Modelling for Software System Development:

Object-Oriented and Empirical Modelling Perspectives

BY

RUYUAN WANG

THESIS

SUBMITTED TO THE UNIVERSITY OF WARWICK IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR ADMISSION TO THE **MASTER OF SCIENCE**

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF WARWICK

JANUARY 2003

Acknowledgements

I would like to express my sincere appreciation to my supervisor **Meurig Beynon**, who has guided me to understand my research topic with great patience. Thanks to Meurig for his kindest care to my life in UK. Thanks to Meurig for his most thourough help with writing of the thesis.

I also would like to thank Ananda Amatya for his selflessly helping with my research and his constructive feedback.

I am also very thankful to Steve Russ who gave me great help with choosing my research topic, and his emotional encouragement at the time when I was depressed.

Furthermore, I am also indebted to Tuanhui Xue for supplying me with significant material for this research at a crucial time.

Last but not least, my gratitude goes to my parents, my two brothers, and my sister-in-law for their emotional and financial support throughout these years of study. They give me the constant support and encouragement which kept my spirit high up during the hard time of this rigorous research, especially, big brother who fulfils my wish of studying abroad.

Contents

List of Figures, Boxes, and Listings

Declarations

This thesis is presented in accordance with the regulations for the degree of Master of Science. It has been composed by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself except where otherwise stated.

Abstract

Software systems have become ever more complex so that their development has become ever more difficult. An influential opinion on software engineering, expressed by Fred Brooks in his famous book 'The Mythical Man-Month' [Bro95a], regards the problems of software development as being divided into problems of essence and accident. The essential problems of software development – complexity, conformity, changeability, and invisibility – are the very nature of software. This thesis is motivated by two further ideas stemming from Brooks's book: that overcoming the essential problems of software development still presents a major challenge ('there is no silver bullet' [Bro87]), and that modelling does address the essential problems of software development.

The principal aim of the thesis is to describe and compare two approaches to modelling for software development: one is the Vanilla Framework proposed by Harel [Har91], and another is Empirical Modelling [EMWeb]. In modern software engineering, the emphasis has shifted from computer programming to system modelling and development. The latest implementation of Harel's Vanilla Framework – the Rhapsody CASE tool – is discussed in detail, focusing on its mechanisms for visual modelling, model execution and testing, and code generation. A radically different modelling approach, Empirical Modelling, which directs modelling for software system development at the *real-world environment* from which the system emerges, rather than at the *system*, is also introduced and illustrated. This emphasis on real-world modelling is consistent with the original aims of object-oriented programming but we propose EM as a new approach to realizing similar objectives.

Keywords:

Vanilla Framework, Software System Development, Rhapsody, Empirical Modelling, Frameworks, Modelling, Code Generation, Model Execution, Testing, Animation, Design-level Debugging, UML

Abbreviation

DoNaLD	Definitive Notation for Line Drawing
EM	Empirical Modelling
Eden	Evaluator of Definitive Notations
UML	Unified Modelling Language
OMG	Object Management Group
OMD	Object Model Diagram
OO	Object-oriented
OOP	Object-oriented Programming
OOSE	Object-oriented Software Engineering
OXF	Object Execution Framework
CASE	Computer-aided Software Engineering
ISM	Interactive Situation Model
RUP	Rational Unified Process
XP	Extreme Programming
SD	Software Development
VCCS	Vehicle Cruise Control System
OOA	Object-oriented Analysis
OOD	Object-oriented Design
TkEden	Tcl/Tk version of the Eden Interpreter

Chapter 1 Introduction

1.1 Research Motivation and Aims

This research is motivated by an interest in investigating current trends in software development techniques. Today, the computer has become a very important part of people's lives. It can help us to work more efficiently and to solve some problems which we cannot handle without it. People use computers through various kinds of software. Because computers become more powerful every year people expect more and more from them. This means that software is ever more widely used and leads to bigger, and more complex systems. Time is becoming more and more crucial for most enterprises, so we not only expect more benefit from software but also want the time of software development to be much shorter.

1.1.1 Essential and accidental problems in software development – Brooks's 'No Silver Bullet'

About fifteen years ago, Brooks [Bro87] argued that there is "No Silver Bullet" to software development because of the very nature of software. The problems with regard to software development have arguably not gone away to date. Brooks divided software development problems into problems of **essence** and **accident**.

> *All software construction involves essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity, and accidental tasks, the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints.*

The inherent properties of the essence of modern software systems considered by Brooks in [Bro87] are:

- Complexity. The complexity of software is due to three reasons as argued in [Bro87]. Firstly, no two software entities can be the same. Secondly, software system can be in very many possible states. Thirdly, software systems do not scale up easily. The complexity of software cannot be handled in the same way that mathematics and physical sciences simplify complex phenomena because "descriptions of a software entity that abstract away its complexity often abstract away its essence". The complexity of software impedes conceptual integrity and leads to difficulty of communication among team members, difficulty of reuse and extension, and management problems.
- Conformity. There are unlikely to be unifying principles in software engineering such as are sought in physics where there are presumed to be elegant explanations of nature. Making software entities conform to other interfaces is difficult because they are typically designed by different designers.
- Changeability. Software entities are much more subject to change than manufactured things. This is because software can be changed comparatively more easily. The more successful a piece of software is, the more likely it is to be subject to change because people would like to use it in other situations. Also, successful software needs to adapt to new platforms other than those for which it is first written.
- Invisibility. "Software is not inherently embedded in space" so that it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. The diagrams for software structure constitute different diagrams that represent different aspects of software, such as flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. According to Brooks, in spite of progress in restricting and simplifying the structures of software, they remain inherently invisible.

It is argued in [Bro87] that high-level languages, object-oriented programming,

artificial intelligence and expert systems, automatic programming, graphical languages, program verification, and hardware improvements are "non-bullets". They deal only with representation issues, which, in Brooks's opinion, are the accidental part of the problem. The ones that Brooks claims will influence the essence include: buying ready-made software instead of constructing it; using rapid prototyping; refining the design iteratively; finding, hiring, and cultivating extremely talented designers.

A full discussion of responses to Brooks's No Silver Bullet is beyond the scope of this thesis. Our primary interest here is in Harel's optimistic response to Brooks' challenge, and the extent to which Harel's proposals are addressed by an alternative approach to software development based on **Empirical Modelling** (EM) to be introduced later.

1.1.2 'Biting the Silver Bullet' – Harel's Vanilla Framework

In his paper "Biting the Silver Bullet" [Har91], Harel proposed a "Vanilla framework" based on an approach to modelling with visual support, together with powerful notions of executability and code-generation. This framework aims to attack what Brooks calls the essence of the problem of developing complex systems. The main contribution of the Vanilla framework is "to free the programmer from having to think at an inappropriate level of detail, enabling him or her to conceive of an idea for solving an algorithmic problem, and to map it easily from the mind into an appropriate high-level medium". One thing to note is that the Vanilla framework is only aimed at reactive systems, which include embedded, concurrent and real-time systems, but excludes data-intensive ones, such as databases and management information systems.

In [Har91], the main ideas proposed by Harel are the following:

- Modelling systems visually with a set of sufficiently general, rigorous and

precise fundamental notions. The systems are mainly described by two kinds of notion – behavioural descriptions and structural (or architectural) descriptions. The behavioural descriptions describe the hierarchy of control activities that appear on all levels. It is argued in [Har91] that:

> *Control activities play the role of the central nervous system, so to speak, of the conceptual model. They are to sense and control the dynamics of that portion of the functional description that is on their level. This includes the ability to activate and deactivate activities, to cause data to be read and written, and dually, to sense when such things have happened, thus affecting subsequent behaviour. The resulting combination is the system's conceptual model.*

The structural descriptions deal with "such notions as subsystems and modules, channels and physical links, and storage components". They can be thought of as the system's physical model. In the Vanilla framework, "the conceptual and physical models are related by a mapping that assigns implementational responsibility for the various parts of the former to those of the latter".

- Model execution and testing. Executable models are possible when behavioural and structural descriptions such as we have described above are available. Model execution can be performed in different ways according to the users' needs. The model can be executed in a step-by-step fashion that ensures that the users' interaction with a prototype system model is involved in every step to transform the system into a new resulting status. There needs to be a tool to support the model execution and reflect the changing status of the model visually. The model execution can also be done in a batch execution fashion. There also needs to be a tool to read the sequence of events and signals from a prepared batch file and to execute them. Sometimes, we might need to run through all possible scenarios of the system to detect the "unpleasant anomalies such as deadlocks or behavioural ambiguities".
- Code generation. Within the Vanilla framework, code generation refers to translation of an entire conceptual model into a program. The generated code

reflects only those design decisions made in the process of preparing the conceptual model. It does not aim to be the final implementation code although Harel argues that "it is not unreasonable to require such a running version of the system model to be a required deliverable in certain development stages". Hence, the generated code is sometimes referred to as prototype code. It can be used to observe the system performing under circumstances that are close to those of the real world – the actual target environment or simulated one. The code also can be linked to "soft" panels (graphical mock-ups) that represent the actual user-interface of the final system. The graphical mock-ups can be used as part of the standard communication between customer and contractor, or contractor and subcontractor.

1.1.3 Evolution of the Vanilla Framework

From the above description in Section 1.1.2 we can see that two things are necessary to realize the ideas of Harel's Vanilla framework:

- A set of notions to model the system
- A tool to support model execution and code generation

Statecharts have been used as a visual formalism for system modelling [Har86, HG97], and the executable features are supported by the STATEMATE tool [H+00]. STATEMATE uses function-based structured analysis. Since structured analysis methods are considered to be lacking when it comes to certain aspects of system development, such as the transition to design and reuse, many people recommend complementing function-based approaches with ones that are object-oriented. Accordingly, Harel and his colleagues embarked on developing a new set of languages for modelling (still built around the statechart concept), and constructing a supporting tool with model execution and code generation ability. The newest tool of this nature is **Rhapsody**, available at I-Logix. As a result of many years of evolution in software engineering, the Unified Modelling Language (UML) has been adopted

by the Object Management Group (OMG) as the standard modelling language. Statecharts have been included in UML diagrams. The Rhapsody tool supports Harel's Vanilla framework using UML, with particular reference to Executable UML, which includes Statecharts to model system behaviour and Object Modelling Diagrams (OMDs) to model system structure (see Chapter 3 for further discussion). Although the object-oriented approach was not in the Vanilla framework when it was first proposed by Harel in [Har91], it has now become a part of the new framework.

The concept of a framework is central to the achievement of Harel's goals for visualization, executability, testing of designs. To achieve these goals without reuse would be prohibitively expensive. In the absence of such a framework, the task of constructing an executable model that has visualization and supports testing from scratch involves a large overhead in object design and specification. Thus, the central practical work for the realization of Harel's vision is the building of an application framework within which one can conceptualize, capture, and represent a system model. This application framework should also provide underlying support for model execution and code generation. Without this framework, the diagram notions are only pictures having no semantics, and the diagrams cannot become executable. Such a framework offers a form of software reuse. Frameworks not only contain reusable components, but also reflect the best design experience of experts in a certain domain.

1.2 System Modelling

As observed in [BCSW99], in modern software engineering, the emphasis has shifted from computer programming to **system modelling** and development, and the most significant and difficult aspects of the software engineering process precede the explicit specification and generation of code. As we can see from [Har91], the Vanilla framework also focuses on models and modelling. Modelling plays an important role in all kinds of system development. A model gives an abstract view of a system, highlighting certain important aspects of its design and ignoring large

amounts of low-level detail [Pri00]. As a result, models are much easier to understand than the complete code of the system and are often used to illustrate aspects of a system's overall structure or architecture. Software design could almost be defined as the construction of a series of models describing important aspects of the system in more and more detail, until sufficient understanding of the requirements is gained to enable coding to begin. The various different software development methods could also be viewed as modelling methods. The use of models is therefore central to software design, and provides two important benefits that help to deal with the complexity involved in developing almost any significant piece of software. Firstly, models provide succinct descriptions of important aspects of a system that may be too complex to be grasped as a whole. Secondly, models provide a valuable means of communication, both between different members of the development teams, and also between the team and outsiders such as the client. Both software systems and the real-world systems that they are supporting or interacting with tend to be highly complex and very detailed. In order to manage this complexity, descriptions of systems need to emphasize structure rather than detail, and to provide an abstract view of the system.

The principal aim of this thesis is to compare two approaches to modelling for software development. These approaches are radically different but both can be viewed as being in the spirit of Harel's Vanilla framework in some key respects. In the thesis, we shall first examine the application of Harel's Vanilla framework for system modelling of reactive systems in more detail with reference to a case study using the Rhapsody tool, then compare and contrast Harel's approach with modelling using EM principles. The following subsections prepare the ground for this exposition by reviewing the principal characteristics of these two alternative approaches to software development.

1.3 Modelling with Rhapsody

Most recently, OOP has been the kernel of large system developments based on

OOSE in which UML artefacts are the central modelling ingredients. One possible objective of these models is to specify the functionality of software to such a degree that it can be generated semi-automatically. UML models also serve a role in communication between the participants in the software development process. With the birth of UML, various UML supporting tools became available in the tool market. Most of these UML supporting tools can generate some code directly from the corresponding UML diagrams. For example, Rational Rose, and TogetherSoft are very successful in the CASE tools market. However, most of the tools only generate skeleton code, and the developers still need to add more code to make the system executable. It is claimed that Rhapsody can generate production-quality code from UML design models [I-LogixWeb]. Code generation in Rhapsody is a form of reuse of the **Object eXecution Framework** (OXF) it provides. Developing software with Rhapsody makes use of an OO method, UML, and the OXF. The key concepts relevant to Rhapsody will be discussed briefly in the following sections. These respectively deal with Object-Oriented Methods, the Unified Modelling Language, and Rhapsody as a CASE tool.

1.3.1 Object-Oriented Methods

An object-oriented approach to the development of software was first proposed in the late 1960s. However, it took almost 20 years for object technologies to become widely used. Through the 1990s, object-oriented software engineering became the paradigm of choice for many software product builders and a growing number of information systems and engineering professionals. As time passes, object technologies are replacing earlier approaches to software development. The reason is that object technologies do lead to a number of inherent benefits that provide advantages at both the management and technical levels.

From an object-oriented viewpoint, the problem domain is characterized as a set of objects that have specific attributes and behaviours. The objects are manipulated

with a collection of functions and communicate with one another through message passing protocols. An object encapsulates both data and the processing that is applied to the data. This important characteristic enables classes of objects to be built and inherently leads to libraries of reusable classes and objects. Because reuse is a critically important feature of modern software engineering, the object-oriented paradigm is attractive to many software development organizations. In addition, the software components derived using the object-oriented paradigm exhibit design characteristics that are associated with high-quality software.

Practitioners of OO believe that:

- Object-oriented methods lead to software reuse, and reuse leads to faster software development and higher-quality programs.
- Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer.
- Object-oriented systems are easier to adapt and easier to scale. Large systems can be created by assembling reusable subsystems.
- The transition between object-oriented development phases is smooth because the same concept, the object, is used throughout the process.

1.3.2 The Unified Modelling Language

The Unified Modelling Language (UML) is, as its name suggests, a unification of a number of earlier object-oriented modelling languages. It originated as a response to the Object Management Group's (OMG) request for a proposal for a standard object-oriented methodology. The three principal designers of UML are Grady Booch, Jim Rumbaugh, and Ivar Jacobson at Rational Software, and the OMG accepted the UML as the standard in late 1997. It was jointly developed by some other software companies, including Digital, HP, Microsoft, I-Logix etc. and contributions have been made by many object modellers, including David Harel, Peter Coad, and Jim Odell.

The UML attempts to be a unifying notation that incorporates in one generally applicable notation the best of a number of other notations as well as current best practice. Each project might make more or less use of different parts of the UML and some parts may be ignored by different projects. However, the UML should act as a common vocabulary for all object-oriented design projects.

The UML is made of a number of models that together describe the system being designed. Each model comprises one or more diagrams with supporting documentation and descriptions. Each model is intended as a complete description of the system being designed from a particular perspective. The primary diagrams that compromise the UML are nine types of diagrams. The full discussion of UML diagrams is beyond the scope of this thesis. More information about UML can be obtained elsewhere [UMLWeb].

1.3.3 CASE Tools and Rhapsody

Computer-aided software engineering (CASE) tools assist managers and practitioners of software engineering in every activity associated with the software process. They automate project management activities, manage all work products produced throughout the process, and assist engineers in their analysis, design, coding and test work. According to [Dou99], CASE tools can be classified by various criteria, such as their function, their use in various steps of the software engineering process, the environment architecture (hardware and software) that supports them etc. Using function as a primary criterion, CASE tools can be classified into Analysis and Design tools, Programming tools, Integration and Test tools, and Prototyping and Simulation tools.

I-Logix Rhapsody4.0.1[1] is a CASE tool which supports the UML specification. It is

[1] There are different versions of Rhapsody. Rhapsody 4.0.1 is the version that has been used in the author's model development.

a visual modelling and programming environment in Java (and also C and C++). The main application area of using Rhapsody is in Real-Time Embedded (pervasive) systems. Rhapsody has some common features of CASE tools such as keeping the design, code and documentation synchronised, and supports an integrated tools environment to include other tools (configuration management, user interface design and etc.). Rhapsody has been represented as a "complete life-cycle model-based development system" [GHP02]. It can be regarded as supplying a framework for software development similar to the Vanilla Framework proposed by Harel in [Har91], but has more features. The key enabling technologies supported by Rhapsody can be summarized as follows:

- **Model-based design**. The idea behind model-based design is that "a picture is worth a thousand words." By enabling designers to work with visual artefacts at a more abstract level, model-based design allows them to concentrate on the problem they are trying to solve, rather than the code.
- **Model-code associativity**. Most design documentation is separate from the implementation code. Whenever the implementation has been changed, the design documentation needs to be updated. In many cases, the design documentation and the implementation code do not conform to each other. The Rhapsody model/code associativity technique aims to overcome this problem. With this technology, coding is not merely the back-end of the design process – it is an intimate part all the way through. The design model generates implementation code and any change to the source code will be automatically reflected in the model. According to [Gou01], the model and the code are simply different views of the same underlying repository of information about the system, ensuring that the design and code will remain synchronized and consistent throughout the development process.
- **Automated implementation generation**. The core of Rhapsody's support for model executability is its implementation generator. The generator generates fully functional, production-ready implementations, employing all the

behavioural semantics specified in the UML model [GHP02]. The implementation generator maps every model artefact into a set of implementation artefacts based on generation rules and a set of predefined parameters for each model element type or *metaclass*.

- **Implementation framework.** According to [GHP02], the execution framework is an infrastructure that augments the implementation language to support the modelling language semantics. The framework provided by Rhapsody is called the Object Execution Framework. The framework provides a set of architectural and mechanistic patterns to support modelling abstractions like active objects, signal dispatching, state-based behaviours, object relationships and life-cycle management.
- **Debugging.** To achieve all of the benefits of a model-based development process, a product should also be debugged against the model – model-based debugging, or **design-level debugging.** The designer executes the compiled system while a debugger environment provides visual feedback on the dynamic aspects of the system in the context of the design model. In an object-oriented system this consists of highlighting modelling constructs such as states and transitions in Statecharts and capturing inter-object communication in the form of messages and events in Sequence Diagrams. By debugging a design up-front, developers can ensure its correctness as the application is developed, thus saving a tremendous amount of time and energy typically expended during integration and test. Design-level debugging is regarded as an effective technique because it provides the best understanding of the problem and the most rapid solution. However, in some situations, performing source-level debugging is the better approach. Rhapsody's combination of design-level and source-level debugging allows an easier mapping between the UML design and its source code implementation. An important consideration in system development is the availability of the target hardware. Software development often starts before the target hardware is available. Consequently, an effective model-based design and debugging environment must provide a means to debug the software prior to the

target hardware's availability. Rhapsody's real-time framework provides the connection between the application software and the target operating system.

- **Animation**. According to [GHP02], there are two main approaches to model execution. The first, which is called **simulation**, is to "construct an interpreter that executes the [prototype system] model based on the run-time semantics". The other one, called **animation**, is to "provide a run-time traceability link between the implementation execution and the run-time [design] model". Animation involves enabling the application code to communicate with the design environment and display the current status of the behavioural elements of the design model (e.g. statecharts and sequence diagrams) at run-time.
- **Iterative development**. Through model/code associativity and model-based debugging, Rhapsody provides debugging and test capabilities at the design level and enables software development as a true iterative process. An application can be tested and debugged while it is being designed.

1.4 Empirical Modelling

Empirical Modelling is a novel approach to computation developed by Dr Meurig Beynon and his collaborators at the University of Warwick, UK since 1983 [EMWeb]. EM provides foundations for broad areas such as engineering design [BC93, BACY94], computer-based learning [Bey97], business modelling [BRR00, CRB00], and software development [Sun99, Chen02]. EM offers an alternative perspective on computing [CS207] in which the focus is on matters of interpretation and cognition. In this respect, it addresses broader issues than system development, such as are involved in the analysis of environments and requirements that precedes system development [BR95].

One aspect of Empirical Modelling has been the investigation of some fundamental problems in software development which are the central focus in this thesis. The issues currently under investigation in the EM project with regard to software development are: changing modelling context; integration of activities in software

development; fixing the system boundary; methods and development phases; communication amongst participants; and interactive artefacts and testability. Some of the solutions proposed in EM are in spirit close to Harel's original proposals in the Vanilla framework in certain key respects. For example, both emphasize the importance of visualization (and its consequences for communication), executability, and testability. However, the EM approach differs from the Vanilla framework for software development in some fundamental respects, for instance, in the integration of activities in software development, the treatment of the system boundary, and the amethodical manner of development (cf. [TBT00]).

Some key issues that motivate the application of EM in software development will be reviewed in the next section. Following Sun [Sun99], we shall refer to EM models developed for software system development as **Interactive Situation Models** (ISMs).

1.4.1 Topical Issues in Empirical Modelling for Software Development

Changing modelling context

As discussed in Section 1.1, changeability is regarded by Brooks as one of the inherent properties of the essence in software systems. It is pointed out in [BCSW99] that the artefacts, built upon static text and diagrams that are predominantly used in current modelling practice are not easily adapted to essential changes in their use. A typical artefact is intimately linked to its situation in the natural and social world, which we shall refer to as the **modelling context**. Because what is happening in the real world situation is complex and potentially transitory, the artefacts typically need to be changed accordingly. Although much attention has already been drawn to the constantly changing nature of the context in the **development domain**, few models take the context in the **operational domain** into account [Sun99]. It is argued in [Sun99] that "the user who is using the developed system is typically not allowed to

change the system in response to his/her evolving environments", however, "faced with a rapidly changing and radically competitive real world, the user often needs to adapt the system in time in order to cope with diverse situations. It is too late and expensive for the user to achieve this need simply though traditional maintenance".

An ISM is a computer-based model that embodies knowledge that relates to the system under development [SRCB99, Sun99]. This knowledge may be associated with the general characteristics of possible environments in which the system might operate or of particular components (both hardware and software) from which the system might be constructed. When the conception of the system is mature, an ISM may incorporate sufficient knowledge to allow simulation of the system in operation. The use of an ISM helps to resolve the conflict between the constantly changing modelling context and conventionally static system model [SRCB99]. ISMs developed using the principles and tools of EM are intended to complement the traditional artefacts used in systems development, and to address the creative conceptual activities that accompany their construction [BCSW99]. Their construction should be adaptive to the changing modelling context both in the development and the operational phases.

Integration of activities in software development

Traditional modelling methods prescribe the transition from one phase to another, or from one iteration to another (see e.g. [JBR99a]), but do not give guidance to the modellers about how such transitions can be done smoothly. Even the seemingly smooth transition from OOA to OOD is actually not so easy [Kai99]. An ISM provides an environment within which the human interpreter can explore the relationships between observables and the patterns of behaviour associated with a system component, with particular reference to its external real-world semantics [BCSW99], and embody new knowledge or understanding of the real world referent simply by interactively updating the computer model.

Fixing the system boundary

The result of traditional modelling is a system. The definition for 'system' is "a regular interacting or interdependent groups of items forming a unified whole" [m-wWeb] and "a system is a collection of elements or components that are organized for a common purpose" [seaWeb]. From the definition for 'system', it has certain behaviours but is circumscribed for its specific "purpose". As pointed out in [RRR00], "with a traditional approach, the modeller has to preconceive what are going to be the inputs and outputs before starting the construction of the model in order to prescribe the behaviour required. That is, the boundary of the proposed system must be determined in advance." EM does not favour the sharp separation by some form of idealisation or abstraction of the model from the domain that is required for a preconceived system boundary. The system boundary is not the first concern of EM, nor is it essential to declare the system boundary in order for an ISM to serve a useful function, whether in a modelling or a programming role. However, in certain applications of EM in software development, it may be appropriate to declare a system boundary for interaction with an ISM and take this boundary into consideration in transforming the model into a software product.

Methods and Development Phases

The concept of method has been strongly impressed on people's thinking about system development. There is an assumption that the system development is effectively managed and controlled by developers guided by a method or **software engineering process.** However, some people argue that development methods do not really describe ongoing system development practice [TBT00]. Most development methods consist of ordered stages or grouped activities bounded by specific events or completed intermediate products. For example, the partition of development into requirement analysis, design, implementation, and testing phases is used to describe the development steps in many process models, e.g. the Waterfall Model, the Spiral Model [Boe88], and the V-Model [V-ModelWeb]. The widely used iterative

development approach is a spiral model in which each step goes through a complete mini-lifecycle of requirement analysis, specification, design, implementation, and testing. Completion of increment *n* is the starting point for increment *n+1*. It is arguable whether the actual development activity strictly follows this lifecycle. The developer does not do a little bit of requirements analysis, following this with a little bit of specification, design, implementation and testing. The concept of a method assumes a progressive nature of the system development process that rarely permits system development to skip stages. The sequential steps form a causal chain. Although certain steps, for example, implementation are allowed to be repeated several times, this is never before analysis has been completed at least once. As pointed out in [Tru00], this does not reflect practice: "the events that determine the real systems development sequence of activities are extremely complex and opportunistic". Hence, "in fact, the development process is not a sequence at all, and the development activities must proceed in parallel rather than in a strict sequence." As discussed in detail in [Sun99], Empirical Modelling treats software development as an amethodical situated problem-solving activity in which ISMs are developed collaboratively to embody the shared understanding of the participants.

Communication amongst participants

One important use of the artefacts created in modelling is communication between participants in the development process. The modelling artefact may also be used as documentation for reference in future maintenance activity. Therefore, the artefacts should serve to convey all the design information to different participants so far as possible without misunderstanding, and, at the same time, be easy to understand. However, the traditional modelling artefacts, being no more than static text documents and diagrams, are passive, and can easily invite different interpretations by users and developers. This misunderstanding of requirement specifications can lead to huge additional cost in the later stages of system development, or even to the failure of the whole system [STM95]. The interactive situation model proposed by EM can assist participants in exploring, understanding and creating knowledge in the

course of collaborative interaction [Sun99].

ISMs potentially serve as an enabling technology for team development of software in which the sharing of prototypes plays a prominent role in communication between participants. Communication of this nature is one of the motivations behind the recent interest in eXtreme Programming (XP). For this reason, there are similarities between principles that have been explored in EM over the last decade (cf. [BACY94]) and the ideas behind XP. Though XP typically uses orthodox representation techniques based on OO, it does not place the main focus on process description, nor does it follow the analysis-specification-design-implementation-testing lifecycle. XP begins with coding without up-front design. In XP, testing cases are written before coding, and testing is performed always by unit testing and integrated testing. Compared to traditional software development approaches such as the Rational Unified Process (RUP), XP values individuals and interactions **over** processes and tools, working software **over** comprehensive documentation, customer collaboration **over** contract negotiation, and responding to change **over** following a plan. That is, while XP acknowledges value in the second element of each pair, it values the first element more.

Interactive artefacts and testability

In traditional software development, most testing work is done through compiling and running a model. The modeller tests the model with a scenario, and whenever any defects are detected, the model is corrected and recompiled, then retested with the same scenario or a new scenario. Sometimes, the modeller is simply interested in the execution status of the model in a certain state. However, in many cases, the modeller has to run the model step by step again from the beginning to arrive at the target state of the model. There are also some circumstances in which the model will never arrive at the intended state. In [Har91], Harel has suggested several different modes of model execution and testing, including batch execution (see Section 1.1.2). However, these all have a similar change-compile-and-run nature. The possible

interactions with the model are determined by the states and scenarios that can be accessed by running the model.

When an ISM is used to model a system under development, the possibilities for interaction are not limited to those which feature as plausible interactions with the system. In the ISM, the modeller can arbitrarily change the state of the model, and restore the model to its original state without being constrained by the current status of a system execution. This interactiveness of the ISM supports a richer notion of testability of the model than either a conventional approach or Harel's alternative approach (a more detailed discussion of this issue will be given in Section 4.2.4).

1.4.2 Basic Concepts of EM

The key concepts and principles that underlie the application of EM to software development can be summarized as follows. The essence of an EM approach to modelling systems is focusing on **observation**, **dependency** and **agency** to build computer-based artefacts (namely, **interactive situation models**) that the user can interact with, and that can be the basis for communication amongst different participants. Informal evidence primarily derived from student projects at the University of Warwick suggests that models built with Empirical Modelling principles are more easily adapted to a changing modelling context. EM promotes the idea that the way to nurture the growth of a system is through experiment with the model and the real-world situation to which it refers (its 'referent'). There is no strict system boundary concept in Empirical Modelling. The model is always open-ended for further change. EM does not favour methodical development, but promotes a conception of software development as a form of *situated problem solving* that is **amethodical** (cf. [TBT00], [Sun99]). An EM model is an interactive situation model which can have a richer role than a static artefact in communication between all the participants. A key concept in the application of EM to software development is the use of **definitive scripts** to represent state.

The three concepts of observable, dependency and agency are fundamental to EM. They lead to a distinctive approach to using computers as instruments capable of embodying our knowledge of a domain and giving us experiential feedback from models which can be directly compared with the results of interaction with the world. We refine our experience of a real world referent through interaction with it. We gain new insight into its functionality, its essence and its attributes through experimenting with the referent. In addition to the three basic concepts, another significant feature of EM is that it emphasises modelling that is **state-based** instead of **behaviour-based**, where the term 'state' is to be understood as referring to 'state-as-experienced' rather than abstract computational state.

An *observable* is some feature of a system to which a value or status can be attributed in a system state. Observables are the entities that the modellers have chosen to monitor. Observables are subject to refinement through experience with the model. Each observable will have a value (e.g. the speed of the car in the vehicle cruise control simulator) that will change as we view it. New observables can be introduced into a model during interaction with the model.

An *agent* is a family of observables whose presence and absence in system states is correlated in time, that is typically deemed to be responsible for particular changes to observables within the system. All changes to the values of observables within a system are typically construed as due to actions on the part of agents. The observables may be under the control of different agents within the model.

A *dependency* is a relationship between observables. It reflects a kind of immediate and predictable causal link between changes to observables. If the value of one observable changes, the values of other observables will be changed automatically according to the patterns defined by the dependency relationship.

Representing state using definitive scripts is a significant characteristic of EM. A well-known example of a system based on definitive principles is a spreadsheet. A cell that contains a formula causes the system to re-calculate the formula whenever any cell on which it depends is changed, and to display the new value on the screen automatically. As observed in [YY88], in a definitive system, one does not need to remember what has to be updated when certain data have been changed so that the modeller can concentrate on the modelling problems. The dependency relationship will make sure any changes to the dependees (the variables which trigger the change of dependency relationship) will trigger the updates to the dependants. This guarantees the consistency amongst the observables when any changes happen. Another advantage of a definitive system is with regard to human-computer interaction. Some definitive systems can be programmed to echo the up-to-date value of observables on the display as soon as certain variables have been changed by the user. The user can know the result of the change immediately. The on-line interpretation mechanism allows a wide range of interaction and run-time changes when developing new models. Definitions are also good for specifying relationships between observables. The side-effects of updating actions are able to simulate the actions of observables responding to a change in the environment [YY88]. EM uses definitive notations in framing the definitions in scripts.

The tool predominantly used in EM currently is the EDEN interpreter. EDEN stands for *Evaluator of DEfinitive Notations*. EDEN supports definitions expressed in a basic definitive notation that allows variables of type integer, real, string, and list to be defined. EDEN also interprets definitions in two other notations, called DoNaLD (Definitive Notation for Line Drawing) and SCOUT (SCreen layOUT), which are specialized definitive notations for line drawing and screen layout respectively. DoNaLD and SCOUT definitions are both converted into basic EDEN definitions for the underlying definition maintainer to manage. Their syntax makes the task of specifying line drawings and screen window displays far easier than it would be if the user had to use the complex representations that these languages become when

translated. In model building with EDEN, the current state of the model is determined by the collection of definitions that currently reside within the system – a **definitive script**. The user can change the state of one or more observables by entering a new definition, or redefining an existing one. It is also possible to specify **actions** in the form of procedures to be triggered by a change in the value of an observable. Such procedures can be used to automate redefinitions. Eden also supports more general forms of procedural code that can be used to implement conventional state-changing actions such as updating the display and user-defined **functions** for specifying complex dependencies.

1.5 Thesis Outline

Chapter 2 discusses issues relating to application frameworks. Chapter 3 describes Harel's ideas on developing a diagrammatic language for Object-Oriented system modelling and introduces a supporting tool – Rhapsody – which can support Object-Oriented modelling with full executability and code synthesis capability. Chapter 4 elaborates the potential contribution of EM to software system development discussed in section 1.4.1. Chapter 5 gives some further thoughts on modelling. UML modelling is reviewed from an EM perspective. Some difficulties commonly confronted when using an object-oriented method are identified, and the importance of directing modelling for software system development at the *real-world environment* rather than the *system* is emphasised. This emphasis on real-world modelling is consistent with the original aims of object-oriented programming but we propose EM as a new approach to realizing similar objectives. Chapter 6 summarizes the thesis, explains the difficulties and limitations of the research carried out, and gives the author's perspectives on future research in this area.

Chapter 2 Software Development Techniques and Framework

2.0 Overview

Reuse of software has been one of the main goals of software engineering for decades. Most efforts at reusing software resulted in small reusable, blackbox components. In recent years, with the maturity of object-oriented technology, reuse of larger components has become available in the form of **object-oriented application frameworks**.

This chapter mainly discusses issues relating to frameworks. Section 2.1 addresses the question: what is an application framework? The strengths and weaknesses of frameworks are discussed in Section 2.2. Section 2.3 explains the differences and relationship between frameworks, components, and patterns. Different ways of using frameworks are given in Section 2.4. The development of frameworks is discussed in Section 2.5.

Frameworks are an object-oriented reuse technique but in some cases frameworks do not require an object-oriented programming language [FSJ99]. Frameworks like Microsoft Foundation Classes (MFCs) and Microsoft's Distributed Common Object Model (DCOM), JavaSoft's Remote Method Invocation (RMI), and implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) play an increasingly important role in contemporary software development. A framework provides both *reuse of source code and design* [ML01]. The code and design reuse capability of object oriented frameworks enables higher productivity and shorter time-to-market of application development when compared with traditional software systems development. Reuse of design is more important

than reuse of code because a design can potentially be reapplied in more contexts and so is more general. Another motivation for design reuse is that high-level design is the main intellectual content of software, and so is more difficult. Another merit of design reuse is that it is applied earlier in the development process, and so can have a larger impact on a project.

2.1 Concept of Framework

There are different definitions for the term 'framework' [Deu89, JF88]. Rogers [Rog97] defines it from the point of view of its usage: "a **framework** is a partially complete application, pieces of which are customized by the user to complete the application". A description of framework from the point of view of its structure could be "a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact". In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted towards particular business units and application domains, such as user interface and real-time embedded system [FSJ99]. A framework dictates the architecture for the application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that the application designer or implementer can concentrate on the specifics of the application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses that can be put to work immediately.

A framework is a reusable design of a system that describes how the system is decomposed into a set of interacting objects. The framework describes the component objects, how these objects interact, the interfaces of each object, the flow of control between them, and how the system's responsibilities are mapped onto its objects [JF88]. As described early in this section, frameworks are an object-oriented reuse technique, and use all three of the distinguishing characteristics of object-oriented

programming languages: encapsulation, polymorphism, and inheritance. Typically, a framework is implemented with an object-oriented language like C++. Each object in the framework is described by an abstract class. The abstract classes define the interface of the components and provide a skeleton that can be extended to implement the components. Since an abstract class has no instances, it is used as a template for creating subclasses rather than as a template for creating objects. An abstract class also provides the abstract methods or hook methods (in other words, a default implementation) [Pre95]. A concrete class must implement all the abstract methods and any of the hook methods. The abstract classes or methods that must be implemented are called **hot spots**. Hot spots provide the flexibility of a framework. Some features of the framework are not mutable and cannot be easily altered. These points constitute the kernel of a framework, also called the **frozen spots** of the framework. Frozen spots are pieces of code already implemented within the framework that call one or more hot spots provided by the implementer. The kernel will be constant and always present as part of each instance of the framework. A good metaphor for a framework is given in [ML01]: think of a framework as an engine which has many power inlets. Each of these power inlets is a hot spot of the framework. Each hot spot must be powered (implemented) for the engine (framework) to work. The power generators are the application specific code that must be plugged into the hot spots. The added application code will be used by the kernel code of the framework. The engine will not run until all plugs are connected.

2.2 Tradeoffs of Frameworks

Obviously, the purpose and strength of frameworks is reuse. Frameworks are usually domain specific. The domain knowledge and prior effort of experienced developers embodied in frameworks can avoid recreating and revalidating common solutions to recurring application requirements. This can yield to substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability, and interoperability of software.

Frameworks provide different levels of reuse. At the level of code reuse, frameworks provide the component library. At the level of design reuse, frameworks provide reusable abstract algorithms and high-level design that decompose a large system into smaller component. At the level of interface design, frameworks describe the internal interfaces between components. Frameworks also reuse analysis. The experts who are using the same framework naturally tend to describe the systems they want to build in similar ways and divide the systems into the same components. This makes it easier for them to understand each other's design. The consequence of reuse is to save time and money during development. Another benefit brought about by frameworks is the uniformity. For example, graphical user interface frameworks give a set of applications a similar look and feel. Uniformity can make the use of software products easier, and also reduce the cost of maintenance, since maintenance programmers can move from one application to the next without having to learn a new design. However, companies attempting to build and use frameworks need to consider the following aspects of frameworks.

Costs and Development Effort

According to [GHJV95], if applications are hard to design, and toolkits are harder, then frameworks are hardest of all. Frameworks are not applications themselves; they generate applications by customization as stated before. They should provide extensible and reusable facilities to whole domains. It is very difficult to access the requirements that must be met for different clients or to anticipate the future use. Also, higher investment will be needed to employ more experienced staff to develop frameworks. However, the effort of building frameworks can pay for itself through repeated use within the proposed domain. If a framework can be successfully reused again and again, then the costs will decrease sharply.

Integratability

It is common to integrate frameworks together with class libraries, legacy systems, and existing components to fulfil the requirements of application developments. If one develops a framework and expects it to be used, integratability is very important. Frameworks are often abandoned or aborted because they cannot be easily integrated with others. According to [MBF99], there are many problems that are encountered by framework developers when integrating two or more frameworks, such as cohesive behaviour, domain coverage, design intention, lack of access to source code, and lack of standards for the framework.

Learning Curve

Learning to use a framework effectively requires considerable investment of effort. Mentoring or training courses are required to teach developers how to use the framework effectively. For example, according to [FSJ99], it takes 6 to 12 months to grasp framework like MFCs, depending on the experience of the developers. The effort to learn the framework can only be paid off over many projects [FSJ99].

Maintainability

Application requirements change frequently. Therefore, the requirements of frameworks also change inevitably. Framework maintenance activities include modification and adaptation of the framework. The maintenance activities may be at the functional level (certain framework functionality does not fully meet the developers' requirements) or at the non-functional level (e.g. concerned with efficiency or portability). These need to be well documented in order to fulfil the maintenance of frameworks. If a framework is ill documented, few people will use or maintain it over time. As frameworks invariably evolve, the applications that use them must evolve with them, so the maintenance includes both maintenance of frameworks themselves as well as of the applications that use the frameworks.

Flexibility vs. Performance, Generality vs. Simplicity

As described earlier in this chapter, frameworks use object-oriented technology. Extensibility of frameworks is achieved by using inheritance and dynamic binding. These are common features of object-oriented languages. However, flexibility often reduces efficiency. For example, dynamic binding introduces overheads. This makes it necessary for the framework designer to strike a balance between flexibility and efficiency.

The tradeoffs between generality and simplicity in application frameworks are similar to those in the design of a software component – a component that does one thing well and is easy to use can only be used in fewer contexts. A component with many parameters and options can be used more often, but will be harder to learn to use. A framework is more general, and can be used more widely, but the learning curve will be even steeper.

Inversion of Control and Creative Freedom

One important feature of frameworks is the inversion of control. When using a conventional class library, the main body of the application needs to be written first and then made to call the reusable code. When using a framework, the main body is reused and the developer writes the code it calls. According to [FSJ99] "this architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive despatching mechanism". Inversion of control allows the framework to determine which set of application-specific methods to invoke in response to external events. However, this means that the developer loses some creative freedom, since many decisions have been made in the design of the framework and sometimes it is not easy to override them.

Problems with Debugging Frameworks

Frameworks generate applications that have an intertwining of application specific code and frozen code. It is usually hard to distinguish bugs in the framework from bugs in the application code. A debug trace of the application code often leads to framework code. Using single-step debugging method will not work because of this intertwining. Furthermore, with any software development, bugs are introduced into a framework from many possible sources, such as failure to understand the requirements, overly coupled design, or incorrect implementation. When customizing a framework for a particular application, the number of possible error sources will increase.

Lack of Standards

Currently, there are no widely accepted standards for designing, implementing, documenting, and adapting frameworks. The lack of common ground creates a gap between framework developers, extenders, and users. Maintenance and training all require frameworks to be well documented, otherwise, they will be abandoned because of the difficulty to maintain them and learn to use them. Despite the vendors' claims, the interoperability between frameworks may be problematic. The emerging industry standard frameworks, such as CORBA, DCOM, and Java RMI, currently lack the semantics, features, and interoperability required in order to be truly effective across multiple application domains.

Framework with Object-oriented Programming Languages

One of the strengths of frameworks is that they can be represented by normal object-oriented programming languages. This makes it easier for programmers to learn and to apply them. However, this restricts frameworks to systems using those languages. In general, different object-oriented programming languages do not work well together. Hence, it is not cost effective to build an application in one language

with a framework written in another. COM and CORBA address this problem, since they let programs in one language interoperate with programs in another.

2.2.3 Frameworks, Components and Patterns

Frameworks sit in the middle level between components and patterns in terms of reuse. Frameworks are much more abstract, flexible, and customizable than most components, but more concrete and easier to reuse than design patterns.

As discussed before, a framework not only defines reusable components, but more importantly it defines interfaces, and interactions between components. Therefore, frameworks provide not only reuse of code, but also reuse of design. Moreover, frameworks can provide a reusable context for components, and can make it easier to develop new components by making a new component out of smaller components.

Design patterns are more abstract than frameworks. Frameworks can be customized and executed directly. Design patterns need to be studied and implemented each time. Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns. Design patterns are less specialized than frameworks. Frameworks always have a particular application domain. In contrast, design patterns can be used in almost any kind of application.

There are benefits in using patterns and frameworks together. According to [ML01], the patterns used in a framework help to make the framework more comprehensible. Frameworks can be documented by patterns. It has been suggested that one way to flatten the learning curve for frameworks is to understand frameworks in terms of patterns. Furthermore, patterns help make the framework's architecture suitable to many different applications without redesign. On the other hand, some frameworks have been implemented several times, so that they represent kinds of pattern.

2.2.4 Use of Frameworks

There are several ways to use a framework. Some ways require a deeper knowledge of the framework than others. The easiest way to use a framework is to connect existing components without knowing their internal structure. This kind of framework is a **blackbox framework**. Sometimes, one needs to define new concrete subclasses and use them to implement an application. This way of using a framework requires more knowledge about the abstract classes than the first way. The frameworks that rely on inheritance are called **whitebox frameworks**. Blackbox frameworks are easier to learn to use and extend than whitebox frameworks, but whitebox frameworks are often more powerful in the hands of experts. Frameworks that contain both whitebox and blackbox characteristics are called **graybox** frameworks.

No matter whether it is a blackbox, whitebox, or graybox framework, users have to learn how to use it. Learning a framework is harder than learning a regular class library. The classes in a framework are designed to work together, so one has to learn them all at once instead of just one class at a time. Frameworks are easier to learn if there are good documentation and training available. Many documentation guidelines have been proposed for frameworks. For example, there are *cookbooks* that discuss the steps to be followed in developing an application based on a given framework [Mat01, Pre95]. These cookbooks contain many "recipes", which describe informally how to solve specific problems while instantiating the framework.

The best way to start learning a framework is by studying an example illustrative of its use. Examples are concrete and they solve a particular problem. One can study their execution to learn the flow of control inside the framework. They demonstrate both how objects in the framework behave and how programmers use the framework. Ideally, a framework should come with a set of examples that range from basic to advanced, and these examples will exercise the full range of features of the

framework.

Another possible way to start learning a framework is through working with a mentor. Mentoring is actually a popular and effective way of learning in many other software development activities, such as exploring a new process. It is a good idea to use a framework for the first time under the direction of an expert.

2.2.5 Development of frameworks

As in any other application development, framework design is an iterative process. A design is iterated mainly because of the uncertainty of its requirements. This is particularly true for designing frameworks.

The design of a framework starts with domain analysis, which collects a number of examples. The first version of the framework is usually designed to be able to implement the examples and is usually a whitebox framework [Joh97, Rob97]. The framework is then tried on some applications. These applications point out weak points in the framework. Experience leads to improvements in the framework, which often make it more blackbox. At some point, design of framework can be finished and released. When more examples are considered, the framework is more general and reusable.

As described before, changing a framework causes the applications that use it to change, too. Hence, a framework should not be used widely until it has proven itself. Because frameworks require iteration and deep understanding of an application domain, it is hard to create them on schedule. This suggests that they should be developed by advanced development or research groups, not by product groups. On the other hand, framework design must be closely associated with application developers because framework design requires experience in the application domain.

2.2.6 Summary

Frameworks are becoming increasingly common and important. They are the way that object-oriented systems achieve the most reuse. Larger object-oriented applications will end up consisting of layers of frameworks that cooperate with each other. Most of the design and code in the application will come from or be influenced by the frameworks it uses. However, those issues listed above need to be borne in mind when developing and applying frameworks. Also, much further work needs to be done before frameworks become truly powerful and pervasive.

Chapter 3 Case Study with Rhapsody

3.0 Overview

This chapter seeks to describe Harel's ideas on developing a diagrammatic language for Object-Oriented system modelling and a supporting tool – Rhapsody – which can support Object-Oriented modelling with full executablility and code synthesis capability. This diagrammatic language consists of: Object Model Diagrams (OMDs) to model system structure, Statecharts to model system behaviour, and Sequence Diagrams (SDs) as reflective diagrams to validate and debug the applications. A Vehicle Cruise Control System (VCCS) example is used to illustrate the use of the features described above.

Since the object-oriented paradigm became dominant, Harel and his collaborators (see e.g. [HG97]) have embarked on developing a set of diagrammatic languages for object-oriented system modelling, built around statechart, and constructing a supporting tool. Rhapsody is the most recent product of this work: it is a tool available from I-Logix which supports object-oriented modelling with full executability and code synthesis capability. At the core of modelling using Rhapsody (Rhapsody approach will be used in this thesis for short from now on) are two constructive modelling languages, called *Object Model Diagram* and *Statechart*, and a reflective language, *Sequence Diagram* (also called Message Sequence Diagram in [HG97, Har00]). *Object Model Diagrams* specify the structure of a system by identifying object relationships and multiplicities. The relationships between objects include uni-directional association, bi-directional association, inheritance, aggregation, dependency, and composition. The concept of composite object is especially noteworthy in Rhapsody (see Section 3.1.5). A *statechart* attached to a class specifies all behavioural aspects of objects. This visual formalism for representing state has been adopted for specifying behaviour by many methodologies

[B, CD, R+, SGW, SM]. However, in many cases, the dynamic semantics of statecharts is not addressed adequately so that the precise behaviour of models over time is not always well-defined [HG97]. In contrast, the Rhapsody approach uses a strictly defined diagrammatic language set to precisely describe the dynamic behaviour of models over time and can generate executable code from the diagrams [HG97].

Section 3.1 discusses the construction of the VCCS model within Rhapsody. Section 3.2 describes two different communication mechanisms – asynchronous and synchronous – used in Rhapsody and their effects on execution. Section 3.3 gives a brief introduction to how Rhapsody uses the Object Execution Framework (OXF) to generate code. Section 3.4 explains an important feature of Rhapsody – design level debugging - to validate and debug the application using the same graphical notations used to construct the design. Section 3.5 illustrates the essential principles of iterative development using Rhapsody: with reference to the author's preliminary work with the VCCS model. Section 3.6 explains Rhapsody's qualities as an open and extensible development environment and its cooperation with other third-party tools.

3.1 The VCCS Example

This thesis illustrates the Rhapsody approach using a Vehicle Cruise Control System example. The VCCS is a classical case study in software development. Booch [Boo86], Åström and Wittenmark [ÅW84], Shaw [Sha91, Sha95a, Sha95b], Gomaa [Gom00] and others have used the cruise control problem to explore software development methods in computing. As framed by Booch, this problem is:

A cruise control system exists to maintain the speed of a car, even over varying terrain. In Figure 3-1 we see the block diagram of the hardware for such a system.

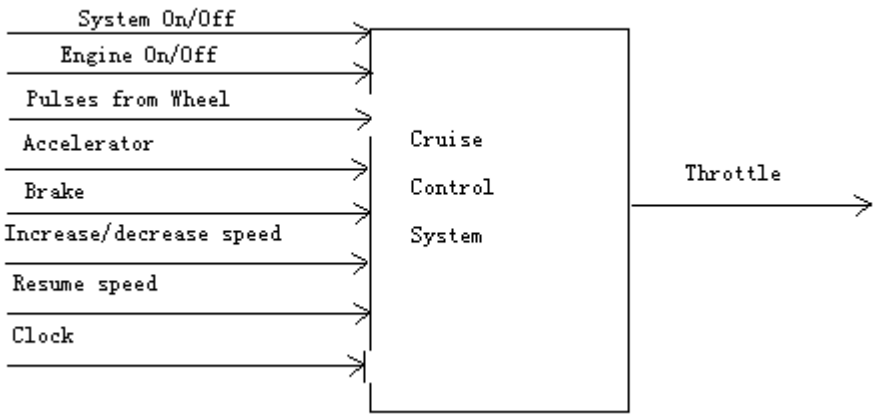


Figure 3-1 Booch block diagram for cruise control

There are several inputs:

System on/off *If on, denotes that the cruise-control system should maintain the car speed.*

Engine on/off *If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.*

Pulses from wheel *A pulse is sent for every revolution of the wheel.*

Accelerator *Indication of how far the accelerator has been pressed.*

Brake *On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.*

Increase/Decrease Speed *Increase or decrease the maintained speed; only applicable if the cruise-control system is on.*

Resume *Resume the last maintained speed; only applicable if the cruise- control system is on.*

Clock *Timing pulse every millisecond.*

There is one output from the system:

Throttle *Digital value for the engine throttle setting.*

The documentation for the author's VCCS, as developed using Rhapsody, is described and discussed in the subsections that follow. Some changes have been made to Booch's model for simplicity and in order to highlight the key features of Rhapsody. For instance, the input `Pulses_from_wheel` and `Clock` are generated by an independent object `Simulator` which simulates both a real

cruising environment and the vehicle speed. The use cases `Preset` and `Set_Preset_Speed` are added as an extension, as shown in Figure 3-2.

3.1.1 Use Case Diagram and Specification

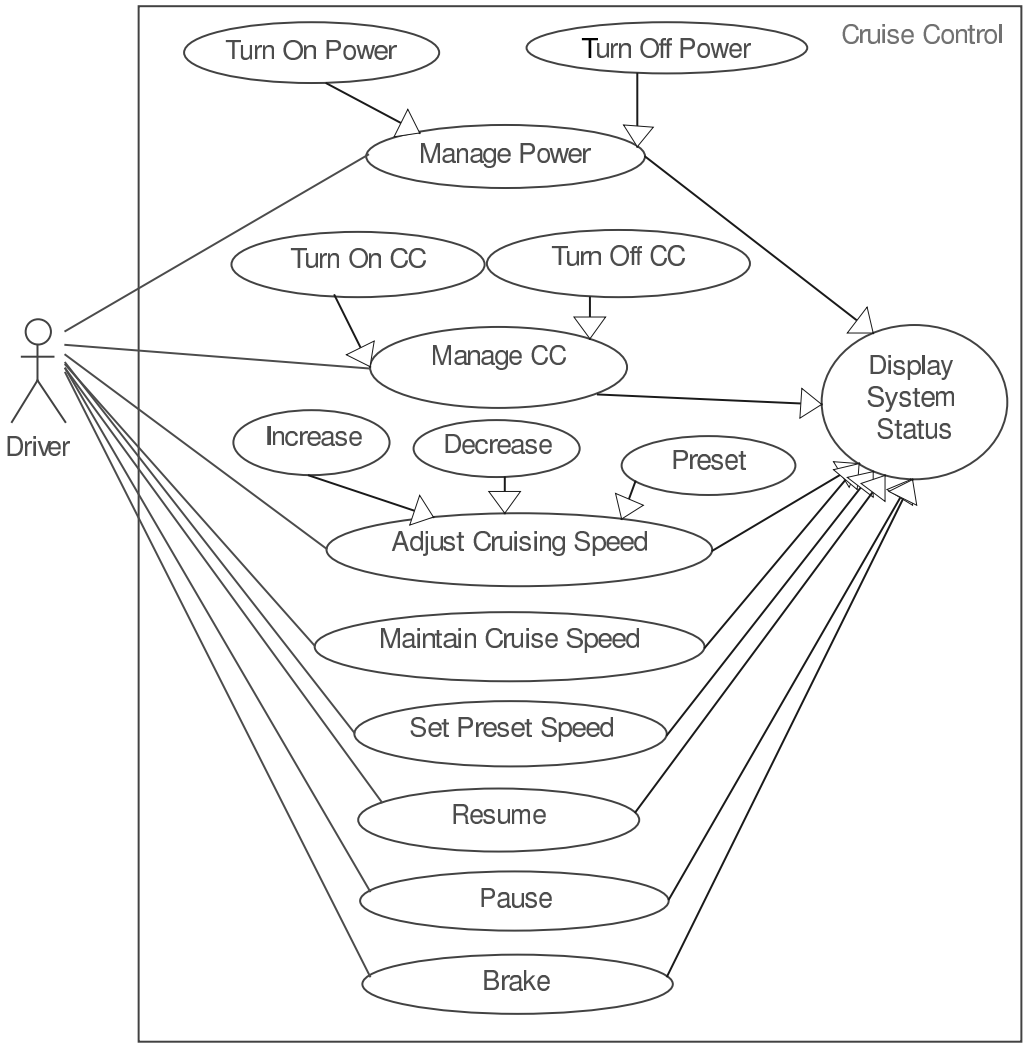


Figure 3-2 VCCS Use Case Diagram

A *Use Case Diagram* shows typical interactions between the system being designed and the external actors who might interact with it. The rectangle represents the system boundary, the stick figures represent the actors, and the ovals represent the use cases. The lines between actors and use cases are associations, allowing the exchange of messages. Use cases may relate to other use cases: for instance, one use

case may be a generalization of another (as depicted by the lines with arrow heads in Figure 3-2).

The use cases in Figure 3-2 are interpreted as follows:

*The primary actor, the **Driver**, uses the system in a number of different ways. First, in order to use any of the other functionality, the driver must turn on the power of the engine. When the power is turned on, the driver can turn on the cruise control system. In this model, after the cruise control system is switched on, the driver needs to set the desired speed. The driver can increase or decrease the current speed using the cruise control. The cruise control can be paused at any time. After pausing, the driver can make the cruise control regulate the vehicle's actual speed again by pushing a "resume" button. When the cruise control system is no longer needed, the driver turns it off.*

1. ***Turn On Power:*** *starts the engine. The Cruise Control is initially off.*
2. ***Turn Off Power:*** *shuts off the engine. The Cruise Control will be turned off as long as Power is off.*
3. ***Turn On CC:*** *activates the cruise control system. This can be invoked only when Power is on.*
4. ***Turn Off CC:*** *deactivates the system to return to manual control. The current cruising speed is cleared from memory without being saved.*
5. ***Increase:*** *increases the current speed and continues to do so until another event occurs or the maximum speed is reached. This can be invoked only when the cruise control is active.*
6. ***Maintain Cruise Speed:*** *maintains the cruise speed using the Vehicle Cruise Control System.*
7. ***Decrease:*** *decreases the current speed and continues to do so until another event occurs or the minimum speed is reached. This can be invoked only when the cruise control is active.*
8. ***Pause:*** *causes the system to stop regulating the vehicle's speed. This can be invoked only when the cruise control system is maintaining cruise speed. The Cruise control is turned off and the control returns to manual. The cruising speed is saved in memory as a speed that may be returned to on resumption (the 'resuming speed').*
9. ***Resume:*** *causes the system to attain the speed stored in the memory (resuming speed) when paused by Pause or Brake. Resume switches the cruise control on.*
10. ***Set Preset Speed:*** *sets preset speed*[2] *to cruising speed. This can only*

[2] A preset speed is used in this model. It is saved in permanent memory and can be changed by the driver. This is for the convenient operation of the driver role. To use the preset speed, the driver only needs to press a button and

be invoked when the cruise control is active.

11. ***Preset:*** *sets the value of preset speed to cruising speed. This can only be invoked only when the cruise control is active.*
12. ***Brake:*** *switches off the cruise control and saves current speed as resuming speed.*
13. ***Display System Status:*** *displays system status, including parameters such as the vehicle speed, cruising speed, preset speed, whether the VCCS is on or off, and the positions of the Accelerator, Throttle and Brake.*

3.1.2 Discovering Classes

An object is a representation of an entity, either real-world or conceptual. An object can represent something concrete or a concept [Qua00]. A universal recipe for finding classes does not exist [Qua00]. As Booch has been known to say of object-oriented analysis, "This is hard!" Different people can come up with completely different subdivisions of objects for the same problem. As argued in [Kai99], OOA objects are generally defined as objects in a problem space, and OOD objects are generally defined as objects in a solution space, so that they are inherently different. As the development goes on, some too complex classes need to be decomposed to make them controllable, some classes might be combined, and some new classes might be added in. From this point of view, the direct match between real-world objects and system objects does not actually exist. The subdivision into objects is often rather arbitrary, and there are many alternative 'equally valid' ways to make this subdivision. Some are indeed discovered from real-world objects through observation, others are conceptual ones which are introduced into a system due to the control needed by system.

3.1.3 Object Model Diagram of the VCCS

Object model diagrams specify the structure and static relationships of the classes and instances in the system. They show the classes, objects, interfaces, and attributes in the system and the static relationships that exist among them. The Rhapsody code

---

the vehicle then automatically attains the cruising speed.

generator directly translates the elements and relationships modelled in OMDs into source code in a number of high-level languages. The Class Diagram and Object Model Diagram in Rhapsody are the same. Harel [Har97] has explained why there is no distinction between these diagrams in Rhapsody as follows:

> *Instances are created during execution by realizing the multiplicities in class definitions and instantiation of composite classes specified using the composite aggregation mechanism.* (This explanation will be elaborated in Section 3.1.5 below).

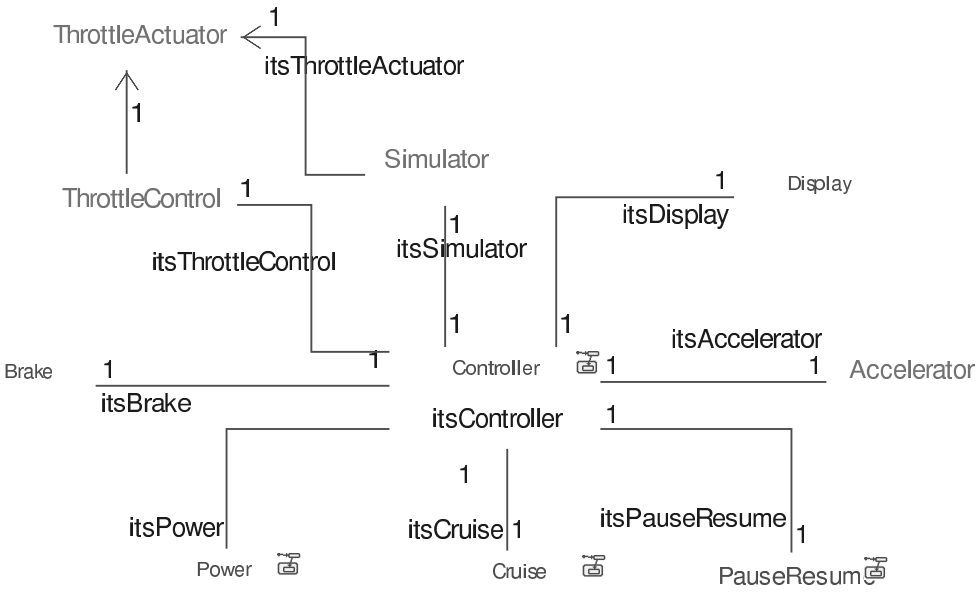


Figure 3-3 VCCS Object Model Diagram

An OMD for the VCCS is given in Figure 3-3. In this figure, each rectangle stands for a class. The lines between classes show their inter-relationships. The name, for instance, `itsBrake` is the *role*[3] *name* for the `Brake` class as referred to by the `Controller`. Similarly, `itsController` is the role name for the `Controller` as referred to by the `Brake`. The designer can specify role names for classes. Rhapsody assigns role names of the form `its<classname>` by default, and they

[3] As defined in [UMLSpec], role is the named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

are used in code generation.

A further issue needs to be clarified in this OMD:

The direction of message passing in the relationship

The direction of a relation is the direction in which participating objects can pass messages to each other. Relations in Rhapsody are of two kinds:

- `symmetric`, meaning that both objects know about and can pass messages to each other. The relation between `Power` and `Controller` is symmetric.
- of the form `from class_A to class_B`, meaning that messages can only pass from instance of class_A to instance of class_B, and not vice versa. In the case of the `ThrottleControl` and the `ThrottleActuator`, the relation is `from ThrottleControl to ThrottleActuator`. The `ThrottleActuator` does not know about the `ThrottleControl`. On the other hand, the `ThrottleControl` knows about the `ThrottleActuator` and can therefore send messages to it.

3.1.4 Class Specification

The `Controller` is responsible for the overall control of the cruise control system and regulates the car's throttle through the `ThrottleActuator`. It receives inputs from `Brake`, `Accelerator` and `Simulator`, which allows it to determine appropriate adjustments to the `ThrottleActuator`. The `Display` is responsible for displaying of current state, such as the cruise status (whether the cruise control is on or off), cruising speed, vehicle speed and preset speed. The `ThrottleActuator` is responsible for adjusting the throttle according to signals it receives from `Controller`. The `ThrottleControl` runs a control thread to maintain the cruise control speed when appropriate. The `Simulator` supplies the VCCS with external parameters such as current wind force, gravitational force, and vehicle speed. The `Power` is the engine in a car: switching on the power in this model is just like starting the engine of a car. The `Cruise` is to enable or disable the

cruise control system. The `PauseResume` is responsible for temporarily halting the cruise control and recovering the cruise control after the pause.

3.1.5 Instantiating Classes and Links[4]

Object model diagrams describe classes and their structure and appear to concentrate on static aspects only. However, when a model is executed, the system consists of instances of classes and actual links which often change dynamically. On this basis, Harel [Har97] points out that:

> *Defining the behavioural semantics specifically enough to enable model execution and full code synthesis hinges on two main things: Initialization and dynamics over time. Initialization concerns the way the model starts out, i.e., which object instances are constructed at the start, and how their attributes and relationships with other objects are initially set up. Dynamics concerns the way the model behaves when running, and it consists of two different kinds of possible changes that can occur in the status of the model: (1) changes in the state of an object, caused by triggering occurrences like events and calls for operations, and (2) changes in the system's structure, i.e., the instantiation and deletion of objects, and the establishment and modification of links between them.*

Rhapsody can initialize classes and relationships in different ways. The initialization can be done at the beginning of system execution or during the execution. This thesis only discusses the former way of initialization.

Before starting the discussion of the initialization, there are two concepts introduced in Rhapsody that need clarification: **Component** and **Configuration**. The settings for these two are relevant to code generation. A **component** in Rhapsody is a physical subsystem in the form of a library or execution program.

[4] In UML, link is the instance of association relationship between two classes.

The name of the component becomes the name of an executable application to build. This component defines classes for which to generate code, and options to apply to the generated code.

Figure 3-4 displays the interface through which Components are configured.

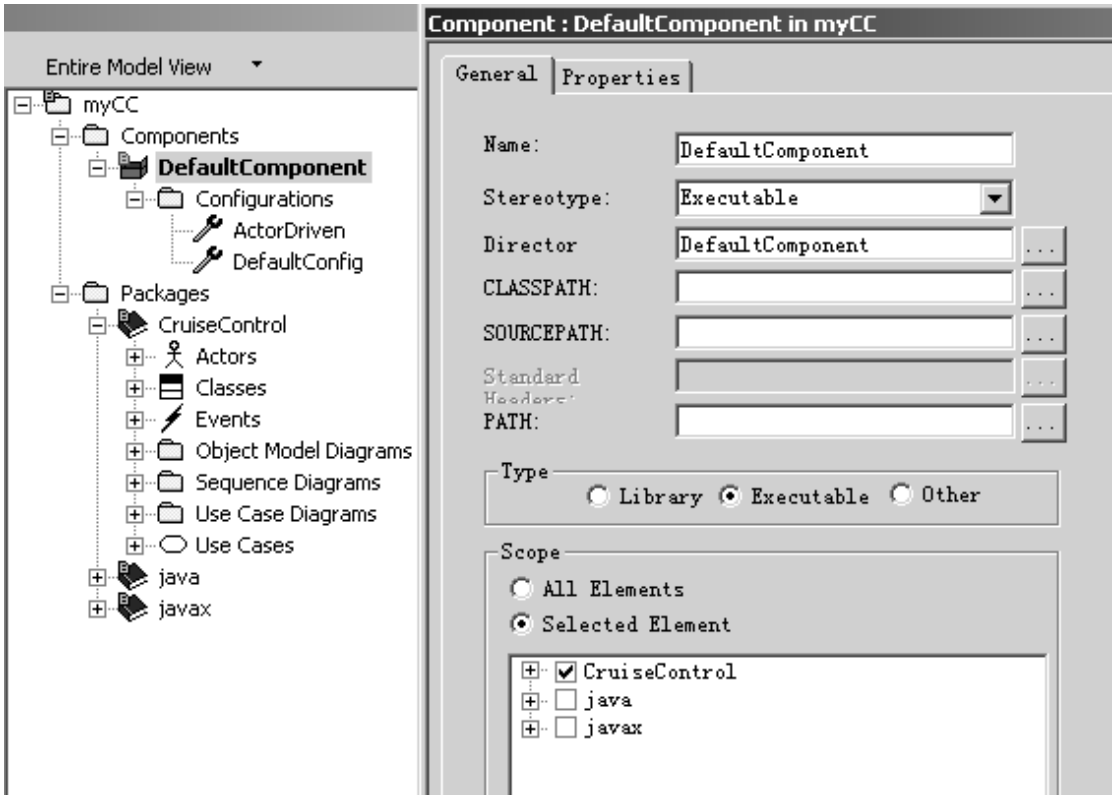


Figure 3-4 VCCS Component in Rhapsody

In Figure 3-4, in the "Scope" area, the box labelled `CruiseControl` is ticked to indicate that Rhapsody is to generate code for the classes in the `CruiseControl` package (these are to be found within the subdirectories `Actors`, `Classes`, etc displayed on the left hand side of Figure 3-4). In this example, the `java` and `javax` packages are also used but there is no code generation for them.

As we can see in the left side of Figure 3-4, the `Components` directory also has a subdirectory called `Configurations`. *Configuration* provides information on

initialization, code checks, and additional settings for code generation. The interface through which this code generation information is supplied is displayed in Figure 3-5. In the 'Initial instances' area, the ticked classes are those for which Rhapsody generates initialization code in the main program. At least one initial instance must be specified to bootstrap the system. In this example, initialization code is generated for all the classes except the `Driver` (the external user of the VCCS). The relationships amongst objects also need to be initialized to make instances communicate with each other during execution.

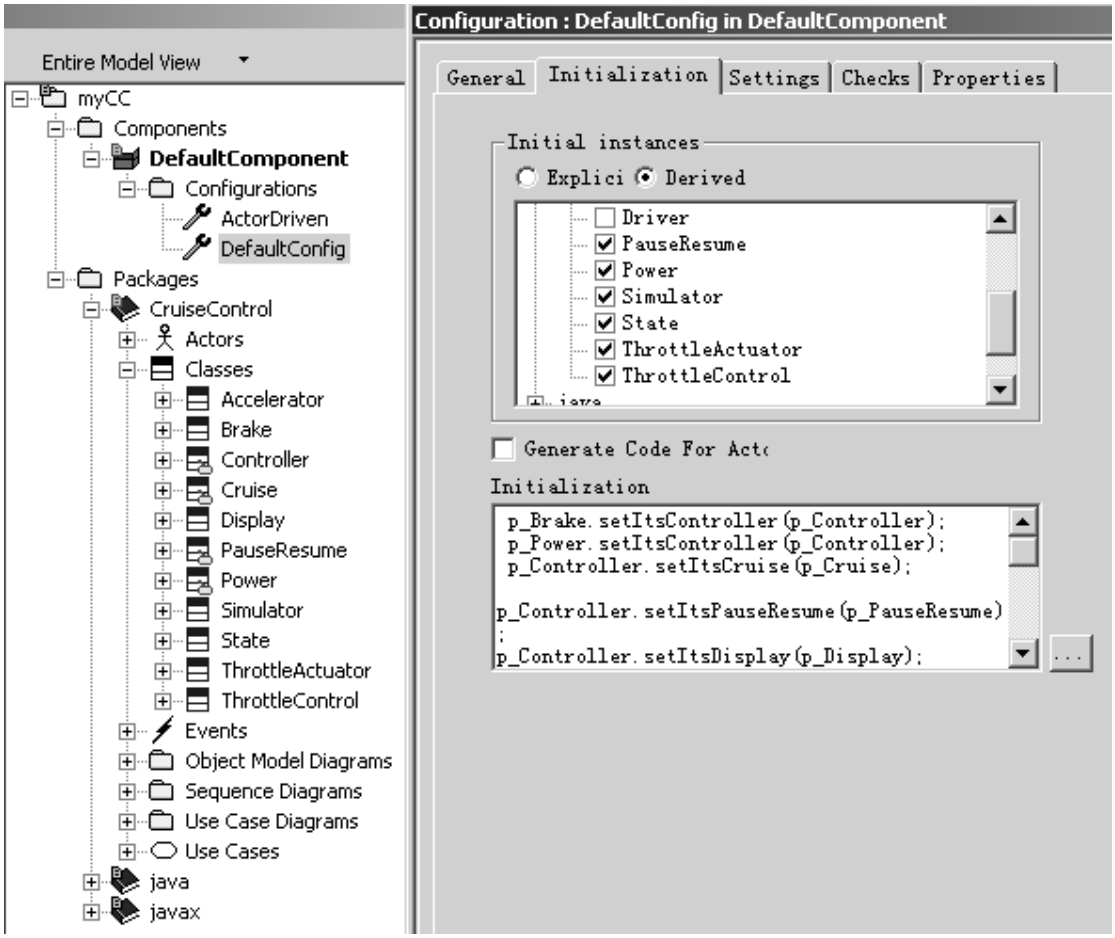


Figure 3-5 Configuration of VCCS in Rhapsody

Rhapsody generates an instance name of the form of <p_Classname> for each class. For example, the instance name of the `Power` class is `p_Power`, and

p_Controller is an instance of class Controller. The Rhapsody syntax for initializing the relationship is then

```
p_Controller.setItsPower(p_Power)
```

After initializing the relationship, the instance of Controller can refer to the instance of Power as itsPower. Therefore, inside class Controller, methods of class Power can be called thus:

```
this.itsPower.<method>(<parameters>)
```

In Rhapsody, if the relationship between classes is symmetric, it is not necessary to invoke initialization in both directions. For instance, since the relationship between Power and Controller is symmetric, the initialization specified above implies the initialization from Power to Controller

```
p_Power.setItsController(p_Controller);
```

To enable convenient reference along relationship links, navigation from one instance to another through intermediate classes is allowed. For example, Power can refer to Simulator using the following navigational expression

```
this.itsController.itsSimulator
```

Another way to initialize classes is using *composite class*, as mentioned before. A composite class is a container class. Instances and relations can be created inside a composite class. Instances in a composite class are called components. As soon as the composite class is initialized, those components will also be initialized. If a relation links instantiated components, Rhapsody initializes the relation at run-time. This saves us effort in writing initialization code.

Figure 3-6 is a UML diagram depicting the VCCS as a composite class consisting of all the classes Controller, Simulator, Brake, Pedal, Power, Cruise, ThrottleActuator, ThrottleControl, PauseResume, Accelerator, Display, inside it. The responsibility of the VCCS class is to instantiate objects and to initialize the relations between them.

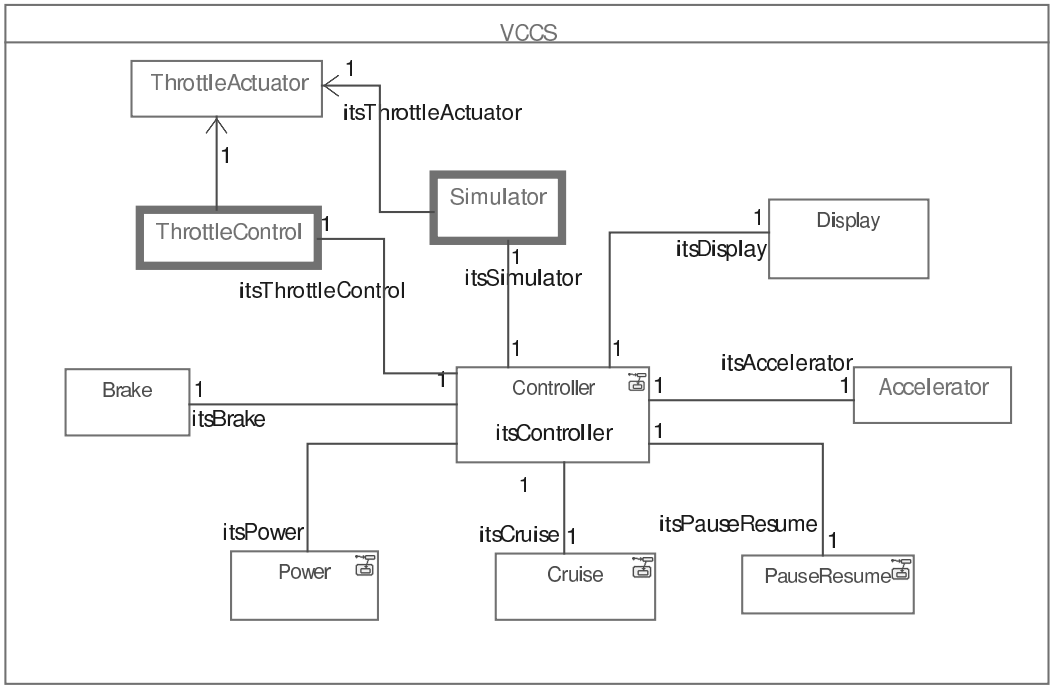


Figure 3-6 Composite Class VCCS

3.1.6 Statechart of the VCCS

Information in the OMDs can only provide part of the initialization and part of the dynamic changes to the model structure. In Rhapsody, statecharts are the central tool used to specify the run-time behaviour of objects, including the various states that an object can enter into over its lifetime and the messages or events that cause it to make a transition from one state to another. At the core of object behaviour are the elementary mechanisms utilized in the statecharts to effect object communication and collaboration. One kind of mechanism is based on generation and sensing of events ('asynchronous event-based communication'), another kind is direct invocation of operations which trigger execution of methods ('operation invocation').

In Rhapsody, an object can generate an event and address it to some other objects. The addresser is called the *client* and addressee is the *server*. The addresser refers to the addressee using a navigational expression as:

`<server>.gen(new <eventname>(<parameters>))`

One special server object to which a client can address an event is `this`, which stands for the client object itself. If we omit the server object in the above expression, `this` is used by default.

Figure 3-7 depicts the statechart for the `Controller` class. This statechart is similar to Gomaa's Hierarchical Cruise Control statechart with activities [Gom00], but the two statecharts specify different VCCS models. Statechart diagrams do not need to be created for every class, only for objects whose states and dynamic behaviour can be evidently described by statecharts.

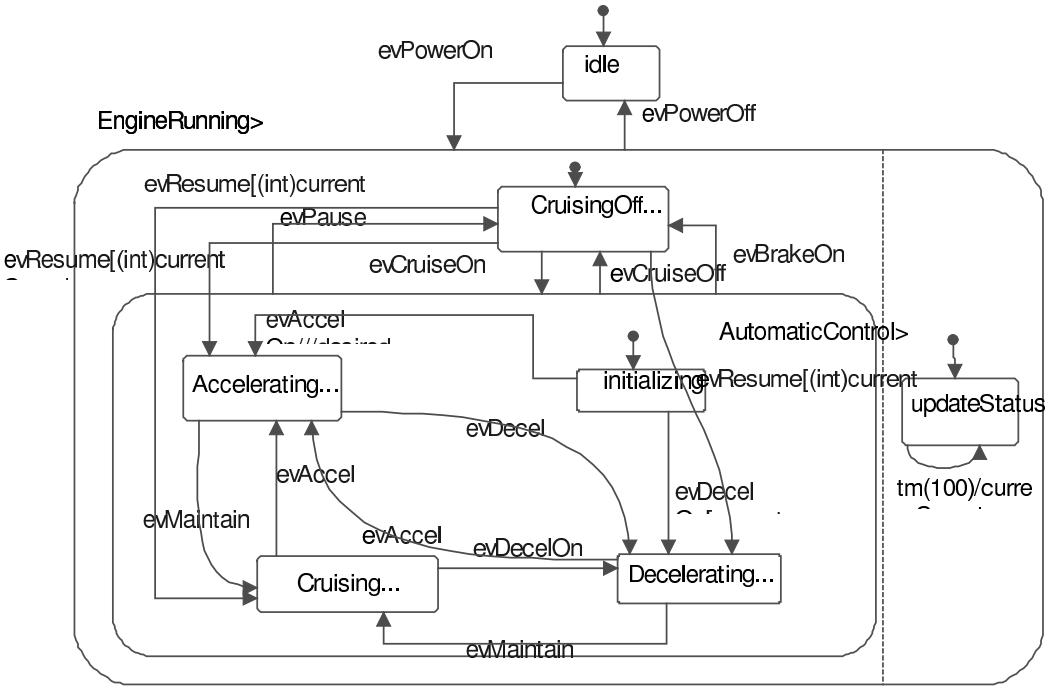


Figure 3-7 Statechart for Controller

Note that not all the information contained in the statechart of `Controller` is displayed in Figure 3-7 for the sake of convenience in presentation. The full information can be found in Appendix I. The same applies to other diagrams in this chapter.

In Figure 3-7, an arrow with a solid dot at one end ( ) represents a *default transition*. The target state of a default transition is the *default state* of the associated class. States are represented by rectangles with round corners, and transitions between states by arrows. Each transition is labelled by a trigger event. Some transitions have *guard conditions* and there may also be actions associated with the transition. The standard form for a transition label is as follows:

```
<Trigger event> [<guard condition>]/ <actions>
```

In this context, "actions" are sequences of event-triggers and operation invocations (see Section 3.2 below), and may also include Java code. The elipses "…" after the name of a state mean that this state has a sub-statechart. A dashed line may be used to partition the states in a statechart in order to indicate that the activities on both sides of the line can be executed concurrently. The annotation `tm(100)` in the above diagram means that the actions written after the slash (/) are executed every 100 milliseconds.

Initially, the statechart is in the `idle` state. When the driver switches on the power of the engine, the `evPowerOn` event occurs and the `Controller` statechart transitions to the `EngineRunning` state. Inside the `EngineRunning` state, the statechart is in the `CruisingOff` state. The driver switches on the cruise control system to trigger the `evCruiseOn` event and the `Controller` statechart transitions to the `AutomaticControl` state. The statechart enters at the `initializing` state as soon as the cruise control system is activated. Then the driver can increase or decrease the vehicle speed by pressing the acceleration or deceleration button. The corresponding event `evAccelOn` or `evDecelOn` event will occur and the statechart transitions to the `Accelerating` or the `Decelerating` state. As soon as the button to maintain cruising speed is pressed, the `evMaintain` event occurs and the statechart transitions to the `Cruising` state. In this state, the cruise control system is used to adjust the vehicle speed as soon as its speed is changed by environmental factors. Such factors include frictional resistance from the road and air drag resistance that depend on the speed of the

vehicle, and gravitational force due to the inclination of the road etc. To simplify the problem of taking these environmental factors into account, in this model a `Simulator` class is used to simulate the positive or negative effects of environmental factors on the vehicle's speed. Pressing the `Pause` button triggers the `evPause` event, hitting the brake triggers the `evBrake` event and switching off the cruise control triggers the `evCruiseOff` event. Transition to the `CruisingOff` state occurs whenever one of these three events is triggered. Pressing the `Resume` button will activate the cruise control system again. If the vehicle's speed is larger than the memorized resume speed then the statechart transitions to the `Decelerating` state otherwise it goes to the `Accelerating` state. If the vehicle's speed is exactly the same as the memorized speed then it will go directly to the `Cruising` state.

Figure 3-8 depicts the statechart of `Power`. The `Off` state is the default state. The event `PowerOn` triggers the transition from the `Off` state to the `On` state, and event `PowerOff` triggers the transition back to the `Off` state.

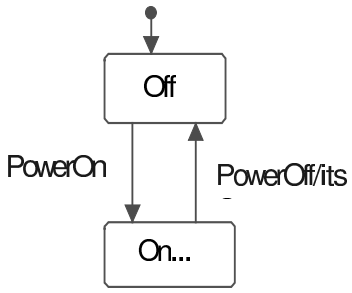


Figure 3-8 Statechart for Power

Figure 3-9 is similar to Figure 3-8. In this statechart (Figure 3-9), `CruiseOn` is the *trigger event*.

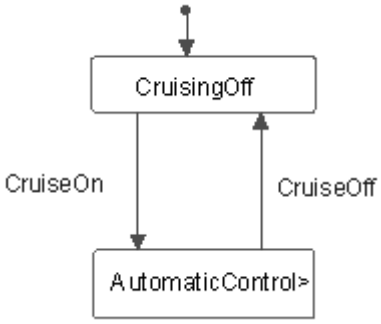


Figure 3-9 Statechart for Cruise

As a representative example of information suppressed in displaying statecharts, note that the guard condition for the event `CruiseOn` is not displayed in Figure 3-9. This is:

```
[Power.status == true]
```

A trigger event with a guard condition means that the transition can only happen when the condition is satisfied. In Rhapsody, there is an `EntryAction` and an `ExitAction` for each state. The `EntryAction` includes actions that should be executed whenever the system enters this state, without regard for how the system arrived here. The `ExitAction` includes actions that should be executed whenever the system exits this state, without regard for how the system exits. In the statechart of the `Cruise` in Figure 3-9, the symbol ">" after `AutomaticControl` means that some information is not shown in the diagram. This is the code for `EntryAction` and `ExitAction`.

EntryAction

```
status = true;

if(Controller.ccSpeed == -1)Controller.ccSpeed = 0;

itsDisplay.cruiseButton.setText("CCOn");

itsDisplay.update();

itsController.gen(new evCruiseOn());
```

ExitAction

```
itsController.itsTrottleControl.haltControl();
status = false;
itsDisplay.cruiseButton.setText("CCOff");
```

```
itsController.gen(new evCruiseOff());
```

In Figure 3-9, as soon as the event `CruiseOn` is addressed to the object `Power`, it addresses `evPowerOn` to the `Controller` using the expression:

```
itsController.gen(new evCruiseOn())
```

This is the syntax used in RhapsodyInJ 4.0.1. Note that `itsController` is the default role name for the `Controller` with respect to `Power` (see Figure 3-3). Inside the body of the class `Power`, the `Controller` can be referred to using this role name. During implementation, `Power` can know all the instantaneous values of attributes in `Controller` (since all the relevant variables are declared to be public). It is as if we had the following code inside the body of the class `Power`:

```
Controller itsController = new Controller (parameters)
```

On this basis, parameter passing does not need to be explicitly specified in Java code. The declaration of a new object is also not necessary, as the information in OMDs can automatically generate the corresponding code. The relations between these objects ensure that the above code can be implemented correctly.

Figure 3-10 is the statechate of `PauseResume`. In this statechart, the initial state is `Idle`. A `Pause` event incurs the transition from `Idle` to `pauseState` and a `Resume` event incurs the transition from `pauseState` to `resumeState`. As soon as the resume speed is attained, the statechart transitions to `Idle`. If the statechart is at `pauseState` at the time the driver switches off the power, it will go back to `Idle` directly from `pauseState`.

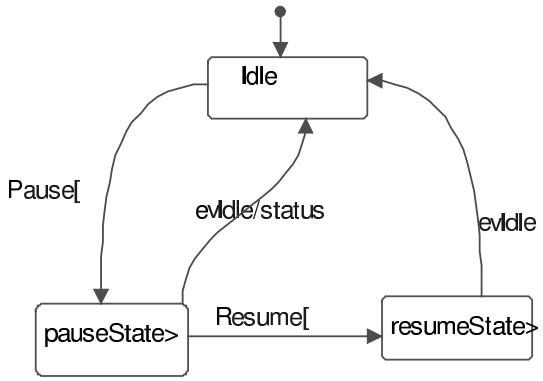


Figure 3-10 Statechart for PauseResume

The solution used in this model only seeks to exemplify model building with Rhapsody. Rhapsody has richer notations to describe object behaviours than have been illustrated here (for more details, see the Rhapsody documents [RhapsodyTutorial, RhapsodyUserGuide]). The full details of the statecharts depicted above can be found in Appendix I.

3.2 Events and Operations

As discussed in Section 3.1, Rhapsody provides two kinds of communication mechanism between objects. One is *asynchronous event-based communication*. The advantage of this mechanism is that it supports client-server relationships, and it frees the modeller from worrying about each and every aspect of sequencing. After an event is sent out, the system's queuing scheduler takes care of passing it on to the server, and the server may deal with it at its own pace. The other mechanism: *operation invocation* involves one object directly causing another to execute a method, by invoking one of its operations. The syntax of invocation is:

```
<server>.<operationname>(<parameters>)
```

This form of communication is synchronous. The thread of control is passed immediately to the called object, which proceeds to execute the relevant method without delay. The client's progress is frozen until execution of the method is completed, at which point it picks up the thread and resumes its work.

Operation invocation is particularly beneficial in cases where the modeller wants close control over sequencing, or where tight synchronization between objects is important. Of the two mechanisms, operation invocation is also the more efficient because the overhead of queuing is avoided, and the translation into an object-oriented programming language is simpler and faster. The event-based mechanism may possibly bring some problems because the modeller does not directly control the sequencing. The following example illustrates the possible problems of the event-based mechanism.

Consider a simple model with two objects `A` and `B` as depicted in Figure 3-11.

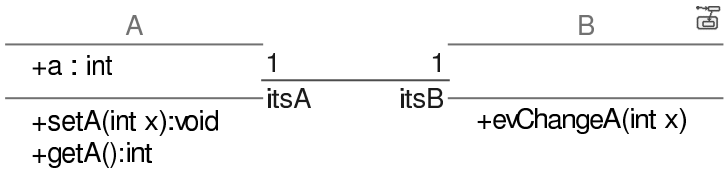


Figure 3-11 Object Model Diagram

In this figure, `a` is an attribute of class `A`, and the symbol "+" indicates public attributes and operations. Class A has two operations, `setA` and `getA`. Class B has an operation which is an event. RhapsodyInJ 4.0.1 allows an event to have its own parameters. Suppose that we add a statechart to Class `B` as in Figure 3-12. The syntax for parameter passing is shown in Figure 3-12.

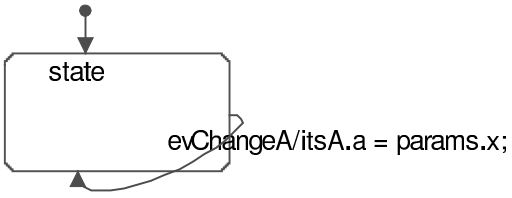


Figure 3-12 Statechart Diagram for Class B

To test the behaviour of the model, we can manually inject events (see more in Rhapsody User Guide) and observe the animated diagrams. It is convenient to

automate the testing, and one way to do this is to implement an 'actor' to drive the model. Figures 3-13 and 3-14 respectively depict the object model and statechart for a `Tester` that can be used to automate a test scenario.

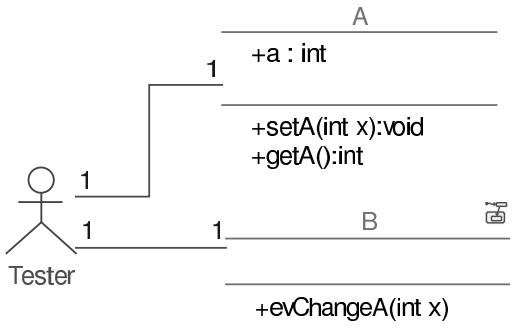


Figure 3-13 Actor Driven Test

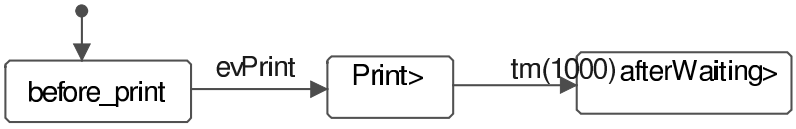


Figure 3-14 Statechart for Actor Tester

The code for the `Entry Action` of the state `Print` is as follows:

```
itsB.itsA.setA(0);

itsB.gen(new evChangeA(5));

System.out.println("a is " + itsB.itsA.getA());
```

The code for the `Entry Action` of the state `afterWaiting` is:

```
System.out.println("a is " + itsB.itsA.getA());
```

The value of the parameter of `evChangeA` is '5' (so that `params.x` is 5 in the context of Figure 3-12), and the value of this parameter is assigned to `a` by `itsA.a = params.x`. Hence, we might expect the result of both print statements in the execution to be "a is 5".

On running the model by generating an `evPrint` event for the `Tester` object, the

following output is produced:

```
a is 0

a is 5
```

From this simple example, we can see that if we are not clear about the sequencing of execution, some unexpected result might be generated. If we do not take account of the asynchronous nature of event execution, we might expect the three statements in the `Entry Action` of `Print` to be executed sequentially, but the actual execution sequence is different. Class `A` delegates an `evChange(5)` event to class `B` and retains the thread of control. For this reason, the third statement is executed. After `A` finishes all the actions, it then passes the thread of control to class `B`. `B` starts acting upon the event `evChange(5)`. After this event has been consumed, the value of attribute `a` is changed to 5.

Another kind of operation, called **triggered operation**, should be mentioned in this connection. It is a synchronous operation implemented by the immediate injection of an event. A triggered operation acts by directly triggering transitions in statecharts. When a caller invokes a triggered operation, a corresponding event is generated and injected directly via the `takeEvent()` method, bypassing its event queue. In the example above, suppose that a triggered operation `changeA (int x)` is added to class `B` and its statechart is changed as follows:

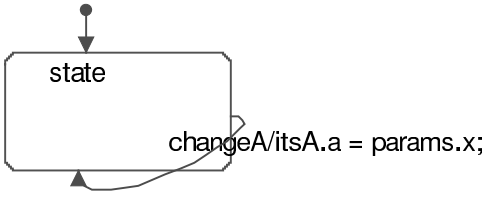


Figure 3-15 Statechart of Class B

Suppose further that we change the second statement of the code in `Entry Action` of state `Print` to:

```
itsB.changeA(5);
```

If we run the model again, we get the expected results:

```
a is 5

a is 5
```

indicating tight synchronization of the message passing between objects `A` and `B`.

3.3 Object eXecution Framework (OXF)

One major benefit of the object-oriented paradigm is reuse. It is common to construct complex systems using predefined frameworks. A framework is a collection of collaborating classes that provide a set of services for a given domain [Boo99]. Frameworks are an open and flexible design; a developer customizes the framework to a particular application by subclassing and composing instances of the framework classes. In Rhapsody, subclassing is done implicitly by drawing classes and setting their properties. For example, if the *concurrency* property of a class is set to *active*, the class automatically is a subclass of the frameworks *OXFThActive* metaclass. If a statechart is added to a class, it is automatically a subclass of *OXFReactive*, which provides the machinery for the execution of state machines.

Code generation in Rhapsody is framework-based. A predefined framework called the *Object Execution Framework*, is used for code generation. The compiled OXF is a run-time library that provides run-time services required by the generated code. It consists of three logical packages: the **behaviour package**, the **operating system package**, and the **service package**. The full discussion of each package is beyond the scope of this thesis. Code generation for the execution of statecharts and OMDs mainly uses the services provided by the *behaviour package* and the *container package* respectively. This thesis will only discuss the services provided by these two packages. The intention is to explain how the model provided above works with this framework. The *behaviour package* consists of a set of collaborative classes that form the fundamental architecture of an object-oriented, reactive, multithreaded system. The *operating system package* is a layer through which the framework and generated code access the operating system services. The *service package* consists of

two subpackages: MemoryManagement and Containers. The *containers package* is a set of template and nontemplate classes used by Rhapsody to implement relationships (association and aggregation) in the application's object model.

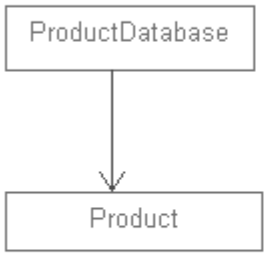
3.3.1 Behaviour Package

An *Active class* in Rhapsody is a class that owns a thread of execution, and has event dispatching and timeout scheduling functionality. It is represented in the framework by `OXFActive`, from which every active class inherits. A *reactive class* is one that has a mechanism for consuming events and triggered operations. It is represented in the framework by `OXFReactive`, from which every reactive class inherits. Every reactive class has an associated event manager, which is an active class. An instance of a reactive class accepts a given event via the operation `gen()`, which queues the event in its associated manager (active class) using `queueEvent()`. The manager will later inject it to the instance for consumption by calling `takeEvent()`. Events are represented in the framework by `OXFEvent`, from which all events inherit. Each event has an association to its target (reactive) object. This allows the reactive's manager (active class) to dispatch the event to the appropriate receiver.

A special kind of event used in Rhapsody is the *Time event*, also called *Timeout*. In Figure 3-7, the event `tm(100)` is such a event. A Timeout is a specialized kind of event represented by `OXFTimeout`. The Timeout class implements timeouts issued by, say statecharts, within reactive classes. Timeout manager (`OXFTimeoutManager`) is responsible for managing the timeout. It runs on a dedicated thread and owns a timer (`OXFSimpleTimer`), which notifies it periodically whenever a fixed time interval has passed. At any given moment, the timeout manager holds a collection of timeouts that should be posted when their time comes. Each time the manager is notified by its timer, it examines the collection and sends the due timeout to the originating object.

3.1.2 Container Package

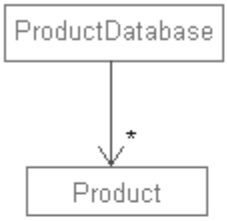
The corresponding code generation for different types of relations between objects is resolved by the *container package* of OXF using *pointers*. When the multiplicity is 0 or 1, the code generator uses a simple pointer to reference an object of the associated class. When the multiplicity is potentially greater than 1, Rhapsody automatically inserts a container class to handle the multiple object instances. An example is given below to illustrate this. Figure 3-16 and Figure 3-17 respectively show the Java code generated by Rhapsody for a relation between `ProductDatabase` and `Product` that has multiplicity 1 and many (*). When the multiplicity for the relation is * (many), Rhapsody generates a *Linked List*, one of the standard Java Containers. Figure 3-18 shows the generated code when multiplicity is * with qualifier[5].




```
//## class ProductDatabase
public class ProductDatabase {

    protected Product itsProduct;

    // Constructors

    public  ProductDatabase() {
    }

    public Product getItsProduct() {
        return itsProduct;
    }

    public void setItsProduct(Product p_Product) {
        itsProduct = p_Product;
    }

}
```

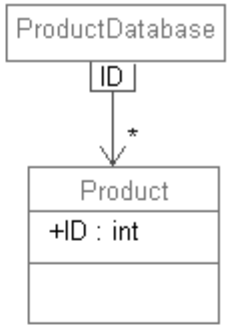
Figure 3-16 Diagram with generated code for ProductDatabase for multiplicity 1

[5] A qualifier is an attribute or list of attributes whose values serve to partition the set of instances associated with an instance across an association. The qualifiers are attributes of the association.




```
//## class ProductDatabase
public class ProductDatabase {

    protected LinkedList itsProduct;

    // Constructors

    public  ProductDatabase() {
        {
            itsProduct=new LinkedList();
        }
    }

    public ListIterator getItsProduct() {
        ListIterator iter=itsProduct.listIterator(0);
        return iter;
    }

    public void addItsProduct(Product p_Product) {
        itsProduct.add(p_Product);
    }

    public void removeItsProduct(Product p_Product) {
        itsProduct.remove(p_Product);
    }

    public void clearItsProduct() {
        itsProduct.clear();
    }

}
```

Figure 3-17 Diagram with generated code for ProductDatabase for multiplicity *




```
//## class ProductDatabase
public class ProductDatabase {
    protected HashMap itsProduct;
    // Constructors
    public  ProductDatabase() {
        {
            itsProduct=new HashMap();
        }
    }
    public Iterator getItsProduct() {
        Iterator iter=itsProduct.keySet().iterator();
        return iter;
    }
    public void clearItsProduct() {
        itsProduct.clear();
    }
    public void removeItsProduct(Product p_Product) {
        Iterator iter=itsProduct.keySet().iterator();
        while(iter.hasNext()) {
          Object key = iter.next();
          if (itsProduct.get(key).equals(p_Product)) {
            itsProduct.remove(key);
            break;
          }
        };
    }
    public Product getItsProduct(int key) {
        return (Product)(itsProduct.get(key));
    }
    public void addItsProduct(int key, Product p_Product) {
        itsProduct.put(key,p_Product);
    }
    public void removeItsProduct(int key) {
        Product p_Product = getItsProduct(key);
        itsProduct.remove(key);
    }

}
```

Figure 3-18 Diagram and generated code for ProductDatabase for multiplicity * with ID as qualifier

3.4 Design Level Debugging and Testing

Rhapsody provides *Design-Level Debugging*. It enables us to validate and debug an application using the same graphical notation that is used to construct the design. Whilst the generated application runs, Rhapsody provides dynamic feedback to the design model. For example, during run-time, animated UML diagrams show the current state of operation of Statecharts, messages in Sequence Diagrams being

passed from one component to another, instances being dynamically created and destroyed, and the values of attributes changing. The design-level debugging provided by Rhapsody aims to reduce the development cycle time and to improve design quality by allowing the developer to identify fundamental design problems early where they are less costly to fix.

3.4.1 Animator

The Animator is used to create test scripts to apply external test stimuli to the system. Animated model executing can be watched in any of the following views: `Sequence diagrams`, `Browser`, `Statecharts`, `Event Queue window`, `Call Stack window`. The full discussion of these views is beyond the scope of this thesis. The first three views will be discussed for illustration.

The *Animator* helps to debug the system at the design level rather than the source-code level by actually executing the model and animating the various UML design diagrams. Rather than merely simulating the application and viewing values of variables and pointers, actual values of instances of states and relations are shown. Simultaneously viewing animated sequence diagrams, animated statecharts, and the animated browser in adjacent windows as the model is executing enables us to verify that the design behaves as intended. Highlighting in the animated diagrams helps to pinpoint the current state of execution. While the model is running, a *control panel* allows us to step through the program, set and clear breakpoints, and inject events to observe how the system reacts in quasi-real time. The system's operation can be observed either in the animated views or by generating an output trace.

- Animated sequence diagrams — These depict messages actually passed between instances during the execution of the application.

  Sequence diagrams are used to show the communications between the objects over time. They are always associated with use cases in the UML

and show a scenario of the corresponding use case. In Rhapsody, they can also serve as reflection diagrams [HG97] to validate and debug the application. During execution, Rhapsody automatically constructs animated sequence diagrams, showing the dynamics of object interaction as they actually happen during execution.

- Animated statecharts—They describe the current states and latest transitions of the object states.

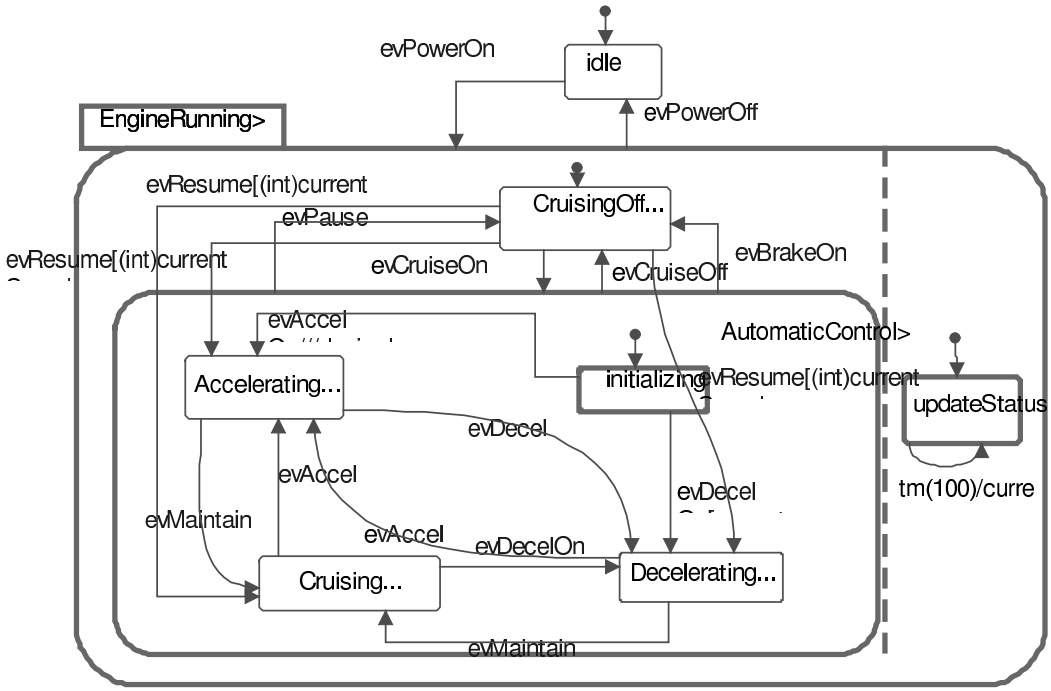


Figure 3-19 Animated statechart of Controller

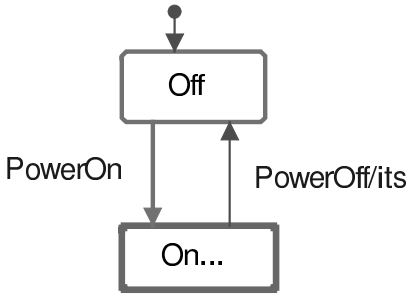


Figure 3-20 Animated statechart of Power

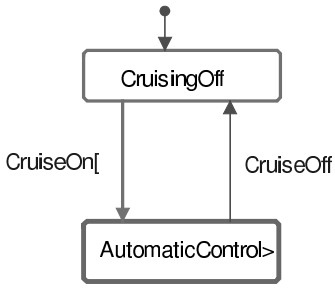


Figure 3-21 Animated statechart of Cruise

Figures 3-19, 3-20 and 3-21 depict the animated statecharts as they appear when both `PowerOn` and `CruiseOn` are invoked. As described in the use case specification in Section 3.1.1, `CruiseOn` can only be invoked when `PowerOn` is activated. This condition can be tested either with reference to Figure 3-19 or to Figures 3-20 and 3-21 in combination. In Figure 3-19, the `Controller` is operating in the `AutomaticControl` mode and with `updateStatus` active within the `EngineRunning` context, and it is currently in the `initializing` state. In Figure 3-20 and Figure 3-21, `Power` is `On` status and `Cruise` is `AutomaticControl` status. From such an animation of statechart we can not only view the current state of objects but can also informally assess whether the system execution meets its requirements.

- Animated Browser—This allows the inspection of instances currently alive in the application.

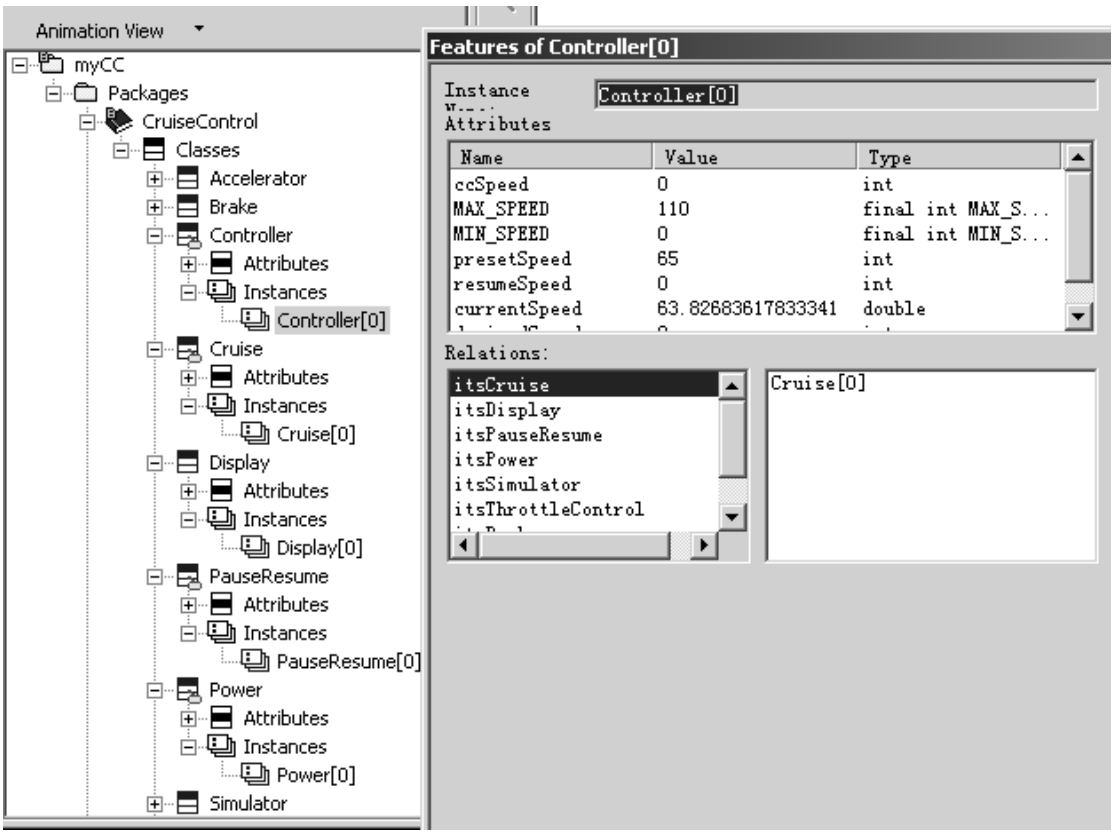


Figure 3-22 Animation of Browser

Figure 3-22 shows the animated browser display as it appears at the time depicted in the animations in Figures 3-19, 3-20 and 3-21. The left side of Figure 3-22 shows the object instances and the right side shows their attributes and their current values and relations. Instance names have the format `classname[n]`, where `n` is greater than or equal to 0. For example, the first instance of the `Brake` class is `Brake[0]`.

3.4.2 Automating the Tests

Rhapsody has an attractive way to test and debug the model against the requirements. To illustrate this, we return to the simple example considered in Section 3.2 where we studied the synchronization of message passing between two objects `A` and `B`. Suppose that Figure 3-23 (‘diagram1’) describes our requirements for the model execution.

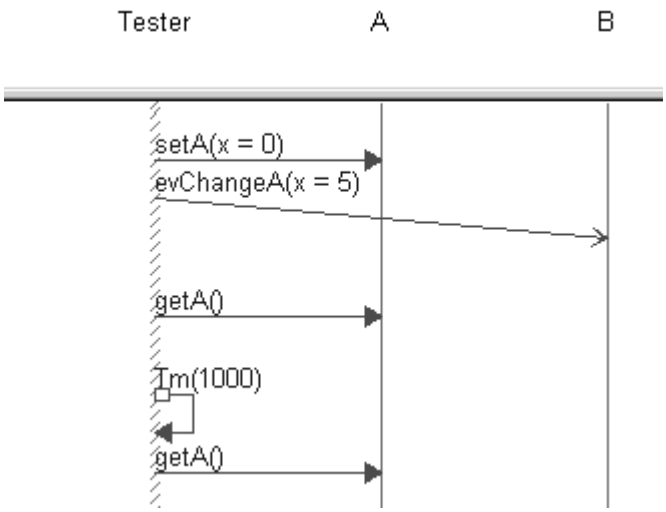


Figure 3-23 SD Describing Message Passing Between Class Tester, A and B

In Figure 3-23, each vertical "instance line" represents an object. The horizontal or slanted directed lines are messages. Messages can be either operation invocations or events. Each message line starts at the originator object and ends at the target object and has a message name on the line. Horizontal directed lines indicate that the message passing is synchronized whilst slanted directed lines indicate asynchronous message passing. The timeout arrow represents a time limit on how long an object can wait before receiving the next message.

During execution, Rhapsody automatically constructs animated sequence diagrams. One possible diagram that can be generated by Rhapsody in this way is depicted on the left side of Figure 3-24 below ('diagram2'). When the execution is completed, we can ask Rhapsody to compare diagram1 and diagram2 to highlight any inconsistencies, such as an event appearing in one diagram but not in the other. Figure 3-24 shows the result of such a comparison. On the left side is the sequence diagram constructed by animation and on the right side is the sequence diagram used as a specification.

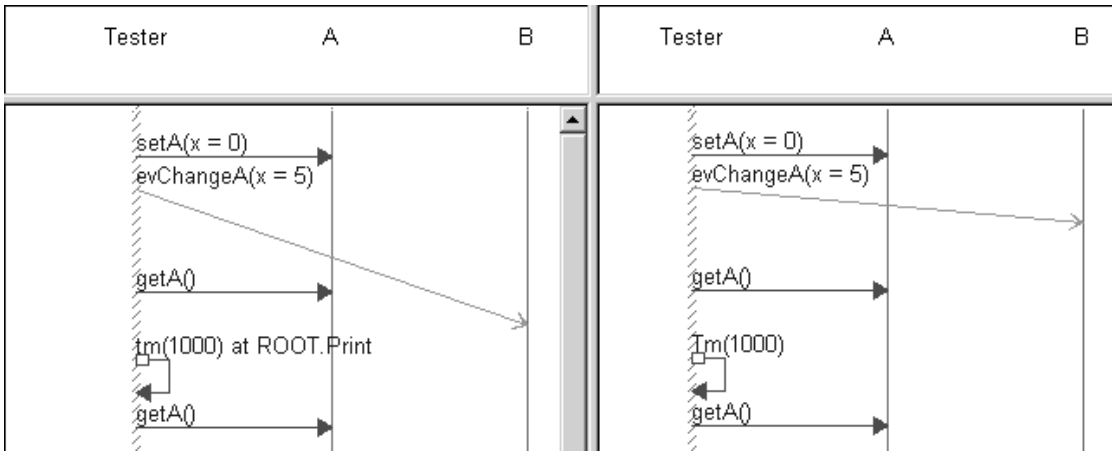


Figure 3-24 Result of Comparing Sequence Diagrams – sequence diagram in Figure 3-23 and the animated sequence diagram

In Figure 3-24, the highlighted red lines show the inconsistency between the two sequence diagrams. From the comparison, we can clearly see that the instance of object `B` receives the event `evChangeA(x = 5)` after the operation invocation `getA()`. This gives a visual confirmation of the explanation given in Section 3.2 to account for the result of the execution.

Figure 3-25 depicts the result of a second comparison where – as explained in detail in Section 3.2 – the execution is based on a triggered operation rather than asynchronous event-based communication.

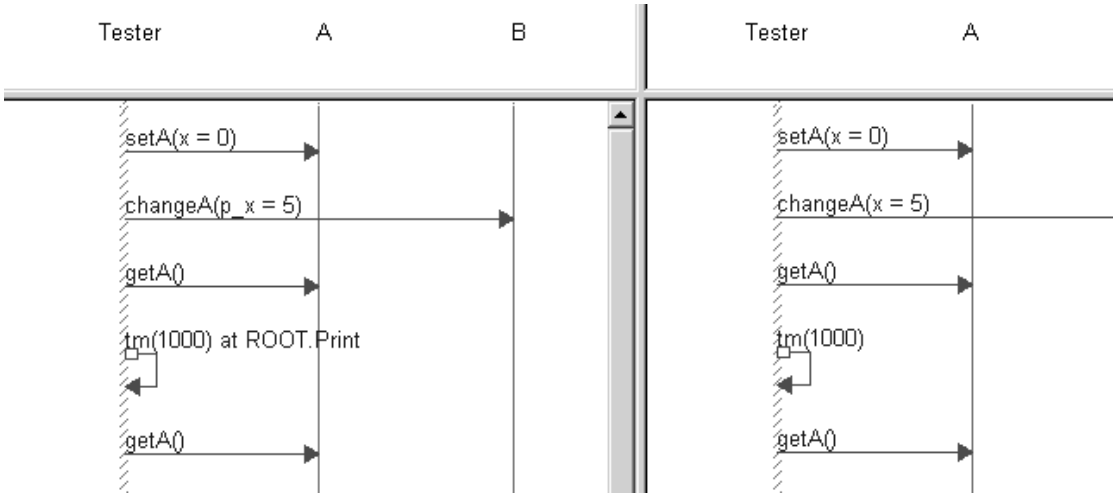


Figure 3-25 Result of Comparing Sequence Diagrams: using triggered operation instead of event reception (changeA instead of evChangeA)

From Figure 3-25, we can see that the animated sequence diagram is exactly the same as the specified sequence diagram, indicating that the implementation has met the requirements.

3.5 Iterative Development Process

It has been widely accepted that software development is an iterative process. Rhapsody supports this process by allowing the introduction of new classes, and letting the old classes cooperate well with newly introduced classes easily. To illustrate this, the development process of the author's VCCS model is separated into two significant iterations. The first iteration will be presented below. The model presented in previous sections of this chapter can be viewed as the result of the second iteration.

To present a prototype of the basic functionalities of VCCS, in the first iteration, the following classes are introduced (see Figure 3-25).

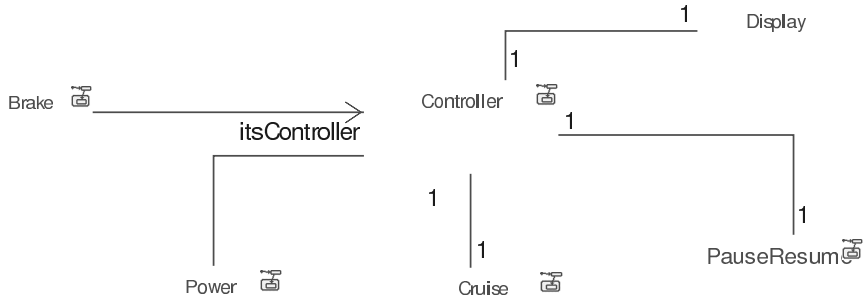


Figure 3-26 Object Model Diagram in Simple VCCS Model

The statecharts created in the first iteration are just the same as the Statecharts for the Controller (Figure 3-7), Power (Figure 3-8), Cruise, (Figure 3-9) and PauseResume (Figure 3-10) classes. The main differences between the models created in the first iteration and the second iteration are as follows:

- In the first model, for simplicity, there is no simulator to simulate the vehicle speed, instead, the speed was arbitrarily assigned a value, say, 45 mph when the power is switched on. The driver turns on the cruise control, sets a cruising speed and lets the car cruise at this speed. A random number generator is used to change the actual vehicle speed. With the cruise control on, whenever the vehicle speed is changed (that is, a discrepancy between the current speed and the cruising speed is detected), the cruise control will take effect to adjust the vehicle speed to attain the cruising speed.
- In the second model (see Figure 3-6), there are four new classes: `Accelerator`, `Simulator`, `ThrottleControl`, and `ThrottleActuator`. As soon as the power is switched on, the `Simulator` starts working to simulate the environmental factors and the responses of the vehicle. Adjusting the position of the `Accelerator` changes the vehicle speed. With the cruise control on, the driver can set the desired speed. The `Simulator` serves to simulate changes to the vehicle speed that might result from changes in the environmental factors (such as road conditions and air drag force). As soon as a discrepancy between current speed and cruising speed is detected, the `ThrottleControl` and

`ThrottleActuator` start working to adjust the position of throttle so as to attain the cruising speed.

Rhapsody makes the extension from the first model to the second one relatively easy. All the statecharts are kept as they are. New classes and new relationships between classes are introduced. The functionalities provided by the newly introduced classes can be integrated without too much extra work. It is only necessary to make the following small modifications to the original model:

- Add the following code in the `Action On Entry` of `On` state of `Power` (see Figure 3-8)

```
itsController.itsSimulator.start()
```

- Add the following code in the `Action On Exit` of `On` state of `Power` (see Figure 3-8)

```
itsController.itsSimulator.terminateSim()
```

- Redefine the vehicle's current speed by

```
CurrentSpeed = itsSimulator.getVehicleSpeed()
```

When these modifications have been made, the cruise control starts maintaining the cruising speed by `itsThrottleControl.start()` and terminates control by `itsThrottleControl.terminateControl()`.

3.6 Rhapsody and the Software Development Tool Chain

Users usually expect a software development tool to provide an open and extensible environment. There are a number of products available for performing various software development activities. Rhapsody provides interfaces to other widely used products, such as the Rational Rose analysis model, that can be imported into Rhapsody. Rhapsody also supports co-debugging with other third-party tools, say, Microsoft Visual C++, Borland C++ Builder, Borland Java Builder etc. Rhapsody

generated models can also be imported into other third-party tools such as DOORS and Tornado. Rhapsody also provides the facility for reusing legacy code, adding external files to the model, and even adding new environments.

Rhapsody can both use its internal editor (the default editor) and call an external editor to view long text fields, such as operation bodies, or to view the generated code. In the author's VCCS model, the class `Display` which is used to show the interface of the control panel is edited with the external editor JBuilder.

Chapter 4 Empirical Modelling Principles and Case Study

4.0 Overview

The discussion of Empirical Modelling in this thesis is oriented towards discovering its potential contribution in software system development. However, the original aim of EM was not directly concerned with system development. As mentioned in Section 1.4, EM models are computer-based artefacts built upon observation and experimentation. The typical product of the EM model development process is a **construal**. Many EM models are closely related to a real world situation; the variables in the model have direct referents in the real world, and the model itself is a direct reflection of the modeller's experience.

The relationship between the concept of EM as a whole and system development can be understood by analogy with cooking. There are various ingredients and utilities for cooking. Their role in cooking is analogous to that of artefacts that support system development. EM offers an environment which includes all the ingredients and utilities. The modeller is like a chef who can make use of everything that is available to do 'experimental' cooking. The chef has some experience of cooking. S/he may make a dish according to her/his experience with cooking. Her/his experience may come from some cookbook or her/his previous experience – following certain steps to make a tasty dish. But s/he may not follow any prescribed steps. S/he can mix any ingredients in making a dish, and can cook following any procedure s/he likes. To make a Chinese dish, the normal procedure is: ignite the hob; put a pan onto the hob; add oil; put the other ingredients into the pan one by one at the proper time (the most skilful step!); transfer the dish onto a plate. If everything goes to plan, dishes are made smoothly following the normal procedure. But there are circumstances in which the chef may forget for some reason that the oil is

warming (this might happen to anyone), and the oil in the pan may catch fire, so that the smoke triggers the fire alarm! What EM provides is a very open environment in which the modeller can imagine everything that might happen within this environment. The construal in EM is open to extension in such a way that it can include any things that exist in the environment, their possible interactions, and their relationships, of which the modeller may be aware or unaware. EM does not deny the usefulness of what a system provides. In our analogy, traditional system-building corresponds to cooking that follows what has been prescribed in a cookbook, and the system to the standard tasty dish that results. One can argue that this is all that system-building needs to achieve – the final tasty dish. If we adopt this limited view, we advocate following the standard recipe and using the given ingredients and utilities in the prescribed way. We can liken this to traditional system building using standard UML artefacts. The result may be that we get a delicious dish, but the recipe may constrain people's imagination with respect to inventing other delicious dishes. The use of artefacts in EM offers broader scope for the imagination, allowing people to explore other means and approaches – using the same ingredients but cooking in a different way may result in a Chinese style dish or an Indian style one.

The main contribution of an EM artefact to system development is that it provides a device for understanding and trying out of the envisioned system. For the sake of producing a useful system that meets the needs for application in a circumscribed situation, a formal specification such as we have in traditional system development can be derived from experiment with the artefact – the successful scenarios which are useful for this system will be inside the system boundary, and the undesirable scenarios will be outside the system boundary. Because of the control imposed by the logic of the program (which will realize the system), only the filtered scenarios will appear in the system.

4.1 EM perspective on Software Development

The potential contribution of EM to software system development has been

discussed in Section 1.4. The application of EM to software development is an extension of previous research into constructing EM models to serve as computer-based physical artefacts [BJ01]. When an engineering team designs a large complex structure, such as a ship, they draw on a lot of resources: theory, data and experience at their disposal, their combined intelligence, imagination and creativity. In addition to all this, they will probably build a model ship, and use the model as an experimental artefact. One research theme of EM is concerned with understanding how to develop and use a computer-based artefact as a physical artefact similar to the model ship built by an engineering team.

As an interactive situation model, an EM model is better adapted to the changing modelling context both in the development domain and the operational domain. Different participants, for example, developers and users, can work with the same model so as to reach a consensus. Developers can imagine all possible situations the system might confront, and users can imagine use of the system outside the scope of its normal use such as is provided by traditional system models. In this way, the new knowledge of requirements captured by developers at the construction stage, and by users at the operational stage, can be structurally coupled with the pre-existing knowledge embedded in computer-based models and in turn used as the basis for further interaction. The EM model is not circumscribed by the traditional 'system boundary' concept. EM does not follow any strict method for modelling. The construction of EM models reflects a very natural activity associated with our understanding of real-world phenomena.

The principles described above can be illustrated by a simple modelling exercise.

```
%donald
viewport SPEED0
point ptr
line needle
real needleLength
needleLength = 100.0
real minA, maxA, A
real ratio
real curSpeed, topSpeed
real labelHdisp, labelVdisp
minA = 4 * pi div 3
maxA = - pi div 3
needle = [{0,0}, {needleLength @ A} ]
topSpeed = 80.0
A = minA + (maxA - minA) * curSpeed div topSpeed
label L0, L10, L20, L30, L40, L50, L60, L70, L80
point P0, P10, P20, P30, P40, P50, P60, P70, P80
real A0, A10, A20, A30, A40, A50, A60, A70, A80
line mark0, mark10, mark20, mark30, mark40, mark50, mark60, mark70, mark80
real gap1, gap2, LSpc
gap1, gap2, LSpc = 10.0, 30.0, 50.0
A0 = minA ↵
P0 = {(needleLength + gap2) @ A0}
mark0 = [P0, {(needleLength + gap1) @ A0}]
L0 = label("0", P0-{labelHdisp,labelVdisp}+{LSpc @ A0})
```

Listing 4- 1 A Donald script for a speedometer

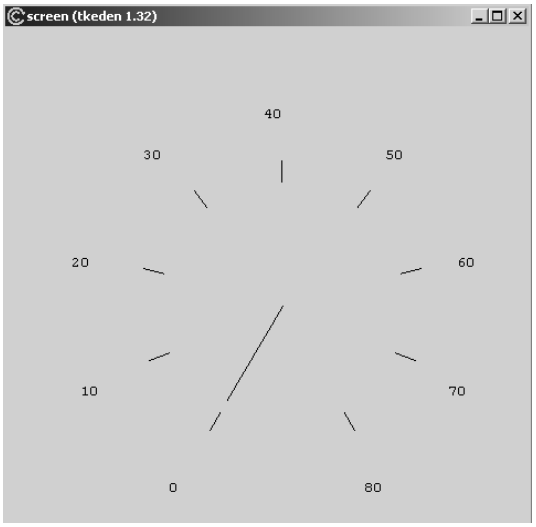


Figure 4- 1 A simple speedometer model

Listing 4-1 is the Donald script which describes the simple speedometer model

depicted in Figure 4-1. The variables in the script represent familiar observables that are associated with a speedometer: these include the length (`needleLength`) and angle (`A`) of the needle (needle) and the top speed that can be displayed (topSpeed). The current speed shown on the speedometer (`curSpeed`) is not defined in Listing 4-1: this has been set to 0 by making the definition:

```
%donald

curSpeed = 0.0;
```

Experiment with the model shows that if `curSpeed` is a negative speed, it can register as large and positive on the speedometer. For instance, we can change the speed to the positive speed 40.0 and a negative speed –40.0. The results are shown in Figure 4-2 and Figure 4-3 respectively..

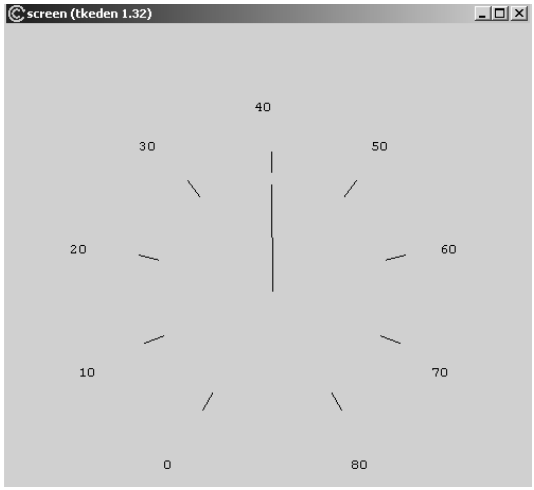


Figure 4- 2 A screen shot when curSpeed = 40.0

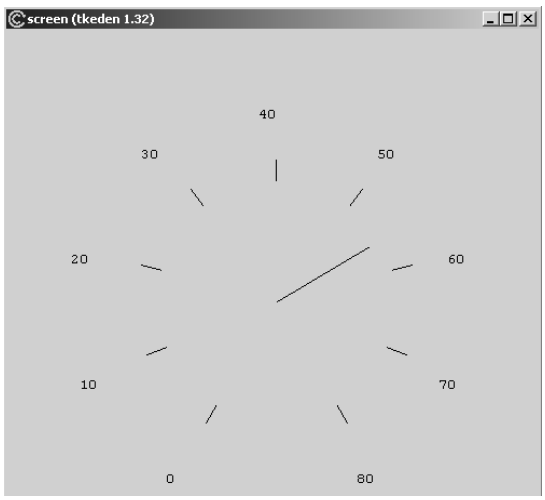


Figure 4- 3 A screen shot when curSpeed = -40.0

To make the model more realistic, we can add a 'stop' to the speedometer by introducing the definition

```
A = if curSpeed < 0 then minA + 0.1 else minA + (maxA - minA) * curSpeed div
topSpeed
```

After the above definition has been input, and the current speed has been set to –40.0, the state of the speedometer is as depicted in Figure 4-4.

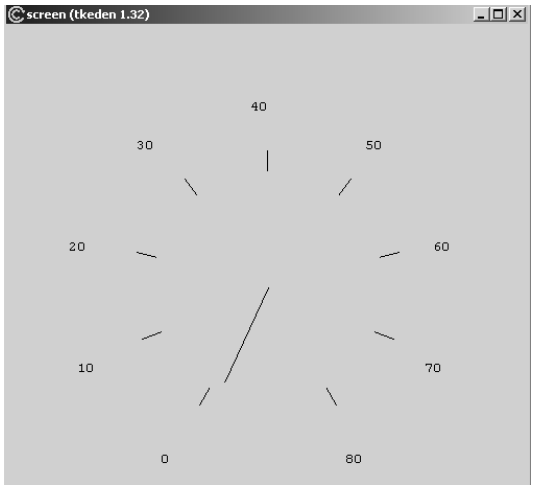


Figure 4- 4 A screen shot when a 'stop' is added to the speedometer (curSpeed = -40.0)

The following redefinitions can be used to confirm that the 'stop' has been correctly specified:

```
%donald

curSpeed = -70.0

curSpeed = 70.0
```

The redefinitions of the current speed relate to the normal use of the speedometer. We can also make redefinitions to change the speedometer design. For instance, the length of the speedometer needle can be changed so as to be proportional to the current speed by the following code:

```
needleLength = 100.0

real newNeedleLength
```

```
newNeedleLength = ( curSpeed * 100.0 ) div topSpeed

needle = [{0,0}, {newNeedleLength @ A}]
```

Figures 4-5(a), 4-5(b), 4-5(c), and 4-5(d) shows the different lengths of the speedometer needle when the current speed is 20.0, 50.0, 80.0, and 0.0 respectively.

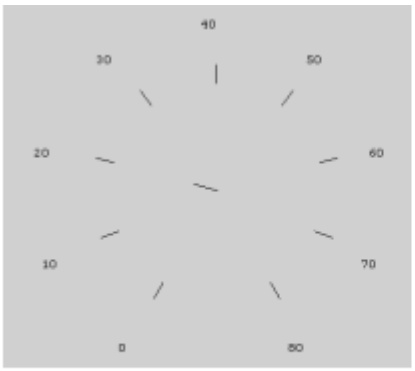


Figure 4-5(a) curSpeed = 20.0

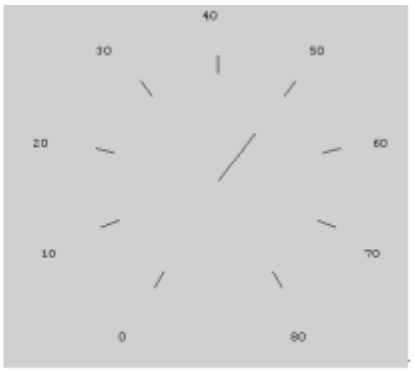


Figure 4-5(b) curSpeed = 50.0

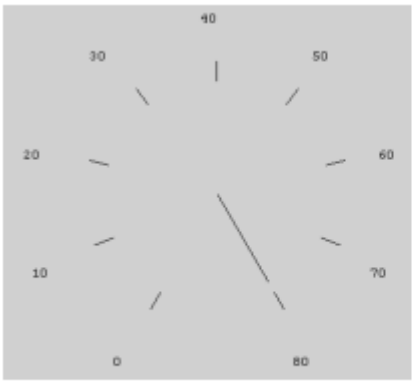


Figure 4-5(c) curSpeed = 80.0

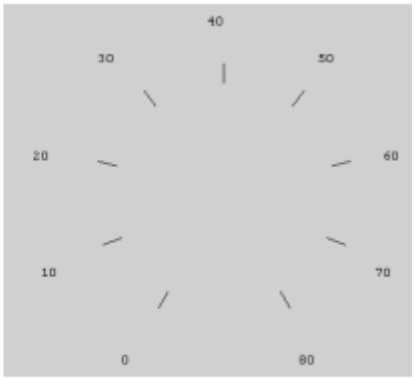


Figure 4-5(d) curSpeed = 0.0

Figure 4- 5 Different lengths of the speedometer when the curSpeed is 20.0, 50, 80.0, and 0.0 respectively

As another example of redesign, the speedometer can be changed to resemble a “rev counter”by inputing the following block of definitions:

```
minA, maxA = pi, 0.0

L0 = label("0", P0-{labelHdisp,labelVdisp}+{LSpc @ A0})

L10 = label("1", P10 - {labelHdisp, labelVdisp} + {LSpc @ A10})

…

L80 = label("8", P80 - {labelHdisp, labelVdisp} + {LSpc @ A80})
```

The result of this redefinition is depicted in Figure 4-6.

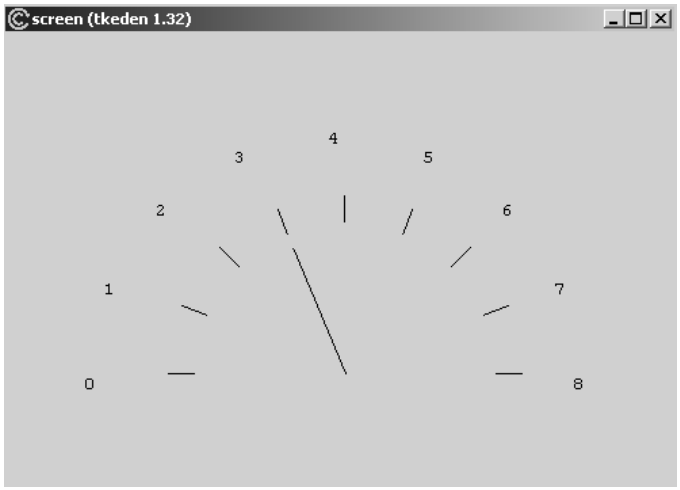


Figure 4- 6 A screen of a "rev counter"

In a sense to be explored in the following discussion, the EM model has a key quality identified by Harel in [Har91] – *executability*. As the above example illustrates, an EM model is executable in a different sense from that in which a Rhapsody model is executable. Like a spreadsheet, an EM model represents a real-world situation. Any interaction by redefinition can be regarded as a form of execution of the model (cf. 'what-if?' interaction in a spreadsheet). This concept of execution is quite different from the traditional 'running of a program', but it shares the most significant characteristic of program execution – it reflects what is going on in the real-world situation. An EM model also admits patterns of interaction via sequences of redefinitions that can either be performed manually or automatically that are similar to traditional program execution.

For EM, executability is a matter of modeller interaction through single redefinition or patterns of redefinition at a preliminary stage of modelling, and progressively more automation of redefinition as the model develops. This executability in EM provides an environment such that the modeller can experiment with the model from its preliminary stage. There is also an important distinction between executing a program

with a preconceived behaviour and making experimental interactions with an EM model. As explained in [Bey94], "The experimenter typically has some autonomous control over the values of particular observables, and may have no clear expectation of what consequent effects will be obtained". When an experimental interaction is performed for the first time, whether by design or by chance, the outcome is a matter of **immediate experience**. When the same outcome is reliably obtained many times, it can be interpreted as representing a **circumscribed behaviour**. As explained in [Bey94], "experiment is a way of bridging the gap between *immediately experienced* and *circumscribed behaviour*". Repeatable and predictable experiment resembles circumscribed behaviour, but "an authentic experiment also has elements of immediate experience". The activity of experimenting with the model is also a means of **testing** the behaviour of the model. The online interaction and immediate display of up-to-date data provide a very convenient facility for testing. The **open-ended** nature of Empirical Modelling makes change or extension of a model very easy.

Because of the special nature of the executability of an EM model, it provides a more integrated conceptual environment for experiment and supports communication among participants more effectively than the traditional static text-based model. Besides serving as a medium for communication among participants, EM models are also a medium for knowledge gathering and knowledge representation. As argued by Rasmequan et al in [RRR00]:

> *During the course of model development it is likely that new observables, dependencies and agents apart from those initially expected will arise during the course of model development. These new perceptions are the product of experimentation with the model and give insights to the user, or they may introduce the user to a new view or idea of the situation or problem being considered. In this way the process of model building proceeds in tandem with the enrichment of the user's own conceptual model of the domain. The scripts we build may reflect explicit propositional knowledge, but their principal merit is that the rich interaction possible with the artefact represented by such*

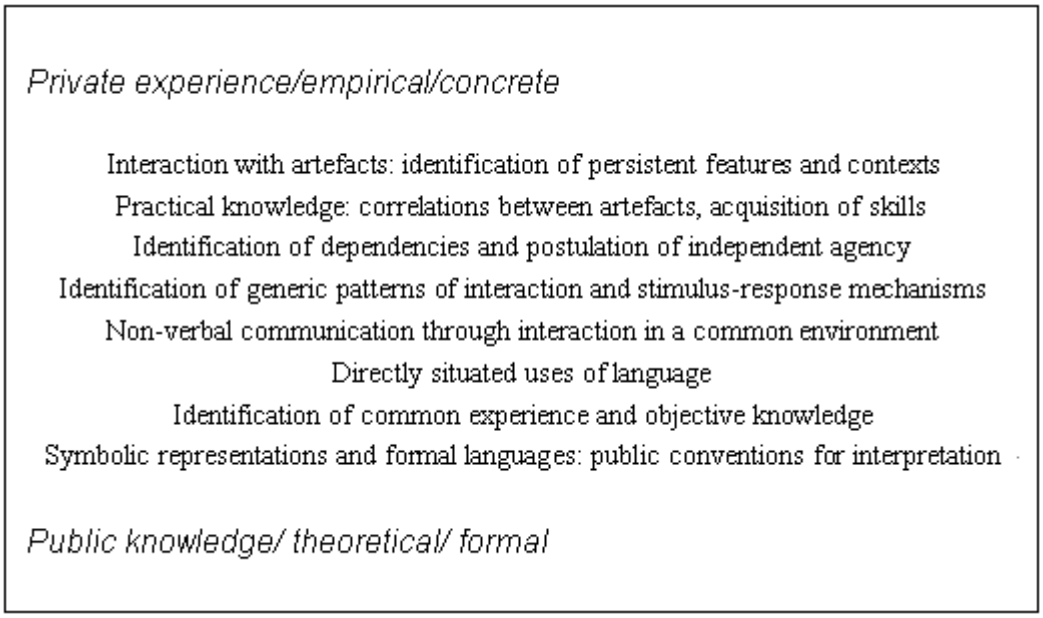
a script offers the engaged user experiential and tacit knowledge of the domain.

EM does not define strict development phases as other methods do. Software development using EM is an extension of the normal process of EM model development. This process is described in Box1 below [Bey99].

```
Whilst model is improving {
    By correlating experience of the world and experience of the model:
    Identify observables: things perceptible by agents
    Identify indivisible relationships
    Identify agents
    Identify stimulus-response patterns and constraints
    Refine context for experience
}
```

Box 1 The Process of EM Model Development

The process described in Box 1 reflects a form of primitive knowledge gathering that is similar in spirit to the informal activities that take place prior to specification and design in traditional software development. The knowledge gathering process is associated with a general progression of activities that are initially quite primitive, but become more sophisticated as the model–building proceeds. Activities representative of the progression are listed from top to bottom in the Empiricist Perspective on Learning (EPL) [RB02] depicted in Box 2.



Box 2 The Empiricist Perspective on Learning (adapted from [RB02])

As is pointed out in [RB02], traditional ways of thinking about computation give least support to the most primitive activities in the EPL – they primarily focus on the activities in the bottom part of Box 1. In contrast, the primary focus of EM is on the more primitive modelling activities which are listed at the top of Box 2.

In its most general form, Empirical Modelling for software development conforms to the following pattern. The model grows up incrementally along with the enrichment of the experience of the modellers. When the model acquires a certain level of maturity, the functionalities of the model may be able to meet all the requirements envisioned, and – if we wish – we can wrap it up by closing its interface and optimizing the model to its function so that it can no longer be so easily modified – just like the traditional software product. In the interest of efficiency of the implementation, the definitive scripts and actions in the EM model can also be translated into procedural programs and be compiled. As we know, compiled code has higher execution efficiency than interpreted code. Adding such an extension of the normal process of EM model development into Box1, we obtain a complete picture of the potential use of EM in software development (See Box 3).

Whilst model is improving {

By correlating experience of the world and experience of the model :

Identify observables: things perceptible by agents

Identify indivisible relationships

Identify agents

Identify stimulus-response patterns and constraints

Refine context for experience

}

Derive a specification from the EM artefact via an act-of-faith

Build a system from the specification using standard software development techniques

or

Derive a system from the specification by semi-automatic translation of the EM model

Box 3 The Extended Process of EM Model Development

In Box 3, the expression "deriving a specification via an act-of-faith" refers to the fact that the modeller can adopt two perspectives on fixing the functionality of the EM artefact: one that involves exercising discretion, and another that involves circumscribing behaviour. Once the behaviour of an EM artefact has been circumscribed, the derivation of a traditional system typically involves optimization that relies on guarantees about the scope for interaction (the 'act-of-faith'). Once such optimization has been introduced, the modeller can no longer exercise the same discretion over the system functionality.

4.2 Case Study – EM Draughts Model

The draughts model was initially implemented by Y.M.Au-Yeung (Regina) in 1996. It was then built upon by Simon Rawles in 1997. This model has been used as a demo of EM models. According to many students' experience, this model is a good example to show the distinctive features of EM models.

The rules for the traditional draughts game can be found in the Encyclopaedia Britannica as entry for "checkers". It is played on an 8 by 8 board. There are 24 pieces and each player holds 12 pieces at the beginning of the game. The grey squares

are not playable. The pieces are initially set on the board as Figure 4-7. Black is first to play. The basic rules of the games are:

- Moving: a piece can be moved diagonally forward into a vacant adjoining square.
- Capturing: if an opponent's piece is in an adjoining square, with a vacant square beyond, it can be captured and removed from play. The capturing piece is moved to the vacant square.
- Compulsion to capture: if a player is in a position to capture, he must capture rather than simply moving.
- Multiple capture: if, after capturing, a player finds he can capture again, he must do so.
- Crowning: if a piece reaches the opponent's back row, it is crowned, and called a king. King pieces can move and capture backwards as well as forwards.
- Victory: a player wins when all his opponent's pieces are either captured or blocked so that they cannot move.

The standard draughts rules are specified in more detail at [CenWeb].

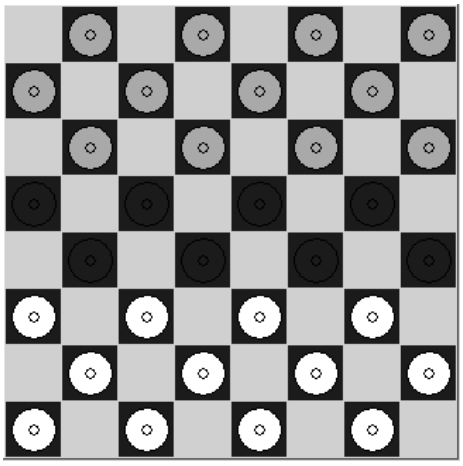


Figure 4- 7 Initial Status of Game

4.2.1 From Artefact to Specification

Traditional system development generally proceeds from a specification of a required behaviour to the construction of a system model to represent this behaviour. If we know the rules of the draughts game, the development of a computer-based draughts

game will consequently be based on the standard rules. With such an approach, the chances of someone's developing another size board rather than an 8 by 8 board for the game are very limited. Any other size boards would not be thought of as a board for the draughts game. In real life and in traditional system development, people tend to take it for granted that a draughts game can only be played on an 8 by 8 board. Factories who make the toys are unlikely to make other-sized boards, and a computer programmer will also only implement the game according to its specification.

The author was first introduced to the draughts model as a study in EM. The incomplete nature of the model as a draughts implementation was drawn to her attention, and she set out to complete the implementation task. In extending the original EM draughts game model, the author started from playing with the model itself, without knowing the rules for a traditional draughts game. Based on her previous experience of some similar games, through experimenting with the model and consulting other references, the author came to believe that the board should be extended to a 10 by 10 board. By coincidence, the first web reference to the rules of draughts that the author located [RulesWeb] in fact described a variant of draughts ("Slovenian Draughts") that is played on a 10 by 10 board and allows moves that can be regarded as a generalization of conventional draughts. This misunderstanding was identified only when the author drew attention to the failure of the original draughts to implement the generalised moves.

The above anecdote illustrates some of the potential virtues and pitfalls of EM. The author's revised version of the draughts game model, which has a 10 by 10 board, and in which each player has 20 pieces is depicted in Figure 4-9. But for the open-ended and unspecified nature of the EM draughts game model, it is most likely that the author would have adopted the traditional draughts game rules, and taken these rules for granted as the specification from which to build a computer-based draughts game with exactly the same functionality as other existing ones. The author's decision to replace an 8 by 8 board by a 10 by 10 one reflects a way of

system building which is contrary to the normal progression from a specification to a system model. The model is here regarded as an artefact which provides a very open-ended environment in which the modeller can exercise her imagination about things that can happen in the real world, rather than an intrinsically constrained system which can only realize the behaviour of its normal intended use. This illustrates the point made by Rasmequan et al in [RRR00], "the influence between developer and system is largely one-way: the functionality is planned and imposed on the artefact produced. In EM the influences between modeller and artefact are two-way; the modeller expects to gain new insights from the development."

4.2.2 Draughts Game Realization with Definitive Notations Scout, DoNaLD, and Eden

The draughts model is set up using a set of definitive notations: Scout, DoNaLD, and Eden. The Scout script defines the windows on the screen and their properties. The result of the Scout script is the whole layout of the model, including the board (left side of Figure 4-8), the information area (upper right side in Figure 4-8), and the control panel area (bottom right side in Figure 4-8).

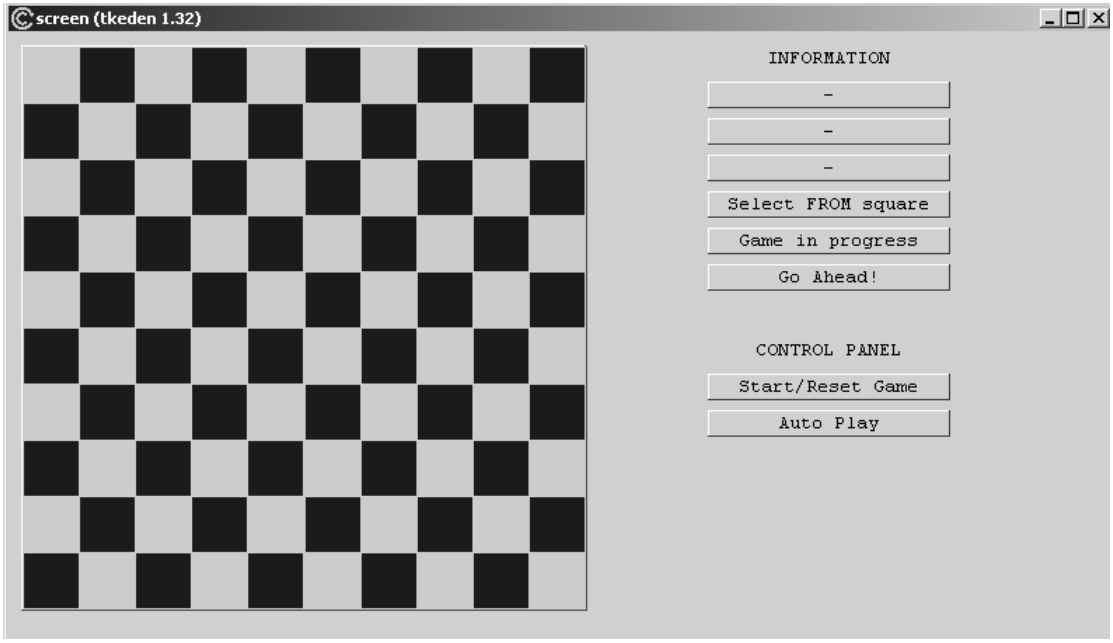


Figure 4- 8 Screen Layout of Scout Script

The board is defined by a sub-script of the model that could in principle be run on its own. In describing the entire model we shall first describe this sub-script. Each blue square in Figure 4-8 is a SCOUT window which acts as a viewport in which to display a line drawing. The DoNaLD script defines the line drawings in each square. Two circles are declared in each viewport; the circle called `Piece`*xy* representing the Piece itself, and the circle called `KingDot`*xy* representing the dot on top of the piece, that optionally marks it as a king piece. The result of the Scout and DoNaLD scripts is as shown in Figure 4-8.

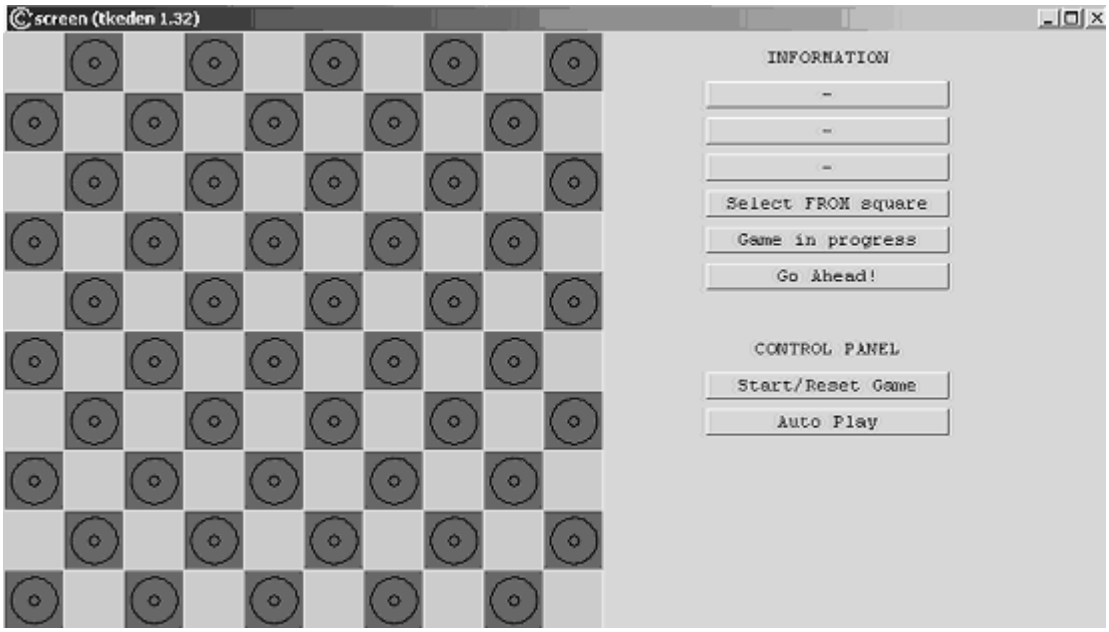


Figure 4- 9 Screen Layout of Scout and DoNaLD Script

Eden is the core of the definitive model, providing the underlying functionality, and the procedures required to implement it. The layout of the pieces on the board is modelled by adding a set of Eden definitions to the DoNaLD and SCOUT scripts. These definitions can be summarized as follows:

- Indicating which squares contain pieces

```
A_Piecexy is "fill=solid,color=" // colsqrxy;
```

- Indicating which squares contain king pieces

```
A_KingDotxy is "fill=solid,color=" // kingcolxy;
```

- Interrogating the current positions of the black and white pieces, and determining the colour of the piece on a particular square

```
Squarexy is checkcol([x,y], bpieces, wpieces);

bpieces is [b1, b2, …, b20];

wpieces is [w1, w2, …, w20];

…

b5 = [1,9]; /* black piece b5 is located on the square (1,9) */
```

- Defining the colours of the circles drawn in each square accordingly

```
colsqrxy is (Squarexy == white)? white:((Squarexy == black)? black:
```

```
bgcolor);

 kingcol11 is (iscrowned([1,1]) == 1)? red: colsqr11; /* iscrowned([x,y])

determines whether there is a king piece on the square (x, y)*/
```

In the EM model, the configuration of the board and pieces can be introduced without considering any other aspects of the draughts game. The modeller can do anything that is possible with a physical board and pieces. For example, the modeller can set up the position depicted in Figure 4-10 using the following script:

```
w1 = [1,1]; w2 = [3,1]; w3 = [5,1]; w4 = [7,1]; w5 = [9,1];
w6 = [2,2]; w7 = [4,2]; w8 = [6,2]; w9 = [8,2]; w10 = [10,2];
w11 = [1,3]; w12 = [3,3]; w13 = [5,3]; w14 = [7,3]; w15 = [9,3];
w16 = [2,4]; w17 = [4,4]; w18 = [6,4]; w19 = [8,4]; w20 = [10,4];
b1 = [1,7]; b2 = [3,7]; b3 = [5,7]; b4 = [7,7]; b5 = [9,7];
b6 = [2,8]; b7 = [4,8]; b8 = [6,8]; b9 = [8,8]; b10 = [10,8];
b11 = [1,9]; b12 = [3,9]; b13 = [5,9]; b14 = [7,9]; b15 = [9,9];
b16 = [2,10]; b17 = [4,10]; b18 = [6,10]; b19 = [8,10]; b20 = [10,10];
toplay = B;
wcrowned = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
bcrowned = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
move (1, 3, 5);
```

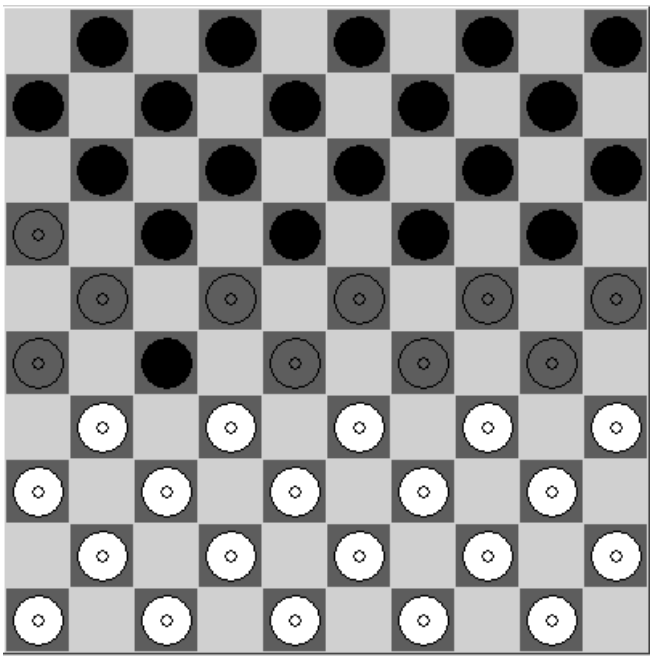


Figure 4- 10 A screen shot of pieces on the board set up by modeller

With the above definitions in place, it is possible for two people to use the model to play draughts. Players move the pieces around by redefining the variables of the form

`b1`, `b2`, ..., `w1`, `w2`, ... appropriately. This level of interaction raises some issues: players have to know the identifying index of each piece; players can make invalid moves; players have to perform two operations in order to capture a piece (e.g. in Figure 4-10, to capture the black piece `b1` on square (3, 5) the White has to redefine the position of the white piece `w16` on square (2, 4) (or the white piece `w17` on square (4, 4)) and that of `b1` so that it is no longer on the board). We can address these issues in the following way:

- To overcome the problem of identifying the pieces, we need to introduce dependencies to link the location of a square on the screen to the identity of the piece (if any) that occupies it.
- To ensure that players can only make valid moves, we need to add the rules of draughts to the model and introduce an interface to restrict the interaction of the players.
- To eliminate the two operations involved in a capture, we need to introduce an automatic action to remove the captured piece from the board.

The problem of identifying the pieces is resolved by the Eden function `idsq` which gives the piece number at any given coordinates. It provides the basis for the dependencies that link locations on the board and the contents of the 'information area' and 'control panel' displays to the internal state. This allows a player to select and move a piece directly by pointing and clicking on the appropriate ('from' and 'to') squares.

The following extract is taken from the script that defines the rules of draughts:

```
/* poss is the list of possible moves for the piece on the square (x, y) */
poss is [possnmm, possnmp, possnpm, possnpp, possn2mm, possn2mp, possn2pm,
possn2pp, posskmm, posskmp, posskpm, posskpp, possk2mm, possk2mp, possk2pm,
possk2pp];
/* possn is the list of possible moves of a normal piece */
possn is [possnmm, possnmp, possnpm, possnpp, possn2mm, possn2mp, possn2pm,
```

```
possn2pp];
/* possk is the list of possible moves of a king piece */
possk is [posskmm, posskmp, posskpm, posskpp, possk2mm, possk2mp, possk2pm,
possk2pp];
/* possnmm indicates whether a piece can make a normal move in the bottom-left
direction */
possnmm is (((toplay == B) && (playerowns)) && ((checkcol([x-1,y-1]) == blank)
&& (onboard ([x-1,y-1]))));
/* toplay is the identity of the current player, playerowns indicates whether
the piece on the square (x, y) is owned by the current player */
…
```

(There are similar definitions for moves in other directions.)

As the comments in the script indicate, the validity of moves is specified with reference to a particular piece and current player. The script determines whether a particular relocation of a piece is valid. It is used to constrain where a piece can move to when the game is played in the normal fashion, although this kind of constraint is not imposed unless the interaction is via the standard draughts interface (as will be shown in later examples). The standard draughts interface is implemented by the SCOUT buttons in the 'information area' and the 'control panel', together with automated actions to move pieces.

The principal Eden action used by the interface is `square`*xy*`_move_dest`, which sets the current piece to the square in which the mouse is clicked when the system detects a mouse click on the board. The procedure uses the `status` variable to keep track of the context in which the mouse click is registered so as to determine whether the click is to specify the 'from square' or the 'to square'. If, when the procedure is called, the value of `status` is `1`, the 'from square' coordinates (`xsel`, `ysel`) and the current piece coordinates `current_piece` are set, and status is assigned the value 2. If, when the procedure is called, the value of `status` is `2`, the 'to square'

coordinates (`xdest`, `ydest`) are set, and status is assigned the value 1. If the specified move is consistent with the rules of draughts, the piece is moved from (`xsel`, `ysel`) to (`xdest`, `ydest`). If the move is a capture, the captured piece is automatically moved to the coordinate position (1000, 1000). This means that the piece is no longer on the board.

The general principle behind the construction of the draughts model is that the modeller first creates the script of definitions necessary to represent a set of basic observables and actions, then implements agents (e.g. as triggered procedures in Eden) to automate selected actions. In effect, the automated elements of the computer-based model become progressively more complex as the model is developed. At the final stage of the model building, it then becomes possible to define a sharp boundary between the observables and actions accessible to the 'user' and the 'observables' and 'actions' to be processed by the machine. In this way, the development of the script can be thought of as a preparation for fixing the interaction between the modeller and the model by optimizing the open-ended interactive situation model, and so transforming it into a software system when it reaches a certain mature stage.

By way of illustration, from the user's perspective, the machine 'observables' in the draughts game playing model include:

- `moveok1`, indicates if a non-capture move is possible.
- `moveok2`, indicates if a capture move is possible.
- `cancapture`, indicates if the current piece can make a capture move.
- `canmove`, indicates if the current piece can make a normal move.

These guide the execution of the automated `move` action, which is invoked by `square`*xy*`_move_dest`.

The contents of the 'information area' and the 'control panel' reflect the observables and actions accessible to the user. These include the observables:

- `whoseturntext`, which tells the user whether black or white is to play

next.

- `clickcontexttext`, which tells the user how the next click will be interpreted by the model, whether it will be interpreted as the *from* square or *to* square, or alternatively, neither, if there is no game in progress.
- `lastsquaretext`, which shows the coordinates of the last square that was clicked.
- `lastsquarecontexttext`, which holds the contents of the last square that was clicked on.
- `game_status`, which shows the outcome when game is over.

They also include the complex action `startgame` to initialize a draughts game which sets up the board and player information ready for a game to begin.

4.2.3 Distinctive Modelling Concepts

The fundamental difference between traditional modelling approaches and EM modelling approach is given in [RRR00].

> *With a traditional approach … the boundary of the proposed system must be determined in advance. If there is a need for a new input or output, e.g. an additional system feature is required, then the whole system may have to be revised and re-designed at substantial cost. With the use of EM, any new observable or property of the model can be added in at any point during the modelling process through redefinition without the need to revise, or even re-start, the whole process. Part of the reason for this flexibility is that in EM we are focusing initially on the state of a model or domain, rather than on the behaviours we wish to produce.*

In the original EM draughts model, the game was initially set up as "black to play" according to the standard rules. In a traditional draughts program, it is impossible for the user to change who is first to play – unless this possibility was conceived in the

requirement, because the user can only interact with the model through the prescribed user interface provided beforehand. However, in this model, changing who is first to play involves just a few redefinitions. However, this change potentially requires the modeller to be familiar with the whole script. It can also be carried out in many conceptually different ways.

The following definitions interchange the colours of the pieces displayed on the board.

```
black = "white";

white = "black";
```

At this point, the 'white' pieces are owned by the Black and vice versa. The following redefinitions interchange the actual sides held by black and White:

```
W = 1; /* W determines who plays the white pieces; W is originally 0 */

B = 0; /* B determines who plays the white pieces; W is originally 1 */
```

Figure 4-11 and Figure 4-12 show the difference between the screen layout before and after this change. The effect of the change can be seen from the 'information area' of the display, where 'white to play' appears when the game is restarted (see Figure 4-13).

It might be argued that all that can be done here can also done using an object-oriented approach. In this context, the significant issues are:

- the kind of change that we have made above will be thought of as unnecessary, since by standard convention, Black is the first to play. On this basis, behaviour of this nature is unlikely to be taken into account;
- the emphasis in EM is on modelling the real-world situation rather than the behaviour of a specific system. As our example illustrates, there is a close connection between the steps that we take in changing the model and things that can happen in real-world situation, such as painting black pieces white and white pieces black, 'turning the board round', or players changing seats.

In these respects, an EM model does not restrict people's imagination, and embodies richer metaphors and meanings.

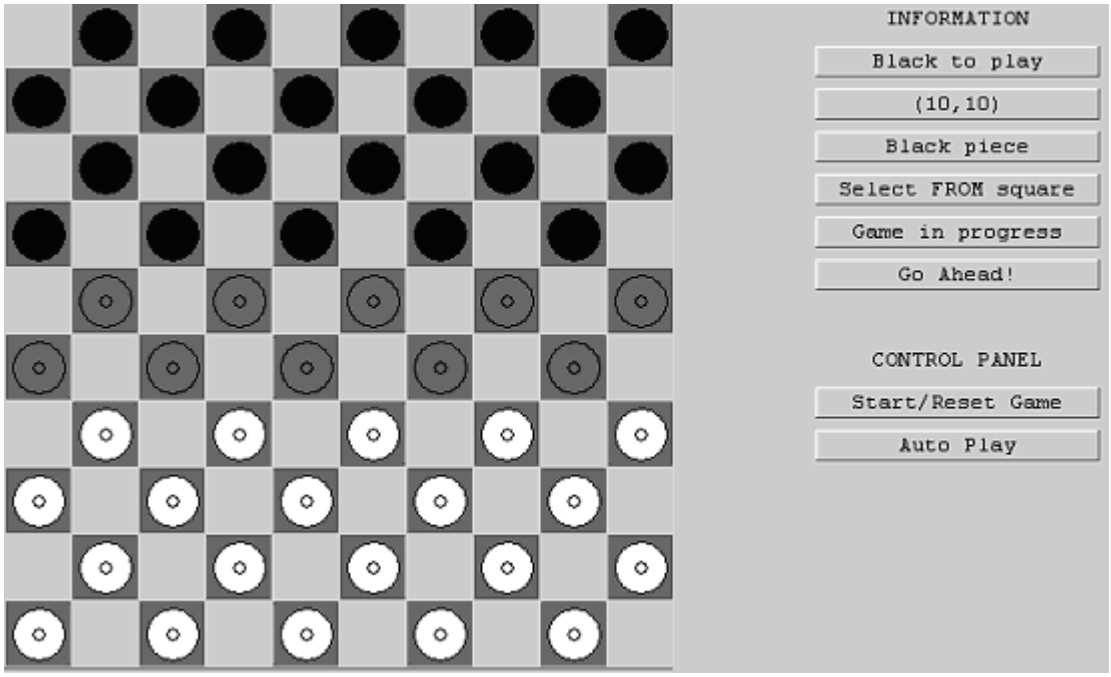


Figure 4- 11 Screen layout before changing the convention for who is first to play

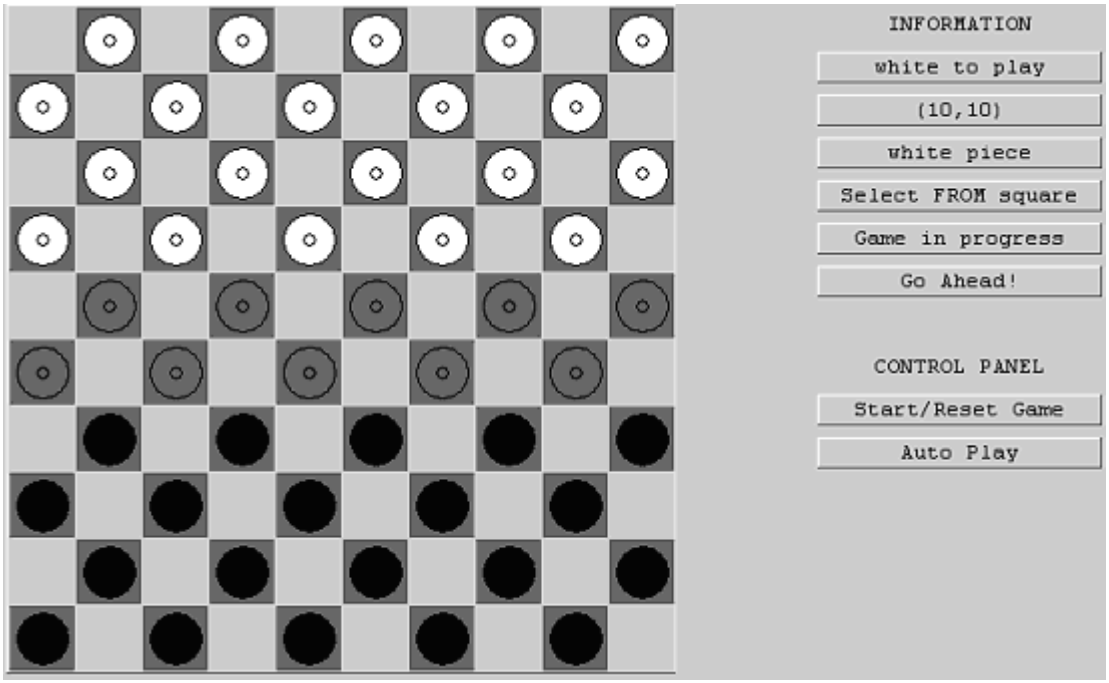


Figure 4- 12 Screen layout after changing the convention of who is first to play

In the original draughts model, there was no automated player, but it is easy to add one. For this purpose, the author introduced the action `autoplay`, which implements a very simple strategy for computer plays, namely, select the first of the list of available moves. As part of this action, the author implemented the compulsion to capture. The author's revised draughts model is depicted in Figure 4-11.

The scope for flexible interaction with the draughts model makes it suitable as an environment for studying the draughts game. In this role, the computer model is used as an artefact for visualization and exploration. For instance, pieces can be arbitrarily relocated in the middle of a sequence of moves, positions can be easily restored, and human or automatic play can be switched on and off. In some circumstances, we might choose to extract a useful application from the model and 'freeze' it into a system. For instance, a system to play draughts openings or particular types of endgame might be developed in this way. However, the model itself is always left open-ended for further exploration and serves as an artefact for participants to gain

more insight into relevant aspects of the domain.

In EM, as soon as certain observables are introduced, and appropriate dependencies between them have been established, a user (or modeller) can start to **experiment** with the model in an exploratory and open-ended manner (cf. the discussion in Section 4.1). The **open-ended** nature of the modelling provided by EM makes any reasonable or seemingly unreasonable changes or extension of model very easy. For example, the author was readily able to extend the original draughts model as described above. On the other hand, it is equally easy to introduce new rules to override the old ones, or to allow the user to place one piece on top of another one. The experimenting process can give the modeller new insight into the model under construction; at the same time, it can convert certain kinds of **immediately experienced behaviour** into **circumscribed behaviour** through repeatable experiments. Experiment is not only an activity to add new behaviours to the model; it also helps to test the reliability of the model already built. The next section provides some examples of its use in testing.

4.2.4 Testing

Testing is one stage amongst several in the typical software development lifecycle – requirement analysis, design, implementation, and testing. It is always the last step in a whole cycle and is always done by a separate development group. This leads to delay in the feedback to the developer. Many people have recognized the deficiency of this late testing. For example, testing is no longer a separate phase in the software life-cycle model in [Sch99], whilst XP advocates 'test first' and 'test always'. However, as already discussed in Section 1.4.1, to run a test case requires execution of the whole program so that certain functionality can be tested in the specific context. It costs time to compile and run the test cases all the time. Furthermore, there are some circumstances in which it is difficult or impossible for the model to reach a specific state no matter whether the testing is performed interactively step-by-step or

in an automatic batch fashion.

The on-line interaction and interpretation facility provided by EM makes it possible to perform the testing work in a much better way as listed here: "what if" test cases can be performed in any particular state; the state of model can be changed to any new state in ways that are not circumscribed by its running status; the values of variables (observables in EM) can be examined interactively on-line without the need to introduce "print" statements; any change that is made on-line to the model is reflected immediately.

The current values of observables can be changed and new definitions can be introduced through the input window provided by TkEden. This makes the interaction with the model more flexible than interaction that can only be performed through the graphical user interface which prescribes the possible ways of interaction. The modeller can also examine the current value of any observable, and obtain details of the observables with which it occurs in a definition, by querying it through the input window. For instance, to determine which is the currently selected piece, the modeller can input "`?current_piece;`" in the input window and the result will be printed out on the console. We can also use such a query in conjunction with the interface to determine information that is otherwise hidden. For instance, to determine the identity of any piece, the user can click on the piece and interrogate the value of the `current_piece` observable. When a new definition is introduced into the model, we can observe any change in the state of the screen or query the value of the variable redefined and its dependants. This is more interactive than the traditional approach of testing through looking at the values printed by print functions or setting break points to watch the execution status at certain moments.

The various features of EM testing are illustrated in the following test case scenarios:
Test Case 1. Testing whether the procedure `move` works as it should: moving the piece to its new coordinates, and switching the player. The modeller can invoke

`move` actions such as:

```
move (1, 1000, 1000);
move (1, 3, 5);
```

at any time, and monitor the effect on the board display.

Test Case 2.Testing whether the function `idsq` works in the way it should: given the parameter of the coordinates of a square on the board (coord), returning the identifying index of the piece on the square, if any.

```
number = idsq ([9,7]);
?number;
number = idsq (current_piece);
?number;
move(idsq(current_piece), 7,5);
```

Test Case 3. Testing whether `game_end` is true when there are no more black pieces on board.

1. start the game.
2. accept the following code

```
/* place all but one of the black pieces off the board */
for (i = 1; i<=19; i++) move(i, 1000, 1000);
/* place the remaining black piece on square (3,5) */
move (20,3,5);
```

3. capture the black piece by moving the white piece on square (2,4).

Figure 4-13 shows the status of the model after this sequence of interactions. The expected outcome “White is the winner” is displayed on the screen.

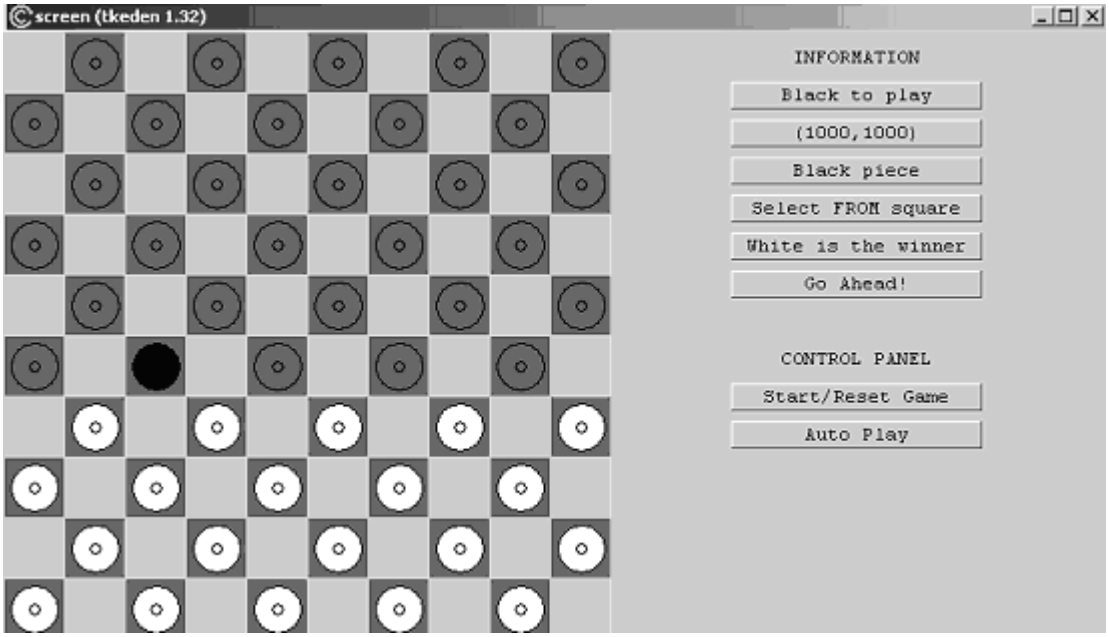


Figure 4- 13 "White is the winner" when black pieces can not move

Test Case 4. Testing whether `game_end` is true when all the black pieces are blocked.

1. start the game.

2. Input the following code and accept it

```
/* place all but one of the black pieces off the board */
for (i = 1; i<=19; i++) move(i, 1000, 1000);
/* place the remaining black piece on square (1,5)*/
move (20,1,5); /* this will change the current player */
move (20,9,5); /* move white piece 20 to square (9,5) */
```

Figure 3-28 shows the status of the model after this sequence of interactions. The expected outcome is that "White is the winner" is displayed on the screen.

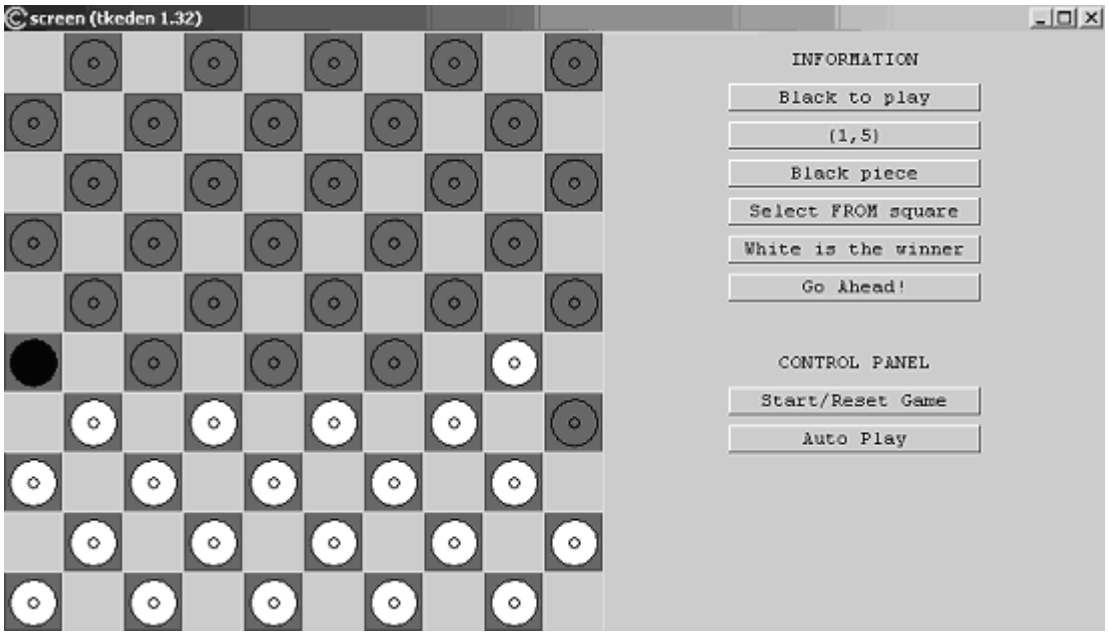


Figure 4- 14 "White is the winner" when black pieces are blocked

This is not routine testing in that it represents a departure from standard play. As can be seen from running the above test cases, the model can be changed to a new state without being restricted by its original state. Normally, a piece can only be removed from the board when it has been captured by the opponent. However, in the first test case, a piece is removed from the board ignoring the circumscription of behaviour imposed by the rules of the game. From the second test case we can see that, in EM, the value and definition of any observable can be examined instantly. Knowledge of the current value and definition of an observable can guide decisions about what to do next. From test cases three and four we can see that, in EM, it is easier to change the model to a particular state. Further experiment and testing can be performed in this state. If some experiments need to be performed repeatedly, it is also easy to restore the model to its original state: for example, an action of the form `move (i, x, y)` can be used to move a piece back to a particular square on the board.

4.2.5 Open-ended development

The EM principles of open-ended development are of topical interest in view of the increasing consensus that requirements are not pre-existent and hidden in the expert's

head waiting to be dug out and put into the specification cabinet [BCDS93]. The problem domain associated with the requirements of the developing system is generally informal, situated and open to the real world [Gog96, Blu93]; it cannot therefore be completely specified in advance. In the traditional software development process, there are two main proposals for solving the problems of constantly changing requirements. One is sufficient up-front design to accommodate new features. The expected result of up-front design is well-designed software that is flexible and extensible with a clear user interface. The other is to use rapid prototyping. Through running a prototype, the client's requirements can be gathered and confirmed. Without up-front design or prototyping, there is a greater risk that an additional system feature will be required, and that when this happens the whole system may have to be revised and re-designed at substantial cost.

EM proposes to use the computer to construct a situated, provisional, subjective model of the domain. A close connection between the model and its referent (the domain) is achieved by modelling that is continuously open to revision through comparison between the experiences of interaction with the domain and those of interaction with the model. As discussed in Section 1.4.1, EM takes account of both the requirements for change from the development domain and from the operational domain.

Chapter 5 Further Thoughts on Modelling

5.0 Overview

This chapter is mainly concerned with some further thoughts on modelling. Section 5.1 reviews UML modelling from the perspective of Empirical Modelling. Section 5.2 addresses some difficulties commonly confronted when using an object-oriented method. Firstly, there are no absolute criteria for classifying objects in the real world. Secondly, the objects used in computer world are different from the objects in real world. Thirdly, it is difficult to transit from the OOA model to the OOD model.

5.1 Comparison of UML Modelling and Empirical Modelling

The previous chapter is concerned with building a model with UML diagrams from which to generate code. UML diagrams provide a view of what the system is like at an abstract level. The designer imagines the complete system and different types of diagram reflect different views of the system. To build a system, the modeller needs to have sufficient knowledge of the system under construction. The diagrams are used to depict the system functionality (e.g. Use Case diagrams), system structure (Object Model Diagrams), and system behaviours (e.g. Statecharts and Sequence Diagrams). Although they are unlike the artefacts of EM, as introduced in 1.4.1, these diagrams are **also** called artefacts. These diagrams are static representations of the system structure, and of the system behaviour, which reflect, but do not explicitly embody, the hidden depth of the modellers' practical knowledge and experience. The distinction we have in mind here is that knowledge and experience *embodied* in an artefact is accessible through interaction with it rather than abstractly encoded. Diagrams are an abstract visual representation of the empirical knowledge that the

modeller derives from observation and experimental experience. In conventional system development, the empirical knowledge derived by the modeller is implicit; it is not explicitly represented by diagrams but is embodied in designers and the working environment. In EM, interactive situation models also play a role in embodying this empirical knowledge.

We can gain insight into the relationship between modelling with UML and EM by considering the nature of the observation that informs the creation of UML artefacts to represent system structure and behaviour.

In UML, the system structure is specified in the object model. An object model describes the key entities that comprise the system and the attributes that are associated with them. What informs the object model is our construal of how the system is assembled from pieces of state that can be observed in isolation and viewed as independent. The modeller may draw on experience of other similar systems and knowledge of familiar objects (or subsystems). There are no specific rules for mapping a real world object to a computer world one. The concept of object used in the object-oriented culture is intended to supply a natural mapping between the two different ‘worlds’. However, it is not unusual to find that an object in the computer world will also typically include constituents that relate to the artificial programming constructs that will be needed to mediate communication between objects. These objects are developed in response to observation of previous programs and reflect knowledge gained from the programmer's experience. In summary, the object model reflects two quite distinct modes of observation: one guided by observation of 'real-world' systems of a particular kind, and the other by observation of OO programs of a particular kind.

The system behaviour is specified – at least partially – by sequence diagrams and statecharts. Sequence diagrams are concerned with observation of interactions between objects. A sequence diagram is associated with observation of the system in

transition over a specific period of time with reference to the achievement of a specific goal (e.g. a use case) and the associated communications between the objects. This observation is concerned with what is characterised in the OO culture as "message passing" between objects. Because of the information hiding principle of OO, the observation is restricted to particular external propagation of state change between objects that is notionally visible in ways that the state changes within objects are not. The construal of behaviour includes identifying active objects responsible for state changes and passive objects involved in recording and mediating state change. Statecharts are concerned with observation of the possible behaviours of constituent objects in isolation. Each statechart diagram is associated with observation of the possible state transitions for its corresponding object. The statechart is developed from observation of transitions of the specific object over 'all' possible scenarios for interaction with the system. As discussed in Section 4.0, the 'all' here refers to those scenarios that fall within the scope of the intended use of the system, but excludes some scenarios for interaction that are possible in the real-world.

As we have remarked, statecharts only show the behaviour of individual objects. It is in general hard to get a detailed state transition model for the whole system because of its complexity. This is connected with the issue of code generation from a UML specification as discussed by Harel in [Har00]. Harel is concerned with translating observation of the system in normal operational use into prescriptions for interactive stimulus-response behaviour of the constituent objects. From an EM perspective, we are led to question the practicability of this strategy for code generation. In effect, Harel in [Har00] is seeking to develop a construal based solely on observation of the system in normal use and yet is sufficiently powerful to account 'completely' for the behaviour of the system. EM believes that developing this construal relies upon open experimental interaction with the system possibly outside the scope of normal use. An important ingredient in code generation from 'play-in' scenarios such as Harel introduces in [Har00] is giving stimulus-response interpretations to the actions of objects in an observed message sequence. Stimulus-response construal of behaviour

is typically most appropriate for components of reactive systems. Human agents do not normally exhibit strict stimulus-response behaviour. They may or may not respond at any time, and when they respond, the speed of response is uncertain. This discussion suggests that code generation from UML specifications is most realistic for reactive systems, in keeping with the emphasis in Rhapsody on the development of embedded system software.

5.2 Some thoughts on OO

The historical motivation for OO is that objects in programming should provide a more natural representation of real-world objects. Most OO introductions to object-oriented development start by illustrating the merits of OO by introducing examples classes such as 'vehicle' and 'animal'. Based on our experience in real life, it is not difficult for us to understand that the vehicle family consists of: the vehicles that provide transportation in the air (e.g. aircrafts, helicopters), the vehicles that provide transportation on the ground (e.g. trains, cars, bicycles), and the vehicles that provide transportation on the water (e.g. ships, hovercrafts). A single vehicle may be regarded as an object, and a family of objects which share some common feature may be grouped into a class. A class is an abstraction of objects. A group of classes which share common features can also be abstracted as a higher level class. A class at a higher abstract level contains the attributes and operations common to its lower level classes (or objects), and lower level classes (or objects) can inherit these common features. Lower level classes can also be given their own characteristic features by adding new features or override the existing features of the higher level classes. Inheritance, information hiding and a clear interface for object manipulation make reuse easier within the object-oriented paradigm. Object-oriented development has also achieved great success in industry. Many people claim that it has become the *de facto* industry standard. All the persuasive advantages brought by OO concepts, together with its predominant role in industry lead most people and companies to be enthusiastic about OO.

The success achieved by Object Orientation should be acknowledged, and all the advantages brought by OO concepts in respect of software evolution should also be appreciated. As the 'vehicle' example illustrates, there are no absolute criteria for classifying objects into classes. There might be circumstances in which we classify different vehicles according to the producer, or according to their ownership – whether vehicles are for public transportation or for private use. For this reason, the mapping of real-world objects to programming-world objects is not as direct as the terminology suggests, but involves creative work on the part of the software developers that perhaps ends up by making the classification of objects fit the problem that the software is intended to solve. The object concept that is in our mind is not the concept of an object entity from the real world but that more of an object entity created for programming world. In addition to the objects that are suggested by the subjective domain, we are also creating objects for programming. Jacobson [Jac92] classifies the objects in programming world into three kinds: **entity objects**, **control objects**, and **interface objects**. According to Jacobson, **entity objects** are things that correspond directly to things in the users' real world. Entity objects include: *concrete objects*, e.g. employee, product, tool; *conceptual objects*, e.g. corporation, strategy, membership, approval; *event and state objects*, e.g. purchase, delivery, arrival, ownership, status. **Interface objects** are used to encapsulate the detail of GUIs, communication protocols, and the like. **Control objects** are used to carry out complex methods that do not obviously belong to any class. From Jacobson's classification of objects, we can see that there is only one type of object – the concrete entity object that has a direct referent in the real world. The others are more or less created for the sake of programming use.

The recognition of the difference between objects in the real world and objects in the computer world leads to two different OO models in the software development process – the OOA model and the OOD model. As discussed in Section 1.3.1, it is claimed that the transition between object-oriented development phases is smooth because the same concept, that of *object*, is used throughout the process. However,

Kaindl argues in [Kai99] that OOA and OOD objects represent inherently different things. For instance, in Kaindl's view: "OOA objects are objects in a problem space", "an OOA model is an abstraction of things that exist in the real world", and "an OOA model helps requirement engineers understand the domain", whereas "OOD objects are generally defined as objects in a solution space" "an OOD model is a model of the proposed system's internal construction … an abstraction of the problem's solution in terms of the objects that will make up this system", and "in the course of OOD, developers must create additional objects that reflect an implementation to supplement OOA objects". The OOD model's purpose is to help developers understand the nature of the proposed system [Jac95]; the OOD model provides a medium for conceptualizing, communicating, and evaluating designs [CY91]; and it describes the system objects and their relations – the interactions among objects that implement the system's external behaviour [Rum97]. On this basis, Kaindl argues in [Kai99] that it is difficult to move from OOA to OOD; an OOA model cannot simply become an OOD model; thus, developers have to perform two difficult OOD tasks concurrently – they have to specify an architecture for the software and build a model of the domain to be used by that software.

The techniques and methodologies of OO have brought fruitful results, especially in respect of software reuse, and they are arguably leading software development towards an engineering process. However, the mapping from real world objects to computer world objects is not direct and natural at all. This might account for the fact that every introduction to an object-oriented method starts by teaching people how to find objects in the problem domain (OOA objects), and then devotes so much attention to explaining how to transform OOA objects into OOD objects. This might also be the reason why software development is always thought to be hard, even with modern object-oriented development techniques, and why Brooks calls for very competent designers.

5.3 Contribution of EM in Modelling Real World Objects

A model used in system development should serve to mediate between the real world requirements domain and the computer world of abstract code, and modelling activity should effect this mediation. However, models and modelling have a narrow interpretation in traditional software development. As discussed in Section 1.2, a model is regarded primarily as an abstraction from code and gives an abstract view of the system. Modelling is focused on the system that is to be built, so that all its activities are focused on obtaining a solution in the computer world for the user's problems in the real world. As a result, the model that is situated between the real world and the computer world will incline towards the computer world. It is difficult to match such a model, even the model generated in OOA, to the real world context in which the system is intended to operate. This might be part of the reason why we always remark in software engineering that "the deliverable is not exactly what the user requires".

Empirical Modelling arguably offers models that are more close to the real world. As discussed at length in Section 1.4 and Chapter 4, the EM approach to modelling systems focuses on *observation*, *dependency* and *agency*. These three concepts lead to a distinctive approach to using computers as instruments capable of embodying our knowledge of a domain and giving us experiential feedback from models which can be directly compared with the results of interaction with the world. In EM, even the computer-based model is concerned with more than just a circumscribed system. The EM model is a construal that provides a device for understanding and trying out the envisioned system in both its normal use and marginal use. Observables in the real world become variables in the EM model through observation; indivisible real-world state change that happens due to a causal chain is realized through a dependency relationship; real-world state change that happens due to stimuli-response reactions is

realized by agency (either through action on the part of the human modeller/user or through automatic action implemented by the computer). The concepts and mechanisms used in EM are more real world and human oriented than the counterparts used in object-orientation which are more computer world and programming oriented.

Although current EM research has its limitations in respect of tool support, as already pointed out by other researchers [e.g. Maad02], the concepts of modelling provide a novel approach which guides modelling closer to the real world and enables models to embody a broader vision than just the required system.

5.4 An EM Vehicle Cruise Control Model

The original motivation for this thesis came from studying an EM model of a vehicle cruise controller. The earliest version of this model was developed by Ian Bridge and Yun Pui Yung as a case study in system development in 1992-3. Figure 5-1 is a screen shot taken from the latest version of the model, as obtained from the EM project website [EMProjectWeb: cruisecontrollerPavelin2002].

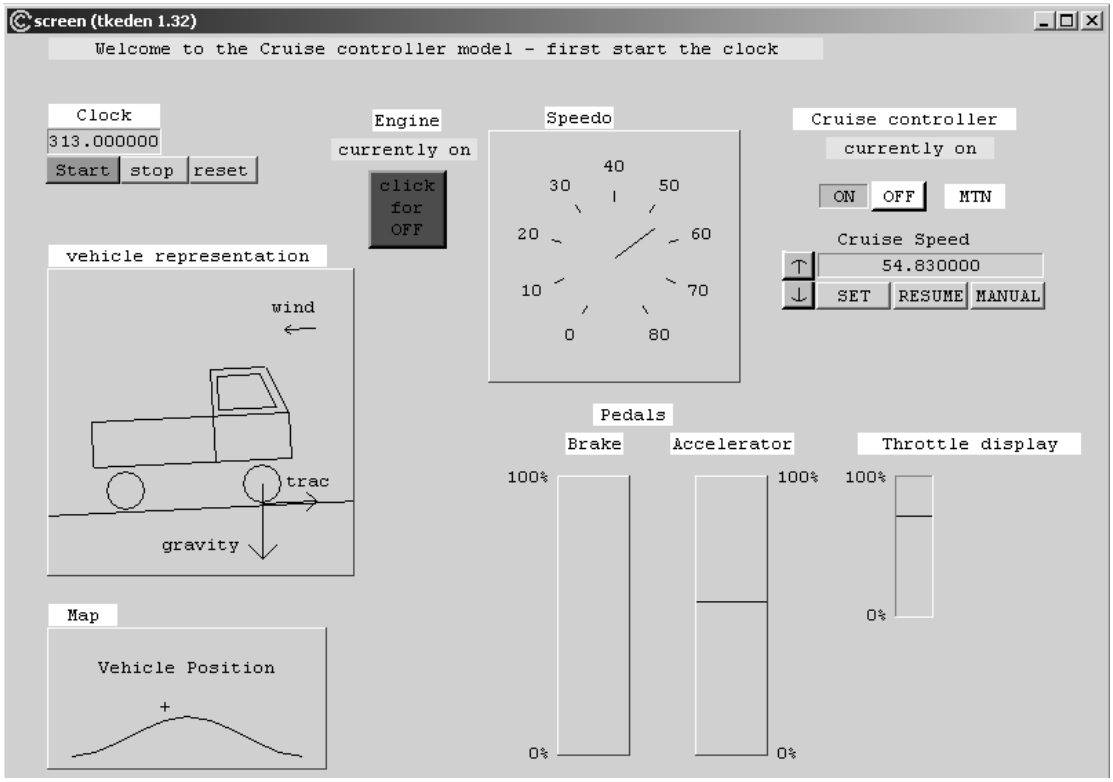


Figure 5- 1 A screen shot of the EM VCCS model

The principles used to construct the VCCS model are similar to those illustrated in the draughts model discussed in Chapter 4. The significant distinction between the two models lies in the degree of automation of agency that is needed to represent the real-time stimulus-response mechanisms in the reactive system. A most important consideration is the essential role that observation of time plays in the VCCS – one can play draughts without giving any consideration to observing 'time'. A full discussion of the techniques used in developing the EM VCCS model is beyond the scope of this thesis (for more details, see the EM references [BBY92, Sun99, Chen02]), but a brief comparison of the EM VCCS model and the Rhapsody VCCS model described in Chapter 3 is appropriate.

Part of the difficulty in comparing the EM VCCS and the Rhapsody VCCS models is that they are intended to address very different issues. The narrow aim of software development for a VCCS is to supply a digital control unit to coordinate the

interactions of various hardware components (e.g. the speed sensor, the throttle actuator, and the cruise control interface unit) so as to meet the requirements. As discussed in [BBY92], the fundamental issue is to what extent such requirements can be met without modelling the architecture and characteristics of the physical system in detail. The primary focus in EM is on the modelling of the physical system. A question such as 'is the speedometer functioning correctly?' is within the scope of the EM model whereas the traditional software developer takes the reliability of the speedometer for granted. Moreover, the open-ended EM VCCS model arguably cannot be thought of as a 'system', whereas the functionality of the Rhapsody VCCS model as a system is abstractly and precisely prescribed.

The above discussion can be illustrated with reference to the role of the speedometer in the EM VCCS model. Listing 4-1 is a sub-script for the speedometer that has been extracted from the VCCS script. As described in [BBY92], the EM VCCS contains sub-scripts to represent both the speedometer and the speed transducer that counts the number of revolutions of a wheel and returns the actual speed of the vehicle (`actSpeed`). The function of the speedometer cable that connects the speed transducer to the speedometer is modelled as a dependency by the single definition:

```
curSpeed is actSpeed;
```

In this context, the observable `actSpeed` represents the voltage output by the speed transducer, and the observable `curSpeed` represents the voltage input to the speedometer. It is the explicit nature of these observables that makes it possible to model failures such as a fault in the speedometer cable.

In some sense, this use of a dependency to link `curSpeed` to `actSpeed` is redundant in normal use of the VCCS. This is because the values of `actSpeed` is computed by a numerical integration and is updated in reality on every clock cycle and conceptually in every instant of time. On this basis, the effect of the dependency can be realized simply by assigning the value of `actSpeed` to `curSpeed` whenever `actSpeed` is updated. By a similar argument, the dependencies that link

the speedometer display to the current speed (see Listing 4-1) could be viewed as redundant. There are two kinds of virtue in using the dependency even in the execution of the VCCS model: One is that the interactions with the speedometer discussed in Section 4.1 can be performed in the context of the entire VCCS model whether or not it is in 'execution mode'. The other is that, with the dependency in place, it is possible to simulate a failure condition in execution such as the breaking of the speedometer cable.

An alternative approach to relating the EM VCCS model to the Rhapsody model would involve creating explicit object models for the speed transducer and speedometer within the Rhapsody framework. There are two possible scenarios to be considered: constructing an object model that conforms to the EM VCCS model in normal use, and constructing an object model to replicate the full functionality of the EM VCCS model in respect of interactions in design and use. In scenario one, the synchronization of messages between the speed transducer and the speedometer has to be such that the speedometer display is kept up-to-date with changes in `actSpeed`. Note that in the EM VCCS model, the dependencies linking `actSpeed`, `curSpeed`, and the position of the speedometer needle (cf. Listing 4-1) are automatically maintained by the Eden interpreter, whereas in the object model, the synchronization has to be explicitly programmed (cf. the discussion in Section 3.2 and 3.4.2). In scenario two, in order to reproduce the what-if style of interaction illustrated in Section 4.1, it would be necessary to implement the relevant dependency maintenance explicitly in an object model. This would entail constructing appropriate mechanisms for maintaining the dependencies between `curSpeed` and the observables in the speedometer display whether or not the 'system clock' is running.

Chapter 6 Conclusion

6.0 Research Summary

This research presented in this thesis grew out of the author's interest in examining the significant role of modelling activity in software development, and the relevant issues we need to be concerned about in such modelling.

The thesis starts by reviewing the problems that Brooks identifies as of the essence in his famous paper "No Silver Bullet" [Bro87]: complexity, conformity, changeability, and visibility. Brooks's paper stimulated a number of optimistic responses. For instance, Cox [Cox90] argues for a reusable and interchangeable component approach as an attack on the conceptual essence of the problem, and Harel [Har91] proposed a modelling technique called "The Vanilla Framework" as a potential silver bullet. This thesis has focused on discussing the ideas proposed in [Har91] and on tracing the subsequent development of Harel's original Vanilla framework. As Brooks indicated in [Bro95b], "Modelling does address the essence, the proper crafting and debugging of concepts, so it is possible that the Vanilla Framework will be revolutionary". Interest in investigating the achievement of the Vanilla Framework leads the author to gain experience with the Rhapsody tool, which integrates the most recent achievements in this area. The key ideas proposed in [Har91], such as visual modelling, model execution, and code generation, are all realized to some extent in Rhapsody. A VCCS model is developed using Rhapsody, and the key UML diagrams used for design and code generation have been illustrated and explained in detail with reference to the VCCS case study. Some key aspects of Rhapsody code generation mechanism are also illustrated. The key enabling technologies that are the basis of Rhapsody are listed in 1.3.3; these are model-based design, model-code associativity, automated implementation generation, an implementation framework, model execution, mixed-level debugging, animation,

debugging on the host, a collaborative development environment, and iterative development. Together these technologies constitute the Object Execution Framework. Issues related to the development and use of such a framework have been discussed in detail in Chapter 2.

Empirical Modelling has been discussed as an alternative modelling technique which is radically different but can be viewed as being in the spirit of Harel's Vanilla framework in some key respects. The focus of this thesis is on its potential theoretical and practical contribution to software system development. Some further thoughts with regard to modelling have been discussed in Chapter 5. These suggest that models and modelling for software development could be ideally oriented more closely to the real world domain rather than towards the specific requirements of the intended software systems.

6.1 Difficulties and Limitations of Research:

The scope of the author's research in this thesis as initially conceived was very broad and ambitious: investigating some past and current techniques of software system development; reviewing the problems identified by Brooks as of the essence of software development; and studying potential attacks on these problems proposed by some researchers. The extensive background to each issue has prevented the author from going into depth about every issue in one short year's research. For this reason, the research has focused on the modelling issues raised by Harel's Vanilla Framework and Empirical Modelling.

One difficulty in this research is the need to address both highly concrete and highly abstract issues: many issues relevant to the Vanilla Framework are very practical while many issues relevant to Empirical Modelling are very abstract and philosophical. As discussed previously, although there are issues relevant to both that are similar in spirit, as one can see from this thesis, the two approaches address

issues relating to software system development from very different points of view. For this reason, making the connection between the two different approaches has been one of the most difficult tasks of this thesis.

Another difficulty in this research has been the lack of appropriate documentation for both Rhapsody and EM. Rhapsody is a new CASE tool, and the tool itself is limited in many respects. For example, models developed with earlier versions of Rhapsody are not compatible with the latest version. As discussed in Chapter 2, training materials and mentoring play a significant role in learning an application framework. However, Rhapsody is quite new, and training material is not widely available. Furthermore, Rhapsody is developed for commercial use so that not all training material can be accessed freely. Because the emphasis in a university is on academic research, it has not been realistic to consult experts on such a new commercially used tool. There may be some functionalities of Rhapsody beyond the author's knowledge which might provide more powerful enabling techniques for real-time embedded systems. These might have been useful in the development of the Vehicle Cruise Control model. The explanation of key enabling features of Rhapsody may not be representative of its true capabilities.

The last difficulty of this research that will be identified relates to the process of understanding Empirical Modelling. As can be seen from this thesis, EM raises some fundamentally novel issues relating to software system development. However, because of the foundational nature of EM research, its commercial use has yet to be taken into serious consideration. The author was initially attracted to research on EM by the novelty of the approach and the prospect for potential practical application. The abstract orientation of EM contradicted the author's original impressions and expectations. For this reason, the author has found it particularly challenging to understand and convey its potential contribution for software system development.

6.2 Further Possible Work

As mentioned earlier in this Chapter, the scope of this thesis is broad, and there is room for many more years of research. In the author's view, a properly informed discussion of the agenda for this research should be based on wide and deep knowledge on both theoretical issues and practical experience of software system development. The research in this thesis is only viewed by the author as a start. Further research might be most usefully conducted alongside practical work in industry. The author is particularly interested in the following issues:

- How to successfully apply UML, RUP, and Rational support tools in OOSD.
- How to make modelling artefacts that indeed have the features of executability, communicability, and testability.
- How to improve the maturity level of software organizations.

Many practitioners who have started using UML, RUP and Rational tools report that these techniques do change something in software development. For example, the process of considering which type of diagram to draw helps to change the way that software engineers think about problems. Although UML is very well-known, it is not always being applied in the way that its designers intended in projects at present. In many cases, UML diagrams are drawn to document the design and development that has been done. Although this is not the way in which UML diagrams are intended to be used, people who have 'abused' UML in this way also report that, in the process of drawing UML diagrams to document the design and development work, some deficiencies of a design are likely to be realized. Bad design can also make drawing the corresponding diagrams to reflect the design difficult. For instance, the overview of a whole system represented by UML should be a diagram consisting of Packages and Relations between these packages. If the distribution of functionalities is not well-designed, it can even be very hard to construct this most general diagram. This implicitly forces the designers to re-think the design and also encourages them to learn some techniques embodied in UML. The drawing of UML diagrams makes the

designer re-think the original design and detect deficiencies earlier. Although UML is seemingly abused in this way, its use does have a positive effect. If the engineering methods identified by some of the best software engineers that are embodied in UML (and RUP) are properly applied in real projects, great improvement might be achieved in software development.

There are a lot of criticisms of UML. The critics have exposed the deficiencies of UML and imply that there is still a lot of work yet to be done. For example, UML is weak in representing the complex internal logic of a function of a class, and in its lack of strictness in representing interactions between objects (c.f. [DH99]). Currently, Activity diagrams are often used to represent the processing logic and Statecharts are used to represent the state transitions of an object. However, it has been found that both Activity diagrams and Statecharts are difficult to use when the complexity of problems increases. Moreover, it is hard to handle the ever more complex diagrams. The simplicity and transparency that pictures are intended to provide (as expressed in the saying: "one picture is worth a thousand words") are decreased as the complexity increases.

The combination of modelling concepts from the Vanilla Framework and EM with UML may contribute much to software engineering as a whole. It might be a big improvement in computer-aided software engineering if the current CASE tools integrated features that EM tools and Rhapsody support, such as executability, communicability, and testability.

Another topic that has aroused the author's particular interest is how to improve the maturity level of software organizations no matter what maturity level they are at. Although the Waterfall Model has been criticized in software engineering, it is still the major software life-cycle model used in practice. According to statistics, both software quality and meeting deadline can be guaranteed if the software activities can be performed strictly following the KPAs (Key Process Areas) prescribed in the

Carnegie-Mellon Software Engineering Institute Capability Maturity Models (CMM) [CMMDoc]. Unfortunately, software organizations can seldom afford to invest enough human resources to improve their maturity level. Even an organization that has already achieved a certain maturity level (level two or above) for some projects under special circumstances, may find that there is no time to follow the KPAs. Making decisions about this dilemma calls for an effective strategies.

Enhancing the maturity level of software organization needs a big investment in human resources. People who are professional in applying CMM and who are familiar with the current management and technical situations are crucial. However, such people are a very scarce resource. Big investment is also a formidable hurdle for many organizations. Another problem stems from the people who implement the KPAs. A good part of work involved in achieving KPAs is seemingly very routine and lacking in creativity (which is the biggest source of job satisfaction for programmers), so that implementers are always reluctant to change their habits. The author's interest in this area is in how to enhance the maturity level, to change software engineers' habits, and to make the transition to a more scientific and manageable practice. This is an issue that has both technical and management aspects which the author believes requires rich work experience of software development.

Bibliography

[ÅW84] Åström, K.J and Wittenmark, B. *Computer-Controlled Systems*. Prentice-Hall 1984.

[Beck00] Beck, K., *Extreme Programming Explained, Embrace Change*, Addison Wesley, 2000

[BC93] Beynon, W.M., and Cartwright, A.J., *Agent-Oriented Modelling for Engineering Design*, Proc CAD '93, New Information Technologies in Science, Education and Business, Yalta, May 1993, pp.49-53

[Bey94] Beynon, W.M., *Agent-Oriented Modelling and the Explanation of Behaviour*, Invited paper for the University of Aizu International Workshop on "Shape Modelling: Parallelism, Interactivity and Applications", 1994

[BACY94] Beynon, W.M., Adzhiev, V.D., Cartwright, A.J., and Yung, Y.P., *An Agent-oriented Framework for Concurrent Engineering*, Proc. IEE Colloquium: Issues of Cooperative working in Concurrent Engineering, Digest 1994/177, 9/1-9/4, October 1994.

[BJ94] Beynon, W. M., and Joy, M. S., *Computer Programming for Noughts-and-Crosses: New Frontiers*, In Proc. PPIG'94, Open university, England, January, 1994.

[BR95] Beynon, W. M., and Russ, S.B., *Empirical Modelling of Requirements*, research report, Dept. of Computer Science, University of Warwick

[Bey97] Beynon, W.M., *Empirical Modelling for Educational Technology*, In

Proc. Cognitive Technology '97, University of Aizu, Japan, IEEE, 1997, pp.54-68

[Bey99] Beynon, W.M., *MSc Lecture Notes on "Empirical Modelling for Concurrent Systems"*, Dept. of Computer Science, University of Warwick, 1999

[BCSW99] Beynon, W. M., Cartwright, R., Sun, P-H and Ward, A., *Interactive situation models for information systems development*, In Proc. SCI'99 and ISAS'99, Orlando, USA, 1999, Vol. 2, 1999, pp.9-16,

[BRR00] Beynon, W.M., Rasmequan, S., and Russ, S., *The Use of Interactive Situation Models for the Development of Business Solutions*, Proceedings workshop on Perspective in Business Informatics Research (BIR-2000), Rostock, Germany, March 31- April 1, 2000.

[BCDS93] Blyth, A.J.C., Chudge, J., Dobson, J.E., and Strens, M.R., *ORDIT: A new methodology to assist in the process of eliciting and modelling organisation requirements*, Technical report No. 456, Dept. of Comp. Sci., Univ. of Newcastle upon Tyne

[Boe88] Boehm, B.W., *A Spiral Model of Software Development and Enhancement*, IEEE Computer, May 1988, pp.61-72

[Boo86] Booch, G., *Object-Oriented Development*, IEEE Trans, On Software Engineering SE-12, 2, February 1986, pp.211-22

[Boo99] Booch, G., *Object-oriented analysis and design with applications*, Benjamin/Commings, 1999

[Bro87] Brooks, F.P., *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, Vol. 20, No. 4, Apr. 1987, pp.10-19. Also appeared in

Information Processing 86, H.-J. Kugler, ed., Elsevier Science Publishers B.V., North- Holland, 1986, pp. 1,069-1,076.

[Bro95a] Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Anniversary Ed.,1995

[Bro95b] Brooks, F.P., *"No Silver Bullet" Refired*, In The Mythical Man-Month, Anniversary Ed., pages 205-26, Addison-Wesley, 1995

[CenWeb] http://www.centralconnector.com/GAMES/checkers.html

[CMMDoc] Software CMM v1.1, available at http://www.sei.cmu.edu/cmm/

[CRB00] Chen, Y-C., Russ, S., and Beynon, W.M., *Empirical Modelling for Business Process Reengineering: An Experience-Based Approach*, Proceedings workshop on Perspective in Business Informatics Research (BIR-2000), Rostock, Germany, March 31- April 1, 2000.

[Chen02] Chen, Y-C., *Empirical Modelling for Participative Business Process Reengineering*, Ph.D. Thesis, University of Warwick, U.K., 2002

[CY91] Coad, P., and Yourdon, E., *Object-Oriented Analysis, second ed.*, Prentice Hall, Englewood Cliff, N.J., 1991

[CD94] Cook, S. and Daniels, J., *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994

[Cox90] Cox, B.J., *There is a Silver Bullet*, *BYTE* (October, 1990), pp. 209-218

[Cro94] Crook, C., *Computers and the collaborative experience of learning*,

Routledge, London, 1994

[DH99] Damm, W. and D. Harel, *LSCs: Breathing Life into Message Sequence Charts*, in Proc. 3rd IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems, P. Ciancarini, A. Fantechi, and R. Gorrieri, eds., Kluwer Academic, New York, 1999, 293-312

[Deu89] Deutsch, L.P., *Design reuse and frameworks in the Smalltalk-80 system*, In Ted J. Biggerstaff and Alan J. Perlis, editors, Software Reusability, Volume II: Applications and Experience, Addison-Wesley, Reading, MA, 1989, pp.57–71

[Dou98] Douglass, B. P., *Real-time UML: developing efficient objects for embedded systems*, Addison Wesley, 1998

[Dou99] Douglass, B.P., *Doing Hard Time: developing real-time systems with UML, objects, frameworks, and patterns*, Addison Wesley, 1999

[EMWeb] http://www.dcs.warwick.ac.uk/modelling

[FSJ99] Fayad, M.E., Schmidt, D.C., and Johnson, R.E., *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, Wiley, 1999

[Fow01] Fowler, M., *Is Design Dead?* XP 2000 Conference, available at http://www.martinfowler.com/articles/designDead.html

[GHJV95] Gamma, E., Helms, R., Johnson, R. and Vlissides, J, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995

[GHP02] Gery, E., Harel, D., and Palachi, E., *Rhapsody: A Complete Life-Cycle Model-Based Development System*, Third International Conference, IFM 2002, Turku,

Finland, May 15-18, 2002

[Gog96] Goguen, J.A., *Formality and informality in requirements engineering*, in Proc. 2nd Inter.Conf. on Requirements Engineering, 1996, pp 102 –108

[Gom00] Gomaa, H., *designing concurrent, distributed, and real-time applications with UML,* Addison Wesley, 2000

[Gou01] Goulding, J., To build and deploy complex embedded software applications. *Embedded Control Europe* (Feb.), 2001, 58—61, Available at http://www.embedded-control-europe.com/magazine.php

[Har00] Harel, D., *From play-in scenarios to code: An achievable dream*, IEEE Computer 34(1) 53--60, Jan. 2001.

[Har86] Harel, D., Statecharts: A visual formalism for complex systems. Sci. Computer. Prog., 1987, pp.231-274

[HG97] Harel, D. and Gery, *Executable object modelling with statecharts*, IEEE Computer, 30(7):31--42, 1997, Also, Proc. 18th Int. Conf. on Software Engineering, Berlin, IEEE Press, March, 1996, pp.246-257

[Jac91] Jacobson, I. 1991. *An Ambitious Goal: Industrial Development of Software with an Object-Oriented Technique*. The Road to Unified Software Development Process, Cambridge University Press, 2000

[Jac92] Jacobson, I., Object-oriented software engineering : a use case driven approach, Addison-Wesley, 1992

[Jac95] Jacobson, I., *A Confused World of OOA and OOD*, Journal of

Object-Oriented Programming Vol.8, No.5, Sept. 1995, pp. 15-20

[JBR99a] Jacobson, I., Booch, G. and Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley, 1999

[JF88] Johnson, R.E and Foote, B., *Designing reusable classes*, Journal of Object-Oriented Programming, 1(2):22–35, June/July 1988

[Joh97] Johnson, R.E. *Frameworks = (Components + Patterns)*, Communications of the ACM, in Object-Oriented Application Frameworks Theme Issue, M.E.Fayad and D.C. Schmidt, editors, 40 (10), October 1997, 39-42.

[Kai99] Kaindl, H., *Difficulties in the Transition from OO Analysis to Design*, IEEE Software Magazine, September/October 1999 (Vol. 16, No. 5), pp. 94-102

[Maad02] Maad., Soha., *An Empirical Modelling Approach to Software Development in Finance: Applications and Prospects*, Ph.D. Thesis, University of Warwick, U.K., 2002

[ML01] Markievicz, M. E. and Lucena, C.J.P., *Object Oriented Framework Development*, ACM Press, 2001, pp. 3–9.

[Mat01] Mattsson, M., *Object-Oriented Frameworks: A Survey of Methodological Issues*, Techical Report 96-167, Dept. of Software Eng. And Computer Science, University of Karlskrona/ Ronneby, 2001

[MBF99] Mattsson, M., Bosch, J., and Fayad, M.E., *Framework Integration Problems, Causes, Solutions*, Communication of the ACM, October 1999, Vol.42, No.10.

[Pre95] Pree, W., *Design Patterns for Object-Oriented Software Development*, Reading, MA: Addison-Wesley, 1995

[Pri00] Priestley, M., *Practical Object-Oriented Design with UML*, Addison-Wesley , 2000

[EMProjectWeb] cruisecontrolPavelin2002
http://www.dcs.warwick.ac.uk/modelling/~empublic/projects/

[Qua00] Quatrani, T., *Visual Modelling with Rational Rose 2000 and UML*, Addison-Wesley, 2000

[RhasodyTutorial] *Rhapsody in J Tutorial*, I-Logix Inc.

[RapsodyUserGuide] *Rhapsody User Guide*, I-Logix Inc.

[RRR00] Rasmequan, S., Roe, C. and Russ, S., Strategic Decision Support Systems: An Experience-Based Approach, Proceedings of the 18th IASTED Conference on Applied Informatics, Innsbruck, Austria, February 14-17 2000.

[Rob97] Roberts, D., and Johnson, R.E., *Evolving Frameworks: A Pattern Language for Developing Frameworks*. Reading, MA: Addison-Wesley, 1997.

[Rog97] Rogers, G., *Framework-Based Software Development in C++*, Upper Saddle River, N.J.: Prentice Hall, 1997

[RB02] Roe, C. and Beynon, M., Empirical Modelling principles to support learning in a cultural context, Dept. of Computer Science, University of Warwick, 2002

[R+91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modelling and Design*, Prentice Hall, 1991

[Rum97] Rumbaugh, J., Models Through the Development Process, *J. Object-Oriented Programming*, Vol. 10, No.2, May 1997, pp5-14.

[m-wWeb] www.m-w.com

[RulesWeb] http://lrv.fri.uni-lj.si/~bojank/rules.html

[STM95] Sallis, P., Tate, G. and MacDonell, S., *Software engineering: practice, management, and improvement*. Addison-Wesley, 1995

[SeaWeb] www.searchWin2000.com

[Sch99] Schach, S.R., *Classical and Object-Oriented Software Engineering with UML and Java, fourth edition*, McGrawHill, 1999

[SGW94] Selic, B., Gullekson, G and Ward, P.T., *Real-Time Object-Oriented Modelling*, John Wiley & Sons, New York, 1994

[Sha91] Shaw, M., *Heterogeneous Design Idioms for Software Architecture*, Proc. Sixth International Workshop on Software Specification and Design, IEEE, October 1991, pp.158-165.

[Sha95a] Shaw, M., Beyond Objects: *A Software Design Paradigm Based on Process Control*, ACM Software Engineering Notes, Vol 20, No 1, January 1995.

[Sha95b] Shaw, M., *Comparing Architectural Design Styles*, IEEE Software, IEEE Computer Society, November 1995, pp27-41

[Smi01] Smith, J., *A Comparison of RUP and XP, Rational Software White Paper*, 2001, available at http://www.rational.com/media/whitepapers/TP167.pdf

[Sta99] Stanglewicz, J., *Design-Level Debugging*, Real-Time Magazine, 99-1, Available at http://www.real-time-info.com.

[SRCB99] Sun, P.H., Russ, S.B., Chen, Y.C., and Beynon, W.M., *Cultivating Requirements in a Situated Requirements Engineering Process*, Dept. of Comp. Sci., Univ. of Warwick, 1999

[Sun99] Sun, P-H., *Distributed Empirical Modelling and its Application to Software System Development*, Ph.D. Thesis, University of Warwick, U.K., 1999

[TBT00] Truex, D., Baskerville, R. and Travis, J., *Amethodical systems development: the deferred meaning of systems development methods*, Accounting Managment and Information Technologies 10 (2000) pp 53-79

[UMLDoc] Unified Modeling Language (UML) documentation, Object Management Group (OMG), http://www.omg.org

[UMLWeb] http://www.uml.org

[UMLSpec] UML1.4 Specification

[V-ModelWeb] http://www.v-modell.iabg.de

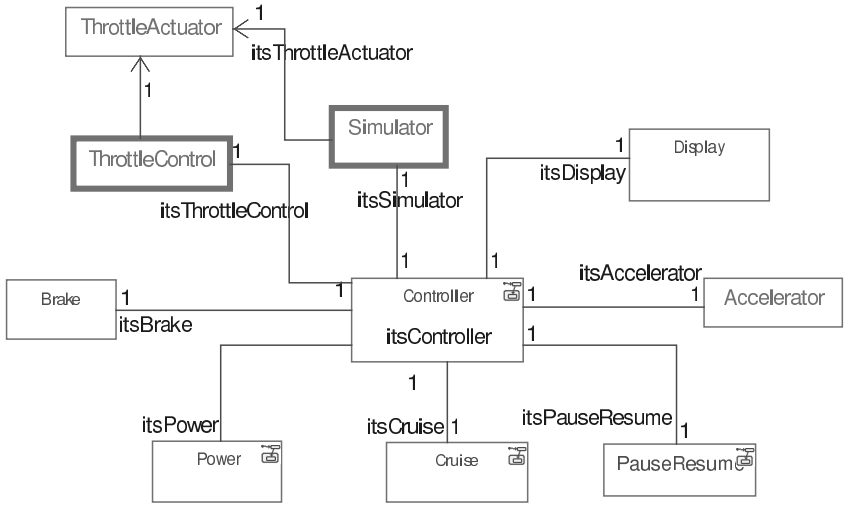
[YY88] Yung, Y.P., and Yung, Y.W., *The EDEN handbook*, Dept. of Computer Science, University of Warwick, 1988. Updated in 1996

[Wake00] Wake, W.C., *Extreme Programming Explored*, Addison Wesley, 2000

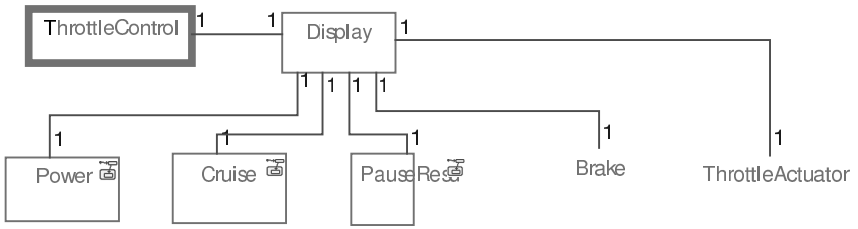
Appendix I VCCS Report

Object Model Diagrams:

CruiseControl



Display



Classes:

Accelerator

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

Brake

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

itsDisplay

Association with Display, Multiplicity of 1, Bi-directional

Operations:

getBrakePosition

Primitive-operation , Public, Return type is int

Body

```
return brakePosition;
```

setBrakePosition

Primitive-operation , Public, Return type is void

Args:

int pos

Body

```
brakePosition = pos;
```

Attributes:

brakePosition

Type of int, Private

Controller

Relations:

itsCruise

Association with Cruise, Multiplicity of 1, Bi-directional

itsDisplay

Association with Display, Multiplicity of 1, Bi-directional

itsPauseResume

Association with PauseResume, Multiplicity of 1, Bi-directional

itsPower

Association with Power, Multiplicity of 1, Bi-directional

itsSimulator

Association with Simulator, Multiplicity of 1, Bi-directional

itsThrottleControl

Association with ThrottleControl, Multiplicity of 1, Bi-directional

itsBrake

Association with Brake, Multiplicity of 1, Bi-directional

itsAccelerator

Association with Accelerator, Multiplicity of 1, Bi-directional

Operations:

evAccelOff

Event

evAccelOn

Event

evBrakeOff

Event

evBrakeOn

Event

evCruiseOff

Event

evCruiseOn

Event

evDecelOff

Event

evDecelOn

Event

evDecrease

Event

evIncrease

Event

evMaintain

Event

evPause

Event

evPowerOff

Event

evPowerOn

Event

evResume

Event

evSteady

Event

Attributes:

ccSpeed

Type of int, Public, Static, Initial Value: -1

currentSpeed

Type of double, Public, Static

desiredSpeed

Type of int, Public, Static

MAX_SPEED

Overridden Properties

Subjects:

JAVA_CG

Metaclasses:

Attribute

Properties:

MutatorGenerate: False

AccessorGenerate: False

Type of 'final int %s', Private, Static, Initial Value: 110

MIN_SPEED

Overridden Properties

Subjects:

JAVA_CG

Metaclasses:

Attribute

Properties:

MutatorGenerate: False

AccessorGenerate: False

Type of 'final int %s', Private, Static, Initial Value: 0

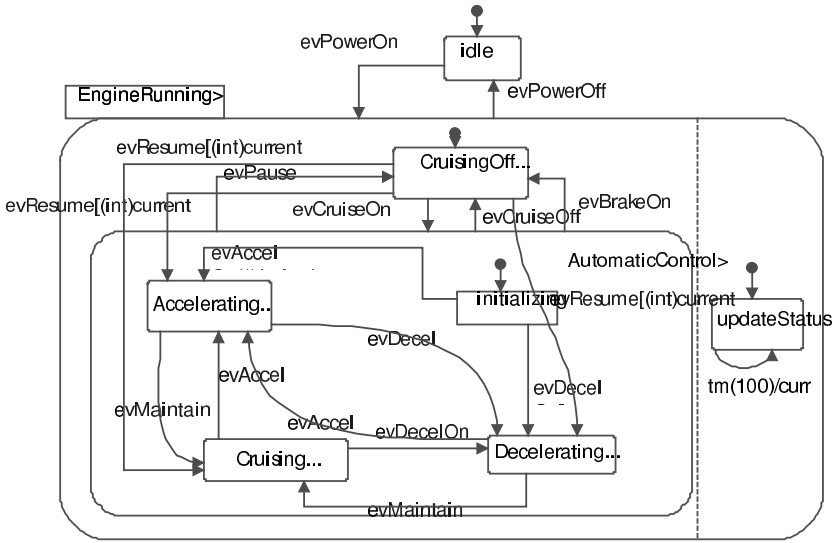
presetSpeed

Type of int, Public, Static, Initial Value: 65

resumeSpeed

Type of int, Public, Static, Initial Value: 0

Statechart



ROOT

Or-state

Substates:

EngineRunning

idle

Default Transition

Target:

idle

EngineRunning

And-state

Substates:

state_29

state_30

Out Transition

`evPowerOff`

Target:

idle

state_29

Or-state

Substates:

AutomaticControl

CruisingOff

Default Transition

Target:

CruisingOff

AutomaticControl

Or-state

Substates:

Accelerating

Cruising

Decelerating

initializing

Out Transition

```
evPause
```

Out Transition

```
evCruiseOff
```

Target:

CruisingOff

Out Transition

```
evBrakeOn
```

Target:

CruisingOff

Default Transition

Target:

initializing

Accelerating

Nested Statechart

Or-state

Out Transition

```
evDecelOn/desiredSpeed = MIN_SPEED;
```

Target:

Decelerating

Out Transition

```
evMaintain
```

Target:

Cruising

Cruising

Nested Statechart

Or-state

EntryAction

```
System.out.println("Maintaining speed..." +
ccSpeed);
```

Out Transition

```
evAccelOn/if(desiredSpeed != presetSpeed)
desiredSpeed = MAX_SPEED;
```

Target:

Accelerating

Out Transition

```
evDecelOn
```

Target:

Decelerating

Decelerating

Nested Statechart

Or-state

Out Transition

```
evMaintain
```

Target:

Cruising

Out Transition

```
evAccelOn///desiredSpeed = MAX_SPEED;
```

Target:

Accelerating

initializing

Or-state

Out Transition

```
evDecelOn[currentSpeed > 45]
```

Target:

Decelerating

Out Transition

```
evAccelOn
```

Target:

Accelerating

CruisingOff

Nested Statechart

Or-state

Out Transition

```
evResume[(int)currentSpeed < resumeSpeed]
```

Target:

Accelerating

Out Transition

```
evResume[(int)currentSpeed > resumeSpeed]
```

Target:

Decelerating

Out Transition

```
evResume[(int)currentSpeed == resumeSpeed]
```

Out Transition

```
evCruiseOn
```

Target:

AutomaticControl

state_30

Or-state

Substates:

updateStatus

Default Transition

Target:

updateStatus

updateStatus

Or-state

Out Transition

```
tm(100)/currentSpeed                                =
itsSimulator.getVehicleSpeed();

itsDisplay.throttleBar.setValue(itsThrottleContr
ol.itsThrottleActuator.getThrottlePosition());

itsDisplay.update();
```

Target:

updateStatus

idle

Or-state

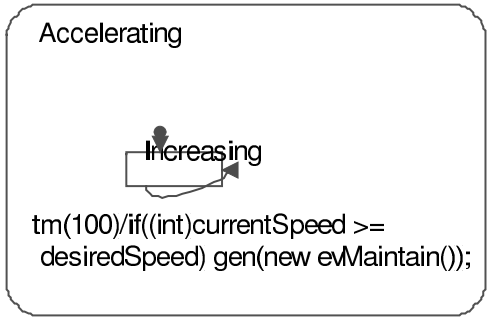
Out Transition

```
evPowerOn
```

Target:

EngineRunning

Statechart of Accelerating



ROOT

Or-state

Substates:

Accelerating

Accelerating

Or-state

Substates:

Increasing

Default Transition

Target:

Increasing

Increasing

Or-state

Out Transition

```
tm(100)/if((int)currentSpeed >=
desiredSpeed) gen(new evMaintain());
```

Target:

Increasing

Statechart of Cruising

ROOT

Or-state

Substates:

Cruising

Cruising

Or-state

Static Reaction

ROOT

Or-state

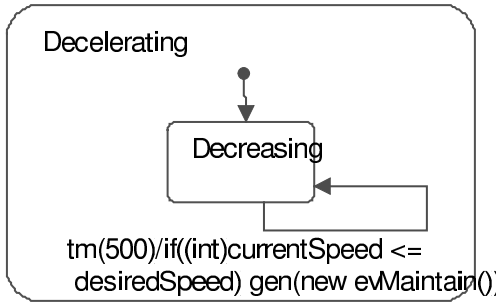
Substates:

CruisingOff

CruisingOff

Or-state

Statechart of Decelerating

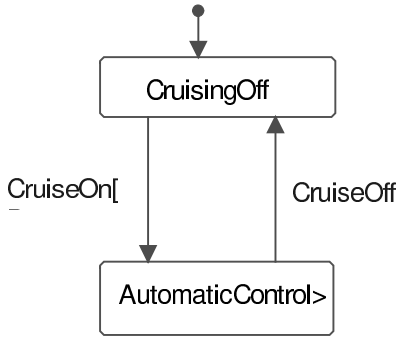


ROOT
Or-state
Substates:
Decelerating
Decelerating
Or-state
Substates:
Decreasing
Default Transition
Target:
Decreasing
Decreasing
Or-state
Out Transition
`tm(500)/if((int)currentSpeed <= desiredSpeed) gen(new evMaintain());`
Target:
Decreasing

Cruise

Relations:
itsController
Association with Controller, Multiplicity of 1, Bi-directional
itsDisplay
Association with Display, Multiplicity of 1, Bi-directional
Operations:
CruiseOff
Event
CruiseOn
Event
Attributes:
status
Type of boolean, Public, Static, Initial Value: false

Statechart



ROOT

Or-state

Substates:

AutomaticControl

CruisingOff

Default Transition

Target:

CruisingOff

AutomaticControl

Or-state

EntryAction

```
status = true;

if(Controller.ccSpeed == -1)Controller.ccSpeed =
0;

itsDisplay.cruiseButton.setText("CCOn");

itsDisplay.update();

itsController.gen(new evCruiseOn());
```

ExitAction

```
itsController.itsThrottleControl.haltControl();

status = false;

itsDisplay.cruiseButton.setText("CCOff");

itsController.gen(new evCruiseOff());
```

Out Transition

```
CruiseOff
```

Target:

CruisingOff

CruisingOff

Or-state

Out Transition

```
CruiseOn[Power.status == true &&
itsController.itsBrake.brakePosition <= 0]
```

Target:

AutomaticControl

Display

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

itsPower

Association with Power, Multiplicity of 1, Bi-directional

itsCruise

Association with Cruise, Multiplicity of 1, Bi-directional

itsPauseResume

Association with PauseResume, Multiplicity of 1, Bi-directional

itsBrake

Association with Brake, Multiplicity of 1, Bi-directional

itsThrottleActuator

Association with ThrottleActuator, Multiplicity of 1, Bi-directional

itsThrottleControl

Association with ThrottleControl, Multiplicity of 1, Bi-directional

Dependencies:

Depends on awt

Usage (Specification)

Depends on event

Usage (Specification)

Depends on swing

Usage (Specification)

Depends on event

Usage (Specification)

Operations:

accel

Primitive-operation , Public, Return type is void

Body

```
if(Cruise.status)

{

        Controller.desiredSpeed          =
Controller.MAX_SPEED;

        Controller.ccSpeed               =
Controller.desiredSpeed;

        itsController.gen(new evAccelOn());



    if (first)   //start the throttle control

   {

     itsThrottleControl.start();

     first = false;
```

```
        itsThrottleControl.terminate = false;
      } else itsThrottleControl.resumeControl();
    }
```

adjustBrake

Primitive-operation , Public, Return type is void

Args:

'ChangeEvent' e

Body

```
   itsCruise.gen(new CruiseOff());
      JSlider s;
      if (e.getSource() instanceof JSlider)
      {
       itsBrake.setBrakePosition(s.getValue());
       }
```

adjustPedal

Primitive-operation , Public, Return type is void

Args:

'ChangeEvent' e

Body

```
      JSlider s;
      if (e.getSource() instanceof JSlider)
      {
       itsThrottleActuator.setPedalPosition(s.g
   etValue());
       }
```

cruise

Primitive-operation , Public, Return type is void

Body

```
   if(Cruise.status == false)
       {
           itsCruise.gen(new CruiseOn());
       }
    else
       {
           itsCruise.gen(new CruiseOff());
           itsController.ccSpeed = -1;
       }
```

decel

Primitive-operation , Public, Return type is void

Body

```
if(Cruise.status)
{
        Controller.desiredSpeed          =
Controller.MIN_SPEED;
        Controller.ccSpeed               =
Controller.desiredSpeed;
        itsController.gen(new evDecelOn());

    if (first)    //start throttle control
   {
     itsThrottleControl.start();
     first = false;
     itsThrottleControl.terminate = false;
   } else itsThrottleControl.resumeControl();
  }
```

Display

Constructor , Public

Body

```
itsJFrame.setBounds(300,200,500,400);
upperPanel.setPreferredSize(new
Dimension(500,400));
itsJFrame.setDefaultCloseOperation(JFrame.EX
IT_ON_CLOSE);
itsJFrame.setResizable(false);

itsContainer = itsJFrame.getContentPane();
itsContainer.setLayout(new BorderLayout());
upperPanel.setLayout(itsGridLayout);

leftPanel.setLayout(new
FlowLayout(FlowLayout.LEFT,15,20));
leftPanel.setBackground(Color.black);
currentSpeedLabel.setPreferredSize(new
Dimension(100,30));
currentSpeedLabel.setFont(new
Font("TimesRoman",Font.PLAIN,15));
currentSpeedLabel.setForeground(Color.red);
leftPanel.add(currentSpeedLabel);
leftPanel.add(currentSpeedTextField);
ccLabel.setPreferredSize(new
```

```
Dimension(100,30));

ccLabel.setFont(new
Font("TimesRoman",Font.PLAIN,15));

ccLabel.setForeground(Color.red);

leftPanel.add(ccLabel);

leftPanel.add(ccTextField);

presetLabel.setPreferredSize(new
Dimension(100,30));

presetLabel.setFont(new
Font("TimesRoman",Font.PLAIN,15));

presetLabel.setForeground(Color.red);

leftPanel.add(presetLabel);

leftPanel.add(presetTextField);


powerButton.setPreferredSize(new
Dimension(90,30));

leftPanel.add(powerButton);

cruiseButton.setPreferredSize(new
Dimension(90,30));

leftPanel.add(cruiseButton);

presetButton.setPreferredSize(new
Dimension(90,30));

leftPanel.add(presetButton);

maintainPresetButton.setPreferredSize(new
Dimension(90,30));

leftPanel.add(maintainPresetButton);

pauseResumeButton.setPreferredSize(new
Dimension(195,30));

leftPanel.add(pauseResumeButton);

leftPanel.add(accelButton);

leftPanel.add(setButton);

leftPanel.add(decelButton);


    rightPanel.setLayout(new
FlowLayout(FlowLayout.CENTER,30,30));

    rightPanel.setBackground(Color.black);


   pedalSlider.setPreferredSize(new
Dimension(20,150));

   pedalSlider.setMinorTickSpacing(120);
```

```
    pedalSlider.setMajorTickSpacing(1200);
    pedalSlider.setPaintTicks(true);
    throttleBar.setPreferredSize(new
Dimension(30,150));
    throttleBar.setEnabled(true);

throttleBar.setBackground(Color.darkGray);
    throttleBar.setForeground(Color.red);
    throttleBar.setStringPainted(true);
    brakeSlider.setPreferredSize(new
Dimension(20,150));
    pedalSlider.setMinorTickSpacing(120);
    pedalSlider.setMajorTickSpacing(1200);
    pedalSlider.setPaintTicks(true);

acceleratorLabel.setForeground(Color.red);
    brakeLabel.setForeground(Color.red);
    throttleLabel.setForeground(Color.red);
    rightPanel.add(acceleratorLabel);
    rightPanel.add(throttleLabel);
    rightPanel.add(brakeLabel);
    rightPanel.add(pedalSlider);
    rightPanel.add(new JLabel(""));
    rightPanel.add(throttleBar);
    rightPanel.add(new JLabel(""));
    rightPanel.add(brakeSlider);
    rightPanel.add(airDragForceLabel);
    rightPanel.add(airDragForceBox);
    rightPanel.add(gravityForceLabel);
    rightPanel.add(gravityForceBox);

upperPanel.add(leftPanel);
upperPanel.add(rightPanel);

itsContainer.add("North",upperPanel);
itsContainer.setBackground(Color.black);

itsJFrame.setVisible(true);
initButtons();
```

initButtons

Primitive-operation , Public, Return type is void

Body

```
powerButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ power();}
  }
);

cruiseButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ cruise();}
  }
);

accelButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ accel(); }
  }
);

decelButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ decel(); }
  }
 );
pauseResumeButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ pauseResume(); }
  }
 );
setButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ setCCSpeed();}
  }
 );
presetButton.addActionListener(new
```

```
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ setPreset();}
  }
);
maintainPresetButton.addActionListener(new
java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e)
{ maintainPreset();}
  }
);

pedalSlider.addChangeListener(new
javax.swing.event.ChangeListener() {
   public void stateChanged(ChangeEvent e) {
       adjustPedal(e);

throttleBar.setValue(itsThrottleActuator.get
ThrottlePosition());
     }
   });
```

```
brakeSlider.addChangeListener(new
javax.swing.event.ChangeListener() {
   public void stateChanged(ChangeEvent e) {
         adjustBrake(e);
       }
     });
```

maintainPreset

Primitive-operation , Public, Return type is void

Body

```
if(Cruise.status)
{
    Controller.desiredSpeed                =
Controller.presetSpeed;
    Controller.ccSpeed                     =
Controller.desiredSpeed;
    if(Controller.currentSpeed             <
Controller.desiredSpeed)
itsController.gen(new evAccelOn());
    else
    if(Controller.currentSpeed             >
Controller.desiredSpeed)
```

```
itsController.gen(new evDecelOn());
    update();

    if (first)                    //start the
throttle control
   {
    itsThrottleControl.start();
    first = false;
    itsThrottleControl.terminate = false;
   } else itsThrottleControl.resumeControl();
}
```

pauseResume

Primitive-operation , Public, Return type is void

Body

```
 if(PauseResume.status == PauseResume.idle)
    {
        itsPauseResume.gen(new Pause());
    }
 else        if(PauseResume.status        ==
PauseResume.pause)
    {
        itsPauseResume.gen(new Resume());
    }
```

power

Primitive-operation , Public, Return type is void

Body

```
   if(Power.status == false)
    {
        itsPower.gen(new PowerOn());
    }
   else
    {
        itsPower.gen(new PowerOff());
    }
```

setCCSpeed

Primitive-operation , Public, Return type is void

Body

```
   itsController.gen(new evMaintain());
   Controller.ccSpeed                          =
   (int)Controller.currentSpeed;
```

setPreset

Primitive-operation , Public, Return type is void

Body

```
if(Cruise.status && Controller.ccSpeed != 0)
{
    Controller.presetSpeed = Controller.ccSpeed;
    update();
}
```

update

Primitive-operation , Public, Return type is void

Body

```
ccTextField.setText(""+ Controller.ccSpeed);
currentSpeedTextField.setText("" + Controller.currentSpeed);
presetTextField.setText("" + Controller.presetSpeed);
airDragForceBox.setText("" + itsController.itsSimulator.getAirDragForce());
gravityForceBox.setText("" + itsController.itsSimulator.getGravityForce());
```

Attributes:

accelButton

Type of 'JButton', Public, Initial Value: new JButton("" + "");

acceleratorLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Accelerator")

airDragForceBox

Type of 'JTextField', Public, Initial Value: new JTextField(7);

airDragForceLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Air Drag Force");

brakeLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Brake");

brakeSlider

Type of 'JSlider', Public, Initial Value: new JSlider(JSlider.VERTICAL,0,12000,0);

ccLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Cruise Speed");

ccSpeed

Type of int, Public

ccTextField

Type of 'JTextField', Public, Initial Value: new

JTextField(5);

cruiseButton

Type of 'JButton', Public, Initial Value: new JButton("CCOff");

cruiseLight

Type of boolean, Public

currentSpeedLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Current Speed");

currentSpeedTextField

Type of 'JTextField', Public, Initial Value: new JTextField(5);

decelButton

Type of 'JButton', Public, Initial Value: new JButton("- ");

first

Type of boolean, Public, Initial Value: true

gravityForceBox

Type of 'JTextField', Public, Initial Value: new JTextField(7);

gravityForceLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Gravity Force ");

itsContainer

Type of 'Container', Public

itsGridLayout

Type of 'GridLayout', Public, Initial Value: new GridLayout(1,4);

itsJFrame

Type of 'JFrame', Public, Initial Value: new JFrame("Display");

leftPanel

Type of 'JPanel', Public, Static, Initial Value: new JPanel()

maintainPresetButton

Type of 'JButton', Public, Initial Value: new JButton("Preset");

pauseResumeButton

Type of 'JButton', Public, Initial Value: new JButton("Pause/Resume");

pedalSlider

Type of 'JSlider', Public, Initial Value: new JSlider(JSlider.VERTICAL,0,12000,0);

powerButton

Type of 'JButton', Public, Initial Value: new JButton("PowerOff");

powerLight

Type of boolean, Public

presetButton

Type of 'JButton', Public, Initial Value: new JButton("SetPreset");

presetLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Preset Speed ");

presetTextField

Type of 'JTextField', Public, Initial Value: new JTextField(5);

rightPanel

Type of 'JPanel', Public, Initial Value: new JPanel();

setButton

Type of 'JButton', Public, Initial Value: new JButton(" = ");

sliderTable

Type of 'Hashtable', Public, Initial Value: new Hashtable();

throttleBar

Type of 'JProgressBar', Public, Initial Value: new JProgressBar(JProgressBar.VERTICAL,0,12000);

throttleLabel

Type of 'JLabel', Public, Initial Value: new JLabel("Throttle");

upperPanel

Type of 'JPanel', Public, Initial Value: new JPanel();

PauseResume

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

itsDisplay

Association with Display, Multiplicity of 1, Bi-directional

Operations:

evIdle

Event

Pause

Event

Resume

Event

Attributes:

idle

Type of 'final int %s', Private, Static, Initial Value: 1

pause

Type of 'final int %s', Private, Static, Initial Value: 2

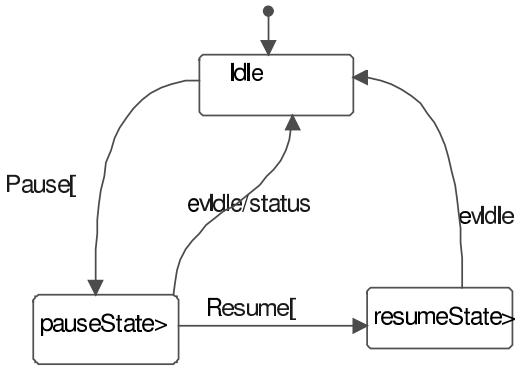
resume

Type of 'final int %s', Private, Static, Initial Value: 3

status

Type of int, Private, Static, Initial Value: idle

Statechart



ROOT

Or-state

Substates:

Idle

pauseState

resumeState

Default Transition

Target:

Idle

Idle

Or-state

Out Transition

```
Pause[Cruise.status == true && itsController.ccSpeed != 0]
```

Target:

pauseState

pauseState

Or-state

EntryAction

```
status = pause;

if (Controller.ccSpeed == Controller.MAX_SPEED ||
Controller.ccSpeed == Controller.MIN_SPEED)

   Controller.ccSpeed =
(int)Controller.currentSpeed;


Controller.resumeSpeed = (int)Controller.ccSpeed;

itsController.itsCruise.gen(new CruiseOff());
```

Out Transition

```
Resume[Power.status == true]
```

Target:

resumeState

Out Transition

```
evIdle/status = idle;
```

Target:

Idle

resumeState

Or-state

EntryAction

```
if (itsDisplay.first)                //start the
throttle control
{
     itsController.itsThrottleControl.start();
     itsDisplay.first = false;
     itsController.itsThrottleControl.terminate
= false;
}                                              else
itsController.itsThrottleControl.resumeControl()
;

itsController.itsSimulator.resumeSim();
status = resume;
Controller.desiredSpeed = Controller.resumeSpeed;
itsController.itsDisplay.update();
itsController.itsCruise.gen(new CruiseOn());
itsController.gen(new evResume());
gen(new evIdle());
```

ExitAction

```
status = idle;
Controller.resumeSpeed = 0;
```

Out Transition

```
evIdle
```

Target:

Idle

Power

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

itsDisplay

Association with Display, Multiplicity of 1, Bi-directional

Operations:

PowerOff

Event

PowerOn

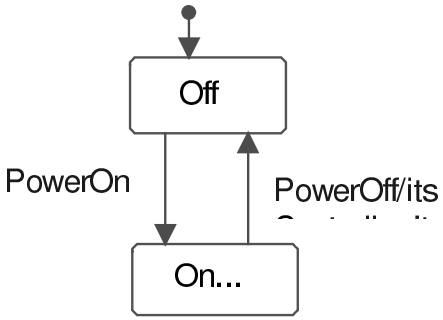
Event

Attributes:

status

Type of boolean, Public, Static, Initial Value: false

Statechart



ROOT

Or-state

Substates:

Off

On

Default Transition

Target:

Off

Off

Or-state

Out Transition

```
PowerOn
```

Target:

On

On

Or-state

EntryAction

```
itsController.itsSimulator.start();

itsController.itsSimulator.terminate = false;

status = true;

itsDisplay.powerButton.setText("PowerOn");

itsController.gen(new evPowerOn());
```

ExitAction

```
itsController.itsSimulator.terminateSim();

status = false;

if(PauseResume.status    ==    PauseResume.pause)
itsController.itsPauseResume.gen(new evIdle());

itsDisplay.cruise();

itsController.gen(new evPowerOff());

itsDisplay.first = true;
```

Out Transition

```
PowerOff/itsController.itsThrottleControl.termin
ateControl();

itsDisplay.pedalSlider.setValue(0);

itsController.itsThrottleControl.itsThrottleActu
ator.setPedalPosition(0);
```

```
itsDisplay.powerButton.setText("PowerOff");

itsDisplay.currentSpeedTextField.setText("--");

itsDisplay.ccTextField.setText("--");

itsDisplay.presetTextField.setText("--");
```

Target:

Off

Simulator

Overridden Properties

Subjects:

CG

Metaclasses:

Class

Properties:

Concurrency: active

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

itsState

Association with State, Multiplicity of 1, Uni-directional

itsThrottleActuator

Association with ThrottleActuator, Multiplicity of 1, Uni-directional

Superclasses:

Thread

Public

Operations:

calcNewVehicleState

Primitive-operation , Public, Return type is void

Body

```
State k1 = new State();

State k2 = new State();

State k3 = new State();

State k4 = new State();

State tempState = new State();

double dts = dt/1000.0;

 k1 = eom(state);

 k1.position = k1.position * dts;

 k1.velocity = k1.velocity * dts;



 tempState.position  =  state.position  +
k1.position / 2.0;

 tempState.velocity  =  state.velocity  +
k1.velocity / 2.0;
```

```
    k2 = eom(tempState);
    k2.position *= dts;
    k2.velocity *= dts;

    tempState.position = state.position +
k2.position / 2.0;
    tempState.velocity = state.velocity +
k2.velocity / 2.0;
    k3 = eom(tempState);
    k3.position *= dts;
    k3.velocity *= dts;

    tempState.position = state.position +
k3.position;
    tempState.velocity = state.velocity +
k3.velocity;
    k4 = eom(tempState);
    k4.position *= dts;
    k4.velocity *= dts;

    state.position = state.position +
1.0/6.0*(k1.position + 2.0*k2.position +
2.0*k3.position + k4.position);
    state.velocity = state.velocity +
1.0/6.0*(k1.velocity + 2.0*k2.velocity +
2.0*k3.velocity + k4.velocity);
```

eom

Primitive-operation , Public, Return type is 'State'

Args:

'State' s

Body

```
   double inputForce =
itsThrottleActuator.getInputForce(); //
input force [lbf-ft/s/s]
   airDragForce =
getAirDragForce(s.velocity);
   gravityForce =
getGravityForce(s.position);

   double acceleration =
(1.0/mass)*((double)inputForce-airDragForce-
```

```
gravityForce                                    -
(double)itsController.itsBrake.getBrakePosit
ion() );

    State stateRate = new State();

    stateRate.position = s.velocity;

    stateRate.velocity = acceleration;

    return stateRate;
```

getAirDragForce

Primitive-operation , Public, Return type is double

Args:

double s1

Body

```
    return alpha*s1*(double)Math.abs(s1);
```

getAirDragForce

Primitive-operation , Public, Return type is double

Body

```
return airDragForce;
```

getGravityForce

Primitive-operation , Public, Return type is double

Body

```
return gravityForce;
```

getGravityForce

Primitive-operation , Public, Return type is double

Args:

double s

Body

```
    double            calcForce            =
1.0/50.0*(Math.sin(s/sinConst)-Math.cos(s/co
sConst)+1.0)*mass*g;

     return calcForce;
```

getVehicleSpeed

Primitive-operation , Public, Return type is double

Body

```
return speed;
```

haltSim

Primitive-operation , Public, Return type is void

Body

```
halt = true;
```

resumeSim

Primitive-operation , Public, Return type is 'synchronized void'

Body

```
    halt = false;

    if (!isRunning) notify();
```

run

Primitive-operation , Public, Return type is void

Body

```
halt = false;

isRunning = true;

simulate();
```

simulate

Primitive-operation , Public, Return type is void

Body

```
Date oldDate = new Date();

Date date;

double oldSpeed = -1.0;          // new
date object

while (!terminate)

{

date = new Date();

dt      =      date.getTime()      -
oldDate.getTime();

calcNewVehicleState();

speed = (state.velocity / 5280 * 3600);


if(speed    <    0)    {terminateSim();
itsController.itsPower.gen(new PowerOff());}

oldDate = date;


if (halt)

{

suspendSim();

oldDate = new Date();

} else try

{

sleep(80);

} catch (InterruptedException e) {}


}
```

Simulator

Constructor , Public

Body

```
dt = 200;

mass = 2000;

Date date = new Date();
```

```
startTime = date.getTime();

lastUpdate = startTime;

state = new State();

halt = true;

terminate = false;

isRunning = false;
```

suspendSim

Primitive-operation , Public, Return type is 'synchronized void'

Body

```
{

  isRunning = false;

  try {

    wait();

  } catch (Exception e) {}

}
```

terminateSim

Primitive-operation , Public, Return type is void

Body

```
terminate = true;

haltSim();

state.position = 0;

state.velocity = 0;


airDragForce = 0;

gravityForce = 0;

speed = 0;
```

Attributes:

airDragForce

Type of double, Public

alpha

Type of 'final double %s', Private, Static, Initial Value: 0.2

cosConst

Type of double, Private, Static, Initial Value: 5018

dt

Type of long, Private

g

Type of 'final double %s', Private, Initial Value: 32.2

gravityForce

Type of double, Public

halt

Type of boolean, Private

isRunning
Type of boolean, Private
lastUpdate
Type of long, Private
mass
Type of double, Private
name
Type of String, Private
sinConst
Type of double, Private, Initial Value: 1010
speed
Type of double, Private
startTime
Type of long, Private
state
Type of 'State', Protected
terminate
Type of boolean, Private

State

Operations:

State
Constructor , Public
Args:
'State' s
Body

```
position = s.position;

velocity = s.velocity;
```

State
Constructor , Public
Body

```
position = 0.0;

velocity = 0.0;
```

Attributes:

position
Type of double, Public
velocity
Type of double, Public

ThrottleActuator

Relations:

itsDisplay
Association with Display, Multiplicity of 1, Bi-directional

Dependencies:

Depends on awt
Usage (Specification)
Depends on event
Usage (Specification)

Depends on swing

Usage (Specification)

Operations:

adjust

Primitive-operation , Public, Return type is void

Args:

int newPosition

Body

```
int pos;

if (pedalPosition > controlPosition) pos =
pedalPosition;

  else pos = controlPosition;

if (newPosition > pos) throttlePosition =
newPosition;

  else throttlePosition = pos;
```

getInputForce

Primitive-operation , Public, Return type is double

Body

```
return (double)throttlePosition;
```

getMaxPosition

Primitive-operation , Public, Return type is int

Body

```
return maxPosition;
```

getMinPosition

Primitive-operation , Public, Return type is int

Body

```
return minPosition;
```

getPedalPosition

Primitive-operation , Public, Return type is int

Body

```
return pedalPosition;
```

getThrottlePosition

Primitive-operation , Public, Return type is int

Body

```
return throttlePosition;
```

setControlPosition

Primitive-operation , Public, Return type is void

Args:

int pos

Body

```
// Invoke upper limit on throttle position

if ( pos > maxPosition )

  pos = maxPosition;


 // Invoke lower limit on throttle position
```

```
if ( pos < minPosition )
   pos = minPosition;
controlPosition = pos;
adjust(controlPosition);
```

setPedalPosition

Primitive-operation , Public, Return type is void

Args:

int pos

Body

```
if (pos > maxPosition) pedalPosition =
maxPosition;
  else if (pos < minPosition) pedalPosition
= minPosition;
  else pedalPosition = pos;
adjust(pedalPosition);
```

ThrottleActuator

Constructor , Public

Body

```
maxPosition = 12000;
minPosition = 0;
throttlePosition = 0;
pedalPosition = 0;
```

Attributes:

controlPosition

Type of int, Private

maxPosition

Type of int, Private

minPosition

Type of int, Private

pedalPosition

Type of int, Private

throttlePosition

Type of int, Private

ThrottleControl

Overridden Properties

Subjects:

CG

Metaclasses:

Class

Properties:

Concurrency: active

Relations:

itsController

Association with Controller, Multiplicity of 1, Bi-directional

itsThrottleActuator

Association with ThrottleActuator, Multiplicity of 1, Uni-directional

itsDisplay

Association with Display, Multiplicity of 1, Bi-directional

Dependencies:

Depends on awt

Usage (Specification)

Depends on event

Usage (Specification)

Depends on swing

Usage (Specification)

Superclasses:

Thread

Public

Operations:

calcError

Primitive-operation , Public, Return type is double

Body

```
double e = itsController.ccSpeed*5280.0/3600.0-itsController.currentSpeed*5280.0/3600.0;  // error

return e;
```

calcThrotPos

Primitive-operation , Public, Return type is int

Args:

double error

double dt

Body

```
double dPositionDotMax = 4000.0;  // max change in throttle position per second [inches * 1e4]

double resetDiff = 5.0;   //   minimum error before the integralSum gets reset.
                          //  this helps forget the integral history from when the car
                          //  is slowing down from air drag and gravity forces


// reset integralSum when vehicle speed is greater than resetDiff above ccSpeed
if (-error > resetDiff)
  integralSum = 0.0;
else
  integralSum += (oldError + error)/2.0*dt;
```

```
  double fp = (error)*Kp;       //
proportional control input term

  double  fd  =  (error  -  oldError)/dt*Kd;
//derivative control input term

    double fi = integralSum*Ki;    //
integral control input term


// calculate input (desired throttle position)

    int input = (int)(fp+fd+fi);


// set input to zero if negative -- cannot control
the throttle position negative

    if (input < 0) input = 0;


// the desired throttle position is proportional
to the input, and since the input

    //  is hidded from the user, the desired
throttle position will just BE the input

    int dPosition = input;


    // limit rate of change of desired throttle
position

    int           currentPosition          =
itsThrottleActuator.getThrottlePosition();

    int            dPositionDot            =
dPosition-currentPosition;

    if  (Math.abs((double)dPositionDot/dt)  >
dPositionDotMax)

       dPosition    =    currentPosition    +
(int)((double)dPositionDotMax                *
(double)sign((double)(dPositionDot)) * dt);


   oldError = error;

    return dPosition;
```

destroy

Primitive-operation , Public, Return type is void

Body

```
    integralSum = 0.0;

    oldError = 0.0;

    controlPosition                          =
itsThrottleActuator.getMinPosition();


itsThrottleActuator.setControlPosition(contr
```

```
olPosition);
```

haltControl

Primitive-operation , Public, Return type is void

Body

```
halt = true;
```

maintainSpeed

Primitive-operation , Public, Return type is void

Body

```
vSpeed = itsController.currentSpeed;

controlPosition = calcThrotPos(calcError(), dt );

if (goodPos)
  {
    if (controlPosition > (lastPos + 120)) controlPosition = lastPos + 120;
    else if (controlPosition < (lastPos - 1000)) controlPosition = lastPos - 1000;
  } else goodPos = true;
  lastPos = controlPosition;

itsThrottleActuator.setControlPosition(controlPosition);
```

```
try
{
    sleep(80);
} catch (InterruptedException e) {}
```

resumeControl

Primitive-operation , Public, Return type is 'synchronized void'

Body

```
halt = false;

if (!isRunning) notify();
```

run

Primitive-operation , Public, Return type is void

Body

```
Date date;
  Date oldDate = new Date();
  halt = false;
  isRunning = true;
  while (!terminate)
{
    date = new Date();
    dt = ((double)(date.getTime() -
```

```
oldDate.getTime())) / 1000.0;

        oldDate = date;

        maintainSpeed();

        if (halt) suspendControl();

    }
```

sign

Primitive-operation , Public, Return type is int

Args:

double x

Body

```
if (x < 0.0) return(-1);

  else if (x == 0.0)

      return(0);

  else

      return(1);
```

suspendControl

Primitive-operation , Public, Return type is 'synchronized void'

Body

```
integralSum = 0.0;

oldError = 0.0;

int          controlPosition          =
itsThrottleActuator.getMinPosition();


itsThrottleActuator.setControlPosition(contr
olPosition);

   while (halt)

   {

     isRunning = false;

     try {

       wait();

     } catch (Exception e) {}

   }
```

terminateControl

Primitive-operation , Public, Return type is void

Body

```
terminate = true;

haltControl();

itsThrottleActuator.setPedalPosition(0);
```

ThrottleControl

Constructor , Public

Body

```
halt = true;
```

```
        terminate = false;

        isRunning = false;
```

Attributes:

controlPosition
Type of int, Private

dt
Type of double, Private

goodPos
Type of boolean, Private

halt
Type of boolean, Private

integralSum
Type of double, Private, Initial Value: 0.0

isRunning
Type of boolean, Private

Kd
Type of double, Private, Initial Value: 1000.0

Ki
Type of double, Private, Initial Value: 5.0

Kp
Type of double, Private, Initial Value: 4500.0

lastPos
Type of int, Private, Initial Value: 0

oldError
Type of double, Private, Initial Value: 0.0

terminate
Type of boolean, Private

vSpeed
Type of double, Private

COMPONENTS

DefaultComponent

COMPONENT SETTINGS:

Build type: Executable

CONFIGURATIONS:

DefaultConfig

Scope type: Derived
Instrumentation type: Animation
Time-model type: Real-time
Statechart generation type: Flat
Libraries: Gui
Additional sorces: ..\guiLib\Lib.lib

Initialization code:

```
p_Brake.setItsController(p_Controller);

p_Power.setItsController(p_Controller);

p_Controller.setItsCruise(p_Cruise);

p_Controller.setItsPauseResume(p_PauseResume);

p_Controller.setItsDisplay(p_Display);

p_Controller.setItsSimulator(p_Simulator);

p_Controller.setItsThrottleControl(p_ThrottleControl);

p_Simulator.setItsState(p_State);

p_Simulator.setItsThrottleActuator(p_ThrottleActuator)
;

p_ThrottleControl.setItsThrottleActuator(p_ThrottleAct
uator);


p_Display.setItsPower(p_Power);

p_Display.setItsCruise(p_Cruise);

p_Display.setItsBrake(p_Brake);

p_Display.setItsPauseResume(p_PauseResume);

p_Display.setItsThrottleActuator(p_ThrottleActuator);

p_Display.setItsThrottleControl(p_ThrottleControl);
```

FILES AND FOLDERS:

Explicit scope:

Package: CruiseControl

Appendix II EM Draughts Script

```
%donald

################################################################
##
##    DoNaLD script
##
##    defines the "visualisation" aspect of the definitive system.
##
################################################################

# set up the necessary viewports and circles to represent the
# pieces and their occupying squares
viewport piece110
circle Piece110
circle KingDot110

viewport piece108
circle Piece108
circle KingDot108

viewport piece106
circle Piece106
circle KingDot106

viewport piece104
circle Piece104
circle KingDot104

viewport piece102
circle Piece102
circle KingDot102

viewport piece99
circle Piece99
circle KingDot99

viewport piece97
circle Piece97
circle KingDot97

viewport piece95
circle Piece95
circle KingDot95

viewport piece93
circle Piece93
circle KingDot93

viewport piece91
circle Piece91
circle KingDot91

viewport piece90
circle Piece90
circle KingDot90

viewport piece88
circle Piece88
circle KingDot88

viewport piece86
circle Piece86
circle KingDot86

viewport piece84
circle Piece84
circle KingDot84

viewport piece82
circle Piece82
circle KingDot82
```

viewport piece79
circle Piece79
circle KingDot79

viewport piece77
circle Piece77
circle KingDot77

viewport piece75
circle Piece75
circle KingDot75

viewport piece73
circle Piece73
circle KingDot73

viewport piece71
circle Piece71
circle KingDot71

viewport piece70
circle Piece70
circle KingDot70

viewport piece68
circle Piece68
circle KingDot68

viewport piece66
circle Piece66
circle KingDot66

viewport piece64
circle Piece64
circle KingDot64

viewport piece62
circle Piece62
circle KingDot62

viewport piece59
circle Piece59
circle KingDot59

viewport piece57
circle Piece57
circle KingDot57

viewport piece55
circle Piece55
circle KingDot55

viewport piece53
circle Piece53
circle KingDot53

viewport piece51
circle Piece51
circle KingDot51

viewport piece50
circle Piece50
circle KingDot50

viewport piece48
circle Piece48
circle KingDot48

viewport piece46
circle Piece46
circle KingDot46

viewport piece44
circle Piece44
circle KingDot44

viewport piece42
circle Piece42
circle KingDot42

```
viewport piece39
circle Piece39
circle KingDot39

viewport piece37
circle Piece37
circle KingDot37

viewport piece35
circle Piece35
circle KingDot35

viewport piece33
circle Piece33
circle KingDot33

viewport piece31
circle Piece31
circle KingDot31

viewport piece30
circle Piece30
circle KingDot30

viewport piece28
circle Piece28
circle KingDot28

viewport piece26
circle Piece26
circle KingDot26

viewport piece24
circle Piece24
circle KingDot24

viewport piece22
circle Piece22
circle KingDot22

viewport piece19
circle Piece19
circle KingDot19

viewport piece17
circle Piece17
circle KingDot17

viewport piece15
circle Piece15
circle KingDot15

viewport piece13
circle Piece13
circle KingDot13

viewport piece11
circle Piece11
circle KingDot11

# Set up the position and radius of each disc, relative
# to the viewport that it occupies

point discpos
int rad
discpos = {500,500}
rad = 400

Piece110 = circle(discpos,rad)
Piece108 = circle(discpos,rad)
Piece106 = circle(discpos,rad)
Piece104 = circle(discpos,rad)
Piece102 = circle(discpos,rad)

Piece99 = circle(discpos,rad)
Piece97 = circle(discpos,rad)
Piece95 = circle(discpos,rad)
Piece93 = circle(discpos,rad)
Piece91 = circle(discpos,rad)

Piece90 = circle(discpos,rad)
Piece88 = circle(discpos,rad)
```

```
Piece86 = circle(discpos,rad)
Piece84 = circle(discpos,rad)
Piece82 = circle(discpos,rad)

Piece79 = circle(discpos,rad)
Piece77 = circle(discpos,rad)
Piece75 = circle(discpos,rad)
Piece73 = circle(discpos,rad)
Piece71 = circle(discpos,rad)

Piece70 = circle(discpos,rad)
Piece68 = circle(discpos,rad)
Piece66 = circle(discpos,rad)
Piece64 = circle(discpos,rad)
Piece62 = circle(discpos,rad)

Piece59 = circle(discpos,rad)
Piece57 = circle(discpos,rad)
Piece55 = circle(discpos,rad)
Piece53 = circle(discpos,rad)
Piece51 = circle(discpos,rad)

Piece50 = circle(discpos,rad)
Piece48 = circle(discpos,rad)
Piece46 = circle(discpos,rad)
Piece44 = circle(discpos,rad)
Piece42 = circle(discpos,rad)

Piece39 = circle(discpos,rad)
Piece37 = circle(discpos,rad)
Piece35 = circle(discpos,rad)
Piece33 = circle(discpos,rad)
Piece31 = circle(discpos,rad)

Piece30 = circle(discpos,rad)
Piece28 = circle(discpos,rad)
Piece26 = circle(discpos,rad)
Piece24 = circle(discpos,rad)
Piece22 = circle(discpos,rad)

Piece19 = circle(discpos,rad)
Piece17 = circle(discpos,rad)
Piece15 = circle(discpos,rad)
Piece13 = circle(discpos,rad)
Piece11 = circle(discpos,rad)

point kingpos
int kingrad
kingpos = {500,500}
kingrad = 100

KingDot110 = circle(kingpos,kingrad)
KingDot108 = circle(kingpos,kingrad)
KingDot106 = circle(kingpos,kingrad)
KingDot104 = circle(kingpos,kingrad)
KingDot102 = circle(kingpos,kingrad)

KingDot99 = circle(kingpos,kingrad)
KingDot97 = circle(kingpos,kingrad)
KingDot95 = circle(kingpos,kingrad)
KingDot93 = circle(kingpos,kingrad)
KingDot91 = circle(kingpos,kingrad)

KingDot90 = circle(kingpos,kingrad)
KingDot88 = circle(kingpos,kingrad)
KingDot86 = circle(kingpos,kingrad)
KingDot84 = circle(kingpos,kingrad)
KingDot82 = circle(kingpos,kingrad)

KingDot79 = circle(kingpos,kingrad)
KingDot77 = circle(kingpos,kingrad)
KingDot75 = circle(kingpos,kingrad)
KingDot73 = circle(kingpos,kingrad)
KingDot71 = circle(kingpos,kingrad)

KingDot70 = circle(kingpos,kingrad)
KingDot68 = circle(kingpos,kingrad)
KingDot66 = circle(kingpos,kingrad)
KingDot64 = circle(kingpos,kingrad)
KingDot62 = circle(kingpos,kingrad)

KingDot59 = circle(kingpos,kingrad)
```

```
KingDot57 = circle(kingpos,kingrad)
KingDot55 = circle(kingpos,kingrad)
KingDot53 = circle(kingpos,kingrad)
KingDot51 = circle(kingpos,kingrad)

KingDot50 = circle(kingpos,kingrad)
KingDot48 = circle(kingpos,kingrad)
KingDot46 = circle(kingpos,kingrad)
KingDot44 = circle(kingpos,kingrad)
KingDot42 = circle(kingpos,kingrad)

KingDot39 = circle(kingpos,kingrad)
KingDot37 = circle(kingpos,kingrad)
KingDot35 = circle(kingpos,kingrad)
KingDot33 = circle(kingpos,kingrad)
KingDot31 = circle(kingpos,kingrad)

KingDot30 = circle(kingpos,kingrad)
KingDot28 = circle(kingpos,kingrad)
KingDot26 = circle(kingpos,kingrad)
KingDot24 = circle(kingpos,kingrad)
KingDot22 = circle(kingpos,kingrad)

KingDot19 = circle(kingpos,kingrad)
KingDot17 = circle(kingpos,kingrad)
KingDot15 = circle(kingpos,kingrad)
KingDot13 = circle(kingpos,kingrad)
KingDot11 = circle(kingpos,kingrad)
%scout

######################################################################
##
##    SCOUT script
##
##    The definitions that are required to describe the screen layout
##
######################################################################

# Create the window for the draughts board
window board;

# Define the background colour for the squares
string bgcol;


# set up the points p and q, which act as reference points for the
# top-left and bottom-right coordinates for each window
# respectively
point p;
point q;


# set up the top-left and bottom-right coordinates for each of the
# individual squares
point p11;
point p13;
point p15;
point p17;
point p19;

point p22;
point p24;
point p26;
point p28;
point p30;

point p31;
point p33;
point p35;
point p37;
point p39;

point p42;
point p44;
point p46;
point p48;
point p50;

point p51;
point p53;
point p55;
point p57;
```

point p59;

point p62;
point p64;
point p66;
point p68;
point p70;

point p71;
point p73;
point p75;
point p77;
point p79;

point p82;
point p84;
point p86;
point p88;
point p90;

point p91;
point p93;
point p95;
point p97;
point p99;

point p102;
point p104;
point p106;
point p108;
point p110;

point q11;
point q13;
point q15;
point q17;
point q19;

point q22;
point q24;
point q26;
point q28;
point q30;

point q31;
point q33;
point q35;
point q37;
point q39;

point q42;
point q44;
point q46;
point q48;
point q50;

point q51;
point q53;
point q55;
point q57;
point q59;

point q62;
point q64;
point q66;
point q68;
point q70;

point q71;
point q73;
point q75;
point q77;
point q79;

point q82;
point q84;
point q86;
point q88;
point q90;

point q91;

```
point q93;
point q95;
point q97;
point q99;

point q102;
point q104;
point q106;
point q108;
point q110;


# Set up 32 windows to represent the 32 squares in draughts
window square11;
window square13;
window square15;
window square17;
window square19;

window square22;
window square24;
window square26;
window square28;
window square30;

window square31;
window square33;
window square35;
window square37;
window square39;

window square42;
window square44;
window square46;
window square48;
window square50;

window square51;
window square53;
window square55;
window square57;
window square59;

window square62;
window square64;
window square66;
window square68;
window square70;

window square71;
window square73;
window square75;
window square77;
window square79;

window square82;
window square84;
window square86;
window square88;
window square90;

window square91;
window square93;
window square95;
window square97;
window square99;

window square102;
window square104;
window square106;
window square108;
window square110;


# Set up some integer variables to hold the co-ordinates of
# the square, with respect to the board, eg (3,4).
integer x11;
integer x13;
integer x15;
integer x17;
integer x19;
```

```
integer x22;
integer x24;
integer x26;
integer x28;
integer x30;

integer x31;
integer x33;
integer x35;
integer x37;
integer x39;

integer x42;
integer x44;
integer x46;
integer x48;
integer x50;

integer x51;
integer x53;
integer x55;
integer x57;
integer x59;

integer x62;
integer x64;
integer x66;
integer x68;
integer x70;

integer x71;
integer x73;
integer x75;
integer x77;
integer x79;

integer x82;
integer x84;
integer x86;
integer x88;
integer x90;

integer x91;
integer x93;
integer x95;
integer x97;
integer x99;

integer x102;
integer x104;
integer x106;
integer x108;
integer x110;

integer y11;
integer y13;
integer y15;
integer y17;
integer y19;

integer y22;
integer y24;
integer y26;
integer y28;
integer y30;

integer y31;
integer y33;
integer y35;
integer y37;
integer y39;

integer y42;
integer y44;
integer y46;
integer y48;
integer y50;

integer y51;
integer y53;
integer y55;
integer y57;
```

```
integer y59;

integer y62;
integer y64;
integer y66;
integer y68;
integer y70;

integer y71;
integer y73;
integer y75;
integer y77;
integer y79;

integer y82;
integer y84;
integer y86;
integer y88;
integer y90;

integer y91;
integer y93;
integer y95;
integer y97;
integer y99;

integer y102;
integer y104;
integer y106;
integer y108;
integer y110;

# Calculate the various co-ordinates for when we define the window
# later in this script
integer sqheight;       # height of a square
integer sqwidth;        # width of a square
integer height;             # height of the board
integer width;              # width of the board
real   pxoffset;        # offset for topleft x-coord
real   pyoffset;        # offset for topleft y-coord
real   qxoffset;        # offset for bottomright x-coord
real   qyoffset;        # offset for bottomright y-coord

integer sqheight = 40;
integer sqwidth = 40;
integer height = 10 * sqheight;
integer width = 10 * sqwidth;
real   pxoffset = 1;
real   pyoffset = 1;
real   qxoffset = -1;
real   qyoffset = -1;
point p = {10,10};
point q = {10 + width, 10 + height};

# Define the top-left co-ordinates of each window and
# define the bottom-right co-ordinates of each window too
# (p and q respectively)

point p11 = { ( x11 - 1 ) * sqwidth + pxoffset , ( 10 - y11 ) * sqheight + pyoffset } + p;
point q11 = { ( x11 * sqwidth ) + qxoffset , ( ( 11 - y11 ) * sqheight ) + qyoffset } + p;
point p13 = { ( x13 - 1 ) * sqwidth + pxoffset , ( 10 - y13 ) * sqheight + pyoffset } + p;
point q13 = { ( x13 * sqwidth ) + qxoffset , ( ( 11 - y13 ) * sqheight ) + qyoffset } + p;
point p15 = { ( x15 - 1 ) * sqwidth + pxoffset , ( 10 - y15 ) * sqheight + pyoffset } + p;
point q15 = { ( x15 * sqwidth ) + qxoffset , ( ( 11 - y15 ) * sqheight ) + qyoffset } + p;
point p17 = { ( x17 - 1 ) * sqwidth + pxoffset , ( 10 - y17 ) * sqheight + pyoffset } + p;
point q17 = { ( x17 * sqwidth ) + qxoffset , ( ( 11 - y17 ) * sqheight ) + qyoffset } + p;
point p19 = { ( x19 - 1 ) * sqwidth + pxoffset , ( 10 - y19 ) * sqheight + pyoffset } + p;
point q19 = { ( x19 * sqwidth ) + qxoffset , ( ( 11 - y19 ) * sqheight ) + qyoffset } + p;


point p22 = { ( x22 - 1 ) * sqwidth + pxoffset , ( 10 - y22 ) * sqheight + pyoffset } + p;
point q22 = { ( x22 * sqwidth ) + qxoffset , ( ( 11 - y22 ) * sqheight ) + qyoffset } + p;
point p24 = { ( x24 - 1 ) * sqwidth + pxoffset , ( 10 - y24 ) * sqheight + pyoffset } + p;
point q24 = { ( x24 * sqwidth ) + qxoffset , ( ( 11 - y24 ) * sqheight ) + qyoffset } + p;
point p26 = { ( x26 - 1 ) * sqwidth + pxoffset , ( 10 - y26 ) * sqheight + pyoffset } + p;
point q26 = { ( x26 * sqwidth ) + qxoffset , ( ( 11 - y26 ) * sqheight ) + qyoffset } + p;
point p28 = { ( x28 - 1 ) * sqwidth + pxoffset , ( 10 - y28 ) * sqheight + pyoffset } + p;
point q28 = { ( x28 * sqwidth ) + qxoffset , ( ( 11 - y28 ) * sqheight ) + qyoffset } + p;
point p30 = { ( x30 - 1 ) * sqwidth + pxoffset , ( 10 - y30 ) * sqheight + pyoffset } + p;
point q30 = { ( x30 * sqwidth ) + qxoffset , ( ( 11 - y30 ) * sqheight ) + qyoffset } + p;
```

```
point p31 = { ( x31 - 1 ) * sqwidth + pxoffset , ( 10 - y31 ) * sqheight + pyoffset } + p;
point q31 = { ( x31 * sqwidth ) + qxoffset , ( ( 11 - y31 ) * sqheight ) + qyoffset } + p;
point p33 = { ( x33 - 1 ) * sqwidth + pxoffset , ( 10 - y33 ) * sqheight + pyoffset } + p;
point q33 = { ( x33 * sqwidth ) + qxoffset , ( ( 11 - y33 ) * sqheight ) + qyoffset } + p;
point p35 = { ( x35 - 1 ) * sqwidth + pxoffset , ( 10 - y35 ) * sqheight + pyoffset } + p;
point q35 = { ( x35 * sqwidth ) + qxoffset , ( ( 11 - y35 ) * sqheight ) + qyoffset } + p;
point p37 = { ( x37 - 1 ) * sqwidth + pxoffset , ( 10 - y37 ) * sqheight + pyoffset } + p;
point q37 = { ( x37 * sqwidth ) + qxoffset , ( ( 11 - y37 ) * sqheight ) + qyoffset } + p;
point p39 = { ( x39 - 1 ) * sqwidth + pxoffset , ( 10 - y39 ) * sqheight + pyoffset } + p;
point q39 = { ( x39 * sqwidth ) + qxoffset , ( ( 11 - y39 ) * sqheight ) + qyoffset } + p;


point p42 = { ( x42 - 1 ) * sqwidth + pxoffset , ( 10 - y42 ) * sqheight + pyoffset } + p;
point q42 = { ( x42 * sqwidth ) + qxoffset , ( ( 11 - y42 ) * sqheight ) + qyoffset } + p;
point p44 = { ( x44 - 1 ) * sqwidth + pxoffset , ( 10 - y44 ) * sqheight + pyoffset } + p;
point q44 = { ( x44 * sqwidth ) + qxoffset , ( ( 11 - y44 ) * sqheight ) + qyoffset } + p;
point p46 = { ( x46 - 1 ) * sqwidth + pxoffset , ( 10 - y46 ) * sqheight + pyoffset } + p;
point q46 = { ( x46 * sqwidth ) + qxoffset , ( ( 11 - y46 ) * sqheight ) + qyoffset } + p;
point p48 = { ( x48 - 1 ) * sqwidth + pxoffset , ( 10 - y48 ) * sqheight + pyoffset } + p;
point q48 = { ( x48 * sqwidth ) + qxoffset , ( ( 11 - y48 ) * sqheight ) + qyoffset } + p;
point p50 = { ( x50 - 1 ) * sqwidth + pxoffset , ( 10 - y50 ) * sqheight + pyoffset } + p;
point q50 = { ( x50 * sqwidth ) + qxoffset , ( ( 11 - y50 ) * sqheight ) + qyoffset } + p;


point p51 = { ( x51 - 1 ) * sqwidth + pxoffset , ( 10 - y51 ) * sqheight + pyoffset } + p;
point q51 = { ( x51 * sqwidth ) + qxoffset , ( ( 11 - y51 ) * sqheight ) + qyoffset } + p;
point p53 = { ( x53 - 1 ) * sqwidth + pxoffset , ( 10 - y53 ) * sqheight + pyoffset } + p;
point q53 = { ( x53 * sqwidth ) + qxoffset , ( ( 11 - y53 ) * sqheight ) + qyoffset } + p;
point p55 = { ( x55 - 1 ) * sqwidth + pxoffset , ( 10 - y55 ) * sqheight + pyoffset } + p;
point q55 = { ( x55 * sqwidth ) + qxoffset , ( ( 11 - y55 ) * sqheight ) + qyoffset } + p;
point p57 = { ( x57 - 1 ) * sqwidth + pxoffset , ( 10 - y57 ) * sqheight + pyoffset } + p;
point q57 = { ( x57 * sqwidth ) + qxoffset , ( ( 11 - y57 ) * sqheight ) + qyoffset } + p;
point p59 = { ( x59 - 1 ) * sqwidth + pxoffset , ( 10 - y59 ) * sqheight + pyoffset } + p;
point q59 = { ( x59 * sqwidth ) + qxoffset , ( ( 11 - y59 ) * sqheight ) + qyoffset } + p;


point p62 = { ( x62 - 1 ) * sqwidth + pxoffset , ( 10 - y62 ) * sqheight + pyoffset } + p;
point q62 = { ( x62 * sqwidth ) + qxoffset , ( ( 11 - y62 ) * sqheight ) + qyoffset } + p;
point p64 = { ( x64 - 1 ) * sqwidth + pxoffset , ( 10 - y64 ) * sqheight + pyoffset } + p;
point q64 = { ( x64 * sqwidth ) + qxoffset , ( ( 11 - y64 ) * sqheight ) + qyoffset } + p;
point p66 = { ( x66 - 1 ) * sqwidth + pxoffset , ( 10 - y66 ) * sqheight + pyoffset } + p;
point q66 = { ( x66 * sqwidth ) + qxoffset , ( ( 11 - y66 ) * sqheight ) + qyoffset } + p;
point p68 = { ( x68 - 1 ) * sqwidth + pxoffset , ( 10 - y68 ) * sqheight + pyoffset } + p;
point q68 = { ( x68 * sqwidth ) + qxoffset , ( ( 11 - y68 ) * sqheight ) + qyoffset } + p;
point p70 = { ( x70 - 1 ) * sqwidth + pxoffset , ( 10 - y70 ) * sqheight + pyoffset } + p;
point q70 = { ( x70 * sqwidth ) + qxoffset , ( ( 11 - y70 ) * sqheight ) + qyoffset } + p;


point p71 = { ( x71 - 1 ) * sqwidth + pxoffset , ( 10 - y71 ) * sqheight + pyoffset } + p;
point q71 = { ( x71 * sqwidth ) + qxoffset , ( ( 11 - y71 ) * sqheight ) + qyoffset } + p;
point p73 = { ( x73 - 1 ) * sqwidth + pxoffset , ( 10 - y73 ) * sqheight + pyoffset } + p;
point q73 = { ( x73 * sqwidth ) + qxoffset , ( ( 11 - y73 ) * sqheight ) + qyoffset } + p;
point p75 = { ( x75 - 1 ) * sqwidth + pxoffset , ( 10 - y75 ) * sqheight + pyoffset } + p;
point q75 = { ( x75 * sqwidth ) + qxoffset , ( ( 11 - y75 ) * sqheight ) + qyoffset } + p;
point p77 = { ( x77 - 1 ) * sqwidth + pxoffset , ( 10 - y77 ) * sqheight + pyoffset } + p;
point q77 = { ( x77 * sqwidth ) + qxoffset , ( ( 11 - y77 ) * sqheight ) + qyoffset } + p;
point p79 = { ( x79 - 1 ) * sqwidth + pxoffset , ( 10 - y79 ) * sqheight + pyoffset } + p;
point q79 = { ( x79 * sqwidth ) + qxoffset , ( ( 11 - y79 ) * sqheight ) + qyoffset } + p;


point p82 = { ( x82 - 1 ) * sqwidth + pxoffset , ( 10 - y82 ) * sqheight + pyoffset } + p;
point q82 = { ( x82 * sqwidth ) + qxoffset , ( ( 11 - y82 ) * sqheight ) + qyoffset } + p;
point p84 = { ( x84 - 1 ) * sqwidth + pxoffset , ( 10 - y84 ) * sqheight + pyoffset } + p;
point q84 = { ( x84 * sqwidth ) + qxoffset , ( ( 11 - y84 ) * sqheight ) + qyoffset } + p;
point p86 = { ( x86 - 1 ) * sqwidth + pxoffset , ( 10 - y86 ) * sqheight + pyoffset } + p;
point q86 = { ( x86 * sqwidth ) + qxoffset , ( ( 11 - y86 ) * sqheight ) + qyoffset } + p;
point p88 = { ( x88 - 1 ) * sqwidth + pxoffset , ( 10 - y88 ) * sqheight + pyoffset } + p;
point q88 = { ( x88 * sqwidth ) + qxoffset , ( ( 11 - y88 ) * sqheight ) + qyoffset } + p;
point p90 = { ( x90 - 1 ) * sqwidth + pxoffset , ( 10 - y90 ) * sqheight + pyoffset } + p;
point q90 = { ( x90 * sqwidth ) + qxoffset , ( ( 11 - y90 ) * sqheight ) + qyoffset } + p;

point p91 = { ( x91 - 1 ) * sqwidth + pxoffset , ( 10 - y91 ) * sqheight + pyoffset } + p;
point q91 = { ( x91 * sqwidth ) + qxoffset , ( ( 11 - y91 ) * sqheight ) + qyoffset } + p;
point p93 = { ( x93 - 1 ) * sqwidth + pxoffset , ( 10 - y93 ) * sqheight + pyoffset } + p;
point q93 = { ( x93 * sqwidth ) + qxoffset , ( ( 11 - y93 ) * sqheight ) + qyoffset } + p;
point p95 = { ( x95 - 1 ) * sqwidth + pxoffset , ( 10 - y95 ) * sqheight + pyoffset } + p;
point q95 = { ( x95 * sqwidth ) + qxoffset , ( ( 11 - y95 ) * sqheight ) + qyoffset } + p;
point p97 = { ( x97 - 1 ) * sqwidth + pxoffset , ( 10 - y97 ) * sqheight + pyoffset } + p;
point q97 = { ( x97 * sqwidth ) + qxoffset , ( ( 11 - y97 ) * sqheight ) + qyoffset } + p;
point p99 = { ( x99 - 1 ) * sqwidth + pxoffset , ( 10 - y99 ) * sqheight + pyoffset } + p;
point q99 = { ( x99 * sqwidth ) + qxoffset , ( ( 11 - y99 ) * sqheight ) + qyoffset } + p;
```

```
point p102 = { ( x102 - 1 ) * sqwidth + pxoffset , ( 10 - y102 ) * sqheight + pyoffset } +
p;
point q102 = { ( x102 * sqwidth ) + qxoffset , ( ( 11 - y102 ) * sqheight ) + qyoffset } +
p;
point p104 = { ( x104 - 1 ) * sqwidth + pxoffset , ( 10 - y104 ) * sqheight + pyoffset } +
p;
point q104 = { ( x104 * sqwidth ) + qxoffset , ( ( 11 - y104 ) * sqheight ) + qyoffset } +
p;
point p106 = { ( x106 - 1 ) * sqwidth + pxoffset , ( 10 - y106 ) * sqheight + pyoffset } +
p;
point q106 = { ( x106 * sqwidth ) + qxoffset , ( ( 11 - y106 ) * sqheight ) + qyoffset } +
p;
point p108 = { ( x108 - 1 ) * sqwidth + pxoffset , ( 10 - y108 ) * sqheight + pyoffset } +
p;
point q108 = { ( x108 * sqwidth ) + qxoffset , ( ( 11 - y108 ) * sqheight ) + qyoffset } +
p;
point p110 = { ( x110 - 1 ) * sqwidth + pxoffset , ( 10 - y110 ) * sqheight + pyoffset } +
p;
point q110 = { ( x110 * sqwidth ) + qxoffset , ( ( 11 - y110 ) * sqheight ) + qyoffset } +
p;


# Set up a DoNaLD window to contain all the squares
window board = {
        type: DONALD
        box: [p,q]
        border: 1
};

bgcol = "blue";


# Set up the windows for the squares themselves
window square11 = {
        type: DONALD
        box: [p11,q11]
        pict: "piece11"
        bgcolor: bgcol
        sensitive: ON
};
window square13 = {
        type: DONALD
        box: [p13,q13]
        pict: "piece13"
        bgcolor: bgcol
        sensitive: ON
};
window square15 = {
        type: DONALD
        box: [p15,q15]
        pict: "piece15"
        bgcolor: bgcol
        sensitive: ON
};
window square17 = {
        type: DONALD
        box: [p17,q17]
        pict: "piece17"
        bgcolor: bgcol
        sensitive: ON
};
window square19 = {
        type: DONALD
        box: [p19,q19]
        pict: "piece19"
        bgcolor: bgcol
        sensitive: ON
};

window square22 = {
        type: DONALD
        box: [p22,q22]
        pict: "piece22"
        bgcolor: bgcol
        sensitive: ON
};
window square24 = {
        type: DONALD
        box: [p24,q24]
        pict: "piece24"
        bgcolor: bgcol
        sensitive: ON
```

```
};
window square26 = {
        type: DONALD
        box: [p26,q26]
        pict: "piece26"
        bgcolor: bgcol
        sensitive: ON
};
window square28 = {
        type: DONALD
        box: [p28,q28]
        pict: "piece28"
        bgcolor: bgcol
        sensitive: ON
};
window square30 = {
        type: DONALD
        box: [p30,q30]
        pict: "piece30"
        bgcolor: bgcol
        sensitive: ON
};

window square31 = {
        type: DONALD
        box: [p31,q31]
        pict: "piece31"
        bgcolor: bgcol
        sensitive: ON
};
window square33 = {
        type: DONALD
        box: [p33,q33]
        pict: "piece33"
        bgcolor: bgcol
        sensitive: ON
};
window square35 = {
        type: DONALD
        box: [p35,q35]
        pict: "piece35"
        bgcolor: bgcol
        sensitive: ON
};
window square37 = {
        type: DONALD
        box: [p37,q37]
        pict: "piece37"
        bgcolor: bgcol
        sensitive: ON
};
window square39 = {
        type: DONALD
        box: [p39,q39]
        pict: "piece39"
        bgcolor: bgcol
        sensitive: ON
};

window square42 = {
        type: DONALD
        box: [p42,q42]
        pict: "piece42"
        bgcolor: bgcol
        sensitive: ON
};
window square44 = {
        type: DONALD
        box: [p44,q44]
        pict: "piece44"
        bgcolor: bgcol
        sensitive: ON
};
window square46 = {
        type: DONALD
        box: [p46,q46]
        pict: "piece46"
        bgcolor: bgcol
        sensitive: ON
};
window square48 = {
        type: DONALD
```

```
        box: [p48,q48]
        pict: "piece48"
        bgcolor: bgcol
        sensitive: ON
};
window square50 = {
        type: DONALD
        box: [p50,q50]
        pict: "piece50"
        bgcolor: bgcol
        sensitive: ON
};

window square51 = {
        type: DONALD
        box: [p51,q51]
        pict: "piece51"
        bgcolor: bgcol
        sensitive: ON
};
window square53 = {
        type: DONALD
        box: [p53,q53]
        pict: "piece53"
        bgcolor: bgcol
        sensitive: ON
};
window square55 = {
        type: DONALD
        box: [p55,q55]
        pict: "piece55"
        bgcolor: bgcol
        sensitive: ON
};
window square57 = {
        type: DONALD
        box: [p57,q57]
        pict: "piece57"
        bgcolor: bgcol
        sensitive: ON
};
window square59 = {
        type: DONALD
        box: [p59,q59]
        pict: "piece59"
        bgcolor: bgcol
        sensitive: ON
};

window square62 = {
        type: DONALD
        box: [p62,q62]
        pict: "piece62"
        bgcolor: bgcol
        sensitive: ON
};
window square64 = {
        type: DONALD
        box: [p64,q64]
        pict: "piece64"
        bgcolor: bgcol
        sensitive: ON
};
window square66 = {
        type: DONALD
        box: [p66,q66]
        pict: "piece66"
        bgcolor: bgcol
        sensitive: ON
};
window square68 = {
        type: DONALD
        box: [p68,q68]
        pict: "piece68"
        bgcolor: bgcol
        sensitive: ON
};
window square70 = {
        type: DONALD
        box: [p70,q70]
        pict: "piece70"
        bgcolor: bgcol
```

```
        sensitive: ON
};

window square71 = {
        type: DONALD
        box: [p71,q71]
        pict: "piece71"
        bgcolor: bgcol
        sensitive: ON
};
window square73 = {
        type: DONALD
        box: [p73,q73]
        pict: "piece73"
        bgcolor: bgcol
        sensitive: ON
};
window square75 = {
        type: DONALD
        box: [p75,q75]
        pict: "piece75"
        bgcolor: bgcol
        sensitive: ON
};
window square77 = {
        type: DONALD
        box: [p77,q77]
        pict: "piece77"
        bgcolor: bgcol
        sensitive: ON
};
window square79 = {
        type: DONALD
        box: [p79,q79]
        pict: "piece79"
        bgcolor: bgcol
        sensitive: ON
};

window square82 = {
        type: DONALD
        box: [p82,q82]
        pict: "piece82"
        bgcolor: bgcol
        sensitive: ON
};
window square84 = {
        type: DONALD
        box: [p84,q84]
        pict: "piece84"
        bgcolor: bgcol
        sensitive: ON
};
window square86 = {
        type: DONALD
        box: [p86,q86]
        pict: "piece86"
        bgcolor: bgcol
        sensitive: ON
};
window square88 = {
        type: DONALD
        box: [p88,q88]
        pict: "piece88"
        bgcolor: bgcol
        sensitive: ON
};
window square90 = {
        type: DONALD
        box: [p90,q90]
        pict: "piece90"
        bgcolor: bgcol
        sensitive: ON
};

window square91 = {
        type: DONALD
        box: [p91,q91]
        pict: "piece91"
        bgcolor: bgcol
        sensitive: ON
};
```

```
window square93 = {
        type: DONALD
        box: [p93,q93]
        pict: "piece93"
        bgcolor: bgcol
        sensitive: ON
};
window square95 = {
        type: DONALD
        box: [p95,q95]
        pict: "piece95"
        bgcolor: bgcol
        sensitive: ON
};
window square97 = {
        type: DONALD
        box: [p97,q97]
        pict: "piece97"
        bgcolor: bgcol
        sensitive: ON
};
window square99 = {
        type: DONALD
        box: [p99,q99]
        pict: "piece99"
        bgcolor: bgcol
        sensitive: ON
};

window square102 = {
        type: DONALD
        box: [p102,q102]
        pict: "piece102"
        bgcolor: bgcol
        sensitive: ON
};
window square104 = {
        type: DONALD
        box: [p104,q104]
        pict: "piece104"
        bgcolor: bgcol
        sensitive: ON
};
window square106 = {
        type: DONALD
        box: [p106,q106]
        pict: "piece106"
        bgcolor: bgcol
        sensitive: ON
};
window square108 = {
        type: DONALD
        box: [p108,q108]
        pict: "piece108"
        bgcolor: bgcol
        sensitive: ON
};
window square110 = {
        type: DONALD
        box: [p110,q110]
        pict: "piece110"
        bgcolor: bgcol
        sensitive: ON
};


# Define the x- and y- coordinates of each individual square
# on the board
integer x11 = 1;
integer x13 = 1;
integer x15 = 1;
integer x17 = 1;
integer x19 = 1;

integer x22 = 2;
integer x24 = 2;
integer x26 = 2;
integer x28 = 2;
integer x30 = 2;

integer x31 = 3;
integer x33 = 3;
```

```
integer x35 = 3;
integer x37 = 3;
integer x39 = 3;

integer x42 = 4;
integer x44 = 4;
integer x46 = 4;
integer x48 = 4;
integer x50 = 4;

integer x51 = 5;
integer x53 = 5;
integer x55 = 5;
integer x57 = 5;
integer x59 = 5;

integer x62 = 6;
integer x64 = 6;
integer x66 = 6;
integer x68 = 6;
integer x70 = 6;

integer x71 = 7;
integer x73 = 7;
integer x75 = 7;
integer x77 = 7;
integer x79 = 7;

integer x82 = 8;
integer x84 = 8;
integer x86 = 8;
integer x88 = 8;
integer x90 = 8;

integer x91 = 9;
integer x93 = 9;
integer x95 = 9;
integer x97 = 9;
integer x99 = 9;

integer x102 = 10;
integer x104 = 10;
integer x106 = 10;
integer x108 = 10;
integer x110 = 10;

integer y11 = 1;
integer y13 = 3;
integer y15 = 5;
integer y17 = 7;
integer y19 = 9;

integer y22 = 2;
integer y24 = 4;
integer y26 = 6;
integer y28 = 8;
integer y30 = 10;

integer y31 = 1;
integer y33 = 3;
integer y35 = 5;
integer y37 = 7;
integer y39 = 9;

integer y42 = 2;
integer y44 = 4;
integer y46 = 6;
integer y48 = 8;
integer y50 = 10;

integer y51 = 1;
integer y53 = 3;
integer y55 = 5;
integer y57 = 7;
integer y59 = 9;

integer y62 = 2;
integer y64 = 4;
integer y66 = 6;
integer y68 = 8;
integer y70 = 10;
```

```
integer y71 = 1;
integer y73 = 3;
integer y75 = 5;
integer y77 = 7;
integer y79 = 9;

integer y82 = 2;
integer y84 = 4;
integer y86 = 6;
integer y88 = 8;
integer y90 = 10;

integer y91 = 1;
integer y93 = 3;
integer y95 = 5;
integer y97 = 7;
integer y99 = 9;

integer y102 = 2;
integer y104 = 4;
integer y106 = 6;
integer y108 = 8;
integer y110 = 10;


# Set up the additional windows to control the game
window Click;
window whoseturn;
window lastsquare;
window lastsquarecontents;
window clickcontext;
window gamestatus;
window startgame;
window resetgame;
window promptwindow;
window computertoplay;
point win;
point win = {500,10};
string whoseturnstring;
string whoseturnstring = "-";
string lastsquarestring;
string lastsquarestring = "-";
string lastsquarecontentsstring;
string lastsquarecontentsstring = "-";
string clickcontextstring;
string clickcontextstring = "Select FROM square";
string gamestatusstring;
string gamestatusstring = "Game in progress";
string promptstring;
string promptstring = "Go Ahead!";
string playstring;
string playstring = "Auto Play";

window Click = {
        type: TEXT
        frame: ([win, win + {20.c + 10, 1.r}])
        string: "INFORMATION"
        alignment: CENTRE
        };

window whoseturn = {
        type: TEXT
        frame: (shift(Click.frame.1, 0, 1.r + 10))
        string: whoseturnstring
        border:1
        alignment:CENTRE
        };

window lastsquare = {
        type: TEXT
        frame: (shift(whoseturn.frame.1, 0, 1.r + 10))
        string: lastsquarestring
        border:1
        alignment:CENTRE
        };

window lastsquarecontents = {
        type: TEXT
        frame: (shift(lastsquare.frame.1, 0, 1.r + 10))
        string: lastsquarecontentsstring
        border:1
        alignment:CENTRE
```

```
        };

window clickcontext = {
        type: TEXT
        frame: (shift(lastsquarecontents.frame.1, 0, 1.r + 10))
        string: clickcontextstring
        border:1
        alignment:CENTRE
        };

window gamestatus = {
        type: TEXT
        frame: (shift(clickcontext.frame.1, 0, 1.r + 10))
        string: gamestatusstring
        border:1
        alignment:CENTRE
        };

window promptwindow = {
        type: TEXT
        frame: (shift(gamestatus.frame.1, 0, 1.r + 10))
        string: promptstring
        border: 1
        alignment: CENTRE
        };

window blankline = {
        type: TEXT
        frame: (shift(promptwindow.frame.1, 0, 1.r + 10))
        string: ""
        alignment: CENTRE
        };

window ctrlpaneltext = {
        type: TEXT
        frame: (shift(blankline.frame.1, 0, 1.r + 10))
        string: "CONTROL PANEL"
        alignment: CENTRE
        };

window startgame = {
        type: TEXT
        frame: (shift(ctrlpaneltext.frame.1, 0, 1.r + 10))
        string: "Start/Reset Game"
        border: 1
        alignment: CENTRE
        sensitive: ON
        };

window computertoplay = {
        type: TEXT
        frame: (shift(startgame.frame.1, 0, 1.r + 10))
        string: playstring
        border: 1
        alignment: CENTRE
        sensitive: ON
        };

# Define what needs to be displayed on the screen
display screen;
display screen = <Click / whoseturn / lastsquare / lastsquarecontents / clickcontext /
gamestatus / blankline / ctrlpaneltext / startgame / promptwindow / computertoplay /
square11 / square13 / square15 / square17 / square19 /square22 / square24 / square26 /
square28 / square30 /square31 / square33 / square35 / square37 / square39 /square42 /
square44 / square46 / square48 / square50 /square51 / square53 / square55 / square57
/square59 / square62 / square64 / square66 / square68 /square70 / square71 / square73 /
square75 / square77 /square79 / square82 / square84 / square86 / square88 /square90
/square91 /square93 /square95 /square97 /square99 / square102 /square104 /square106
/square108 /square110 / board>;

%eden

/*
 *    The EDEN script
 *
 *    The underlying functionality of the model
 *    The procedures required to implement the modely
 */

/*
 *    Set up some initial contstants to make the code easier to
```

```
  *    read
  */
B = 1;
W = 0;
black = "black";
blank = "blank";
white = "white";
red = "red";

/*
  *    Set the computer colour to 0 (white)
  */
computer_color = 0; /* computer is using white piece to play */

/*
  *    Set the square background colour to blue
  */
bgcolor = "blue";

/*
  *    Set up definitions for how to fill in each piece;
  *    each is filled in solidly, and takes the colour of colsqr??
  *    which is defined later as the colour of the piece in a
  *    particular square
  */
A_Piece11 is "fill=solid,color=" // colsqr11;
A_Piece13 is "fill=solid,color=" // colsqr13;
A_Piece15 is "fill=solid,color=" // colsqr15;
A_Piece17 is "fill=solid,color=" // colsqr17;
A_Piece19 is "fill=solid,color=" // colsqr19;

A_Piece22 is "fill=solid,color=" // colsqr22;
A_Piece24 is "fill=solid,color=" // colsqr24;
A_Piece26 is "fill=solid,color=" // colsqr26;
A_Piece28 is "fill=solid,color=" // colsqr28;
A_Piece30 is "fill=solid,color=" // colsqr30;

A_Piece31 is "fill=solid,color=" // colsqr31;
A_Piece33 is "fill=solid,color=" // colsqr33;
A_Piece35 is "fill=solid,color=" // colsqr35;
A_Piece37 is "fill=solid,color=" // colsqr37;
A_Piece39 is "fill=solid,color=" // colsqr39;

A_Piece42 is "fill=solid,color=" // colsqr42;
A_Piece44 is "fill=solid,color=" // colsqr44;
A_Piece46 is "fill=solid,color=" // colsqr46;
A_Piece48 is "fill=solid,color=" // colsqr48;
A_Piece50 is "fill=solid,color=" // colsqr50;

A_Piece51 is "fill=solid,color=" // colsqr51;
A_Piece53 is "fill=solid,color=" // colsqr53;
A_Piece55 is "fill=solid,color=" // colsqr55;
A_Piece57 is "fill=solid,color=" // colsqr57;
A_Piece59 is "fill=solid,color=" // colsqr59;

A_Piece62 is "fill=solid,color=" // colsqr62;
A_Piece64 is "fill=solid,color=" // colsqr64;
A_Piece66 is "fill=solid,color=" // colsqr66;
A_Piece68 is "fill=solid,color=" // colsqr68;
A_Piece70 is "fill=solid,color=" // colsqr70;

A_Piece71 is "fill=solid,color=" // colsqr71;
A_Piece73 is "fill=solid,color=" // colsqr73;
A_Piece75 is "fill=solid,color=" // colsqr75;
A_Piece77 is "fill=solid,color=" // colsqr77;
A_Piece79 is "fill=solid,color=" // colsqr79;

A_Piece82 is "fill=solid,color=" // colsqr82;
A_Piece84 is "fill=solid,color=" // colsqr84;
A_Piece86 is "fill=solid,color=" // colsqr86;
A_Piece88 is "fill=solid,color=" // colsqr88;
A_Piece90 is "fill=solid,color=" // colsqr90;

A_Piece91 is "fill=solid,color=" // colsqr91;
A_Piece93 is "fill=solid,color=" // colsqr93;
A_Piece95 is "fill=solid,color=" // colsqr95;
A_Piece97 is "fill=solid,color=" // colsqr97;
A_Piece99 is "fill=solid,color=" // colsqr99;

A_Piece102 is "fill=solid,color=" // colsqr102;
A_Piece104 is "fill=solid,color=" // colsqr104;
A_Piece106 is "fill=solid,color=" // colsqr106;
```

```
A_Piece108 is "fill=solid,color=" // colsqr108;
A_Piece110 is "fill=solid,color=" // colsqr110;


/*
  *    Set up definitions for how to draw the dot on top
  *    of each piece
  */
A_KingDot11 is "fill=solid,color=" // kingcol11;
A_KingDot13 is "fill=solid,color=" // kingcol13;
A_KingDot15 is "fill=solid,color=" // kingcol15;
A_KingDot17 is "fill=solid,color=" // kingcol17;
A_KingDot19 is "fill=solid,color=" // kingcol19;

A_KingDot22 is "fill=solid,color=" // kingcol22;
A_KingDot24 is "fill=solid,color=" // kingcol24;
A_KingDot26 is "fill=solid,color=" // kingcol26;
A_KingDot28 is "fill=solid,color=" // kingcol28;
A_KingDot30 is "fill=solid,color=" // kingcol30;

A_KingDot31 is "fill=solid,color=" // kingcol31;
A_KingDot33 is "fill=solid,color=" // kingcol33;
A_KingDot35 is "fill=solid,color=" // kingcol35;
A_KingDot37 is "fill=solid,color=" // kingcol37;
A_KingDot39 is "fill=solid,color=" // kingcol39;

A_KingDot42 is "fill=solid,color=" // kingcol42;
A_KingDot44 is "fill=solid,color=" // kingcol44;
A_KingDot46 is "fill=solid,color=" // kingcol46;
A_KingDot48 is "fill=solid,color=" // kingcol48;
A_KingDot50 is "fill=solid,color=" // kingcol50;

A_KingDot51 is "fill=solid,color=" // kingcol51;
A_KingDot53 is "fill=solid,color=" // kingcol53;
A_KingDot55 is "fill=solid,color=" // kingcol55;
A_KingDot57 is "fill=solid,color=" // kingcol57;
A_KingDot59 is "fill=solid,color=" // kingcol59;

A_KingDot62 is "fill=solid,color=" // kingcol62;
A_KingDot64 is "fill=solid,color=" // kingcol64;
A_KingDot66 is "fill=solid,color=" // kingcol66;
A_KingDot68 is "fill=solid,color=" // kingcol68;
A_KingDot70 is "fill=solid,color=" // kingcol70;

A_KingDot71 is "fill=solid,color=" // kingcol71;
A_KingDot73 is "fill=solid,color=" // kingcol73;
A_KingDot75 is "fill=solid,color=" // kingcol75;
A_KingDot77 is "fill=solid,color=" // kingcol77;
A_KingDot79 is "fill=solid,color=" // kingcol79;

A_KingDot82 is "fill=solid,color=" // kingcol82;
A_KingDot84 is "fill=solid,color=" // kingcol84;
A_KingDot86 is "fill=solid,color=" // kingcol86;
A_KingDot88 is "fill=solid,color=" // kingcol88;
A_KingDot90 is "fill=solid,color=" // kingcol90;

A_KingDot91 is "fill=solid,color=" // kingcol91;
A_KingDot93 is "fill=solid,color=" // kingcol93;
A_KingDot95 is "fill=solid,color=" // kingcol95;
A_KingDot97 is "fill=solid,color=" // kingcol97;
A_KingDot99 is "fill=solid,color=" // kingcol99;

A_KingDot102 is "fill=solid,color=" // kingcol102;
A_KingDot104 is "fill=solid,color=" // kingcol104;
A_KingDot106 is "fill=solid,color=" // kingcol106;
A_KingDot108 is "fill=solid,color=" // kingcol108;
A_KingDot110 is "fill=solid,color=" // kingcol110;


/*
  *    ???
  *    To check what colour piece is on the square
  */
Square11 is checkcol([1,1], bpieces, wpieces);
Square13 is checkcol([1,3], bpieces, wpieces);
Square15 is checkcol([1,5], bpieces, wpieces);
Square17 is checkcol([1,7], bpieces, wpieces);
Square19 is checkcol([1,9], bpieces, wpieces);

Square22 is checkcol([2,2], bpieces, wpieces);
Square24 is checkcol([2,4], bpieces, wpieces);
Square26 is checkcol([2,6], bpieces, wpieces);
```

```
Square28 is checkcol([2,8], bpieces, wpieces);
Square30 is checkcol([2,10], bpieces, wpieces);

Square31 is checkcol([3,1], bpieces, wpieces);
Square33 is checkcol([3,3], bpieces, wpieces);
Square35 is checkcol([3,5], bpieces, wpieces);
Square37 is checkcol([3,7], bpieces, wpieces);
Square39 is checkcol([3,9], bpieces, wpieces);

Square42 is checkcol([4,2], bpieces, wpieces);
Square44 is checkcol([4,4], bpieces, wpieces);
Square46 is checkcol([4,6], bpieces, wpieces);
Square48 is checkcol([4,8], bpieces, wpieces);
Square50 is checkcol([4,10], bpieces, wpieces);

Square51 is checkcol([5,1], bpieces, wpieces);
Square53 is checkcol([5,3], bpieces, wpieces);
Square55 is checkcol([5,5], bpieces, wpieces);
Square57 is checkcol([5,7], bpieces, wpieces);
Square59 is checkcol([5,9], bpieces, wpieces);

Square62 is checkcol([6,2], bpieces, wpieces);
Square64 is checkcol([6,4], bpieces, wpieces);
Square66 is checkcol([6,6], bpieces, wpieces);
Square68 is checkcol([6,8], bpieces, wpieces);
Square70 is checkcol([6,10], bpieces, wpieces);

Square71 is checkcol([7,1], bpieces, wpieces);
Square73 is checkcol([7,3], bpieces, wpieces);
Square75 is checkcol([7,5], bpieces, wpieces);
Square77 is checkcol([7,7], bpieces, wpieces);
Square79 is checkcol([7,9], bpieces, wpieces);

Square82 is checkcol([8,2], bpieces, wpieces);
Square84 is checkcol([8,4], bpieces, wpieces);
Square86 is checkcol([8,6], bpieces, wpieces);
Square88 is checkcol([8,8], bpieces, wpieces);
Square90 is checkcol([8,10], bpieces, wpieces);

Square91 is checkcol([9,1], bpieces, wpieces);
Square93 is checkcol([9,3], bpieces, wpieces);
Square95 is checkcol([9,5], bpieces, wpieces);
Square97 is checkcol([9,7], bpieces, wpieces);
Square99 is checkcol([9,9], bpieces, wpieces);

Square102 is checkcol([10,2], bpieces, wpieces);
Square104 is checkcol([10,4], bpieces, wpieces);
Square106 is checkcol([10,6], bpieces, wpieces);
Square108 is checkcol([10,8], bpieces, wpieces);
Square110 is checkcol([10,10], bpieces, wpieces);


/*
 *  Define the set of white pieces and the set of black pieces
 *  in play
 */
wpieces                                                                    is
[w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17,w18,w19,w20];
bpieces is [b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19,b20];

/*
 *  Defines the colour of a particular piece in a square
 *  for the purposes of drawing.
 *  Possible values are white, black or the background colour.
 */
colsqr11 is (Square11 == white)? white:((Square11 == black)? black: bgcolor);
colsqr13 is (Square13 == white)? white:((Square13 == black)? black: bgcolor);
colsqr15 is (Square15 == white)? white:((Square15 == black)? black: bgcolor);
colsqr17 is (Square17 == white)? white:((Square17 == black)? black: bgcolor);
colsqr19 is (Square19 == white)? white:((Square19 == black)? black: bgcolor);

colsqr22 is (Square22 == white)? white:((Square22 == black)? black: bgcolor);
colsqr24 is (Square24 == white)? white:((Square24 == black)? black: bgcolor);
colsqr26 is (Square26 == white)? white:((Square26 == black)? black: bgcolor);
colsqr28 is (Square28 == white)? white:((Square28 == black)? black: bgcolor);
colsqr30 is (Square30 == white)? white:((Square30 == black)? black: bgcolor);

colsqr31 is (Square31 == white)? white:((Square31 == black)? black: bgcolor);
colsqr33 is (Square33 == white)? white:((Square33 == black)? black: bgcolor);
colsqr35 is (Square35 == white)? white:((Square35 == black)? black: bgcolor);
colsqr37 is (Square37 == white)? white:((Square37 == black)? black: bgcolor);
colsqr39 is (Square39 == white)? white:((Square39 == black)? black: bgcolor);
```

```
colsqr42 is (Square42 == white)? white:((Square42 == black)? black: bgcolor);
colsqr44 is (Square44 == white)? white:((Square44 == black)? black: bgcolor);
colsqr46 is (Square46 == white)? white:((Square46 == black)? black: bgcolor);
colsqr48 is (Square48 == white)? white:((Square48 == black)? black: bgcolor);
colsqr50 is (Square50 == white)? white:((Square50 == black)? black: bgcolor);

colsqr51 is (Square51 == white)? white:((Square51 == black)? black: bgcolor);
colsqr53 is (Square53 == white)? white:((Square53 == black)? black: bgcolor);
colsqr55 is (Square55 == white)? white:((Square55 == black)? black: bgcolor);
colsqr57 is (Square57 == white)? white:((Square57 == black)? black: bgcolor);
colsqr59 is (Square59 == white)? white:((Square59 == black)? black: bgcolor);

colsqr62 is (Square62 == white)? white:((Square62 == black)? black: bgcolor);
colsqr64 is (Square64 == white)? white:((Square64 == black)? black: bgcolor);
colsqr66 is (Square66 == white)? white:((Square66 == black)? black: bgcolor);
colsqr68 is (Square68 == white)? white:((Square68 == black)? black: bgcolor);
colsqr70 is (Square70 == white)? white:((Square70 == black)? black: bgcolor);

colsqr71 is (Square71 == white)? white:((Square71 == black)? black: bgcolor);
colsqr73 is (Square73 == white)? white:((Square73 == black)? black: bgcolor);
colsqr75 is (Square75 == white)? white:((Square75 == black)? black: bgcolor);
colsqr77 is (Square77 == white)? white:((Square77 == black)? black: bgcolor);
colsqr79 is (Square79 == white)? white:((Square79 == black)? black: bgcolor);

colsqr82 is (Square82 == white)? white:((Square82 == black)? black: bgcolor);
colsqr84 is (Square84 == white)? white:((Square84 == black)? black: bgcolor);
colsqr86 is (Square86 == white)? white:((Square86 == black)? black: bgcolor);
colsqr88 is (Square88 == white)? white:((Square88 == black)? black: bgcolor);
colsqr90 is (Square90 == white)? white:((Square90 == black)? black: bgcolor);

colsqr91 is (Square91 == white)? white:((Square91 == black)? black: bgcolor);
colsqr93 is (Square93 == white)? white:((Square93 == black)? black: bgcolor);
colsqr95 is (Square95 == white)? white:((Square95 == black)? black: bgcolor);
colsqr97 is (Square97 == white)? white:((Square97 == black)? black: bgcolor);
colsqr99 is (Square99 == white)? white:((Square99 == black)? black: bgcolor);

colsqr102 is (Square102 == white)? white:((Square102 == black)? black: bgcolor);
colsqr104 is (Square104 == white)? white:((Square104 == black)? black: bgcolor);
colsqr106 is (Square106 == white)? white:((Square106 == black)? black: bgcolor);
colsqr108 is (Square108 == white)? white:((Square108 == black)? black: bgcolor);
colsqr110 is (Square110 == white)? white:((Square110 == black)? black: bgcolor);


/*
  *   Define the colour scheme for a king piece (ie, red if it is a
  *   king, and the piece colour if it isn't
  */
kingcol11 is (iscrowned([1,1]) == 1)? red: colsqr11;
kingcol13 is (iscrowned([1,3]) == 1)? red: colsqr13;
kingcol15 is (iscrowned([1,5]) == 1)? red: colsqr15;
kingcol17 is (iscrowned([1,7]) == 1)? red: colsqr17;
kingcol19 is (iscrowned([1,9]) == 1)? red: colsqr19;

kingcol22 is (iscrowned([2,2]) == 1)? red: colsqr22;
kingcol24 is (iscrowned([2,4]) == 1)? red: colsqr24;
kingcol26 is (iscrowned([2,6]) == 1)? red: colsqr26;
kingcol28 is (iscrowned([2,8]) == 1)? red: colsqr28;
kingcol30 is (iscrowned([2,10]) == 1)? red: colsqr30;

kingcol31 is (iscrowned([3,1]) == 1)? red: colsqr31;
kingcol33 is (iscrowned([3,3]) == 1)? red: colsqr33;
kingcol35 is (iscrowned([3,5]) == 1)? red: colsqr35;
kingcol37 is (iscrowned([3,7]) == 1)? red: colsqr37;
kingcol39 is (iscrowned([3,9]) == 1)? red: colsqr39;

kingcol42 is (iscrowned([4,2]) == 1)? red: colsqr42;
kingcol44 is (iscrowned([4,4]) == 1)? red: colsqr44;
kingcol46 is (iscrowned([4,6]) == 1)? red: colsqr46;
kingcol48 is (iscrowned([4,8]) == 1)? red: colsqr48;
kingcol50 is (iscrowned([4,10]) == 1)? red: colsqr50;

kingcol51 is (iscrowned([5,1]) == 1)? red: colsqr51;
kingcol53 is (iscrowned([5,3]) == 1)? red: colsqr53;
kingcol55 is (iscrowned([5,5]) == 1)? red: colsqr55;
kingcol57 is (iscrowned([5,7]) == 1)? red: colsqr57;
kingcol59 is (iscrowned([5,9]) == 1)? red: colsqr59;

kingcol62 is (iscrowned([6,2]) == 1)? red: colsqr62;
kingcol64 is (iscrowned([6,4]) == 1)? red: colsqr64;
kingcol66 is (iscrowned([6,6]) == 1)? red: colsqr66;
kingcol68 is (iscrowned([6,8]) == 1)? red: colsqr68;
```

```
kingcol70 is (iscrowned([6,10]) == 1)? red: colsqr70;

kingcol71 is (iscrowned([7,1]) == 1)? red: colsqr71;
kingcol73 is (iscrowned([7,3]) == 1)? red: colsqr73;
kingcol75 is (iscrowned([7,5]) == 1)? red: colsqr75;
kingcol77 is (iscrowned([7,7]) == 1)? red: colsqr77;
kingcol79 is (iscrowned([7,9]) == 1)? red: colsqr79;

kingcol82 is (iscrowned([8,2]) == 1)? red: colsqr82;
kingcol84 is (iscrowned([8,4]) == 1)? red: colsqr84;
kingcol86 is (iscrowned([8,6]) == 1)? red: colsqr86;
kingcol88 is (iscrowned([8,8]) == 1)? red: colsqr88;
kingcol90 is (iscrowned([8,10]) == 1)? red: colsqr90;

kingcol91 is (iscrowned([9,1]) == 1)? red: colsqr91;
kingcol93 is (iscrowned([9,3]) == 1)? red: colsqr93;
kingcol95 is (iscrowned([9,5]) == 1)? red: colsqr95;
kingcol97 is (iscrowned([9,7]) == 1)? red: colsqr97;
kingcol99 is (iscrowned([9,9]) == 1)? red: colsqr99;

kingcol102 is (iscrowned([10,2]) == 1)? red: colsqr102;
kingcol104 is (iscrowned([10,4]) == 1)? red: colsqr104;
kingcol106 is (iscrowned([10,6]) == 1)? red: colsqr106;
kingcol108 is (iscrowned([10,8]) == 1)? red: colsqr108;
kingcol110 is (iscrowned([10,10]) == 1)? red: colsqr110;



/*
  *   Define a string variable to "b" or "w" depending on whether black
  *   is playing or not, appropriate to the current player
  *   Also define the set of currently available pieces to the current
  *   player.
  */
ctp is (toplay == B)? "b": "w";
curr_pieces is (toplay == B)? bpieces: wpieces;



/*
  *   Define the strings which will appear in the control panel of the
  *   screen.
  */
whoseturntext is (toplay == B)? "Black to play": "White to play";
proc updatewhs : whoseturntext {
        whoseturnstring = whoseturntext;
        }
clickcontexttext is (status == 1)? "Select FROM square": (status == 2 )? "Select TO
square": "No game";
proc updateccs : status {
        clickcontextstring = clickcontexttext;
        }
lastsquaretext is "(" // str(x) // "," // str(y) // ")";
proc updatelst : lastsquaretext {
        lastsquarestring = lastsquaretext;
        }
lastsquarecontentstext is (checkcol([x,y]) == black)? "Black piece": (checkcol([x,y]) ==
white)? "White piece": "No piece";
proc updatelsct : lastsquarecontentstext {
        lastsquarecontentsstring = lastsquarecontentstext;
        }
game_status is ((toplay == 0) && game_end) ? "Black is the winner": ((toplay == 1)
&& game_end ) ? "White is the winner": "Game in progress";
gamestatustext is game_status;
proc updategs : gamestatustext {
        gamestatusstring = gamestatustext;
        }

prompt is (cancapture) ? "Capture!" : "Go Ahead!";
prompttext is prompt;
proc updateprompt : prompttext{
        promptstring = prompttext;
}

play_status is (computer_to_play == 0) ? "Auto Play" : "Two People Play";
playstatustext is play_status;
proc updateps : playstatustext{
        playstring = playstatustext;
}
/*
  *   Here we define the rules of the game.
  *   Defined as a set of possible situations in which moves are valid
  */
```

```
poss is [possnmm, possnmp, possnpm, possnpp, possn2mm, possn2mp, possn2pm,
possn2pp, posskmm, posskmp, posskpm, posskpp, possk2mm, possk2mp, possk2pm,
possk2pp];

possn is [possnmm, possnmp, possnpm, possnpp, possn2mm, possn2mp, possn2pm,
possn2pp];
possn2mm is ( ( ( toplay == B ) && ( playerowns ) ) && ((checkcol([x-2,y-2]) == blank)
&& (onboard([x-2,y-2])) && ((toplay == B && checkcol([x-1,y-1]) == white ) || (toplay
== W && checkcol([x-1,y-1]) == black ))));
possn2mp is ( ( ( toplay == W ) && ( playerowns ) ) && ((checkcol([x-2,y+2]) == blank)
&& (onboard([x-2,y+2])) && ((toplay == B && checkcol([x-1,y+1]) == white ) ||
(toplay == W && checkcol([x-1,y+1]) == black ))));
possn2pm is ( ( ( toplay == B ) && ( playerowns ) ) && ((checkcol([x+2,y-2]) == blank)
&& (onboard([x+2,y-2])) && ((toplay == B && checkcol([x+1,y-1]) == white ) ||
(toplay == W && checkcol([x+1,y-1]) == black ))));
possn2pp is ( ( ( toplay == W ) && ( playerowns ) ) && ((checkcol([x+2,y+2]) == blank)
&& (onboard([x+2,y+2])) && ((toplay == B && checkcol([x+1,y+1]) == white ) ||
(toplay == W && checkcol([x+1,y+1]) == black ))));
possnmm is ( ( ( toplay == B ) && ( playerowns ) ) && ((checkcol([x-1,y-1]) == blank)
&& (onboard ([x-1,y-1]))));
possnmp is ( ( ( toplay == W ) && ( playerowns ) ) && ((checkcol([x-1,y+1]) == blank)
&& (onboard ([x-1,y+1]))));
possnpm is (( ( toplay == B ) && ( playerowns ) ) && ((checkcol([x+1,y-1]) == blank)
&& (onboard ([x+1,y-1]))));
possnpp is ( ( ( toplay == W ) && ( playerowns ) ) && ((checkcol([x+1,y+1]) == blank)
&& (onboard ([x+1,y+1]))));

possk is [posskmm, posskmp, posskpm, posskpp, possk2mm, possk2mp, possk2pm,
possk2pp];
possk2mm is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x-2,y-2]) ==
blank) && (onboard([x-2,y-2])) && ((toplay == B && checkcol([x-1,y-1]) == white ) ||
(toplay == W && checkcol([x-1,y-1]) == black ))));
possk2mp is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x-2,y+2]) ==
blank) && (onboard([x-2,y+2])) && ((toplay == B && checkcol([x-1,y+1]) == white )
|| (toplay == W && checkcol([x-1,y+1]) == black ))));
possk2pm is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x+2,y-2]) ==
blank) && (onboard([x+2,y-2])) && ((toplay == B && checkcol([x+1,y-1]) == white )
|| (toplay == W && checkcol([x+1,y-1]) == black ))));
possk2pp is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x+2,y+2]) ==
blank) && (onboard([x+2,y+2])) && ((toplay == B && checkcol([x+1,y+1]) == white )
|| (toplay == W && checkcol([x+1,y+1]) == black ))));

posskmm is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x-1,y-1]) == blank)
&& (onboard ([x-1,y-1]))));
posskmp is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x-1,y+1]) == blank)
&& (onboard ([x-1,y+1]))));
posskpm is ( ( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x+1,y-1]) == blank)
&& (onboard ([x+1,y-1]))));
posskpp is (( pieceisking ) && ( ( ( playerowns ) ) && ((checkcol([x+1,y+1]) == blank)
&& (onboard ([x+1,y+1]))));

playerowns is ((( toplay == B ) && ( checkcol([x,y]) == black )) || (( toplay == W ) &&
( checkcol([x,y]) == white )));
pieceisking is ( iscrowned([x,y]) );


/*
 *   Shorthard for the x- and y-coordinates for the current
 *   piece
 */
x is current_piece[1];
y is current_piece[2];


/*
 *   FUNCTION:       checkcol
 *
 *   Gets the colour of a particular square on the board
 *
 *   ARGUMENTS           coord
 *   RETURN VALUE        blank, black or white
 */
func checkcol {
        para coord;
        auto ans,i;
        ans = blank;

        for (i=1;i<=20;i++){
        if (`"b"//str(i)` == coord)
                ans = black;
                }
        for (i=1;i<=20;i++){
        if(`"w"//str(i)` == coord)
```

```
                ans = white;
                }
        return ans;
        }

/*
 *   FUNCTION:       idsq
 *
 *   Gives the piece number at any given coordinate
 *
 *   ARGUMENTS           coord
 *   RETURN VALUE        1 <= i <= 20
 */
func idsq {
        para coord;
        auto ans, i;
        ans = 0;

        for (i=1; i<=20; i++){
        if (`"b"//str(i)` == coord)
        ans = i;
                }
        for (i=1;i<=20;i++){
        if(`"w"//str(i)` == coord)
        ans = i;
                }
        return ans;
        }

/*
 *   FUNCTION:       onboard
 *
 *   Determines if a given co-ordinate pair represents a square
 *   on the board
 *
 *   ARGUMENTS           coord
 *   RETURN VALUE        true or false
 */
func onboard {
        para coord;
        return (coord[1]>=1)&&(coord[1]<=10)&&(coord[2]>=1)&&(coord[2]<=10);
}

/*
 *   FUNCTION        piecestomove
 *
 *   Makes a list of possible pieces to move
 *
 *   ARGUMENTS           none
 *   RETURN VALUE        list of co-ordinates of possible pieces
 *                       to move
 */
func piecestomove {
        auto result, i, j;
        result=[];
        for (i=1;i<=20;i++) {
                current_piece = curr_pieces[i];
                xx = 0;                                 /* ???? */
                for (j=1;j<=16;j++) {
                        if (poss[j] != 0)
                        result = result // [[i,j]];
                }
        }
        return result;
}


/*
 *   The following defines moveok1, which is true when a non-capture
 *   move is possible.
 */
moveok1 is ( ( ( toplay == W ) && ( moveok1w ) ) ||
                ( ( toplay == B ) && ( moveok1b ) ) ||
                ( ( iscrowned([xsel,ysel]) ) && ( moveok1k ) ) );

moveok1k is ((xdest-xsel == -1 && ydest-ysel == -1 ) && (posskmm) ||
        (xdest-xsel == -1 && ydest-ysel == 1 ) && (posskmp) ||
        (xdest-xsel == 1 && ydest-ysel == -1 ) && (posskpm) ||
        (xdest-xsel == 1 && ydest-ysel == 1 ) && (posskpp));

moveok1w is ((xdest-xsel == -1 && ydest-ysel == 1 ) && (possnmp) ||
        (xdest-xsel == 1 && ydest-ysel == 1 ) && (possnpp));
```

```
moveok1b is ((xdest-xsel == -1 && ydest-ysel == -1 ) && (possnmm) ||
        (xdest-xsel == 1 && ydest-ysel == -1 ) && (possnpm));


/*
  *    The following defines moveok2, which is true when a capture move
  *    is possible.
  */

moveok2 is ( ( ( toplay == W ) && ( moveok2w ) ) ||
                ( ( toplay == B ) && ( moveok2b ) ) ||
                ( ( iscrowned([xsel,ysel]) ) && ( moveok2k ) ) );

moveok2k is ((xdest-xsel == -2 && ydest-ysel == -2 ) && (possk2mm) ||
        (xdest-xsel == -2 && ydest-ysel == 2 ) && (possk2mp) ||
        (xdest-xsel == 2 && ydest-ysel == -2 ) && (possk2pm) ||
        (xdest-xsel == 2 && ydest-ysel == 2 ) && (possk2pp));

moveok2w is ((xdest-xsel == -2 && ydest-ysel == 2 ) && (possn2mp) ||
        (xdest-xsel == 2 && ydest-ysel == 2 ) && (possn2pp));

moveok2b is ((xdest-xsel == -2 && ydest-ysel == -2 ) && (possn2mm) ||
        (xdest-xsel == 2 && ydest-ysel == -2 ) && (possn2pm));

/*
  *    Define the end of the game as when there are no pieces left
  *    on the board to move
  */


game_end is (pcm# == 0);
nopcm is pcm#;                                  /* ???? */
pcm is piecestomove(toplay);

/*
  *    PROCEDURE     move
  *
  *    Moves the piece to its new co-ordinates
  *    Switches the player
  *
  *    PARAMETERS          piece id, new x coord, new y coord
  *    RETURNED VALUES none
  */
proc move {
        para i, xd, yd;
        pieceid = `ctp//str(i)`;
        `ctp//str(i)` = [xd,yd];
        if ( ( toplay == W ) && ( yd == 10 ) ) { wcrowned[i] = 1; };
        if ( ( toplay == B ) && ( yd == 1 ) ) { bcrowned[i] = 1; };
        if (onboard([xd,yd])){
                toplay = 1 - toplay;
        }
}

/*
  *    PROCEDURE    startgameaction
  *
  *    sets up the board and player information ready for a game
  *    to begin.
  *
  */
proc stgame : startgame_mouse_1 {
/*
  *    Set the initial co-ordinates of the black (prefixed with `b')
  *    and white (prefixed with `w') pieces
  */
        w1 = [1,1];
        w2 = [3,1];
        w3 = [5,1];
        w4 = [7,1];
        w5 = [9,1];

        w6 = [2,2];
        w7 = [4,2];
        w8 = [6,2];
        w9 = [8,2];
        w10 = [10,2];

        w11 = [1,3];
        w12 = [3,3];
        w13 = [5,3];
```

```
        w14 = [7,3];
        w15 = [9,3];

        w16 = [2,4];
        w17 = [4,4];
        w18 = [6,4];
        w19 = [8,4];
        w20 = [10,4];

        b1 = [1,7];
        b2 = [3,7];
        b3 = [5,7];
        b4 = [7,7];
        b5 = [9,7];

        b6 = [2,8];
        b7 = [4,8];
        b8 = [6,8];
        b9 = [8,8];
        b10 = [10,8];

        b11 = [1,9];
        b12 = [3,9];
        b13 = [5,9];
        b14 = [7,9];
        b15 = [9,9];

        b16 = [2,10];
        b17 = [4,10];
        b18 = [6,10];
        b19 = [8,10];
        b20 = [10,10];

        toplay = B;
        wcrowned = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
        bcrowned = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
        status = 1;
        }


/*
 *    FUNCTION iscrowned
 *
 *    given a set of coordinates, returns if a piece is crowned or not
 *
 */
func iscrowned {
        para coord;
        auto ans, pid;

        ans = 0;
        pid = idsq(coord);
        if ( checkcol(coord) == white ) { ans = wcrowned[pid]; }
        if ( checkcol(coord) == black ) { ans = bcrowned[pid]; }
        return ans;
        }


/*
 *    PROCEDURES   square??_move_dest
 *
 *    This set of procedures set the current piece to the square
 *    the mouse is clicked in, when the system detects a mouse
 *    click.
 *
 *    status          defines the context in which the mouse
 *                    click is recieved, ie FROM or TO click?
 *
 *    DEPENDENCIES        square??_mouse  (ie mouse click)
 *    PARAMETERS          none
 *    RETURNED VALUES     none
 *
 *    the
 *       move(idsq(mcood), 1000, 1000);
 *    should be sorted out.
 */
proc square11_move_dest: square11_mouse {
if (square11_mouse[2] == 4 && (computer_to_play == 0 || (computer_to_play == 1 &&
toplay == B) ) ) {
        if (status == 1) {
                xsel = 1;
                ysel = 1;
```

```
                current_piece = [xsel,ysel];
                status = 2;
        }
        else { xdest = 1;
                ydest = 1;
        if (moveok1) move (idsq(current_piece), xdest, ydest);
                if (moveok2) {
                        move(idsq(current_piece), xdest, ydest);
                        xmid = (xsel+xdest) / 2;
                        ymid = (ysel+ydest) / 2;
                        mcood = [xmid,ymid];
                        move(idsq(mcood), 1000, 1000);
                        }
                        status = 1;
                }
        }
}
…
proc square110_move_dest: square110_mouse {
if (square110_mouse[2] == 4 && (computer_to_play == 0 || (computer_to_play == 1
&& toplay == B) )) {
        if (status == 1) {
                xsel = 10;
                ysel = 10;
                current_piece = [xsel,ysel];
                status = 2;
        }
        else { xdest = 10;
                ydest = 10;
        if (moveok1) move (idsq(current_piece), xdest, ydest);
                if (moveok2) {
                        move(idsq(current_piece), xdest, ydest);
                        xmid = (xsel+xdest) / 2;
                        ymid = (ysel+ydest) / 2;
                        mcood = [xmid,ymid];
                        move(idsq(mcood), 1000, 1000);
                        }
                        status = 1;
                }
        }
}
```

```
cancapture is (possn2mm || possn2mp || possn2pm || possn2pp || possk2mm || possk2mp ||
possk2pm || possk2pp);

canmove is (possnmm || possnmp || possnpm || possnpp ||  posskmm || posskmp ||
posskpm || posskpp );

/* brings cyclic dependency ==== npcm---->cancaputure--->possn2mm,...

---->canmove------->possnmm,...--->havetocapture--->cancapture       */

computer_to_play = 0;

proc computerplay : computertoplay_mouse_1 {

if(computertoplay_mouse_1[2] == 4)
computer_to_play = !computer_to_play;

}


/*
proc autoplay : computer_to_play, toplay{

auto i;
if(computer_to_play == 1 && toplay == W)
{
  for(i=1; i<20; i++)
    {
        current_piece = curr_pieces[i];
        xsel = x;
        ysel = y;

       if(cancapture)
        {

        if(poss[6] || poss[14])
        {
                xdest = xsel - 2;
                ydest = ysel + 2;
```

```
        move(idsq(current_piece),xdest,ydest);
        xmid = (xsel + xdest)/2;
        ymid = (ysel + ydest)/2;
        mcood = [xmid,ymid];
        move(idsq(mcood), 1000, 1000);
        break;
}
else if(poss[8] || poss[16])
{
        xdest = xsel + 2;
        ydest = ysel + 2;
        move(idsq(current_piece),xdest,ydest);
        xmid = (xsel + xdest)/2;
        ymid = (ysel + ydest)/2;
        mcood = [xmid,ymid];
        move(idsq(mcood), 1000, 1000);
        break;
}
else if(poss[13])
{
        xdest = xsel - 2;
        ydest = ysel - 2;
        move(idsq(current_piece),xdest,ydest);
        xmid = (xsel + xdest)/2;
        ymid = (ysel + ydest)/2;
        mcood = [xmid,ymid];
        move(idsq(mcood), 1000, 1000);
        break;

}
else if(poss[15])
{
        xdest = xsel + 2;
        ydest = ysel - 2;
        move(idsq(current_piece),xdest,ydest);
        xmid = (xsel + xdest)/2;
        ymid = (ysel + ydest)/2;
        mcood = [xmid,ymid];
        move(idsq(mcood), 1000, 1000);
        break;
        }
        }

        else if(canmove)
        {
        if(poss[2] || poss[10])
        {
                xdest = xsel - 1;
                ydest = ysel + 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
        else if(poss[4] || poss[12])
        {
                xdest = xsel + 1;
                ydest = ysel + 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
        else if(poss[9])
        {
                xdest = xsel - 1;
                ydest = ysel - 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
        else if(poss[11])
        {
                xdest = xsel + 1;
                ydest = ysel - 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
        }

    }

}
}

*/
```

```
proc autoplay : computer_to_play, toplay{

auto i;
if(computer_to_play == 1 && toplay == W)
{

if(cancapture)
  for(i=1; i<=20; i++)
    {
        current_piece = curr_pieces[i];
        xsel = x;
        ysel = y;

        if(poss[6] || poss[14])
        {
                xdest = xsel - 2;
                ydest = ysel + 2;
                move(idsq(current_piece),xdest,ydest);
                xmid = (xsel + xdest)/2;
                ymid = (ysel + ydest)/2;
                mcood = [xmid,ymid];
                move(idsq(mcood), 1000, 1000);
                break;
        }
        else if(poss[8] || poss[16])
        {
                xdest = xsel + 2;
                ydest = ysel + 2;
                move(idsq(current_piece),xdest,ydest);
                xmid = (xsel + xdest)/2;
                ymid = (ysel + ydest)/2;
                mcood = [xmid,ymid];
                move(idsq(mcood), 1000, 1000);
                break;
        }
        else if(poss[13])
        {
                xdest = xsel - 2;
                ydest = ysel - 2;
                move(idsq(current_piece),xdest,ydest);
                xmid = (xsel + xdest)/2;
                ymid = (ysel + ydest)/2;
                mcood = [xmid,ymid];
                move(idsq(mcood), 1000, 1000);
                break;

        }
        else if(poss[15])
        {
                xdest = xsel + 2;
                ydest = ysel - 2;
                move(idsq(current_piece),xdest,ydest);
                xmid = (xsel + xdest)/2;
                ymid = (ysel + ydest)/2;
                mcood = [xmid,ymid];
                move(idsq(mcood), 1000, 1000);
                break;

        }
    }   /* the end of loop */

else if(canmove)
    for(i=1; i<=20; i++)
        {
        current_piece = curr_pieces[i];
        xsel = x;
        ysel = y;
        if(poss[2] || poss[10])
        {
                xdest = xsel - 1;
                ydest = ysel + 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
        else if(poss[4] || poss[12])
        {
                xdest = xsel + 1;
                ydest = ysel + 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
```

```
        else if(poss[9])
        {
                xdest = xsel - 1;
                ydest = ysel - 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
        else if(poss[11])
        {
                xdest = xsel + 1;
                ydest = ysel - 1;
                move(idsq(current_piece),xdest,ydest);
                break;
        }
         } /* the end of canmove */

    }
}
```