated to create multiple objects with the same behaviour". Thus, every object is a member of a class which contains the description of the behaviour possessed by each of its objects. Advancing these notions, given a class, any two of its objects with the same state will behave in an independent, but exactly similar, manner when responding to a request of the same form and content.

The behaviour of an object is provided by its service methods, each of which is described by Soley (1992, p. 70) as the "code that may be executed to perform a requested service". While general terminology refers to these procedures as "methods", this study uses "service methods" in order to identify more explicitly their essential function. Wirfs-Brock & Johnson (1990, 106) points out that, for an object to satisfy its responsibility, "performing a request involves executing some code, a [service] method, on the associated data". This is supported by *Smalltalk/V for Windows Tutorial and Programming Handbook* (Digitalk, 1992, p. 68), which describes service methods as "the algorithms that determine an object's behaviour and performance" and are "like function definitions in . . . [structured] languages".

Apart from the classification of behaviour, the paradigm employs the software engineering technique of encapsulation to preserve the integrity of each object. A service method of an object operates only on that object's data and, as Wirfs-Brock & Johnson (1990, 106) say, other objects "are prevented from making direct access to the data". Rumbaugh et al. (1991, p. 7) describe this use of encapsulation as "separating the external aspects of an object, which are external to other objects, from the implementation details of the object, which are hidden from other objects". The Henderson-Sellers (1992, p. 19) perspective is that "it's gluing together data and functionality", while Jacobson (1991, 35) indicates that only the object knows its

internal structure. In the paradigm, the encapsulation process may realise the modularity principles espoused by software engineering authors such as Page-Jones (1988, pp. 57 - 102) for:

- low coupling - "the degree of interdependence between two modules"; and
- high cohesion - the manner in which "the activities within a single module are related to one another".

Due to the need for encapsulation, objects inherently lead a discrete existence. Thus, in a world made of many objects, another criterion to be addressed is a mechanism for communication. In the object-oriented paradigm, such communication is enacted by a message handling system supervised entirely by the environment, which are "similar to function calls in . . . languages" (Digitalk, 1992, p. 46). When a message is sent to an object, a service method is performed. In an example provided by Soley (1992, p. 42), if the date-of-birth is required of a Person object, a date-of-birth request is sent as a message to that Person object causing its date-of-birth service method to respond. Furthermore, it may be necessary to allow one object to cause complex resultant actions via a multiple message handling mechanism. Thus, an object-oriented programming environment may allow a structure of concatenated messages, when successive messages are sent to temporary objects resulting from preceding messages.

The data structure and behaviour of an object are not only available from the class to which an object belongs, but may also be "inherited" from other defined classes. Korson & McGregor (1990, 42 - 43) describe inheritance as "a relationship between classes that allows for the definition and implementation of one class to be based on that of other existing classes". That is, the class provides its objects not only with the behaviour described within it, but also with the behaviour of classes to which it is linked in a hierarchy. To provide inheritance within object-oriented

environments, the mechanisms of multiway tree data structures are provided which, as Booch (1987, p. 296) says, "derive much of their utility from the fact that they can represent a hierarchy among items". Specifically, the tree mechanism is described by Booch (1987, p. 297) in the following terms:

> A tree is a collection of *nodes* that can have an arbitrary number of references to other nodes. There can be no cycles or short-circuit references; for every two nodes there exists a unique simple *path* connecting them. . . . One [base] node is designated as the *root* of a tree. . . . If a given node references any other nodes, we say that it is the *parent* of these subordinate nodes; each of the subordinate nodes is a *child* of the parent. . . . we say that a parent node is the *ancestor* of its children and the children are *descendants* of their parent.

Within the inheritance hierarchy offered by the tree mechanism, a single parent class makes its data structure and service methods available to all its descendant classes. In the literature, a parent class is also known as a superclass and the descendant classes as subclasses. From the nature of the inheritance mechanism, a detailed account of the message handling process is as follows:

- A message is sent to an object under the environment's control.
- The system will parse the message in order to identify the service method to be performed on the object.
- If the object's class contains the service method, then the service method is performed.
- If the object's class does **not** contain the service method, then [in the purest object-oriented sense] a search of the tree mechanism is initiated. The search follows a single thread from the object's superclass towards the root, terminating at the first class containing the relevant service method. Then, the ancestral service method is performed on the object's data structure.

- If the search reaches the root class and if there exists no match of message to service method, then the message is erroneous.

Because the concept described above allows any class only a single superclass node, it is known as single inheritance. A discussion on multiple inheritance is deferred until later.

The environment allows a developer to add classes to any existing class library by employing the inheritance tree mechanism. This stresses the importance of inheritance to aid reusability of behaviour between objects of different classes. Furthermore, it provides a ready means of packaging classes in such a way that they may be conveniently used, with little or no modification, to solve new problems. An example of this use of inheritance is given by de Champeaux & Faure (1992, 24) as follows:

> Inheritance supports reuse of code in the following way: if a class A in a library is sufficiently close to fulfill a particular task, we incorporate A in the implementation, introduce B as a subclass of A, and make additions to B to reach the desired functionality . . . . It is permitted that these additions to B overwrite functionality available in A.

Modern programming languages may use the software engineering concept of overloading, an ability of service methods in different classes to respond correctly to identical messages. Overloading is defined by Henderson-Sellers (1992, p. 252) as "using the same operator symbol to mean two different things". For example, the '+' operator may be used to add:

- integers - for example (3 + 4);
- floats - for example (3.3 + 4.4); and
- points - for example (30, 60 + 40, 80).

Khoshafian & Abnous (1990, pp. 68 - 73) state that "overloading allows operations with the same name but different semantics and implementations to be invoked for objects of different types" which is "one of the most powerful and useful concepts of object orientation". These same authors continue with "object-oriented systems take overloading one step further and make it available for any operation of any object type" of which "the most important advantage is code saving". This ability, called polymorphism, is defined by Booch (1994, 37) as:

> a concept in type theory, according to which a name . . . may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways.

The polymorphic capability relies not only on the inheritance tree mechanism, but also on an ability to bind the service method to the object exactly at the time of message reception. That is, a class service method may be used by its subclasses without redefinition. Instead of describing common service methods for every class, they may be provided in classes towards the top of the hierarchical inheritance tree. Consequently, the object-oriented paradigm encourages a reduction in the amount of code that might be developed.

Khoshafian & Abnous (1990, pp. 133 - 136) point out that "in many situations . . . it is very convenient to allow a subclass to inherit from more than one immediate superclass" but that "combining instance variables or [service] methods of immediate predecessors is not . . . simple. The problem is that predecessors could have instance variables or [service] methods with the same name, but with totally unrelated semantics". Recognising these difficulties, Booch (1991, p. 110) states that "the need for multiple inheritance in object-oriented programming languages is still a topic of great debate". This may be, according to de Champeaux & Faure (1992, 25), because "multiple inheritance induces an ambiguity when a class inherits

conflicting features from parent classes. Each programming language has its own recipe for resolving such an ambiguity of which disallowing multiple inheritance is the most rigorous one". Until an environment is able to resolve automatically any conflict that may be caused by multiple inheritance, single inheritance is the safer option.

Inheritance fulfils the open-closed principle espoused by Meyer (1988, pp. 23 - 24), who says that each software module should be: open, as it is "still available for extension" because the developer will "seldom grasp all the implications of a subprogram"; and simultaneously closed, as it "may be compiled and stored in a library, for others to use". While the principle may appear contradictory, Henderson-Sellers (1992, p. 63) points out that, for the object-oriented paradigm, "once a class is tested and accepted into a library, it should not need to be 'opened-up' . . . while remaining 'open' to further extendibility by inheritance".

Many languages support, in varying degrees, the object-oriented paradigm. Indeed, Booch (1991, p. 494) provides a list of 112 such languages, three of which are deemed by Booch to be the "most influential and widely used", namely, Smalltalk, C++ and CLOS. To this collection, Rumbaugh et al. (1991, p. 318) add the Eiffel language. Each of these four languages supports the principles of objects and classes, encapsulation, single inheritance and polymorphism. Furthermore, Rumbaugh et al. (1991, p. 318) point out that "some languages, such as C++ (Version 2), CLOS [and] Eiffel . . . support multiple inheritance. Many others do not". Although Smalltalk employs single inheritance, Borning & Ingalls (cited in Booch, 1991, pp. 475 - 476) point out that multiple inheritance is also possible by redefining service methods within the language. Additionally, Ada 95 has recently become available; however, at the time of writing there is insufficient evidence for a comparison with the aforementioned languages.

As mentioned above, the first language identified by Booch is Smalltalk. According to Rumbaugh et al. (1991, p. 325), "Smalltalk is not only a language but also a development environment incorporating some functions of an operating system" of which a "strength is the class library, which was designed to be extended and adapted to meet the needs of the application". These authors add that "for a single-user development, it offers arguably the best features of both language and environment. . . . [to achieve] the goals of extensibility and reusability". In his discussion of Smalltalk, Booch (1991, pp. 474 - 475) says that "it is a 'pure' object-oriented programming language, in that everything is viewed as an object - even integers and classes"; further, that it is a "most important object-oriented programming language, because its concepts have influenced . . . almost every subsequent object-oriented programming language"; and finally, that the language laid "much of the conceptual foundation of . . . the ideas of message passing and polymorphism".

The second major object-oriented language is C++, of which Rumbaugh et al. (1991, pp. 326 - 327) say that "the implementation of run-time [service] method resolution is efficient" and further that:

> because of its origin as an extension of C, its backing by major computer vendors, the perception of it as a nonproprietary language, and the availability of free compilers, C++ seems likely to become the dominant language for general use.

However, reflecting that this language is a superset of the C language, Rumbaugh et al. (1991, p. 326) point out that "C++ is a hybrid language, in which some entities are objects and some are not", that "a C++ data structure is not automatically object-oriented", thereby placing "a serious restriction on the ability to reuse library classes by creating subclasses" and also that "C++ does not contain a standard class library

as part of its environment" with the consequence that "different libraries may be incompatible". The view of Booch (1991, p. 483 - 5) is that, instead of undergoing a formal design process, "design, documentation, and implementation went on simultaneously" until the language was considered complete and that "the definition of C++ does not include a class library".

Of the above two languages, Booch (1994, 38) says that:

> C++ and Smalltalk are the most pervasive object-oriented programming languages. It is likely that this situation will not change, but only become more entrenched over time. C++ has developed a following with organizations that are already experienced with C. . . . A striking difference between C++ and Smalltalk, however, is that the C++ environment is relatively tool-poor, whereas Smalltalk is relatively tool-rich.

The third language is CLOS - the Common Lisp Object System - which is the result, according to Booch (1991, p. 486), of a project undertaken in 1986 to standardise object-oriented dialects of LISP, "many of which were invented to support ongoing research in knowledge representation". Furthermore, Booch (1991, p. 488) points out that "the definition of CLOS does not include a class library". Independently, Rumbaugh et al. (1991, p. 328) say that "CLOS . . . has most of the advantages of a 'pure' object-oriented language" but that "CLOS currently does not have a class library", instead of which, class libraries are developed by individual users, with "some sharing of classes between organizations".

The last object-oriented language considered is Eiffel, about which Rumbaugh et al. (1991, p. 327) say "Eiffel has good software engineering facilities for encapsulation, access control, renaming, and scope. . . . [and that it] is arguably the best

commercial OO language in terms of its technical capabilities" but that only "a modest class library is provided". Henderson-Sellers (1992, pp. 263 - 236) points out that the Eiffel language has an intelligent compiler which undertakes class linkage and computation "without programmer prescription or intervention" and with a syntax that is relatively easy to learn. Because the output is C code, Eiffel is portable across hardware and operating system platforms.

One of the outcomes of this study is a software development tool which will assist in the selection and utilisation from a class library linked with an object-oriented language. To realise this outcome, a desirable characteristic of the software environment is a comprehensive class library from which a profile of each class may be developed. Given the above findings and the desire for an extensive class library, the Smalltalk language proved to be the only timely suitable choice.

2.2 Defining the Approach to Analysis and Design Methods

The purpose of this section is twofold: firstly, a clarification of the position of the study on analysis and design; and secondly, a determination of the latter-day literature to be reviewed, deemed necessary because there are numerous tomes that address these processes.

The first issue is the ambiguous delineation of the analysis and design processes that take place before programming begins. Olle et al. (1991, pp. 1 - 2) explain that "most information systems methodologies use the term 'analysis' to refer to an activity which precedes that of 'design' . . . . [and] before the design commences, it is logical enough to 'analyse' the environment in some way" in order to derive the resultant design specification, which "is what a designer can hand to a system constructor after he or she has completed the design". Extending this, Olle et al. (1991, p. 47 - 49) say that "any breakdown of the systems lifecycle into stages is

arbitrary" and that "it is often difficult to determine exactly where business analysis ends and system design begins". Independently, Kendall & Kendall (1992, p. 3) link the two activities together with a task description as follows:

> Systems analysis and design, as performed by systems analysts, seeks to analyze systematically the data input . . . and information output within the context of a particular business. Further, systems analysis and design is used to analyze, design, and implement improvement in the functioning of businesses.

Some authors differentiate analysis and design, as de Champeaux & Faure (1992, 21) explain:

> Twenty years ago, a distinction was made between analysis and design. Analysis is aimed at describing *what* a target system is supposed to do to obtain agreement with a customer . . . while design is aimed at describing *how* the desired system will work without going into implementation details . . . . in design, a solution is outlined, the required number of processes is determined, processes are allocated to processors, and algorithms and data structures are selected while satisfying additional resource, performance and contextual constraints.

In this vein, but with a contrasting differentiation, Pressman (1992, p. 146) describes the objectives of the analyst as:

- identify the customer's need;
- evaluate the system concept for feasibility;
- perform economic and technical analysis;
- allocate functions to hardware, software, people, database, and other system elements;
- establish cost and schedule constraints;

- create a system definition that forms the foundation for all subsequent engineering work.

The subsequent steps, according to Pressman (1992, p. 317), are:

- preliminary design which is "the transformation of requirements into data and software architecture"; and
- detail design which "focuses on refinements to the architectural representation that lead to detailed data structure and algorithmic representations for software".

As exemplified by the various positions of the above authors on this matter, the delineation is clearly a subjective selection. Consequently, the approach taken in this study is to treat the object-oriented analysis and design exercise as a continuum of detail description. At the beginning, the analyst/designer is concerned with the problem domain and a direction towards the solution. By the end, the detail of the solution within the specific environment of available class libraries should be understandable by the implementors of the solution.

The second issue (the aforementioned determination of literature to be reviewed) is based upon an investigation of object-oriented analysis and design conducted by de Champeaux & Faure (1992, 21), who point out that "in this early stage [of emerging new analysis and design methods] the methods diverge, as is to be expected".

Recapping the discussion in Section 2.1, it may be accepted that the method of analysis and design should incorporate the essential characteristics of encapsulation, inheritance and polymorphism. A number of the available methods do not appear to

support these essential characteristics and, of these methods, de Champeaux & Faure (1992, 27 - 29) observe the following:

- Edwards' method - "it appears that behaviour encapsulation is not supported".
- Coad & Yourdon's method - "seems not to offer parallelism for the objects"; that is, every object must cease operation and wait for a response to each request, rather than allowing independent operation for each object.
- Schlaer & Mellor's method - appears to be directly built from data analysis without the encapsulation of process.
- Bailin's method - "inheritance . . . is not mentioned".
- Colbert's method - "doesn't mention inheritance".
- Gibson's method - "inheritance . . . is not discussed".

The findings of de Champeaux & Faure (1992) are supported by Embley et al. (1992, p. 16), who describe the evolution that has occurred in some of these methods, stating that:

- The Object-Oriented Systems Analysis (OOSA) method of Schlaer & Mellor in 1988 is based on Entity Relationship models.
- The declarative, behavioural and interactive information presented within the Object-Oriented Analysis (OOA) method of Coad & Yourdon in 1990 is extended by the Object-oriented Modeling Technique (OMT) method of Rumbaugh et al. in 1991.

While the method described by Booch (1991) has not been included within the de Champeaux & Faure study, it is worthy of consideration for its development of multiple models that equate with the methods of Rumbaugh et al. (1991) and Embley et al. (1992), both of which are described below. Inherently, the Booch

model development requires the use of a computer drawing package in order to render a model. This need for graphic rendering sets Booch's modelling method apart from the others in the area and it is not obvious that the graphic intricacies benefit the analysis and design processes. Perhaps, practitioners may be more comfortable with model diagrams that may be easily sketched and immediately understandable by the implementors.

Further, Rumbaugh (1995, 21) advises that he has "accepted a position with Rational Software Corporation", leading to a partnership with Booch and "working to bring . . . methods together by a process of mutual evolution, so that eventually the differences will be minor and can be ignored. . . . [and] learning and using OMT will be protected under future method evolution".

According to de Champeaux & Faure (1992), the methods that satisfy the above essential characteristics are those of Wirfs-Brock, Wilkerson and Weiner (1990), Rumbaugh et al. (1991) and one under development by Kurtz, Woodfield, & Embley (n.d.). It is assumed that this last method has subsequently been described by Embley, Kurtz & Woodfield (1992). In support of the method described by Rumbaugh et al. in 1991, Embley et al. (1992, p. 16) observe that it "extends the declarative behaviour, ... has wide recognition and is the latest in the evolutionary sequence described". Further, D'Souza & Graff (1995, 23) say that "the OMT methodology is arguable one of the most popular for object-oriented development".

An examination of the object-oriented analysis and design literature is therefore based mainly on the well-founded methods described by Wirfs-Brock, Wilkerson & Weiner (1990), Rumbaugh et al. (1991) and Embley et al. (1992) and these are developed in the next two sections. The first of these sections investigates the

process of object-oriented analysis and design; then, the second concentrates on modelling the analysis and design aspects of object-oriented systems.

2.3 The Analysis and Design Process

Following the discussion above, this section establishes a holistic process for object-oriented analysis and design. The process is based particularly on the work of Wirfs-Brock, Wilkerson & Weiner (1990), Wirfs-Brock & Johnson (1990) and Rumbaugh et al. (1991).

From the writing of Wirfs-Brock, Wilkerson and Weiner (1990) and continued by Rumbaugh et al. (1991), object-oriented analysis and design for any problem demands the sequence of steps set out below:

- Understand the problem - its *raison d'être*, domain and specific parameters.
- Identify the objects - their abstract classes and subsystems (groups of classes).
- Determine the responsibilities of objects - their service methods.
- Determine the associations between objects - the messages they send and receive.
- Detail the attributes contained by objects - the data structures to represent their states.
- Build the inheritance links - their optimal positioning in the hierarchy.

These steps are achieved by the methods described in the remainder of this section and are undertaken in an iterative and incremental fashion. Booch (1991, p. 190) explains that object-oriented analysis and design is "an iterative process: implementing classes and objects often leads us to the discovery or invention of new classes and objects whose presence simplifies and generalises our design" and, further, that it "is an incremental process: the identification of new classes and objects usually causes us to refine and improve upon the semantics of and relationships among existing classes and objects". In support, Rumbaugh et al.

(1991, p. 166) say that "the entire software development process is one of continual iteration; different parts of a model are often at different stages of completion".

Understand the Problem

To obtain an understanding of the problem, Booch (1991, p. 191) explains that "by studying the problem's requirements and/or by engaging in discussions with domain experts, the developer must learn the vocabulary of the problem domain". Rumbaugh et al. (1991, p. 150) recommend the production of a "problem statement [which] should state what is to be done and not how it is to be done. It should be a statement of needs, not a proposal for a solution".

Identify the Objects

The objects and their abstract classes are determined once the problem is understood, for which Wirfs-Brock, Wilkerson and Weiner (1990, p. 38) explain that, from a written specification, the developer is "looking for noun phrases", in which the plural is changed to the singular, adding that "if you can formulate a statement of purpose for that candidate class, the chances are even higher it will be included in your design". The authors (1990, pp. 38 - 39) offer the following guidelines for choosing candidate classes:

- Model physical objects such as disks or printers on the network.
- Model conceptual entities that form a cohesive abstraction, such as a window or display, or a file.
- If more than one word is used for the same concept, choose [the] one that is most meaningful in terms of the rest of the system. . . .
- Be wary of the use of adjectives. . . . If the adjective signals that the behaviour of the object is different, then make a new class.
- Be wary of sentences in the passive voice, or those whose subjects are not part of the system. . . . Is it masking a subject that might be a class required

by your application? . . . Subjects [may be] things which are outside the system. . . . Does the sentence suggest an object that may need to be modeled? . . .

- Model categories of classes . . . . as individual, specific classes. . . . You will probably alter the taxonomy of classes later.
- Model known interfaces to the outside world, such as the user interface, or interfaces to other programs or the operating system, as fully as your initial understanding allows.

Wirfs-Brock, Wilkerson and Weiner (1990, p. 39) then say that "the result of this procedure is the first, tentative list of the classes in your program". In a similar vein, Rumbaugh et al. (1991, p.153) explain that "objects include physical entities, such as houses, employees, and machines, as well as concepts, such as trajectories, seating arrangements, and payment schedules" and advise "don't be too selective; write down every class that comes to mind. Classes often correspond to nouns".

Wirfs-Brock, Wilkerson and Weiner (1990, p. 47) advise that the list of classes may be refined, adding that the developer should re-examine the candidate classes "in order to identify as many abstract classes as possible . . . . in order to help identify the structure of the software . . . and to help identify classes . . . overlooked". Rumbaugh et al. (1991, pp. 153 - 155) explain that "if two classes express the same information, the most descriptive name should be kept"; that "if a class has little or nothing to do with the problem, it should be eliminated"; and that "a class should be specific. Some tentative classes may have ill-defined boundaries or be too broad in scope". Scharenberg & Dunsmore (1991, 32) discovered that an important step "was to realize that in the domain of objects . . . . [an] object became anthropomorphic (i.e., we began to think of it almost as a living entity able to tell us things about related objects) . . . . [although] not all objects became anthropomorphic". The step of identifying classes adds to the understanding of the

requirements and, according to Coad (1991, 44), "expands and refines the strategy of 'where to look, what to look for, and what to consider or challenge.' The strategy places extra emphasis on examining the problem domain and establishing the system's responsibilities in that context".

Taking abstraction further, Wirfs-Brock & Johnson (1990, 111) state that:

> a complex system requires many levels of abstraction, one nested within the other. Classes are a way of partitioning and structuring an application for reuse. But a design often has groups of classes that collaborate to fulfill a larger purpose. A subsystem is a set of such classes (and possibly other subsystems) collaborating to fulfill a common set of responsibilities.

They further explain that "subsystems simplify a design. A large application is made less complex by identifying subsystems within it and treating those subsystems as classes". They add that "a subsystem is not just a bunch of classes" and that "one way to test if a group of classes form a subsystem is to try and name the group. If the group can be named, the larger role they cooperate to fulfill has been named".

Determine the Responsibilities

The desired behaviour of the objects grouped within a class may be defined as the responsibility of the class. Each object accomplishes this responsibility with its own methods or the methods of objects which may be in other classes. Wirfs-Brock, Wilkerson and Weiner (1990, pp. 62 - 63) define responsibilities as "a sense of the purpose of an object and its place in the system" and "all the services it provides" and advise that responsibility should be shared among the classes of objects sharing a task so that they "evenly distribute system intelligence". According to Gibson (1990, 246), the objective is to "elicit a list of desired and necessary behaviors for

the system" and the way to achieve this is to "interview the users of the prospective application and observe them in action to see what they do, who and what they interact with, in what order, and what the outcomes of different actions are".

Determine the Associations

When an object requires the method of an object in another class to accomplish its responsibility, the object is said to form an association with that other object. An association is defined by Rumbaugh et al. (1991, p. 156) as "any dependence between two or more classes" and "a reference from one class to another". Furthermore, Tanzer (1995, 43) defines an association as "the set of the set of links between two (or more) objects of a single class or of different classes" where "a link is a physical or conceptual connection between objects". To understand the associations, Wirfs-Brock, Wilkerson and Weiner (1990, p. 91) advise the developer to:

> ask the following questions for each responsibility of each class. . . Is the class capable of fulfilling this responsibility itself? . . If not, what does it need? . . From what other class can it acquire what it needs? . . . [and] for each class, ask: What does this class do or know? . . What other classes need the result or information?

They further advocate that "if a class turns out to have no interactions with other classes, it should be discarded . . . [after] rigorous . . . check and cross-check". Another form of association is aggregation, described by Wirfs-Brock, Wilkerson and Weiner (1990, p. 92) as when "X's are composed of Y's" which "can sometimes imply a responsibility for maintaining information" and "often fulfill a responsibility by delegating the responsibility to one or more of their parts". Rumbaugh et al. (1991, pp. 156 - 160) advise that "associations often correspond to stative verbs or verb phrases. These include physical location . . ., directed actions . . .,

communication . . ., ownership . . ., or satisfaction of some condition". The authors warn that the developer should:

- eliminate any associations that are outside the problem domain or deal with implementation constructs;
- omit associations that can be defined in terms of other associations because they are redundant; and
- if one of the classes in the association has been eliminated, then the association must be eliminated or restated in terms of the other classes.

These same authors contend that at this stage the developer should "specify multiplicity, but don't put too much effort into getting it right".

Detail the Attributes

Rumbaugh et al. (1991, pp. 161 - 162) say that the next step is to identify the attributes of the object, which are the "properties of individual objects" and which "usually correspond to nouns followed by possessive phrases", however "if the independent existence of an entity is important, rather than just its value, then it is an object". Coad (1991, 44) says that "defining attributes adds strategy steps . . . and an overall emphasis on what an object is responsible to know over time (its state)".

Build the Inheritance Links

In order to identify the source of the behaviour of classes, the class hierarchy is determined. Wirfs-Brock, Wilkerson and Weiner (1990, pp. 119 - 121) say that:

> a good design balances the goal of small, easily understood and reused classes with the conflicting goal of a small number of classes whose relationships with each other can be easily grasped.

To achieve this, the authors advise the developer to seek "similar responsibilities that can be generalized, thus allowing them . . . to be moved higher in the hierarchy", then "remove unnecessary classes, and reassign their responsibilities

where needed". Rumbaugh et al. (1991, p. 163) add that "inheritance can be added in two directions: by generalizing common aspects of existing classes into a superclass (bottom up) or by refining existing classes into specialized subclasses (top down)". According to Rumbaugh et al. (1991, pp. 163 - 165), the former is achieved "by searching for classes with similar attributes, associations, or operations" while the latter "are often apparent from the application domain". They further advise that "attributes and associations must be assigned to specific classes in the class hierarchy. Each one should be assigned to the most general class for which it is appropriate" and that "multiple inheritance may be used to increase sharing, but only if necessary, because it increases both conceptual and implementation complexity".

In conclusion, the overall process for the developer is to:

1. gain sufficient understanding to be able to begin to solve the problem;
2. group real-life objects that exhibit identical behaviour into classes and, subsequently, classes into subsystems;
3. identify the responsibility of each object and, when appropriate, the associations they need to form in order to meet their commitments;
4. understand which information must be held by an object in order to perform its desired behaviour; and
5. for each class of objects, determine the class that may provide suitable inherited behaviour.

The means of achieving this process is described in the next section using the modelling techniques described by Rumbaugh et al. (1991) and Embley et al. (1992).

2.4 Modelling the Process

The need for modelling is best stated by Olle et al. (1991, p. 45), who explain that "the concept of modelling is inherent in any information system methodology" and that a "methodology should start with analytical modelling of a business area and continue with a prescriptive model for each information system".

Independently, Embley et al. (1992, p. 5) say that, to gather information and document systems, the analyst "concentrates on building a model . . . . [which] captures specific characteristics exhibited by system objects, and model construction drives the process of acquiring knowledge and asking questions about the system". Models are used extensively within the object-oriented paradigm and this is supported by de Champeaux & Faure (1992, 23) with their observation that, from structured analysis, "object-oriented analysis . . . methods have inherited . . . the usage of graphics instead of text to represent the models".

The information contained within the models may be supplemented by a dictionary description because, as Rumbaugh et al. (1991, p. 156) say, "isolated words have too many interpretations, so prepare a . . . dictionary . . . . [which] describes associations, attributes and operations".

The two methods described - the Object Modeling Technique (OMT) by Rumbaugh et al. (1991) and the Object-oriented Systems Analysis (OSA) method by Embley et al. (1992) - suggest similar modelling approaches for the object-oriented paradigm. Rumbaugh (1995, 21) points out that, guided by "user experience and good ideas from other authors, and new insights", OMT is evolving and a further book is in progress, which "will be the legitimate descendant of both [OMT and Booch] methods". Given the intrinsic need to understand the problem and using the process

steps discussed in the previous section as a foundation, the modelling techniques are described below.

Identify the Objects

The first step is to identify the object groups, which may be either classes or, as discussed previously, subsystems: the latter, for analysis and design purposes, may be modelled in the same manner as a class. A class is represented in the OMT *Object Model* by a rectangle as shown in Figure 1.

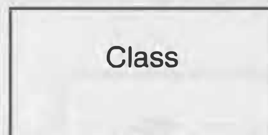

Figure 1. The symbol for a class.

If the model is complex, understanding is aided by dividing portions of the system into subsystems, each of which, according to Rumbaugh et al. (1991, p. 199), "encompasses aspects of the system that share some common properties".

Within the OSA *Object-Relationship Model*, object groups are represented by the same rectangle symbol. The OSA method specifically identifies abstract entities, which Embley et al. (1992, pp. 127 - 128) describe as "independent high-level object classes [that] have an identity of their own, . . . related to, but separate from the information they request" and which use the modelling symbol of a shaded rectangle shown in Figure 2.

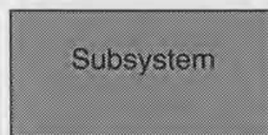

Figure 2. The symbol for a subsystem.

Determine the Responsibilities

The Rumbaugh et al. (1991, pp. 84 - 85) OMT *Dynamic Model* describes the behaviour of an object, employing state diagrams, which are "a graphical representation of finite state machines", to represent events and states depicting the internal changes that result from messages received. The authors explain that a state represents "attribute values and links" and an event "is something that happens at a point in time. . . . [because] an event has no duration . . . compared to the granularity of the time scale of a given abstraction". A state is depicted in the OMT *Dynamic Model* by a rounded rectangle, as shown in Figure 3.

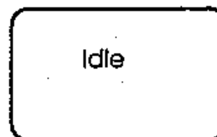

Figure 3. Depiction of a state.

In Figure 4, the event *time-out* causes the *Connect* state to be changed to the *Disconnect* state, with the dot describing the event *number-dialled* as an initial event for the model.

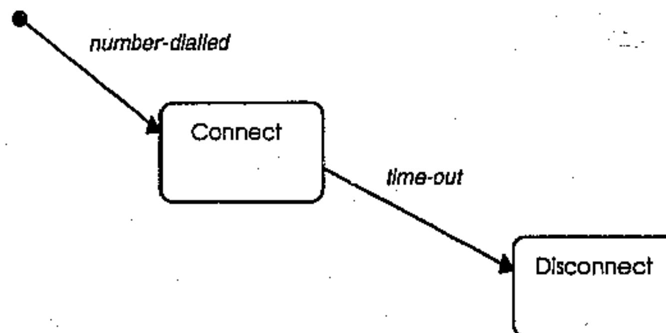

Figure 4. An event causing a change in state.

Entering a state may cause an activity to be initiated. An activity is an operation that takes time to complete and will be terminated when the state changes. In OMT,

Rumbaugh et al. (1991, p. 93) show an activity within a state symbol preceded by a "do:" notation, as shown in Figure 5.

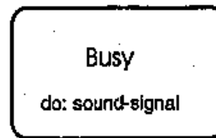

Figure 5. A state in which an activity is initiated.

An event may cause an action to take place, which Rumbaugh et al. (1991, pp. 92 - 93) describe as "an instantaneous operation. . . . associated with an event. . . . [that] represents an operation whose duration is insignificant compared to the resolution of the state diagram", such as the action **disconnect-line**, as shown in Figure 6.

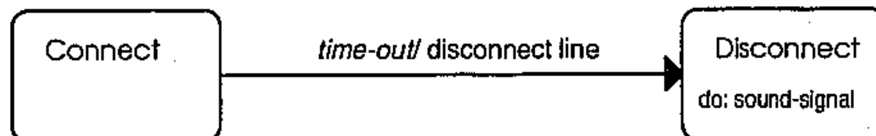

Figure 6. An event with an action.

The event may also carry information about a condition that must exist before the event may occur and which is included together with the event name in the form "*event* [condition]".

Additionally, OMT makes provision for modelling noninteractive programs with the *Functional Model*, of which Rumbaugh et al. (1991, pp. 123 - 124) say:

- it uses a data flow diagram technique, which is "a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects";
- it consists of "processes that transform data, data flows that move data, actor objects that produce and consume data, and data store objects that store data passively"; and

- it specifies "the results of a computation without specifying how or when they are computed".

The symbols used in a *Functional Model* are shown in Figure 7. Here, a rectangle symbolises an external actor, processes are shown by an ellipse, data stores are represented by parallel lines and arrows depict data flows.

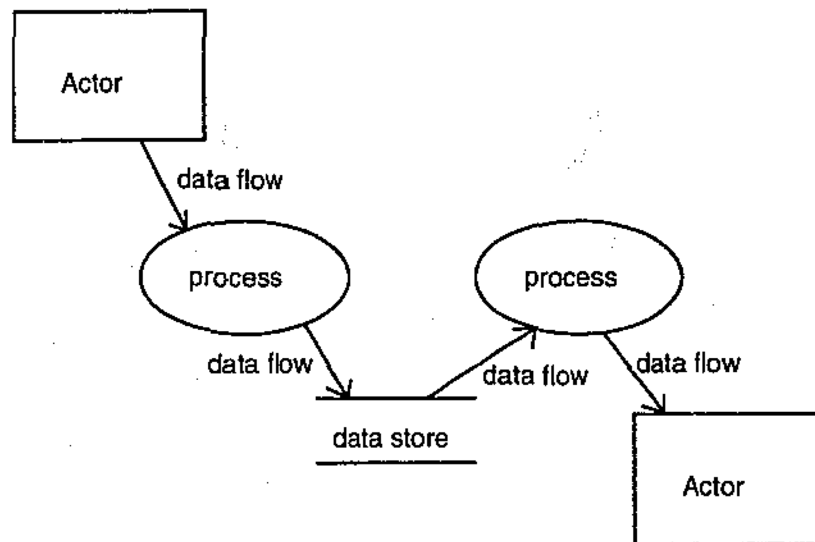

Figure 7. The symbols for a *Functional Model.*

According to Embley et al. (1992, pp. 60 - 79), "the objective of behaviour modelling is to understand and document the way each object in a system interacts, functions, responds or performs". They primarily employ state net diagrams to model the behaviour of each object, whilst providing a "shorthand version . . . similar to traditional finite state machine notation" such as that described above for the OMT approach.

From the description by Embley et al., 1992, p. 60), a state net diagram consists of:

- states, which represent "an object status, phase, situation, or activity";
- the change to another state by the connected transitions which consist of triggers, "the events and conditions that activate state transitions"; and

- actions, which "may cause events, create or destroy objects and relationships, observe objects and relationships, and send or receive messages".

In Figure 8, the states **Connect** and **Disconnect** are represented by the rounded rectangles and the transition between the states is represented by the rectangle, in which the top section contains the trigger description and the bottom section contains the action description. From the OSA description by Embley et al. (1991, p. 64), "the @ symbol . . . designates that the trigger is based upon an event. . . . [and is] read . . . as 'at', 'when', or 'upon'".

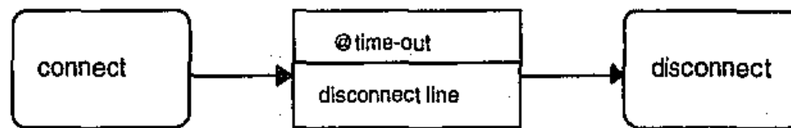

Figure 8. State and transition symbols for state net diagram.

A change of state in one object may cause a message to be sent to another object and OSA uses the 'lightning-strike' symbol shown in Figure 9 to represent, within one object, the message that becomes an event in another object. The message symbol may be shown emanating from either the transition or the state.

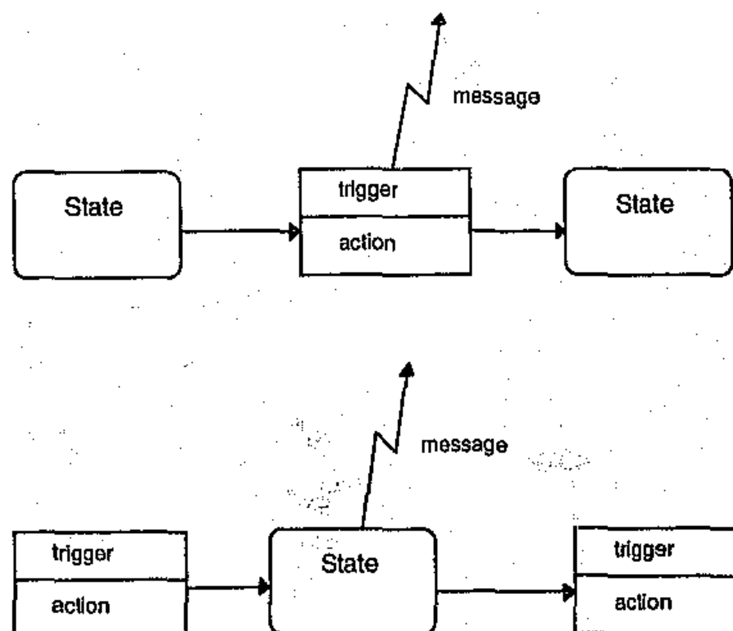

Figure 9. Sending a message to another object.

Determine the Associations

Within the OMT *Object Model,* Rumbaugh et al. (1991, pp. 27 - 28) symbolise an association as a straight line connecting the object groupings, as shown in Figure 10, explaining that "the name of a binary association usually reads in a particular direction, but the binary association can be traversed in either direction". The object groups may be classes or subsystems, depending on the level of abstraction to be represented, where each may appear within the same diagram.

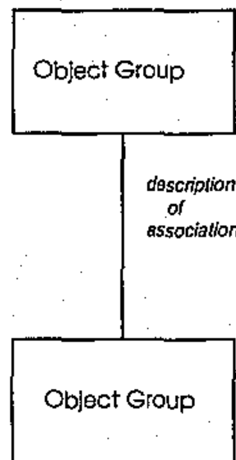

Figure 10. An association between object groups.

Rumbaugh et al. (1991, p. 30) state that "many instances of one class may relate to a single instance of an associated class". The symbols used to show the multiplicity of association are shown in Figure 11, where 11 (a) shows one class associated with a known range of many object groups, as indicated by the numbers at the multiple relationship end; 11 (b) shows one class associated with an unbounded upper range of many object groups, represented in a similar manner; and 11 (c) shows one class associated with an unknown number of many object groups symbolised by a solid circle on the tail.

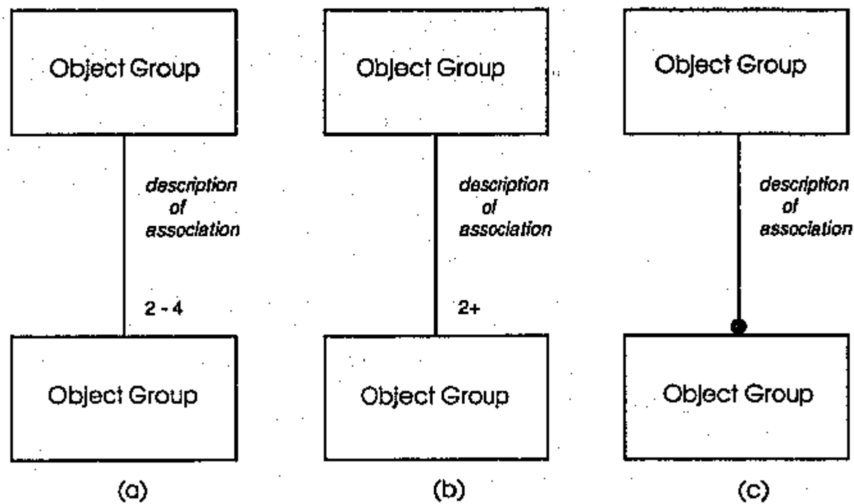

Figure 11. Various representations of one to many relationships.

A many to many relationship is symbolised by a solid circle on both the head and the tail, as shown in Figure 12.

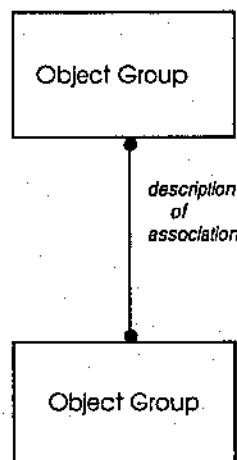

Figure 12. Many to many object group associations.

Rumbaugh et al. (1991, p. 58) say that "aggregation is a special form of association" where multiple objects "are tightly bound by a part-whole relationship" including "part explosions and expansions of an object into constituent parts", for example, "a company is an aggregation of its divisions". This type of association is symbolised by a diamond, as shown in Figure 13.

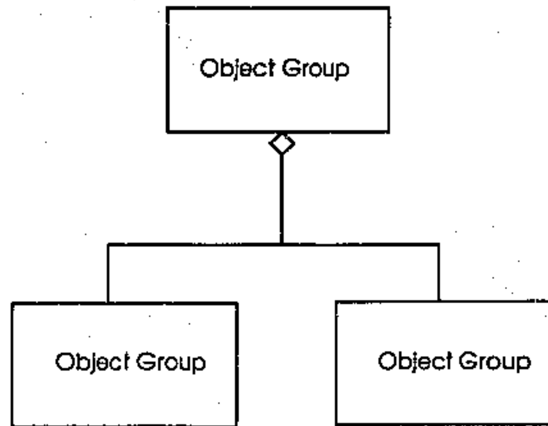

Figure 13. Aggregation of object groups.

As in the OMT method, associations are represented in the OSA *Object-Relationship Model* by straight lines joining the object groups. The multiplicity is described as participation constraints, shown in Figure 14, where 14 (a) indicates one object group associated with a known range of many object groups, 14 (b) shows one object group associated with an unbounded upper range of many object groups and 14 (c) shows one object group associated with an unknown number of many object groups.

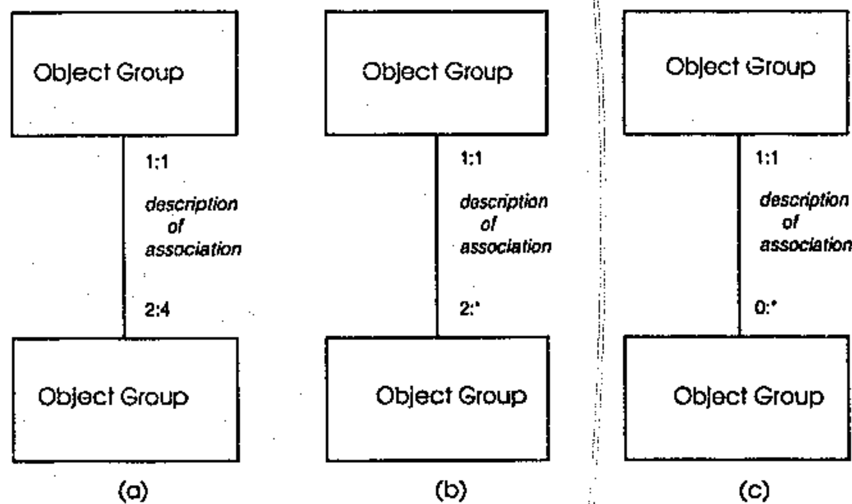

Figure 14. One to many relationships with participation constraints.

The multiplicity representation for many to many associations between object groups is shown in Figure 15, employing participation constraints.

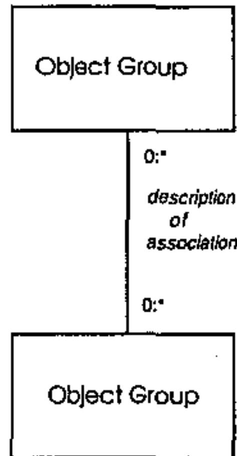

Figure 15. Many to many relationships with participation constraints.

The OSA method includes the ability to model aggregation in a similar manner to the OMT method, employing a solid filled triangle as shown in Figure 16.

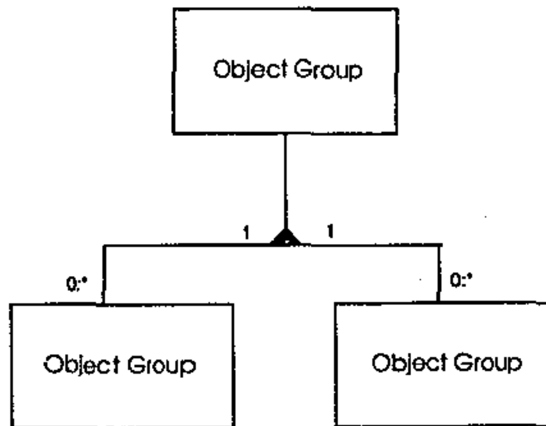

Figure 16. Aggregation symbolised by a triangle.

Embley et al. (1992, p. 167) point out that objects may interact with other objects in many ways, as "an object may send information to another object, . . . request information from another object, . . . alter another object, and . . . cause another

object to do some action" and, as shown in Figure 17, for synchronous interaction the object classes are joined with an extension of the message symbol.

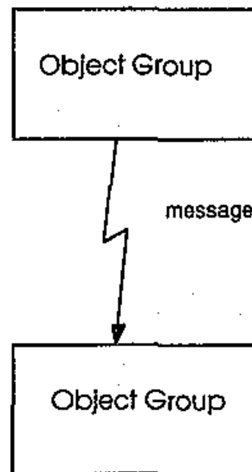

Figure 17. Synchronous object group interaction.

Asynchronous interaction is also covered by Embley et al. (1992, pp. 172 - 173), as they explain that objects "frequently interact indirectly with each other" because they may leave messages at an intermediate place and, in this case, the symbol is a solid circle as shown in Figure 18.

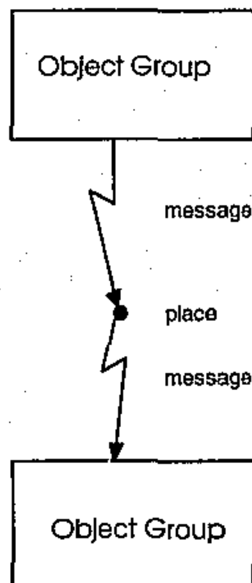

Figure 18. Asynchronous object group interaction.

Detail the Attributes

Within the OMT *Dynamic Model*, events convey information that may be data values - the object's attributes - and these are included with the event name, as shown in Figure 19 and which are listed in a dictionary.

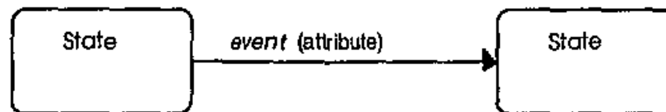

Figure 19. Conveying attribute information.

Build the Inheritance Links

Inheritance is symbolised by Rumbaugh et al. (1991, pp. 39 - 40) as a hollow triangle pointing from the subclass (or subclasses) to the superclass, as shown in Figure 20.

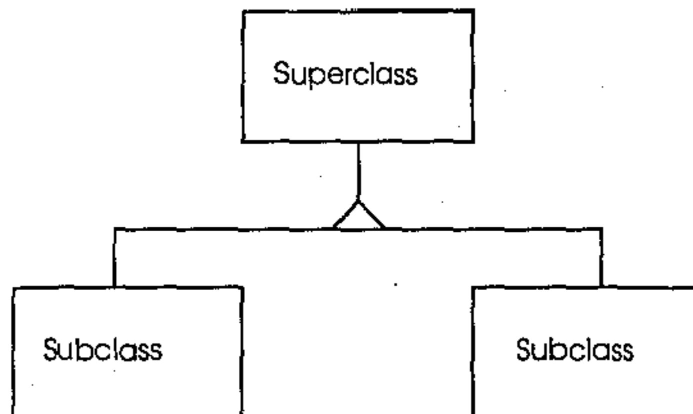

Figure 20. Modelling inheritance.

Embley et al. (1992, pp. 38 - 39) use the same symbol for inheritance (the *is-a* relationship), explaining that "we do not label an *is-a* relationship set because the transparent triangle tells us to read the relationship set as *is-a*".

2.5 Summary

This chapter has outlined the available literature on the object oriented paradigm and also some approaches to design employing the paradigm. The essential principles of the object-oriented paradigm include encapsulation, inheritance and polymorphism. A selection from the approaches of Wirfs-Brock, Wilkerson & Weiner (1990), Rumbaugh et al. (1991) and Embley et al. (1992) support the essential principles and describe the elements of a method that may be used. In the next chapter, an example analysis and design is undertaken.

3 Applying the Object-oriented Paradigm

The previous chapter provides a literature review of the object-oriented paradigm and of the applicable analysis and design process. Dependant upon the identified steps of this process, this chapter employs a modelling approach to analyse and design a partial solution for a selected example. The solution is partial because the choice of appropriate classes is deferred until later, when a search tool will be used to assist in the discovery of suitable reuse classes.

3.1 The Example

To ensure adherence to given specifications, the example is not fabricated specifically for this study, rather it is selected from the available literature in the field of object-oriented analysis and design. The example is taken from *Designing Object-Oriented Software* by Wirfs-Brock, Wilkerson & Wiener (1990, pp. 51 - 52) and is an Automatic Teller Machine (ATM) application. However, the hardware component for an ATM is unavailable and the example must simulate an ATM on a personal computer. The full description of the example is as follows:

> An automated teller machine (ATM) is a machine through which bank customers can perform a number of the most common financial transactions. The machine consists of a display screen, a bank card reader, numeric and special input keys, a money dispenser slot, a deposit slot and a receipt printer.
>
> When the machine is idle, a greeting message is displayed. The keys and deposit slot will remain inactive until a bank card has been entered.

When a bank card is inserted, the card reader attempts to read it. If the card cannot be read, the user is informed that the card is unreadable, and the card is ejected.

If the card is readable, the user is asked to enter a personal identification number (PIN). The user is given feedback as to the number of digits entered at the numeric keypad, but not the specific digits entered. If the PIN is entered correctly, the user is shown the main menu (described below). Otherwise, the user is given up to two additional chances to enter the PIN correctly. Failure to do so on the third try causes the machine to keep the bank card. The user can retrieve the card only by dealing directly with an authorised bank employee.

The main menu contains a list of the transactions that can be performed. These transactions are:

deposit funds to an account,
withdraw funds from an account,
transfer funds from one account to another, and
query the balance of any account.

The user can select a transaction and specify all relevant information. When a transaction has been completed, the system returns to the main menu.

At any time after reaching the main menu and before finishing a transaction (including before selecting a transaction), the user may press the cancel key. The transaction being specified (if there is one) is

cancelled, the user's card is returned, the receipt of all transactions is printed and the machine once again becomes idle.

If a deposit transaction is selected, the user is asked to specify the account to which the funds are to be deposited and the amount of the deposit and is asked to insert a deposit envelope.

If a withdrawal transaction is selected, the user is asked to specify the account from which funds are to be withdrawn and the amount of the withdrawal. If the account contains sufficient funds, the funds are given to the user through the cash dispenser.

If a transfer of funds is selected, the user is asked to specify the account from which the funds are to be withdrawn, the account to which the funds are to be deposited and the amount of the transfer. If sufficient funds exist, the transfer is made.

If a balance enquiry is selected, the user is asked to specify the account whose balance is requested. The balance is not displayed, but is printed on the receipt.

3.2 The Analysis and Design

In this section, the specification from the previous section, which supplies an understanding of the problem, is modelled for each of the process steps. There is no definitive modelling methodology; indeed, the one employed here is drawn from both Rumbaugh et al. (1991) and Embley et al. (1992). However, while generally adhering to the former, this section avails of features from both of them wherever, in

the author's opinion, these features better suit the object-oriented analysis and design continuum.

Identify the Objects

The first step is to identify and represent the object abstractions in an *Object Model* and, to assist the reader, a convention of bold print will be adopted within the thesis text to identify object groupings (classes and subsystems). From nouns within the example, the candidate object groups immediately identified are:

ATM	**Card Reader**
Customer	**Dispenser**
Screen	**Receipt Printer**
Keys	**Account**
Deposit Slot	

Of these candidate classes, the **ATM** may be considered a subsystem that associates with the **Customer** and **Account** classes, leading to a high level view of the *Object Model* shown in Figure 21.

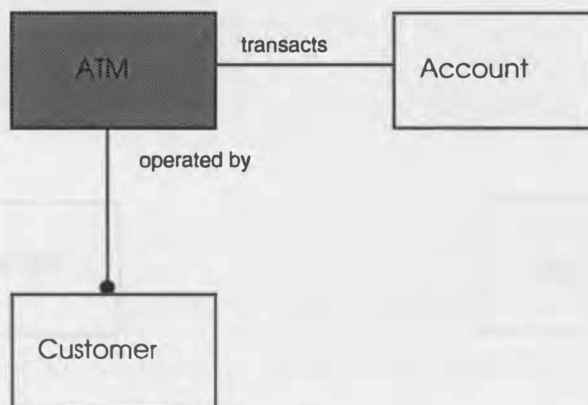

Figure 21. The high-level *Object Model* for the ATM.

The nouns **Screen, Keys, Deposit Slot, Card Reader, Dispenser** and **Receipt Printer** signify aggregate components of an **ATM**, represented in Figure 22.

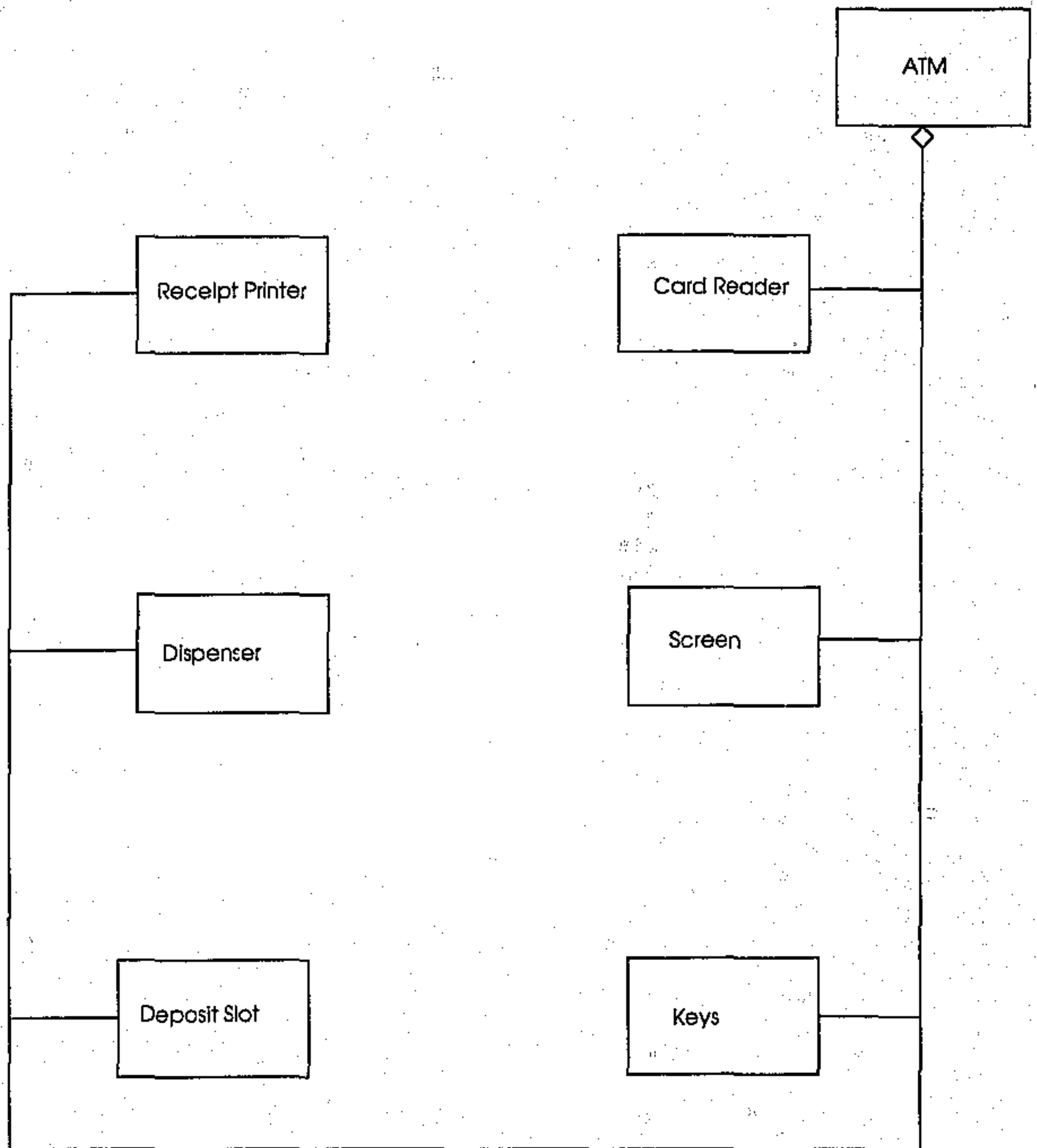

Figure 22. The ATM subsystem as an aggregation of classes.

The classes **Deposit Slot, Card Reader, Dispenser** and **Receipt Printer** may be considered to be self-sufficient; however, the same cannot be said of the classes **Screen** and **Keys** which, as is shown in Figure 23, exhibit a strong dependency on each other.

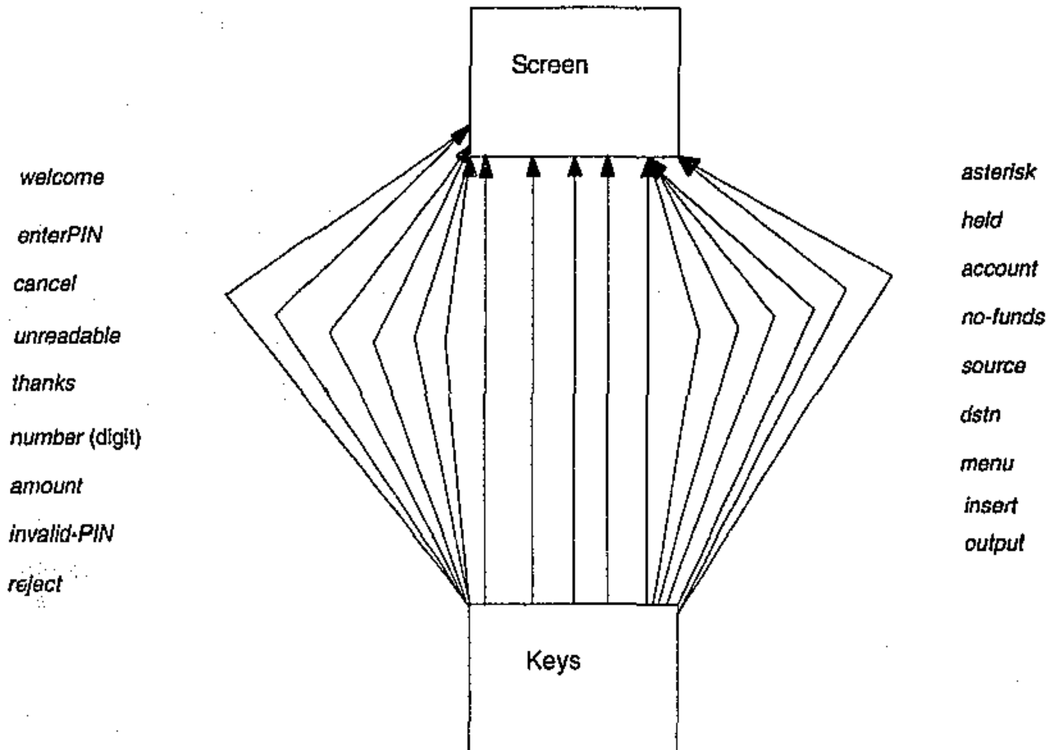

Figure 23. High dependency between objects.

The single self-reliant class **User Interface** may be substituted for the classes **Screen** and **Keys** and the *Object Model* may be redrawn devoid of associations except for the aggregation association, as shown in Figure 24.

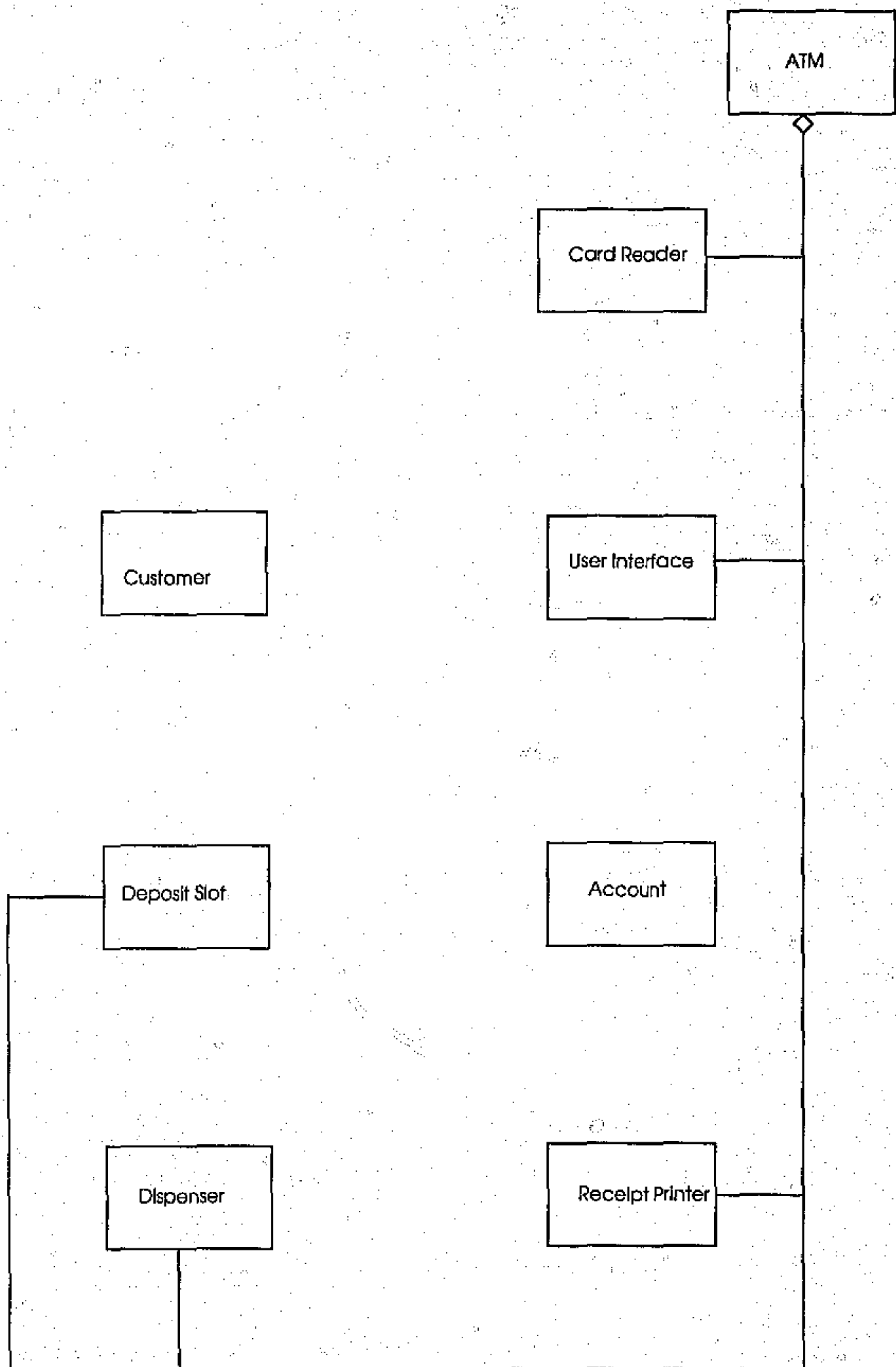

Figure 24. The initial detailed *Object Model* for the ATM.

Determine the Responsibilities

The next step is to model the responsibilities of the objects identified above by constructing *Dynamic Models*, each of which will determine a service method. To assist the reader in distinguishing text relating to these models, the adopted convention is to represent *Dynamic Model* information in Helvetica font, with the following additional identification: **States** in bold; *events* in italic; (attributes) bracketed; and 'activities' and 'actions' enclosed by single quotation marks.

For the **Card Reader**, Figure 25 shows that from an **Idle** state the event *readable-card-inserted* causes the action 'read-pin-id' to be initiated. The object then enters a **Hold Card** state that initiates an association with **User Interface** by sending the message validate and passing the attributes (PIN and id), which is an event causing a change of state in **User Interface**. The **Hold Card** state remains until, under normal circumstances, the event *eject* - a message from another object - initiates the action 'eject-card' and returns the object to the **Idle** state. Alternatively, the *not-PIN* event initiates the 'keep-card' activity, then the *card-kept* event changes the state to **Idle**. Inserting an unreadable card causes the event *unreadable-card-inserted*, changing the state to **Unreadable** and causing a message to be sent to the **User Interface**. Then, the *message-sent* event results in an 'eject-card' action and a return to the **Idle** state.

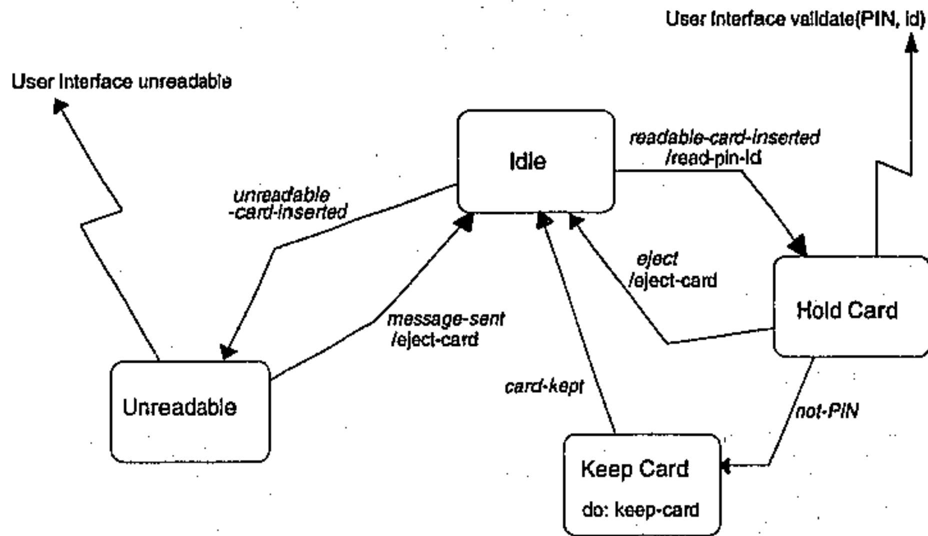

Figure 25. The *Dynamic Model* for Card Reader.

The *Dynamic Model* for the **User Interface** is modelled in Figure 26, showing that the event *validate* changes the state to **Validate,** which initiates the 'validate' activity. If the PIN is incorrect, the event *close-sent* causes a return to the **Idle** state, otherwise the *validated* event initiates the 'menu' activity to display the transaction selection menu within the **User Interface.** The selection of a transaction key causes either a *deposit-pressed, withdraw-pressed, transfer-pressed* or *query-pressed* event, which changes the state to **Deposit, Withdraw, Transfer** or **Query** and initiates the relevant activity, following which the *complete* event changes the state to **Idle.**

User Interface

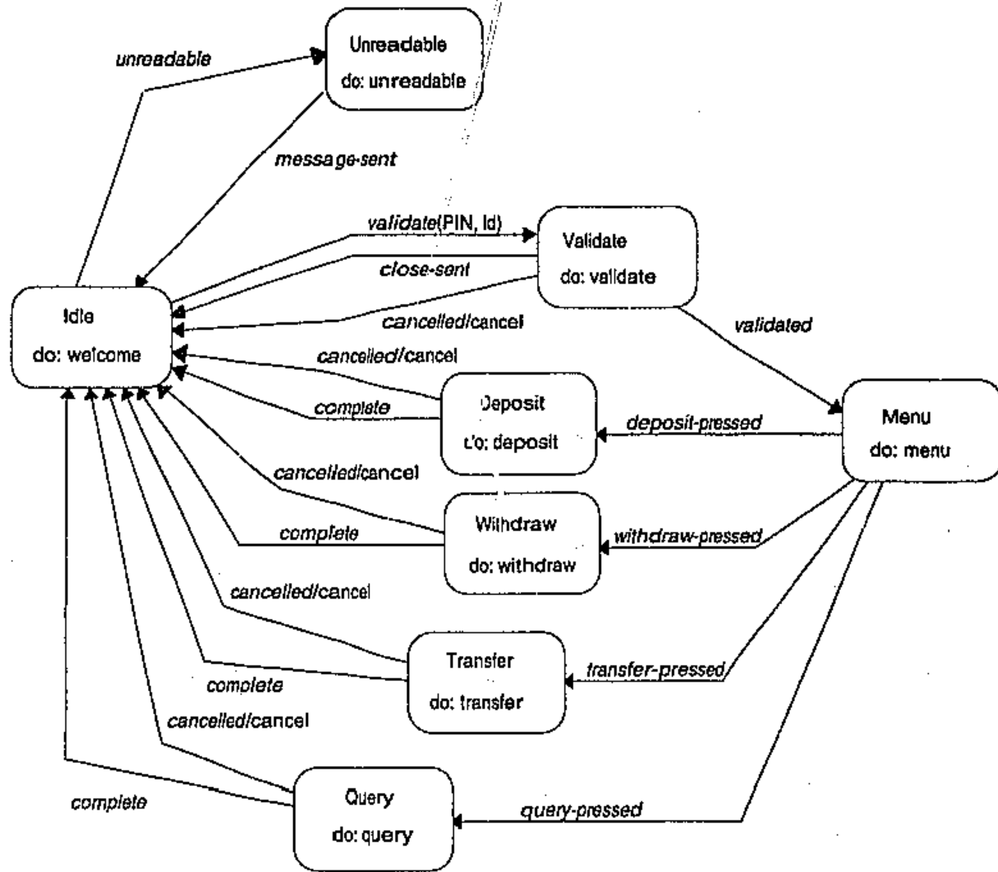

Figure 26. The *Dynamic Model* for User Interface.

The detail of the 'validate' activity may be found in another model, which is displayed in Figure 27. The events *validate, validated* and *close-sent* are repeated in this model as a point of reference in order to locate the model for the 'validate' activity in the environment of the **User Interface**. Within the **Store** state, the (PIN) and (id) attributes are held for later comparison, then the *attributes-stored* event changes the state to **Enter PIN**. As each digit of the PIN is pressed, the *number-key-pressed* initiates the 'asterisk' activity. When the PIN has been entered, the *OK-key-pressed* event changes the state to **Compare PIN**. At most, three invalid entries are allowed, therefore the *not-PIN* event initiates the 'invalid-PIN' activity, then either the *retry-allowed* event iterates the process through the **Enter PIN** state or the *PIN-rejected* event changes the state to **Held**, in which the user is informed that the card

will not be returned and a message sent to **Card Reader**, then the *close-sent* event follows. The *is-PIN* event changes the state to **Validate**, in which a message is sent to **Account** to check the (id) attribute. An *invalid-account* event changes the state to **Held** or a *valid-account* event changes the state to **Validated**, resulting in a *validated* event.

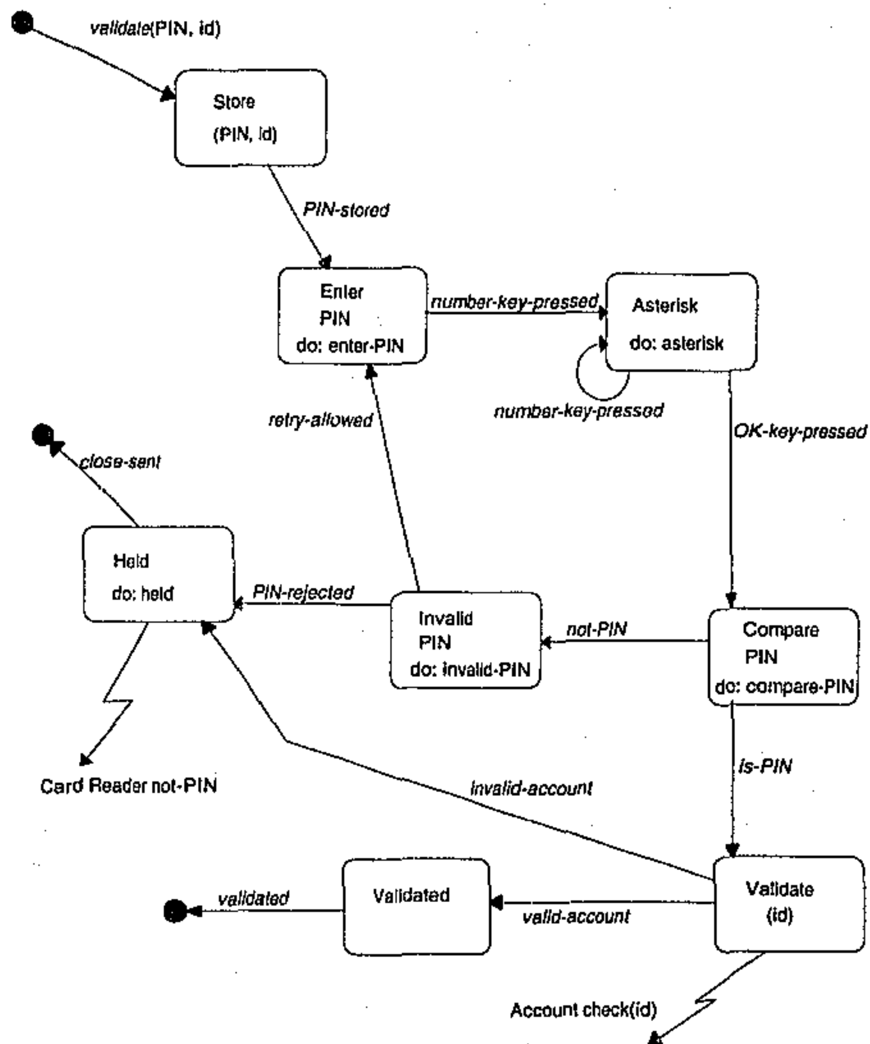

Figure 27. The validate model for User Interface.

On pressing the cancel key, the action 'cancel' sends a message to the **Card Reader** and the 'cancel' action informs the user that the transaction has been cancelled, as shown in Figure 28.

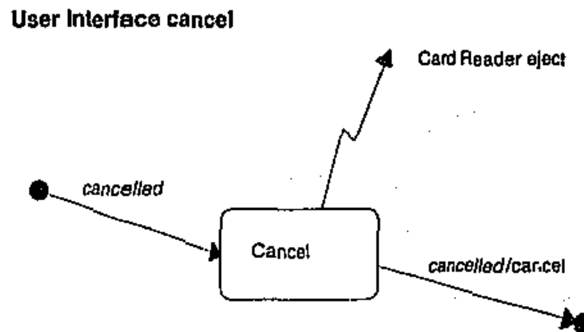

Figure 28. The cancel model for User Interface.

In Figure 29, the event *deposit-pressed* initiates the 'account' activity to ascertain the banking account number. The *account-selected* event changes the state to **Store** the attribute (account), then the *account-stored* event initiates the 'amount' activity to obtain the deposit amount. Each *numeric-pressed* event iterates through the **Getting Amount** state until the *ok-pressed* event occurs, initiating the 'insert' activity and sending a message to **Account**. The *action-complete* event initiates the 'thanks' activity, sending a message to the **Card Reader** to cause the card to be ejected, which is followed by the *complete* event.

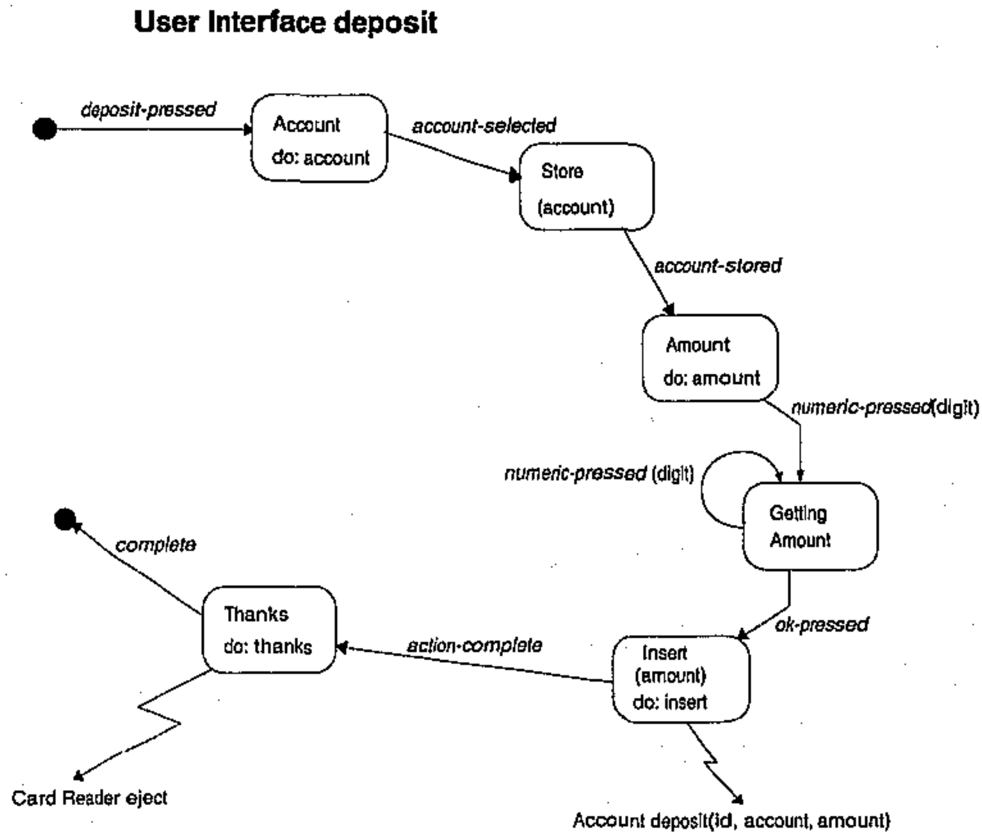

Figure 29. The deposit model for User Interface.

In a similar manner, the event *withdraw-pressed* initiates the 'account' activity, as shown in Figure 30. The *account-selected* event changes the state to **Store** the attribute (account), then the event *account-stored* initiates the 'amount' activity to obtain the withdrawal amount. Each *numeric-pressed* event iterates through the **Getting Amount** state until the *ok-pressed* event occurs, initiating the 'output' activity and sending a message to **Account.** Then, either the *insufficient-funds* event initiates the 'no-funds' action and the 'thanks' activity or the *action-complete* event initiates the 'thanks' activity, sending a message to the **Card Reader** to cause the card to be ejected and resulting in the *complete* event.

User Interface withdraw

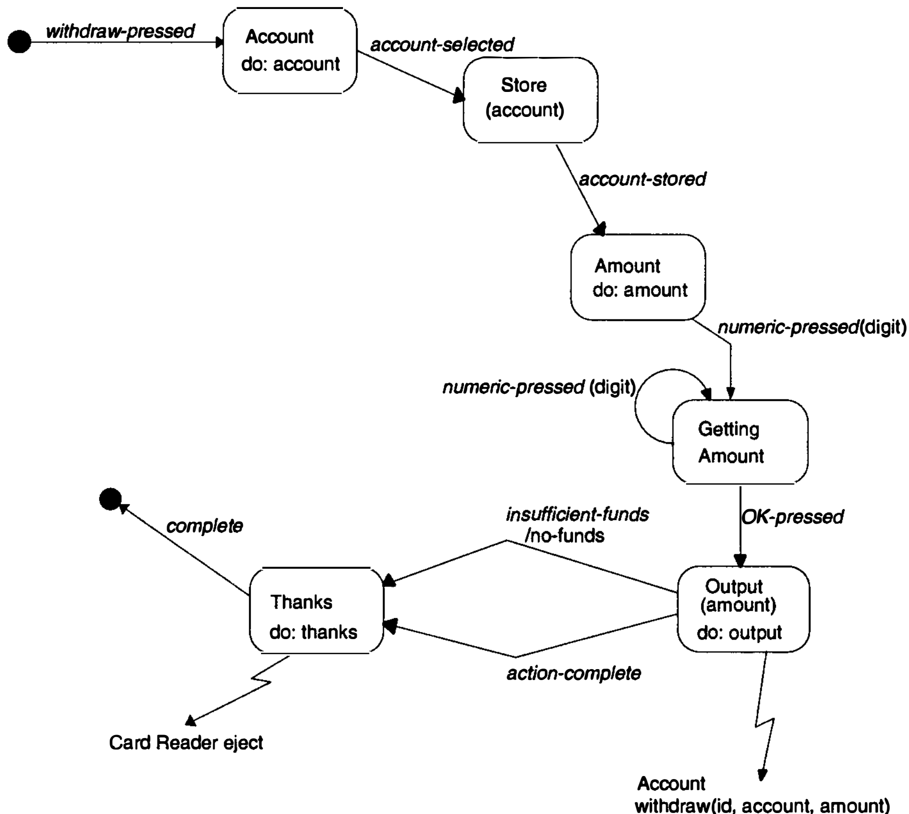

Figure 30. The withdraw model for User Interface.

The event *transfer-pressed* results in behaviour similar to those in the two previous models, as shown in Figure 31. The 'source' activity obtains the account number from which funds are transferred, then the *src-account-selected* event changes the state to **Store** the attribute (src-account), following which the event *src-account-stored* initiates the 'dstn' activity to obtain the account to which funds are transferred. The *dstn-account-selected* event changes the state to **Store** the (dstn-account) attribute, then the *dstn-account-stored* event initiates the 'amount' activity to determine the amount to be transferred. Each numeric-pressed event iterates through the **Getting Amount** state until the *ok-pressed* event occurs, changing the state to **Store** the (amount) attribute and sending a message to **Account**. Either the

insufficient-funds event initiates the 'no-funds' action and the 'thanks' activity or the *action-complete* event initiates the 'thanks' activity, sending a message to the **Card Reader** to cause the card to be ejected and resulting in the *complete* event.

User Interface transfer

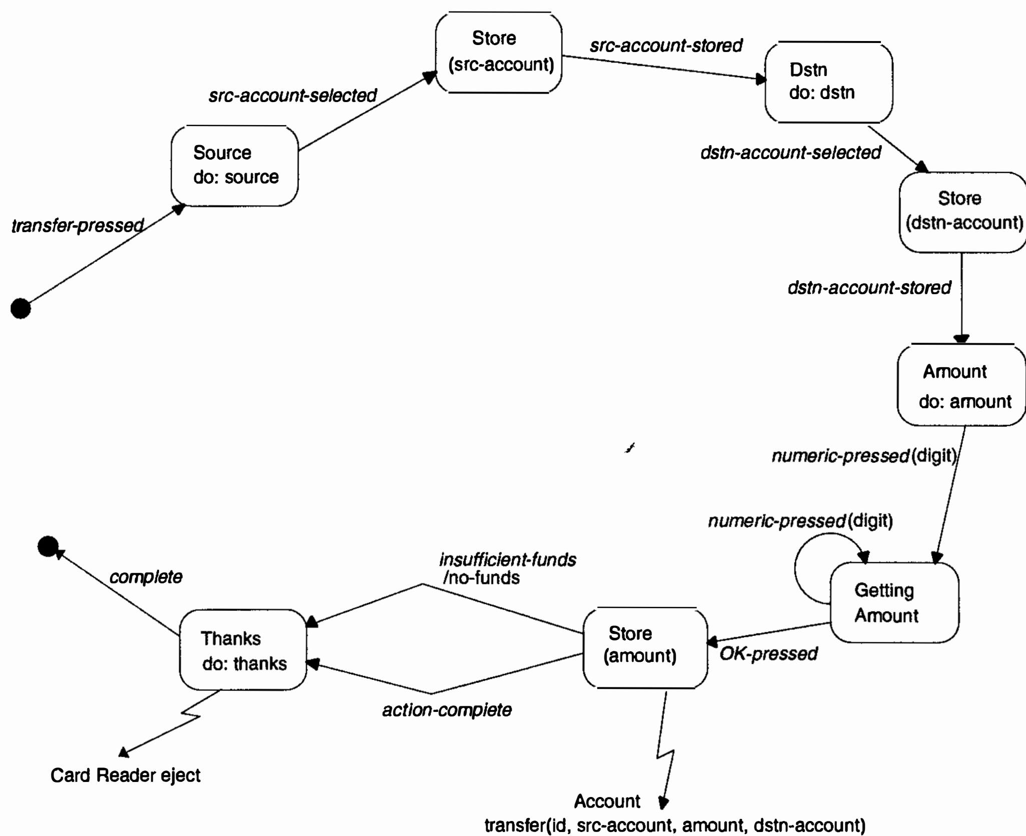

Figure 31. The transfer model for User Interface.

The event *query-pressed* results in the initiation of the 'account' activity, as shown in Figure 32. The *account-selected* event changes the state to **Store** the (account) attribute, then the *account-stored* event changes the state to **Query Account** in which a message is sent to **Account** - initiating the account balance printout - then the *action-complete* event changes the state to **Thanks** where a message is sent to the **Card Reader**, then the *complete* event follows.

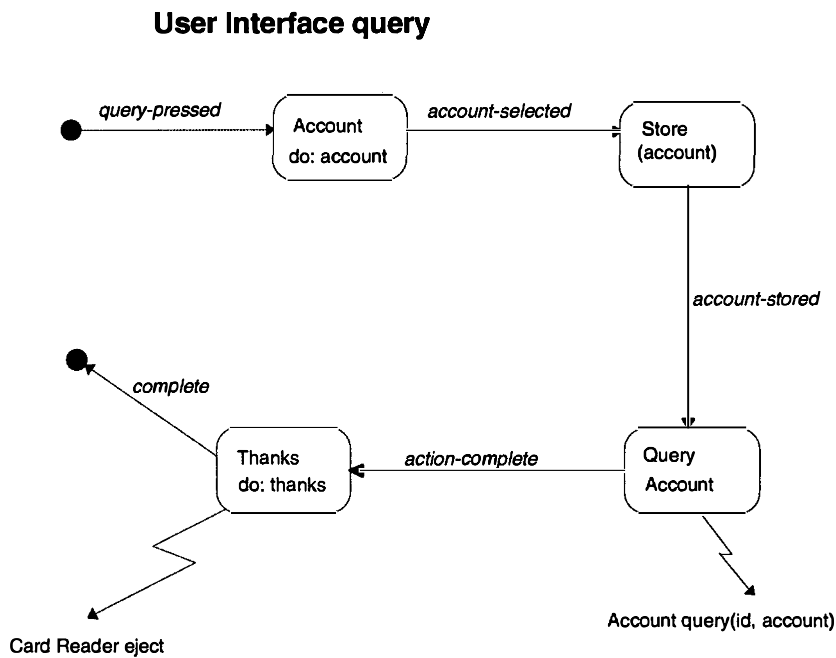

Figure 32. The query model for User Interface.

To perform its transaction responsibilities the **User Interface** associates with **Account**, which is modelled in Figure 33. A *check* event changes the state from **Idle** to **Check Account**, in which messages are sent to the **User Interface**, then the *id-checked* event changes the state to **Idle**. A *query* event changes the state to **Query Amount** in which messages are sent to the **User Interface** and the **Receipt Printer**, then returning to the **Idle** state as a result of the *transaction-complete* event. The *deposit*, *withdraw* and *transfer* events change the state to **Deposit, Withdraw** or **Transfer** respectively, then either a *timeout* or *transaction-complete* event returns the state to **Idle**.

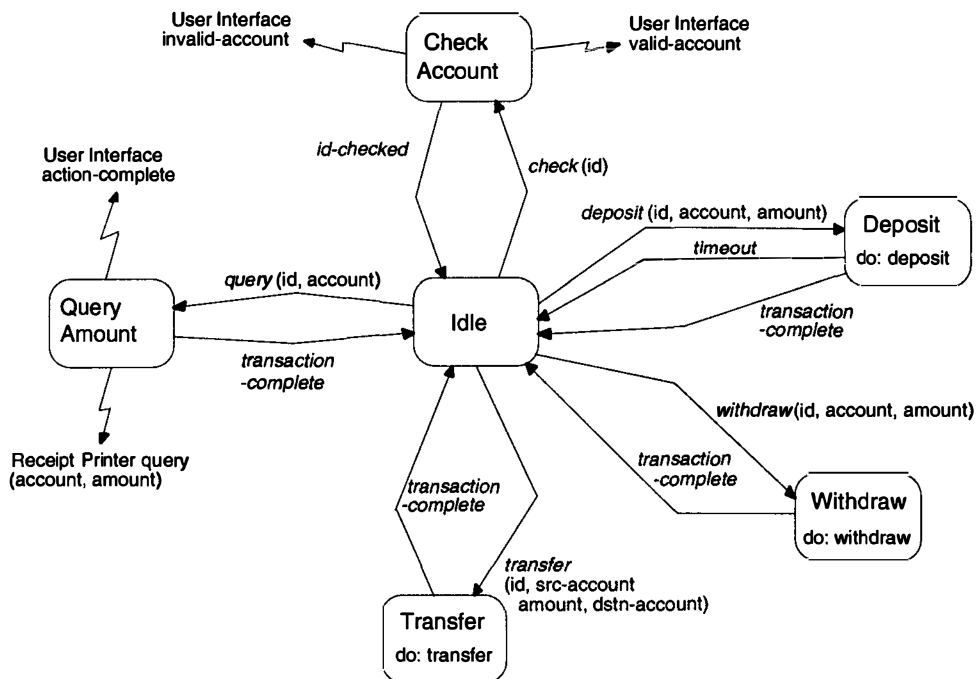

Figure 33. The *Dynamic Model* for Account.

The behaviour of the 'deposit' activity is displayed in Figure 34, showing that the event *deposit* changes the state to **Wait for Deposit**, in which a message is sent to the **Deposit Slot**. Then, either the *timeout* event occurs or the *action-complete* event changes the state to **Add**, when the (amount) attribute is added to the balance for the (account) attribute and messages are sent to the **User Interface** and the **Receipt Printer**, then the *transaction-complete* event follows.

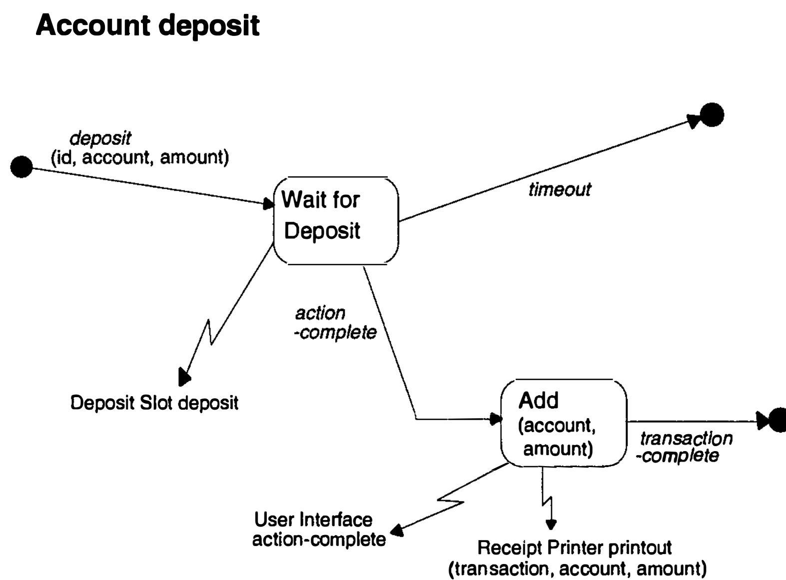

Figure 34. The deposit model for Account.

The 'withdraw' activity is displayed in Figure 35, showing that the *withdraw* event changes the state to **Subtract** where the (amount) attribute is subtracted from the (account) attribute. The *insufficient-funds* event changes the state to **No Funds** where a message to that effect is sent to the **User Interface**, followed by the *transaction-complete* event. The *sufficient-funds* event changes the state to **Wait for Withdraw**, in which a message is sent to the **Dispenser**, then the *action-complete* message changes the state to **Withdrawal Complete**, in which messages are sent to the **User Interface** and the **Receipt Printer**, followed by the *transaction-complete* event.

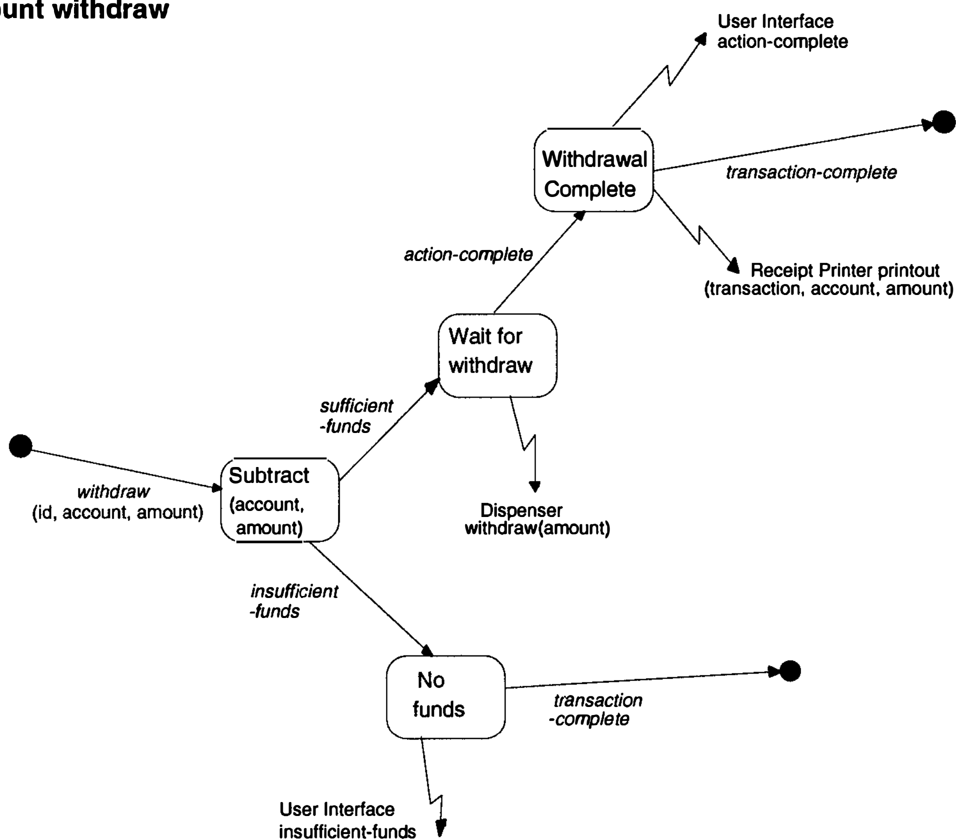

Figure 35. The withdraw model for Account.

As shown in Figure 36, the *transfer* event changes the state to **Subtract** where the (amount) attribute is subtracted from the (account) attribute. The *insufficient-funds* event changes the state to **No Funds** at which time a message to that effect is sent to the **User Interface**, then the *transaction-complete* event follows. The *sufficient-funds* event changes the state to **Add**, where the (amount) attribute is added to the balance for the (account) attribute and messages are sent to the **User Interface** and the **Receipt Printer**, followed by the *transaction-complete* event.

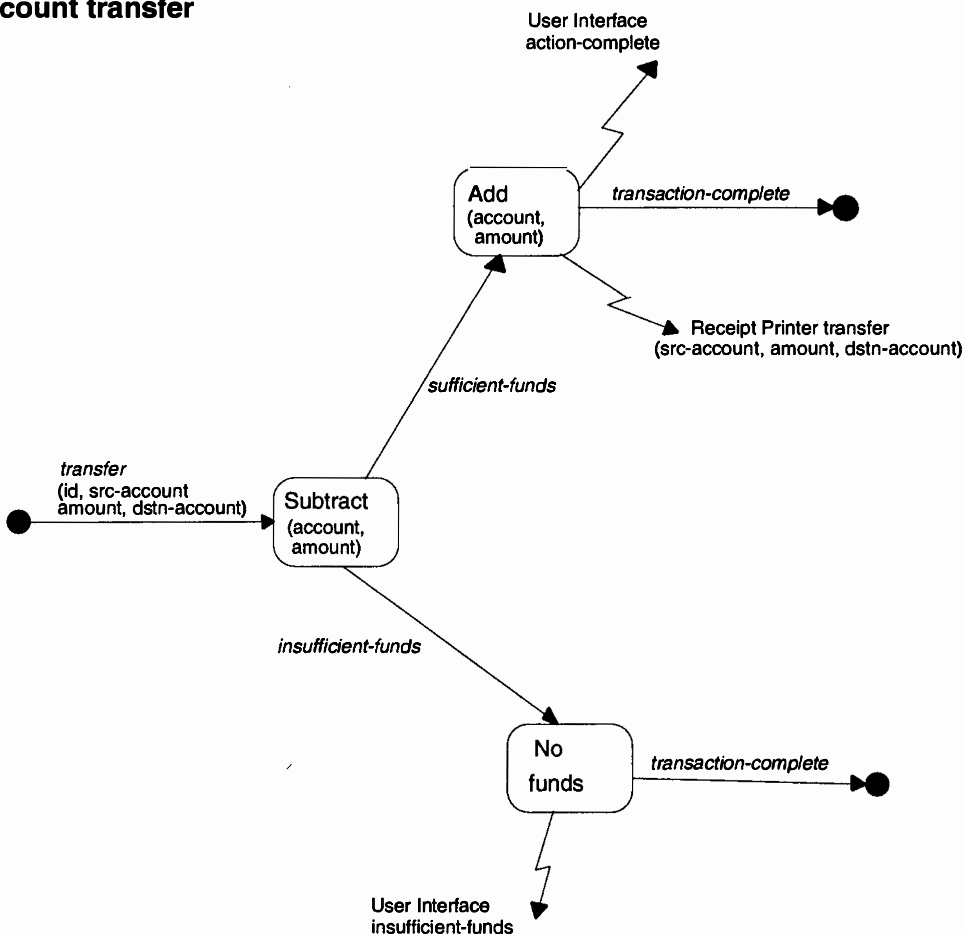

Figure 36. The transfer model for Account.

As shown in Figure 37, an association between the **Account** and the **Deposit Slot** objects leads to the event *deposit* changing the state from **Idle** to **Money**. Then, either the *timeout* event changes the state to **Timed Out**, at which time a message is sent to **Account** then the *message-sent* event follows, or the *money-input* event changes the state to **Input Received**, in which a message is sent to **Account** followed by the *message-sent* event to change the state to **Idle**.

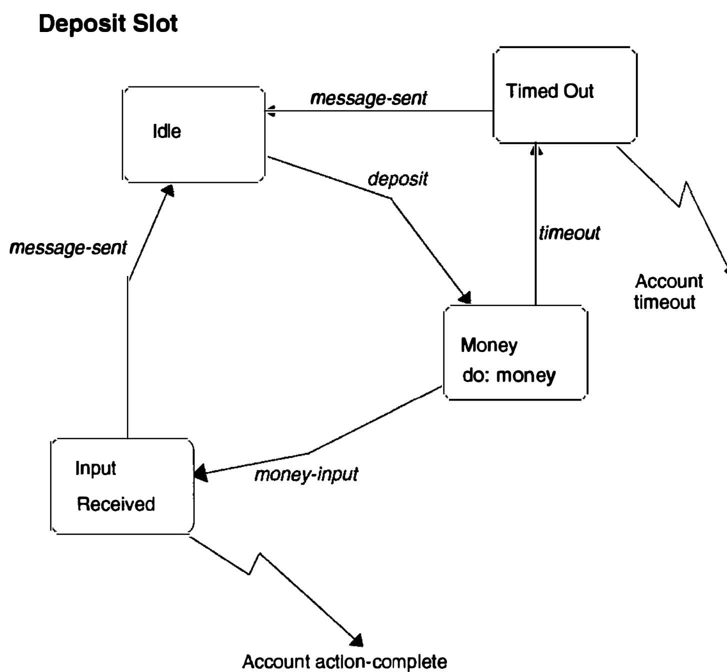

Figure 37. The *Dynamic Model* for Deposit Slot.

An association also exists between the **Account** and **Dispenser** objects, as shown in Figure 38. The *withdraw* event changes the state from **Idle** to **Dispense Money**, then the *money-dispensed* event changes the state to **Advise**, when a message is sent to **Account**, followed by the *message-sent* event to change the state to **Idle**.

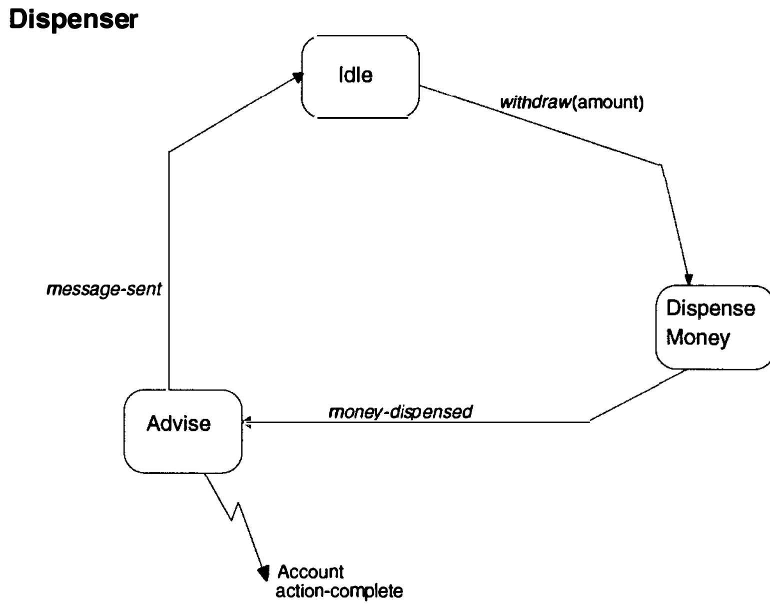

Figure 38. The *Dynamic Model* for Dispenser.

The **Receipt Printer** issues receipts as a result of *printout* or *transfer* or *query* events, as shown in Figure 39. Then, the *receipt-printed* event changes the state to **Eject Receipt**, followed by the *receipt-ejected* event to change the state to **Idle**.

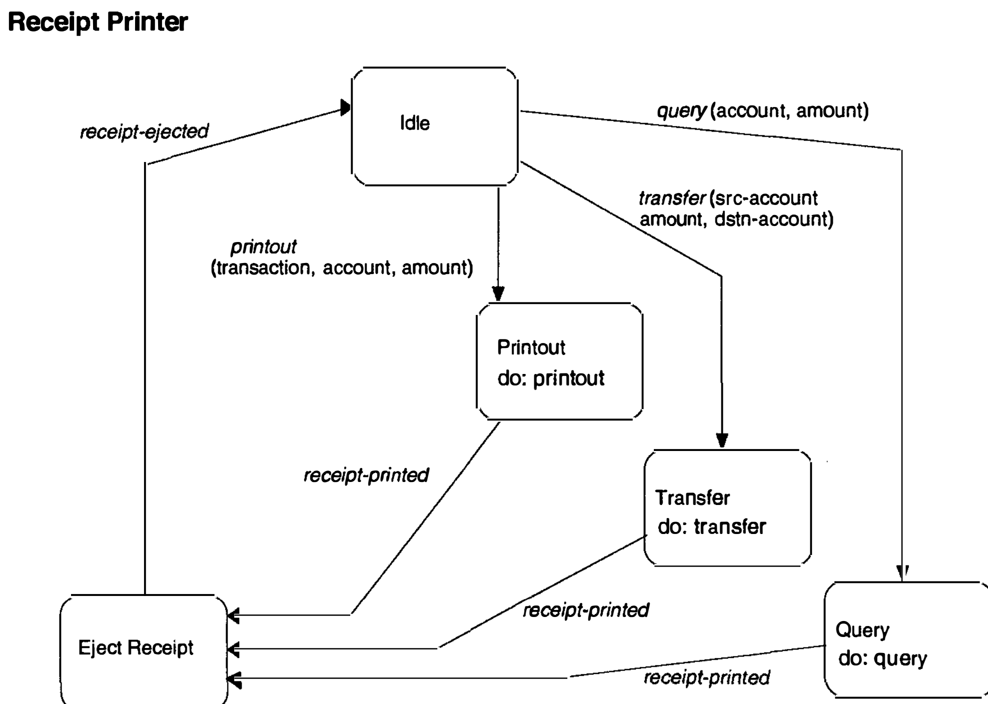

Figure 39. The *Dynamic Model* for Receipt Printer.

The **Customer** sends messages to the classes that make up the ATM in response to events initiated by them and, although not resulting in the construction of software, the **Customer** is modelled as an integral part of the system and serves as mechanism for checking the completeness of the other models. As shown in Figure 40, the **Customer** inserts a card and then, in response to the *enter-PIN* event, enters the PIN - which may require re-entry - changing the model to the **Transact** state and initiating the 'transact' activity, after which the card is ejected. Entry of an unreadable card results in an *unreadable* event, a message is displayed in state **Bad Card**, then the *eject-card* event returns the state to **Not Involved**.

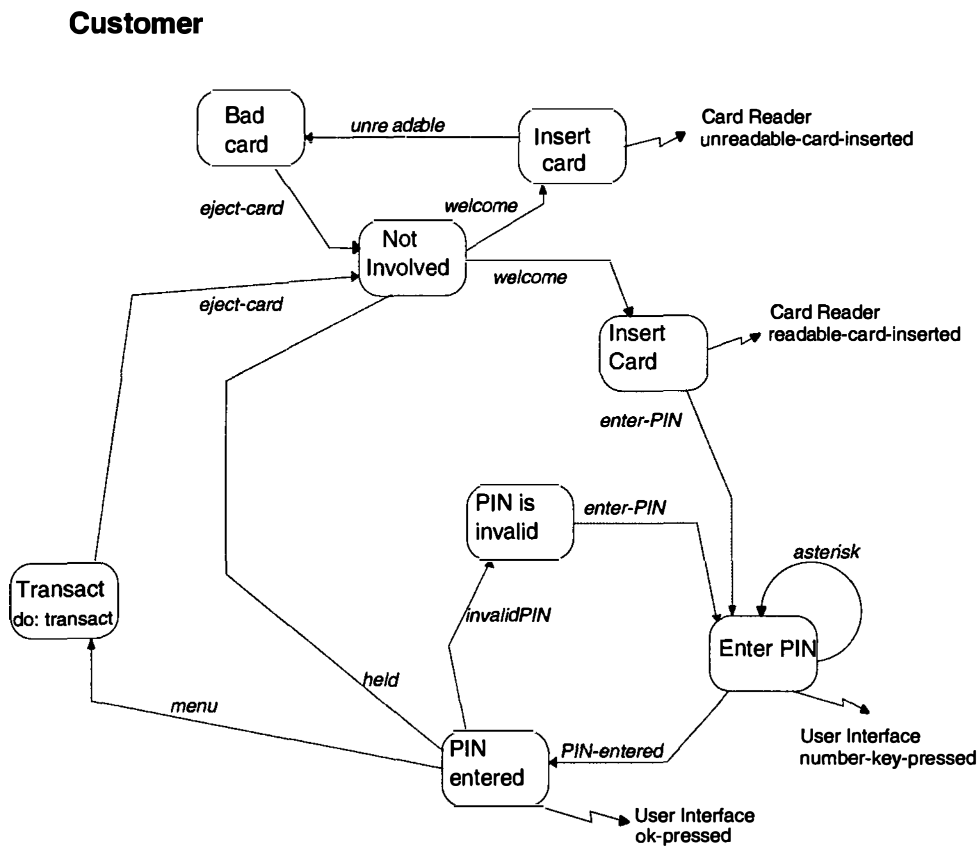

Figure 40. The *Dynamic Model* for Customer.

The 'transact' activity is modelled in Figure 41, which shows that a key is pressed in response to the *menu* event, resulting in either the **Deposit** state initiating a 'deposit' activity, the **Withdraw** state initiating a 'withdraw' activity, the **Transfer** state initiating a 'transfer' activity or, for a query, the selection of an account resulting in the printout of a receipt.

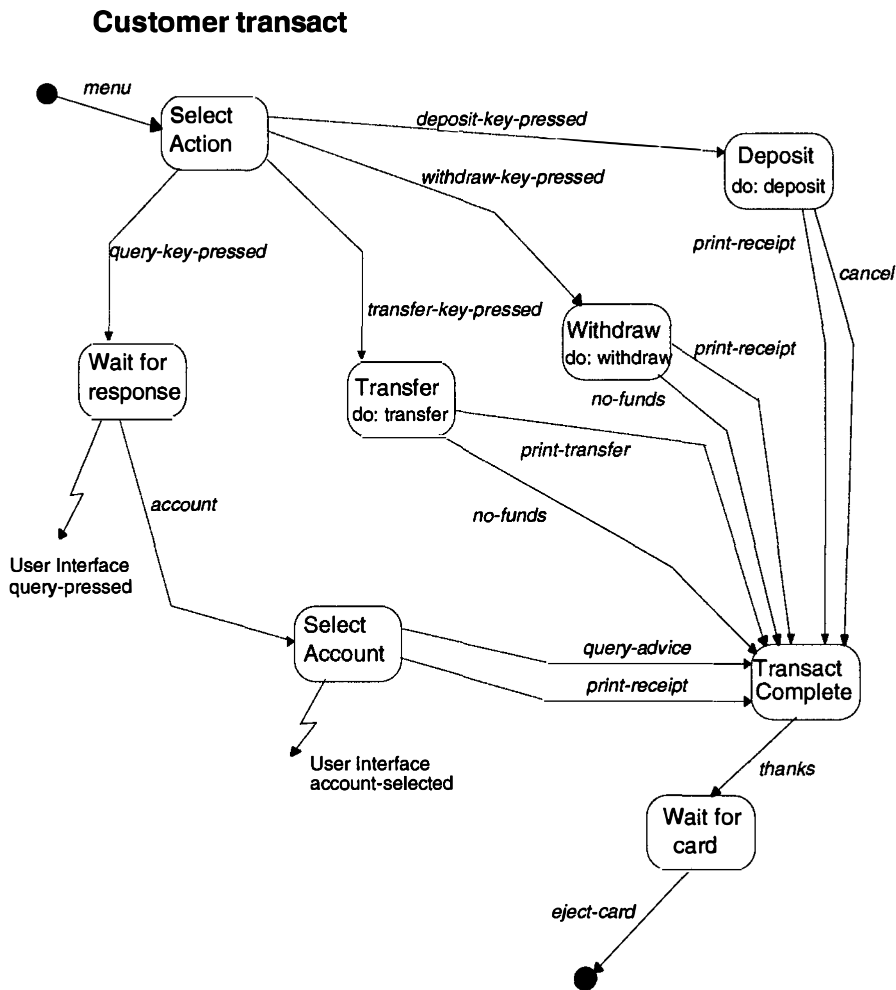

Figure 41. The transact model for Customer.

The 'deposit' activity from Figure 41 is modelled in Figure 42, showing that an account is selected in response to the *account* event, an amount is entered following the *amount* event and money is deposited following the *insert* event.

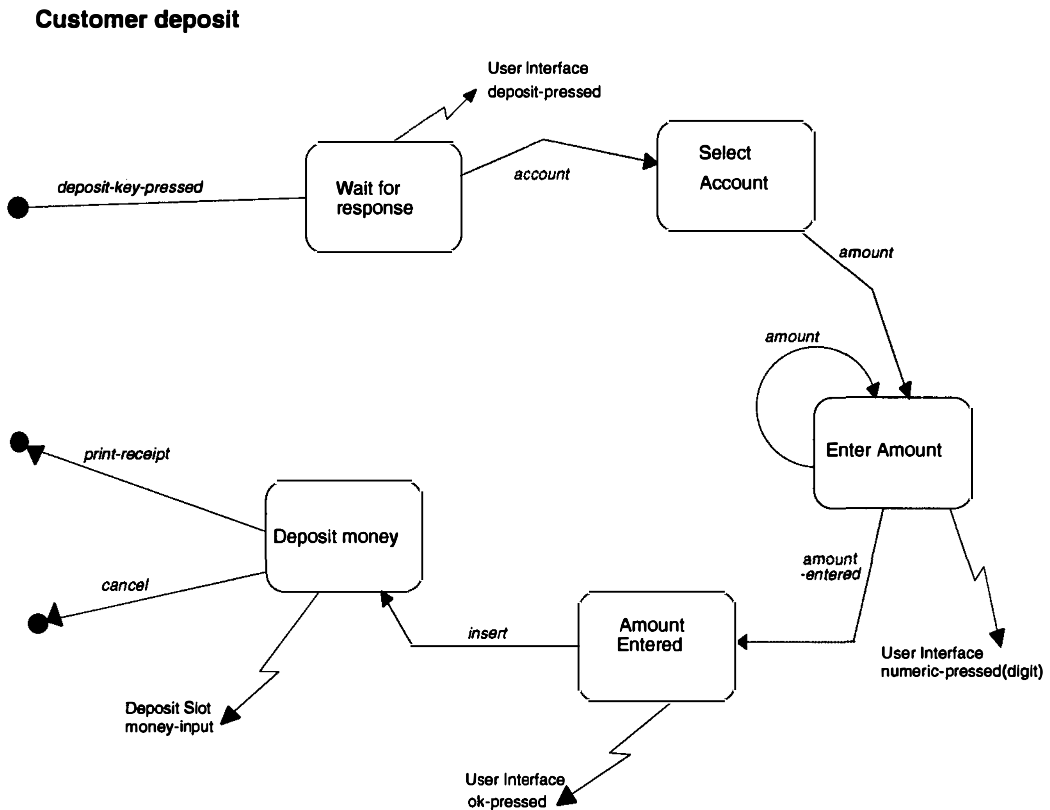

Figure 42. The deposit model for Customer.

As shown in Figure 43, the **'withdraw'** activity results in the *account* event changing the state to **Select Account**, the *amount* event changing the state to **Enter Amount** and a change of model state to **Retrieve Money** following the *output* event.

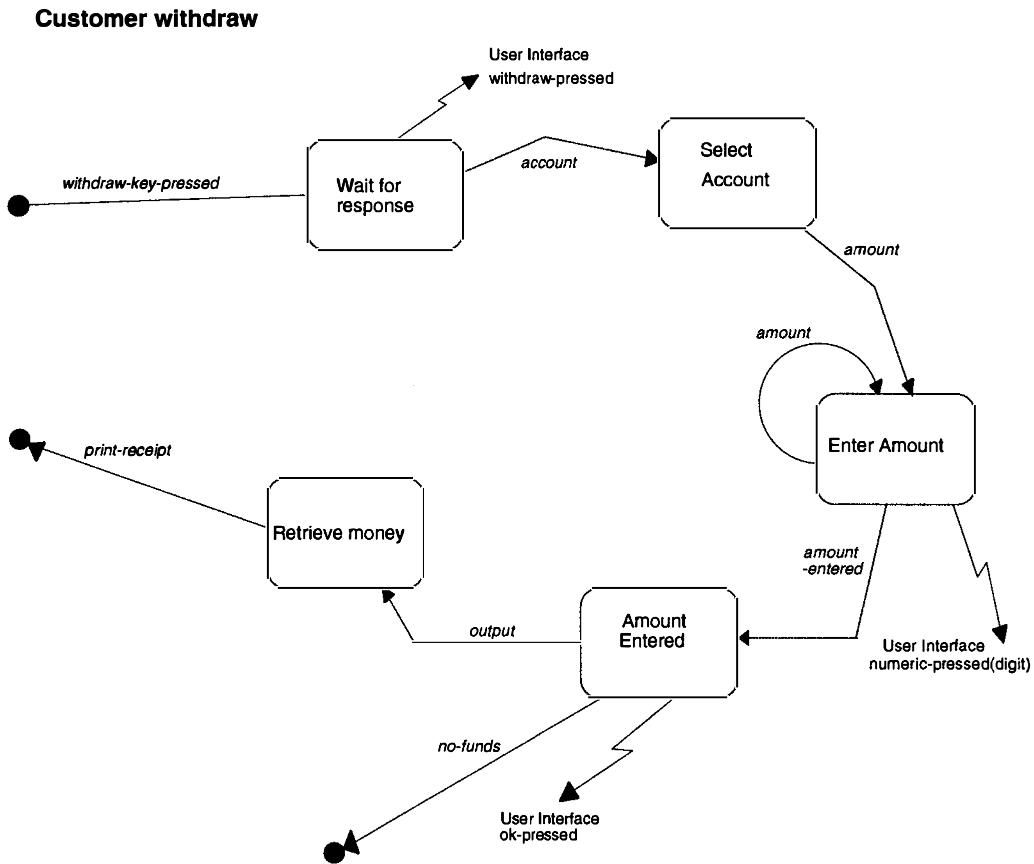

Figure 43. The withdraw model for Customer.

The 'transfer' activity for **Customer** is modelled in Figure 44, showing that a source and then a destination account is selected, then an amount is entered to complete the transaction.

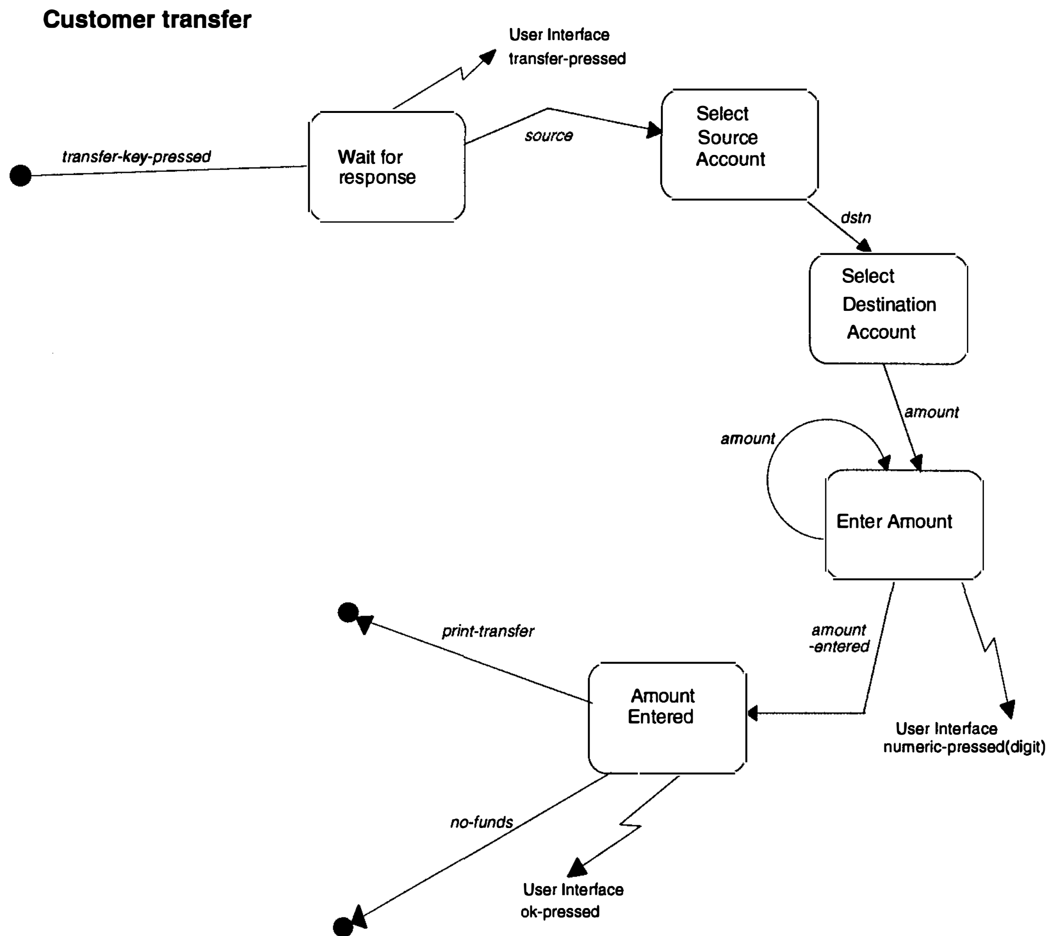

Figure 44. The transfer model for Customer.

A dictionary contains the information for activities and actions that are not described by a *Dynamic Model*, as shown in Figure 45.

Class	Activity / Action	Description
Card Reader	eject-card	eject card from ATM.
Card Reader	keep-card	place card into an internal store for retrieval by banking staff.
Card Reader	read-pin-id	read PIN and customer id from magnetic stripe on card.
Deposit Slot	money	start rollers; stop rollers when paper received.
Receipt Printer	printout	print "The amount of $" (amount) " has been " (transaction) " on your " (account) " account".
Receipt Printer	transfer	print "The amount of $" (amount) " has been transferred from your " (src-account) " account to your " (dstn-account) " account".
Receipt Printer	query	print "Your " (account) " account balance is $" (amount).
User Interface	account	show "Please select the account".
User Interface	amount	show "Enter amount to be deposited in whole dollars".
User Interface	asterisk	show "*".
User Interface	cancel	show "The transaction has been cancelled".
User Interface	compare-PIN	entered-PIN := PIN ifTrue: [is-PIN], ifFalse: [not-pin].
User Interface	dstn	show "Select the account for destination of funds".
User Interface	enter-PIN	show "Please enter your PIN and press OK".
User Interface	held	show "You must contact your branch to regain your card".
User Interface	insert	show "Place deposit in slot".
User Interface	invalid-PIN	if (times-validated) > 3, retry-allowed, otherwise show "The PIN entered is invalid - try again", increment (times-validated), PIN-rejected.
User Interface	menu	show "select action key to deposit, withdraw, transfer or query".
User Interface	no-funds	show "There are insufficient funds to complete the transaction".
User Interface	output	show "Take money from dispenser".
User Interface	source	show "Select the account for source of funds".
User Interface	thanks	show "Thank you for banking with us".
User Interface	unreadable	show "The card cannot be read by this ATM".
User Interface	welcome	show "Welcome to the ATM, please insert your card".

Figure 45. Dictionary of activities and actions for the ATM.

Determine the Associations

In determining the behaviour of objects, the associations between classes have been defined by the lightning strike symbol and these may be described within the *Object Model*, as shown in Figure 46.

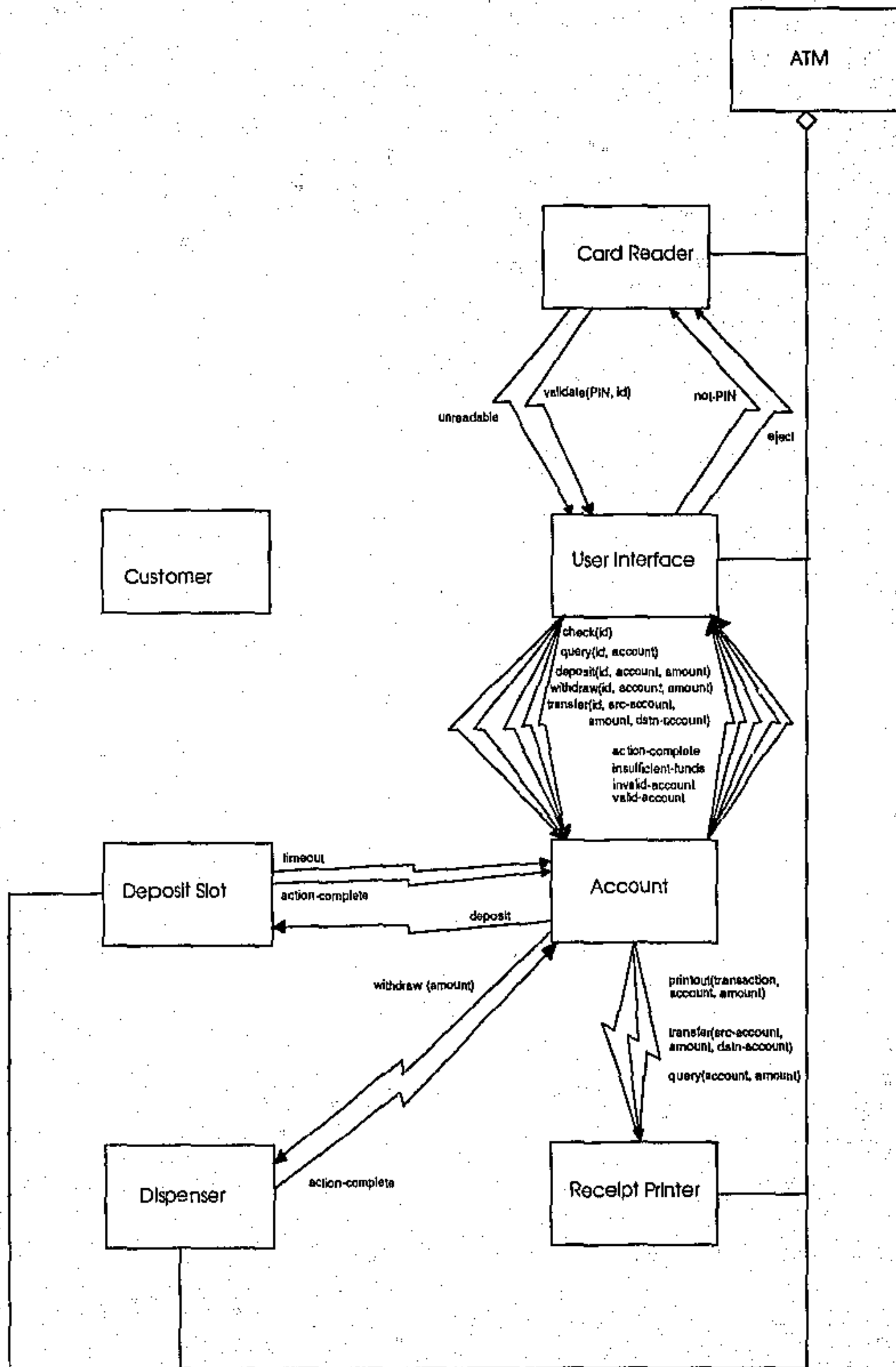

Figure 46. The ATM *Object Model* with associations.

For completeness, the behaviour initiated by the **Customer** which interacts with other classes is obtained from the **Customer** *Dynamic Model* and displayed within the *Object Model*, as shown in Figure 47.

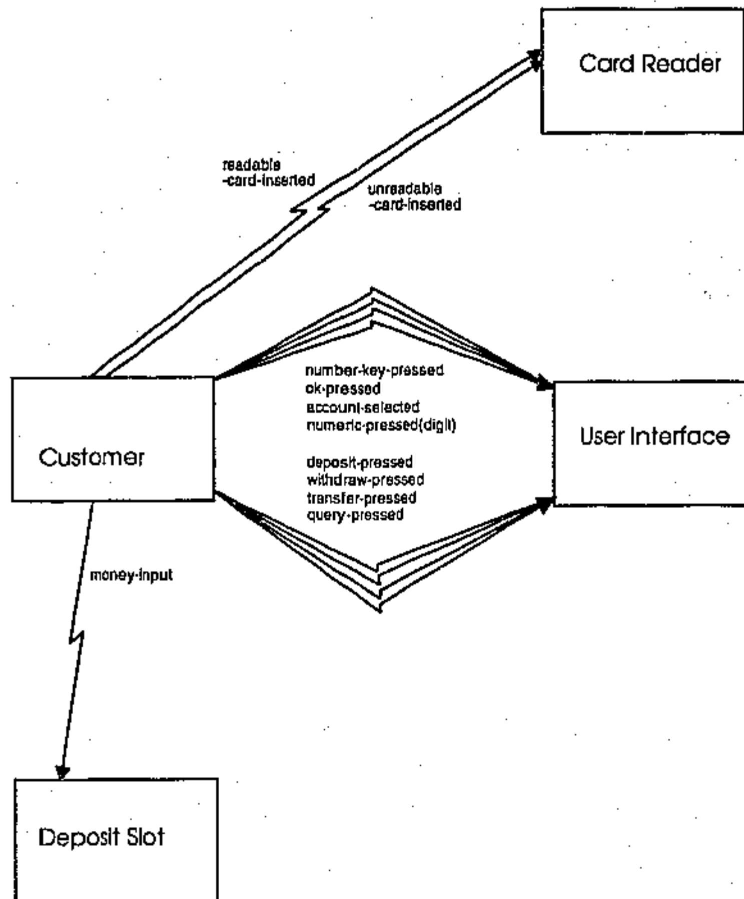

Figure 47. The *Object Model* with Customer interaction.

Detail the Attributes

Attributes have been determined within the *Dynamic Models*, represented within brackets, which are described in the dictionary as shown in Figure 48.

Class	Attribute	Description
Account	account	A choice of "cheque" or "savings".
Account	amount	The value of the transaction in whole dollars.
Account	dstn-account	The choice of "cheque" or "savings" that describes the account **to** which funds are transferred.
Account	id	The unique identification for a customer.
Account	src-account	The choice of "cheque" or "savings" that describes the account **from** which funds are transferred.
Dispenser	amount	The value of the transaction in whole dollars.
Receipt Printer	account	A choice of "cheque" or "savings".
Receipt Printer	amount	The value of the transaction in dollars and cents.
Receipt Printer	dstn-account	The choice of "cheque" or "savings" that describes the account **to** which funds are transferred.
Receipt Printer	src-account	The choice of "cheque" or "savings" that describes the account **from** which funds are transferred.
Receipt Printer	transaction	One of "deposit", "withdraw" or "query".
User Interface	account	A choice of "cheque" or "savings".
User Interface	amount	The value of the transaction in whole dollars.
User Interface	dstn-account	The choice of "cheque" or "savings" that describes the account **to** which funds are transferred.
User Interface	id	The unique identification for a customer.
User Interface	PIN	The four digit Personal Identification Number contained on the transaction card.
User Interface	src-account	The choice of "cheque" or "savings" that describes the account **from** which funds are transferred.
User Interface	times-validated	The number of times a PIN entry has been validated for a card insertion.

Figure 48. Dictionary of attributes for the ATM.

3.3 Summary

Apart from the inheritance links, the development of an ATM has been specified by employing the process steps and modelling approaches outlined in the previous chapter. This specification will be extended following the investigation of reuse issues and the construction of a search tool described in the next two chapters.

4 Achieving Reuse Of Software Components

Reuse is an important goal for the software industry because, as Booch (1987, p. 6) points out, "it simply makes sound engineering sense to apply the principles of reuse to the discipline of software development". As cited in Chapter 1, Meyer (1988, p. 27) says, "there should be catalogues of software modules . . . [so that] we would write less software, and perhaps do a better job at that which we do get to develop", explaining that "as early as 1968, . . . D. McIlroy was advocating 'mass produced software components'".

This chapter specifically addresses reuse, the barriers to achieving it and the means by which it may be applied in the object-oriented paradigm. In the first section, reuse is defined, its rationale is examined, its relationship with the object-oriented paradigm is described and barriers to its employment are outlined. Then, having identified the retrieval of information about components as the principal barrier to reuse, the second section examines approaches to alleviate this barrier. In some cases, the method of retrieval directly affects the way in which the information is stored. Furthermore, some methods have the requirement that searchers must follow those thought processes employed in the storage classification phase. This requirement raises a difficulty which is overcome in the approach of full and free text retrieval. The third section defines this approach and, based mainly on Salton (1989), examines the issues in efficiently implementing it.

4.1 The Reuse of Software Components

To understand reuse, Horowitz & Munson (cited in Booch, 1987, p. 7) explain that it "can come in many forms . . . [such as] prototyping, reusable code, reusable designs, application generators, formal specifications and transformation systems,

and off the shelf commercial packages". Meyer (1988, p. 31) points out that "once everything has been said, software is defined by code. A satisfactory policy for reusability must ultimately produce reusable programs". Dusink & Hall (1991, pp. 1 - 3) define reuse as:

> a means to support the construction of new programs using in a systematical way existing designs, design fragments, program texts, documentation or other forms of program representation.

These authors classify reuse either as transformational, in which "abstract specifications . . . . are transformed automatically into efficient target programs using transformation rules", or as compositional, in which "software components . . . are used as basic building blocks in the software construction process".

There are compelling reasons for achieving the goal of efficient reuse, as Matsumoto (cited in Prieto-Diaz & Jones, 1988, p. 152) points out that "in process control applications, Toshiba has reported productivity increases of 14 percent every year by applying a 'software factory' approach that emphasizes reusability". The opportunity exists to increase productivity because, according to Jones (cited in Prieto-Diaz & Jones, 1988, p. 152), "only [about] 15 percent of all software is unique and specific to a single application. The remaining 85 percent is common, generic and potentially reusable across applications". Booch (1987, p. xvii) asserts that "a carefully engineered collection of reusable software components can reduce the cost of software development, improve the quality of software products, and accelerate software production". Booch (1987, p. 6) adds that "just as with hardware components, we may develop classes of reusable software components that are functionally similar but that exhibit different time and space behaviour, and then we can use them to create more complex software systems".

In an earlier chapter, the inheritance mechanism of the object-oriented paradigm was explained showing that, instead of changing and adding to the existing source code, a new class inherits some of its behaviour from the existing classes. This is supported by Hooper & Chester (1991, p. 74) who say that "object-oriented design . . . promises increased software reuse" with the ability "to 'grow' software . . . through inheritance". However, Jette & Smith (cited in Hooper & Chester, 1991, p. 77) explain that "extensive use of inheritance and message passing will increase the need for good browsing tools to isolate which class/method is responsible for a problem".

Following an experiment in which object-oriented programmers were compared with a control group of structured programmers, Lewis, Henry, Kattura & Schulman (1992, 38 - 40) conclude that:

- The object-oriented paradigm does promote higher productivity than the procedural paradigm.
- When reuse is not a factor, the object-oriented paradigm does not promote higher productivity.
- The object-oriented paradigm does promote higher productivity than the procedural paradigm when reuse is employed.
- Given moderate encouragement to reuse, the object-oriented paradigm does promote higher productivity than the procedural paradigm.
- Given strong encouragement to reuse, the object-oriented paradigm does promote higher productivity than the procedural paradigm.
- The object-oriented paradigm demonstrates a particular affinity to the reuse process.

There are inherent overheads in achieving reuse of software components. One is the cost of reuse and Booch (1987, p. 6), pointing out that "if it costs more to find a

component and understand its behavior than it does to build it from scratch, then there will be little chance for reuse", outlines the following cost issues:

- It simply takes more effort to build a component that is generalized and hence appropriate for reuse than one that is not designed with reuse in mind.
- A component that is reused must be robust enough to suffer the abuse of a wide range of applications. Obscure, stressful applications will often uncover failure cases that would not be detected in general use, and so the component must be repaired if it is to continue to be reusable.
- Once a component is designated as reusable, there must be configuration management in place to track the component over its life.

Another barrier is the need to persuade programmers to use modules developed by another person because, as Meyer (1988, p. 28) points out, "the 'Not Invented Here' complex is well known" and therefore "the psychological difficulties should not be underestimated", even though "the main roadblocks are technical".

The ability to locate the desired software module is perhaps the most important barrier and Booch (1987, p. 6) states the need for "a mechanism with which to efficiently retrieve an individual component". More emphatically, Frakes & Nejmeh (1988, pp. 142 - 143) stress that "a fundamental problem in software reuse is the lack of tools to locate potential code for reuse" and that "if potentially reusable software components cannot be located, retrieved and reviewed effectively, reuse is neither feasible nor valuable". Meyer (1988, p. 28) is cited in Chapter 1, claiming that "the best reusable components in the world are useless if nobody knows they exist, [or] if it takes a long time to obtain them". Explaining the steps to reuse, Dusink and Hall (1991, p. 4) say that "some form of software component library must be available, for the storage of components (and their descriptions) and to allow forms of browsing and querying".

The next section addresses the last of these barriers - the approaches for storing and retrieving software components.

4.2 Storing and Retrieving Reuse Components

The traditional approach has been to store information using a classification scheme, which is a means of obtaining order based on a controlled and structured index vocabulary. Prieto-Diaz & Jones (1988, p. 155) explain that classification consists of "names or symbols representing concepts or classes, listed in systematic order, to display relationships between classes".

According to Prieto-Diaz & Jones (1988, p. 155), "two types of classification schemes are used in library science - enumerative and faceted", of which "the traditional, enumerative method postulates a universe of knowledge divided into successively narrower classes". An example is the Dewey Decimal system, used to locate the shelf position of books within a library, where all possible classes are predefined and listed in the classification schedule. A problem arises when a chosen classification may not be obvious to all searchers, because a librarian could place the title *Structured Systems Programming* in any of the categories shown in Figure 49 (drawn from Prieto-Diaz & Jones, 1988, p. 156).

001.61	system analysis
001.6425	software
003	systems
620.72	systems analysis
620.73	systems construction

Figure 49 - Dewey classification.

Of faceted classification, Prieto-Diaz & Jones (1988, p. 155 - 157), say that it "is more straightforward . . . . [because] to classify a title, a term is selected from each

facet to best describe the concepts in the title" so that it "relies not on the breakdown of the universe, but on the building up or synthesizing from the subject of particular documents". Each component is matched into elemental classes, which are grouped into facets, considered by Prieto-Diaz & Jones (1988, p. 155) to be "the perspectives, viewpoints or dimensions of a particular domain".

The above methods, however, rely on a common understanding between the person selecting the storage criteria and the people retrieving the information. Additionally, for these methods, Huang (cited in Frakes & Nejmeh, 1988, p. 143) concludes that "no methodology for large scale software development provides a reliable storage and retrieval mechanism for a code-level library". Frakes & Nejmeh (1988, p. 144) also point out that these methods "are usually limited in their ability to handle data that is not highly structured, such as text or source code."

Mortimer Taube (cited in Cortez & Kazlauskas, 1986, pp. 56 - 58) developed a concept of using some of the actual words contained in a document as the search keys in order to cope with "the massive growth of scientific and technical information, and the need to store and retrieve this information rapidly". This full and free text retrieval concept is supported by Frakes & Nejmeh (1988, p. 144), who explain that because systems employing this approach "are capable of handling unstructured data, they can be used to store and retrieve products . . . such as . . . design documents [and] code". Support is also provided by Gibbs, Tsichritzis, Casais, Nierstrasz and Pintado (1990, 93) who say that one way "of representing an object class so that the information needed to use the class can be easily located and incorporated within an application . . . . would be to represent classes by source text". According to Horton (1990, p. 59):

> the main advantage of full text . . . is the ability to identify precise words and phrases and subtleties of meaning in the original context,

without the filter of an abstract, which can omit important information or terms.

The concept is also supported by Freeman & Henderson-Sellers (1991, p. 175), who say that "a method of cataloguing and classifying Object Classes utilising Full text storage . . . . may obviate the high intellectual requirement required to establish controlled vocabulary entries".

To apply the full and free text process, Salton (1989, p. 232) explains that, for each word, "a separate index is constructed that stores the record identifiers, or record addresses, for all records identified by that term". Using words related to the search, the documents which contain them may lead to the desired information, or, as Kimmel (1990, p. 106) explains, "often the documents themselves . . . act only as pointers to the information source". The approach is detailed in the next section.

4.3 Full and Free Text Retrieval

Although the concept of indexing every word for storage appears onerous, the text retrieval process may be very fast because many words are used frequently. Salton (1989, pp. 105 - 108) points out that "six words (the, of, and, to, a and in) account for over 20 percent of all word occurrences in English" while "the 50 most frequent words cover more than half of all word occurrences in ordinary text". As a result, beyond a moderate size, the number of words to be indexed grows at a lower rate than the size of the text, as shown in Figure 50 (drawn from Salton, 1989, p. 108).

Number of word occurrences	Average number of separate words	Average increase in number of words
500	223.6	-
1000	316.2	92.6
2000	447.2	131.0
4000	632.5	185.3
6000	774.6	142.1
8000	894.4	119.8
10000	1000	105.6
12000	1095.4	95.4

Figure 50. Growth of new words with an increase in the text size.

The method, explained by Salton (1989, pp. 232 - 233) is to store words - or terms - in a file "known as an inverted index or inverted file" as follows:

- The complete file is first represented as an array of indexed records, where each row represents a record, or document, and each column specifies the assignment of a particular term.
- The record-term array is inverted (actually transposed) in such a way that each row of the inverted array then specifies the records corresponding to some particular term.
- The rows of the inverted term-record array are manipulated in accordance with a particular query specification to determine the set of records that respond to the query.

For example, the document positions stored in the inverted file construction for the IBM STAIRS (Storage and Information Retrieval System) software link each term to a set of addresses, each one of which denotes the paragraph - sentence - word position of an occurrence of the term. As an illustration, if the second sentence in the third paragraph is 'The quick brown fox jumps over the lazy dog', then the address for each word is specified by the number of paragraphs from the start of the document, the number of sentences from the beginning of the paragraph and the

number of words from the beginning of the sentence, as shown in Figure 51. Note that word storage is not case sensitive.

Term	Para	Sentence	Word	Para	Sentence	Word
brown	3	2	3			
dog	3	2	9			
fox	3	2	4			
jumps	3	2	5			
lazy	3	2	8			
over	3	2	6			
quick	3	2	2			
the	3	2	1	3	2	7

Figure 51. An inverted list

Following the storage of the document in the manner described above, queries may be formulated by retrieving the address of search words. Salton (1989, p. 232) explains that the query may be enhanced by joining search words with boolean operators, of which the following are used:

- The *or* operator treats two terms as effectively synonymous. In particular; given the query (term 1 or term 2), the presence of either term in a record suffices to retrieve that record.
- The *and* operator combines terms into term phrases; thus the query (term 1 and term 2) indicates that both terms . . . must be present for retrieval.
- The *not* operator is a restriction, or term-narrowing, operator that is normally used in conjunction with the *and* operator to restrict the applicability of particular terms; thus the query (term 1 and not term 2) leads to the retrieval of all records containing term 1, provided that term 2 is not also present in the words.

According to Salton (1989, pp. 232 - 236), the inverted index free and full text method is faster than "sequential searches of the record file, as well as access methods based on pointer-chain tracing as in the multilist method" and that it "exhibits substantial advantages in terms of processing efficiency". The method does have disadvantages and Salton (1989, p. 236) explains that:

- the records are normally retrieved in the order in which they appear in the inverted lists;
- a large output may overwhelm the user; and on the other hand,
- narrowly formulated queries using *and* operators may generate very little output.

Additionally Salton (1989, p. 246) describes "various methods designed to reduce the size of the index term set, and hence the inverted index itself", such as:

- the use of truncated terms instead of full word forms;
- the implementation of hash-table transformations to reduce variable-length word forms to short fixed-length codes; and
- replacement of full term entries with word fragments.

However, Salton (1989, p. 246) warns that these methods have "the disadvantage . . . of a loss of subject discrimination, possibly leading to reduced retrieval effectiveness, because the short forms of the terms do not always specify topics precisely". Another method described by Salton (1989, p. 279) is to "eliminate common function words from the document texts by consulting a special dictionary, or stop list, containing a list of high frequency function words", taking advantage of the knowledge that "most function words are characterized by high occurrence frequencies in ordinary texts". A further capability, according to Salton (1989, pp. 299 - 301), is the use of a thesaurus which "takes low-frequency, overly specific terms and replaces them with less-specific, medium-frequency thesaurus 'heads'" which "broadens index terms whose scope is too narrow to be useful in retrieval".

The author explains, however, that "thesauruses valid for subject areas of reasonable scope are constructed manually, or intellectually by committees of experts".

4.4 Summary

The chapter has examined the issues of reuse, including those that apply to the object-oriented paradigm, concluding that an important barrier to reuse is the lack of a capability to retrieve relevant information about software modules. The full and free text retrieval method overcomes the difficulty that the searcher has in understanding the thought processes used in the storage of information and, in the next chapter, a search tool is constructed to employ pertinent parts of this method.

5 The Search Tool

The previous chapter establishes the requirement for an objected-oriented search tool to indicate potential classes based upon search criteria, then it espouses the technique of full and free text retrieval as a beneficial search mechanism.

This chapter details both the construction of the tool and its usage. More specifically, the first section develops the requirement, describes the analysis and design and outlines the test procedures. Then, the second section provides details on the environment required to implement the tool, on operation of the tool, on entering the search criteria and on evaluating the result.

5.1 Development of the Search Tool

Complying with the process described in Chapter 2, the analysis and design of the search tool is undertaken within this section by an understanding of the needs of a software constructor together with the manner in which these needs may be fulfilled, an identification of the objects involved, their responsibilities, their associations, their attributes and a determination of the inheritance links that enhance the implementation.

Understand the Problem

An application developer within the object-oriented paradigm, wishing to implement a desired functionality, can articulate a natural language specification of a desired software class but may not understand the capability of all classes in the library. To relieve this lack of understanding, it has been shown in previous discussion that a search tool is needed to find suitable candidate classes by initiating a search based on the natural language specification. Thus, words that express the main theme of a

natural language specification may be input to the tool, whose resultant output lists potentially suitable classes. The identification of classes by the tool should take place quickly, even if preparation for the search requires a significant amount of time.

The search may produce a list of many or few classes, in which cases the application developer may need to refine the search. Additionally, the application developer will examine in detail the service methods of identified classes. As a result of the detailed examination, the search may be extended to further classes that associate with the class under inspection. If an appropriate class is available, then ultimately it will be found.

Identify the Objects

From the above problem identification, a **Class Find** search tool is developed. The tool requires access to service method text code for all classes within the library. This access is provided indirectly to **Class Find** by Smalltalk's **Behavior** object, which is able to read every service method within all classes.

The **Class Find** tool will establish a **Dictionary** object to store the information in a manner designed for full and free text retrieval. Each word of the service method text code, except stop words as identified by Salton (1989), will form **Dictionary** keys, where for each key the **Dictionary** value will be a set of all classes that contain that key word. A **Window** object will be established by the **Class Find** object for the output list developed. For further investigation, a Smalltalk **Class Browser** object may be established for a detailed examination of a class.

All of the objects thus identified - **Class Find, Dictionary, Window, Class Browser** and **Behavior,** together with **Behavior's** relationships with all classes - are shown in Figure 52.

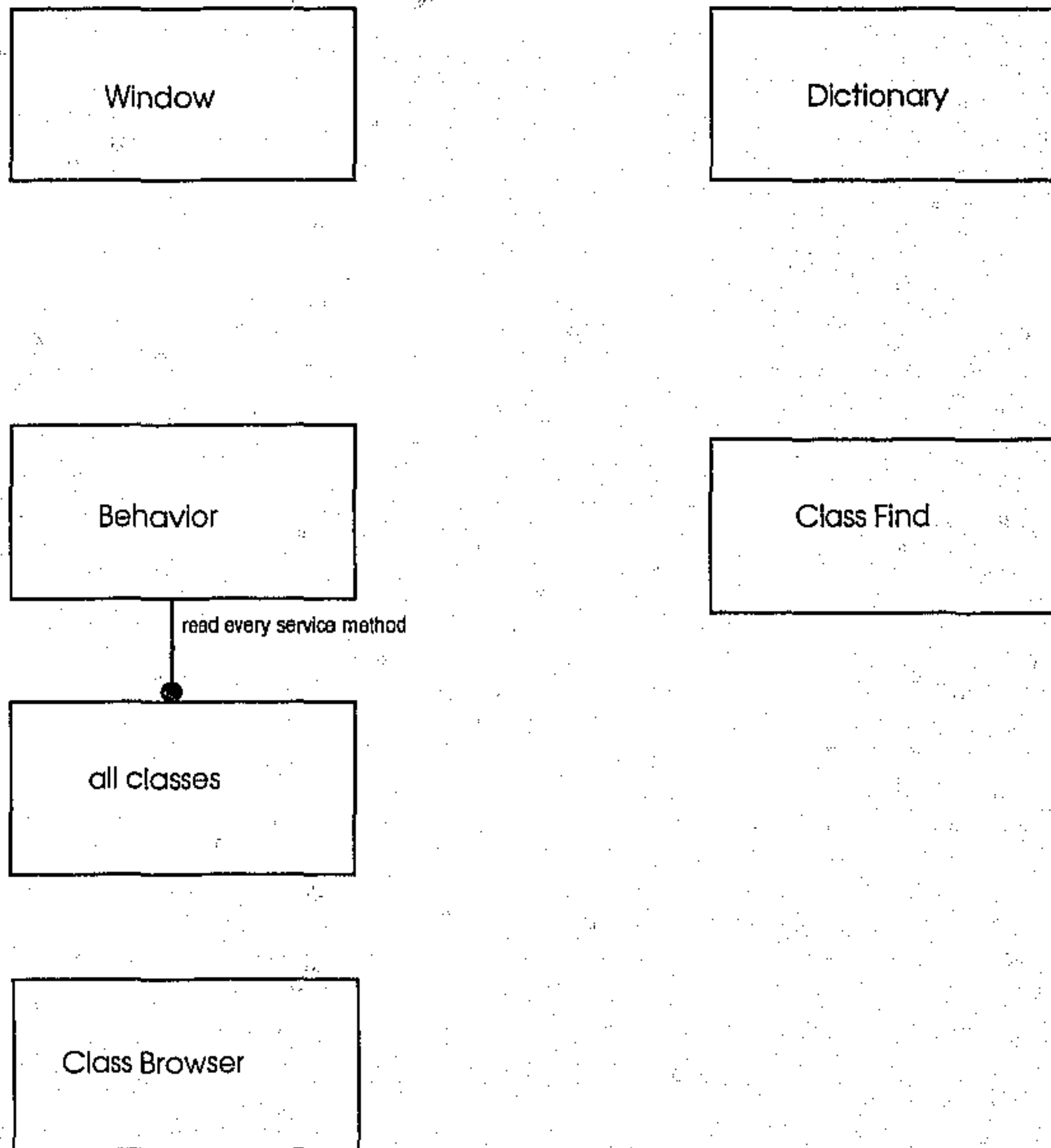

Figure 52. The initial *Object Model* for Class Find.

Determine the Responsibilities

The behaviour of the **Class Find** object is described in Figure 53 which shows that, from the **Closed** state, construction of the search tool is commenced by invoking the 'initialise' activity to build the dictionary. Here, the keys are the service method text words - except stop words - and the values are sets of classes that contain the words. Then, the 'open' action invokes the formation of a window for the search tool. When the *window-is-open* event occurs, the **Class Find** tool is in the **Idle** state. The event *class-menu*, resulting from making a menu choice, changes the state to **Operate**, which is described in detail below. The *edit-message-sent* event returns the state to **Idle**. From either the **Operate** or **Idle** states, a *close* event returns the **Class Find** tool to the **Closed** state.

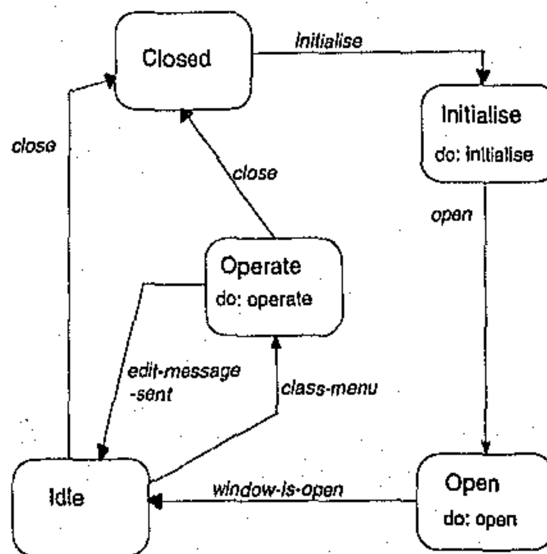

Figure 53. The *Dynamic Model* for Class Find.

Detail of the 'operate' activity is shown in Figure 54. The event *class-menu* results in the following choices:

- "clear and enter", initiating the 'clear' activity which clears previous output from the window and prompts for the search terms, followed by initiation of the 'get-criteria' activity;
- "enter more criteria", initiating the 'get-criteria' activity which prompts for the search terms; or,
- "return to last entry", initiating the 'default-criteria' activity which presents the previous search terms for possible modification.

Then, the *locate-classes* event results in a display of each class and the search criteria met. A *select* event, resulting from the use of a mouse to select a class, changes the state to **Browse**, while a *close* event may occur as described previously. Within the **Browse** state, a message is sent to the identified class to open a **Class Browser** object, resulting in the *edit-message-sent* event.

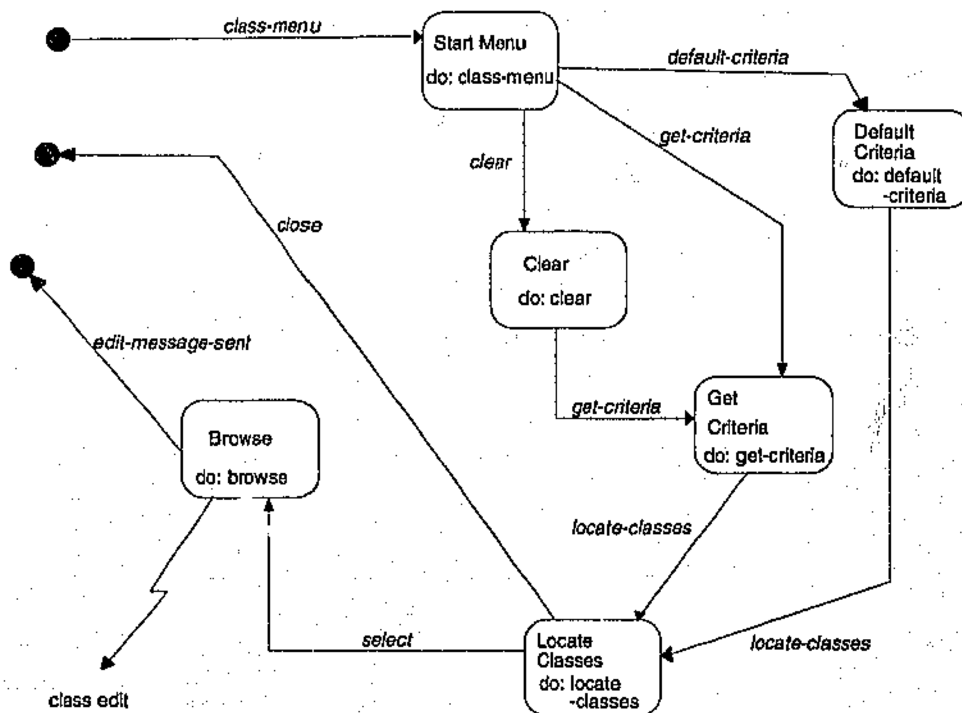

Figure 54. The operate model for Class Find.

The search tool is built as a result of the *initialise* event, as shown in Figure 55. From the **Get Code** state, a message sent to the **Behavior** object results in the *each-method* event and **Class Find** changes to the **Stream** state, at which time non-alphabetic characters are removed from the (code-stream) attribute. As it is normal Smalltalk practice to define identifiers as compound words, each of which may commence with a capital letter, these are separated into individual words, then all characters are changed to lower case. **Class Find** returns to the **Get Code** state as a result of the *more-code* event or changes to the **Add** state as a result of the *all-code* event, at which point the (code-stream) attribute is added to the **Dictionary** object and the *open* event occurs.

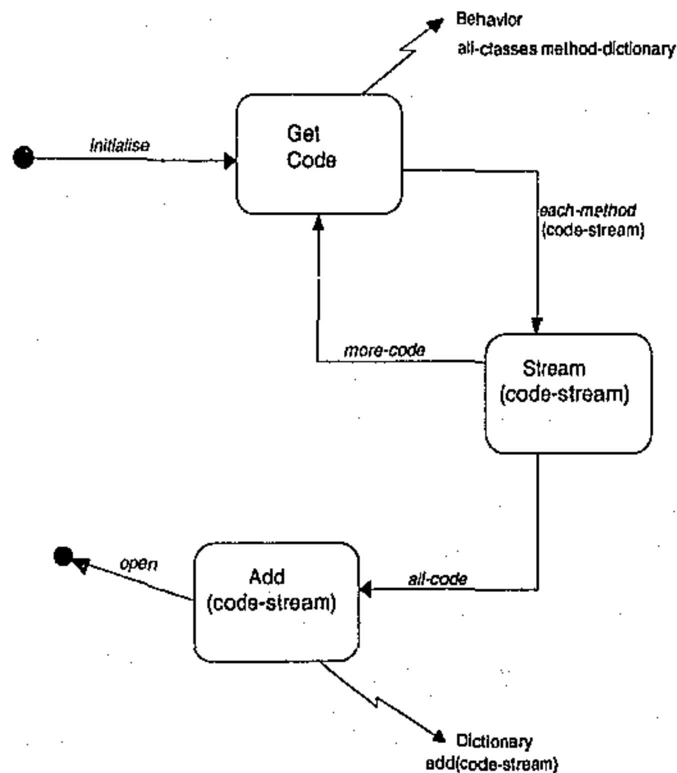

Figure 55. The initialise model for Class Find.

The 'locate-classes' activity, resulting in a display of each class that meets the search criteria, is shown in Figure 56. Within the **Lookup** state, for every search word other than *or*, *and* or *not*, each class containing that word is found and added to the (class-output) attribute. The *lookup-completed* event changes the state to **Transfer Output**, where the (class-output) attribute is prepared for listing, then the *output-transferred* event changes the state to **Output**, sending the (class-output) attribute to the **Window**. If the *more-criteria* event occurs, the actions are iterated, otherwise either the *select* event occurs because a class is selected or a *close* event occurs.

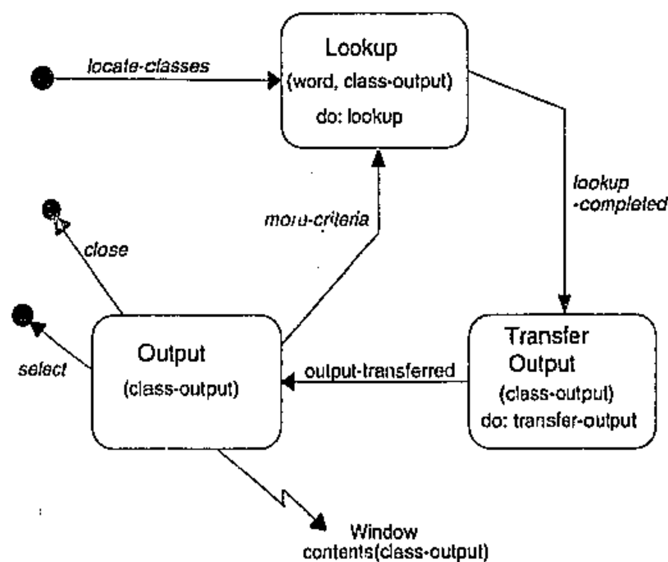

Figure 56. The locate-classes model for Class Find.

The 'lookup' activity, illustrated in Figure 57, is initiated by the *locate-classes* or *more-criteria* events. From the **Find** state, a message is sent to the **Dictionary** object to locate the (word) attribute, resulting in the *find-completed* event and a change of state to **Not**. If the (word) attribute has been negated in the search term, then the (class-list) attribute is complemented and the *complement-completed* event changes the state to **And**. If the (word) attribute is part of a boolean expression linked by an *and* operation, a union is undertaken between the current and previous (class-list) attributes and the *union-completed* event changes the state to **Add**. The (class-list) attribute is added to the output and the *lookup-completed* event occurs.

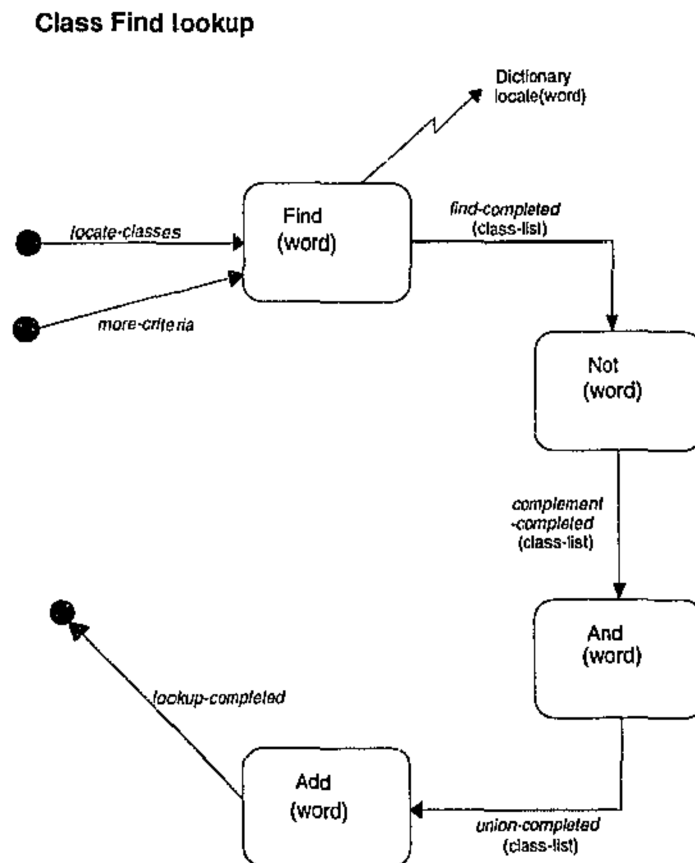

Figure 57. The lookup model for Class Find.

The 'transfer-output' activity, initiated by the *lookup-completed* event, is shown in Figure 58. From the **Notify** state, a user display is prepared indicating the number of responses to the search. When the *size-transferred* event occurs, the state is

changed to **Output Candidate**, in which the (class) attribute is prepared for output. When the *class-transferred* event occur. ... (search-term) attribute is output and the *output-transferred* event occurs.

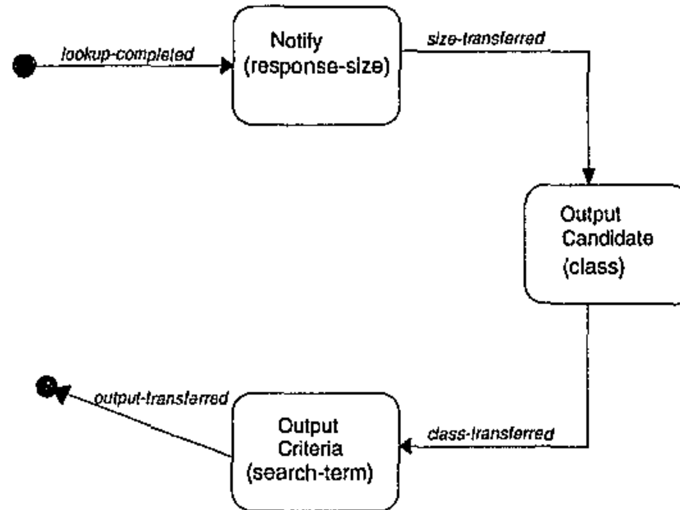

Figure 58. The transfer-output model for Class Find.

Figure 59 shows a dictionary for activities and actions that are not described by a *Dynamic Model.*

Class	Activity / Action	Description
Class Find	browse	Open Class Browser on selection.
Class Find	class-list	Show the accumulated output from the search.
Class Find	class-menu	Show "clear and enter" - select for *clear*, show "enter more criteria", select for *get-criteria*; show "return to last entry", select for *default-criteria*.
Class Find	clear	Initialise (class-output), then induce *get-criteria*.
Class Find	default-criteria	Show previous entry in prompting for search input, then induce *locate-classes*.
Class Find	get-criteria	Prompt for search input, then induce *locate-classes*.
Class Find	open	Open list window. For menu - induce *locate-classes*; for change - induce *class-list*; for selection - induce *browse*.

Figure 59. Dictionary of activities and actions for Class Find.

Determine the Associations

From the behaviour explained above, the associations between classes are defined by the *Object Model*, as shown in Figure 60.

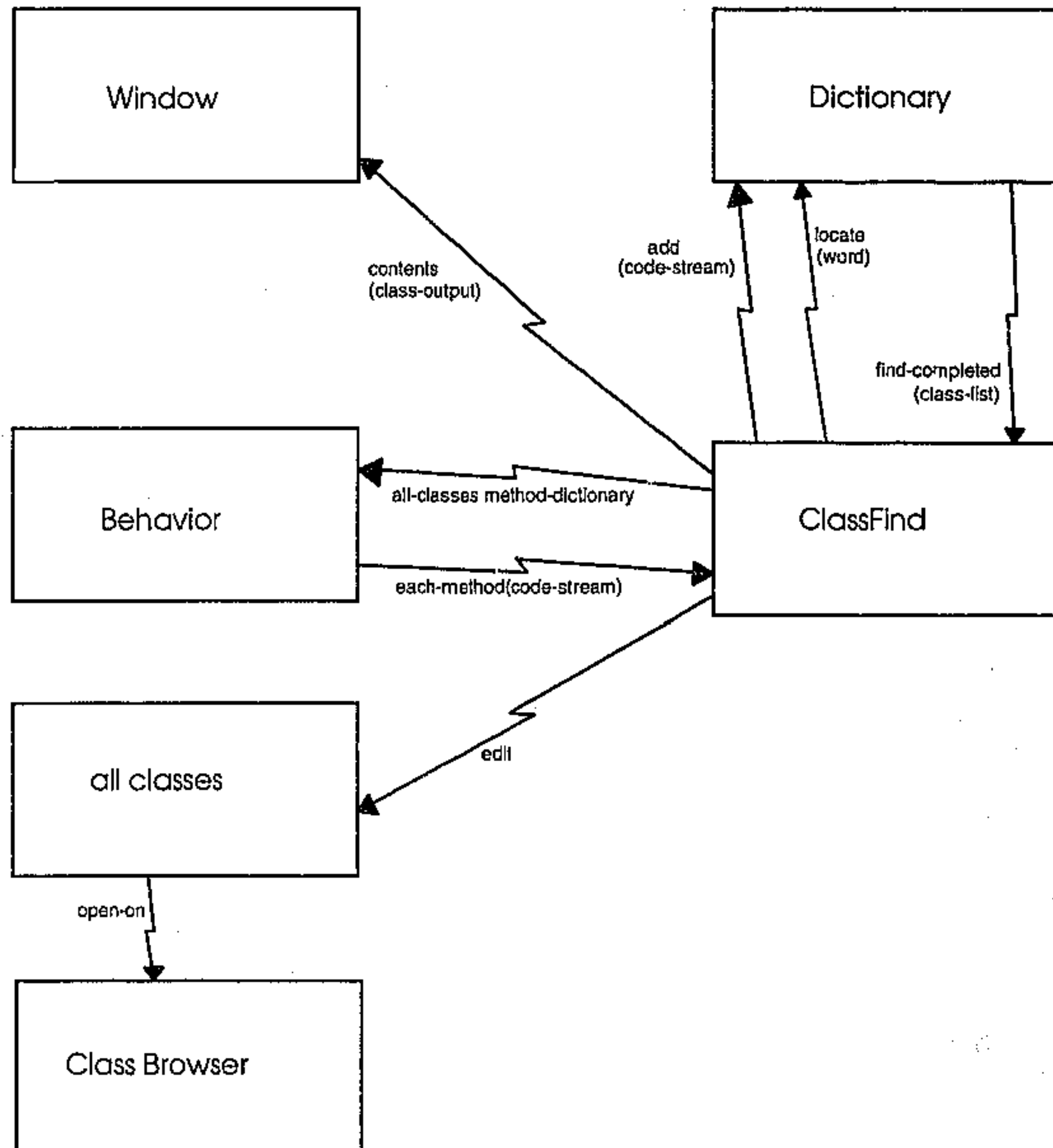

Figure 60. Associations for Class Find.

Detail the Attributes

From the *Dynamic Models*, attributes are identified and described in a dictionary, as shown in Figure 61.

Class	Attribute	Description
Class Find	class	The name of the class that contains the search term.
Class Find	class-list	A list of classes that contain the search term.
Class Find	code-stream	All text for each service method.
Class Find	class-output	The search output, consisting of the number of responses, each class matching the search term and the relevant search term.
Class Find	response-size	The number of classes matching the search term.
Class Find	search-term	The words and boolean expressions on which the search is based.
Class Find	word	Each word input to the search.
Dictionary	code-stream	A stream of words, each of which is to be stored.
Dictionary	word	The key word to be located.

Figure 61. Dictionary of attributes for Class Find.

Build the Inheritance Links

The desired behaviour to open and manage a **Window** is provided by inheritance from the **View Manager** class. Available from the Smalltalk/V vendor, additional to the standard class library, is the **Word Index** class, which incorporates the necessary behaviour to store and retrieve text words in a **Dictionary** using full and free text principles. Inheriting from **View Manager** and **Word Index**, as shown in Figure 62, relieves the development requirement for detailed interaction with the **Window** and **Dictionary** classes.

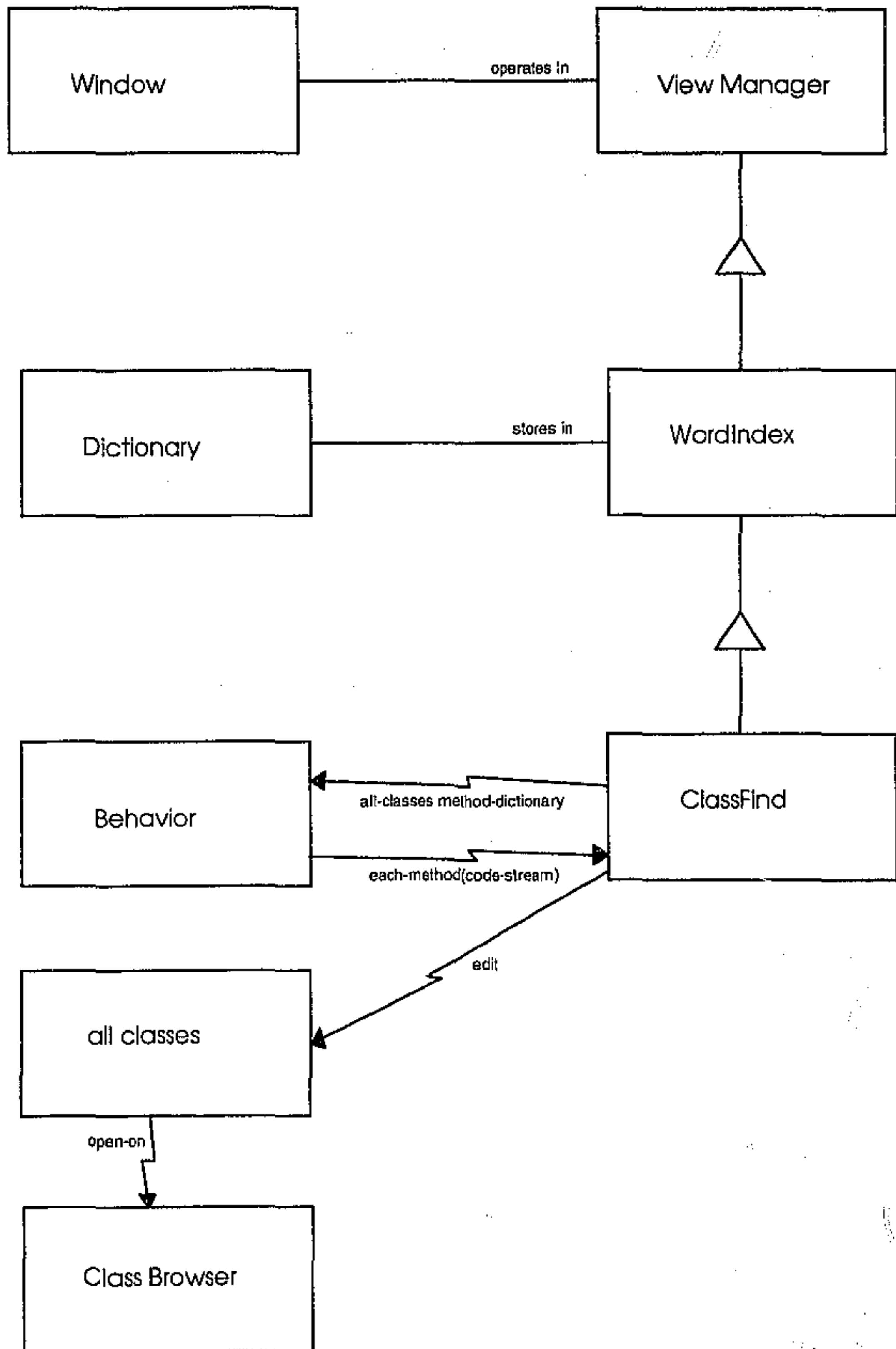

Figure 62. The final *Object Model* for Class Find.

Testing the Search Tool

Test procedures are developed to ensure that the search tool accurately fulfils the requirement described above. As Page-Jones (1988, p. 268) points out, these procedures "should contain test cases comprising test data deliberately and fiendishly crafted to expose as many defects as possible, together with the predicted output for each test input". The test procedures cover:

- accuracy, ensuring that each and every class that contains the search word is exhibited; and
- functionality, affirming predicted behaviour for normal input and expected behaviour for erroneous input.

For a credible test plan, modification for the test process is minimised and the final code is completely tested.

Accuracy and normal functionality are tested by searching for occurrences of words within an environment for a known result: namely, none, one and multiple; negation of none, one and multiple; intersection; union; and combinations of negation, intersection and union. The known environment consists of four new classes, as follows:

- A class named **Class0**, with no methods;
- A class named **Class1**, inherited from class **Class0**, with a method named 'methoda' containing the words **wordinone** and **wordinonetwo** and another method named 'methodb' containing the words **wordinonetwothree** and **wordinonethree**;
- A class named **Class2**, inherited from class **Class1**, with a method named 'methodc' containing the words **wordinonetwo, wordinonetwothree, wordintwo** and **wordintwothree**; and
- A class named **Class3**, inherited from class **Class2**, with a method named 'methodd' containing the words **wordinonetwothree, wordinonethree** and

wordintwothree, a method named 'methode' containing the word **wordinthree** and a method named 'methodf' containing no additional words.

Therefore, a test environment is created covering all possibilities where classes have none, one and multiple methods, where methods contain none, one and multiple words and where the occurrence of words is as follows:

- **wordinzero** is not present in any class;
- **wordinone** occurs once, in **Class1**;
- **wordintwo** occurs once, in **Class2**;
- **wordinthree** occurs once, in **Class3**;
- **wordinonetwo** occurs twice, in **Class1** and **Class2**;
- **wordinonetwothree** occurs three times, in **Class1**, **Class2** and **Class3**;
- **wordinonethree** occurs twice, in **Class1** and **Class3**; and
- **wordintwothree** occurs twice, in **Class2** and **Class3**.

With minimal modification, the test environment is limited to **Class0**, **Class1**, **Class2** and **Class3**, for which the Venn diagram is shown in Figure 63.

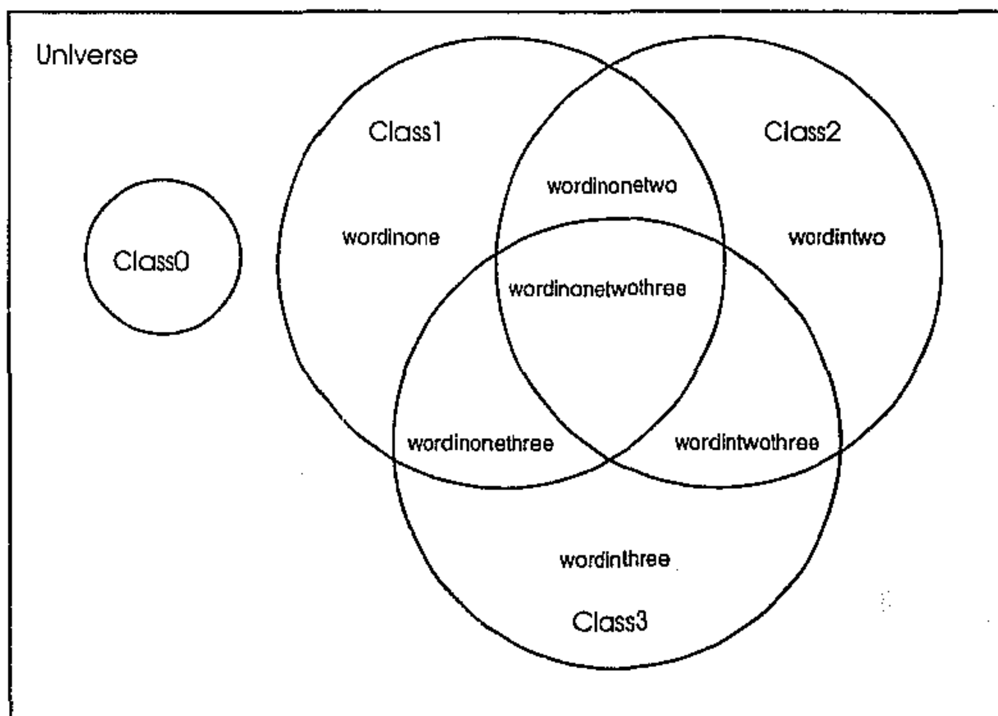

Figure 63. A Venn diagram for the test environment.

To demonstrate satisfactory performance of the search tool, the following plan tests each region of the above Venn diagram and each logical combination.

Test	Input	Expected Output
1.	**wordinzero**	Number of responses is zero.
2.	**wordinone**	Number of responses is one, showing class **Class1**.
3.	**wordintwo**	Number of responses is one, showing class **Class2**.
4.	**wordinthree**	Number of responses is one, showing class **Class3**.
5.	**wordinonetwo**	Number of responses is two, showing classes **Class1** and **Class2**.
6.	**wordinonetwothree**	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.
7.	**wordinonethree**	Number of responses is two, showing classes **Class1** and **Class3**.
8.	**wordintwothree**	Number of responses is two, showing classes **Class2** and **Class3**.
9.	**not wordinzero**	Number of responses is four, showing classes **Class0**, **Class1, Class2** and **Class3**.
10.	**not wordinone**	Number of responses is three, showing classes **Class0, Class2** and **Class3**.

Test	Input	Expected Output
11.	**not wordintwo**	Number of responses is three, showing classes **Class0, Class1** and **Class3.**
12.	**not wordinthree**	Number of responses is three, showing classes **Class0, Class1** and **Class2.**
13.	**not wordinonetwo**	Number of responses is two, showing classes **Class0** and **Class3.**
14.	**not wordinonetwothree**	Number of responses is one, showing class **Class0.**
15.	**not wordinonethree**	Number of responses is two, showing classes **Class0** and **Class2.**
16.	**not wordintwothree**	Number of responses is two, showing classes **Class0** and **Class1.**
17.	**wordinone or wordinonetwo**	Number of responses is two, showing classes **Class1** and **Class2.**
18.	**wordinone or wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3.**
19.	**wordinzero or wordinone or wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3.**
20.	**wordinone and wordinonetwo**	Number of responses is one, showing class **Class1.**
21.	**wordinone and wordinonetwo and wordinonetwothree**	Number of responses is one, showing class **Class1.**

Test	Input	Expected Output
22.	**wordinzero and wordinone and wordinonetwo and wordinonetwothree**	Number of responses is zero.
23.	**not wordinone or not wordinonetwo**	Number of responses is three, showing classes **Class0, Class2** and **Class3.**
24.	**not wordinone and wordinonetwo**	Number of responses is one, showing class **Class2.**
25.	**not wordinone and not wordinonetwo**	Number of responses is two, showing classes **Class0** and **Class3.**
26.	**not wordinone and not wordinonetwo and not wordinonetwothree**	Number of responses is one, showing class **Class0.**
27.	**wordinone and wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3.**
28.	**not wordinone and wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3.**
29.	**not wordinone and not wordinonetwo or wordinonetwothree**	Number of responses is four, showing classes **Classs0, Class1, Class2** and **Class3.**
30.	**not wordinone and not wordinonetwo or not wordinonetwothree**	Number of responses is two, showing classes **Class0** and **Class3.**

Then, after reverting to the original code - unmodified for testing - the tests are repeated using the full environment. Using an approach independent to that used by the search tool, the total number of classes - denoted by N - is derived.

The test processes are:

Test	Input	Expected Output
31.	**wordinzero**	Number of responses is zero.
32.	**wordinone**	Number of responses is one, showing class **Class1**.
33.	**wordintwo**	Number of responses is one, showing class **Class2**.
34.	**wordinthree**	Number of responses is one, showing class **Class3**.
35.	**wordinonetwo**	Number of responses is two, showing classes **Class1** and **Class2**.
36.	**wordinonetwothree**	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.
37.	**wordinonethree**	Number of responses is two, showing classes **Class1** and **Class3**.
38.	**wordintwothree**	Number of responses is two, showing classes **Class2** and **Class3**.
39.	**not wordinzero**	Number of responses is N.
40.	**not wordinone**	Number of responses is $(N - 1)$.
41.	**not wordintwo**	Number of responses is $(N - 1)$.
42.	**not wordinthree**	Number of responses is $(N - 1)$.
43.	**not wordinonetwo**	Number of responses is $(N - 2)$.

Test	Input	Expected Output
44.	**not wordinonetwothree**	Number of responses is (N - 3).
45.	**not wordinonethree**	Number of responses is (N - 2).
46.	**not wordintwothree**	Number of responses is (N - 2).
47.	**wordinone or wordinonetwo**	Number of responses is two, showing classes **Class1** and **Class2**.
48.	**wordinone or wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3**.
49.	**wordinzero or wordinone or wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3**.
50.	**wordinone and wordinonetwo**	Number of responses is one, showing class **Class1**.
51.	**wordinone and wordinonetwo and wordinonetwothree**	Number of responses is one, showing class **Class1**.
52.	**wordinzero and wordinone and wordinonetwo and wordinonetwothree**	Number of responses is zero.
53.	**not wordinone or not wordinonetwo**	Number of responses is (N - 1).
54.	**not wordinone and wordinonetwo**	Number of responses is one, showing class **Class2**.
55.	**not wordinone and not wordinonetwo**	Number of responses is (N - 2).

Test	Input	Expected Output
56.	**not wordinone and not wordinonetwo and not wordinonetwothree**	Number of responses is (N - 3).
57.	**wordinone and wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3.**
58.	**not wordinone and wordinonetwo or wordinonetwothree**	Number of responses is three, showing classes **Class1, Class2** and **Class3**.
59.	**not wordinone and not wordinonetwo or wordinonetwothree**	Number of responses is N.
60.	**not wordinone and not wordinonetwo or not wordinonetwothree**	Number of responses is (N - 2).

Acceptable behaviour resulting from erroneous input may be tested as follows:

Test	Input / Description	Expected Output
61.	**or wordinone** An *or* expression truncated at the beginning.	Number of responses is one, showing class **Class1**.
62.	**wordinone or** An *or* expression truncated at the end.	Number of responses is one, showing classes **Class1**.
63.	**and wordinone** An *and* expression truncated at the beginning.	A message advising a zero response.

Test	Input / Description	Expected Output
64.	**wordinone and** An *and* expression truncated at the end.	A message advising a zero response.
65.	**and not wordinone or wordinonetwo** A compound expression truncated at the beginning.	Number of responses is two, showing classes **Class1** and **Class2**.

5.2 Operating the search tool

The above description comprises the analysis and design for a search tool to aid the object-oriented developer, where the models incorporated within the description provide information required for future maintenance of the search tool. The environment for which the search tool is built is as follows:

- Dual 486 series personal computer;
- DOS version 6.2 operating system;
- Microsoft Windows version 3.11; and
- Digitalk Smalltalk/V for Windows version 2.0.

Appendix A contains the Smalltalk/V for Windows code for the search tool, which is a Smalltalk class named ClassFind. Appendix B contains the code for the WordIndex class which, although supplied by Digitalk, is additional to the standard Smalltalk/V for Windows environment.

Within this section, actions undertaken within the Smalltalk environment by the application developer are shown in bold Helvetica, such as press the **Enter ↵** key.

Within Smalltalk, the action required by the developer to build the search tool is the selection of the expression **ClassFind new initialise** followed by the selection of **Smalltalk** and **Do It**, as shown in Figure 64.

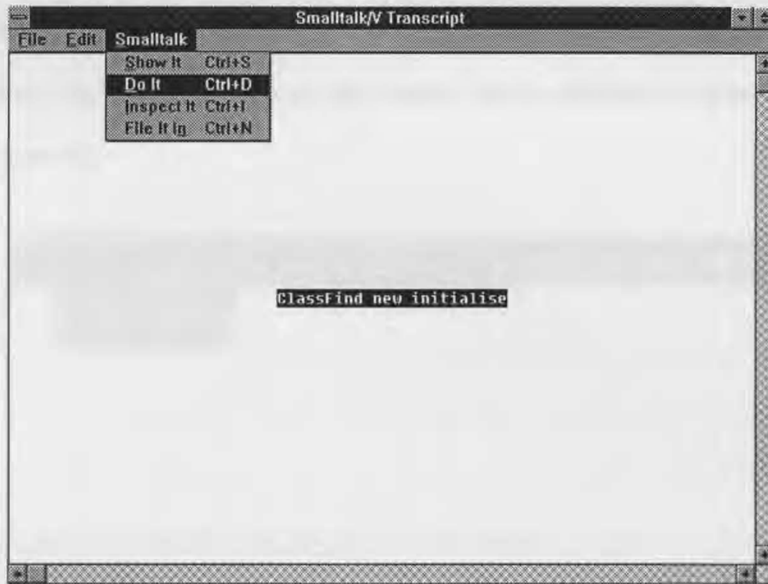

Figure 64. Initiating the search tool.

The search tool, for which the visible portion is a control bar and a display window, is shown in Figure 65.

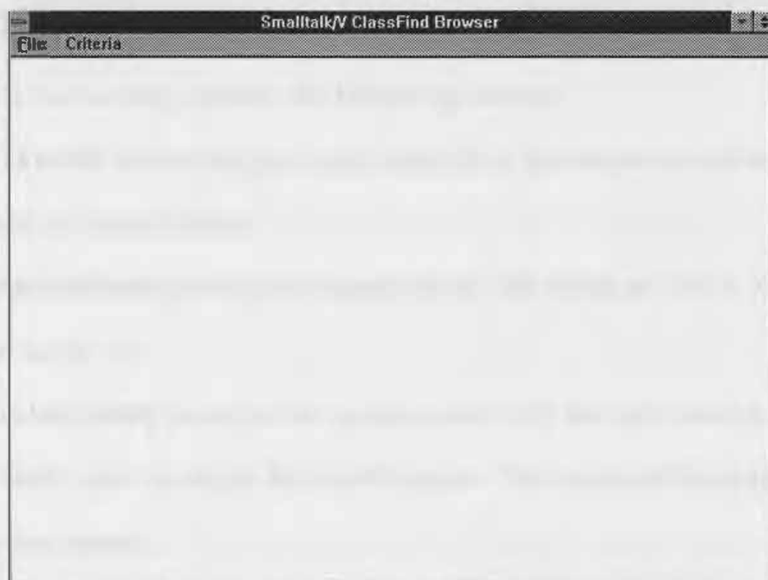

Figure 65. The initialised search tool.

It should be noted that the initialisation involves a time consuming build of a dictionary, requiring some minutes, in order to achieve fast search output. The use of the tool is not unduly compromised, however, as developers will close and rebuild the search tool only following the completion of each new application. Using a mouse, the **Criteria** entry on the control bar is selected to open a menu, as shown in Figure 66.

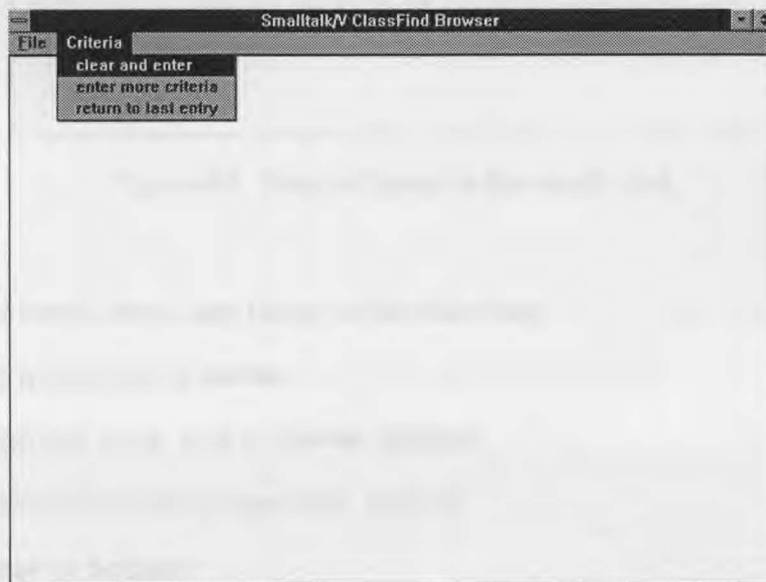

Figure 66. The search tool showing menu choices.

Selection of a menu entry enables the following actions:

- **clear and enter** clears the previous results from the output windows before prompting for search terms;
- **enter more criteria** prompts for search terms, the result of which follows the previous output; or
- **return to last entry** prompts for search terms, with the last entered search criteria visible and available for modification. The result of the search follows the previous output.

All of the above choices enable entry of the search terms, as shown in Figure 67.

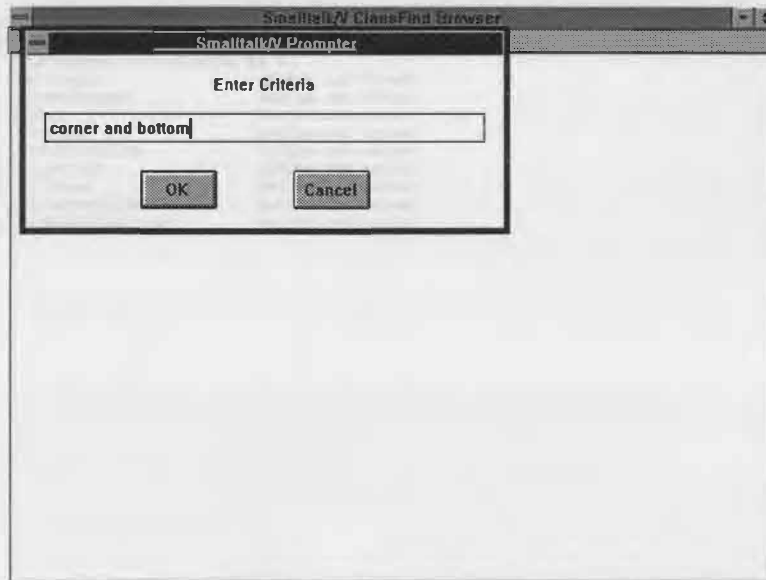

Figure 67. Entry of terms to the search tool.

The entry of search terms may be any of the following:

- A single word, such as **corner.**
- More than one word, such as **corner bottom**.
- Words joined by boolean operators, such as

  corner or bottom

  corner and bottom

  corner and not bottom.

  The boolean *and* and *or* operators are evaluated from left to right.
- Any combination of the above, including separate terms not explicate joined by an *or* operator, where each term is separately searched, such as

  corner and bottom point or bottom.

When the search terms have been entered, selection of the **OK** button with the mouse or pressing the **Enter** ↵ key, results in a display of the classes meeting the search criteria. Figure 68 illustrates a result in which nine classes match the selection criteria.

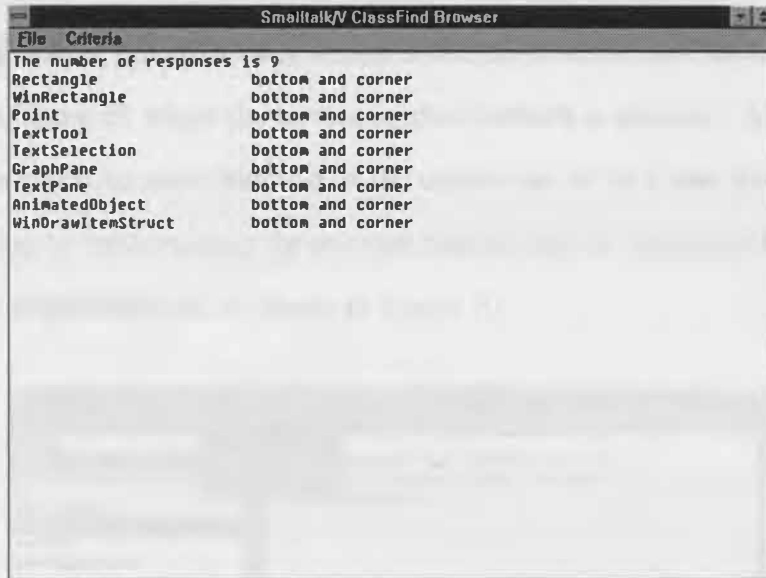

Figure 68. The result of a search.

Selection of any one class - any line - with the mouse opens a **Class Browser** object on that class. Figure 69 shows the result of selecting **Rectangle**.

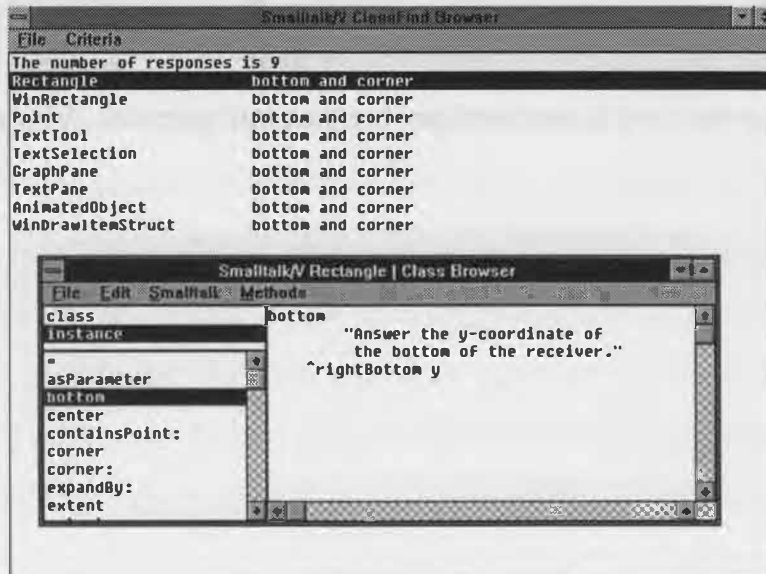

Figure 69. Selection of the result of a search.

The **Class Browser** provides ready access to the text code of each service method, as shown in Figure 69 where the service method **bottom** is selected. Additionally, by using the mouse to select **Method** on the control bar of the **Class Browser**, every class initiating or implementing the selected method may be ascertained by selecting **Senders** or **Implementors**, as shown in Figure 70.

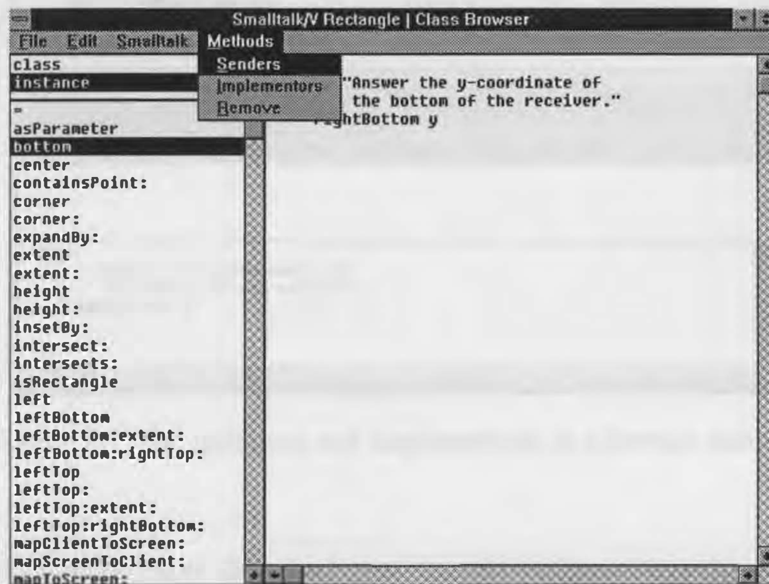

Figure 70. Selecting initiators and implementors of a service method.

The result of selecting **Senders** and **Implementors**, with the service method text displayed, is shown in Figure 71.

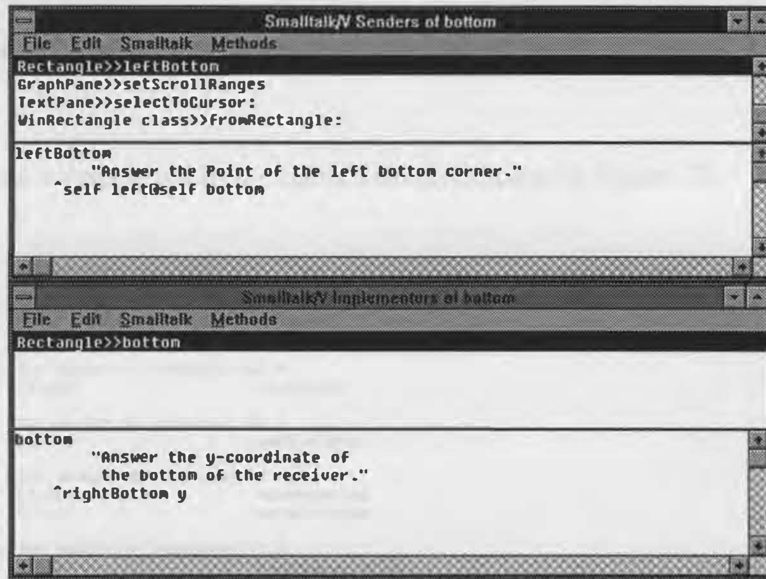

Figure 71. The initiators and implementors of a service method.

With the implementation of the search tool, the test processes developed in the previous section may be applied. Classes **Class0, Class1, Class2** and **Class3** are developed for test purposes such that **wordinzero** is not present in any class, **wordinone** is present in **Class1, wordinonetwo** is present in **Class1** and **Class2** and **wordinonetwothree** is present in **Class1, Class2** and **Class3**. Additionally, a new method 'onlyTestClasses' is defined within the class **Behavior** which answers a set consisting of the classes **Class0, Class1, Class2** and **Class3**. The **Class Find** text code is modified by changing two references from 'allClasses' to 'onlyTestClasses'.

The search tool may be employed to ensure that words are held in the **Class Find** dictionary in the manner described above, that is:

- **wordinone** occurs once - in **Class1**;
- **wordintwo** occurs once - in **Class2;**
- **wordinthree** occurs once - in **Class3;**

- **wordinonetwo** occurs twice - in **Class1** and **Class2;**
- **wordinonetwothree** occurs three times - in **Class1, Class2** and **Class3;**
- **wordinonethree** occurs twice - in **Class1** and **Class3**; and
- **wordintwothree** occurs twice - in **Class2** and **Class3.**

Entry of these words leads to the correct results shown in Figure 72.

Smalltalk/V ClassFind Browser

File Criteria

```
The number of responses is 1
Class1                    wordinone

The number of responses is 1
Class2                    wordintwo

The number of responses is 1
Class3                    wordinthree

The number of responses is 2
Class1                    wordinonetwo
Class2                    wordinonetwo

The number of responses is 3
Class1                    wordinonetwothree
Class2                    wordinonetwothree
Class3                    wordinonetwothree

The number of responses is 2
Class1                    wordinonethree
Class3                    wordinonethree

The number of responses is 2
Class2                    wordintwothree
Class3                    wordintwothree
```

Figure 72. Search tool dictionary contents for the test environment.

On the basis of the modification described above, the test procedures are completed as shown in Figure 73.

Test	Resulting Output	Verified
1.	Number of responses is zero.	✓
2.	Number of responses is one, showing class **Class1**.	✓
3.	Number of responses is one, showing class **Class2**.	✓
4.	Number of responses is one, showing class **Class3**.	✓
5.	Number of responses is two, showing classes **Class1** and **Class2**.	✓
6.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
7.	Number of responses is two, showing classes **Class1** and **Class3**.	✓
8.	Number of responses is two, showing classes **Class2** and **Class3**.	✓
9.	Number of responses is four, showing classes **Class0**, **Class1**, **Class2** and **Class3**.	✓
10.	Number of responses is three, showing classes **Class0**, **Class2** and **Class3**	✓
11.	Number of responses is three, showing classes **Class0**, **Class1** and **Class3**.	✓
12.	Number of responses is three, showing classes **Class0**, **Class1** and **Class2**.	✓
13.	Number of responses is two, showing classes **Class0** and **Class3**.	✓
14.	Number of responses is one, showing class **Class0**.	✓
15.	Number of responses is two, showing classes **Class0** and **Class2**.	✓
16.	Number of responses is two, showing classes **Class0** and **Class1**.	✓
17.	Number of responses is two, showing classes **Class1** and **Class2**.	✓
18.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
19.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
20.	Number of responses is one, showing class **Class1**.	✓
21.	Number of responses is one, showing class **Class1**.	✓
22.	Number of responses is zero.	✓
23.	Number of responses is three, showing classes **Class0**, **Class2** and **Class3**.	✓
24.	Number of responses is one, showing class **Class2**.	✓
25.	Number of responses is two, showing classes **Class0** and **Class3**.	✓
26.	Number of responses is one, showing class **Class0**.	✓
27.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
28.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
29.	Number of responses is four, showing classes **Classs0**, **Class1**, **Class2** and **Class3**.	✓
30.	Number of responses is two, showing classes **Class0** and **Class3**.	✓

Figure 73. Test results on modified code for the search tool.

Then, after reverting to the original code - unmodified for testing - the tests are repeated using the full environment. An approach, independent of that used by the search tool, is used to determine the total number of classes, as shown in Figure 74.

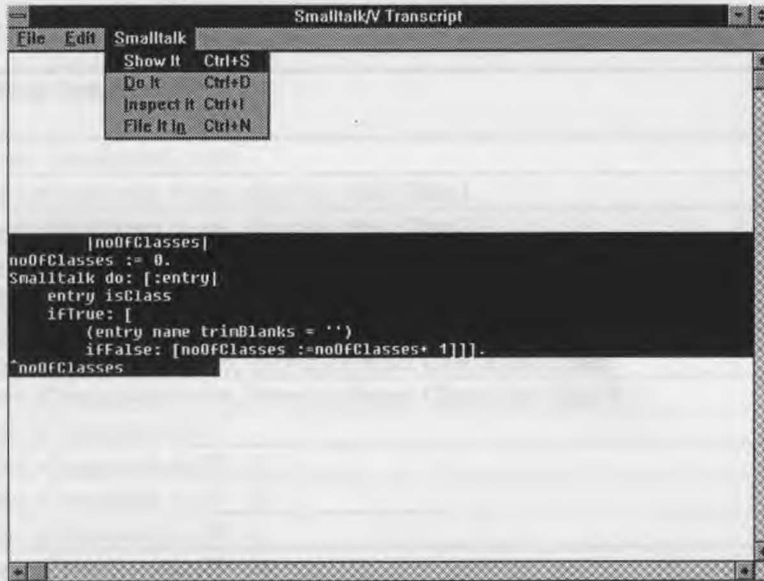

Figure 74. Determining the number of classes.

The test procedures are repeated on the unmodified code, as shown in Figure 75, where *N* is the total number of classes.

Test	Resulting Output	Verified
31.	Number of responses is zero.	✓
32.	Number of responses is one, showing class **Class1**.	✓
33.	Number of responses is one, showing class **Class2**.	✓
34.	Number of responses is one, showing class **Class3**.	✓
35.	Number of responses is two, showing classes **Class1** and **Class2**.	✓
36.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
37.	Number of responses is two, showing classes **Class1** and **Class3**.	✓
38.	Number of responses is two, showing classes **Class2** and **Class3**.	✓
39.	Number of responses is N.	✓
40.	Number of responses is $(N - 1)$.	✓
41.	Number of responses is $(N - 1)$.	✓
42.	Number of responses is $(N - 1)$.	✓
43.	Number of responses is $(N - 2)$.	✓
44.	Number of responses is $(N - 3)$.	✓
45.	Number of responses is $(N - 2)$.	✓
46.	Number of responses is $(N - 2)$.	✓
47.	Number of responses is two, showing classes **Class1** and **Class2**.	✓
48.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
49.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
50.	Number of responses is one, showing class **Class1**.	✓
51.	Number of responses is one, showing class **Class1**.	✓
52.	Number of responses is zero.	✓
53.	Number of responses is $(N - 1)$.	✓
54.	Number of responses is one, showing class **Class2**.	✓
55.	Number of responses is $(N - 2)$.	✓
56.	Number of responses is $(N - 3)$.	✓
57.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
58.	Number of responses is three, showing classes **Class1**, **Class2** and **Class3**.	✓
59.	Number of responses is N.	✓
60.	Number of responses is $(N - 2)$.	✓

Figure 75. Test results on unmodified code for the search tool.

The test procedures for erroneous input are shown in Figure 76.

Test	Resulting Output	Verified
61.	Number of responses is one, showing class **Class1**.	✓
62.	Number of responses is one, showing class **Class1**.	✓
63.	A message advising a zero response.	✓
64.	A message advising a zero response.	✓
65.	Number of responses is two, showing classes **Class1** and **Class2**.	✓

Figure 76. Test results for erroneous input to the search tool.

5.3 Summary

Following the principles and process outlined in earlier chapters, the construction, usage and testing of the search tool is described. In the next chapter, the tool is used to complete the ATM design.

6 The Demonstration System

For the development of an ATM, previous chapters have demonstrated an analysis and design method to identify the objects, determine the responsibilities and the associations and detail the attributes. In this chapter, the method is completed by ascertaining the inheritance links - using the search tool to discover suitable classes - and by developing test procedures. Then, an ATM system that corresponds with both the requirement and the analysis and design method is described.

6.1 Completion of the Analysis and Design

To complete the analysis and design of an ATM, the method requires that the developer build the inheritance links and establish test procedures, described in this section.

Build the inheritance links

A number of classes have been established in previous chapters. From the requirements for each class, the search tool is used to locate suitable classes based on the text content of each class. In practice, this is an iterative process, refining the search terms until suitable classes are discovered. Described below are both the requirement for each class and the end result for each search operation, indicating that the search tool is successful at discovering suitable classes.

- **Card Reader**

  Requirement for class: Represent the basic operation of a mechanical device by responding with a boolean value (true or false) and manage numbers. To simulate insertion of a card, prompt for an entry of the card details.

  Search tool operation: An input of **basic and boolean and number** into the search tool directs the developer to the **Object** class, while an input of **prompt and entry** indicates a relationship with the **Prompter** class.

- **User Interface**

  Requirement for class: Enable text to be displayed or an entry cancelled. Initiate events with a pushbutton. Append, search for and accept text.

  Search tool operation: From an entry of **display and text and enter and cancel** the developer is directed by the search tool to the **Window** class and its inherited **Sub Pane** class. An input of **event and pushbutton** to the search tool directs the developer to the **Button** class. An input of **append and search and accept and text** directs the developer to the **Text Pane** or the **Text Window** classes, however, further investigation with the **Class Browser** shows that the **Text Window** class is only used for initiating a single window.

- **Account**

  Requirement for class: Look up the customer identification number - a unique key - and operate on the value contained by the key. To establish the account, prompt for entry of the account details. Build and restore a list of the indexed values for each account.

  Search tool operation: The search tool entry of **look and up and key and value and contains** directs the developer to the **Dictionary** class. A search tool entry of **prompt and entry** indicates the use of the **Prompter** class. A search tool entry of **build and restore and list and index and value** directs the developer to the **List Box** and **Debugger** classes, while further investigation of each class leads to the selection of the **List Box** class.

- **Deposit Slot and Dispenser**
  Requirement for class: Represent the basic operation of a mechanical device by responding with a boolean value (true or false) and manage numbers.

  Search tool operation: An input of **basic and boolean and number** into the search tool directs the developer to the **Object** class.

- **Receipt Printer**
  Requirement for class: Represent the basic operation of a mechanical device by responding with a boolean value (true or false) and manage numbers. Append text within a form that may be closed.

  Search tool operation: An entry to the search tool of **basic and boolean and number** directs the developer to the **Object** class, while entry of **append and text and form and close** leads to the selection of the **Text Pane** class.

The aforementioned search operations successfully discover suitable classes, although in practice, the search demands many iterations. Salton (1989, p.236), cited in Chapter 4, explains that the resultant output from the search may be too little or too large, depending on the input for the search. With experience, the developer learns to use boolean operators to broaden or narrow the search. Additionally, provision of a thesaurus may reduce the experience required; however, Salton (1989, p. 301), cited in Chapter 4, describes the difficulty in constructing a thesaurus.

Within Smalltalk/V, the **View Manager** class contains the behaviour to open and manage the **Text Pane** and **List Box** classes and their parent **Sub Pane** and **Window** classes. The final *Object Model* for the demonstration ATM is shown in Figure 77.

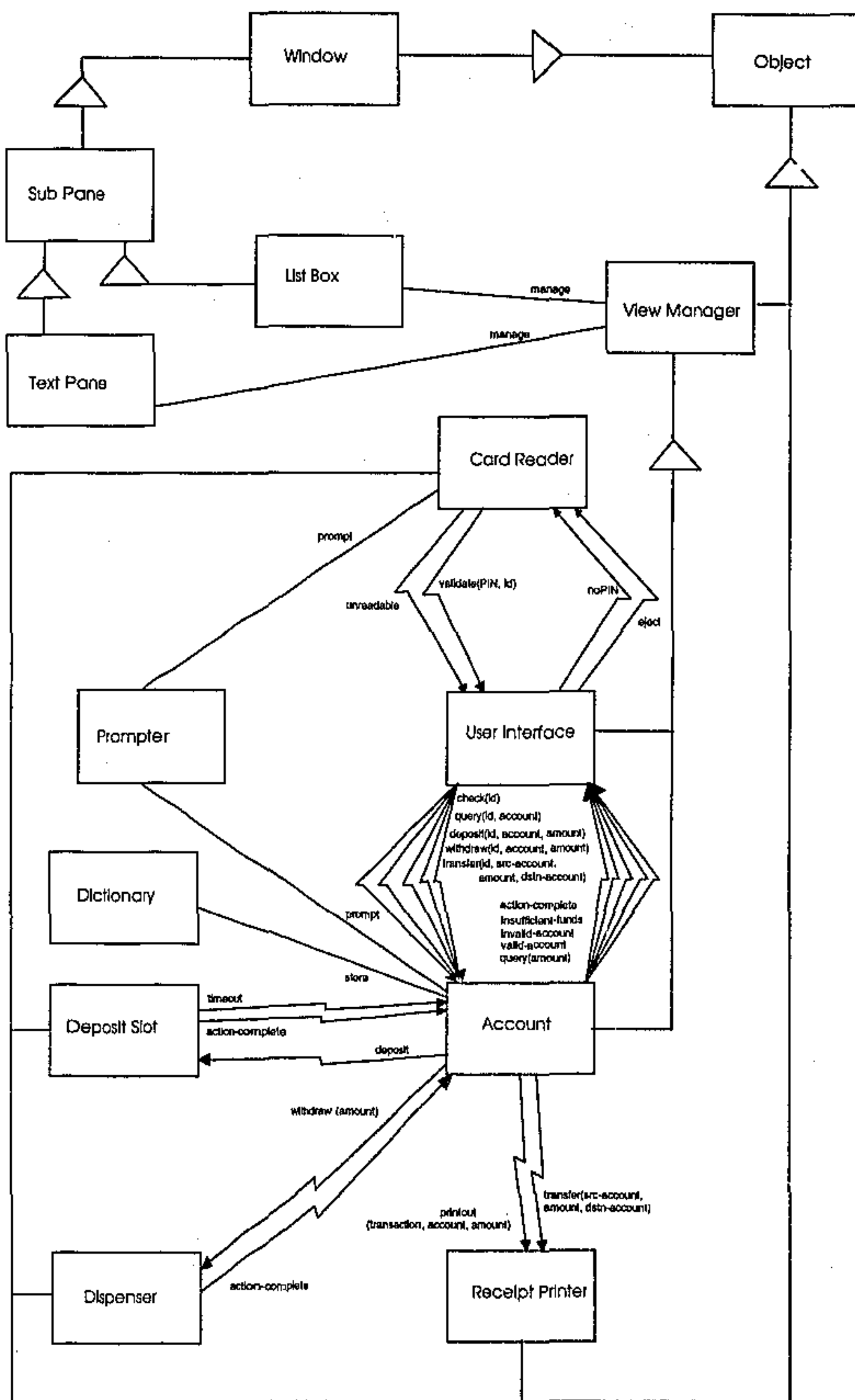

Figure 77. The final *Object Model* for the ATM.

Testing the ATM

The following procedures are developed to test the functionality and accuracy of the demonstration ATM, based on setting up and operating an account with all of the required functions.

Test	Input	Expected Output
1.	Set up an account with an identification number 10 and with a cheque balance of $123.15 and a savings balance of $462.55.	The Account window shows id: 10; cheque balance: $123.15; and savings balance: $462.55.
2.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1111.	Request for PIN re-entry.
3.	Enter a PIN of 1111 a further two times.	Because only three attempts are allowed, the card is held and the user informed.
4.	Insert card with PIN 1234 and identification number 11. Enter a PIN of 1234.	Because there is no identification number 11, the card is held and the user informed.
5.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1234. Deposit $50 in the cheque account.	A transaction receipt is produced for a $50 deposit. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $462.55.
6.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1234. Withdraw $50 from the savings account.	A transaction receipt is produced for a $50 withdrawal. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $412.55.

Test	Input	Expected Output
7.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1234. Withdraw $500 from the savings account.	Because there are insufficient funds, the user is informed and the card ejected.
8.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1234. Transfer $50 from the cheque account to the savings account.	A transaction receipt is produced for a $50 transfer. The Account window shows id: 10; cheque balance: $123.15; and savings balance: $462.55.
9.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1234. Transfer $50 from the savings account to the cheque account.	A transaction receipt is produced for a $50 transfer. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $412.55.
10.	Insert card with PIN 1234 and identification number 10. Enter a PIN of 1234. Query the balance of the cheque account.	A transaction receipt shows a cheque account balance of $173.15. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $412.55.

6.2 Description of the ATM System

The ATM system developed for this study simulates the operation of the system analysed and designed in previous chapters. Operation of a button is achieved by placing the mouse over the graphic button and depressing the left mouse button. Appendix C contains the Smalltalk/V for Windows code for the demonstration ATM, consisting of the Smalltalk classes **ATM**, **Card Reader**, **User Interface**, **Account**, **Deposit Slot**, **Dispenser** and **Receipt Printer**.

Within the Smalltalk environment, the action required to initiate the demonstration is the selection of the expression **ATM new initialise**, followed by the selection of **Smalltalk** and **Do It**, as shown in Figure 78.

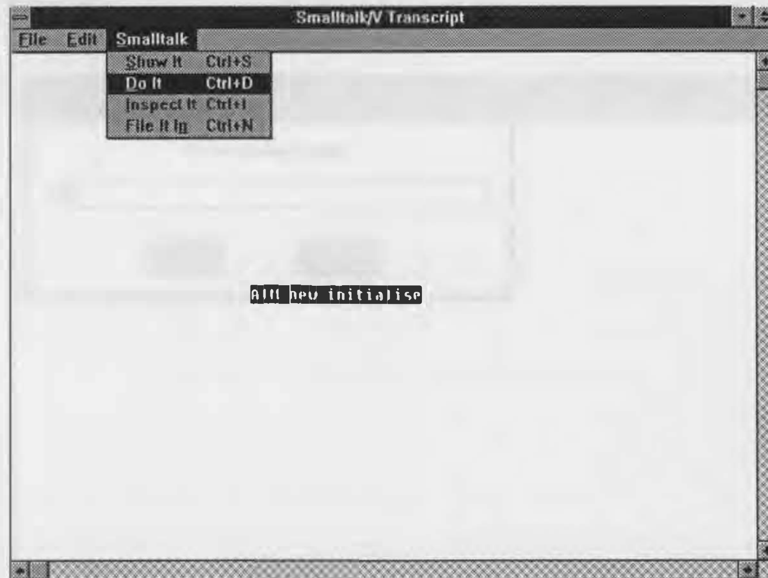

Figure 78. Initiating the demonstration ATM.

The operations involved in using the ATM are:

- establish or modify account details;
- insert a bank card and enter the PIN;
- deposit funds;
- withdraw funds;
- transfer funds;
- query funds; and
- cancel a transaction.

The operations are described below in more detail.

Establish or Modify Account Details

The first task following the initialisation of the system is to establish the accounts available to the ATM. A prompt is made for the account identification number, as shown in Figure 79.

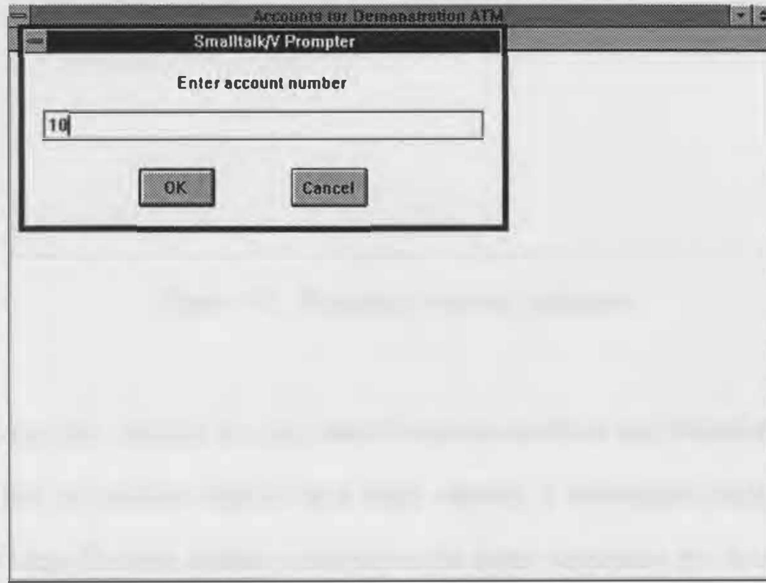

Figure 79. Entering an account identification number.

Then, two balances are entered, where the first balance is for the cheque account and the second balance is for the savings account. Figure 80 shows entries of $123.15 for the cheque account balance and $462.55 for the savings account balance.

Figure 80. Entering account balances.

In the same manner, further account identification numbers and balances may be entered. When all account details have been entered, a subsequent null response for the account identification number completes the entry sequence and the account details are available for inspection in a window. This is shown in Figure 81, in which a window reflects the status of each account, allowing inspection during the ATM transaction.

Figure 81. Account details.

Further account details may be added and existing account balances modified by selecting **Account Entry** and **enter or change account balances**, as shown in Figure 82, enabling entry of the account identification number and account balances as previously described.

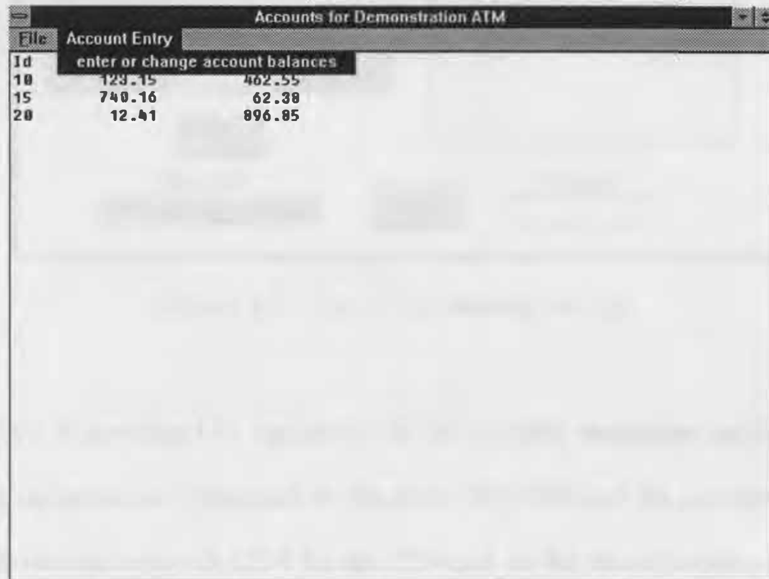

Figure 82. Changing account details.

Insert a Bank Card and Enter the PIN

The appearance of the ATM as it first appears with a welcome message is shown in Figure 83.

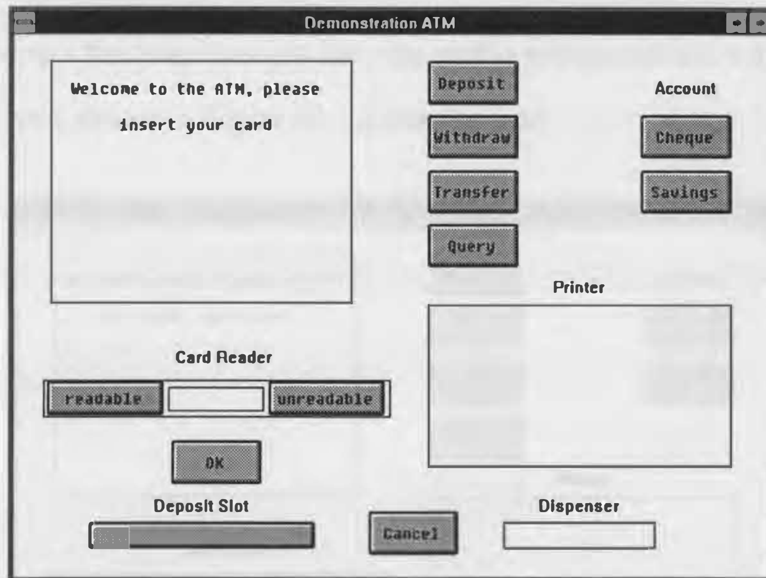

Figure 83. The ATM waiting for use.

Entry of a card is simulated by operating the card reader **readable** button, then transmitting information contained on the card - the PIN and the account number. Figure 84 shows the entry of 1234 for the PIN and 10 for the account number.

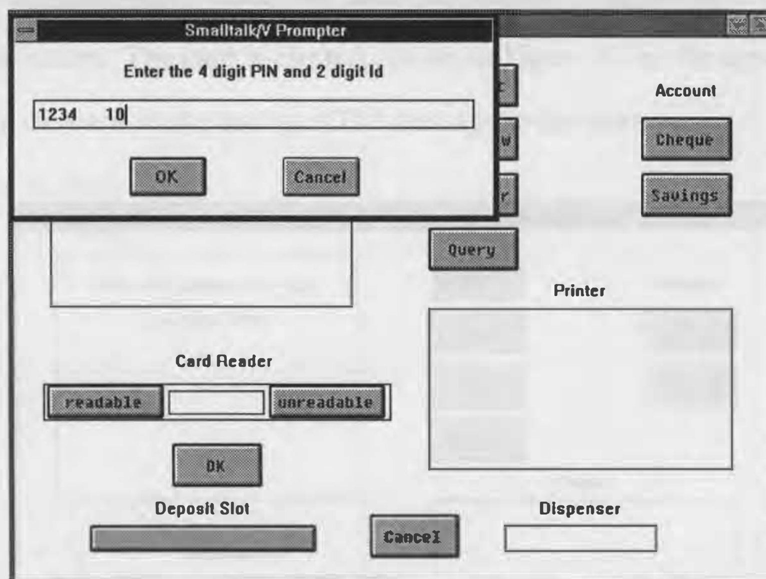

Figure 84. Entering the card details.

The PIN is then input to the ATM and the **OK** button operated. If either the account identification number contained on the card does not exist or the PIN tendered to the

ATM is incorrect for three attempts, then the card is not ejected and a message on the ATM screen, shown in Figure 85, informs the user.

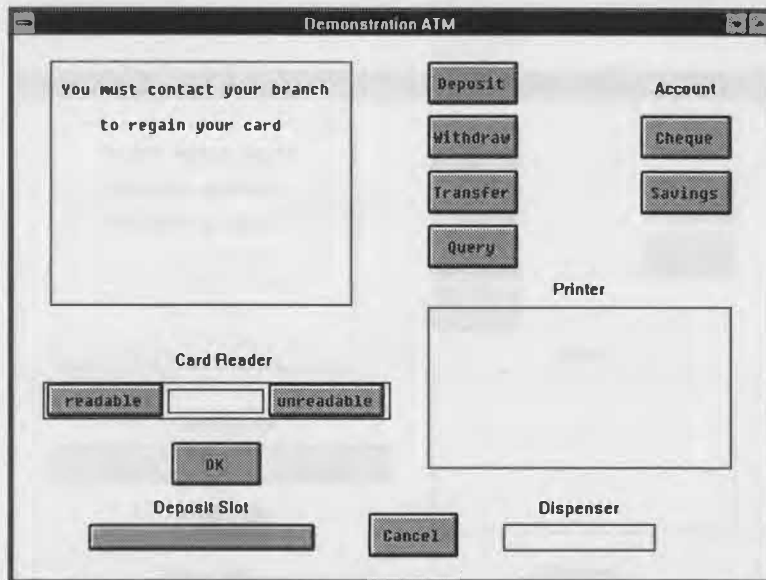

Figure 85. Card held by ATM.

Entry of a bank card which cannot be read is simulated by operating the card reader **unreadable** button. The card is ejected, shown in Figure 86 by the appearance of 'eject' within the card reader and an ATM message to the user.

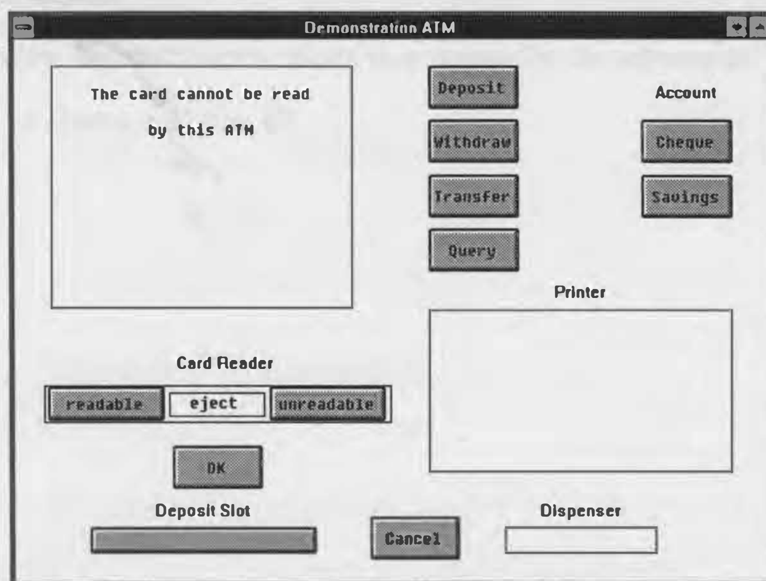

Figure 86. An unreadable bank card inserted.

If the card is valid, a selection of the transaction type is requested, as shown in Figure 87.

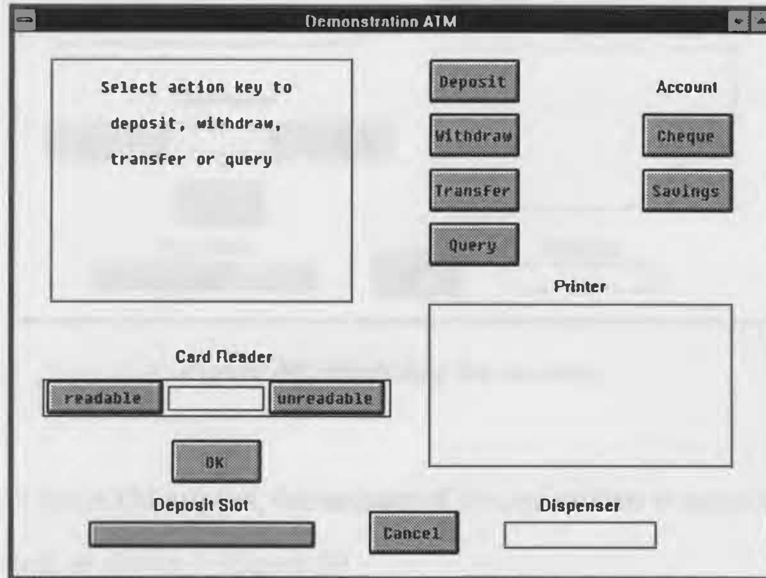

Figure 87. Selecting the transaction.

A transaction selection is made by operating the **Deposit**, **Withdraw, Transfer** or **Query** button.

Deposit transaction

Operation of the **Deposit** button results in a request for the appropriate account to be selected, as shown in Figure 88.

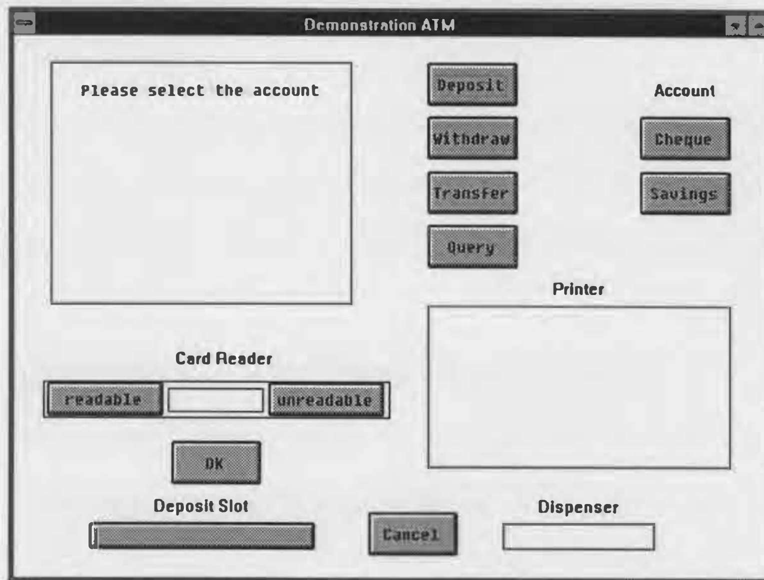

Figure 88. Selecting the account.

In response to the ATM request, the amount of the transaction is entered and the **OK** button operated, as shown in Figure 89.

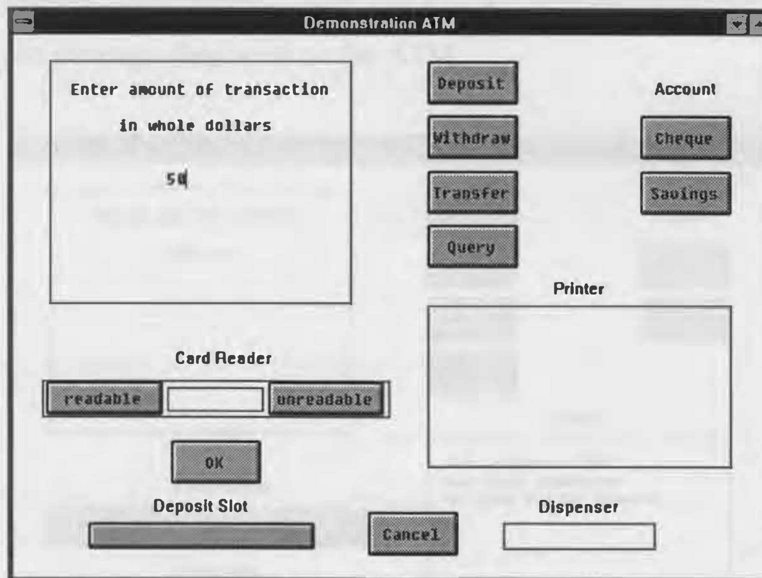

Figure 89. Entering the amount.

Insertion of money into the deposit slot is then requested, as shown in Figure 90. This insertion is simulated by operating the **Deposit Slot** button.

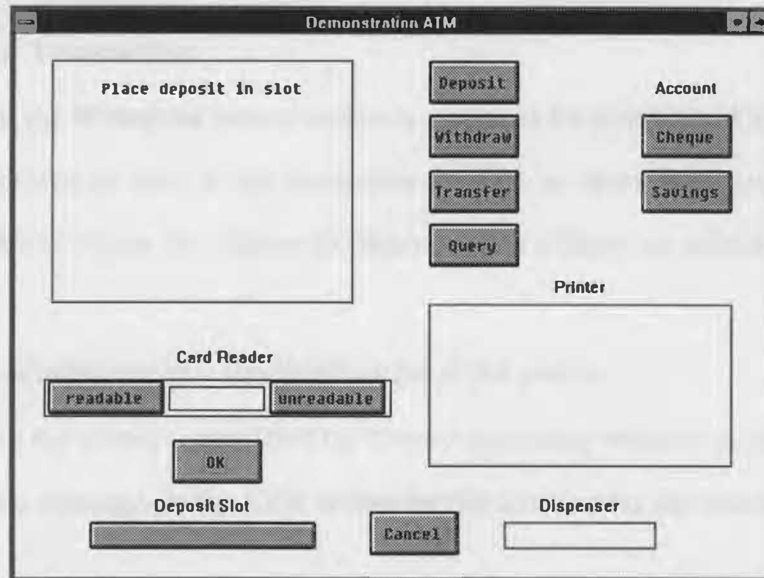

Figure 90. Request for insertion of deposit.

Figure 91 shows the completion of the transaction, consisting of:

- a printed receipt - simulated on the ATM receipt printer;
- the card ejected - simulated by 'eject' appearing within the card reader; and
- a thank you message displayed on the ATM.

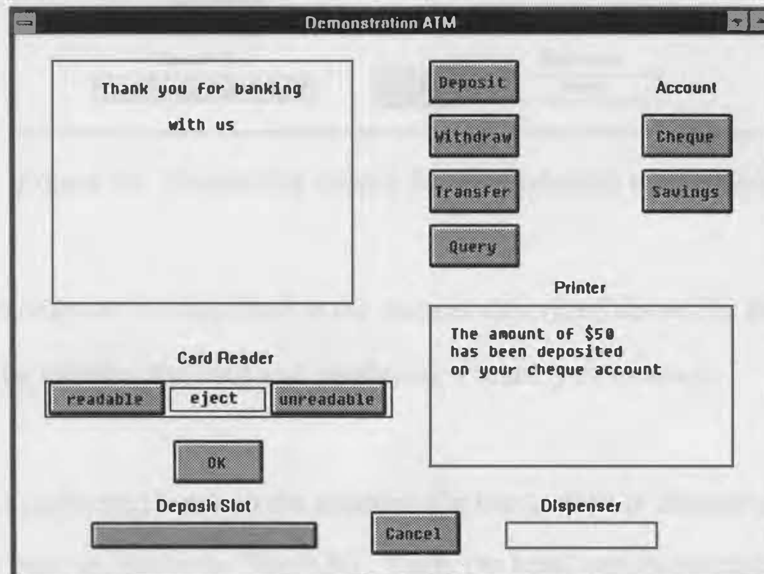

Figure 91. Completion of a deposit transaction.

Withdrawal Transaction

Operation of the **Withdraw** button results in a request for selection of the relevant account, followed by entry of the transaction amount, as described above and shown in Figure 88 and Figure 89. Figure 92 illustrates that if there are sufficient funds, the ATM:

- issues a printed receipt - simulated on the ATM printer;
- dispenses the money - simulated by 'money' appearing within the dispenser; and
- displays a message on the ATM screen for the user to take the money.

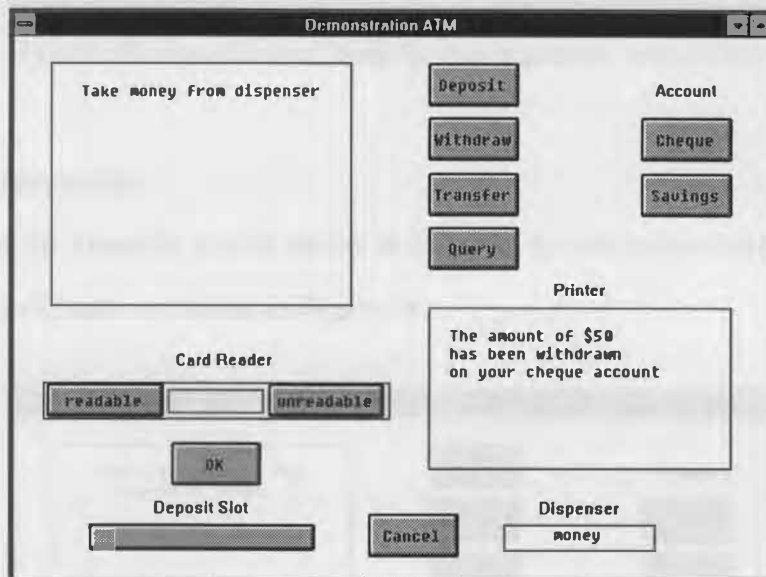

Figure 92. Dispensing money for a withdrawal transaction.

Then, the transaction is completed in the manner described above for the deposit transaction by ejecting the card and displaying a thank you message.

If there are insufficient funds in the account, the transaction is aborted and the ATM informs the user, as shown in Figure 93. Then, the bank card is ejected and the transaction terminated.

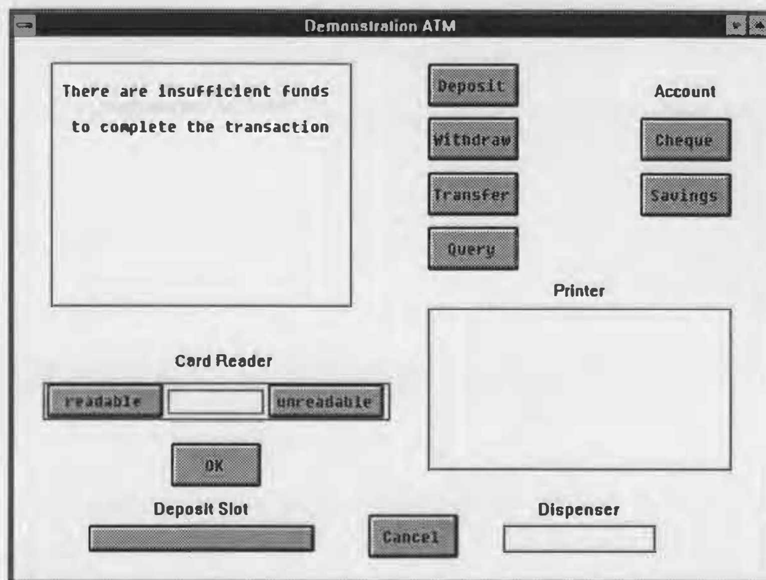

Figure 93. Insufficient funds for the requested transaction.

Transfer Transaction

Operation of the **Transfer** button results in a request for selection of the account that is the source of funds, as shown in Figure 94.

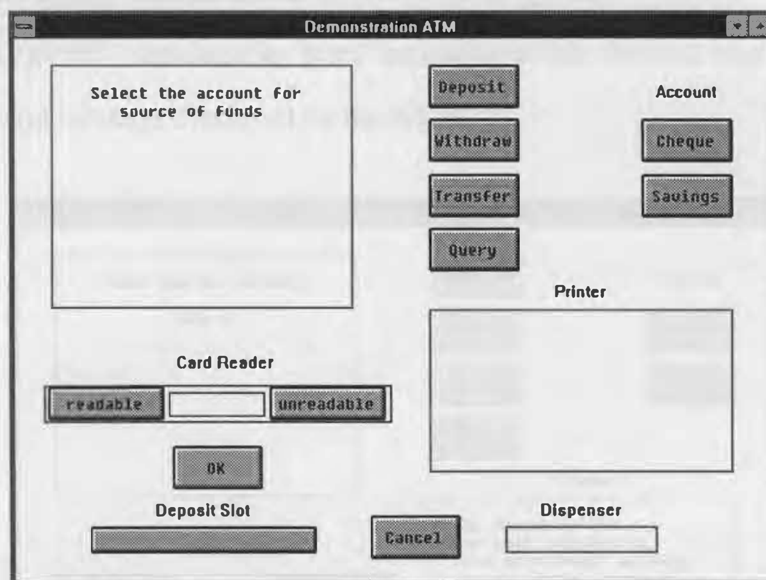

Figure 94. Request for source of funds.

The destination account is then requested, as shown in Figure 95.

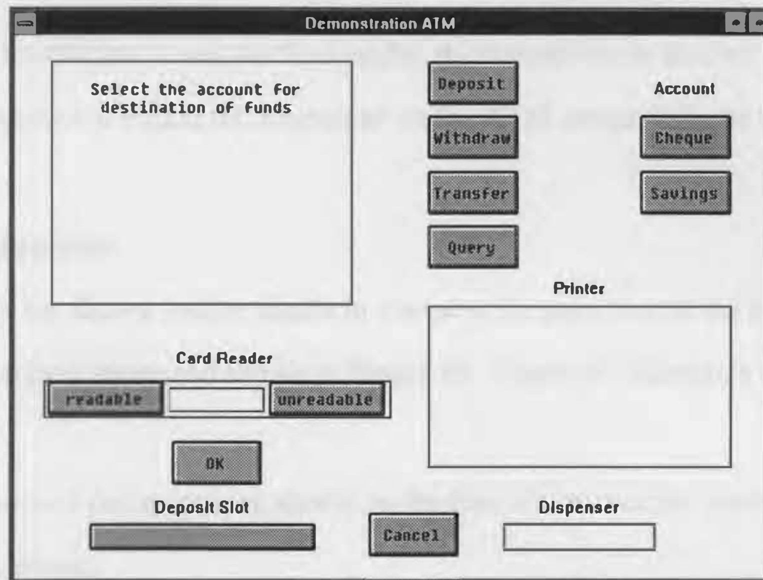

Figure 95. Request for destination of funds.

Following the selection of an account for the destination of funds, the amount to be transferred is requested in the manner described above and shown in Figure 89. Then, the transaction is completed as shown in Figure 96, consisting of:

- a printed receipt - simulated on the ATM receipt printer;
- the card ejected - simulated by 'eject' appearing within the card reader; and
- a thank you message displayed on the ATM.

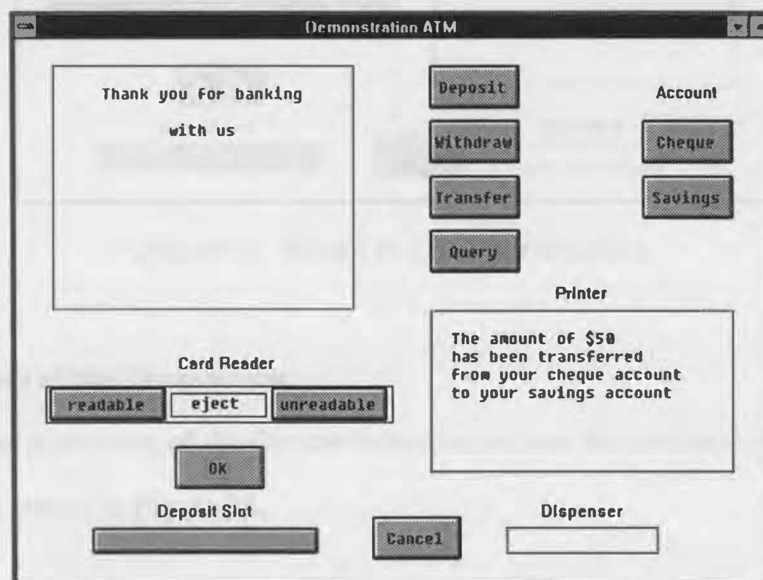

Figure 96. Completion of a transfer transaction.

If there are insufficient funds for the transfer, the transaction is aborted and, as previously shown in Figure 93, a message on the ATM screen informs the user.

Query Transaction

Operation of the **Query** button results in a request for selection of the account, in the manner described above and shown in Figure 88. Figure 97 illustrates the following result:

- the balance of the account is shown on the transaction receipt, simulated on the receipt printer;
- the card is ejected; and
- a thank you message is displayed on the ATM.

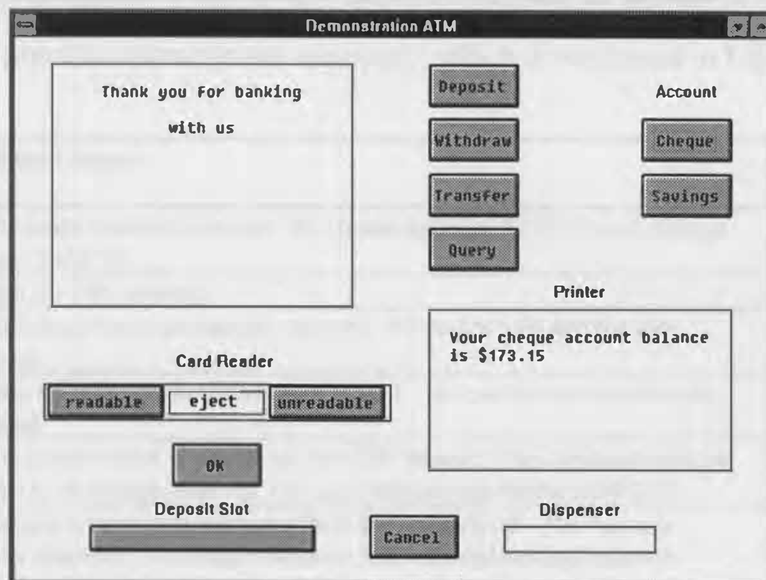

Figure 97. Result of a query transaction.

Cancellation of the Transaction

At any time, depression of the **Cancel** button terminates the transaction and ejects the card, as shown in Figure 98.

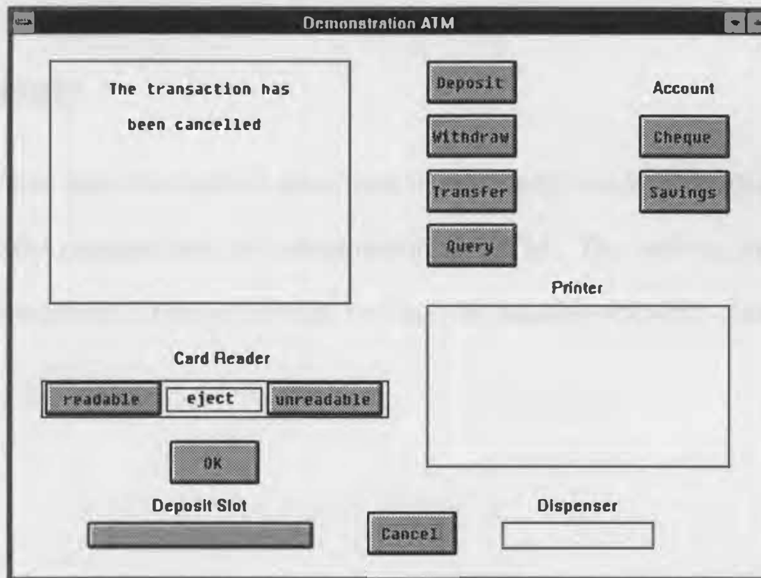

Figure 98. Cancellation of a transaction.

The system description in this chapter demonstrates that all aspects of the test procedures function correctly and accurately, which is confirmed in Figure 99.

Test	Resulting Output	Verified
1.	The Account window shows id: 10; cheque balance: $123.15; and savings balance: $462.55.	✓
2.	Request for PIN re-entry.	✓
3.	Because only three attempts are allowed, the card is held and the user informed.	✓
4.	Because there is no identification number 11, the card is held and the user informed.	✓
5.	A transaction receipt is produced for a $50 deposit. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $462.55.	✓
6.	A transaction receipt is produced for a $50 withdrawal. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $412.55.	✓
7.	Because there are insufficient funds, the user is informed and the card ejected.	✓
8.	A transaction receipt is produced for a $50 transfer. The Account window shows id: 10; cheque balance: $123.15; and savings balance: $462.55.	✓
9.	A transaction receipt is produced for a $50 transfer. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $412.55.	✓
10.	A transaction receipt shows a cheque account balance of $173.15. The Account window shows id: 10; cheque balance: $173.15; and savings balance: $412.55.	✓

Figure 99. Test results from operation of the ATM.

6.3 Summary

The chapter has used the method described in this study to complete the analysis and design, then the construction, of a demonstration ATM. The method includes the successful operation of the search tool to discover suitable reusable classes.

7 Conclusion

In order to utilise the object-oriented paradigm effectively, a developer requires tools that aid the selection of classes, a statement that is supported in the study by such authors as Booch (1987), Meyer (1988), Frakes & Nejmeh (1988), Hooper & Chester (1991) and Dusink & Hall (1991). The study demonstrates an effective method of achieving reuse by employing modern text retrieval techniques.

The structure of the study follows a logical pattern, commencing with an examination of the principles of the object-oriented paradigm ascertained from sources such as Wirfs-Brock, Wilkerson & Weiner (1990), Wirfs-Brock & Johnson (1990), Booch (1991), Rumbaugh et al. (1991), Embley et al. (1992), Henderson-Sellers (1992), Booch (1994), Tanzer (1995), Rumbaugh (1995) and from organisations such as the Object Management Group (Soley, 1992) and Digitalk (1992). Then, the aspects of analysis and design for the object-oriented paradigm are examined, particularly as described by Rumbaugh et al. (1991) and Embley et al. (1992). Methodical steps for object-oriented analysis and design are outlined as:

(i) identify the objects;

(ii) determine the responsibilities;

(iii) determine the associations;

(iv) detail the attributes; and

(v) build the inheritance links.

Steps (i) to (iv) above are demonstrated in the initial analysis and design of an ATM application, specified in Wirfs-Brock, Wilkerson & Weiner (1990). While step (v) is as germane as the other steps, its treatment is deferred until a reuse background is established and a suitable direction identified.

Issues in reusing software components are examined, particularly as they relate to object-orientation. Lewis et al. (1992) are cited to demonstrate that it is this reuse capability which may lift the object-oriented paradigm to higher levels of productivity than procedural methods. Frakes & Nejmeh (1988) describe the difficulties in employing standard information technology approaches to store and retrieve unstructured information and they support the concept of using full and free text retrieval concepts to store and, subsequently, locate reusable software code. An explanation of full and free text retrieval is provided, citing Salton (1989), then the principles are employed to develop a search tool. Following analysis and design, the tool is constructed using Smalltalk; then, a description of the search tool and instruction in its use are included. Verification of the tool is provided by working to a comprehensive test plan.

Operation of the tool is convenient for a system developer, because:

- it may be prepared for use within a few minutes;
- it rapidly responds to search queries which may then be expanded or narrowed; and
- the tool window may be iconised out of the developer's work area while remaining available for use as required.

The tool is easy to use because natural language is input. Horton (1990) is cited as claiming that, employing natural language, subtleties of meaning may be found without filtering out important information. The search tool incorporates a ready ability to extend the search, in line with Kimmel's (1990) advice that a search may act as a pointer to further information.

The search tool is successfully employed by the developer to build the inheritance links - step (v) of the methodical steps outlined earlier in this chapter - and thereby

complete the analysis and design of the ATM. The system is then demonstrated and its operation verified.

Whilst the process of employing full and free text retrieval is effective, the experience obtained in developing a demonstration ATM suggests a further direction in its development, namely the incorporation of a thesaurus. This may reduce both the experience required of the developer and the search time using the tool. The advantages of thesaurus usage are addressed within the study by citing Salton (1989), together with the difficulty of thesaurus construction requiring a committee of experts.

Additionally, Booch (1994) is cited promoting both Smalltalk and C++ as the most pervasive languages. Reporting the issues in following the analysis and design in this study to develop a search tool for a class library in a C++ environment would add to a general understanding of object-oriented analysis and design.

Enhancing reuse within the object-oriented paradigm, the study demonstrates that the text retrieval techniques of a modern library may be employed in finding suitable classes within the object-oriented paradigm.

Reference List

Booch, G. (1987). *Software components with Ada.* Reading, MA: Benjamin/Cummings Publishing Company.

Booch, G. (1991). *Object-oriented design, with applications.* Menlo Park, Calif: Benjamin/Cummings Publishing Co.

Booch, G. (1994, November). Coming of age in an object-oriented world. *IEEE Software*, 33 - 41.

Coad, P. (1991, January). New advances in object-oriented analysis. *Journal of Object Oriented Programming*, 44 - 49.

Cortez, E. M. & Kazlauskas, E. J. (1986). *Managing information systems and technologies.* New York: Neal-Schuman Publishers, Inc.

de Champeaux, D. & Faure, P. (1992, March/April). A comparative study of object-oriented analysis methods. *Journal of Object Oriented Programming*, 21 - 33.

Digitalk, Inc. (1992). *Smalltalk/V for Windows tutorial and programming handbook.* Los Angeles, CA: Author.

D'Souza, D. & Graff, P. (1995, February). Working with OMT: Model integration. *Journal of Object Oriented Programming*, 23 - 29.

Dusink, L. & Hall, P. (Eds.) (1991). *Software re-use Utrecht 1989.* London: Springer-Verlag.

Embley, D. W., Kurtz, B. D. & Woodfield, S. N. (1992). *Object-oriented systems analysis: A model-driven approach.* Englewood Cliffs, NJ: Yourdon Press.

Frakes, W. B. & Nejmeh, B. A. (1988). An information system for software reuse. In W. Tracz (Ed.) *Software reuse: Emerging technology* (pp. 142 - 151). Washington, DC: Computer Society Press.

Freeman, C. & Henderson-Sellers, B. (1991). OLMS: An object library management support system. In J. Potter, M. Tokoro & B. Meyer (Eds.) *Technology of object-oriented languages and systems: Tools 6* (pp. 175 - 180). Sydney: Prentice-Hall, Inc.

Gehani, N. (1989). *Ada: An advanced introduction.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Gibbs, S., Tsichritzis, D., Casais, D., Nierstrasz, O. & Pintado, X. (1990). Class management for software communities. *Communications of the ACM,* Vol 33, No 9, 90 - 103.

Gibson, E. (1990, October). Objects - born and bred. *Byte*, 245 - 254.

Henderson-Sellers, B. (1992). *A book of object-oriented knowledge.* Sydney: Prentice-Hall, Inc.

Hooper, J. W. & Chester, R. O. (1991). *Software reuse: Guidelines and methods.* New York, NY: Plenum Press.

Horton, S. (1990). Handling full text. In P. Gillman (Ed.) *Text retrieval: The state of the art* (pp. 56 - 64). London, UK: Taylor Graham.

Jacobson, I. (1991, March/April). Industrial development of software with an object-oriented technique. *Journal of Object Oriented Programming*, 30 - 41.

Kendall, K. E. and Kendall, J. E. (1992). *Systems analysis and design*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Khoshafian, S. & Abnous, R. (1990). *Object orientation: Concepts, languages, databases, user interfaces*. New York, NY: John Wiley and Sons.

Kimmel, C. (1990). Integrating text into management systems. In P. Gillman (Ed.) *Text retrieval: The state of the art* (pp. 101 - 111). London, UK: Taylor Graham.

Korson, T. & McGregor, J. D. (1990). Understanding object oriented: A unifying paradigm. *Communications of the ACM*, Vol. 33 No. 9, 40 - 60.

Kuhn, T. S. (1970). *The structure of scientific revolutions.* Chicago: The University of Chicago Press.

Lewis, J. A., Henry, S. M., Kattura, D. G. & Schulman, R. S. (1992, July/August). On the relationship between the object-oriented paradigm and software reuse: An empirical investigation. *Journal of Object Oriented Programming*, 35 - 41.

Meyer, B. (1988). *Object oriented software construction.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Olle, T. W., Hagelstein, J., Macdonald, I., Rolland, C., Sol, H. G., Van Assche, F. J. M. & Verrijn-Stuart, A. A. (1991). *Information systems methodologies: A framework for understanding*. Wokingham, England: Addison-Wesley Publishing Company.

Page-Jones, M. (1988). *The practical guide to structured systems design.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Pressman, R. S. (1992). *Software engineering. A practitioner's approach.* New York, NY: McGraw-Hill, Inc.

Prieto-Diaz, R. & Jones, G. A. (1988). Breathing new life into old software. In W. Tracz (Ed.) *Software reuse: Emerging technology* (pp. 152 - 160). Washington, DC: Computer Society Press.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-oriented modeling and design.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Rumbaugh, J. (1995, January). OMT: The object model. *Journal of Object Oriented Programming*, 21 - 46.

Salton, G. (1989). *Automatic text processing.* Reading, Mass: Addison-Wesley.

Scharenberg, M. E. & Dunsmore, H. E. (1991, January). Evolution of classes and objects during object-oriented design and programming. *Journal of Object Oriented Programming*, 30 - 34.

Soley, R. M. (Ed.) (1992). *Object management architecture guide.* Framingham, MA: Object Management Group.

Tanzer, C. (1995, February). Remarks on object-oriented modeling of associations. *Journal of Object Oriented Programming*, 43 - 46.

Wirfs-Brock, R. J. & Johnson, R. E. (1990). Surveying current research in object-oriented design. *Communications of the ACM*, Vol. 33 No. 9, 104 - 124.

Wirfs-Brock, R. J., Wilkerson, B. & Wiener, L. (1990). *Designing object-oriented software.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Appendices

Appendix A. Code for the Class Find Search Tool.

The search tool is the Class Find class within Smalltalk\V for Windows, which is listed below as a result of a Smalltalk File Out operation:

```
WordIndex variableSubclass: #ClassFind
  instanceVariableNames:
    'classOutput classWindow criteria searchAssoc '
  classVariableNames: ''
  poolDictionaries: ''  !

!ClassFind class methods ! !

!ClassFind methods !

browse: aPane
"************
*************
          Open a ClassBrowser on the class selected.
          Ensure the selection is not a blank or
          text line."
     | selection className class |
selection := aPane selectedItem.
selection = ''
ifFalse: [
    className := selection asArrayOfSubstrings first.
    className = 'The'
    ifFalse: [
        class := Smalltalk
                  at: className trimBlanks asSymbol
                  ifAbsent: [UndefinedObject].
        class edit]]!
```

```
classList: aPane
"***************
****************
          Answer the contents of the classWindow."
aPane contents: classOutput!

classMenu: aPane
"***************
****************
          Answer a Menu for the class aPane."
aPane setMenu: ((Menu
         labels: 'clear and enter\enter more criteria\return
to last entry' withCrs
         lines: #(0)
         selectors: #(clear getCriteria defaultCriteria))
         title: 'Criteria').!

clear
"****
*****
          Clear ClassFind browser before
          listing new search matches."
classOutput := OrderedCollection new.
self getCriteria!

defaultCriteria
"**************
***************
          Use the previous entry as the search words
          which make up the search criteria."
    |defaultCriteria|
criteria = nil
ifTrue: [
    criteria := ''].
defaultCriteria := criteria.
criteria :=
    Prompter
        prompt: 'Enter Criteria'
```

```
        default: defaultCriteria.
self locateClasses!

getCriteria
"**********
***********
         Obtain the search words which make
         up the search criteria."
criteria :=
    Prompter
        prompt: 'Enter Criteria'
        default: ''.
self locateClasses!

initialise
"*********
**********
         For every method in every class, build the
         searchAssoc dictionary where the key is
         every word in the instance or class method
         name and comment, and the value in every
         class which contains the indexed word.  The
         words are cleaned by removing non-alphabetic
         bytes, and composite words are separated
         into single words, each of which is stored in
         the WordIndex class by its addWord:for:
         method.  The List Pane is opened."
    |classText codeStream word|
super initialize.
classOutput := OrderedCollection new.
Behavior allClasses do: [:eachClass|
    classText := ReadWriteStream on: ''.
```

```
"##################################################
 # For each class, stream all of the source code #
 # for each instance method                       #
 ##################################################"
    eachClass methodDictionary
              keysDo: [:eachMethod|
        codeStream := (eachClass
                          sourceCodeAt: (eachMethod
                                          asSymbol))
                          asStream.

"################################################
 # Stream over the instance method source.      #
 # Substitute a space for non alpha bytes and   #
 # insert a space before upper case characters  #
 ################################################"
        codeStream do: [:char|
            char isLetter
            ifTrue: [
                char isUpperCase
                ifTrue: [
                    classText space].
                classText nextPut: char]
            ifFalse: [
                classText space]]].

"######################################################
 # For each class, the class methods are streamed     #
 # using the instance source, repeated for a          #
 # 10.5 percent performance increase over initiating #
 # a new method twice                                 #
 ######################################################"
    eachClass class
              methodDictionary
              keysDo: [:eachMethod|
        codeStream := (eachClass
                          sourceCodeAt: (eachMethod
                                          asSymbol))
```

```
                    asStream.
        codeStream do: [:char|
            char isLetter
            ifTrue: [
                char isUpperCase
                ifTrue: [
                    classText space].
                classText nextPut: char]
            ifFalse: [
                classText space]]].

"################################################
 # Add all words, except the elementary stopwords, #
 # in self stream described by className string to #
 # the words dictionary                            #
 ################################################"
    classText reset.
    [(word := classText nextWord) == nil]
    whileFalse: [
        (#('a' 'and' 'in' 'of' 'the' 'to') "stopwords"
        includes: word asLowerCase)
        ifFalse: [
            self addWord: word asLowerCase
                for: (eachClass name)]]].
self open.!

locateClasses
"************
*************
        Determine the classes which use the words
        contained in the instance variable *criteria*
        and develop a list for the ClassFind Browser."
    |criteriaStream inAndMode inNotMode lastWordError|
CursorManager execute change.
inNotMode := false.
inAndMode := false.
searchAssoc := Dictionary new.
criteria = nil
```

```
ifTrue: [
    criteria := ''].
criteriaStream :=
    ReadWriteStream on: (criteria asLowerCase
                          asArrayOfSubstrings).

"############################################
 # Stream the search words.  Set inNotMode on #
 # discovering *not* and inAndMode on         #
 # discovering *and*.  Set lastWordError if   #
 # the last word is *and* or *or*             #
 ############################################"
criteriaStream do: [:word|
    lastWordError := false.
    (criteriaStream peek isNil)
    ifTrue: [
        word = 'and'
        ifTrue: [
            word := ''.
            inAndMode := true.
            lastWordError := true].
        word = 'or'
        ifTrue: [
            word := ''.
            lastWordError := true]].
    word = 'or'
    ifFalse: [
        word = 'not'
        ifTrue: [
            inNotMode := true]
        ifFalse: [
            word = 'and'
            ifTrue: [
                inAndMode := true]
            ifFalse: [
                self lookup: word
                    withAndMode: inAndMode
                    andNotMode: inNotMode.
```

```
                inNotMode := false.
                inAndMode := false]].

"########################################
 # Initiate the transferOutput method #
 # when the term has been evaluated   #
 ########################################"
    ((criteriaStream peek isNil)
    or: [((criteriaStream peek = 'and') not
    and: [(word = 'and') not
    and: [(criteriaStream peek = 'or') not
    and: [(word = 'or') not
    and: [searchAssoc notNil
    and: [inNotMode = false]]]]])])
    ifTrue: [
        self transferOutput.
        searchAssoc := Dictionary new]]].
CursorManager normal change.
classWindow contents: classOutput!

lookup: aWord withAndMode: inAndMode
              andNotMode: inNotMode
"**********************************
**********************************
         Determine which classes use each search
         word and undertake boolean operations on
         the classes.  The output is the *searchAssoc*
         dictionary containing the result for each
         boolean operation.  The method
         locateDocuments: is inherited from the
         WordIndex class."
    |classList allClasses dynamicArray value addWord|

"####################################
 # Locate classes containing aWord #
 ####################################"
classList :=
(self locateDocuments:
```

```
(aWord asArrayOfSubstrings)).

"#####################################################
 # If inNotMode, find the complement of the selected #
 # classes from the universe of classes              #
 #####################################################"
inNotMode = true
ifTrue: [
    allClasses := OrderedCollection new.
    Behavior allClasses do: [:class|
        allClasses add: (class name)].
    dynamicArray := classList asOrderedCollection.
    allClasses do: [:class|
        dynamicArray remove: class
                     ifAbsent: [
        dynamicArray add: class]].
    classList := dynamicArray asArray].

"##################################################
 # If inAndMode, AND classList and searchAssoc,   #
 # which is the result to-date of the boolean AND #
 ##################################################"
inAndMode = true
ifTrue: [
    dynamicArray := classList asOrderedCollection.
    classList do: [:class|
        ((searchAssoc keys) includes: class)
        ifFalse: [
            dynamicArray remove: class]].
    (searchAssoc keys) do: [:class|
        (dynamicArray includes: class)
        ifFalse: [
            searchAssoc removeKey: class]].
    classList := dynamicArray asArray].
```

```
"######################################
 # Add aWord to the resultant classes #
 ######################################"
classList do: [:class|
    value := searchAssoc at: class
             ifAbsent: [value := OrderedCollection new].
    addWord := ReadWriteStream on: ''.
    ((value size) > 0)
    ifTrue: [
        inAndMode = true
        ifTrue: [
            addWord nextPutAll: ' and ']
        ifFalse: [
            addWord nextPutAll: ' or ']].
    inNotMode = true
    ifTrue: [
        addWord nextPutAll: 'not '].
    addWord nextPutAll: aWord;
            reset.
    value add: (addWord nextLine).
    searchAssoc at: class put: value]!

open
"***
****
         Create a browser window consisting of a
         List Pane with the menu classMenu for
         entering search criteria and a selection
         facility to open a ClassBrowser."
    | topPane|
Cursor offset: 250@100.
(topPane := TopPane new)
            label: 'ClassFind Browser';
            model: self.
topPane addSubpane:
        (classWindow := ListBox new
         model: self;
         changed: #selectCriteria;
```

```
            framingRatio: (0@0 extent: 1 @ 1);
            when: #getMenu perform: #classMenu:;
            when: #getContents perform: #classList:;
            when: #select perform: #browse:).
topPane openWindow.!

transferOutput
"*************
**************
            Transfer the result of the searchAssoc dictionary
            to classOutput, the List Pane stream."
      |line|
"################################
 # Show the number of responses #
 ################################"
line := ReadWriteStream on: (String new).
line reset;
     nextPutAll: 'The number of responses is ';
     nextPutAll: (searchAssoc keys size printString);
     cr;
     reset.
classOutput add: (line nextLine).

"##################
 # Show the class #
 ##################"
searchAssoc associationsDo: [:classPair|
    line reset;
         nextPutAll: (classPair key).

"################################################
 # Show the search words contained in the class #
 ################################################"
    50 timesRepeat: [line space].
    line position: 25.
    (classPair value) do: [:wordExamined|
        line nextPutAll: wordExamined].
```

```
line cr;
     reset.
classOutput add: (line nextLine)].
line reset;
     cr;
     reset.
classOutput add: (line nextLine)! !
```

Appendix B. Code for the Word Index Class from Digitalk.

The ClassFind search tool inherits methods from the Word Index class, supplied by Digitalk separately from the Smalltalk\V for Windows environment, which is listed below as a result of a Smalltalk File Out operation:

```
ViewManager subclass: #WordIndex
  instanceVariableNames:
    'documents words '
  classVariableNames: ''
  poolDictionaries: '' !

!WordIndex class methods ! !

!WordIndex methods !

addDocument: pathName
        "Add all words in document described by
         pathName string to the words dictionary."
    | word wordStream |
    (documents includes: pathName)
        ifTrue: [self removeDocument: pathName].
    wordStream := File pathName: pathName.
    documents add: pathName.
    [(word := wordStream nextWord) == nil]
        whileFalse: [
            self addWord: word asLowerCase for: pathName].
    wordStream close!

addWord: wordString for: pathName
        "Add wordString to words dictionary for
         document described by pathName."
    (words includesKey: wordString)
        ifFalse: [words at: wordString put: Set new].
    (words at: wordString) add: pathName!
```

```
initialize
        "Initialize a new empty WordIndex."
    documents := Set new.
    words := Dictionary new!

locateDocuments: queryWords
        "Answer an array of the pathNames for
         all documents which contain all words
         in queryWords."
    | answer bag |
    bag := Bag new.
    answer := Set new.
    queryWords do: [ :word |
        bag addAll:
            (words at: word ifAbsent: [#()])].
    bag asSet do: [ :document |
        queryWords size =
            (bag occurrencesOf: document)
                ifTrue: [answer add: document]].
    ^answer asSortedCollection asArray!

removeDocument: pathName
        "Remove pathName string describing a
         document from the words dictionary."
    words do: [ :docs | docs remove: pathName].
    self removeUnusedWords!

removeUnusedWords
        "Remove all words which have empty
         document collection."
    | newWords |
    newWords := Dictionary new.
    words associationsDo: [ :anAssoc |
        anAssoc value isEmpty
            ifFalse: [newWords add: anAssoc]].
    words := newWords! !
```

Appendix C. Code for the ATM.

The demonstration ATM encompasses the Smalltalk classes ATM, Card Reader, User Interface, Account, Deposit Slot, Dispenser and Receipt Printer. They are listed below as a result of Smalltalk File Out operations:

```
ViewManager subclass: #ATM
  instanceVariableNames:
    'cardReader userInterface account depositSlot dispenser
printer state '
  classVariableNames: ''
  poolDictionaries: ''  !

!ATM class methods ! !

!ATM methods !

initialise
"*********
*********
        Establish link to User Interface class."
userInterface := UserInterface new initialise!

printerOutput: aPane
"*******************
*******************
        Answer printer Text Pane contents."
aPane contents: (printer contents)!
```

```
wait: anInteger
"**************
***************
        Wait for anInteger seconds."
    | endTime |
endTime := Time fromSeconds:
          ((Time totalSeconds) + anInteger).
[((Time now) < endTime)] whileTrue: []! !


Object subclass: #CardReader
  instanceVariableNames:
    'pinAndId '
  classVariableNames: ''
  poolDictionaries: ''    !

!CardReader class methods ! !

!CardReader methods !

eject
"****
*****
        Answer true if card ejection complete.
        Always answer true."
^true!

notPIN
"*****
******
        Answer true if card has been kept.
        Always answer true."
    | keepCard |
keepCard := true.
^keepCard!
```

```
readableCardInserted
"*******************
*******************
        Get PIN and customer identification
        number from card."
    | cardRead |
cardRead := self readPinId.
^cardRead!

readPinId
"*********
*********
        Get PIN and customer identification
        number from simulated card entry, which
        must consist of a four digit integer for
        the PIN and a two digit integer for the id."
    | validEntry entry number|
pinAndId = nil
ifTrue: [
    pinAndId := ''].
validEntry := false.
[validEntry = false] whileTrue: [

"#############
 # Get input #
 #############"
    pinAndId :=
        Prompter
        prompt: 'Enter the 4 digit PIN and 2 digit Id'
        default: pinAndId.

"##################
 # Validate input #
 ##################"
    (pinAndId = nil
    or: [
    pinAndId isEmpty])
    ifFalse: [
```

```
        entry := pinAndId asArrayOfSubstrings
                          first.
        number := entry asInteger.
        (number isInteger
        and: [
        entry size = 4])
        ifTrue: [
            entry := pinAndId asArrayOfSubstrings
                              last.
            number := entry asInteger.
            (number isInteger
            and: [
            (entry size = 2)])
            ifTrue: [validEntry := true]]]].
^pinAndId!

unreadableCardInserted
"*********************
*********************
        Answer true if card unreadable.
        Always answer true."
^true! !


ATM subclass: #UserInterface
  instanceVariableNames:
    'screen pin id accountNo amount srcAccount dstnAccount
eject timesValidated '
  classVariableNames: ''
  poolDictionaries: ''  !

!UserInterface class methods ! !

!UserInterface methods !
```

```
account
"*******
*******
        Show selection message."
screen contents: '  ';
       cr;
       nextPutAll: '    Please select the account'.
self screenOutput: screen.!

amount
"******
******
        Show message to enter amount."
screen contents: '  ';
       cr;
       nextPutAll: '   Enter amount of transaction';
       cr;
       cr;
       nextPutAll: '       in whole dollars';
       cr.
self screenOutput: screen!

cancel: aPane
"*************
*************
        Result of *Cancel* button selection.  Show
        cancellation message and abort transaction."
screen contents: '  ';
       cr;
       nextPutAll: '      The transaction has';
       cr;
       cr;
       nextPutAll: '       been cancelled'.
self screenOutput: screen.
self eject.
self wait: 2.
self welcome!
```

```
chequeAccount: aPane
"*******************
********************
        Result of *Cheque* account button selection
        following transaction selection."
accountNo := 1.
"######################
 # Deposit or Withdraw #
 ######################"
(state = 'Deposit'
or: [
state = 'Withdraw'])
ifTrue: [self amount].

"#########
 # Query #
 #########"
state = 'Query'
ifTrue: [self queryAccount].

"############
 # Transfer #
 ############"
state = 'Transfer Source'
ifTrue: [
    srcAccount := 1.
    state := 'Transfer Dstn'.
    screen contents: '  ';
           cr;
           nextPutAll: '    Select the account for';
           cr;
           nextPutAll: '     destination of funds'.
    self screenOutput: screen]
ifFalse: [
    state = 'Transfer Dstn'
    ifTrue: [
        dstnAccount := 1.
        self amount]]!
```

```
deposit: aPane
"*************
**************
        Result of *Deposit* button selection"
state = 'Enter PIN'
ifTrue: [
    state := 'Deposit'.
    self account]!

depositMade: aPane
"*****************
******************
        Result of *Deposit Slot* button selection
        to complete the deposit transaction."
    | actionComplete |
state = 'Deposit'
ifTrue: [
    actionComplete := account depositId: id
                              account: accountNo
                              amount: amount.
    printer contents: actionComplete contents.
    self printerOutput: printer;
         thanks]!

eject
"*****
*****
        Complete a normal transaction - wait:
        is a method of the superclass."
    | complete |
complete := cardReader eject.
complete
ifTrue: [
    eject contents: '  eject'.
    self wait: 3.
    eject contents: '          '.
    ^true]
```

```
ifFalse: [
    ^false]!

enterPIN
"********
********
        Show PIN entry message"
screen contents: '  ';
       cr;
       nextPutAll: '       Please enter your PIN';
       cr;
       cr;
       nextPutAll: '           and press OK';
       cr.!

held
"***
****
        Show held message and close with no card eject."
    | closeSent |
screen contents: '  ';
       cr;
       nextPutAll: ' You must contact your branch';
       cr;
       cr;
       nextPutAll: '      to regain your card'.
self screenOutput: screen.
closeSent := cardReader notPIN.
closeSent ifTrue: [
    self wait: 5.
    self welcome]!

initialise
"**********
**********
        Establish link to Account and Card Reader,
        and build User Interface view."
account := Account new initialise.
```

```
cardReader := CardReader new.
self open;
     welcome!

initWindowSize
"*************
*************
        Window is full size."
^(Display width @ (Display height)).!

insert
"*****
*****
        Determine and validate the entered amount in
        any screen position following the text.  Show
        deposit slot insertion message."
    | actionComplete |
(self validAmount)
ifTrue: [
    amount := screen contents
                     asArrayOfSubstrings
                     last
                     asFloat.
    screen contents: '  ';
           cr;
           nextPutAll: '     Place deposit in slot'.
    self screenOutput: screen]!

invalidPIN
"*********
*********
        Re-enter the PIN up to two additional
        times.  Hold the card if PIN incorrect."
timesValidated := timesValidated + 1.
(timesValidated > 3)
ifTrue: [self held]
ifFalse: [self enterPIN]!
```

```
menu
"****
****
        Show screen based menu.  Note that this is
        not a Smalltalk menu."
screen contents: '  ';
       cr;
       nextPutAll: '     Select action key to';
       cr;
       cr;
       nextPutAll: '      deposit, withdraw,';
       cr;
       cr;
       nextPutAll: '      transfer or query';
       cr;
       selectAtEnd.
self screenOutput: screen!

noFunds
"******
******
        Show insufficient funds message."
screen contents: '  ';
       cr;
       nextPutAll: ' There are insufficient funds';
       cr;
       cr;
       nextPutAll: '  to complete the transaction'.
self screenOutput: screen.
self wait: 3!

okKeyPressed: aPane
"*****************
*****************
        Common use *entry* button."
state = 'Enter PIN'
ifTrue: [self validate].
state = 'Deposit'
```

```
ifTrue: [self insert].
state = 'Withdraw'
ifTrue: [self output].
state = 'Transfer Dstn'
ifTrue: [self store].
state = 'Query'
ifTrue: [self query].

"##################################
 # Prevent Text Pane save message #
 ##################################"
screen modified: false!

open
"***
****
       Create the view of the User Interface."
    | reader |
self
    labelWithoutPrefix: 'Demonstration ATM';
    noSmalltalkMenuBar.
self addSubpane:
     (screen := TextPane new
     changed: #screenOutput:;
     framingBlock: [:box|
        (box leftTop
             rightAndDown: ((box width * 5//100)
                        @   (box height * 5//100)))
         extentFromLeftTop: ((box width * 40//100)
                        @  (box height * 45//100))];
     style: (SubPane noScrollbarsFrameStyle)).
self addSubpane:
     (StaticText new
     centered;
     contents: 'Card Reader';
     framingBlock: [:box|
        (box leftTop
             rightAndDown: ((box width * 18//100)
```

```
                    @   (box height * 58//100)))
        extentFromLeftTop: ((box width * 20//100)
                    @  (box height * 5//100))]).
self addSubpane:
    (reader := GroupPane new        "Card Reader"
    framingBlock: [:box|
       (box leftTop
           rightAndDown: ((box width * 4//100)
                    @   (box height * 64//100)))
        extentFromLeftTop: ((box width * 46//100)
                    @  (box height * 7//100))]).
reader addSubpane:
    (Button new
    defaultPushButton;
    contents: 'readable';
    framingBlock: [:box|
       (box leftTop
           rightAndDown: ((box width * 1//100)
                    @   (box height * 2//100)))
        extentFromLeftTop: ((box width * 34//100)
                    @  (box height * 99//100))];
     when: #clicked perform: #readableCardInserted:).
reader addSubpane:
    (eject := TextPane new
    framingBlock: [:box|
       (box leftTop
           rightAndDown: ((box width * 36//100)
                    @   (box height * 20//100)))
        extentFromLeftTop: ((box width * 28//100)
                    @  (box height * 70//100))];
    style: (SubPane noScrollbarsFrameStyle)).
reader addSubpane:
    (Button new
    defaultPushButton;
    contents: 'unreadable';
    framingBlock: [:box|
       (box leftTop
           rightAndDown: ((box width * 65//100)
```

```
                        @   (box height * 2//100)))
          extentFromLeftTop: ((box width * 34//100)
                        @  (box height * 99//100))];
       when: #clicked perform: #unreadableCardInserted:).
self addSubpane:
     (Button new
     defaultPushButton;
     contents: 'OK';
     idOK;
     framingBlock: [:box|
        (box leftTop
             rightAndDown: ((box width * 21//100)
                        @   (box height * 75//100)))
         extentFromLeftTop: ((box width * 12//100)
                        @  (box height * 8//100))];
     when: #clicked perform: #okKeyPressed:).
self addSubpane:
     (StaticText new
     centered;
     contents: 'Deposit Slot';
     framingBlock: [:box|
        (box leftTop
             rightAndDown: ((box width * 15//100)
                        @   (box height * 85//100)))
         extentFromLeftTop: ((box width * 20//100)
                        @  (box height * 5//100))]).
self addSubpane:
     (Button new          "Deposit Slot"
     defaultPushButton;
     framingBlock: [:box|
        (box leftTop
             rightAndDown: ((box width * 10//100)
                        @   (box height * 90//100)))
         extentFromLeftTop: ((box width * 30//100)
                        @  (box height * 5//100))];
        when: #clicked perform: #depositMade:).
self addSubpane:
     (Button new
```

```
        defaultPushButton;
        contents: 'Cancel';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 47//100)
                           @    (box height * 88//100)))
            extentFromLeftTop: ((box width * 12//100)
                           @   (box height * 8//100))];
        when: #clicked perform: #cancel:).
    self addSubpane:
        (Button new
        defaultPushButton;
        contents: 'Deposit';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 55//100)
                           @    (box height * 5//100)))
            extentFromLeftTop: ((box width * 12//100)
                           @   (box height * 8//100))];
        when: #clicked perform: #deposit:).
    self addSubpane:
        (Button new
        defaultPushButton;
        contents: 'Withdraw';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 55//100)
                           @    (box height * 15//100)))
            extentFromLeftTop: ((box width * 12//100)
                           @   (box height * 8//100))];
        when: #clicked perform: #withdraw:).
    self addSubpane:
        (Button new
        defaultPushButton;
        contents: 'Transfer';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 55//100)
```

```
                        @   (box height * 25//100)))
                extentFromLeftTop: ((box width * 12//100)
                        @  (box height * 8//100))];
        when: #clicked perform: #source:).
self addSubpane:
        (Button new
        defaultPushButton;
        contents: 'Query';
        framingBlock: [:box|
            (box leftTop
                rightAndDown: ((box width * 55//100)
                        @   (box height * 35//100)))
                extentFromLeftTop: ((box width * 12//100)
                        @  (box height * 8//100))];
        when: #clicked perform: #query:).
self addSubpane:
        (StaticText new
        centered;
        contents: 'Account';
        framingBlock: [:box|
            (box leftTop
                rightAndDown: ((box width * 83//100)
                        @   (box height * 8//100)))
                extentFromLeftTop: ((box width * 12//100)
                        @  (box height * 5//100))]).
self addSubpane:
        (Button new
        defaultPushButton;
        contents: 'Cheque';
        framingBlock: [:box|
            (box leftTop
                rightAndDown: ((box width * 83//100)
                        @   (box height * 15//100)))
                extentFromLeftTop: ((box width * 12//100)
                        @  (box height * 8//100))];
        when: #clicked perform: #chequeAccount:).
self addSubpane:
        (Button new
```

```
        defaultPushButton;
        contents: 'Savings';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 83//100)
                            @   (box height * 25//100)))
             extentFromLeftTop: ((box width * 12//100)
                            @  (box height * 8//100))];
        when: #clicked perform: #savingsAccount:).
    self addSubpane:
        (StaticText new
        centered;
        contents: 'Printer';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 55//100)
                            @   (box height * 45//100)))
             extentFromLeftTop: ((box width * 40//100)
                            @  (box height * 5//100))]).
    self addSubpane:
        (printer := TextPane new
        changed: #printer;
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 55//100)
                            @   (box height * 50//100)))
             extentFromLeftTop: ((box width * 40//100)
                            @  (box height * 30//100))];
        style: (SubPane noScrollbarsFrameStyle)).
    self addSubpane:
        (StaticText new
        centered;
        contents: 'Dispenser';
        framingBlock: [:box|
           (box leftTop
                rightAndDown: ((box width * 65//100)
                            @   (box height * 85//100)))
             extentFromLeftTop: ((box width * 20//100)
```

```
                        @  (box height * 5//100))]).
self addSubpane:
    (dispenser := TextPane new        "Dispenser"
    framingBlock: [:box|
       (box leftTop
           rightAndDown: ((box width * 65//100)
                        @   (box height * 90//100)))
        extentFromLeftTop: ((box width * 20//100)
                        @  (box height * 5//100))];
    style: (SubPane noScrollbarsFrameStyle)).
self openWindow.!

output
"*****
******
       Determine and validate the entered amount in
       any screen position following the text, then
       complete the withdrawal transaction."
    | actionComplete |
(self validAmount)
ifTrue: [
    amount := screen contents
                    asArrayOfSubstrings
                    last
                    asFloat.
    screen contents: '  ';
          cr;
          nextPutAll: '   Take money from dispenser'.
    self screenOutput: screen.
    actionComplete := account withdrawId: id
                              account: accountNo
                              amount: amount.
    actionComplete
    notNil ifTrue: [
        printer contents: actionComplete contents.
        self printerOutput: printer.
        dispenser contents: '     money'.
        self wait: 2.
```

```
          dispenser contents: ' '].
     actionComplete
     isNil ifTrue: [
          self noFunds].
     self thanks]!

query: aPane
"***********
***********
          Result of *Query* button selection."
state = 'Enter PIN'
ifTrue: [
     state := 'Query'.
     self account]!

queryAccount
"***********
***********
          Complete query transaction."
     | actionComplete |
actionComplete := account queryId: id
                                account: accountNo.
printer contents: actionComplete contents.
self printerOutput: printer;
      thanks!

readableCardInserted: aPane
"*************************
*************************
          Result of card reader *readable* button selection."
     | pinAndId |
state = ''
ifTrue: [
     timesValidated := 1.
     pinAndId := cardReader readableCardInserted.
     self validate: pinAndId]!
```

```
savingsAccount: aPane
"********************
*********************
        Result of *Savings* account button selection
        following transaction selection."
accountNo := 2.
"#######################
 # Deposit or Withdraw #
 #######################"
(state = 'Deposit'
or: [
state = 'Withdraw'])
ifTrue: [self amount].

"#########
 # Query #
 #########"
state = 'Query'
ifTrue: [self queryAccount].

"############
 # Transfer #
 ############"
state = 'Transfer Source'
ifTrue: [
    srcAccount := 2.
    state := 'Transfer Dstn'.
    screen contents: '  ';
           cr;
           nextPutAll: '    Select the account for';
           cr;
           nextPutAll: '     destination of funds'.
    self screenOutput: screen]
ifFalse: [
    state = 'Transfer Dstn'
    ifTrue: [
        dstnAccount := 2.
        self amount]]!
```

```
screenOutput: aPane
"******************
******************
        Answer screen Text Pane contents."
aPane contents: (screen contents)!

source: aPane
"************
************
        Result of *Transfer* button selection."
state = 'Enter PIN'
ifTrue: [
    state := 'Transfer Source'.
    screen contents: '  ';
           cr;
           nextPutAll: '    Select the account for';
           cr;
           nextPutAll: '       source of funds'.
    self screenOutput: screen]!

store
"****
*****
        Determine and validate the entered amount in
        any screen position following the text, and
        complete the transfer transaction."
    | actionComplete |
(self validAmount)
ifTrue: [
    amount := screen contents
                  asArrayOfSubstrings
                  last
                  asFloat.
    actionComplete := account transferId: id
                              source: srcAccount
                              amount: amount
                              dstn: dstnAccount.
```

```
    actionComplete
    notNil ifTrue: [
        printer contents: actionComplete contents.
        self printerOutput: printer].
    actionComplete
    isNil ifTrue: [
        self noFunds].
    self thanks]!

thanks
"*****
******
        Show thank you message, eject card and show
        welcome message."
    | complete |
screen contents: '  ';
       cr;
       nextPutAll: '     Thank you for banking';
       cr;
       cr;
       nextPutAll: '              with us'.
self screenOutput: screen.
complete := self eject.
complete ifTrue: [
    self wait: 2.
    self welcome]!

unreadableCardInserted: aPane
"****************************
*****************************
        Result of card reader *unreadable* button
        selection."
    | messageSent complete |
state = ''
ifTrue: [
    messageSent := cardReader unreadableCardInserted.
    messageSent ifTrue: [
        screen contents: '  ';
```

```
                cr;
                nextPutAll: '    The card cannot be read';
                cr;
                cr;
                nextPutAll: '           by this ATM'.
        self screenOutput: screen.
        complete := self eject.
        complete ifTrue: [
            self wait: 3.
            self welcome]]]!

validAmount
"**********
***********
        Validate amount entered on the screen
        as a number of whole dollars."
    | entry entryAmount|
entry := screen contents asArrayOfSubstrings.

"####################################
 # Ensure that screen was not deleted #
 ####################################"
entry isEmpty
ifTrue: [
    entry := #(0)].
entryAmount := entry last asFloat.
(entryAmount = 0)
ifFalse: [

"##########################################
 # Answer true if entry is integer or float #
 # with zero cents                          #
 ##########################################"
    ((entryAmount - (entryAmount asInteger)) = 0)
    ifTrue: [^true]].
```

```
"###############################
 # Re-enter if amount invalid #
 ###############################"
self amount;
     screenOutput: screen.
^false

validate
"*******
********
        Validate PIN entry against PIN on card,
        then validate customer identification
        number on card is a valid account."
    | entry enteredPin|
entry := screen contents asArrayOfSubstrings.

"######################################
 # Ensure that screen was not deleted #
 ######################################"
entry isEmpty
ifTrue: [
    entry := #(0)].
(entry last asInteger = 0)
    ifTrue: [
        self enterPIN;
             screenOutput: screen]
    ifFalse: [

"################################
 # Validate entry against card #
 ################################"
        (entry last asInteger = pin)
        ifFalse: [self invalidPIN]
        ifTrue: [
```

```
"#########################
 # Validate id in Account #
 #########################"
            (account check: id)
            ifTrue: [self menu]
            ifFalse: [self held]]]!

validate: aString
"****************
*****************
        Store PIN and id."
    | pinAndId|
pinAndId := aString.
pin := pinAndId asArrayOfSubstrings first asInteger.
id := pinAndId asArrayOfSubstrings last asInteger.
state := 'Enter PIN'.
self enterPIN.!

welcome
"******
*******
        New customer transaction."
state := ''.
"#############################
 # Printer action is complete #
 #############################"
printer contents: ''.
self printerOutput: printer.
```

```
"#######################
 # Show welcome message #
 #######################"
screen contents: '  ';
       cr;
       nextPutAll: '  Welcome to the ATM, please';
       cr;
       cr;
       nextPutAll: '       insert your card'.
self screenOutput: screen!

withdraw: aPane
"**************
**************
        Result of *Withdraw* button selection."
state = 'Enter PIN'
ifTrue: [
    state := 'Withdraw'.
    self account]! !


ViewManager subclass: #Account
  instanceVariableNames:
    'account accountWindow accountContents dispenser
depositSlot printer '
  classVariableNames: ''
  poolDictionaries: ''   !

!Account class methods ! !

!Account methods !

accountMenu: aPane
"****************
****************
         Answer a Menu for aPane."
aPane setMenu: ((Menu
      labels: 'enter or change account balances' withCrs
```

```
        lines: #(0)
        selectors: #(changeBalances))
        title: 'Account Entry').!

changeBalances
"*************
**************
        Enter or modify a numeric customer identification
        number and two account balances."
    | validEntry end id key balances value default |
balances := OrderedCollection new.
validEntry := false.
end := false.

"############################################
 # Get customer id, terminate when no entry #
 ############################################"
[validEntry = false
or: [end = false]]
whileTrue: [
    validEntry := true.
    id :=
    Prompter
        prompt: 'Enter account number'
        default: ''.
    (id = nil
    or: [
    id isEmpty])
    ifTrue: [
        end := true]
    ifFalse: [

"###############
 # Validate id #
 ###############"
        key := id asInteger.
        key isInteger
        ifFalse: [validEntry := false]
```

```
        ifTrue: [
            value := account at: key
            ifAbsent: [
                value := (#(0.00 0.00) asOrderedCollection)].
            default := ReadWriteStream on: ''.
            value do: [:field|
                default nextPutAll: (field printString);
                        nextPutAll: '  '].

"###################################
 # Get cheque and savings balances #
 ###################################"
            balances :=
            Prompter
                prompt: 'Enter cheque and savings account
                       balances'
                default: (default contents).

"#####################
 # Validate balances #
 #####################"
            balances isNil
            ifFalse: [
                (balances asArrayOfSubstrings size = 2)
                ifTrue: [
                    value := OrderedCollection new.
                    (balances asArrayOfSubstrings)
                    do: [:field|
                        field asFloat
                              isNumber
                        ifFalse: [
                            validEntry := false]
                        ifTrue: [
                            value add: (field asFloat)]]]]].
    account at: key put: value.
    self updateContents]]!
```

```
check: aNumber
"*************
**************
        Answer true if aNumber is an Account key,
        otherwise answer false."
    | answer |
answer := ((account keys) includes: aNumber).
^answer!

depositId: anId account: anAccount amount: anAmount
"*************************************************
**************************************************
        Deposit transaction.  Add deposit to
        account balance."
    | actionComplete value balance
      transactionComplete |
actionComplete := depositSlot deposit.
actionComplete ifTrue: [
    value := account at: anId.
    balance := value at: anAccount.
    balance := balance + anAmount.
    value at: anAccount put: balance.
    account at: anId put: value.

"#################
 # Advise printer #
 ##################"
    transactionComplete :=
        printer printout: 'deposited'
                account: anAccount
                amount: anAmount.
    self updateContents.
    ^transactionComplete]!
```

```
initialise
"*********
**********
        Establish link to Dispenser, Deposit Slot
        and Receipt Printer.  Open a Dictionary."
dispenser := Dispenser new.
depositSlot := DepositSlot new.
printer := ReceiptPrinter new.
account := Dictionary new.
self open;
     changeBalances.!

listContents: aPane
"*****************
******************
        Answer account details window contents."
aPane contents: accountContents!

open
"***
****
        Create a window on the account details,
        consisting of a List Box and the menu
        accountMenu for modifying details."
    | line |
self labelWithoutPrefix:
        'Accounts for Demonstration ATM'.

self addSubpane:
        (accountWindow := ListBox new
         changed: #selectCriteria;
         framingRatio: (0 @ 0 extent: 1 @ 1);
         when: #getMenu perform: #accountMenu:;
         when: #getContents perform: #listContents:).
self openWindow.!
```

```
queryId: anId account: anAccount
"*******************************
********************************
       Query transaction to determine balance."
    | value balance transactionComplete |
value := account at: anId.
balance := value at: anAccount.

"#################
 # Advise printer #
 ##################"
transactionComplete := printer query: anAccount
                                amount: balance.
^transactionComplete!

transferId: anId source: aSrcAccount amount: anAmount dstn:
aDstnAccount
"*********************************************************
**********************************************************
       Transfer transaction.  Subtract from source
       account and add to destination account."
    | value srcBalance dstnBalance
      transactionComplete |
value := account at: anId.
"#####################
 # Get source balance #
 #####################"
srcBalance := value at: aSrcAccount.
srcBalance := srcBalance - anAmount.

"#############################
 # Determine sufficient funds #
 #############################"
(srcBalance negative)
ifTrue: [
    transactionComplete := nil]
ifFalse: [
```

```
"##################
 # Transfer funds #
 ##################"
    value at: aSrcAccount put: srcBalance.
    dstnBalance := value at: aDstnAccount.
    dstnBalance := dstnBalance + anAmount.
    value at: aDstnAccount put: dstnBalance.
    account at: anId put: value.

"##################
 # Advise printer #
 ##################"
    transactionComplete :=
        printer transferSrc: aSrcAccount
                amount: anAmount
                dstn: aDstnAccount.
    self updateContents].
^transactionComplete!

updateContents
"*************
*************
        Update the account details window in key order."
    | line count sortedKeys|
accountContents := OrderedCollection new.
accountContents add: 'Id        Cheque        Savings'.
sortedKeys := account keys asSortedCollection.
sortedKeys do: [:id|
    line := ReadWriteStream on: (String new).
    count := 1.
    line nextPutAll: (id printString).
    50 timesRepeat: [line space].
    (account at: id) do: [:balance|
        line position: ((15 * count)
                      - (balance printString size));
            nextPutAll: (balance printString).
        count := count + 1].
    line cr;
```

```
            reset.
        accountContents add: (line nextLine)].
        accountWindow contents: accountContents!

withdrawId: anId account: anAccount amount: anAmount
"*****************************************************
*****************************************************
          Withdraw transaction.  Subtract withdrawal
          from account balance."
        | actionComplete value balance
          transactionComplete |
actionComplete := dispenser withdraw: anAmount.
"###############################
 # Dispenser action completed #
 ###############################"
actionComplete ifTrue: [
        value := account at: anId.

"###############
 # Get balance #
 ###############"
        balance := value at: anAccount.
        balance := balance - anAmount.

"###############################
 # Determine sufficient funds #
 ###############################"
        (balance negative)
        ifTrue: [
            transactionComplete := nil]
        ifFalse: [
            value at: anAccount put: balance.
            account at: anId put: value.

"##################
 # Advise printer #
 ##################"
            transactionComplete :=
```

```
            printer printout: 'withdrawn'
                    account: anAccount
                    amount: anAmount.
        self updateContents].
    ^transactionComplete]! !


Object subclass: #DepositSlot
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''  !

!DepositSlot class methods ! !

!DepositSlot methods !

deposit
"******
*******
        Answer true if deposit action complete.
        Always answer true."
    | actionComplete |
actionComplete := self money.
actionComplete ifTrue: [^true]!

money
"****
*****
        Answer true if parcel inserted.
        Always answer true."
    | moneyInput |
moneyInput := true.
^moneyInput! !
```

```
Object subclass: #Dispenser
  instanceVariableNames:
    'amount '
  classVariableNames: ''
  poolDictionaries: ''   !

!Dispenser class methods ! !

!Dispenser methods !

withdraw: aFloat
"***************
***************
        Answer true if dispenser action complete.
        Always answer true."
amount := aFloat.
^true! !


Object subclass: #ReceiptPrinter
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''   !

!ReceiptPrinter class methods ! !

!ReceiptPrinter methods !

printout: aTransaction account: anAccount amount: anAmount
"*****************************************************
*****************************************************
        Formulate printout message."
    | output |
output := ReadWriteStream on: ''.
output cr;
       nextPutAll: '  The amount of $';
       nextPutAll: (anAmount asInteger printString);
       cr;
```

```
        nextPutAll: '  has been ';
        nextPutAll: aTransaction;
        cr;
        nextPutAll: '  on your '.
anAccount = 1
ifTrue: [
    output nextPutAll: 'cheque'].
anAccount = 2
ifTrue: [
    output nextPutAll: 'savings'].
output nextPutAll: ' account';
        cr;
        reset.
^output!

query: anAccount amount: aBalance
"******************************
******************************
         Formulate query message."
    | output |
output := ReadWriteStream on: ''.
output cr;
        nextPutAll: '  Your '.
anAccount = 1
ifTrue: [
    output nextPutAll: 'cheque'].
anAccount = 2
ifTrue: [
    output nextPutAll: 'savings'].
output nextPutAll: ' account balance';
        cr;
        nextPutAll: '  is $';
        nextPutAll: (aBalance printString);
        cr;
        reset.
^output!
```

```
transferSrc: srcAccount amount: anAmount dstn: dstnAccount
"*******************************************************
*******************************************************
        Formulate printout message."
    | output |
output := ReadWriteStream on: ''.
output cr;
       nextPutAll: '  The amount of $';
       nextPutAll: (anAmount asInteger printString);
       cr;
       nextPutAll: '  has been transferred';
       cr;
       nextPutAll: '  from your '.
srcAccount = 1
ifTrue: [
    output nextPutAll: 'cheque'].
srcAccount = 2
ifTrue: [
    output nextPutAll: 'savings'].
output nextPutAll: ' account';
       cr;
       nextPutAll: '  to your '.
dstnAccount = 1
ifTrue: [
    output nextPutAll: 'cheque'].
dstnAccount = 2
ifTrue: [
    output nextPutAll: 'savings'].
output nextPutAll: ' account';
       cr;
       reset.
^output! !
```