Programming with Dependency

by

Timothy Heron

Supervisor: Dr. S. B. Russ

A thesis submitted for the award of
a Master's degree by research in Computer Science

Department of Computer Science
University of Warwick
Coventry
Warwickshire
CV4 7AL
U.K.

September 2002

Contents

List of Figures

Acknowledgements

This thesis would not have come about without the careful and expert guidance of my supervisor, Steve Russ.

I wish to thank all the people who have helped throughout my research, especially Meurig Beynon for constantly exciting me with ideas since I have known him.

I would like to give extra thanks to all the members of the Empirical Modelling Group at Warwick University for providing such a stimulating environment in which to work and think. Ashley Ward and Chris Roe provided friendly and invaluable help and assistance.

Declaration

This thesis is presented in accordance with the regulations for the degree of Master of Computer 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. The appendices contain source listings of the programs developed by myself during this research.

Abstract

An exploration of how dependency can be employed in the construction of traditional procedural and object-oriented programs. Practical examples developed with (the Java Maintainer Machine) JaM2 are used to compare definition-based modelling with object-oriented development.

Keywords: definitive scripts, Empirical Modelling, dependency, programming

Chapter 1 – Introduction to Dependency

Introduction

Empirical Modelling (EM) is a novel style of modelling which focuses on state changes and agency. Interaction with the model is conducted in an open exploratory manner and in an experimental way to refine the model. EM uses definitive scripts as the basis for describing the models. The experimental interaction is performed by making redefinitions to the definitive scripts used within the model.

This thesis concentrates on definitive scripts rather than the more general subject of Empirical Modelling. The main focus is on dependency which forms the mechanism for updating state within definitive scripts. It explores dependency within programming, how definitive scripts use dependency and how using dependency within procedural and object-oriented programming can be a powerful technique.

It is worth noting that Empirical Modelling incorporates far more than simply modelling with definitive scripts and its theoretical emphasis is on defining a broad, principled foundation for computing [Bey99]. The Empirical Modelling paradigm includes notations for the initial analysis of a domain (LSD), a number of notations and tools for modelling with definitive scripts and an observation-oriented modelling approach to represent agent interaction.

Definitive scripts are used to represent state and also as the mechanism for carrying out state change within an EM Model.

Thesis Outline

This thesis is divided into six chapters, of which this is the introductory first chapter.

Chapter 2 looks at modelling with definitive scripts and object-oriented modelling and draws some comparisons between the two for static models and for dynamic models. The focus of this chapter is on the structural differences between an OO model and an EM model.

Chapter 3 examines the way we can program object-oriented applications if we add a dependency maintainer to our tool set. A comparison is made between two different ways of maintaining dependencies (one a dependency tracker developed by Michael Perry [Perry01], the other a dependency maintainer API by Richard Cartwright [Car01]).

Chapter 4 looks at the fundamentals of definitive programming by visiting the Abstract Definitive Machine (ADM) and re-implements some of the models written with the ADM in more recent dependency maintainers. It looks at some of the problems with the ADM and attempts to use object-oriented principles to resolve these problems.

Chapter 5 examines modelling with spreadsheets and adds some theoretical foundation by focusing on Brian Cantwell Smith's ideas about programming and their relationship with Empirical Modelling.

The thesis ends in Chapter 6 by drawing conclusions from the research described and by proposing further work.

Research Motivation and Aims

In this dissertation I shall show the value of adding explicit dependency maintenance to areas of programming where the dependencies have normally been purely implicit and thus show the power of combining implicit and explicit dependency maintenance as is done in some of the tools used in Empirical Modelling. I shall also show some of the advantages that this

spreadsheet style development can lend to application development by making applications more ‘open’. In addition I shall demonstrate the advantages that can be brought to object-oriented software development by considering a limited form of dependency maintenance called dependency tracking.

In this chapter I shall try to make clear the concepts of definitive scripts and two different forms of dependency that I believe make a significant difference to the complexity of programming.

Dependency

Dependency exists in all aspects of life; the position of a cup depends on the position of the table it is resting on. Moving the table also moves the cup. The position of the cup can be defined in terms of the position of the table; if the cup falls off the table then its position is redefined in terms that do not include the table position.

We use ‘dependence’ in natural language in a number of different ways [Dict00]:

1. The state of being dependent, as for support.
    a. Subordination to someone or something needed or greatly desired.
    b. Trust; reliance.
2. The state of being determined, influenced, or controlled by something else.
3. A compulsive or chronic need; an addiction: *alcohol dependence.*

We are interested in use number 2 and within this use of dependence we can identify many different kinds of dependency:

> **Temporal** - The position of a shadow on a sun dial *depends* on the current time.
>
> **Physical** - The fact that my cup of coffee stays on my desk *depends* on the forces present such as gravity, the opposing force from the desk, etc.
>
> **Mathematical** - The amount of money I have to pay for car tax *depends* on the capacity of the car’s engine.

Computer applications that model real world artefacts often have aspects of dependency within them. A home finance package will exhibit dependency whereby the bank balance will depend on the transactions entered. The number of pages a word processor uses when printing a document depends on the amount of text entered, font sizes, etc.

The term “dependency” within programming is usually used to describe cause and effect relationships between observables [Run02]. A common high-level programming statement is that of assignment e.g. “`a = b + 2`”. This statement describes a state change of the memory locations at `a`. Before the statement the values `a` and `b` have no dependency between them. While the statement is being executed `a` is dependent upon `b`. Immediately after the statement has executed `a` is up to date but the dependency no longer exists. If a subsequent statement modifies either `a` or `b` then there is no attempt to maintain the dependency that was set up.

This is in contrast to other forms of modelling (such as spreadsheets and application packages such as MathCAD and Mathematica) where once a relationship is described it is maintained throughout the life of the model. If we could employ the use of these kinds of maintained relationships within programming then we could potentially reduce the amount of state change code that we would need to write.

Within these forms of modelling definitions are created. A definition is any well-formed statement that expresses dependency. This dependency is not as short-lived as the programming assignment statement. Once a definition has been created then that dependency is valid until such time as it is re-defined or deleted. This creates a different kind of assignment statement that has a lifetime and a persistency associated with the relationship it defines.

Forms of dependency

We propose in this dissertation to develop a distinction between expressing dependency in explicit and implicit forms.

1. Dependency expressed in an **explicit** form uses a well-defined underlying algebra between values of data types, using functions over known data types. A spreadsheet is an example of explicit dependency using basic mathematical types and the spreadsheet formula language.
2. Dependency can also be expressed **implicitly** through a system of procedural actions performed by software objects (class instances or pointers to abstract data types) to signal when they should update. State changes rely on procedures or methods performing the correct updates. The `make` application that is used to maintain dependency between source files for an executable when it needs to be rebuilt is an example of implicit dependency.

The difference between these two can be summed up as follows: Within an explicit dependency there is enough information known to infer the dependencies and order the necessary state changes to update the dependency. With implicit dependency this information is not there and the dependencies have to be stated before automatic updates are possible. For example, `make` is an implicit dependency maintainer since each command's dependencies have to be declared in order for `make` to order the execution of commands while within a spreadsheet the evaluation engine derives the dependencies from the formula itself.

The extra information provided by explicit dependency allows a dependency maintainer to determine the evaluation order of relationships. For example given the two definitions:

```
a is b + c
x is y + z
```

adding a new definition

```
c is 2 * x
```

affects the order of evaluation, this order can be determined from the definitions themselves. An implicit dependency maintainer would need to be told that `c` depended on `x` before it had sufficient information to correctly work out the order.

There are other forms of dependency in use which are often associated with declarative programming languages such as Prolog. In Prolog declarations are used to describe constraints which a constraint satisfaction technique can be used to solve, there may be many possible solutions to a set of constraints or none at all. A set of constraints combined with a particular constraint satisfaction technique provide a form of dependency maintenance. When a variable is changed the other variables are modified so that the declared constraints are still adhered to (if this is possible).

Functional programming has a similar focus on defining relationships as definitive programming. Functional programming is a style of programming that emphasizes the

evaluation of expressions, rather than the execution of commands. The expressions in these languages are formed by using functions to combine basic values. Functional relationships are different to relationships made in definitive scripts. Within definitive programming the values and definitions are regarded as state that can be changed while functional programming does not have a notion of state and all processing is performed by the application of functions.

Implicit dependency can be analysed prior to development and if implemented correctly is efficient, however it is easy to mistakenly do too much work by performing needless updates or to make errors where a variable is not updated when it should be. This preconception of dependencies before a development starts naturally prevents a development from being very open. The difficulty of the analysis of all the dependencies is greatly magnified in developments that require parallel execution and synchronisation between updates.

The Spreadsheet, an explicit dependency maintainer

A spreadsheet application is the most common example of explicit dependency. A spreadsheet allows a modeller (for example, an accountant) to establish dependency between elements of a model that are related in the real world (such as expenditure and profit). These elements are then automatically updated as the situation (fortunes of the company) changes over a period of time.

Formulae in a spreadsheet are a common form of definitions. If the formula in cell A1 references cell B1 then the cell A1 formula will be recalculated automatically if cell B1 changes. We can think of a definitive script as a generalised spreadsheet where we are not limited to the pre-defined formula and syntax of a spreadsheet but we have more freedom to define our own types, operations and notations.

Changing the formula in a spreadsheet cell can be considered the equivalent of a redefinition in a definitive script.

Traditional spreadsheets are constrained in several ways:

A fixed naming convention (cells are referenced as A1, B2, etc).

A limited number of data types, we cannot define our own abstract types or operations over them.

Spreadsheets have a fairly limited notation for describing relationships. The data types supported are limited to numbers, dates, strings and error codes and although the operations that can be performed are extensive there is no way to extend the types available. The naming convention for definitions is usually fixed into a 2-d cell structure although on some spreadsheets it is possible to label cells so that they can be referenced by a label rather than the grid reference.

Spreadsheets recalculate those cells that are dependent on other cells whenever the values in these other cells change. This type of calculation helps to avoid unnecessary calculations. If a formula refers back to one of its own cells (i.e. a circular reference), the user must determine how many times the formula should recalculate.

The spreadsheet formula language is a task specific end-user programming language; it is one of the few cases where a text-based language is used by the end-user of the product. Many programs contain macro languages and scripting tools for the so called ‘power’ user but the spreadsheet requires the use of a programming notation even for trivial calculations. The language used is expressive and easy to use, even fairly basic spreadsheet users are happy working with the more complex functions available (such as the ‘if’ function).

Writing a spreadsheet differs from software development in that very rarely is a spreadsheet specified (or even planned) in advance. The user typically writes the spreadsheet as they go along. This can lead to "spaghetti logic" where relationships are created all over the spreadsheet. Users however normally use fewer than 10 functions within their spreadsheets [Nar93 p43]. This is because the spreadsheet is targeted towards the kinds of problems its users want to solve; it uses a definitive notation that has a wide area of applicability to everyday problems.

Spreadsheets provide control statements that allow decisions based on conditions. These conditions are normally what give traditional programming its complexity. As control is transferred between different sections of program it can be extremely difficult to follow the logic of a complex program. Spreadsheet conditionals are different, they have only a local effect; the value of a cell changes and nothing more; propagation of the decision does not affect the rest of the spreadsheet. This restriction to local conditionals removes a lot of the power of a spreadsheet compared to a programming language but it is far easier for users to understand the effects of spreadsheet `if` function compared with Java's `if` statement even if the user fully understands the syntax of both [Nar93 p47].

Spreadsheet functions remove the need for intermediate values (such as loop counters) and the variable names associated with these values. Users often find such constructs difficult to grasp within procedural programming because they are not relevant to the task they are trying to perform. These issues of spreadsheets are discussed at greater length in Chapter 5.

Other definitive notations exist, the languages used in products such as Mathematica or Maple can be viewed as a form of definitive notation targeted towards scientific mathematical calculations.

Re-constructing a typical spreadsheet model using a general purpose programming language is far harder than constructing the original model and is far less amenable to change. Obviously, the wide range of complex programs that can be written in a language such as C++ cannot be written in a spreadsheet but it is certainly feasible that definitive notations can be used to aid or simplify traditional programming by reducing programming complexity.

Spreadsheets and Programming

Spreadsheets for the last few years have been supplied with general-purpose programming languages to provide sophisticated macro programming facilities. One such example is Microsoft Excel that (along with other Microsoft applications) has support for the Visual Basic for Applications (VBA) programming language.

An object model has been designed that exposes the functionality of the Excel spreadsheet to the programming language to provide access to Workbooks, Worksheets, Cells, Values, Formulae, etc. Spreadsheet events (such as selection, recalculation, printing, opening and closing) can cause VBA code to be triggered to perform further manipulations. VBA functions can be referenced just like standard Excel functions.

Defining the following function in VBA:

```
Public Function times2(x As Integer)
    times2 = 2 * x
End Function
```

allows the function to be referenced in the following formula:

```
=times2(B1)
```

This associates procedural actions with recalculations and provides a mechanism to link the procedural world and the definitive world by adding the ability to use implicit dependency within an explicit dependency model. The programmer is, however, still constrained by the inability to define their own data types. This means that all modelling that is done within a spreadsheet must be done by manipulating primitive data types, no abstraction to a problem domain is possible.

UNIX `make`*, an implicit dependency maintainer*

The UNIX `make` utility is a common utility that uses dependency relationships to keep files up to date.

The intended use of the `make` utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. However, it can be used to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use `make`, you must write a file called the `Makefile` that describes the relationships among files in your program. This file states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable `Makefile` exists, each time you change some source files, the simple shell command `make` suffices to perform all necessary recompilations. The `make` program uses the `Makefile` data and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the `Makefile`.

`make` updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

A `Makefile` consists of a sequence of commands which describes how a program (or other target) can be constructed from source files. The construction sequence is described by ***dependency rules*** and ***construction rules***.

A dependency rule has two parts - a left and right side separated by a '`:`'

```
left side : right side
```

The `left side` gives the names of the ***target*** (the name of the program) to be built, whilst the `right side` gives names of files on which the target depends (e.g. source files)

If the ***target*** is **out of date** with respect to the constituent parts, ***construction rules*** following the dependency rules are obeyed.

An Example

Let's look at an example `Makefile` :

```
prog: prog.o f1.o f2.o
      cc prog.o f1.o f2.o -lm etc.
prog.o: header.h prog.c
      cc -c prog.c
f1.o: header.h f1.c
      cc -c f1.c
f2.o: ---
```

```
----
```

make would interpret the file as follows:

1. prog depends on three files: prog.o, f1.o and f2.o. If any of the object files have been changed since last compilation the files must be relinked.

2. prog.o depends on two files. If these have been changed prog.o must be recompiled.

The last three commands in the Makefile are called ***explicit rules***, since the files in commands are listed by name. We can use ***implicit rules*** in our Makefile which let us generalise our rules and save typing.

We can take the following bit of Makefile

```
f1.o: f1.c
      cc -c f1.c
f2.o: f2.c
      cc -c f2.c
```

and then generalise this to:

```
.c.o:   cc -c $<
```

We read this as .source_extension.target_extension: command

$< is shorthand for file name with .c extension.

The use of the term 'implicit' within a Makefile is not the same as that in implicit dependency within a Makefile. Within a Makefile implicit rules are generalisations of a single dependency rule and a single construction rule while implicit dependency is the use of procedural actions to keep the dependency up to date. An example of explicit dependency that used implicit rules could be imagined (such as a spreadsheet that allowed formulas to be specified using regular expressions).

Using make in an unconventional way, as a spreadsheet

make is not restricted to compiling large programs and the tasks associated with these builds. It can also be use in more general ways. Here is how make can be used to perform the dependency maintenance required by a spreadsheet:

We use the UNIX command bc to perform the arithmetic calculations of the spreadsheet. Our simple spreadsheet will place the value 40 into cell A1, 60 into cell B1 and A1+B1 into cell C1 thus performing a simple addition.

We are going to use the convention that files named *.formula are shell scripts that perform calculations and create files called *.value that are the results of the calculations. We have three cells and the files for these cells are:

```
A.formula:
      echo "40" | bc >A1.value
B.formula:
      echo "60" | bc >B1.value
```

C.formula[1]:

[1] This script looks quite complicated but most of it is to do with the fact that 'bc' does not work if there is excess white space within the input formulae. cat and tr are standard unix commands.

```
echo "+" | cat A1.value - | tr -d [:space:] | cat - B1.value | bc
>C1.value
```

We could then write a simple shell script that runs each one in turn but we can produce a more efficient solution using the following `Makefile`:

```
C1.value : A1.value B1.value C1.formula
      sh C1.formula
A1.value : A1.formula
      sh A1.formula
B1.value : B1.formula
      sh B1.formula
```

This `Makefile` declares that A1.value and B1.value depend on their respective formulas being evaluated first. It also declares that C1.value depends on these two values.

Running '`make` Sheet'

Typing '`make`' without any *.value files existing runs the scripts A1.formula, B1.formula and C1.formula in that order. Modifications can then be made to any file and when '`make`' is run again only the files that need to be updated are updated. So, modifying A1.formula and re-running 'make' will cause A1.formula and C1.formula to run, B1.formula is not executed.

Limitations of `make`

Make uses the timestamp of each file to decide if it needs to execute the construction rule for that file, if the file has not changed but its timestamp is recent (for example after running the command '`touch`') then the construction rule is run without any need.

The dependency maintenance is not automatic, i.e. when a source file is updated the construction rules are not executed until `make` is run again.

The dependency is left to the user to figure out, the construction rules could have side effects or additional dependency that `make` will not have information about. For example a compiler will likely create and delete a number of temporary files that are not specified in the dependency list.

Timestamps are one way of determining whether values are out of date. This technique is fine for `make` since it is not run many times a second, when programming with dependencies that will need to be updated quickly a more efficient and reliable mechanism is needed.

The difference between implicit and explicit dependency

Now that we have seen some examples of dependency in practice we can summarise and clarify the difference between implicit and explicit dependency. Explicit dependency is that found within a spreadsheet and within definitive scripts, it relies on a formal notation. The spreadsheet definitive machine can interpret and work out the dependencies from the script itself.

Implicit dependency as found in a `Makefile` relies on side effects caused by procedural commands (i.e. compiling). Since there is no formal notation to describe these side effects then the dependency maintainer (in this case `make`) requires that the dependencies are spelt out to it.

Traditional programming makes no use of explicit dependency and any dependencies that exist (whether they are message passing, propagating updates, etc.) require specific

procedural actions to be written and maintained by the programmer to keep the dependencies valid.

Definitive Notations

Empirical Modelling makes use of sets of definitions (definitive scripts) to represent the views of agents and the interfaces between them. Such scripts provide a powerful means to represent the state of a complex system, and the perceptions of the agents within it. The term 'definitive notation' has been adopted for programming notations that are 'definition-based'. The Empirical Modelling group has implemented several definitive notations:

Eden - a general-purpose language with syntax modelled on C, this is more than a definitive notation since it provides support for procedural code. Eden refers to a family of languages including `ttyeden` (command line Eden), `tkeden` (Eden graphical environment), `dtkeden` (distributed Eden). It is used as the basis for the following implementations.

DoNaLD - a "Definitive Notation for Line Drawing" used for simple 2-d graphics.

Scout - a "definitive notation for describing Screen layout" describes the configuration and status of windows in a display.

Sasami - used for describing 3-d graphics.

Eddi - an "Eden Definition Database Interpreter" definitive database notation.

Versions of these tools for some standard platforms and various papers and previous theses are available from http://www.warwick.ac.uk/modelling.

Other definitive notations are in the process of development but do not have implementations as of yet. It is clear that a well designed definitive notation provides an easy way to model sets of problems within a specific domain. Chapter 4 shows a network design notation that can be used to build up models of sophisticated computer networks easily.

Definitive Scripts

A program created from definitions written in a definitive notation is called a definitive script.

This dependency is maintained while the definition exists. These definitions are of the form "identifier **is** expression", for example "b **is** a + 2". If the value of 'a' changes the expression is re-evaluated automatically to maintain the definition. In this way the value of 'b' is always up to date. This **is** notation is originally from Eden and most definitive notations have followed suit, however some notations like DoNaLD use = if they do not have any other meaning for =.

Combining Explicit and Implicit Dependency - Empirical Modelling Tools

Empirical Modelling relies on defining synchronised patterns of changes in observables. Observables that are seen as being dependent on others can have their dependencies expressed by describing the relationship in a definition. EM models that are built are normally expressed as definitive scripts.

The most successful and widely used Empirical Modelling tools have combined the powerful explicit dependency of definitive scripts with the less formally defined implicit dependency of procedural programming. Tools such as Eden combine definitive notations with functions and procedures in a similar way to the modern programmable spreadsheet.

Both are hampered by the lack of ability to define new data types easily (in Excel this is impossible, in Eden it requires a new notation to be developed, although with the advent of Chris Brown's [Brown00] agent-oriented parser the mechanics of this has become considerably easier).

Chris Roe of the Empirical Modelling group at the University of Warwick has developed a prototype spreadsheet using the group's main tool `tkeden`. This exhibits all the functionality of traditional spreadsheets but also has the ability to manipulate non-standard data types within cells. For example, it is possible to place a geometric shape in a spreadsheet cell and write formulae that transform this shape so the user can define a cell as '`D3=rotate(C3,45)`' and this will fill the cell D3 with the shape in C3 rotated by 45 degrees. Similar functionality is available in the Forms/3 spreadsheet developed by the Forms/3 Group at the Department of Computer Science, Oregon State University [Burn01] where they are exploring the boundaries of spreadsheet usability and end-user programming.

Summary

This chapter has introduced dependency and described some of the different forms it can take. We have seen how dependency can be expressed implicitly and explicitly and how this is used by existing tools. It has also described definitive scripts as used in the Empirical Modelling group at the University of Warwick.

Chapter 2 - Two Approaches to Modelling

Objective

The aim of this chapter is to present two particular models and two approaches to the modelling process. We compare an EM Model of a draughts game with an object-oriented version which places emphasis on static modelling. We use these two models to look at the differences in structure between them. A comparison between the main analysis technique (LSD, explained further in Chapter 5) used within Empirical Modelling and the many different techniques used within the OO paradigm will not be made. There are too many approaches (all applicable in different contexts) to provide a complete study within this document and it will be more useful to use an OO model that follows the basic concept rather than one that exhibits particular peculiarities that may arise from a particular analysis technique.

We also compare an OO development of a vehicle cruise control simulator developed by the current author with an EM Model of the same simulator; this model introduces the use of time and we use it to compare aspects of modelling a dynamic system.

Principles of OO Development

The core concept of OO is that systems are built out of Objects with a clearly defined exterior and a completely opaque implementation. Objects can only be interacted with by sending them messages, and a system achieves its functionality through the behaviour associated with those messages. Objects have their own internal representation of data that is inaccessible to other objects, update (mutator) methods and accessor methods manipulate and return this data in a predefined specified format to other objects. OO is a simple concept, but good design takes lots of learning and experience.

Software design and object-oriented design are vast subjects and both have a huge literature covering them. There are many methodologies, techniques and guidelines for OO development but we are concerned with the aspects of dependency.

Object-oriented Dependency

In the first chapter we discussed two types of dependency, implicit and explicit. The implicit type of dependency is easy to implement within object-oriented programming. All dependencies that will be represented by an application must be preconceived before the application is compiled. Methods must be added to objects since their internal data representation is linked directly to these update methods. These methods must be called at the correct time so that the update can take place and the state of the system changes in a consistent way.

Method calls are not monitored so unnecessary updates often take place (if the coder has not preconceived and optimised their code to avoid all unnecessary updates). Parallelisation and synchronisation of updates cannot be determined at runtime and must be considered by the developer before executing the program.

Object-oriented programming techniques do not support explicit dependency; the method calls used to perform explicit updates would require ordering information that is not present in a format interpretable by a machine. The meta-information that is required for this ordering is not available from the procedural code used within OO programming. It is by adding this meta-information through definitive programming that ordering becomes possible. A developer can implement dependency using explicit definitions of dependency between data

values. Instead of calling methods, dependency is expressed between data values using operators. The meta-information that is required for ordering updates is contained within these definitions.

Data values are instantiated and updated as necessary to satisfy the dependencies expressed. These dependencies can be dynamically changed at any time in a completely open-ended way.

Empirical Modelling makes use of explicit dependency in a way that object-oriented modelling does not. Definitive scripts are produced as a computer-based artefact of the modelling process. Definitive scripts describe dependency using definitions like these:

```
weight is accel_due_to_gravity * mass;
```

This creates a persistent relationship that is maintained throughout the life of the model. Definitions, and therefore relationships, can be redefined or removed entirely from a definitive script. Creating a model with the intent to produce a particular artefact, rather than simply modelling for analysis or exploration, is equivalent to programming. Definitive scripts can be used as programming languages by treating definitive scripts as programs (not purely as models). Programming in this way is termed within this thesis 'programming in the definitive paradigm'.

The Draughts Model

Draughts is an EM model developed in tkeden, originally by Y M Au-Yeung and further enhanced by S. Rawles in [Draughts95] and [Draughts96] respectively.

The original aim of studying the game of draughts (or 'checkers') was to construct a model that initially simply described the structure of a draughts board and the pieces on the board without any reference to a game of draughts. The idea was to separate definitions into different layers with each layer describing a game in more detail. The bottom layers would describe the lines making up the board display, layers above would describe how and where pieces could be added to the board, the top layers would allow legal moves within a game and the uppermost layer would control which player's turn it was and have general game control.

By constructing the model this way, not only did it provide several layers of abstraction to aid the understanding of such a model but it also kept the purpose of the model open, it was only at the very last stage that the model was 'locked' into a game. At points before this the model was just that, a model of a draughts board and each layer was a different way of thinking about this board (whether it was thinking about the construction of the pieces, or the possible legal moves that could be played from a particular position).

The Eden version of the model [Draughts95] illustrates some of the key concepts of definitive programming.

Eden EM Version

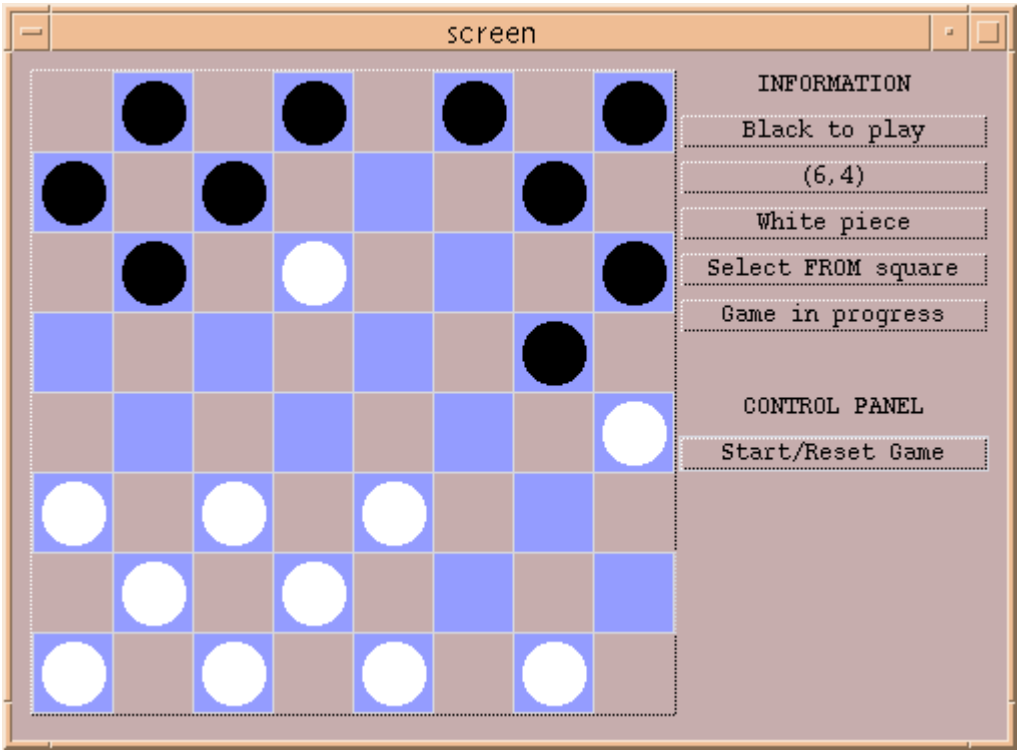

Figure 1 – Eden Draughts

The board shown in Figure 1 – Eden Draughts is described as follows:

First a grid of 8x8 squares is defined, this is done by defining the upper-left and bottom-right points of each square, they are named following a convention that makes it obvious where the points are intended to be in relation to the squares they locate. Definitions are made that are global to the entire board; these include things such as the height and width the squares should be and offsets that locate the board on the screen. Each point's definition is constructed by a calculation involving these global definitions.

```
point p11 = { ( x11 - 1 ) * sqwidth + pxoffset , ( 8 - y11 ) * sqheight + pyoffset } + p;
point q11 = { ( x11 * sqwidth ) + qxoffset , ( ( 9 - y11 ) * sqheight ) + qyoffset } + p;
point p13 = { ( x13 - 1 ) * sqwidth + pxoffset , ( 8 - y13 ) * sqheight + pyoffset } + p;
point q13 = { ( x13 * sqwidth ) + qxoffset , ( ( 9 - y13 ) * sqheight ) + qyoffset } + p;
…
```

where `xab` is the x-coordinate of square (a,b) and `yab` the y-coordinate. `sqwidth` and `sqheight` are the width and height of each individual square and `pxoffset`, `pyoffset`, `qxoffset` and `qyoffset` are used to centre the board on the display.

Once all the points have been defined the squares are defined as Scout windows located on these points:

```
window square11 = {
        type: DONALD
        box: [p11,q11]
        pict: "piece11"
        bgcolor: bgcol
};

window square13 = {
        type: DONALD
        box: [p13,q13]
        pict: "piece13"
        bgcolor: bgcol
};
…
```

The pieces are represented by DoNaLD definitions that define a circle for a normal piece and two concentric circles for a King piece. Each piece can be coloured White (a white piece exists on the square), Black (a black piece exists on the square) or Background (no piece exists on the square). The colour of each square is defined by a definition that uses that position of the square and the pieces on the board to determine if a piece is on the square and what colour it should be if there is.

For each piece that is about to be played there are 8 possible moves. The piece can perform 4 capturing moves and 4 non-capturing moves (ignoring the direction of the player). Definitions are set up that describe the circumstances when these moves are allowed for each piece. So definitions are created that specify 8 possible moves by a King piece and further definitions are made to restrict the normal pieces from the 4 reverse movements that normal pieces are not allowed to perform.

Performing moves is done by the player clicking on the piece to move followed by the square to move it to. A definition is made that controls the current state of the moving, i.e. whether the user is selecting a 'from' piece or a 'to' piece. If the user selects a 'from' piece then a definition is made that makes it the current piece. The definitions that depend on this are then automatically evaluated and the possible moves that the player can make calculated. If the user selects a 'to' square then a check is made that this square is in the list of the possible moves made from the current 'from' piece, and if it is, a move is made.

The game is controlled by a set of definitions that keep track of whose turn it is, the currently playing pieces for each player, and the current state of the visible control panel.

The model is based purely upon definitions and is very flexible to change. By making redefinitions nearly any aspect of the game can be changed without serious implication for other sections of the games. The positions of the squares can be changed without affecting the way the game plays. It is only by convention that all 64 squares are needed in a game of draughts; only 32 are ever played upon. It is easy to modify this model to rearrange the board layout so that only 32 squares are displayed.

The appearance of the draughts definitive script listing is that of a spreadsheet with only the formulas displayed and the cell boundaries removed i.e. there is very little procedural code and it is possible to envisage this model being developed on a spreadsheet (since the non-display datatypes used are simple, mainly boolean) with cell references instead of named definitions. There are many repeated definitions as often occur within a spreadsheet, for example, in a spreadsheet a column B of numbers can be doubled by applying a formula to every cell; thus creating as many formulae as cell values. The same applies to the draughts model where there are as many definitions for determining whether a piece is crowned or not as cells exist on the board. Traditional programs would have a single statement that is looped to perform this same decision; this brings in the added complexity of control statements and execution flow that spreadsheets (and definitive scripts) avoid.

There are limitations of the implementation of the tkeden environment that prevent the full adoption of the definitive modelling approach. The event handling of tkeden is inspired by the Visual Basic approach of calling procedural code when a user interface event occurs. A 'more' definitive version of this model would cause a chain of recalculation to occur.

This is a common problem when joining traditional procedural code with definitive scripts. It is possible to write definitive scripts that produce output in the following way:

```
output = concat (output, |new|)
```

Here the current value of new is taken and the output is redefined as the current output plus the new output, this can be written as a print operator.

However since operating systems are procedural code and do not follow the definitive paradigm all current implementations of definitive machines on top of existing operating systems require swapping to the procedural paradigm to display this output.

This output is an example of a continuing conflict between definitive systems and the procedural systems they are implemented upon. A purely definitive machine can be created if all existing operating system procedural code is ignored, for example the DAM (Definitive Assembler Machine) [Car98] is an example of such a system whereby definitions form the basis of the whole machine instead of an application running on top of a procedural operating system.

This conflict between the definitive paradigm and the procedural is most obvious when transferring data between the two paradigms, for example when a script must produce some output to a console or file, or when interfacing to existing code such as a C library function. Two main techniques are used:

1. A polling system whereby a procedural program monitors a definitive script repeatedly and updates values based on the current values of the script.
2. Side effects where re-evaluation of certain definitions trigger procedural code to cause updates, this is a more efficient method than polling but requires extra state to be stored within the procedural code to take into account dependency maintenance factors (such as definitions being evaluated more than once, or not being evaluated until needed).

Java Object-oriented Version

The Java Object-oriented version was initially inspired by the Eden draughts model [Draughts95] and developed by Simon Rawles in order to provide a comparison between object-oriented and EM techniques [Draughts96]. This version illustrated some of the differences between definitive modelling and conventional programming but the implementation, although procedural was not particularly object-oriented. As part of this comparison work the model was revised by the current author [Appendix 2] using refactoring techniques as documented by Martin Fowler [Fow00]. This has resulted in a more object-oriented design and provides a stronger basis to compare the two modelling paradigms. A screenshot is shown:

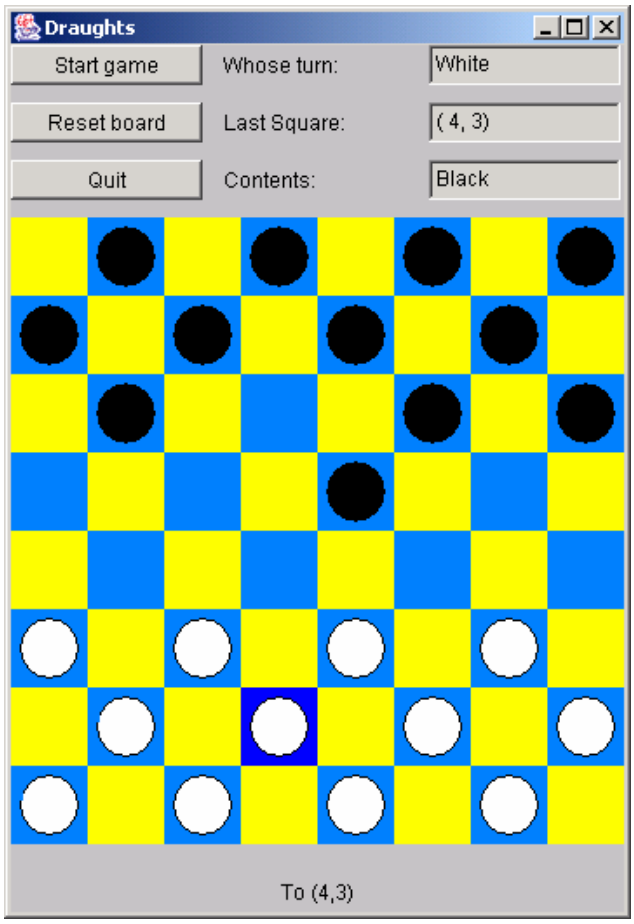

Figure 2 – Java Draughts

The board is modelled as a 2-dimensional array of Square objects. Each Square has a graphical representation (A BoardSquare) and can contain a Piece object. The BoardSquare defines the colour the square should be drawn and also handles events such as the user clicking on the square.

The Pieces themselves are modelled by a single Piece object, a piece defines properties such as **isKing** and **isWhite**.

By simply manipulating the information model (IM) it is possible to write a section of code that will build up a board with pieces on it, change the locations of pieces and set up positions without having to play a game. A diagram of the IM model is shown in .

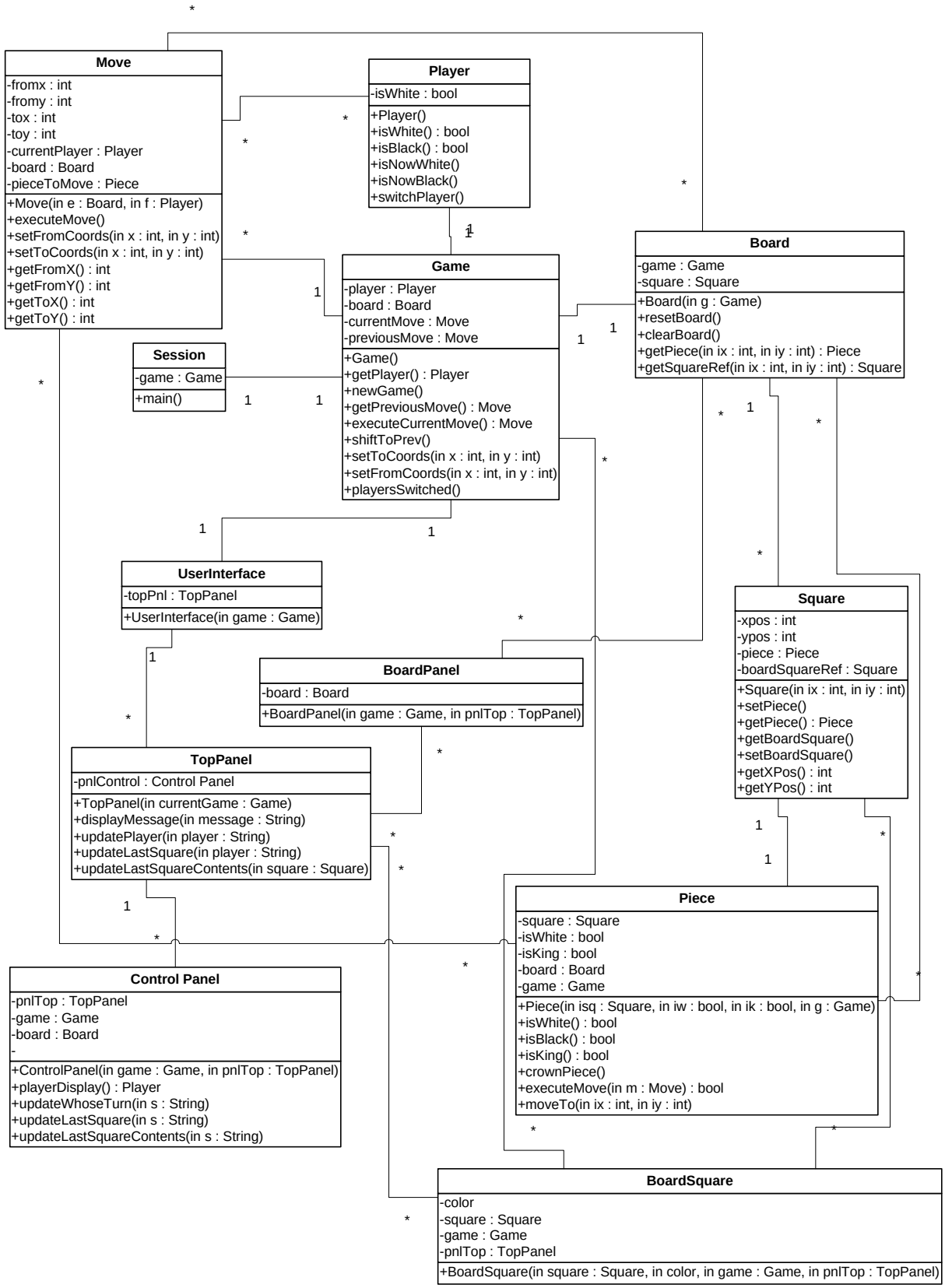

Figure 3 – Draughts Static Object Model

Making changes to the board is simple if the possible changes have been preconceived and the model is flexible. For example it is easy to add a third colour of piece (e.g. red) by altering the Piece class. However because the board has been defined as a 2D array of squares it is difficult to change the shape of the board arbitrarily. While in the tkeden version it was easy to compress the board to ignore the white squares it is more difficult in the OO version since the squares have been defined in a loop and not declared individually by the programmer.

This procedural version defines and processes the elements of the board using a nested loop. While the definitive script would make as many definitions as squares on the board, the procedural one repeats the same statement and applies it to each board square. While this reduces the size of the program code it makes the code harder to understand and verify since additional information such as loop iterators and conditional branching are added which have no direct equivalence in the physical model. It also makes exceptions to the rule harder to implement, if we decided that a single square was to look different (for example to include the logo of the board manufacturer in a corner square) then the definitive version requires changes only to the visual definition of that particular square. The procedural version requires a conditional statement within the square drawing loop that before drawing the square checks its location and then executes a different draw square routine depending on whether or not the square should have the logo.

If we decided every square should have completely different properties then the procedural version would have to be completely re-written so that a loop is not used. It is possible to argue that the procedural version need not have been written with a loop in the first place but this goes against the traditional aim of procedural programming, that of generalising code so that lines of code are not repeated unnecessarily.

Object-oriented programming focuses analysis and design on the structure of the objects within the model at an early stage, the first stages of analysis and design (whatever specific OO methodology is being used) are to establish potential objects and then create a static structure with objects related by association or composition. This emphasizes the importance of good structure in making a good OO design. OO design methodologies and languages enforce structural constraints, so that even though an OO program has access to all of the memory it uses only specific objects can access certain data members. These access modifiers are imposed by the programmer in order to constrain accesses to the data so that data can only be changed in a predictable way to maintain consistency.

Definitive scripts have no such mechanism for constraining access to data, this is the same as a spreadsheet where any cell can reference any other cell. Any structure has to be maintained solely by the modeller through their own discipline. If a programmer tries to violate the structure of an OO program then the program will not compile or will throw an exception upon running. In a definitive script structural violations are possible, this contributes to the openness of the model (anything is possible at any time) and allows experiment. Once the experiments have been conducted and approved then the modeller has to go back and rework the structure to incorporate the experimental definitions in order to keep the structure clean i.e. remove unnecessary redefinitions, ensure consistent naming, etc.

There are guidelines as to what makes a good object and there are techniques for changing the structure of an OO model. For example, the following maxims are from Kent Beck's Smalltalk Best Practice Patterns [Beck97]. These describe properties that should be in 'good' OO designs and 'good' OO code:

1. **Once and only once**, in a program written with good style, everything is said once and only once.

2. **Lots of little pieces,** good code invariably has small methods and small objects.
3. **Replaceable objects,** good style leads to easily replaceable objects.
4. **Movable objects,** objects should be easily moved to new contexts.
5. **Isolate rates of change,** don't put two rates of change together.

It is relatively straightforward to analyse an OO program and to see if the objects fit these maxims.

Agent-oriented analysis as in Empirical Modelling supplies definitive modelling with a framework for definitive programming, every modeller is given complete freedom to model as they see fit and each modeller builds their own style and set of common definition structures that they use. It is quite clear, however that writing definitive scripts is opposed to some of these object-oriented maxims. For example, the first maxim of only writing a section of code once and reusing a general routine if the code is required more than once is directly opposed to the practice of definitive scripts (and spreadsheets) making many similar definitions.

There have been attempts to define good practices for building spreadsheet models. [ISL95] advocates dividing a spreadsheet into two views, the physical view describes where cell is positioned how each cell is formatted, while the logical view defines the schema and data properties of the cells. Once these two views have been separated out then a process of factoring is carried out every time the spreadsheet is modified. Factoring reduces a spreadsheet to its principal properties. Upon modification of the spreadsheet these properties are compared and user-verification is sought if fundamental changes to the spreadsheet are detected.

The Vehicle Cruise Control Simulator (VCCS)

The VCCS model was developed in 1991 by Ian Bridge and is a successful attempt to model a real world system using the EM group's tool 'tkeden'. It has become a standard demonstration of the tkeden tool although in many ways it is not representative of a typical EM model.

This model displays a vehicle, a driver's view of the important controls - ignition switch, speedometer, brake and accelerator. External views of the vehicle and its progress over a hill are also displayed. The point of engineering interest is that the model includes a cruise control - a device which uses a feedback mechanism to control the car's throttle automatically (instead of being under manual control from the accelerator). The mechanism seeks to maintain a constant target speed regardless of the changing gradient on the road.

It is unusual in that it is a dynamic model that uses a clock counter to trigger changes. This makes much of the model seem like a procedural program that has a program counter rather than a definitive script.

Eden EM model

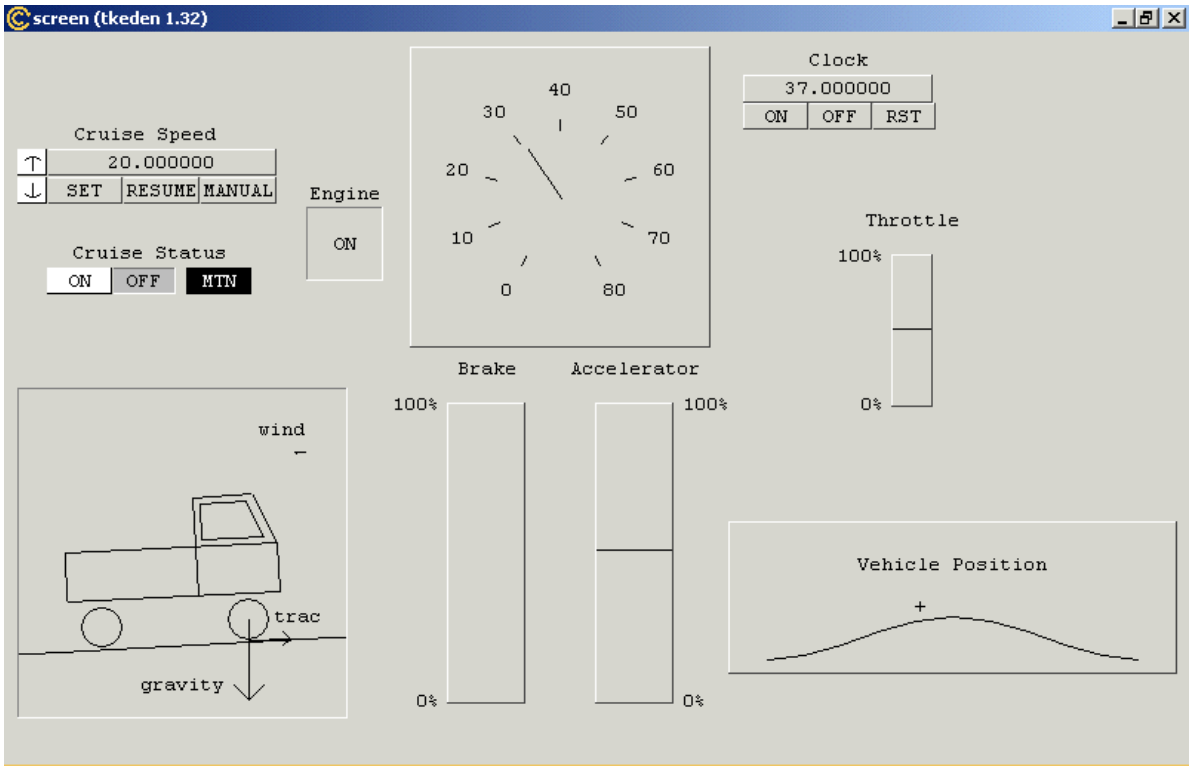

Figure 4 – Eden VCCS Model

Redefinitions can be made to a definitive script to change the model. The user interface is an intrinsic part of the model and cannot be easily separated; it is not possible to make a change to the data contained in the model without an associated visual effect.

Elements of the model that are easy to change in the VCCS EM Model can be divided into two categories:

1. Visualisation Redefinitions
    a. The speedometer (minimum and maximum values, adding a 'stop', allowing the speedometer to handle negative speeds, etc.)

b. The car visualisation (size of car wheels, length of car, car shape)

2. Car Model
    a. The performance of the car (engine power, aerodynamic efficiency)
    b. The environment (hill surface and shape)

The Empirical Modelling group use the VCCS model for teaching tkeden and have built up a number of exercises for students to work through. These exercises involve making redefinitions to alter the model. For example, the speedometer can react differently to negative speed inputs, such as registering them as positive values or by displaying zero. The speedometer reading is defined as:

```
A = minA + (maxA - minA) * curSpeed div topSpeed
```

The notation used is DoNaLD which uses the = operator (rather than the more common `is` operator) to define a dependency. The angle A of the speedometer needle is determined by the value of the variable curSpeed. To make the speedometer display zero for negative speeds we can make the following redefinition:

```
A = if curSpeed < 5 then minA + 0.1 else minA + (maxA - minA) *
          curSpeed div topSpeed
```

This will make any speed below 5 mph register as about zero (actually a couple of mph above zero) on the speedometer gauge.

Analysing the script in detail

Modelling time by adding a clock counter

The VCCS model has a counter that is incremented after each set of redefinitions is carried out. On each update of the counter a number of definitions are re-evaluated and a number of procedures are triggered. This counter is made possible by an Eden feature ‘todo’ that lets us make a delayed redefinition. This counter acts as a system clock or program counter that ‘ticks’ regularly.

A delayed redefinition is a definition that is made after the current recalculation of definitions and dependencies has been completed. This needs to be done so that the calculation can be completed and the recalculation engine does not enter a continuous unbreakable cycle of recalculation. Since the clock triggers many redefinitions and procedural actions modifying the clock within the current recalculation cycle would be equivalent to creating a cyclic definition that cannot be evaluated. The term ‘delayed’ merely refers to fact that the definition is performed after the current recalculation, it should not be taken that a script using ‘todo’ is a time-driven model.

```
proc clocking : iClock
{
      nextClock++;
      todo("iClock = nextClock;");
}
```

This section of code will create a redefinition that is to be made after all the current redefinitions and updates have been done. This redefinition will itself trigger a new round of updates. This way the clock divides time up into rounds of redefinitions and updates where each round is to be regarded as atomic.

Responding to User Input

tkeden allows procedural code to be attached to DoNaLD and Scout items. This code is executed when the user performs certain actions (such as clicking a button or typing in a value). This code typically makes redefinitions to the script to alter the behaviour of the system.

For example the following redefinition is performed if the user clicks the left mouse button on the button named 'ignition_mouse_1'. The engineStts is a boolean variable that has the values esOff, esOn. The definition of engineStts is redefined to be the complement of its existing value when the button is clicked.

```
proc chgEngineStts : ignition_mouse_1
{
      if (ignition_mouse_1[2] == 4) //Check for left mouse button
      {
            engineStts = (engineStts == esOn) ? esOff : esOn;
      }
}
```

The VCCS model files:

The VCCS model is created from the following tkeden, DoNaLD and Scout files:

Run.e	Run.e simply references cruise.s. This is done to aid modellers to find the first file of the model.
cruise.s	Evaluates the definitions in main.e and defines the screen layout with the Scout and DoNaLD notations.
main.e	main.e includes references to the other Eden files and makes the following definitions: Defines the initial state of the VCCS simulator (sets clock to 0) Defines a function for setting the clock running (stops when 1000 steps have been completed) A reporting mechanism (setting `report=true` will produce a textual report to the console on every clock tick) Defines 3 integrations using the macro in 'wmb.macros.e'. These are used to calculate actSpeed (the actual speed of the vehicle).
control_panel.e	Defines the initial state of the VCCS dashboard and Engine Management Unit (EMU) and defines the state changes that are carried out when the controls are used by the vehicle driver.
driver.e	Gets actions and input from the driver (in this model done through asking a human)
engine.e	Defines maxEngineTorque and a definition that sets engineTorque depending on maxEngineTorque (maxium torque of the engine), engineStts (whether the engine is running or not) and throttlePos (position of the throttle).
enum.e	Defines enumerated data types (and string representations of them) that are used within the model. Types defined are boolean (true/false), pushBtn (whether a button is up/down), cruiseStts (On,Maintain,Off), engineStts (On,Off), throttleStts

	(Off,Man,Auto).
environment.e	Asks the human user of the model for the current hill gradient
io.e	Defines the following input/output routines: get_enum output current value of 'enumVal' as a string and read its value as a string (value unchanged if only <CR> entered) and return its enumerated value get_real output current value of 'realVal' and read its new value as a string (value unchanged if only <CR> entered) and return its value as a real valid_real_string true if string is real number
speed_transducer.e	Calculates the speed of the vehicle from the rotational speed of the wheels. Relies on actSpeed which is the vehicle's real speed and introduces the possibility of errors due to the measurement technique.
throttle_manager.e	Defines the behaviour of the cruise control controller that governs the use of the throttle when the vehicle is asked to maintain a set speed.
utils.e	Defines some utility functions
vehicle_dynamics.e	Defines the properties of the vehicle and sets up definitions to update the position of the vehicle.
macros.e	Defines a macro that generates an integration function with respect to time. Includes a supporting procedure 'macro' that aids this macro generation.

Important Definitions

Definition File	Defintion
control_panel.e	cruiseSpeed is mph_to_mps(cruiseSpeed_mph); braking is (brakePos != 0.0);
engine.e	engineTorque is (engineStts == esOn) ? maxEngineTorque * throttlePos : 0.0 ;

main.e	speed is round(mps_to_mph(measSpeed),2); /* [mph] */
	update is 100; /* clock ticks per report on speedo and throttlePos */
	update2 is 100; /* clock ticks per report on vechicle status */
	gradient is dist_to_grad(distance);
	gradient_in_rad is gradient / 200.0 * pi;
	integ_wrt_time("actSpeed", "distance", "0.0");
	integ_wrt_time("actSpeed * cos(gradient_in_rad)", "HDisplacement", "0.0");
	integ_wrt_time("actSpeed * sin(gradient_in_rad)", "VDisplacement", "0.0");

speed_transducer.e	pulseRate is int(actSpeed * wheelPuls / wheelCirc);
	countVal is int(pulseRate * countPeriod) % maxCountVal;
	measSpeed is (countVal * wheelCirc) / (countPeriod * wheelPuls);
throttle_manager.e	speedErr is (cruiseSpeed - measSpeed) / cruiseSpeed;
	deltaAutoThrottle is ((GainK * speedErr) - autoThrottle) / TimeK;
	throttlePos is (throttleStts == tsOff) ? 0.0 : (throttleStts == tsMan) ? accelPos : limit(autoThrottle, accelPos, 1.0);
vehicle_dynamics.e	windF is ((actSpeed >= 0) ? 1 : -1) * windK * pow(actSpeed, 2.0);
	rollF is rollK * actSpeed;
	gravF is gravK * mass * sin(gradient * pi / 200);
	brakF is brakK * brakePos * actSpeed;
	tracF is forcK * engineTorque;
	sticF is sticK * sgn(actSpeed) * bound(actSpeed, -0.01, 0.01);
	accel is (tracF - brakF - gravF - rollF - windF - sticF) / mass;
	integ_wrt_time(“accel”,”actSpeed”,0.0)

It is surprising that relatively few mathematical definitions are made in the VCCS model, the majority of the code is concerned with defining the screen display and with the procedural code associated with input events from the user. We will see later that this proportion is similar to an object-oriented model of the same domain.

Significance of Definitions

In the VCCS model engineTorque is defined in engine.e with the following definition:

```
engineTorque is (engineStts == esOn) ? maxEngineTorque * throttlePos
: 0.0;
```

Under Eden this could be written as a piece of triggered procedural code (called a ‘triggered action’) in the following way:

```
proc engineTorque_update : engineStts, maxEngineTorque, throttlePos
{
      engineTorque = (engineStts == esOn) ? maxEngineTorque *
      throttlePos : 0.0;
}
```

On first looking at this there appears to be little difference between the two ways of specifying the change. However, there is a significant difference in the meta-information that can be obtained about this change; the procedural version of the definition has the opportunity to produce side effects that the original definition does not allow. Although no side effects are present in the code above the implementation language cannot decide what a side effect is. Please note that since esOn is a constant nothing can depend on it! By this I mean that esOn cannot trigger any updates or redefinitions since it will never change.

By specifying this relationship as a definition conforming to a definitive notation we are providing information about this relationship in a form that we can derive meta-information

and thus dependencies from. For example it may be the case that throttlePos is no longer required in the calculation, by changing the definition we do not have to worry about the dependencies since they are worked out for us. Compared to a traditional procedural language where we cannot specify the dependencies explicitly but rather have to propagate them by method calls then we gain even more than the triggered code above since we do not have to modify or even check existing code after making a redefinition.

The analysis technique used before the VCCS model was constructed differentiates between dependency and agency. In the above example we have converted an agent action to a dependency relationship. We gain programming efficiency by using a dependency but we lose the concept of agency involved in this state change.

Java Object-oriented Version

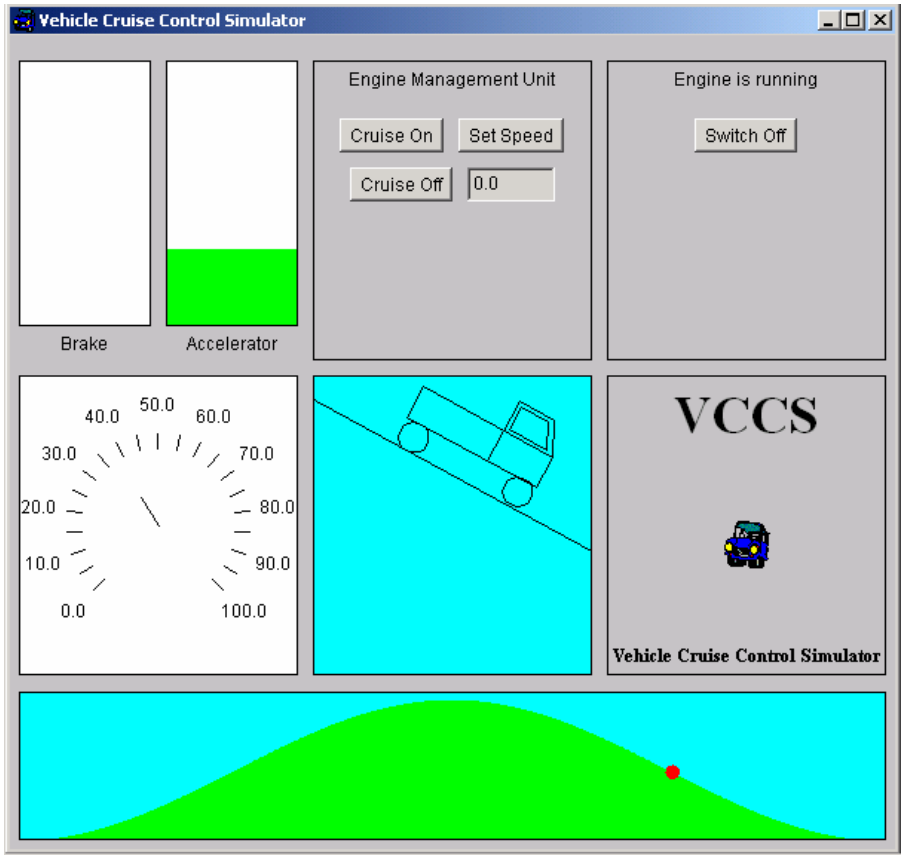

Figure 5 – Java VCCS Model

The VCCS model was re-developed from scratch by the current author using UML and Java following object-oriented principles. Use cases and an object model were developed and then the Java development followed this model. The existing mathematical model was used but no structure of the EM Model was intentionally carried over.

Designing the information model

The first step in designing an OO application is to design an information model (IM). An information model is a system of software objects that models a given domain. Because our problem domain is an automotive vehicle, our IM consists of objects such as accelerator, brake, engine management unit, etc. An IM's client is a person or software system that understands the problem domain. The IM interface (public class methods) should be designed specifically for someone knowledgeable in that domain. In other words, a car designer with

the cruise control IM, a Java compiler, and knowledge of programming should be able to construct a car cruise control simulation straight from the source code.

The information model was designed to follow the physical implementation of the cruise control system as closely as possible. This includes representations of the physical objects present within the system such as car, accelerator, brake, speedometer, etc.

Traditionally in OO models the user interface is separated from the object model so that the OO development consists of an Information Model (IM) and a User Interface (UI) with an interface between them. This pattern of OO development has not been followed here. Since the aim was to create an OO model that followed the physical model as closely as possible the user interface is implemented within the physical objects. For example the speedometer user interface is implemented within the speedometer object. This is not to say that the IM cannot be used without a user interface, it is just a design choice concerning the degree of separation between the IM and the UI.

Two utility classes were developed – Utility.java to provide mathematical functions (similar to util.e in the original model), and ImageCanvas.java used to aid visual presentation. Exercises were written to make the same model changes as the EM exercises do.

Limitations of the Java Implementation

The arrows representing the forces acting on the vehicle have not yet been added to the CarDiagram class. The cruise control logic is very basic and is not as sophisticated as the original Empirical Model. The SpeedTransducer is not as detailed as the original model; it assumes a perfect reading of the vehicle's speed while the original model allows for errors caused by the speed measurement mechanism.

Exercises with the OO model

The same exercises that are provided as exercises for Empirical Modellers to try with the VCCS model can be applied to the OO version:

The Java speedometer takes the absolute value of the speed of the car; this means it draws the correct speed whether or not the car is travelling forwards or backwards. This behaviour is easily altered in the Speedometer.drawIndicator() method. We looked at a DoNaLD redefinition that altered the behaviour of the speedometer in a similar way earlier in this chapter.

```
double angle = indicator_value * factor + offset;
int xpos = (int) (radius * (Math.cos(angle)));
int ypos = (int) (radius * (Math.sin(angle)));
g.drawLine(centerx, centery, xpos + centerx, ypos + centery);
```

We can add the following line of code before the first line of the above section to make our speedometer register 0 at speeds below 5 mph:

```
if (indicator_value < 5) indicator_value = 0;
```

Making the length of the speedometer needle depend on the speed of the car is also easily done from within the Speedometer.drawIndicator() method.

Adding a dependency between the position of the wheels and the length of the vehicle is accomplished within the CarDiagram constructor. Altering the appearance of the car is done in the same method and altering the performance of the car is done by changing the attributes of the Car object.

VCCS Object Model

(excluding user interface attributes and methods):

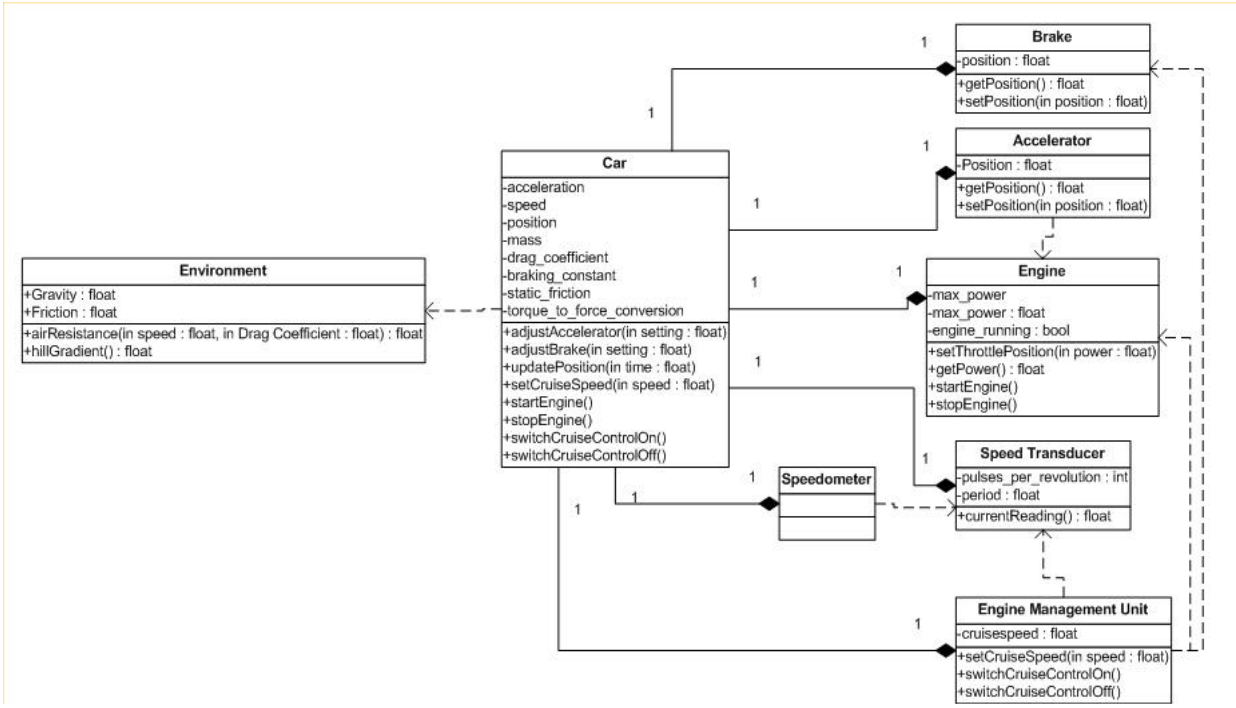

Figure 6 – VCCS Static Object Model

Comparison of the two modelling approaches

The EM Model is controlled by a single clock that regulates the animation, the physical modelling of the car and the cruise control logic. In the OO model this has been replaced by two threads that act as asynchronous clocks, one for the animation and physical modelling and a separate one for the cruise control logic (how the throttle position is modified in response to the inputs to the engine management unit). It was felt that this provided a closer approximation to the arrangement within the physical world than a single thread regulating everything.

The tkeden tool provides the DoNaLD notation that greatly simplifies the specification and production of line drawings used throughout this model. This line drawing capability has had to be translated to the relatively limited line drawing capability of Java's Graphics object. The two models both rely heavily on the user interface, the tkeden model has a lot of definitions to define the display and the Java OO version has a lot of code to draw the screen and handle events from the windowing system.

By going through the existing exercises that have been developed for the VCCS as learning exercises for tkeden it appears that everything that can be done easily with the EM Model can also be done easily with the OO model. The particular features of tkeden that make coding this model easy are the dependency maintenance and the line drawing notation that is particular to the animation used.

In the object-oriented model of this system much more has needed to be done by the programmer; all method calls used to propagate changes to the state have needed to be worked out beforehand and ordered. The version developed in tkeden has not required this same level of work. A good example of this is when the brake is being modelled.

Cruise Cancellation Implementation

When the brake is pressed the cruise control should cancel its operation of maintaining the speed. This is achieved in the cruise_control.e file.

The definition is made: `braking is (brakePos != 0.0);`

The following procedural code is executed when this definition is re-evaluated:

```
proc agent_control_7a : braking /* brake */
{
	if ((braking) && (cruiseStts == csMaintain))
	{
		cruiseStts = csOff;
	}
}
```

This corresponds to an agent monitoring the brake and performing an action if the brake is pressed.

Under the OO version the brake has been modelled as an object that when pressed notifies any dependants by method calls. In order to receive these method calls every object that depends on knowing whether the brake has been pressed needs to register with the brake object. In addition a level of abstraction is required in the Java implementation to keep the object type consistent (objects that are to be notified of the brake being pressed must implement the BrakeListener interface which defines the brakePressed() method).

In this model the EMU (Engine Management Unit) defines itself as a brake listener. When we construct the car we link the brake to the EMU with the following statement:

```
brake.addBrakeListener(emu);
```

When the brake position is changed the method fireEvent is called that calls the brakePressed method of every BrakeListener that is registered to this brake component.

```
void fireEvent()
{
	Enumeration e = listeners.elements();
	while (e.hasMoreElements())
		((BrakeListener) e.nextElement()).brakePressed();
}
```

In this way the dependencies within an OO program can be made dynamic and flexible but the propagation of change is still managed by the programmer who has to register the dependent object with the event firing component.

Summary

The tkeden version of the VCCS simulation does not fully utilise the features of the tkeden language from a programming point of view, many procedural actions that are modelled as agency could be incorporated more efficiently as dependency. The simulation relies on a ticking clock to trigger events and although a clock is required for animation it makes the model seem very procedural with the clock seemingly acting as a traditional program counter.

It is clear that the character of the VCCS EM Model and the Draughts EM Model are very different. The VCCS has the feel of a traditional procedural program with some relationships specified as definitions. The draughts model is almost purely definitive and has the feel of a spreadsheet with its many similar (practically duplicated) definitions.

This illustrates two possible uses of dependency maintenance quite well, one as an aid to traditional programming and one as the basis for a different kind of modelling (Empirical Modelling). It is easy to become confused between the two and there is little difference between the VCCS tkeden model and traditional programming. Although tkeden fully supports purely definitive modelling it also has the capability for procedural action and this can be abused to create models such as the VCCS which use tkeden's declarative style for little more than defining a user interface and then performing most of the computation in a procedural manner.

The physical cruise control system upon which the model was based does not have much dependency within it, the relationships between components are well-defined and the Empirical Modelling analysis performed during the construction of this model was thorough and succeeded in describing all the dependencies that were present. However the final script seems to be more procedural than definitive and many of the advantages of a model described by explicit dependencies are not present here. This procedural flavour is due to the nature of the cruise control system and that the system has limited dependencies within it.

Although the VCCS is an awkward fit for a fully explicit dependency maintenance system it is a good demonstration of how a dependency maintainer can be put to good use within a procedural program. The next chapter explores how two different dependency maintenance strategies can be employed within object-oriented development to ease the burden on the programmer.

Chapter 3 – Dependency as a Programming Tool

Introduction

In any OO system it is widely accepted as good practice to keep the number of interconnections and dependencies between objects to a minimum. The more connections there are then the harder it is to make changes to part of a system without affecting other parts of it. This is a key problem when programming user interfaces. A degree of separation is desired so that the information model can be programmed independently. However users expect a fully functional user interface to manipulate this model. This often leads to a large number of interconnections between the user interface objects and the information model. There are a number of strategies to reduce the complexity of programming these interconnections.

There are two main kinds of dependency within an object-oriented program:

> **Functional dependencies** refer to procedures or sections of code that are called from other sections of code.
>
> **Variable dependencies** occur when the attributes contained within objects affect the behaviour of the code.

Dependencies also have scope, they may be embedded within a function or method, they may be local to a particular object or they may have remote dependencies that extend across modular boundaries.

This chapter explores two strategies based on dependency to keep these connections manageable. The first is Michael Perry's Automated Dependency Tracker [Perry01] and the second is Richard Cartwright's (Java Maintainer API) JaM2 Dependency Maintainer [Car01]. We shall first consider existing object-oriented strategies to solve the problem of appending a user interface to an information model.

A detailed look at Michael Perry's automatic dependency tracker and how it can be used in the development of user interfaces will take place. Finally, we will work through Michael Perry's example (a network modelling tool called Nebula) to convert it to a set of data types and operators and then provide an implementation of the example that maintains its dependencies with JaM2.

User Interface Design Patterns

Useful object-oriented designs are now often generalised into 'Design Patterns' [GoF95]. These are descriptions of common object-oriented design structures that provide specific properties to an OO design if used correctly. A number of design patterns can be applied to linking a separate information model and user interface together each with their own advantages and disadvantages:

Model/View/Controller (MVC)

Used by most RAD (Rapid Application Development) tools and IDE's (Integrated Development Environment), examples include Java's Observable classes and Swing/AWT event handlers which use the Publisher/Subscriber (also known as Observer or Dependents) pattern. Key features of this pattern are:

- Onus is on components to notify others of updates and events.

- Objects called ‘Listeners’ subscribe to the component to receive notification of these updates/events.
- In Java this is done by creating a Listener interface (or inheriting from java.util.Observable) that specifies which methods are available for notification.

The MVC pattern (also called IM/Widgets/Frame) uses a Controller between the IM and the UI. The Controller creates and updates the UI components, and events generated by the user are passed to the Controller so it can update the UI and the IM.

Although the Controller and the UI are separated from the IM they are linked very closely to each other (often the Controller is a UI component itself such as a java.awt.Frame).

MVC provides the following properties:

- Separate UI from the Information Model
- Output – Read from IM and display
- Input – Put data back into IM

Every subscriber that wants to be notified of events has to know about the publishers and has to register every interest they have with them. This could lead to a very complicated system even if only a few objects are involved.

Event Notifier

Using the event notifier pattern moves the process of subscribers registering for events to a central service. They register with an event service to receive particular types of events and are not concerned about where the events originated from. This makes it easier for a subscriber that wishes to receive the same type of event from several publishers since they only need to register once with the event service to receive these events. More publishers can be added/removed easily without existing subscribers having to sign up to every new publisher.

If a link to a specific publisher is required then this can be achieved by adding a filter to the event service and identifying the source through an event attribute. Characteristic features of this pattern are:

- Subscription is based on event type rather than a list of publishers; subscribers receive notification of all event types.
- Subscribing to an event type automatically subscribes to all its subtypes.
- Events can be filtered if actions are required on specific event sources.
- Multiple publishers and subscribers for each event.
- Subscribers and publishers may be transient; the burden of adding/removing publishers is removed.
- Reduces compile time (static) type safety, the Event notifier does not preserve event sources when events pass through it. This generalisation is needed (events are generalised to types of events).
- The event service presents a bottleneck and a single point of failure since all events that are fired have to pass through the event notifier.

Using Statecharts

Statecharts [Horr99] are a way of designing reliable object-oriented user interfaces. They use the user interface-control-model architecture in a similar way to Model/View/Controller architecture but instead of each user interface component receiving the user's events they are passed to a small number of control objects that have been designed with statecharts. By using statecharts the controller logic (to the input of any user generated events) can be verified.

Michael Perry's OO Dependency Tracker

Michael Perry developed his OO dependency tracker in response to the chore of developing update methods to keep user-interfaces up-to-date. In the past, manual notification mechanisms were the only way of updating dependent data in an object-oriented application. Patterns such as Observer, Publish-Subscribe, Document-View, and Model-View-Controller were certainly important as graphical user interfaces (GUIs) matured. But now that interactive applications have grown more sophisticated, expressive, and complex, the chore of notification now burdens productivity.

With a manual dependency mechanism, a program addition usually requires the developer to revisit prior dependencies to keep the user interface synchronised with the model. However, when dependency is tracked automatically, the program discovers new dependencies on its own at runtime, with no programmer intervention. Michael Perry uses automatic dependency tracking to simplify application development and maintenance. When a program discovers dependencies and updates attributes automatically the developer can focus on application logic instead of propagating updates. The program can also respond to changes in object dependency as we update the program, meaning that as the problem changes, the solution changes. We don't have to go back and revisit old dependencies as we add new features.

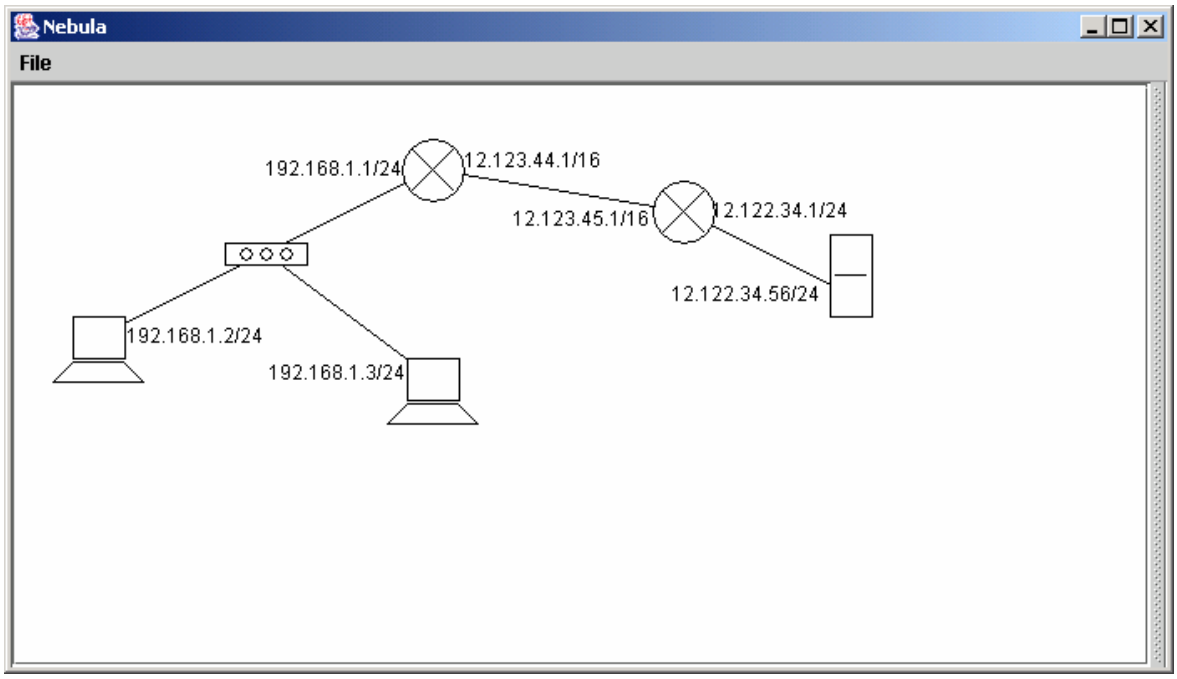

Figure 7 – Nebula Network Modeller

Automatic Dependency Tracking

Nebula is a TCP/IP network modeller, it allows a user to add components of a network and to analyse the routing patterns throughout the model.

Nebula contains an Information Model (IM) that is separated from the GUI, it was straight-forward to extract the IM and run tests to build networks on this without encountering any problems. The IM is firmly based in the problem domain and a Java programmer would be able to construct a network simulation using this IM alone.

Using Dependency

Using dependency to maintain consistency between the IM and UI has a number of advantages. There is no need to manually program or route messages from the UI or even for an IM object to register for notification of changes.

Update methods can be provided for dependent attributes that can be separated out and reused no matter what the trigger is (they need not be targeted toward UI events). Dependent attributes represent observations of the current system state and events represent requests to change to this system state, the mechanism by which these requests are carried out is defined in the update method.

The IM is constructed by declaring and instantiating Java objects; for the purposes of automatic dependency maintenance the objects that are created have their attributes divided into two categories (Definitive and Dynamic):

> Definitive state does not change throughout the life of the object (it is usually supplied at object instantiation)
>
> Dynamic state can change and it is this state that we wish to keep track of using the dependencies.

It is unfortunate that Michael Perry uses the word ‘definitive’ to mean almost the opposite meaning to that of ‘definitive’ in the EM sense. Note that although the tracker is ‘automatic’ we have to declare our Dynamic state.

For each dynamic attribute a ‘Sentry’ is created (A sentry is an instance of the tracker class Dynamic).

> Before a dynamic attribute is read the sentry’s ‘onGet()’ method is called to ensure that the attribute is up to date.
>
> After a dynamic attribute is changed the ‘onSet()’ method is called to propagate the updates to other objects.

The Sentry performs these actions:

> When its onGet() method is called it constructs a new inner class called ‘me’ and sets this as the current Active Dependent. This dependent is added to an internal list of Dependents.
>
> When a sentry’s onSet() method is called it goes through its list of Dependents and invalidates them all.
>
> This way the sentry builds up a list of all the attributes that are declared as dynamic and also affect the current Active Dependent when the IM is first built up.

The Dependent works in the following way:

> When an object requires an attribute that is dependent on another object’s attribute, it not only declares the attribute but also an inner class that is a Dependent and contains a method called ‘update’ to re-calculate the dependent variable.

There now follows a list of the main classes involved in the automated dependency tracker, it is not immediately clear from this list how the dependency tracker works but the example after it clarifies how it should be used and the information it requires.

A full explanation would go into Java language details such as inner classes, interface inheritance and method overriding and Michael Perry provides a full explanation of how the internals of the tracker works.

The Precedent Class

The precedent class within the dependency tracker contains the following attributes and methods:

Attributes

static IDependent activeDependent – this is the current Active Dependent (the attribute that is being evaluated)

List of dependents – A list of the dependents that affect the value current Active Dependent

Methods

makeDependentsOutOfDate() – removes all dependents from this precedent

recordDependent() – Establishes a two-way link between the activeDependent and this precedent

The Dependent Class inherits from the Precedent Class

Attributes

List of Precedents -

Status – The current status of this dependent either Out of Date, Up to date or Updating

Update – Instance of IUpdate that gives the rule for updating this Dependent

Methods

cleanup() – Makes all the precedents of this dependent forget about this dependent

onGet() – Push this dependent onto the active dependent stack, calls the update method to update the attribute and pop the dependent off the stack

addPrecedent() – Add a precedent to this dependent's list of precedents

The Dynamic Class inherits from the Precedent Class

Methods

onSet() – simply calls makeDependentsOutOfDate() in Precedent

onGet() – simply calls recordDependent() in Precedent

Registering a Dynamic Attribute

Each class that contains dynamic state declares the attribute and constructs a sentry for it:

```
int kind = WORKSTATION;             //The Data
Dynamic dynKind = new Dynamic();    //The Sentry
```

Before the attribute kind is set the method dynKind.onSet() is called and before the attribute kind is read the method dynKind.onGet() is called. In order to ensure this access and mutator methods are used:

```
public int getKind()
{
      dynKind.onGet();
      return kind;
}
public void setKind( int kind )
{
      dynKind.onSet();
      kind = kind;
}
```

Updating a Dynamic Attribute

The dynKind.onSet() method removes all the dependents of this sentry.

The dynKind.onGet() method checks the current dependent sentry that is being updated and records a dependency between it and this Dynamic.

Defining the Dependent Sentries

Sentries are declared by creating a new instance of the Dependent class and registering an IUpdate object with it.

To write a method that uses the dynamic attribute (in this case ‘kind’) we create an instance of the Dependent class and also an instance of the IUpdate interface that performs the calculations we want.

```
Dependent m_depKind = new Dependent( new IUpdate()
{
      public void onUpdate()
      {
            System.out.println(“Kind has been updated.”);
      }
});
```

A Simple Example

```
public class DependencyTest
{
static Value value1 = new Value();
static Value value2 = new Value();


public static void main(String[] args)
{
      //Register a recalculation method (i.e. A dependency definition)
      //This is equivalent to "value2.value is value1.value;"
      Dependent depValue = new Dependent( new IUpdate()
```

```
        {
            public void onUpdate()
            {
                      value2.setValue(value1.getValue());
            }
        });

        value1.setValue(10);
        System.out.println("Current Value 1:"+value1.getValue());
        System.out.println("Current Value 2:"+value2.getValue());

        System.out.println("Make sure that Value 2 is up to date");
        depValue.onGet();   //Necessary Recalculations are performed now
        System.out.println("Current Value 1:"+value1.getValue());
        System.out.println("Current Value 2:"+value2.getValue());
}
}

public class Value
{
        public int getValue()
        {
                dynValue.onGet();
                return value;
        }

        public void setValue(int newvalue)
        {
                dynValue.onSet();
                value = newvalue;
        }

    // Dynamic data
    int value = 0;  //The Data
    Dynamic dynValue = new Dynamic();   //The Sentry
}
```

The output from this is as follows:

```
Current Value 1:10
Current Value 2:0
Make sure that Value 2 is up to date
Current Value 1:10
Current Value 2:10
```

Nested Dependencies

Michael Perry's dependency tracker does not support nested dependency automatically. Nested dependencies have to be explicitly declared as shown in the following example:

```
public class DependencyTest
{
	static Value value1 = new Value();
	static Value value2 = new Value();
	static Value value3 = new Value();

	public static void main(String[] args)
	{
		//Register a recalculation method (i.e. A dependency definition)
		//This is equivalent to "value1.value is 2 * value2.value;"
		//This one must be declared final so that depValue2 can access it
		final Dependent depValue1 = new Dependent( new IUpdate()
		{
		    public void onUpdate()
		    {
				value2.setValue(2 * value1.getValue());
		    }
		});

		Dependent depValue2 = new Dependent( new IUpdate()
		{
		    public void onUpdate()
		    {
				//Demonstrates how we can do nested dependencies
				depValue1.onGet();
				value3.setValue(2 * value2.getValue());
		    }
		});

		value1.setValue(10);
		System.out.println("Current Value 1 :"+value1.getValue());
		System.out.println("Current Value 2 :"+value2.getValue());
		System.out.println("Current Value 3 :"+value3.getValue());

		//Telling DT that we want to make sure that Value 3 is updated
		depValue2.onGet();

		System.out.println("Current Value 1 :"+value1.getValue());
		System.out.println("Current Value 2 :"+value2.getValue());
		System.out.println("Current Value 3 :"+value3.getValue());
	}
```

```
}
```

In this example we have to explicitly call depValue1.onGet() from within depValue2.onGet(). We cannot rely on the update being performed automatically by the value2.getValue() call within depValue2.onGet().

Dependency Form

Before we considered the automatic dependency tracker we described two distinct forms of dependency within programming, implicit (as used within OO) and explicit (as used within definitive programming). However by adding his dependency tracker to an OO information model Michael Perry has extended the notion of dependency within OO to beyond that of simple implicit dependency. There now exists more information about the attribute updates and method calls than there would be within a standard OO model and this removes some of the burden from the programmer of developing and maintaining these messages.

JaM2

JaM2 Overview

The Java Maintainer Machine API version 1 (JaM) was developed as a demonstration of a generic object-oriented dependency maintenance system illustrating a new parallel method for definition update, the DM model. Developed in 1996 using Java 1.0, the original JaM is documented as part of Richard Cartwright's PhD thesis [Car98].

JaM2 was later developed from JaM by Richard Cartwright. JaM2 provides a generic dependency maintenance system for object-oriented programmers to utilise in their applications. In addition to this, JaM2 provides facilities for organising the information in a dependency structure into virtual directories and manages user and group permissions to access and modify the data and dependencies.

JaM2 is written in Java 1.1 and provides a wide range of standard data types and operations over those types. It also provides a parser that accepts Eden-like script to make definitions and re-definitions. In this way JaM2 is similar to the original Eden but without the ability to add procedural code that is executed when triggered by re-definitions. There is no equivalent to tkeden's DoNaLD or Scout in JaM2 (it would be an interesting and useful project to implement these in Java with JaM2 as the dependency maintainer).

Explicit dependency representation introduces a formalisation to the maintenance of dependency in computer applications. When using JaM2, it is necessary to formally register well-defined data types and operators with JaM2. JaM2 performs a number of checks on these types and operators to build well-defined and concurrent dependency structures. All data values have to be considered as values of atomic data types and tricks that can be played with pointers or object references are not guaranteed to work with JaM2. JaM2 must be informed about a change or a dependency otherwise it cannot keep the state of values consistent.

More meta-data about data is required for JaM2 to work correctly than is required for directly implemented implicit dependency maintenance.

JaM2 offers the following benefits:

- Explicit representation of dependencies between objects of any well-defined underlying algebra;
- Registration of data types and operators describing underlying algebras on-the-fly;
- Description of data values by parameter sets, allowing the parsing and value description functions of JaM2 to be separated;
- Organisation of definitions describing dependencies into directories;
- User, group and other permissions similar to a multi-user operating system;
- Automatic update of dependencies between objects when state is changed, considering a set of definitions simultaneously and executed efficiently in parallel where facilities are available.

Representing Dependency in the Java Maintainer Machine API

By combining the OO implementation of the VCCS, JaM2 (the Java Maintainer Machine API) and the original Eden definitions for the VCCS script we can envisage a new model

being produced that has an OO structure but uses Eden's dependency principles to change state.

For example:

The Accelerator object can be declared as a JaM2 data type (by simply implementing the JaM2.Type interface. This exposes the Accelerator's attributes (a position primitive) to the JaM2 script.

The definition for creating the Accelerator would become (a default value of 0.0 is assumed for the position attribute):

```
A is createAccelerator();
```

Since the Accelerator's attribute 'position' is modified by the user of the system the accelerator must notify JaM2 that this attribute has changed with some procedural code when the user interface event is received.

A definition called accelPosition would be created:

```
accelPosition = 0.0;
```

When the user changes the accelerator position a redefinition would be made to the script to update this.

The engine would be defined in terms of this variable:

```
E is createEngine(accelPosition,emuThrottlePos,…)
```

and the engine would in turn make a redefinition to alter `engineOutput`.

Eden requires a similar notification to make redefinitions from user interface events:

```
proc chgAccelPos : accPedal_mouse
{
      if (accPedal_mouse[2]==4)
      {
            accelPos = accPedal_mouse[5] / 100.0;
      }
}
```

This extra notification code needs to be included because neither system has a way of linking the definition to the user interface toolkit directly. An exercise would be to extend one of the Java toolkits (like the AWT or Swing) to include these notification events automatically.

Used in this simple way redefinitions look like an alternative way of making method calls where instead of remembering to make the appropriate calls you must update the appropriate variables that the methods will require.

To completely translate the Java OO model into a JaM2 model the references between objects must be replaced by definitions in a JaM2 script. This means that the object constructors must be replaced by operators that declare the object as a data type. Object attributes must be represented by definitions in the script also. This leaves several stages to our modelling conversion process:

The first stage is to separate the attributes and methods of the objects.

The second part is to redefine the objects as JaM2 data types by implementing the JaM2.Type interface.

Thirdly a JaM2 script must be constructed that builds the VCCS model based on these data types and operators.

Finally a visualisation must be added in order to complete the system.

Visualisation

The original Eden model relies on a definitive notation for line drawing (DoNaLD) that fits with the definitive programming paradigm. However Java does not support such a notation and so it made sense to maintain the existing visual representation of the OO model by keeping the user interface construction done with objects. The objects (now JaM2 data types) exist in a JaM2 script. References to these objects can be obtained from JaM2 and methods called on them. By adding these references to the standard Java user interface Containers a visualisation will be created. Some calls to repaint the user interface components may be required in the data type's doJaMAction() method.

Namespace, Scope and Accessibility

Eden and JaM2 both use a hierarchical namespace where any definition can reference any other definition. This could be regarded as programming with global variables since all variable and definitions have the same scope (albeit with a naming convention to construct, though not enforce a hierarchy). Traditionally, use of global variables with structured programming is considered poor practice and is the opposite of good object-oriented design that attempts to limit the scope of all variables and methods.

The Eden tool does not place any scope restrictions on variables in order to keep the model 'open', i.e. allowing redefinitions to change any part of the system instead of being restricted to a pre-determined (or evolved) structure.

Object-oriented design attempts to provide a level of safety to objects. By making attributes private and only allowing access through get and mutator methods the internal representation of the object's data can be separated from the view the object presents to the outside world. The object's methods mediate between the data and the interface and ensure that the internal structure of the object is kept consistent. Since it is impossible to modify the data directly the object's writer can be sure that their code cannot be modified by other objects or processes in the model.

Converting the Nebula Example to JaM2

Michael Perry's Nebula example contains its own automatic dependency tracker. This goes some way to adding automatic dependency maintenance to an object-oriented program but has significant limitations when dealing with nested dependencies. By using the dependency tracker Michael Perry has moved the Nebula example away from purely explicit dependencies and has been able to introduce some automation to the dependency maintenance. In this section we look at what needs to be done to make the Nebula example fully implicit so that all involved dependencies are maintained for us.

In order to do this we will create a JaM2 model of a simplified version of Nebula. We will encounter the following differences between JaM2 and Michael Perry's tracker:

JaM2 is script based; every dependency that is to be defined must be added to a script within JaM2.

The dependencies are discovered as the objects in Nebula are created and not declared beforehand as in JaM2.

Object construction happens independently from creating definitions, in JaM2 an object is created by adding it to the script (an object cannot exist before it is added to the script).

All the objects in a system that are related by dependency must be known to JaM2.

JaM2 can maintain dependencies across multiple threads and even processors, the Nebula dependency maintainer assumes everything is in a single thread. Dependency updates are not atomic and no synchronization mechanism has been used in their deployment. For example, in the period before onSet() is called the system is in an undefined state where the value has been updated but no dependents have been told about it.

A problem with converting this example to use JaM2 is that JaM2 must know about every attribute update – it needs to be informed of any object-oriented side effects being involved. This requires a reference to the JaM2 script being available to every object that generates side effects so that we can call JaM2's forceEvaluation() method.

JaM2 is written to maintain dependency using operators over objects that have been written to conform to specific interfaces. In order to use JaM2 to maintain these dependencies we need to convert Nebula's IM into a definitive script.

One way of doing this is to represent the Nebula IM as objects and operations over those objects. For example the following Nebula objects can be viewed as JaM2 data types:

Nebula Object	Example JaM2 Declaration
Hub	`H is hub;`
Computer	`C is Computer;`
Router	`R is Router;`
Network Interface Card	`NIC is createNIC(C);`
Cable	`CableA is connect(NIC,H);`

Nebula's objects have attributes that are not related to other objects, for example a Cable object can have one of two states: either a PATCH cable or a CROSSOVER cable. If we wanted to be able to change the state of the cable object (and be aware of dependency) we would have to create a definition for each attribute and reference them in the construction as follows:

```
CableStateA is PATCH;
CableA is connect(NIC,H,CableStateA);
```

We could then change CableStateA to change the attribute of CableA. This technique requires all attributes to be specified in the declaration.

Nebula IM Simplified

A simplified version of the Nebula example has been developed with JaM2. The Nebula information model was reduced in complexity and then written as a combination of JaM2 data types and operators.

The simplification removes all elements of routing and reduces the IM to a single kind of device. Here is a UML class diagram of the simplified IM:

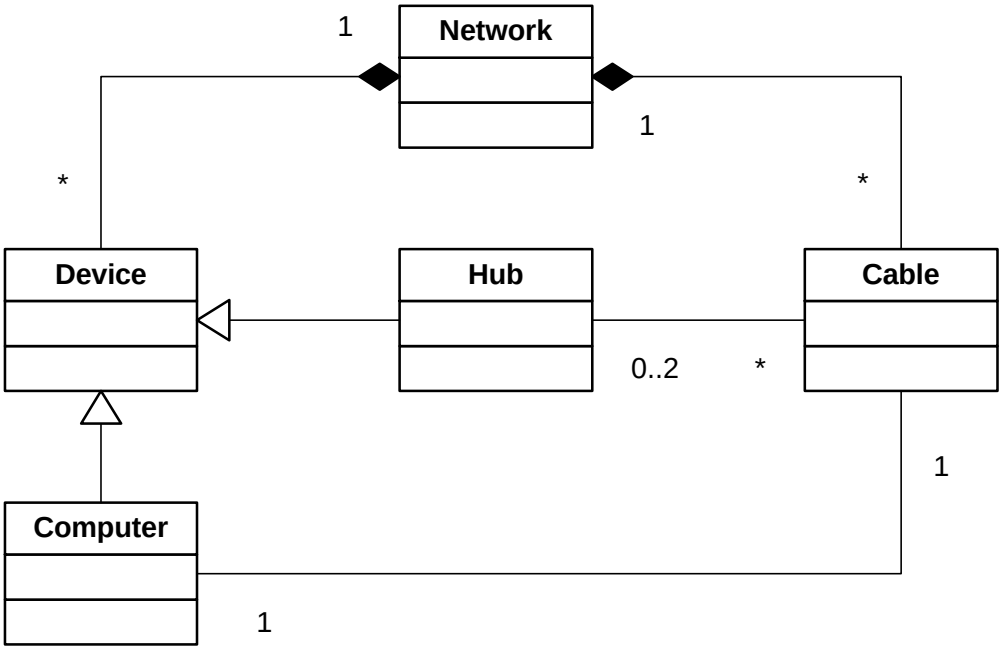

Figure 8 - Simplified Nebula Class Model

Here are the data types involved:

Data Type	Explanation
Device	An abstract data type that both Computer and Hub inherit from.
Computer	A computer on the network that can accept a single network cable. A computer has a name attribute.
Cable	A network cable that connects two devices together. A cable can either be a PATCH cable (when connecting a computer to a hub) or a CROSSOVER cable (when connecting a computer to another computer).
Hub	A hub on the network that can accept an arbitrary number of cables plugged into it.

Here are the operations over these data types (there are separate operations since JaM2 does not support overriding of operations:

Operation	Example Use
join (Computer c1, Computer c2)	C is join(C1, C2);
jointohub (Computer c1, Hub h)	C is jointohub(C1, H);
joinhubs (Hub h1, Hub h2)	C is joinhubs(H1, H2);

Here is an example script that is used to construct a simple network model:

```
comp1 is computer { name “Raspberry” };
comp2 is computer { name “Strawberry” };
```

```
h is hub { };
ca1 is jointohub(comp1,h);
ca2 is jointohub(comp2,h);
update;
```

A diagram of the network model declared above:

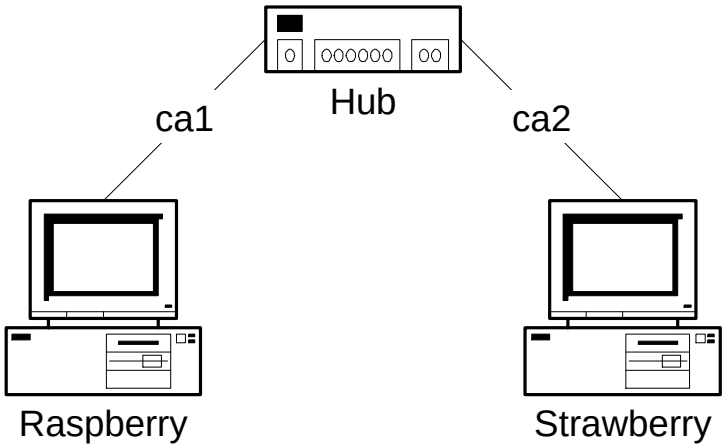

Figure 9 – Example Network Model

Adding the UI

The IM that has been constructed so far has no knowledge of the UI we are to use with it. We need to add a new layer that adds UI specific information to this model.

We add the following data types:

Data Type	Explanation
Location	An (x,y) pair of coordinates
gDevice	A device on the network that combines Device information with Location information
gCable	A graphical representation of a cable on the network

and the following operations:

Operation	Example Use
drawDevice(Device d, Location l)	./gui/gComp1 is drawDevice(./comp1,./gui/comp1_location);
drawCable(Cable c)	./gui/gCable1 is drawCable(./ca1);

drawDevice selects the appropriate drawing method to use from the type of d. This operation can be performed on every device (thereby adding location information to every network device).

drawCable draws the interconnection between two devices.

When the system must display the network IM procedural code goes through the list of dependencies. It finds all the gDevice definitions and gCable definitions and draws them on a Java Graphic object.

When working with dependencies between objects in this way (i.e. using JaM2) it is possible to represent information in two different ways: Either as attributes of objects or as operations over objects. For example two devices linked to each other can either be represented by a joining operation over the two devices or by attributes in each device describing what it is connected to.

To draw a Device we need to create a gDevice that references a Device and a Location. When writing this gDevice we have two options: Either have attributes of a gDevice that are the Location and Device or leave gDevice without attributes and have the definition (in terms of the expression) store this information. Should we regard the dependency expressions as merely a mechanism to describe the dependencies or should they store information in their own right?

We would most likely want to make the glyph that is used to draw the devices part of the dependency as well but for simplicity this is not done here (the location data type demonstrates the same principle as would be shown by doing this).

Screenshots

The following screenshot shows the JaM2 version of Nebula with a simple network displayed. The bottom section of the window shows a definition of one of the icons used to draw a computer.

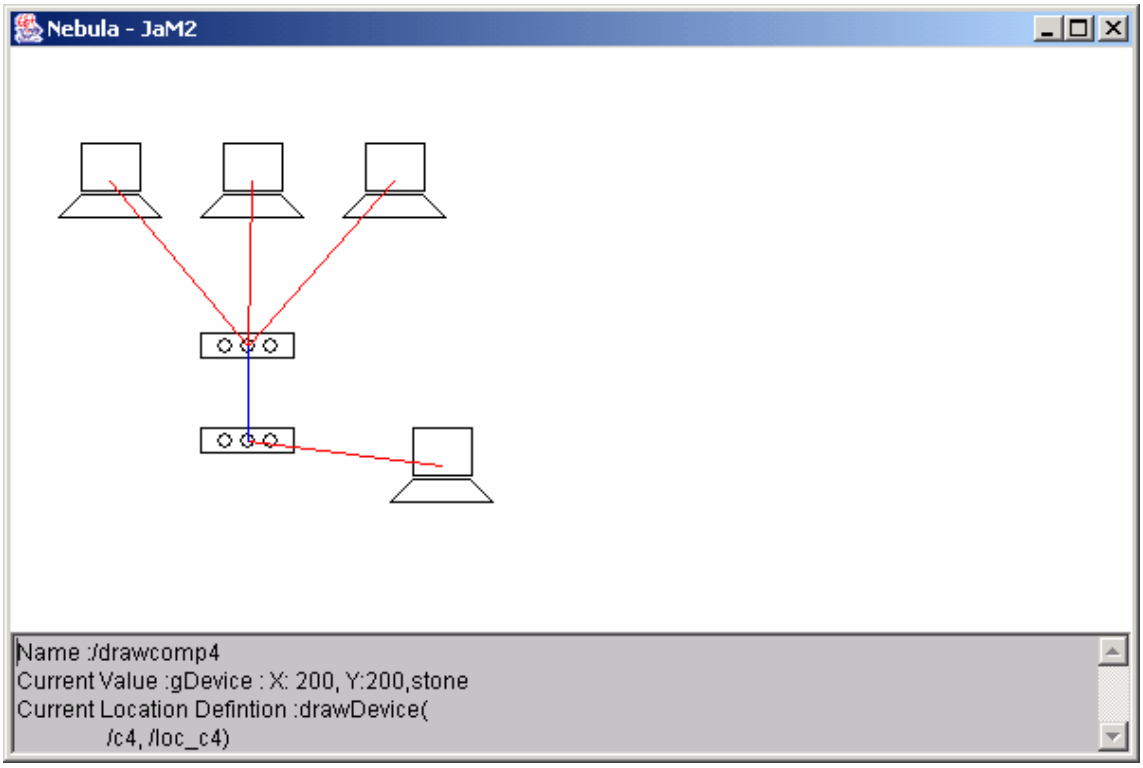

Figure 10 – JaM2 Nebula

Updating Definitions

By changing the definitions we can change the displayed information. By altering the location information of a gDevice the display will draw it in a different location; by changing the definition of a cable we can reposition it between different devices.

```
cable10 is join(c10,c11);
drawcable10 is drawCable(/cable10,/drawcomp_c10,/drawcomp_c11);
```

```
t12 is addToCanvas(/drawcable10);
update;
```

Here a new cable is added between the computers named c10 and c11. Then a graphical representation of this cable is added to the script. This representation is passed the reference to the cable we are drawing (to allow access to the properties of the cable) and references to the two icons that represent the computers c10 and c11 (to find the locations of the computers on the screen). A side effect of the addToCanvas operator actually draws the line representing the cable on the screen.

Applicability to Rapid Application Development (RAD) tools

RAD tools typically use the model/view/controller design pattern to excess. The system being developed is designed as a user interface with functionality 'hanging off' rather than as an IM with a user interface representation [Cooper95].

By developing an alternative way to couple the Information Model with the user interface either through a dependency tracker or through a definitive script there is the feasibility of a RAD tool being developed that exploits the kind of dependency relationships that these techniques provide. It is foreseeable that a GUI design tool could create the user interface components and link them to the information model not through method calls as used in traditional event handling but by constructing a dependency relationship between the attributes of the IM and the values displayed on the screen.

The dependency tracker or maintainer could then be responsible for maintaining the consistency between the on screen representation and the data stored in the IM.

Relationship to Empirical Modelling

Empirical Modelling is an approach to constructing an artefact (an example of which could be a software system) that relies on continuous refinement of a computer based model with respect to a real-world referent. At all times the model is open to modification. Development of an artefact is regarded as a side effect to the main task of model construction.

The computer-based modelling is supported by tools that provide dependency maintenance with a definitive script. These tools are interpreter based and combine definitions with procedural code to allow the interactive nature of model development.

Multiple definitive notations can be used within a single model to provide the most appropriate way of expressing the desired definitions.

Existing user interface tools used in the EM group at the University of Warwick are based around definitive notations for line drawing (DoNaLD) and window manipulation (Scout).

The problem of manipulating attributes in a script is evident in these notations. In Donald a circle can be defined the following way:

```
%donald
circle circ
circ = circle({50,50},25)
```

and its attributes are set as follows:

```
%Eden
A_circ is "color=black,fill=solid";
```

This will place a filled, black circle centred at (50,50) with a radius of 25 on the screen. To change these attributes the above string has to be redefined in its entirety:

Defining `A_circ is “color=red”;` will remove the fill property and the circle will be left hollow. A way round this is to perform the following definitions:

```
colour = "red";
fill = "solid";
```

and add the following definition to perform the updates:

```
A_circ is "color="//colour//",fill="//fill;
```

Changing either `colour` or `fill` will update the circle as originally intended.

This technique is the same technique used before with JaM2, in order to be able to change the attributes of an object all the attributes of the object must be specified as separate definitions and then when the object is constructed the definition must include references to these attributes.

Although `circ` is not an object as it is defined in the OO paradigm there are parallels between this problem of managing attributes and the problem of supporting dependency between object attributes in JaM2.

Reducing Debugging Effort

The amount of dependency within an object-oriented program is not just a problem when initially designing and implementing the software, it is also a problem when maintaining the software. It is still the case that the majority of the cost associated with software development is spent in the maintenance phase of the software to fix bugs and provide new features. Both of these require a developer to have a thorough understanding of the existing system before commencing changes.

A program with high levels of dependency is not only difficult to design but it is also difficult to fully understand, if we can reduce the need for this understanding then the time and effort required to understand a program will fall as well. Such complexity of understanding or ‘Cognitive Complexity’ can be measured [Hen96] by measuring the functional and variable dependencies that must be traced throughout the code by a programmer who wishes to understand the software. When the developer is reading the program code in order to understand it they only need to trace the dependencies to the extent necessary for them to have a general idea about the ‘control plan’ of a method, or to have a ‘variable plan’ of the way a particular attribute is used. Once the developer decides on a modification (or to trace a bug) then they must trace the changes through the dependencies until they are convinced that they have discovered all the state changes and side effects.

An automated dependency tracker would prove invaluable for these kinds of debugging purposes; at any point the developer could display a list of dependents on any particular attribute. This would reduce this tracing effort considerably. Definitive scripts are tested and analysed by continuous interaction with the modeller while they are being constructed and do not require the same level of tracing effort to discover state changes.

Summary

This chapter has focused on using two different dependency tools to solve a particular problem, that of linking the user interface of a software application to the internal data values within the computer (the Information Model). Michael Perry’s Automated Dependency Tracker was developed through a need to create a very flexible user interface that would have normally required keeping track of a large number of dependencies through generating method calls. Instead his technique centralises the problem of maintaining consistency

between the UI and the IM as well as automatically discovering the dependencies that cause this relationship.

Cartwright's JaM2 dependency maintainer machine was used to redevelop a simple version of Michael Perry's Nebula Network designer tool by constructing a definitive notation to represent the IM of a network. An extension to the notation was then used to describe the UI from this IM and an implementation was developed to illustrate the technique. Initial conclusions from this exercise are that in order to add full implicit dependency maintenance to an object-oriented program then the information model must be reworked into a well-defined notation involving operators and data types. Models developed in this notation are then able to provide enough information about their dependencies so that a dependency maintainer can perform all the core state change (although side effects of operators may be required to interface to existing traditional code and class libraries).

Chapter 4 - Formalising Definitive Models

Introduction

The last two chapters have shown dependency maintenance used in two very different ways, one as a tool used within programming and one as the basis of a different kind of modelling, Modelling with Definitive Scripts.

Definitive scripts form a paradigm whereby dependency is considered a corner-stone of the modelling process, relationships are represented as definitions that are evaluated and kept up to date by a dependency maintainer. The definitions provide additional meta-information that is not present in a procedural program that allows the dependency maintainer to work out the most efficient way of evaluating the expressions. This meta-information also provides additional validation of the script by verifying that no cyclic dependencies are introduced.

This chapter studies the ADM (Abstract Definitive Machine). The ADM provides the simplest implementation of a useful environment for working with definitive scripts. Other environments are far more sophisticated and support a wider range of data types but even the simple ADM can be used to demonstrate the power of definitive scripts. The ADM is not merely a dependency maintainer but has a wider purpose within EM as a framework to interpret and animate an LSD account; however, this chapter continues to concentrate on dependency.

The ADM, a model for definitive programming

The ADM was introduced in Mike Slade's thesis 'Definitive Parallel Programming' [Slade90] in an attempt to show that definitions can be used to program in a highly parallel, interference-free manner. M Slade showed how to translate an LSD account to a family of ADM programs.

LSD (a name, not an acronym) is a simple notation that is used when making an initial analysis of a domain. It is used to describe the classification of observables with agents. The features can be grouped according to whether the agent 'owns' them, called **state** variables, whether the agent can respond to them, called **oracles**, or whether the agent can change them, called **handles**. An LSD account can be viewed as a description of a domain and as an informal specification for a program.

The ADM was designed by Meurig Beynon, Mike Slade and Edward Yung in 1988. It was developed in collaboration with Mark Norris at British Telecom as part of an investigation into animation tools based on LSD, an agent-oriented specification language.

The ADM can be viewed as a machine that supports purely explicit dependency maintenance. It can represent state only through definitions and it can only change that state through redefinitions. It was designed to provide an implementation of a framework that enforced principled access to state through definitions. Models developed using the ADM are free from the types of abuse, such as an over-reliance on procedures, that can easily occur in eden models.

An ADM program is a set of entities that are instantiated upon execution. Each entity comprises a set of definitions and a set of actions. Each action is guarded by expressions that evaluate to a Boolean value. Each action can either be the redefinition of a variable or the creation of an entity instance or an entity's deletion.

Existing ADM implementations (am, adm)

Mike Slade produced an implementation of the ADM called `am` as part of his thesis [Slade90]. It was a command line interpreter for parsing and executing ADM scripts. `am` was written in C using the `lex` and `yacc` parser generator tools. It only supports the integer data type and this limits its functionality somewhat. It has been used in conjunction with DoNaLD to produce on screen line drawings.

Other implementations (such as `a2e` developed by Y. P. Yung) have attempted to translate from ADM script to `eden` script. Although these have been functionally more successful (due to the wider range of data types within `eden`) the code produced bears little resemblance to the original ADM script.

As an example here is an ADM script that swaps two definitions a and b:

```
entity swap()
{
definition
        a = 1,
        b = 2
action
        true print("a is ", a, " and b is ", b) -> a = |b|; b = |a|
}

swap()
```

The translation into `eden` produces (the supporting `eden` code is omitted, the full version can be found in Appendix 6):

```
proc swap {
      if (!Silent) writeln("instantiating swap()");
      autocalc=0;
      execute("
            a = 1;
            b = 2;
      ");
      execute("
            proc swap_action_1 : sysClock {
            if (sysClock == -1) return;
                  if (TRUE) {
                        writeln(\"a is \",a,\" and b is \",b);
                        todo(\"a = \"//STR(b)//\"; b =
            \"//STR(a)//\"; \");
                  }
            }
      ");
      autocalc=1;
}

swap();
```

The ADM supports the ability to remove entities, but removing definitions from a definitive script can be a very tricky process and the translators do not support this functionality properly.

Despite these drawbacks models have been developed successfully using the a2e translator, such models include a game of cricket, five-a-side football and a simulated telephone handset.

Comparison of AM and ADM

Since we are talking about two implementations of the same application, I would like to start with the following distinction. `am` here refers to the original interpreter for the Abstract Definitive Machine described in Mike Slade's thesis and `adm` refers to Simon Yung's translator to `eden`. Although this comparison is not directly related to the main theme of this chapter it provides two useful benchmark's for evaluating the JaM2 version that is developed later. The following shows the relative strength and weaknesses for the two implementations.

`am`

1. Parallel actions
2. Conflict detection
3. More reliable
4. Has semi-evaluation operator (| |)
5. Limited algebra (only has integer and boolean data types)
6. Restricted interface to definitive notations

`adm`

1. Accepts most (if not all) `eden` definitions
2. Fully supported by other definitive notations
3. More advanced parameter specification
4. Sequential execution
5. No conflict detection
6. Poor handling of entity deletion

The principal aim of translating ADM script to `eden` was to enhance the underlying algebra of Abstract Definitive Machine and to provide it with a wider range of primitive data types. With `adm` it is possible to program more sophisticated simulations but it requires a translation step that adds a lot of syntactic complexity to the original ADM script. The ADM is designed for guards and actions to be evaluated in parallel, however, because `eden` is basically sequential in its execution, the true power of Abstract Definitive Machine cannot be utilised with the current implementation of the ADM to `eden` translator.

Building an ADM implementation with JaM2

We have already used JaM2 to implement a simplified version of Michael Perry's Nebula network modelling tool using a specially designed definitive notation. JaM2 as a general purpose dependency maintainer API provides us with a good base for developing future modelling tools. This section will look at simulating ADM models within JaM2 with the eventual aim of a developing a new implementation of the ADM that is both extensible (with new data types) and accurate (including genuine parallel evaluation). JaM2 supports a number of useful features that make an ADM implementation built with JaM2 particularly appropriate:

True parallel evaluation of redefinitions, JaM2 not only uses multiple threads to perform expression evaluation but it can also distribute those threads across multiple processors (although not yet over multiple machines).

A wide range of data types and the ability to add new data types easily (JaM2 comes with an extensive base library of common data types but as we saw in Chapter 3 it readily supports user-defined types).

JaM2 is implemented with Java. There are many advantages to using a widely accepted modern language such as Java as the implementation language. A few are listed here:

- Dynamic class loading and reflection, not all classes need to be registered at design-time, some compilation is performed at run-time. This is essential for achieving the flexibility that JaM2 requires.
- Portability, Java eases development over multiple platforms by developing for a virtual machine instead of a specific platform.
- A large standard class library including many built-in abstract data types and user interface components available for use.
- It is an 'active' language with a large amount of industrial backing and many enthusiasts so it is easy to obtain development advice and help.
- Java supports object-oriented development.
- Java has Garbage Collection capabilities which reduce the possibility of memory leaks.
- Java development tools (compiler, debugger, etc.) are freely available.
- Undergraduate courses now teach Java as the primary language, in order to encourage more EM based projects it is advantageous to base them in Java so as not to put off students.

JaM2 ADM Examples

This section shows how a couple of ADM models can be simulated in the JaM2 environment by constructing some extra framework and by converting the syntax of the original ADM scripts. As examples two scripts from Mike Slade's original thesis were taken, the GCD script is used to illustrate the basic principles and the Jugs script is a more complex example that shows not only the power of ADM scripts but also some of the drawbacks to the current framework.

Greatest Common Divisor

Here is the original ADM script for calculating the greatest common divisor of two integers:

```
entity one(_first)
{
      definition
            var1 = _first, change1 = (var1 > var2)
      action
            change1 -> var1 = | var1 – var2 |,
            !change1 && !change2 print("gcd is ",var1) -> delete
      one(_first)
}
entity two(_second)
{
      definition
            var2 = _second, change2 = (var2 > var1)
      action
            change2 -> var2 = | var2 – var1 |,
            !change1 && !change2 -> delete two(_second)
}
```

The ADM GCD program is executed by instantiating an instance of each entity and then starting the ADM:

```
one(24)
two(16)
start
```

The output will be:

```
gcd is 8
```

Although this script does not illustrate the power of definitive scripts it gives a simple example of the structure of entities and the entries within them.

Analysis of the GCD script

In order to simulate this ADM script in JaM2 an analysis of each section of the script becomes necessary to both discover how this script works and demonstrate the main sections of a generic ADM script.

The script consists of two 'entities' denoted by the `entity` keyword and surrounding braces. Immediately following the entity keyword is a parameter that not only provides a method for

input to the entity but also names an instance of the entity (there are problems with this approach which are discussed later on).

Within each entity there are two sections, one marked ‘definition’ and the other marked ‘action’. The definition sections are straightforward to understand and each is simply a section of definitive script where definitions are made, variables declared, relationships set up, etc. The definition section is purely definitive, there are no procedural actions, and operations are not allowed to have side effects.

The syntax of the ADM is different to the syntax employed by Eden and JaM2. When creating a definition the ‘`a = b`’ is used to mean what Eden and JaM2 normally mean by the ‘`a is b`’ statement, that is, a definition that should be maintained is declared. The Eden and JaM2 syntax of ‘`a = b`’ meaning ‘assigning the current value’ of a definition is represented by the expression ‘`a = |b|`’.

The action section of each entity is divided into parts, the left-hand side expressions are guards and the right hand side are definitions or procedural actions. Within the ADM all guards are evaluated in parallel and if the evaluation results in true the corresponding action is performed (also in parallel). Once the actions have been performed, evaluation of the guards restarts; evaluation is repeated until all guards are false.

The procedural actions that are allowed within the action section are limited to making redefinitions to the definitions defined in the definition section of the entity (or other entities) or to instantiate/delete entities.

JaM2 GCD Conversion

Here we describe how each section of the GCD ADM script was converted to JaM2 script.

Entities - It was clear that entities represented a separation between different sets of definitions. JaM2 provided the concept of a hierarchy of directories and this feature was used to create a separate directory for each entity. Thus all the definitions for the first entity were placed in a directory called ‘one’. This step provides a partial solution to the namespace problems with the ADM that will be discussed later.

Definition - The definition section is easy to convert, since it is a section of definitive script it is simply translated into JaM2 syntax without any modification and so the definitions for each entity were simply placed in the entities directory.

Guards – the guards are expressions that evaluate to a Boolean result. Apart from this restriction they are just the same as other definitions. By creating a definition (that has been named `guardNNN`) that stores the result of evaluating the guards then the conditions are fulfilled since JaM2 evaluates definitions in parallel.

Actions – Actions are executed on each cycle when their guards evaluate to true. There was no direct equivalent within JaM2 so an operator called ‘action’ was defined. This operator has as its first parameter the guard that should be monitored and as its second parameter the redefinition that should be performed when the guard evaluates to true. This action operator is extremely unusual since it performs only side effects and makes no direct changes to the script. It adds the definitions that should be performed to an ‘action list’ which are then performed before the start of the next guard evaluation cycle.

Here is the code for the Action Operator:

```
/**
 * This Operator checks the value of the guard and if true adds the
definition
```

```
 * into the script definition set. No useful information is returned as a
JaM2
 * datatype
 */
public class ActionOperator implements Operator
{
    //The actions since the last update
    static Vector actions = new Vector();

    /**
     * Clears the Action List, called after each update
     */
    public static void clearActionList()
    {
        actions.clear();
    }

    /**
     * Returns the current Action List
     *
     * @return   Vector of String with each String a script redefinition
     */
    public static Vector getActionList()
    {
        return actions;
    }

    /**
     * This operator has the side effect of adding the guarded action
command to
     * our action list if the guard evalutes to true, the operator itself
only
     * returns the boolean value of the guard condition.
     */
    public Type f(ExprList parameters, Type toReturn)
    {
        //There are only ever two arguments (a guard and a definition)
        JaMBoolean guard = (JaMBoolean) parameters.getValueAt(0);

        //If the guard evaluates to true then add the action to our action
list
        if (guard.value)
            actions.addElement(((JaMString)
parameters.getValueAt(1)).value);

        return toReturn;
    }
}
```

In order to complete the GCD ADM simulation an application had to be built to provide an environment that would allow interaction with the user to control each execution cycle. This application presents the action list that is to be executed to the user for inspection. Each execution cycle is triggered by the user clicking ‘Perform Single Update’ or continuous updates can be performed by clicking ‘Start Update Cycle’. This runs until no actions are specified.

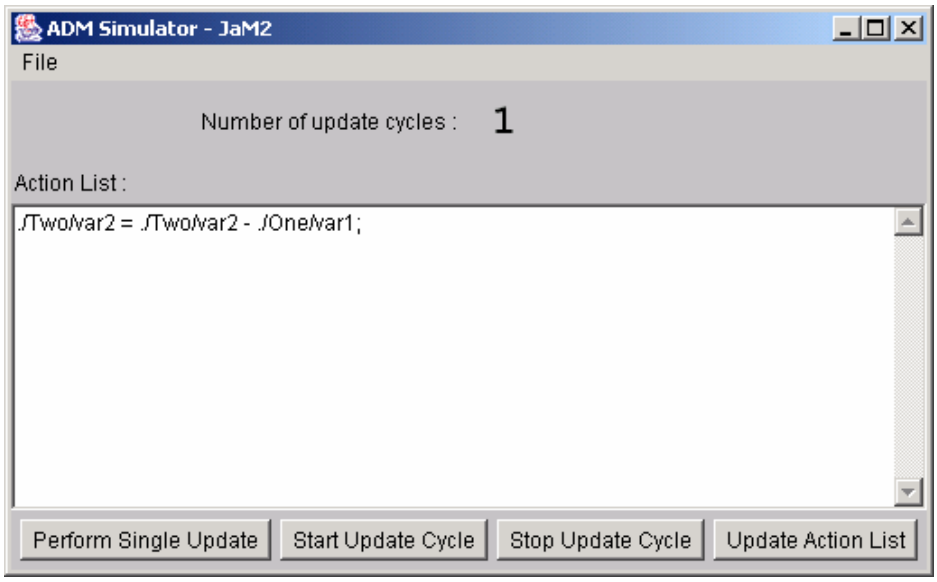

Figure 11 – JaM2 ADM Simulator

Another operator was also added to allow simple console output from the script. The print() operator simply outputs whatever it is passed to the console (as another side effect).

Here is the whole JaM2 simulated ADM GCD script that can be loaded into this environment and executed:

```
mkdir One;
mkdir Two;
./One/var1 = 24;
./Two/var2 = 16;
./One/change1 is greater(./One/var1,./Two/var2);
./Two/change2 is greater(./Two/var2,./One/var1);
./One/guard1 is ./One/change1;
./One/guard2 is
booleanAnd(booleanNot(./One/change1),booleanNot(./Two/change2));
./Two/guard1 is ./Two/change2;
./Two/guard2 is
booleanAnd(booleanNot(./One/change1),booleanNot(./Two/change2));
./One/a1 is action(./One/guard1,"./One/var1 = ./One/var1 -
./Two/var2;");
./One/a2 is action(./One/guard2,"./p1 = print(cat(\"Gcd is :\",
numToString(./One/var1)));");
./Two/a1 is action(./Two/guard1,"./Two/var2 = ./Two/var2 -
./One/var1;");
update;
```

This example has some syntactic awkwardness, there is a lack of infix operators available for creating the Boolean expressions and in order to insert a quote character within a string the character must be ‘escaped’ so \″ means ″ inside a string. Despite these shortcomings the syntax still compares favourably with the output from the `a2e` translator.

Jugs

The Jugs model has been implemented with both the original version of the ADM and also with successive versions of Eden and `tkeden`. It is described in many publications [Slade90] [Yung93] and has been modelled and re-modelled. However it was used in M Slade's thesis to illustrate the process of how an LSD account could be converted to an ADM definitive script.

The Jugs application is an educational application where the student is presented with two jugs, a sink and a tap. They have a number of operations they can perform on these items:

1. Fill Jug A from the tap
2. Fill Jug B from the tap
3. Empty Jug A into the sink
4. Empty Jug B into the sink
5. Pour from one jug into the other, until either the pouring jug is empty or the receiving jug is full

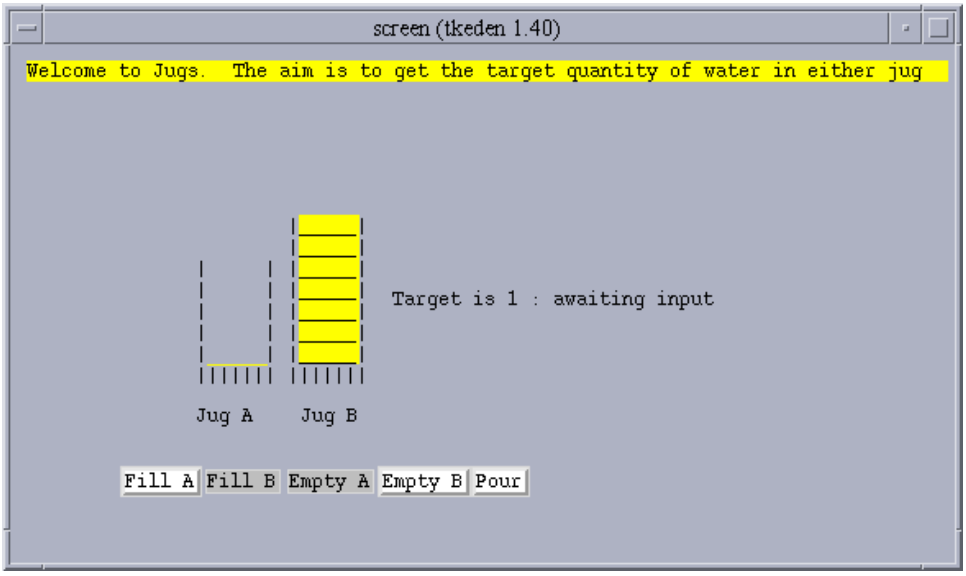

Figure 12 - `tkeden` implementation of Jugs Model

The student explores the number properties presented by this environment. Only contents that are multiples of the greatest common divisor of the two jug capacities can be reached. So if the capacities are co-prime, then any target liquid content within the capacities of the jugs can be made.

The jugs model illustrates the power of definitive scripts far better than the GCD script. The model defines a relationship between two jugs (when pouring one jug into another) without having to modify (or even consider) any code that is performing the pouring action. By having a simple routine that pours a jug and altering the definitions to describe the link between what is being poured from and what is being poured to you can add the ability to pour into different containers without having to change the pouring routine.

To illustrate, the original Jugs ADM script contained the following definition for filling Jug B from the tap:

```
contentB = |contentB| + 1
```

when Jug A was filled from Jug B then the following redefinition was made:

```
contentB = |contentA + contentB| - contentA
```

This added a relationship from Jug B to Jug A (that didn't previously exist) so that as the contents of Jug A increased the contents of Jug B were decreased. When the pouring was over this definition was deleted and the relationship ceased. The vertical bars (|) indicate that the current value of the expression contained between them should be taken and fixed in the definition rather than re-evaluating the expression every time.

This method of describing the relationship between the jugs is far closer to what is experienced in the physical world than a procedural program would allow. In a procedural program we can imagine that we would write a loop that moves water from one jug to another, we would need a different loop to move water from the jug to the sink, we would have to divide the time over which the water was poured into steps. By using a definition we describe the second jug's content in terms of the first and it is then entirely open as to how often the actual values are updated. We are describing what the relationships between our entities are, rather than the procedural view which describes a sequence of steps to maintain that relationship (without specifying exactly what the relationship is). This is clearly a powerful technique.

ADM Jugs Model

The ADM model specifies four entities:

User

> This simulates the user agent for testing purposes; it randomly selects one of the available actions to perform and creates the Init_pour entity with the selected action.

Jugs (Capacity of Jug A, Capacity of Jug B)

> This entity defines the Jugs and a number of definitions that monitor the state of the jugs, these include which jugs are full/empty and the current contents of each jug.

Init_pour (User Selection)

> Init_pour checks to see which actions are possible and if the user has selected a valid option creates a pour entity to perform the actual pouring. Creates definitions to aid the pouring (i.e. if the user wants to pour liquid from Jug A to Jug B a definition is set up to describe this relationship).

Pour (Option)

> Pours liquid in and out of the jugs 1 unit at a time until the pouring action has finished.

Converting Jugs to JaM2

The original model uses a feature of the ADM which is the rand(x) function. This returns a random integer between 0 and x. The User entity uses this to simulate the user choosing options at random. To implement this functionality in the JaM2 version we will have to create the rand(x) operator ourselves.

JaM2 expects operators to be deterministic and optimises its internal structures so that the minimal amount of processing is done when changes are made to a script. Such optimisation includes that JaM2 only re-evaluates a definition if a variable that the definition depends upon changes. When implementing a random operator there is the possibility that the random operator will return the same value as the last call to the random operator. JaM2 will not re-evaluate the dependents of this value in order to maximise efficiency. While this makes sense within a purely definitive world, if we are depending on a side effect of some procedural code

within an operator that references this random value then since the definition will not be re-evaluated then the operator's procedural code is not executed and the side effect not performed. Here follows an example of JaM2 script:

```
a is rand(3);
update;
state a; returns a = 2;
force a;    Required to force JaM2 to re-evaluate this definition
update;
state a; returns a = 1;
```

But if b is defined as follows:

```
b is any_operator(a);
```

then b is only evaluated when a changes (if rand(3) returns the same number on subsequent calls then b is not re-evaluated).

Problems with the ADM

This section describes some problems with the ADM and proposes some possible solutions.

Flat namespace

The ADM specification, along with implementations of Eden, uses a flat namespace. This means that there is no segregation of data items and no scoping of variables. Variable scope is an important programming tool and its usefulness has been proven time and time again. Segregating data items is an important part of structuring programs whether it is done by combining data and operations that are relevant to that data in objects or simply by separating the data away from local variables, loop iterators and other operational variables.

JaM2 adds a hierarchical naming system so that variables can be placed within directories, however although this aids structuring a JaM2 script it does not enforce any access restrictions upon definitions or variables. There is a user level access setting (with password) but this is intended to enforce security privileges and is not provided to be used as a programming construct.

This flat namespace contributes to the structural issues that were discussed in chapter 2 while looking at the Draughts model. Without a way of segregating data the modeller is left to their own discipline to structure the model. While this promotes openness it requires a lot of work by the modeller to maintain the structure and this maintenance gets harder the larger the model becomes.

Attempts to solve the namespace problem

Even though the ADM has the concept of separate entities, there is no way to name these entities. The translator `a2e` attempts to solve this by noticing that ADM entities are referenced not only by the entity but also by the parameters that are passed to the entity, so that by passing an extra, unused parameter (i.e. a name) the entities can be named indirectly. This technique is used in the cricket simulation model.

Using OO principles to solve the namespace problem

The syntax commonly used in OO style languages of the dot (.) notation can be used in the context of the ADM. For example a definition a within an entity b could be referred to with

b.a instead of simply b. This would allow another definition a to exist within a different entity.

So by using this new scheme the GCD script would become:

```
entity one(_first)
{
	definition
		var1 = _first, change1 = (var1 > two.var2)
	action
		change1 -> var1 = | var1 – two.var2 |,
		!change1 && !change2 print("gcd is ",var1) -> delete
	one(_first)
}
entity two(_second)
{
	definition
		var2 = _second, change12 = (var2 > one.var1)
	action
		change2 -> var2 = | var2 – one.var1 |,
		!change1 && !change2 -> delete two(_second)
}
```

Extending OO principles into the ADM

Once the ADM has been given a way to divide up its namespace by introducing an OO naming scheme then the entities used in ADM scripts start to look very similar to classes that have only static variables and static methods. There are a number of possible investigations that can be attempted at this point. One is to try to establish whether or not ADM entities are the same as OO classes (or even objects conforming to the singleton design pattern [GoF95]). This seems unlikely since ADM entities rely on changing definitions to propagate changes instead of sending messages to other entities.

Summary

Assessing the ADM purely as a dependency maintainer is to ignore the main purpose of the ADM, that of providing a way of animating LSD specifications. The ADM has other important concepts aside from dependency (agency, observables) that also contribute towards Empirical Modelling.

This chapter has concentrated on dependency within the ADM and why this adds a lot of flexibility to definitive scripts. It has considered existing implementations of the ADM and simulated ADM scripts using JaM2. It has shown the ADM is still relevant even though we now have more sophisticated dependency maintainers such as `tkeden` and JaM2 and that the fundamental principles (and some of the difficulties) of definitive scripts still remain.

It illustrates, using the Jugs example, why writing definitive scripts is so powerful. The ability to define a relationship between two variables on the fly without having to preconceive the relationship is very flexible. The ADM also shows what seems to be one of the limitations of current definitive script implementations, that of a flat namespace.

Chapter 5 – Modelling and Programming

Introduction

This chapter will look in greater detail at modelling. The close relationship between spreadsheets and definitive scripts will be introduced and the analysis that has been reached about spreadsheet modelling will be extended and applied to modelling with definitive scripts. First we will look at the modelling capability that a spreadsheet can provide to end-users and we will consider the key differences in modelling with a spreadsheet compared to writing a program. The capabilities that a spreadsheet provides will be generalised to definitive modelling to illustrate the advantages that such modelling can have over traditional programming.

We will then consider the ideas of Brian Cantwell Smith about what the essence of programming or modelling is and how this relates to definitive scripts.

Modelling with Spreadsheets

Since their introduction in the late 1970s spreadsheets have been one of the most common and also most successful applications available on the personal computer. Spreadsheets became the most widely used decision support tool in business. Spreadsheet modelling has changed little despite spreadsheet programs becoming vastly most sophisticated than their original implementations.

Spreadsheets can be viewed as a restricted form of definitive modelling where the definitive notation is the spreadsheet formula language operating over a very limited number of data types (numbers, dates, text). Instead of declaring definitions with specific names definitions are stored in cells and like definitive notation implementations the variables (or cells) have two parts to them, a formula or definition and a current value.

The key advantages for users of spreadsheets as found in usability studies (according to [Nardi93]) are:

Spreadsheets are expressive - Spreadsheets permit a wide range of modelling activity and are used from financial and engineering calculations to sophisticated scientific calculations and even basic database functions. End users develop impressively complex applications that combine many variables within a problem domain. The complexity of a spreadsheet is not derived from the formulae that are used (most of them are very simple and straight forward) but from the relationships present in the domain.

Spreadsheets are High Level - Spreadsheets embody a lot of functionality that in a programming language would require programmer effort. Users of spreadsheets do not have to name or declare variables (you could claim that each cell is declared when a new spreadsheet is started). They have a large library of built in functions that are targeted either towards specific domains or for general purpose cell manipulation. For example the SUM function is predefined within a spreadsheet, most programming languages will require the programmer to declare an index variable, a total and then iterate through the variables they wish to sum.

Spreadsheets are Task Specific - Most users use fewer than 10 functions within their spreadsheets, this is because the functions are task specific. Spreadsheets do not have to provide the kind of functions found in programming languages because they are not targeted toward the same general purpose computational problems. They are aimed at a subset of users' modelling needs and fill these needs very well.

Spreadsheets are Accessible - The concepts of a spreadsheet are easy to understand and to learn, and since so few functions are needed to build useful applications the learning curve for new users is gentle. It is possible to become productive with a spreadsheet within only a few hours while a programming language requires far greater knowledge not of only the language syntax but often the vast class library along with many techniques only picked up by experience.

Spreadsheets 'grow' as the user builds them. The user constantly has a 'working' spreadsheet that can be modified, tinkered with, added to, etc. Errors detected when the spreadsheet is being built can be quickly corrected and the dependencies involved are simple mathematical ones. Traditional programming requires planning in order to produce a successful application that is of more than trivial complexity.

The spreadsheet maintains user interest and confidence in what they are doing. It is not uncommon for a new programmer to become bewildered by their own creation and decide to scrap it in favour of starting again from scratch.

Simple Control Constructs - Programming languages present to the user a number of complex conditional constructs, such as if, for, while..wend, do..while, switch, the tertiary operator '?' all of which have unique syntax and "gotchas" if used incorrectly. In contrast spreadsheets have simple conditionals usually just an 'if' statement that accepts a conditional expression and two possible values.

Spreadsheet conditionals do not transfer control from one part of the spreadsheet to another, they can only cause localised change (the cell they are placed in).

Iteration over values it simple in spreadsheets, users simply choose a range of cells. There is no need to specify a variable or an update expression for a loop. When copying a range of cells the cell ranges are updated automatically by the user interface.

Textual Nature - Despite the sophisticated graphical user interfaces of modern spreadsheets that allow charting and graphics to be inserted into a spreadsheet the formula language still exists and users still have to learn its basic syntax. There are 'wizards' and 'short cuts' that can be used to build these formulae by clicking the mouse but they do not attempt to hide the formula from the user once it has been built.

That users can quickly get to grips with such a notation and that programming in any traditional language seems harder suggests that it is not the notational syntax that is difficult about programming languages but rather something inherent about programming itself.

Problems with Spreadsheet Modelling

The majority of spreadsheet use is informal and users do not follow a sound or proven methodology when constructing models. The same freedom and openness that spreadsheets provide to users also provides opportunities for error.

Spreadsheets suffer from several limitations even once the models are built [ISL95]. They are very prone to maintenance mistakes such as deleting cells that are required by other cells, overwriting cell formulae with constant values, adding items without modifying totals, and so on.

The appearance of a spreadsheet is rarely the same as its logical structure. The imposed grid structure can lead users to assumptions about the logical formulae that make up this structure; the visual on screen representation can be deceiving. Cells have no associated interpretation, in a traditional programming language every value has an associated variable name that

should be used to aid interpretation (although often it is not), the cells in a spreadsheet are however effectively nameless and their interpretation can only be inferred from their position.

The spreadsheet is such a general purpose tool it is impossible to write a spreadsheet program that can detect inconsistencies within a specific model. The spreadsheet program is not specific to the application domain, in contrast a financial package will check for consistency within the accounts entered into it, a Computer Aided Engineering (CAE) package will only allow possible physical constructions to be modelled. A spreadsheet allows the user total freedom to make mistakes as well as design decisions.

Just as there are well documented cases of traditional programming causing major projects to fail there are similar stories and lessons to be learnt from spreadsheet modelling mistakes. Cragg and King [CK93] sampled spreadsheet examples from ten organisations that depend on spreadsheets for basic business support functions. They concluded that 25% of them contained logical design errors ranging from trivial cell references to errors in formulae. Moreover this error rate was the same whether the user was a novice or an expert. The users involved had great confidence in their spreadsheet models and Cragg and King concluded that the appearance of the spreadsheet could be misleading; a good looking spreadsheet with a strong visual structure gave a misleading confidence in the quality of the modelling behind it.

Programming and Modelling

At this point this dissertation will turn to the ideas and thinking of Brian Cantwell Smith in order to explore what we mean by programming and to distinguish the facets that make modelling with definitive scripts different from traditional programming.

Brian Cantwell Smith's paper 'One Hundred Billion Lines of C++' [BCS97] states that programming is a symbolic process involving the composition of atomic symbols (such as identifiers) according to syntactic formation rules (such as conditionals, assignment statements, procedure definitions, etc). However once the program is written then the execution of it forms a 'process' that has no symbolic meaning. If the translation from program to process has been correctly carried out then the process should carry out the intended behaviour of the program.

Brian Cantwell Smith divides a computational system into 3 components as illustrated by the following diagram:

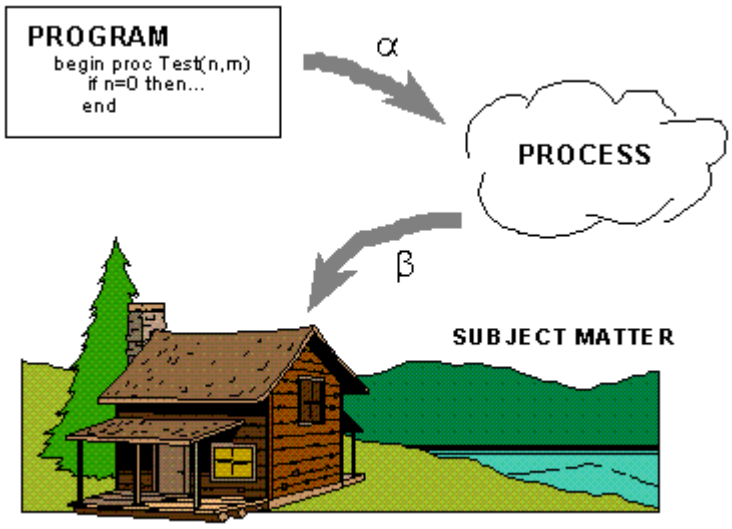

Figure 13 – Program, Process, Subject

Program

A program is a static entity that is usually textual. It can be read, edited, printed, copied and is normally stored as a file on a computer system.

Process

Programs are written to produce behaviour. Programs once written can be executed directly (by an interpreter) to produce behaviour or an intermediate step of compilation can be added. Compilation simply takes one program and produces another program that can be executed. When a program is executed it produces behaviour and this resulting behaviour is what Brian Cantwell Smith calls a 'Process'.

Subject Matter

The (often external) domain the computation is about.

α

The relationship α is the traditional computer science translation of program script to executing code that formal methods and semantics concentrates on.

ß

The relationship ß is how the running artefact relates to the subject matter, its place in the subject matter and the effects it has. This is the relationship that traditional software engineering constantly wrestles with to produce specifications that can be taken back to influence the start of the model, the 'program'.

An analogy to engineering would be that a blueprint for a combustion engine is a symbolic representation of the engine. The Program is analogous to the blueprints and the Process to the physical entity that is the engine. Traditional computing theory would study semantics and would concentrate on the relationship of Program to Process (i.e. the specification and manufacture of the engine) but it would set aside the Process to Subject matter relationship (the relationship between the engine and the rest of the physical world).

Modelling with Definitive Scripts

A lot of the time it seems that modelling with a definitive script is a very similar exercise to programming. Certainly, on the surface the actions of a modeller are similar to that of a programmer. They both use text editors and programming environments to construct a textual script that is either executed or interpreted. They work with the script interactively creating new statements and modifying existing code. However it is the proposal of Empirical Modelling that modelling with a definitive script is a fundamentally different human process with different objectives and different thinking involved [Run02]. The artefact produced as a result embodies a different kind of understanding about the domain (subject matter) than a traditional programmer would have.

Comparisons between Modelling with Definitive Scripts and programming are naturally difficult since they are fundamentally different approaches that are not even aiming at the same target. However, it is perfectly possible to concentrate on specific issues of one approach and apply knowledge gained to the other. Dependency is one of those issues that is vital to Modelling with Definitive Scripts but can be applied usefully to programming in isolation from the wider aspects of Empirical Modelling.

Usually upon becoming a process the program is fixed and cannot be changed by the programmer, however when writing definitive scripts the modeller is free to alter the script while it is in this state of being a 'process' with the definitive notation available. In other

words the process retains the full compositional power of the original programming language (or notation).

Why do definitive scripts allow this openness? There is no distinction made between design time and run time in a definitive environment, the definitive script is always in a runtime 'mode'. Definitive scripts allow changes to be made while the model is 'running'. Few traditional programming languages support the same degree of interaction with a running program. It could be argued that this openness comes with using an interpreted implementation of a language but even interpreted languages mostly leave user interaction down to an 'immediate' debugging environment where only simple variable changes are permitted. Substituting a method, or part of a method so that program behaviour is modified is unusual, while changing a definition that alters behaviour is commonplace within a definitive script.

We have seen that from end-user's interaction with notations such as those used in the spreadsheet formula language that a textual notation is not the obstacle that it initially appears.

Applying Brian Cantwell Smith's model to Empirical Modelling yields an adjusted model where the distinction between program and process is all but removed and the focus of modelling is between this new entity and the subject matter. The relationship ß becomes Empirical Modelling's continuous comparison with the referent where the modeller performs experiments with their model to ensure it matches what they are trying to model. Empirical Modelling analysis studies a referent by looking at the agency involved and attempts to associate theories with circumscribed and reliably occurring patterns of experience from each agent's individual perspective. The Empirical Modelling approach claims that to create a circumscribed closed world it is 'essential to pass through the experimental realm' although this experimentation could be drawn from previous knowledge [Bey99].

Comparing modelling with definitive scripts to spreadsheet modelling yields a number of close similarities but also a number of important differences:

Definitive scripts are open and allow a large amount of design freedom in a similar way to spreadsheets. Constructing a definitive model is similar to building a spreadsheet, changes to definitions can be made without having to constantly stop and start the model (as in a traditional program).

The notations are more domain specific than the general purpose spreadsheet language, a wider range of data types (including definable ones) are available for use. This allows a greater amount of validation and checking to be done. A spreadsheet is only domain specific in that it deals with mathematical constructs, it will not allow formulae that do not obey the rules of arithmetic. A definitive notation can also be based in a specific domain and can provide more constraint so that models created with the notation can be checked for consistency.

Definitions are named instead of being placed in a fixed structure e.g. they are given names like `car_speed` instead of cell reference C3. This avoids the problems that result from misrepresentation within spreadsheets, that of over confidence in the model and misinterpretation of the structure of the model from the visual appearance.

Techniques for changing executing code

Definitive scripts can be modified at any point, the notions of design-time or run-time do not exist. Traditional programming has a strong distinction between writing code and executing programs and program code cannot be modified during execution. In the next section we look at a couple of areas of programming where modifying or even completely changing executing

code occurs in the traditional procedural world. This is not to say that if traditional programming was more flexible then it would be able to match the same level of changes that are possible with definitive scripts.

Self-modifying code

Self-modifying code (SMC) is a specialised programming technique that is normally only used in machine code (or assembly language) programs. The reason for this is that programs that use SMC cannot be compiled since compilation fixes the structure of an executable program. SMC programs must be interpreted and because of this machine code can be viewed as interpreted code run by a hardware interpreter (or machine) instead of a software interpreter; the same argument applies to modern JIT compiled languages such as Java and C#.

SMC was popular in the early days of programming but as machines have become more powerful and high-level languages widespread the practice has fallen out of favour. D Knuth's 'Fundamental Algorithms' uses an ideal computer, MIX, with which Knuth illustrates his concepts. The standard subroutine calling convention of MIX is based on self-modifying instructions.

Nowadays SMC is often regarded as a 'hacker's trick' and is usually employed to obscure code to make it hard to follow. Two common 'uses' of SMC are:

> **Stealth** - Viruses attempt to hide themselves from detection. Many of today's viruses implement some form of SMC so that searching for a static code signature is impossible.
>
> **Anti-Debugging** - Commercial programs that try to keep the source code a secret use SMC to obscure their code preventing de-compilers and reverse engineering tools from following the executable code correctly.

Although SMC is usually only employed in specialist low-level situations there is no real reason why it cannot be applied to high-level interpreted languages when they are running as an interpreted implementation.

SMC is non-interactive; it only allows a program to modify itself in a pre-determined manner. It does not allow an external user to modify the code as the program runs. So, although it exhibits one of the properties of modelling with definitive scripts (that of modifying the code at runtime) it does not present the openness that this modelling provides.

The modern interpreted language

An interpreted program environment executing a program maintains an instruction counter to keep track of where in the program script the execution currently is. Since a program written in an interpreted language is nothing more than text stored in memory (albeit in an efficient tokenized form), commands could be made available to the programmer to manipulate this text and alter the flow of execution. Furthermore the user/developer of the program could temporarily halt the execution of the program, make changes and then continue from where the program left off.

This technique is used within modern interpreted development systems. Within Microsoft Visual Basic it is possible to halt execution, make changes to the source code (within certain restrictions, for example changing a variable declaration prevents continuation) and then restart execution from the point at which it was paused. An 'immediate' window is also provided where the developer can enter any runnable expression or statement to view

variable's values or to make changes to the current state. Developing using these tools has a very similar feel to that of exploring a definitive model.

These techniques are not employed by the user of the system (since it requires the development environment that is not re-distributable according to the terms of the license), but is a powerful development and debugging tool for the programmer. Other programming environments (such as Borland's Delphi) support similar debugging systems (with greater or lesser restrictions on the changes that can be made).

The developer can write their program and make modifications without having to constantly go through test scripts to take the program through a set series of states to test the particular section of code. Instead they can run the program up to the section of code they want to test, pause, modify it and restart execution. The development environment monitors the changes and can detect when the changes have become too great to maintain a consistent state (by violating the imposed restrictions) and prevent the continuation of the program.

In this way an interpreted program can be kept open to modification during execution in a similar way to a definitive script. This does not lower the risk of data inconsistency that can arise from modifying the program (or the procedural actions in a script).

Summary

This chapter has explored the similarity of spreadsheet modelling and modelling with definitive scripts. We have considered that definitive scripts have similar properties to spreadsheets in that they are easy to understand, open to modification and have a shallow learning curve. They also have some drawbacks that are shared with spreadsheets in that there is little enforced structure and spaghetti-style code is easy to produce without discipline. Definitive scripts avoid some of the other problems associated with spreadsheets by dispensing with the grid structure that forms the basis of spreadsheets. Instead of referencing cells, definitions with names are used instead which provide a context for each definition and reduce errors through misunderstanding.

Brian Cantwell Smith's ideas about modelling and programming were considered and we saw how definitive scripts do not have the same gap between the script and the resulting artifact that exists between writing a program and the artifact that is produced when the program is compiled and executed.

We then briefly looked at whether traditional programming languages could provide this same level of openness or whether the fact that they have to be written, compiled and run to produce the artifact prevents this direct manipulation of the artifact. Two techniques for modifying a running program were considered.

Chapter 6 - Conclusions

In this chapter we summarise the material presented in the thesis and appropriate further work that could be undertaken in this area is proposed.

Summary

The first chapter introduced definitive notations and made a distinction between implicit and explicit dependency, it explored spreadsheets from a programming perspective and demonstrated how the implicit dependency of `make` was different from the explicit dependency of a spreadsheet.

Chapter two explored the differences between definitive modelling and object-oriented development using two examples, a draughts game and a vehicle cruise control simulator. The dependencies that arose from using the two techniques were compared. The draughts model illustrated a good model of the environment using a definitive script and the VCCS model showed that with the current EM tools it is easy to fall into the trap of writing a definitive script that is more akin to a traditional program, the ease of this trap did however show the ease with which limited dependency can be utilised within procedural programming.

In chapter three we considered dependency as a programming tool, first by using Michael Perry's automated dependency tracker and his Nebula example and secondly with Richard Cartwright's JaM2 Dependency Maintainer Machine. The Nebula example was re-designed and re-coded in Java using JaM2.

Chapter four went back to the basic essence of definitive scripts by using the Abstract Definitive Machine and compared definitions with entities as considered in the ADM. This showed that dependency can form the basis of a paradigm of modelling using definitive scripts. Ideas to improve the ADM and bring it up to date were considered and implementations of some ADM scripts using JaM2 were illustrated.

Chapter five considered the differences between programming and modelling and used Brian Cantwell-Smith's ideas of semantic relationships to illustrate the differences between Empirical modelling and traditional development. A comparison of modelling provision between spreadsheets and definitive scripts was made.

The appendices contain program and model listings of the developments carried out during this research.

Conclusions

1. Spreadsheet users create sophisticated end-user applications [Nar90] with only a fairly simple formula language, limited data types and simple conditionals. The users typically have no programming experience and do not have comprehensive knowledge of their own spreadsheet package. They use a small subset of the formula language and are still able to develop complex models with relative ease.

These complex models, however, are often poorly structured because they have grown and evolved without proper planning. The simple two dimensional cell layout is good for stimulating ideas of where values and formulae should be entered and it encourages growth and experimentation since there is always a bit more room to try out calculations. However this two dimensional structure is not appropriate for all models.

It is common to find 'spaghetti' style dependencies between cells that make it difficult to follow calculations. Spreadsheet applications are often not robust and a simple error can be

propagated through the whole calculation without warning. There are no checks on cell values to check ranges. The spreadsheet is not protected from the user, the design time and run time of a spreadsheet are the same thing and there is nothing to stop a user from having the same power as the developer of the spreadsheet, for example it is easy for a user who does not understand how a particular spreadsheet works to delete a vital formula by simply typing in a value to a cell.

Using definitive scripts as the basis for modelling avoids some of the problems of spreadsheets while retaining the ease of use. There is no pre-defined structure and this forces the modeller to think about the structure their script will take. However, like spreadsheets there are few checks in place to ensure the modeller does not make simple mistakes. These checks cannot be put in place without knowledge of the domain of the modeller. Using specific definitive notations designed for particular domains more checks can be performed that aid the modeller and restrict them to valid models in the chosen domain. For example, the JaM2 version of Nebula only allows construction of valid networks within the script.

2. As well as providing the basis for definitive scripts (and therefore Empirical Modelling) dependency can be a useful tool within more traditional programming environments. Chapter three showed how Michael Perry's Automated Dependency Tracker can simplify message propagation between the user interface and the information model within an OO application.

Referring back to the definitions of implicit and explicit dependency stated in Chapter 1 we showed that the dependency tracker helps maintain implicit dependencies within an OO program, but it does not add the formalism of explicit dependency. This means that since the dependency tracker is not given extra meta-information about the dependencies it is monitoring it cannot provide the order and automatic updating that exists within dependency maintainers such as Eden and JaM2.

The following diagram should give some clarity as to where the different kinds of dependency maintenance studied in this dissertation fit between the two extremes of implicit and explicit dependency:

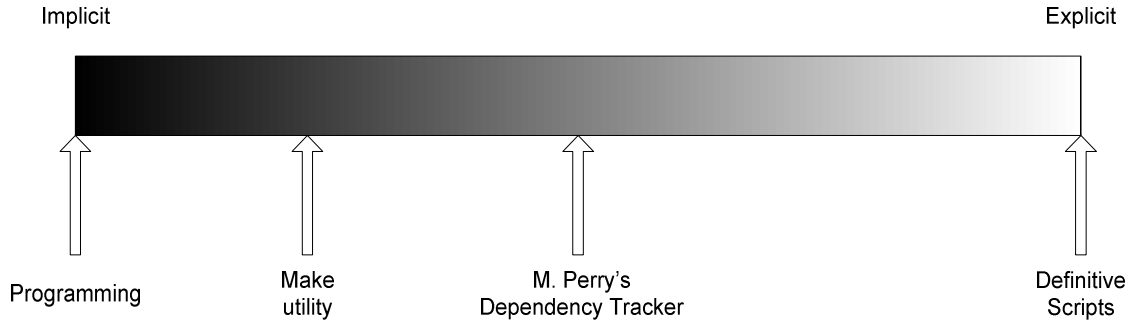

Figure 14 – Implicit to Explicit Dependency Scale

Based on the evidence of this research we suggest it is reasonable to claim the techniques for specifying and maintaining dependencies by means of definitive scripts can be applied usefully to more traditional programming paradigms. This would have benefits not only for developers using existing languages but would also expose Empirical Modelling to a wider audience and would build closer links with existing practice in the software development community.

Using a dependency maintainer that interfaces with more traditional programming paradigms and languages can provide a powerful tool to software developers to reduce the burden of

propagating state change throughout a software system. This viewpoint has hopefully been properly explored and justified in this dissertation.

3. The third conclusion I have reached through my research is on the comparison of the role of modelling to the role of programming. A good modelling process identifies reliable dependencies. A program that is easy to maintain has straight-forward functional and variable dependencies. The dependencies identified through good modelling can often be carried over to a resulting program intact. The models studied show that no matter how the dependencies are discovered (whether though Empirical Modelling or some other means) the resulting programs (whether object-oriented or not) represent all the identified reliable dependencies. This leads to the conclusion that dependency is not restricted to programming or modelling but is an intrinsic part of any domain.

Further Work

The software development community is currently moving away from pure object-oriented development and is concentrating on component oriented development. The initial promise of mass reuse that made object-oriented development very attractive has not been fully realised. By separating out independent units of functionality into modules that have well defined interfaces it is hoped that the promise of mass reuse can finally be achieved. For components to be independently deployed their mutual dependencies have to be carefully controlled. Even in the latest software technology dependency is still a complex issue. Comparing the dependency maintainers as used in Empirical Modelling with the dependency logic used within installers and configurators could yield some interesting applications for dependency maintenance.

Empirical Modelling and modelling with definitive scripts provide a great deal of openness to the modeller, anything can be changed at any time without any difference between design time and run time environments. Modern software development techniques that work well with small groups of programmers attempt to have the same degree of openness. 'Agile' methods (such as Extreme Programming) aspire to the same process of model development that is achieved with Empirical Modelling. Small changes are made to the program (redefinitions are made), which are tested (compared with a referent) and the software is available for release at any point (the model is 'ready' at all times). It would be extremely interesting to see how 'agile' development techniques match to Empirical Modelling.

In chapter 5 we considered applying ideas drawn from OO that could possibly improve the ADM, the main one considered was extending the ADM entity and definition naming scheme in order to provide more elegant expression and modularity than a flat namespace could provide. The possibility exists for other OO concepts to migrate into definitive scripts, to further improve modularity (and therefore abstraction) entity instantiation could be considered whereby entity declarations are blueprints (or factories) whereby entity instances are created. This would allow for simplification of scripts where there are many repeated definitions that define similar dependencies but on unrelated entities.

Bibliography

[BBY92] W. M. Beynon, I. Bridge and Y. P. Yung. Agent-oriented Modelling for a Vehicle Cruise Controller.In *Proc. ESDA Conference, ASME PD,* Vol. 47-4, pages 159-165, 1992.

[BcMan] Unix bc man page.

[BCS97] B. Cantwell-Smith. *One Hundred Billion Lines of C++*. CogSci News, Volume 10, Number 1, Spring 1997.

[BCS98] B. Cantwell-Smith. *On the Origin of Objects*. MIT Press, 1998.

[Beck99] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[Beck97] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[Bey99] W. M. Beynon. Empirical Modelling and the Foundations of Artificial Intelligence. *Lecture Notes in Artificial Intelligence 1562*, Springer-Verlag, 1999, pages 322-364, 1999.

[Burn01] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, S. Yang. *Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm*, Journal of Functional Programming 11(2), pages 155-206, March 2001

[BMRSS00] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd. 2000

[Brook95] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering Anniversary Edition*. Addison-Wesley Publishing Company, Inc. 1995

[Brown00] C.Brown. *An agent-based parsing system in EDEN*. 3rd year undergraduate project report, 2000.

[BSY89] W. M. Beynon, M. Slade and Y. W. Yung. Parallel Computation in Definitive Models. *CONPAR'88, British Computer Society Workshop Series CUP*, pages 359-367, 1989.

[Car01] R. I. Cartwright. *JaM2 API Documentation*.

[Car98] R. I. Cartwright. *Geometric Aspects of Empirical Modelling: Issues in Design and Implementation*. PhD thesis, Department of Computer Science, University of Warwick, Sept 1998.

[Cooper95] A. Cooper. The Essentials of User Interface Design. IDG Books 1995.

[CK93] P.B. Cragg & M. King. Spreadsheet Modelling Abuse: An Opportunity for OR ? , *Journal of the Operational Research Society*, 44:8, August 1993.

[Dict00] The American Heritage Dictionary of the English Language, Fourth

Edition. Houghton Mifflin Company, 2000

[Draughts95] R. Y. M. Au-Yeung. *Draughts* 3rd year undergraduate project report, 1995.

[Draughts96] S. Rawles. *Draughts*. 3rd year undergraduate project report, 1996.

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

[Fla97] D. Flanagan. *Java in a Nutshell*. O'Reilly and Associates, second edition, 1997.

[Fow00] M. Fowler. *Refactoring, Improving the design of existing code*. Addison-Wesley, 2000.

[GHR98] S. Gupta, J. Hartkopf, S. Ramaswamy. *Event Notifier, a pattern for Event Notification*. Java Report July 1998 Vol 3 Number 7. http://www.users.qwest.net/~hartkopf/notifier/

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hen96] B. Henderson-Sellers. *Object-oriented Metrics, Measures of Complexity*. Prentice Hall, 1996.

[Hol99] A. Holub. *Build User Interfaces for Object-Oriented Systems: The Visual-Proxy Architecture*. JavaWorld, September 1999. http://www.javaworld.com/javaworld/jw-09-1999/jw-09-toolbox_p.html

[Horr99] I. Horrocks, *Constructing the User Interface with Statecharts*. Addison-Wesley, 1999.

[ISL95] T. Isakowitz, S. Schocken and H. C. Lucas. Toward a Logical/Physical Theory of Spreadsheet Modeling. *ACM Transactions on Information Systems*, Vol. 13, No. 1, January 1995, pages 1-37, 1995.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Drive Approach*. Addison-Wesley, 1992.

[KaRi94] K. Christian and S. Richter. *The Unix Operating System, third edition*. Wiley, 1994.

[Loui] R. P. Loui. *Review of B.C.Smith's On the Origin of Objects*.

[MakeMan] Unix Make man page

[MakeWeb] Unix Make Programming Web Page

[Nar90] B. A. Nardi, and J. Miller. The spreadsheet interface: A basis for end user programming. *Proceedings Interact'90*. 27-31 August, pages 977-983, 1990.

[Nar93] B. A. Nardi. *A small matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

[Perry01] M. L. Perry. *Automate dependency tracking parts 1 – 3*. Javaworld, 2001. http://www.javaworld.com/javaworld/jw-08-2001/jw-0817-automatic_p.html

[Run02] J. Rungrattanaubol. *A treatise on Modelling with definitive scripts.* PhD thesis, Department of Computer Science, University of Warwick, 2002.

[Slade90] M. Slade. *Definitive Parallel Programming.* MSc thesis, Department of Computer Science, University of Warwick, 1990.

[Somm95] I. Sommerville. *Software Engineering, fifth edition*. Addison Wesley, 1995.

[Szy98] C. Szyperski. *Component Software, beyond Object-Oriented Programming*. Addison-Wesley, 1998.

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

[Yung93] S Yung. *Definitive Programming- a Paradigm for Exploratory Programming.* PhD thesis, Department of Computer Science, University of Warwick, 1993.

Appendices

Appendix 1 - Jugs JaM2 Script

```
mkdir Jugs;
mkdir User;
mkdir InitPour;
mkdir Pour;

cd Jugs;
capacityA = 5;
capacityB = 7;
contentA = 0;
contentB = 0;
emptyA is equal(contentA,0);
emptyB is equal(contentB,0);
fullA is equal(contentA,capacityA);
fullB is equal(contentB,capacityB);
avail1 is booleanNot(fullA);
avail2 is booleanNot(fullB);
avail3 is booleanNot(emptyA);
avail4 is booleanNot(emptyB);
avail6 is booleanAnd(avail2,avail3);
avail7 is booleanAnd(avail1,avail4);
avail5 is booleanOr(avail6,avail7);
updating = false;
cd ..;

cd User;
x is rand(5);
y = x;
guard1 is booleanAnd(booleanNot(./Jugs/updating),equal(0,y));
guard2 is booleanAnd(booleanNot(./Jugs/updating),equal(1,y));
guard3 is booleanAnd(booleanNot(./Jugs/updating),equal(2,y));
guard4 is booleanAnd(booleanNot(./Jugs/updating),equal(3,y));
guard5 is booleanAnd(booleanNot(./Jugs/updating),equal(4,y));
guard6 is booleanOr(./Jugs/updating,booleanNot(./Jugs/updating));
action1 is action(guard1,"p1 = print(\"Pressed 1\"); ./InitPour/option = 1;
./Jugs/updating = true;");
action2 is action(guard2,"p2 = print(\"Pressed 2\"); ./InitPour/option = 2;
./Jugs/updating = true;");
action3 is action(guard3,"p3 = print(\"Pressed 3\"); ./InitPour/option = 3;
./Jugs/updating = true;");
action4 is action(guard4,"p4 = print(\"Pressed 4\"); ./InitPour/option = 4;
./Jugs/updating = true;");
action5 is action(guard5,"p5 = print(\"Pressed 5\"); ./InitPour/option = 5;
./Jugs/updating = true;");
action6 is action(guard6,"p6 = print(cat(\"A:
\",numToString(./Jugs/contentA),\"- B: \",numToString(./Jugs/contentB)));
force ./User/x; ./User/y = ./User/x;");
cd ..;

cd InitPour;
option = 0;
guard1 is equal(option,1);
guard2 is equal(option,2);
guard3 is equal(option,3);
```

```
guard4 is equal(option,4);
guard5 is equal(option,6);
guard6 is equal(option,7);
guard7 is booleanAnd(equal(option,5),booleanNot(./Jugs/avail5));
guard8 is booleanAnd(equal(option,5),./Jugs/avail6);
guard9 is booleanAnd(equal(option,5),./Jugs/avail7);
action1 is action(guard1,"./Pour/option = 1;");
action2 is action(guard2,"./Pour/option = 2;");
action3 is action(guard3,"./Pour/option = 3;");
action4 is action(guard4,"./Pour/option = 4;");
action5 is action(guard5,"./Pour/option = 6;");
action6 is action(guard6,"./Pour/option = 7;");
action7 is action(guard7,"p7 = print(\"Can't perform pour\");
./Jugs/updating = false;");
action8 is action(guard8,"/Jugs/contentTemp = ./Jugs/contentA +
./Jugs/contentB; ./Jugs/contentB is ./Jugs/contentTemp - ./Jugs/contentA;
./InitPour/option = 6;");
action9 is action(guard9,"/Jugs/contentTemp = ./Jugs/contentA +
./Jugs/contentB; ./Jugs/contentB is ./Jugs/contentTemp - ./Jugs/contentA;
./InitPour/option = 7;");
cd ..;

cd Pour;
option = 0;
guard1 is booleanAnd(equal(option,1),./Jugs/avail1);
guard2 is booleanAnd(equal(option,1),booleanNot(./Jugs/avail1));
guard3 is booleanAnd(equal(option,2),./Jugs/avail2);
guard4 is booleanAnd(equal(option,2),booleanNot(./Jugs/avail2));
guard5 is booleanAnd(equal(option,3),./Jugs/avail3);
guard6 is booleanAnd(equal(option,3),booleanNot(./Jugs/avail3));
guard7 is booleanAnd(equal(option,4),./Jugs/avail4);
guard8 is booleanAnd(equal(option,4),booleanNot(./Jugs/avail4));
guard9 is booleanAnd(equal(option,6),./Jugs/avail6);
guard10 is booleanAnd(equal(option,6),booleanNot(./Jugs/avail6));
guard11 is booleanAnd(equal(option,7),./Jugs/avail7);
guard12 is booleanAnd(equal(option,7),booleanNot(./Jugs/avail7));
action1 is action(guard1,"./Jugs/contentA = ./Jugs/contentA + 1;");
action2 is action(guard2,"./Jugs/updating = false;");
action3 is action(guard3,"./Jugs/contentB = ./Jugs/contentB + 1;");
action4 is action(guard4,"./Jugs/updating = false;");
action5 is action(guard5,"./Jugs/contentA = ./Jugs/contentA - 1;");
action6 is action(guard6,"./Jugs/updating = false;");
action7 is action(guard7,"./Jugs/contentB = ./Jugs/contentB - 1;");
action8 is action(guard8,"./Jugs/updating = false;");
action9 is action(guard9,"./Jugs/contentA = ./Jugs/contentA - 1;");
action10 is action(guard10,"./Jugs/updating = false; ./Jugs/contentB =
./Jugs/contentB;");
action11 is action(guard11,"./Jugs/contentA = ./Jugs/contentA + 1;");
action12 is action(guard12,"./Jugs/updating = false; ./Jugs/contentB =
./Jugs/contentB;");
cd ..;

update;
```

Appendix 2 - Java Draughts Source Code

```
package Draughts;

public class Board {
	private Square[][] boardContents = new Square[8][8];
	private Game associatedGame;

	public Board(Game g) {
		for ( int i = 0; i < 8; i++ )
			for ( int j = 0; j < 8; j++ )
				boardContents[i][j] = new Square(i,j);

		associatedGame = g;
		clearBoard();
	}

	public void resetBoard() {
		for ( int i = 0; i < 8; i++ ) {
			for ( int j = 0; j < 8; j++ ) {
				int iMod = ( ( i + j ) % 2 );
				if ( ( iMod == 1 ) && ( j < 3 ) ) {
					boardContents[i][j].setContents(new
Piece(boardContents[i][j],false,false,associatedGame));
				}
				else if ( ( iMod == 1 ) && ( j > 4 ) ) {
					boardContents[i][j].setContents(new
Piece(boardContents[i][j],true,false,associatedGame));
				}
				else {
					boardContents[i][j].setContents(null);
				}
			}
		}
	}

	public void clearBoard() {
		for ( int i = 0; i < 8; i++ )
			for ( int j = 0; j < 8; j++ )
				boardContents[i][j].setContents(null);
	}

	public Piece getPiece(int ix, int iy) {
		return boardContents[ix][iy].getContents();
	}

	public Square getSquareRef(int ix, int iy) {
		return boardContents[ix][iy];
	}

	public Game getAssociatedGame() {
		return associatedGame;
	}
}

package Draughts;

import java.awt.*;

public class BoardPanel extends Panel {
	private Board board;

	public BoardPanel(Game game, TopPanel pnlTop) {
		board = game.getCurrentBoard();
		Color sqcol;
		GridLayout boardLayout = new GridLayout(8,0,0,0);
		setLayout(boardLayout);
		for ( int j = 0; j < 8; j++ )
			for ( int i = 0; i < 8; i++ ) {
				if ( ( i + j ) % 2 == 0 )
					sqcol = Color.yellow;
				else
					sqcol = Color.blue;
				BoardSquare bs = new BoardSquare(board.getSquareRef(i,j), sqcol,
game, pnlTop);
```

```
                                add(bs);
                        }
                }
        }

package Draughts;

import java.awt.*;
import java.awt.event.*;

public class BoardSquare extends Canvas implements MouseListener {
        private Color squareColor;
        private Square square;
        private Game game;
        private TopPanel pnlTop;

        public BoardSquare(Square sq, Color col, Game currentgame, TopPanel currentTopPanel) {
                square = sq;
                squareColor = col;
                game = currentgame;
                pnlTop = currentTopPanel;
                square.setBoardSquareRef(this);
                setBackground(squareColor);
                setSize(40,40);
                addMouseListener(this);
        }

        public void paint(Graphics g) {
                g.setColor(squareColor);
                g.fillRect(0,0,getSize().width,getSize().height);

                if (square.getContents() == null)
                        return;

                int oval_width = getSize().width - 10;
                int oval_height = getSize().height - 10;

                if (square.getContents().isWhite()) {
                        g.setColor(Color.white);
                        g.fillOval(5,5,oval_width,oval_height);
                        g.setColor(Color.black);
                        g.drawOval(5,5,oval_width - 1,oval_height - 1);
                }
                else {
                        g.setColor(Color.black);
                        g.fillOval(5,5,oval_width,oval_height);
                }

                if (square.getContents().isKing()) {
                        g.setColor(Color.red);
                        g.fillOval(getSize().width /2 - oval_width/4 - 1,
                                        getSize().height/2 - oval_height/4 - 1,
                                        oval_width/2 + 1,oval_height/2 + 1);
                }
        }

        public void mousePressed(MouseEvent evt)
        {
                if (game.getClickedOnce())
                {
                        game.getCurrentMove().setToCoords(square.getXPos(),square.getYPos());
                        pnlTop.DisplayMessage("To (" + Integer.toString(square.getXPos()) + ","
+ Integer.toString(square.getYPos()) + ")");
                        try {
                                game.getCurrentMove().ExecuteMove();
                        }
                        catch(InvalidMoveException e) {
                                pnlTop.DisplayMessage(e.getMessage());
                        }
                        game.swapClickedStatus();
                        pnlTop.getControlPanel().updateLastSquare("( " +
Integer.toString(square.getXPos()) + ", " + Integer.toString(square.getYPos()) + ")");

        pnlTop.getControlPanel().updateLastSquareContents(square.getContentsDescription());
                }
                else
                {
```

```
                        game.shiftToPrev();
                        game.getCurrentMove().setFromCoords(square.getXPos(),square.getYPos());

                        pnlTop.DisplayMessage("From (" + Integer.toString(square.getXPos()) +
","  + Integer.toString(square.getYPos()) + ")");
                        game.swapClickedStatus();
                        pnlTop.getControlPanel().updateLastSquare("( " +
Integer.toString(square.getXPos()) + ", " + Integer.toString(square.getYPos()) + ")");

        pnlTop.getControlPanel().updateLastSquareContents(square.getContentsDescription());
                }
        }
        public void mouseClicked(MouseEvent evt) {}
        public void mouseEntered(MouseEvent evt) {}
        public void mouseExited(MouseEvent evt) {}
        public void mouseReleased(MouseEvent evt) {}
}

package Draughts;

import java.awt.*;
import java.awt.event.*;

public class ControlPanel extends Panel implements ActionListener {
        private Board          associatedBoard;
        private TopPanel       associatedTopPanel;
        private Game           associatedGame;
        private TextField      whoseturn = new TextField("No game");
        private TextField      lastsquare = new TextField("No square");
        private TextField      lsqcontents = new TextField("No game");

        Button btnStart = new Button("Start game");
        Button btnReset = new Button("Reset board");
        Button btnQuit = new Button("Quit");

        public ControlPanel(Game currentGame, TopPanel currentTopPanel) {
                associatedBoard = currentGame.getCurrentBoard();
                associatedTopPanel = currentTopPanel;
                associatedGame = currentGame;
                setLayout(new GridLayout(3,0,10,10));

                add(btnStart);
                add(new Label("Whose turn:"));
                add(whoseturn);
                add(btnReset);
                add(new Label("Last Square:"));
                add(lastsquare);
                add(btnQuit);
                add(new Label("Contents:"));
                add(lsqcontents);

                whoseturn.setEditable(false);
                lastsquare.setEditable(false);
                lsqcontents.setEditable(false);

                btnStart.addActionListener(this);
                btnReset.addActionListener(this);
                btnQuit.addActionListener(this);
        }

        public void playerDisplay(Player toPlay) {
                if (toPlay.isWhite())
                        whoseturn.setText(Player.WHITE);
                else
                        whoseturn.setText(Player.BLACK);
        }

        public void actionPerformed(ActionEvent evt)
        {
                if (evt.getSource() == btnStart) {
                        associatedGame.newGame();
                        playerDisplay(associatedGame.getPlayer());
                        associatedTopPanel.DisplayMessage("A new game has begun.");
                }

                if (evt.getSource() == btnReset) {
                        associatedBoard.resetBoard();
```

```
                        associatedTopPanel.DisplayMessage("The board has been reset.");
                }

                if (evt.getSource() == btnQuit) {
                        associatedTopPanel.DisplayMessage("Quitting...");
                        System.exit(0);
                }
        }

        public void updateWhoseTurn (String s) {
                whoseturn.setText(s);
        }

        public void updateLastSquare (String s) {
                lastsquare.setText(s);
        }

        public void updateLastSquareContents (String s) {
                lsqcontents.setText(s);
        }
}

package Draughts;

import java.awt.*;
import java.util.*;

public class Game implements PlayerListener {
        private Player currentPlayer = new Player();
        private Board currentBoard = new Board(this);
        private boolean clickedOnce = false;
        private Move currentMove;
        private Move previousMove;
        Vector listeners = new Vector();

        public Game() {
                currentMove = new Move(currentBoard, currentPlayer);
                previousMove = new Move(currentBoard, currentPlayer);
                currentPlayer.addPlayerListener(this);
        }

        public Player getPlayer() {
                return(currentPlayer);
        }

        public void newGame() {
                currentBoard.resetBoard();
                currentPlayer.isNowBlack();
        }

        public Move getPreviousMove() {
                return previousMove;
        }

        public Move getCurrentMove() {
                return currentMove;
        }

        public void shiftToPrev() {
                previousMove = currentMove;
        }

        public Board getCurrentBoard() {
                return currentBoard;
        }

        public boolean getClickedOnce() {
                return clickedOnce;
        }

        public void swapClickedStatus() {
                clickedOnce = !clickedOnce;
        }

        public void addDraughtsListener(DraughtsListener listener)
        {
                if (!listeners.contains(listener))
```

```
                        listeners.addElement(listener);
        }

        public void removeDraughtsListener(DraughtsListener listener)
        {
                listeners.addElement(listener);
        }

        public void playersSwitched()
        {
                //Notify all PlayerListener's that the players have switched
                Enumeration e = listeners.elements();
                while (e.hasMoreElements())
                        ((DraughtsListener) e.nextElement()).playersSwitched();
        }
}

package Draughts;

public class InvalidMoveException extends Exception
{
        public InvalidMoveException() {};

        public InvalidMoveException(String errorString)
        {
                super(errorString);
        }
}

package Draughts;

public class Move {
        private int     fromx;
        private int     fromy;
        private int     tox;
        private int     toy;
        private Player currentPlayer;
        private Board associatedBoard;
        private Piece pieceToMove;

        public Move(Board e, Player f) {
                fromx = -1;
                fromy = -2;
                tox = -3;
                toy = -4;
                associatedBoard = e;
                currentPlayer = f;
        }

        public void ExecuteMove() throws InvalidMoveException {
                pieceToMove = associatedBoard.getPiece(fromx, fromy);

                if ( pieceToMove == null )
                        throw new InvalidMoveException("From-square is empty");

                if ( ( fromx == tox ) && ( fromy == toy ) )
                        throw new InvalidMoveException("Squares are not distinct.");

                if ( associatedBoard.getPiece(tox, toy)  != null )
                        throw new InvalidMoveException("To-square is non-empty");

                if ( pieceToMove.isWhite() != currentPlayer.isWhite() )
                        throw new InvalidMoveException("To-square is occupied by opponents
piece");

                if (pieceToMove.ExecuteMove(this))
                        currentPlayer.switchPlayer();
                else
                        throw new InvalidMoveException("Invalid Move.");
        }

        public void setFromCoords(int x, int y) {
                fromx = x;
                fromy = y;
        }

        public void setToCoords(int x, int y) {
```

```
                tox = x;
                toy = y;
        }

        public int getFromX() {
                return fromx;
        }

        public int getFromY() {
                return fromy;
        }

        public int getToX() {
                return tox;
        }

        public int getToY() {
                return toy;
        }
}

package Draughts;

public class Piece {
        private Square  contentsOf;
        private boolean pieceIsWhite;
        private boolean pieceIsKing;
        private Board   associatedBoard;
        private Game    associatedGame;

        public Piece(Square isq, boolean iw, boolean ik, Game g) {
                contentsOf = isq; pieceIsWhite = iw; pieceIsKing = ik;
                associatedGame = g;
                associatedBoard = g.getCurrentBoard();
        }

        public boolean isWhite() {
                return pieceIsWhite;
        }

        public boolean isBlack() {
                return !pieceIsWhite;
        }

        public boolean isKing() {
                return pieceIsKing;
        }

        public void crownPiece() {
                pieceIsKing = true;
        }

        public boolean ExecuteMoveWhite(Move m)
        {
                        if ( ( m.getToX() == ( m.getFromX() - 1 ) ) && ( m.getToY() == (
m.getFromY() - 1 ) ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() - 1 ) ) && ( m.getToY() == (
m.getFromY() + 1 ) ) && ( associatedBoard.getPiece( m.getFromX(), m.getFromY()).isKing() ==
true ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 1 ) ) && ( m.getToY() == (
m.getFromY() - 1 ) ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 1 ) ) && ( m.getToY() == (
m.getFromY() + 1 ) ) && ( associatedBoard.getPiece( m.getFromX(), m.getFromY()).isKing() ==
true ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
```

```
                        if ( ( m.getToX() == ( m.getFromX() - 2 ) ) && ( m.getToY() == (
m.getFromY() - 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY()
- 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY() - 1).isBlack() ==
true ) ) {
                                        associatedBoard.getSquareRef(m.getFromX() - 1,
m.getFromY() - 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        if ( ( m.getToX() == ( m.getFromX() - 2 ) ) && ( m.getToY() == (
m.getFromY() + 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY()
+ 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY() + 1).isBlack() ==
true ) && ( associatedBoard.getPiece(m.getFromX(), m.getFromY()).isKing() == true ) ) {
                                        associatedBoard.getSquareRef(m.getFromX() - 1,
m.getFromY() + 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 2 ) ) && ( m.getToY() == (
m.getFromY() - 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY()
- 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY() - 1).isBlack() ==
true ) ) {
                                        associatedBoard.getSquareRef(m.getFromX() + 1,
m.getFromY() - 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 2 ) ) && ( m.getToY() == (
m.getFromY() + 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY()
+ 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY() + 1).isBlack() ==
true ) && ( associatedBoard.getPiece(m.getFromX(), m.getFromY()).isKing() == true ) ) {
                                        associatedBoard.getSquareRef(m.getFromX() + 1,
m.getFromY() + 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                return(false);
        }

        public boolean ExecuteMoveBlack(Move m)
        {
                        if ( ( m.getToX() == ( m.getFromX() - 1 ) ) && ( m.getToY() == (
m.getFromY() - 1 ) ) && ( associatedBoard.getPiece( m.getFromX(), m.getFromY()).isKing() ==
true ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() - 1 ) ) && ( m.getToY() == (
m.getFromY() + 1 ) ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 1 ) ) && ( m.getToY() == (
m.getFromY() - 1 ) ) && ( associatedBoard.getPiece( m.getFromX(), m.getFromY()).isKing() ==
true ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 1 ) ) && ( m.getToY() == (
m.getFromY() + 1 ) ) ) {
                                MoveTo(m.getToX(),m.getToY());
                                return true;
                        }
                        if ( ( m.getToX() == ( m.getFromX() - 2 ) ) && ( m.getToY() == (
m.getFromY() - 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY()
- 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY() - 1).isWhite() ==
true ) && associatedBoard.getPiece(m.getFromX(), m.getFromY()).isKing() == true ) {
```

```
                                        associatedBoard.getSquareRef(m.getFromX() - 1,
m.getFromY() - 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        if ( ( m.getToX() == ( m.getFromX() - 2 ) ) && ( m.getToY() == (
m.getFromY() + 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY()
+ 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() - 1, m.getFromY() + 1).isWhite() ==
true ) ) {
                                        associatedBoard.getSquareRef(m.getFromX() - 1,
m.getFromY() + 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 2 ) ) && ( m.getToY() == (
m.getFromY() - 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY()
- 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY() - 1).isWhite() ==
true ) && associatedBoard.getPiece(m.getFromX(), m.getFromY()).isKing() == true ) {
                                        associatedBoard.getSquareRef(m.getFromX() + 1,
m.getFromY() - 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        if ( ( m.getToX() == ( m.getFromX() + 2 ) ) && ( m.getToY() == (
m.getFromY() + 2 ) ) ) {
                                if ( ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY()
+ 1 ) != null) && ( associatedBoard.getPiece( m.getFromX() + 1, m.getFromY() + 1).isWhite() ==
true ) ) {
                                        associatedBoard.getSquareRef(m.getFromX() + 1,
m.getFromY() + 1).setContents(null);
                                        MoveTo(m.getToX(),m.getToY());
                                        return true;
                                }
                        }
                        return(false);
        }

        public boolean ExecuteMove(Move m) {
                if ( associatedGame.getPlayer().isWhite() )
                        return(ExecuteMoveWhite(m));
                else
                        return(ExecuteMoveBlack(m));
        }

        public void MoveTo(int ix, int iy) {
                if (((iy == 0) && (associatedGame.getPlayer().isWhite())) || (iy == 7) &&
(associatedGame.getPlayer().isBlack()))
                        crownPiece();
                associatedBoard.getSquareRef(ix,iy).setContents(this);
                contentsOf.setContents(null);
                contentsOf = associatedBoard.getSquareRef(ix,iy);
        }
}

package Draughts;

import java.util.*;
public class Player {
        public static final String WHITE = "White";
        public static final String BLACK = "Black";

        private boolean playerIsWhite;

        Vector listeners = new Vector();

        public Player() {
                playerIsWhite = false;
        }

        public boolean isWhite() {
                return(playerIsWhite);
        }
```

```
	public boolean isBlack() {
		return(!playerIsWhite);
	}

	public void isNowWhite() {
		playerIsWhite = true;
	}

	public void isNowBlack() {
		playerIsWhite = false;
	}

	public void switchPlayer() {
		playerIsWhite = !playerIsWhite;

		//Notify all PlayerListener's that the players have switched
		Enumeration e = listeners.elements();
		while (e.hasMoreElements())
			((PlayerListener) e.nextElement()).playersSwitched();
	}

	public void addPlayerListener(PlayerListener listener)
	{
		if (!listeners.contains(listener))
			listeners.addElement(listener);
	}

	public void removePlayerListener(PlayerListener listener)
	{
		listeners.addElement(listener);
	}
}
```

```
package Draughts;

import  java.awt.*;

public class Session {
	public static void main(String[] args) {
		Game game = new Game();
		UserInterface currentUI = new UserInterface(game);
		game.addDraughtsListener(currentUI);
		currentUI.setSize(360,525);
		currentUI.setVisible(true);
	}
}
```

```
package Draughts;

public class Square {
	private int     xpos;
	private int     ypos;
	private Piece contents;
	private BoardSquare boardSquareRef;

	public Square(int ix, int iy) {
		xpos = ix;
		ypos = iy;
	}

	public void setContents(Piece newPiece) {
		contents = newPiece;
		if (boardSquareRef != null)
			boardSquareRef.repaint();
	}

	public Piece getContents() {
		return contents;
	}

	public BoardSquare getBoardSquareRef() {
		return boardSquareRef;
	}

	public void setBoardSquareRef(BoardSquare bs) {
		boardSquareRef = bs;
	}
```

```
	public int getXPos() {
		return xpos;
	}

	public int getYPos() {
		return ypos;
	}

	public String getContentsDescription() {
		if (contents == null)
			return "Nothing";
		else {
			if (contents.isWhite())
				return Player.WHITE;
			else
				return Player.BLACK;
		}
	}
}

package Draughts;

import java.awt.*;

public class TopPanel extends Panel {
	private Label message = new Label("-=>  Draughts, by Simon Rawles  <=-",Label.CENTER);
	private ControlPanel associatedControlPanel;

	public TopPanel(Game currentgame) {
		setLayout(new BorderLayout(10,10));
		ControlPanel controlPan = new ControlPanel(currentgame, this);
		associatedControlPanel = controlPan;
		BoardPanel boardPan = new BoardPanel(currentgame, this);
		add("North",controlPan);
		add("Center",boardPan);
		add("South",message);
	}

	public void DisplayMessage(String s) {
		this.message.setText(s);
		}

	public ControlPanel getControlPanel() {
		return associatedControlPanel;
		}
}

package Draughts;

import java.awt.*;
import java.awt.event.*;

public class UserInterface extends Frame implements WindowListener,DraughtsListener {
	private TopPanel	topPan;
	Game game;

	public UserInterface(Game currentgame) {
		setTitle("Draughts");
		setBackground(Color.lightGray);
		game = currentgame;
		topPan = new TopPanel(game);
		add("Center", topPan);
		addWindowListener(this);
	}

	public TopPanel getTopPanel() {
		return topPan;
	}

	public void playersSwitched()
	{
		if (game.getPlayer().isWhite())
			topPan.getControlPanel().updateWhoseTurn("White");
		else
			topPan.getControlPanel().updateWhoseTurn("Black");
		topPan.repaint();
```

```
        }

        public void windowClosing(WindowEvent evt)
        {
                System.exit(0);
        }
        public void windowClosed(WindowEvent evt) {}
        public void windowOpened(WindowEvent evt) {}
        public void windowIconified(WindowEvent evt) {}
        public void windowDeiconified(WindowEvent evt) {}
        public void windowActivated(WindowEvent evt) {}
        public void windowDeactivated(WindowEvent evt) {}
}
package Draughts;

public interface PlayerListener {
        public void playersSwitched();
}
package Draughts;

/**
 * Contains all the events that are generated by a draughts program
 * in order to update the UI
 */
public interface DraughtsListener {
        public void playersSwitched();
}
```

Appendix 3 - Java ADM Framework Source Code

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

import JaM2.*;
import JaM2.Base.*;
import JaM2.Parser.*;

/**
 * A demonstration of how to implement an ADM script using JaM2 and some
 * additional framework
 */
public class GCD
{
    //JaM2 Setup
    String rootPw = "secret";
    Script script = new Script(rootPw);

    //Update Control
    int cycles = 0;
    boolean running = false;

    //Class wide GUI Components
    Frame frame = new Frame("ADM Simulator - JaM2");
    TextArea text = new TextArea();
    Label lblCount = new Label("0        ");
    Button btnSingle = new Button("Perform Single Update");
    Button btnStart = new Button("Start Update Cycle");
    Button btnStop = new Button("Stop Update Cycle");
    Button btnUpdate = new Button("Update Action List");

    void buildDisplay()
    {
        Panel pnlButtons = new Panel(new FlowLayout(FlowLayout.CENTER));

        pnlButtons.add(btnSingle);
        pnlButtons.add(btnStart);
        pnlButtons.add(btnStop);
               pnlButtons.add(btnUpdate);

        Panel pnlLabels = new Panel(new FlowLayout(FlowLayout.CENTER));
        pnlLabels.add(new Label("Number of update cycles :"));
        pnlLabels.add(lblCount);
        lblCount.setFont(new Font("Monospaced", Font.BOLD, 24));

        Panel pnlText = new Panel(new BorderLayout());
        pnlText.add(new Label("Action List :"), BorderLayout.NORTH);
        pnlText.add(text, BorderLayout.CENTER);

               MenuBar menuBar = new MenuBar();
               Menu mnuFile = new Menu("File");
               MenuItem mnuFileNew = new MenuItem("New");
               MenuItem mnuFileOpen = new MenuItem("Open");
               MenuItem mnuFileExit = new MenuItem("Exit");

               mnuFile.add(mnuFileNew);
               mnuFile.add(mnuFileOpen);
               mnuFile.addSeparator();
               mnuFile.add(mnuFileExit);

               menuBar.add(mnuFile);

        frame.setBackground(Color.lightGray);
        frame.add(pnlLabels, BorderLayout.NORTH);
        frame.add(pnlText, BorderLayout.CENTER);
        frame.add(pnlButtons, BorderLayout.SOUTH);
               frame.setMenuBar(menuBar);

        btnSingle.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
```

```
                {
                    btnSingle.setEnabled(false);
                    btnStop.setEnabled(false);
                    btnStart.setEnabled(false);
                    singleUpdate();
                    btnSingle.setEnabled(true);
                    btnStop.setEnabled(true);
                    btnStart.setEnabled(true);
                }
            });

        mnuFileExit.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                    exit();
                }
            });

        mnuFileNew.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                    new ScriptWindow(script,rootPw);
                }
            });

        mnuFileOpen.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                        FileDialog fd = new FileDialog(frame,"Choose input file
:",FileDialog.LOAD);
                        fd.show();

                        try
                        {
                                File file = new File(fd.getDirectory() + fd.getFile());
                                FileReader fr = new FileReader(file);
                                StringWriter strw = new StringWriter();

                                char[] buffer = new char[1024];
                                int bytes_read = fr.read(buffer,0,buffer.length);
                                while (bytes_read > 0)
                                {
                                        strw.write(buffer,0,bytes_read);
                                        bytes_read = fr.read(buffer,0,buffer.length);
                                }

                                fr.close();

                               ScriptWindow sw = new ScriptWindow(script,rootPw);
                               sw.setInputText(strw.toString());
                          }
                          catch (IOException ioe)
                          {
                                ioe.printStackTrace();
                          }
                }
            });

        btnUpdate.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                    updateActionListDisplay();
                }
            });

        btnStart.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
```

```
            {
                start();
            }
        });

    btnStop.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                stop();
            }
        });

    frame.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent evt)
            {
                exit();
            }
        });

    frame.pack();
    frame.setVisible(true);
}

/**
 * Exit the application
 */
void exit()
{
    frame.dispose();
    System.exit(0);
}

/**
 * Perform a single execution cycle
 */
void singleUpdate()
{
    cycles++;
    updateCountDisplay();
    ActionOperator.clearActionList();

    //Create a parser to parse our built-in script
    StringReader sr = new StringReader(text.getText() + "\n update;");
    Parser parserBuiltIn = new Parser(new Yylex(sr));
    parserBuiltIn.setUser("root", rootPw);
    parserBuiltIn.setScript(script);
    parserBuiltIn.setOutputStream(System.out);
    parserBuiltIn.setConnection(null);
    try
    {
        parserBuiltIn.parse();
    }
    catch (Exception e)
    {
        System.err.println("Exception thrown during parsing:\n");
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
    }

    //Perform the redefinitions and update the action list display
    script.waitForEvaluation();
    updateActionListDisplay();
}

/**
 * Start the execution cycle
 */
void start()
{
    //Check we aren't already performing the cycle
    if (!running)
    {
        running = true;
```

```
        //Prevent inappropriate user interactions while we cycle
        btnStart.setEnabled(false);
        btnStop.setEnabled(true);
        btnSingle.setEnabled(false);

        //Execution cycle performed in a new thread
        Thread thread = new Thread(
            new Runnable()
            {
                public void run()
                {
                    while (running)
                    {
                        singleUpdate();

                        if (ActionOperator.getActionList().isEmpty())
                            running = false;

                        try
                        {
                            Thread.currentThread().sleep(2000);
                        }
                        catch (InterruptedException ie)
                        {
                        }
                    }
                    btnStart.setEnabled(true);
                    btnStop.setEnabled(false);
                    btnSingle.setEnabled(true);
                }
            });
        thread.start();
    }
}

void stop()
{
    running = false;
}

public void updateActionListDisplay()
{
    StringBuffer sb = new StringBuffer();
    Enumeration e = ActionOperator.getActionList().elements();
    while (e.hasMoreElements())
        sb.append((String) e.nextElement() + "\n");

    text.setText(sb.toString());
}

void updateCountDisplay()
{
    String strCycles = Integer.toString(cycles);

    if (!strCycles.equals(lblCount.getText()))
        lblCount.setText(strCycles);
}

public GCD()
{
    buildDisplay();

    //Add the base data types
    (new BaseUtils()).RegisterBaseClasses(script, rootPw);

    //Register the Action Operator
    String addOperators = "";
    addOperators += "addtomap JaMBoolean JaMString -> JaMBoolean;";
    addOperators += "addoperator action \"ActionOperator\";";
    addOperators += "clearmap;";
    addOperators += "addtomap JaMString -> JaMBoolean;";
    addOperators += "addoperator print \"PrintOperator\";";
    addOperators += "clearmap;";

    //Register the Random Operator
    addOperators += "addtomap JaMInteger -> JaMInteger;";
    addOperators += "addoperator rand \"JaMRand\";";
```

```
            addOperators += "clearmap;";
            addOperators += "update;";

            StringReader sr = new StringReader(addOperators);

            //Create a parser to parse our built-in script
            System.out.println("Creating Parser for built-in input");
            Parser parserBuiltIn = new Parser(new Yylex(sr));
            parserBuiltIn.setUser("root", rootPw);
            parserBuiltIn.setScript(script);
            parserBuiltIn.setOutputStream(System.out);
            parserBuiltIn.setConnection(null);
            try
            {
                parserBuiltIn.parse();
            }
            catch (Exception e)
            {
                System.err.println("Exception thrown during parsing:\n");
                System.err.println(e.getClass().getName() + ": " + e.getMessage());
            }

            script.waitForEvaluation();
            updateActionListDisplay();

            //Create a parser to parse user input
            System.out.println("Creating Parser for user input");
            Parser p = new Parser(new Yylex(System.in));
            p.setUser("root", rootPw);
            p.setScript(script);
            p.setOutputStream(System.out);
            try
            {
                p.parse();
            }
            catch (Exception e)
            {
                System.err.println("Exception thrown during parsing:\n");
                System.err.println(e.getClass().getName() + ": " + e.getMessage());
            }
        }

        public static void main(String[] args)
        {
            new GCD();
        }
    }

    import java.util.*;

    import JaM2.*;
    import JaM2.Base.*;

    /**
     * This Operator checks the value of the guard and if true adds the definition
     * into the script definition set. No useful information is returned as a JaM2
     * datatype
     */
    public class ActionOperator implements Operator
    {
        //The actions since the last update
        static Vector actions = new Vector();

        /**
         * Clears the Action List, called after each update
         */
        public static void clearActionList()
        {
            actions.clear();
        }

        /**
         * Returns the current Action List
         *
         * @return   Vector of String with each String a script redefinition
         */
        public static Vector getActionList()
```

```
    {
        return actions;
    }

    /**
     * This operator has the side effect of adding the guarded action command to
     * our action list if the guard evalutes to true, the operator itself only
     * returns the boolean value of the guard condition.
     */
    public Type f(ExprList parameters, Type toReturn)
    {
        //There are only ever two arguments (a guard and a definition)
        JaMBoolean guard = (JaMBoolean) parameters.getValueAt(0);

        //If the guard evaluates to true then add the action to our action list
        if (guard.value)
            actions.addElement(((JaMString) parameters.getValueAt(1)).value);

        return toReturn;
    }
}
import java.util.*;

import JaM2.*;
import JaM2.Base.*;

public class PrintOperator implements Operator
{
    public Type f(ExprList parameters, Type toReturn)
    {
        //There is only ever one argument
        JaMString string = (JaMString) parameters.getValueAt(0);

        System.out.println();
        System.out.println("OUTPUT:" + string.value);

        return toReturn;
    }
}

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import JaM2.*;
import JaM2.Parser.*;

public class ScriptWindow
{
       String last = "";
       Script script;
       String rootPw;

    Frame frmScript = new Frame("Script");

    Panel pnlText = new Panel(new GridLayout(0,1,0,0));
    TextArea txtInput = new TextArea();
    TextArea txtOutput = new TextArea();

    public ScriptWindow(Script scr, String pass)
    {
           Panel pnlButtons = new Panel(new FlowLayout(FlowLayout.RIGHT));
               Button btnRecall = new Button("Recall Last");
               Button btnSubmit = new Button("Submit");

               script = scr;
               rootPw = pass;

        pnlText.add(txtInput);
        pnlText.add(txtOutput);

               pnlButtons.add(btnRecall);
               pnlButtons.add(btnSubmit);

               txtOutput.setEditable(false);

        frmScript.add(pnlText, BorderLayout.CENTER);
        frmScript.add(pnlButtons, BorderLayout.SOUTH);
```

```
                frmScript.setBackground(Color.lightGray);

        frmScript.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing(WindowEvent evt)
                {
                    frmScript.dispose();
                }
            });

        btnSubmit.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                       last = txtInput.getText();

                       ByteArrayOutputStream baos = new ByteArrayOutputStream();
                       PrintStream ps = new PrintStream(baos);

                       //Create a parser to parse our built-in script
                               StringReader sr = new StringReader(last);
                               Parser parserBuiltIn = new Parser(new Yylex(sr));
                               parserBuiltIn.setUser("root", rootPw);
                               parserBuiltIn.setScript(script);
                               parserBuiltIn.setOutputStream(ps);
                               parserBuiltIn.setConnection(null);
                               try
                               {
                                   parserBuiltIn.parse();
                               }
                               catch (Exception e)
                               {
                                   System.err.println("Exception thrown during parsing:\n");
                                   System.err.println(e.getClass().getName() + ": " +
e.getMessage());
                               }

                               //Perform the redefinitions and update the action list display
                               script.waitForEvaluation();

                                       txtOutput.setText(baos.toString());
                       txtInput.setText("");
                }
            });

        btnRecall.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                       txtInput.setText(last);
                }
            });

        frmScript.pack();
        frmScript.setVisible(true);
    }

    public void setInputText(String text)
    {
       txtInput.setText(text);
    }
}

import JaM2.*;
import JaM2.Base.*;
import java.util.Random;

/**
 * Returns a pseudorandom, uniformly distributed int value between 0 (inclusive)
 * and the specified value (exclusive), drawn from this random number
 * generator's sequence.
 *
 * @author Tim Heron
 */
```

```
public class JaMRand implements Operator
{
    static Random random = new Random();

    public Type f(ExprList args, Type toReturn)
    {
        // There will always be 1 argument
        int maximum = 0;

        if (args.getTypeNameAt(0).equals("JaMInteger"))
            maximum = ((JaMInteger) args.getValueAt(0)).value;
        else
        {
            System.err.println("Error - Unknown type!");
            System.exit(1);
        }

        JaMInteger result = (JaMInteger) toReturn;

        result.value = (int) random.nextInt(maximum);

        return (result);
    }
}
```

Appendix 4 - Java Cruise Control Source Code

```
package vccs;

import java.awt.*;
import java.awt.event.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Accelerator extends Canvas
{
    Engine engine;
    double position = 0.0;
    Image offscreen = null;

    /**
     *  Constructor for the Accelerator object
     *
     * @param  engine  Description of the Parameter
     */
    public Accelerator(Engine engine)
    {
        this.engine = engine;

        class MouseAdapterPlus extends MouseAdapter implements MouseMotionListener
        {
            /**
             *  Description of the Method
             *
             * @param  evt  Description of the Parameter
             */
            public void mousePressed(MouseEvent evt)
            {
                double pos = 1 - (double) evt.getPoint().y / (double) getSize().height;
                setPosition(pos);
                repaint();
            }

            /**
             *  Description of the Method
             *
             * @param  evt  Description of the Parameter
             */
            public void mouseDragged(MouseEvent evt)
            {
                mousePressed(evt);
            }

            /**
             *  Description of the Method
             *
             * @param  evt  Description of the Parameter
             */
            public void mouseMoved(MouseEvent evt) { }
        }
        ;

        MouseAdapterPlus ma = new MouseAdapterPlus();
        addMouseListener(ma);
        addMouseMotionListener(ma);
    }

    /**
     *  Gets the position attribute of the Accelerator object
     *
     * @return    The position value
     */
    public double getPosition()
    {
        return position;
    }
```

```
    /**
     *  Sets the position attribute of the Accelerator object
     *
     * @param  position  The new position value
     */
    public void setPosition(double position)
    {
        if (position >= 0 && position <= 1)
        {
            this.position = position;

            //Also set the power of the Engine
            engine.setThrottlePosition(this.position);
        }
    }

    /**
     *  Description of the Method
     */
    public void invalidate()
    {
        offscreen = null;
        super.invalidate();
    }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void update(Graphics g)
    {
        paint(g);
    }

    /**
     *  Description of the Method
     *
     * @param  g2  Description of the Parameter
     */
    public void paint(Graphics g2)
    {
        int width = this.getSize().width;
        int height = this.getSize().height;

        if (offscreen == null)
            offscreen = createImage(width, height);

        Graphics g = offscreen.getGraphics();

        g.setColor(Color.white);
        g.fillRect(0, 0, width - 1, height - 1);
        g.setColor(Color.green);
        g.fillRect(0, (int) ((double) height * (1 - position)), width, height);
        g.setColor(Color.black);
        g.drawRect(0, 0, width - 1, height - 1);

        g2.drawImage(offscreen, 0, 0, this);
    }
}

package vccs;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Brake extends Canvas
{
    double position = 0.0;
```

```
Image offscreen = null;
Vector listeners = new Vector();

/**
 *  Constructor for the Brake object
 */
public Brake()
{

    class MouseAdapterPlus extends MouseAdapter implements MouseMotionListener
    {
        /**
         *  Description of the Method
         *
         * @param  evt  Description of the Parameter
         */
        public void mousePressed(MouseEvent evt)
        {
            double pos = 1 - (double) evt.getPoint().y / (double) getSize().height;
            setPosition(pos);
            fireEvent();
            repaint();
        }

        /**
         *  Description of the Method
         *
         * @param  evt  Description of the Parameter
         */
        public void mouseDragged(MouseEvent evt)
        {
            mousePressed(evt);
        }

        /**
         *  Description of the Method
         *
         * @param  evt  Description of the Parameter
         */
        public void mouseMoved(MouseEvent evt) { }
    }
    ;

    MouseAdapterPlus ma = new MouseAdapterPlus();
    addMouseListener(ma);
    addMouseMotionListener(ma);
}

/**
 *  Adds a feature to the BrakeListener attribute of the Brake object
 *
 * @param  bl  The feature to be added to the BrakeListener attribute
 */
public void addBrakeListener(BrakeListener bl)
{
    listeners.addElement(bl);
}

/**
 *  Description of the Method
 *
 * @param  bl  Description of the Parameter
 */
public void removeBrakeListener(BrakeListener bl)
{
    listeners.removeElement(bl);
}

/**
 *  Description of the Method
 */
void fireEvent()
{
    Enumeration e = listeners.elements();
    while (e.hasMoreElements())
        ((BrakeListener) e.nextElement()).brakePressed();
}
```

```
    /**
     *  Gets the position attribute of the Brake object
     *
     * @return    The position value
     */
    public double getPosition()
    {
        return position;
    }

    /**
     *  Sets the position attribute of the Brake object
     *
     * @param  position  The new position value
     */
    private void setPosition(double position)
    {
        if (position >= 0 && position <= 1)
        {
            this.position = position;
        }
    }

    /**
     *  Description of the Method
     */
    public void invalidate()
    {
        offscreen = null;
        super.invalidate();
    }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void update(Graphics g)
    {
        paint(g);
    }

    /**
     *  Description of the Method
     *
     * @param  g2  Description of the Parameter
     */
    public void paint(Graphics g2)
    {
        int width = this.getSize().width;
        int height = this.getSize().height;

        if (offscreen == null)
            offscreen = createImage(width, height);

        Graphics g = offscreen.getGraphics();

        g.setColor(Color.white);
        g.fillRect(0, 0, width - 1, height - 1);
        g.setColor(Color.red);
        g.fillRect(0, (int) ((double) height * (1 - position)), width, height);
        g.setColor(Color.black);
        g.drawRect(0, 0, width - 1, height - 1);

        g2.drawImage(offscreen, 0, 0, this);
    }
}

package vccs;

/**
 *  Description of the Interface
 *
 * @author     tim
 * @created    25 February 2002
 */
```

```
public interface BrakeListener
{
    /**
     *  Description of the Method
     */
    public void brakePressed();
}

package vccs;

import java.awt.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Car extends Panel
{
    double mass = 2500.0;
    /*
     *  total mass of car & contents [kg]
     */
    double roll = 50.0;
    /*
     *  rolling resistance factor [N m^-1 s]
     */
    final double g = 9.81;
    /*
     *  acceleration due to gravity [m s^-2]
     */
    double brak = 1500.0;
    /*
     *  braking (viscous) constant [N m^-1 s]
     */
    double forc = 40.0;
    /*
     *  torque to force conversion [m^-1]
     */
    double stic = 100.0;
    /*
     *  static friction force [N]
     */
    double position = 0.0;
    double speed = 0.0;
    double accel = 0.0;

    SpeedTransducer transducer = new SpeedTransducer();
    Brake brake = new Brake();
    Engine engine = new Engine();
    Accelerator accelerator = new Accelerator(engine);
    Speedometer speedometer = new Speedometer(transducer);
    CarDiagram carDiagram = new CarDiagram();
    Environment environment;
    EMU emu = new EMU(transducer, engine);

    Panel pnlAccelerator = new Panel(new BorderLayout(10, 0));
    Panel pnlBrake = new Panel(new BorderLayout(10, 0));
    Panel pnlBandA = new Panel(new GridLayout(0, 2, 10, 10));

    /**
     *  Constructor for the Car object
     *
     * @param  environment  Description of the Parameter
     */
    public Car(Environment environment)
    {
        this.environment = environment;
        setLayout(new GridLayout(2, 2, 10, 10));
        pnlAccelerator.add("Center", accelerator);
        pnlAccelerator.add("South", new Label("Accelerator", Label.CENTER));
        pnlBrake.add("Center", brake);
        pnlBrake.add("South", new Label("Brake", Label.CENTER));
        pnlBandA.add(pnlBrake);
        pnlBandA.add(pnlAccelerator);
        add(pnlBandA);
```

```
        add(emu);
        add(engine);
        add(speedometer);
        add(carDiagram);
        add(new About());

        //Let the cruise controller know when the brake has been pressed
        brake.addBrakeListener(emu);
    }

    /**
     *  Description of the Method
     *
     * @param  time  Description of the Parameter
     */
    public void updatePostion(double time)
    {
        //Recalculate position based on last position, last speed
        speed += accel * time;

        //Recalculate speed for next time
        updateVehicleDynamics(time);

        position += speed * time;
        position = position % 1000;
        if (position < 0)
            position = 1000 - position;

        environment.markPosition(position);
    }

    /**
     *  Description of the Method
     *
     * @param  time  Description of the Parameter
     */
    void updateVehicleDynamics(double time)
    {
        double windF = environment.airResistance(speed);
        double rollF = roll * speed;
        double gravF = g * mass * Math.sin(environment.hillGradient(position) * Math.PI /
200);
        double brakF = brak * brake.getPosition() * speed;
        double tracF = forc * engine.getPower();
        double sticF = stic * Utility.sgn(speed) * Utility.bound(speed, -0.01, 0.01);

        accel = time * (tracF - brakF - gravF - rollF - windF - sticF) / mass;

        //Update Forces diagram
        carDiagram.setAngle(environment.hillGradient(position));
        carDiagram.repaint();

        //Update Speed and Speedometer
        transducer.current_speed = speed;
        speedometer.repaint();
    }
}

package vccs;

import java.awt.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class CarDiagram extends Canvas
{
    double angle = 0.0;
    double brake_force = 0.0;
    double accel_force = 0.0;
    double wind_force = 0.0;
    Image offscreen = null;

    //Car Points
```

```
Point car[] = new Point[11];
Point ground[] = new Point[2];
Point wheels[] = new Point[2];
int wr = 10;
Line carl[] = new Line[12];

int scale = 10;
int car_height = 462 / scale;
int frontLength = 385 / scale;
int fullLength = 1000 / scale;
int roofLength = 277 / scale;
int gap = 31 / scale;

/**
 *  Constructor for the CarDiagram object
 */
public CarDiagram()
{
    car[0] = new Point(0, 0);
    car[1] = new Point(car[0].x, car_height / 2);
    car[2] = new Point(car[1].x - fullLength, car[1].y);
    car[3] = new Point(car[0].x - fullLength, car[0].y);
    car[4] = new Point(car[0].x - frontLength, car[0].y);
    car[5] = new Point(car[4].x, car[4].y + car_height);
    car[6] = new Point(car[5].x + roofLength, car[5].y);
    car[7] = new Point(car[1].x + -2 * gap, car[1].y + gap);
    car[8] = new Point(car[6].x - gap, car[6].y - gap);
    car[9] = new Point(car[5].x + gap, car[5].y - gap);
    car[10] = new Point(car[1].x - frontLength + gap, car[1].y + gap);

    ground[0] = new Point(-1000, 20);
    ground[1] = new Point(1000, 20);

    wheels[0] = new Point(-10, 10);
    wheels[1] = new Point(-90, 10);

    for (int i = 0; i < car.length; i++)
        car[i].y *= -1;

    carl[0] = new Line(car[0], car[1]);
    carl[1] = new Line(car[1], car[2]);
    carl[2] = new Line(car[2], car[3]);
    carl[3] = new Line(car[0], car[3]);
    carl[4] = new Line(car[4], car[5]);
    carl[5] = new Line(car[1], car[6]);
    carl[6] = new Line(car[5], car[6]);
    carl[7] = new Line(car[7], car[8]);
    carl[8] = new Line(car[8], car[9]);
    carl[9] = new Line(car[9], car[10]);
    carl[10] = new Line(car[10], car[7]);
    carl[11] = new Line(ground[0], ground[1]);
}

/**
 *  Description of the Method
 *
 * @param  g  Description of the Parameter
 */
public void update(Graphics g)
{
    paint(g);
}

/**
 *  Description of the Method
 */
public void invalidate()
{
    offscreen = null;
    super.invalidate();
}

/**
 *  Sets the angle attribute of the CarDiagram object
 *
 * @param  angle  The new angle value
 */
```

```
public void setAngle(double angle)
{
    this.angle = angle / 2;
}

int width = 0;
int height = 0;

/**
 *  Description of the Method
 *
 * @param  g2  Description of the Parameter
 */
public void paint(Graphics g2)
{
    width = this.getSize().width;
    height = this.getSize().height;

    if (offscreen == null)
        offscreen = createImage(width, height);

    Graphics g = offscreen.getGraphics();

    g.setColor(Color.cyan);
    g.fillRect(0, 0, width - 1, height - 1);

    drawVehicle(g);
    drawTraction(g);
    drawBraking(g);
    drawAcceleration(g);
    drawGravity(g);
    drawWind(g);

    g.setColor(Color.black);
    g.drawRect(0, 0, width - 1, height - 1);

    g2.drawImage(offscreen, 0, 0, this);

}

/**
 *  Description of the Method
 *
 * @param  g  Description of the Parameter
 */
public void drawVehicle(Graphics g)
{
    g.translate(fullLength + fullLength / 2, 75);
    g.setColor(Color.black);
    for (int i = 0; i < carl.length; i++)
    {
        Point p1 = Utility.rotate(carl[i].p1, -angle);
        Point p2 = Utility.rotate(carl[i].p2, -angle);
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
    Point w1 = Utility.rotate(wheels[0], -angle);
    Point w2 = Utility.rotate(wheels[1], -angle);
    g.drawOval(w1.x - wr, w1.y - wr, 2 * wr, 2 * wr);
    g.drawOval(w2.x - wr, w2.y - wr, 2 * wr, 2 * wr);
    g.translate(-(fullLength + fullLength / 2), -75);
}

/**
 *  Description of the Method
 *
 * @param  g  Description of the Parameter
 */
public void drawTraction(Graphics g) { }

/**
 *  Description of the Method
 *
 * @param  g  Description of the Parameter
 */
public void drawBraking(Graphics g) { }

/**
```

```
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void drawGravity(Graphics g) { }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void drawAcceleration(Graphics g) { }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void drawWind(Graphics g) { }
}

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
class Line
{

    Point p1 = null;
    Point p2 = null;

    /**
     *  Constructor for the Line object
     *
     * @param  p1  Description of the Parameter
     * @param  p2  Description of the Parameter
     */
    Line(Point p1, Point p2)
    {
        this.p1 = p1;
        this.p2 = p2;
    }

    /**
     *  Constructor for the Line object
     *
     * @param  l  Description of the Parameter
     */
    Line(Line l)
    {
        this(l.p1, l.p2);
    }
}

package vccs;

import java.awt.*;
import java.awt.event.*;

/**
 *  Vehicle Cruise Control Simulator To finish : Code Vehicle Dynamics
 *  Visualisation (need to add forces) Code SpeedTransducer Code Cruise Logic
 *  (in EMU) (incorrect version done) Verify vehicle dynamics to EM model
 *
 * @author     tim
 * @created    25 February 2002
 */
public class CruiseControl extends Frame implements Runnable
{
    Environment environment = new Environment();
    Car car = new Car(environment);

    Thread clock = new Thread(this);
    boolean running = true;
    boolean exit = false;
```

```
Panel pnlButtons = new Panel(new GridLayout(0, 1, 0, 5));
Panel pnlCruise = new Panel(new BorderLayout(5, 10));

Button btnStartClock = new Button("Start Clock");
Button btnStopClock = new Button("Stop Clock");

Image icon = null;

/**
 *  Constructor for the CruiseControl object
 */
public CruiseControl()
{
    super("Vehicle Cruise Control Simulator");

    //Load Icon
    icon = Toolkit.getDefaultToolkit().getImage("caricon.gif");
    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(icon, 0);
    try
    {
        tracker.waitForAll();
    }
    catch (InterruptedException ie)
    {}
    setIconImage(icon);

    setBackground(Color.lightGray);
    setLayout(new BorderLayout(10, 10));
    pnlButtons.add(btnStartClock);
    pnlButtons.add(btnStopClock);

    add("Center", car);
    add("South", pnlCruise);

    pnlCruise.add("Center", environment);
    //pnlCruise.add("East",pnlButtons);

    btnStopClock.setEnabled(false);

    addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent evt)
            {
                exit = true;
            }
        });

    btnStartClock.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                running = true;
                btnStartClock.setEnabled(false);
                btnStopClock.setEnabled(true);
            }
        });

    btnStopClock.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                running = false;
                btnStartClock.setEnabled(true);
                btnStopClock.setEnabled(false);
            }
        });

    clock.start();
}

/**
 *  Main processing method for the CruiseControl object
```

```
     */
    public void run()
    {
        long last_time = System.currentTimeMillis();

        while (!exit)
        {
            if (running)
            {
                car.updatePostion((System.currentTimeMillis() - last_time) / 10.0);
                last_time = System.currentTimeMillis();
            }

            try
            {
                clock.sleep(10);
            }
            catch (InterruptedException ex)
            {}
        }

        System.exit(0);
    }

    /**
     *  Gets the insets attribute of the CruiseControl object
     *
     * @return    The insets value
     */
    public Insets getInsets()
    {
        return new Insets(40, 10, 10, 10);
    }

    /**
     *  The main program for the CruiseControl class
     *
     * @param  args  The command line arguments
     */
    public static void main(String[] args)
    {
        CruiseControl cruiseControl = new CruiseControl();
        cruiseControl.setSize(640, 480);
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        cruiseControl.setLocation(screenSize.width / 2 - cruiseControl.getSize().width / 2,
screenSize.height / 2 - cruiseControl.getSize().height / 2);
        cruiseControl.setVisible(true);
    }
}

package vccs;

import java.awt.*;
import java.awt.event.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class EMU extends Panel implements Runnable, BrakeListener
{
    SpeedTransducer transducer;
    Engine engine;
    boolean cruiseEnabled = false;
    double cruiseSpeed = 0.0;

    double GainK = 100.0;
    // auto throttle controller gain

    Button btnCruiseOn = new Button("Cruise On");
    Button btnCruiseOff = new Button("Cruise Off");
    Button btnSetSpeed = new Button("Set Speed");

    Panel pnlButtons = new Panel(new FlowLayout(FlowLayout.CENTER, 10, 10));
```

```
Label lblEMU = new Label("Engine Management Unit", Label.CENTER);
TextField txtCruiseSpeed = new TextField(5);

Thread thdSpeedMaintainer = new Thread(this);

/**
 *  Constructor for the EMU object
 *
 * @param  transducer  Description of the Parameter
 * @param  engine      Description of the Parameter
 */
public EMU(SpeedTransducer transducer, Engine engine)
{
    this.transducer = transducer;
    this.engine = engine;

    txtCruiseSpeed.setEditable(false);
    txtCruiseSpeed.setText(Double.toString(cruiseSpeed));

    pnlButtons.add(btnCruiseOn);
    pnlButtons.add(btnSetSpeed);
    pnlButtons.add(btnCruiseOff);
    pnlButtons.add(txtCruiseSpeed);

    setLayout(new BorderLayout(5, 5));
    add("North", lblEMU);
    add("Center", pnlButtons);

    btnCruiseOn.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                setCruiseControlOn();
            }
        });

    btnCruiseOff.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                setCruiseControlOff();
            }
        });

    btnSetSpeed.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                try
                {
                    setCruiseSpeed(new Double(txtCruiseSpeed.getText()).doubleValue());
                }
                catch (NumberFormatException nfe)
                {}
            }
        });

    thdSpeedMaintainer.start();
}

/**
 *  Description of the Method
 */
public void brakePressed()
{
    setCruiseControlOff();
}

/**
 *  Main processing method for the EMU object
 */
public void run()
{
    while (true)
```

```
    {
        if (cruiseEnabled)
        {
            adjustThrottle();
        }

        try
        {
            thdSpeedMaintainer.sleep(10);
        }
        catch (InterruptedException ie)
        {}
    }
}

double deltaAutoThrottle = 0.0;

/**
 *  Description of the Method
 */
void adjustThrottle()
{
    double speedError = Utility.mph_to_mps(cruiseSpeed) - transducer.currentReading();

    if (speedError > 0.0)
        deltaAutoThrottle += 0.1;
    else
        deltaAutoThrottle -= 0.1;

    if (deltaAutoThrottle > 1.0)
        deltaAutoThrottle = 1.0;

    if (deltaAutoThrottle < 0.0)
        deltaAutoThrottle = 0.0;

    engine.setEMUThrottlePosition(deltaAutoThrottle);
}

/**
 *  Sets the cruiseSpeed attribute of the EMU object
 *
 * @param  cruiseSpeed  The new cruiseSpeed value
 */
public void setCruiseSpeed(double cruiseSpeed)
{
    if (cruiseSpeed >= 0)
        this.cruiseSpeed = cruiseSpeed;

    txtCruiseSpeed.setText(Double.toString(cruiseSpeed));
}

/**
 *  Sets the cruiseControlOn attribute of the EMU object
 */
public void setCruiseControlOn()
{
    cruiseEnabled = true;
    txtCruiseSpeed.setEditable(true);
    txtCruiseSpeed.setText(Double.toString(cruiseSpeed));
}

/**
 *  Sets the cruiseControlOff attribute of the EMU object
 */
public void setCruiseControlOff()
{
    cruiseEnabled = false;
    txtCruiseSpeed.setEditable(false);
    txtCruiseSpeed.setText(Double.toString(cruiseSpeed));
    engine.setEMUThrottlePosition(0.0);
}

/**
 *  Gets the insets attribute of the EMU object
 *
 * @return    The insets value
 */
```

```
    public Insets getInsets()
    {
        return new Insets(1, 1, 1, 1);
    }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
    }
}

package vccs;

import java.awt.*;
import java.awt.event.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Engine extends Panel
{
    //Maximum Engine Torque [kg m]
    double max_torque = 180.0;
    boolean running = false;

    double position = 0.0;
    double emu_position = 0.0;

    Label lblEngine = new Label("Engine is not running", Label.CENTER);
    Button btnEngine = new Button("Switch On");
    Panel pnlButtons = new Panel(new FlowLayout(FlowLayout.CENTER));

    /**
     *  Constructor for the Engine object
     */
    public Engine()
    {
        setLayout(new BorderLayout(10, 10));
        pnlButtons.add(btnEngine);
        add("North", lblEngine);
        add("Center", pnlButtons);
        btnEngine.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
                {
                    if (!running)
                        startEngine();
                    else
                        stopEngine();
                }
            });
    }

    /**
     *  Description of the Method
     */
    public void startEngine()
    {
        running = true;
        btnEngine.setLabel("Switch Off");
        lblEngine.setText("Engine is running");
    }

    /**
     *  Description of the Method
     */
    public void stopEngine()
```

```
    {
        running = false;
        btnEngine.setLabel("Switch On");
        lblEngine.setText("Engine is not running");
    }

    /**
     *  Gets the power attribute of the Engine object
     *
     * @return    The power value
     */
    public double getPower()
    {
        if (running)
            return Math.max(position, emu_position) * max_torque;
        else
            return 0.0;
    }

    /**
     *  Sets the throttlePosition attribute of the Engine object
     *
     * @param  position  The new throttlePosition value
     */
    public void setThrottlePosition(double position)
    {
        if (position >= 0.0 && position <= 1.0)
            this.position = position;
    }

    /**
     *  Sets the eMUThrottlePosition attribute of the Engine object
     *
     * @param  emu_position  The new eMUThrottlePosition value
     */
    public void setEMUThrottlePosition(double emu_position)
    {
        if (emu_position >= 0.0 && emu_position <= 1.0)
            this.emu_position = emu_position;
    }

    /**
     *  Gets the insets attribute of the Engine object
     *
     * @return    The insets value
     */
    public Insets getInsets()
    {
        return new Insets(1, 1, 1, 1);
    }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
    }
}

package vccs;

import java.awt.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Environment extends Canvas
{
    double markedPosition = -1.0;
    Image offscreen = null;
```

```
/**
 *  Description of the Field
 */
public final static double LENGTH = 1000.0;

/**
 *  Description of the Method
 *
 * @param  markedPosition  Description of the Parameter
 */
public void markPosition(double markedPosition)
{
    if (markedPosition != this.markedPosition)
    {
        this.markedPosition = markedPosition;
        repaint();
    }
}

/**
 *  Description of the Method
 *
 * @param  speed  Description of the Parameter
 * @return        Description of the Return Value
 */
public double airResistance(double speed)
{
    // wind resistance factor [N m^2 s^2]
    final double WIND_RESISTANCE = 5.0;

    return ((speed >= 0) ? 1 : -1) * WIND_RESISTANCE * Math.pow(speed, 2.0);
}

/**
 *  Description of the Method
 *
 * @param  position  Description of the Parameter
 * @return           Description of the Return Value
 */
public double hillGradient(double position)
{
    final double STEP = 0.001;
    double y1 = hillFunction(position);
    double y2 = hillFunction(position + STEP);
    return ((y2 - y1) / STEP) / Math.PI;
}

/**
 *  Gets the preferredSize attribute of the Environment object
 *
 * @return    The preferredSize value
 */
public Dimension getPreferredSize()
{
    return new Dimension(100, 100);
}

/**
 *  Description of the Method
 */
public void invalidate()
{
    offscreen = null;
    super.invalidate();
}

/**
 *  Description of the Method
 *
 * @param  g  Description of the Parameter
 */
public void update(Graphics g)
{
    paint(g);
}
```

```
    /**
     *  Description of the Method
     *
     * @param  g2  Description of the Parameter
     */
    public void paint(Graphics g2)
    {
        int width = this.getSize().width;
        int height = this.getSize().height;
        final int radius = 5;
        double xscale = width / LENGTH;
        double yscale = (height - radius) / LENGTH;

        if (offscreen == null)
            offscreen = createImage(width, height);

        Graphics g = offscreen.getGraphics();

        g.setColor(Color.cyan);
        g.fillRect(0, 0, width - 1, height - 1);

        int[] xpoints = new int[1001];
        int[] ypoints = new int[1001];

        for (int x = 0; x < xpoints.length; x++)
        {
            xpoints[x] = (int) (x * xscale);
            ypoints[x] = height - (int) (hillFunction(x) * yscale);
        }

        xpoints[1000] = xpoints[0];
        ypoints[1000] = ypoints[0];

        g.setColor(Color.green);
        g.fillPolygon(xpoints, ypoints, xpoints.length);

        //Draw the marked position
        if (markedPosition >= 0.0 && markedPosition <= LENGTH)
        {
            g.setColor(Color.red);
            g.fillOval((int) (markedPosition * xscale) - radius, height - (int)
(hillFunction(markedPosition) * yscale) - radius, radius * 2, radius * 2);
        }

        g.setColor(Color.black);
        g.drawRect(0, 0, width - 1, height - 1);

        g2.drawImage(offscreen, 0, 0, this);
    }

    /**
     *  Description of the Method
     *
     * @param  x  Description of the Parameter
     * @return    Description of the Return Value
     */
    double hillFunction(double x)
    {
        return ((-Math.cos(x / LENGTH * 2.0 * Math.PI) + 1.0) * 500.0);
    }
}

package vccs;

import java.awt.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Speedometer extends Canvas
{
    SpeedTransducer transducer;
    double min_speed = 0.0;
    double max_speed = 100.0;
```

```
double tick_spacing = 5.0;
int tick_length = 10;
double label_spacing = 10.0;

int centerx;
int centery;
int radius;
double factor;
double offset;

Image offscreen = null;

/**
 *  Constructor for the Speedometer object
 *
 * @param  transducer  Description of the Parameter
 */
public Speedometer(SpeedTransducer transducer)
{
    this.transducer = transducer;
}

/**
 *  Description of the Method
 */
public void invalidate()
{
    offscreen = null;
    super.invalidate();
}

/**
 *  Description of the Method
 *
 * @param  g  Description of the Parameter
 */
public void update(Graphics g)
{
    paint(g);
}

/**
 *  Description of the Method
 *
 * @param  g2  Description of the Parameter
 */
public void paint(Graphics g2)
{
    int width = this.getSize().width;
    int height = this.getSize().height;

    if (offscreen == null)
        offscreen = createImage(width, height);

    Graphics g = offscreen.getGraphics();

    g.setColor(Color.white);
    g.fillRect(0, 0, width - 1, height - 1);

    g.setColor(Color.black);
    centerx = width / 2;
    centery = height / 2;
    radius = Math.min(width, height) / 3;

    if (max_speed > min_speed)
    {
        factor = (1.5 * Math.PI) / (max_speed - min_speed);
        offset = (0.75 * Math.PI);
        drawIndicator(g);
        drawTicks(g);
        drawLabels(g);
    }

    g.setColor(Color.black);
    g.drawRect(0, 0, width - 1, height - 1);
```

```
            g2.drawImage(offscreen, 0, 0, this);
        }

        /**
         *  Description of the Method
         *
         * @param  g  Description of the Parameter
         */
        public void drawTicks(Graphics g)
        {
            if (tick_spacing != 0)
            {
                for (double tick = min_speed; tick <= max_speed; tick += tick_spacing)
                {
                    double angle = factor * tick + offset;

                    int xpos_outer = (int) (radius * (Math.cos(angle)) + centerx);
                    int ypos_outer = (int) (radius * (Math.sin(angle)) + centery);

                    int xpos_inner = (int) ((radius - tick_length) * (Math.cos(angle)) + centerx);
                    int ypos_inner = (int) ((radius - tick_length) * (Math.sin(angle)) + centery);

                    g.drawLine(xpos_inner, ypos_inner, xpos_outer, ypos_outer);
                }
            }
        }

        /**
         *  Description of the Method
         *
         * @param  g  Description of the Parameter
         */
        public void drawIndicator(Graphics g)
        {
            double indicator_value = Utility.mps_to_mph(Math.abs(transducer.currentReading()));

            if (max_speed > min_speed && indicator_value <= max_speed && indicator_value >=
min_speed)
            {
                //Calculate the angle of the indicator (in radians)
                double angle = indicator_value * factor + offset;

                //int xpos = (int) (0.7 * radius * (Math.cos(angle)));
                //int ypos = (int) (0.7 * radius * (Math.sin(angle)));

                //The speedometer length depends on the speed of the car
                int xpos = (int) (indicator_value / max_speed * radius * (Math.cos(angle)));
                int ypos = (int) (indicator_value / max_speed * radius * (Math.sin(angle)));

                g.drawLine(centerx, centery, xpos + centerx, ypos + centery);
            }
        }

        /**
         *  Description of the Method
         *
         * @param  g  Description of the Parameter
         */
        public void drawLabels(Graphics g)
        {
            if (label_spacing != 0)
            {
                int label_offset = -20;

                for (double tick = min_speed; tick <= max_speed; tick += label_spacing)
                {
                    double angle = factor * tick + offset;

                    int xpos = (int) ((radius - label_offset) * (Math.cos(angle)) + centerx);
                    int ypos = (int) ((radius - label_offset) * (Math.sin(angle)) + centery);

                    FontMetrics fm = g.getFontMetrics();
                    int fwidth = fm.stringWidth(Double.toString(tick));
                    int fheight = fm.getHeight() / 3;
                    g.drawString(Double.toString(tick), xpos - fwidth / 2, ypos + fheight);
                }
            }
```

```
    }
}

package vccs;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class SpeedTransducer
{
    /**
     *  Description of the Field
     */
    public double current_speed = 0.0;

    /**
     *  Description of the Method
     *
     * @return    Description of the Return Value
     */
    public double currentReading()
    {
        return current_speed;
    }
}

package vccs;

import java.awt.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class Utility
{
    /**
     *  Returns +1 or -1 if 'val' is +ve or -ve respectively
     *
     * @param  value  Description of the Parameter
     * @return        Description of the Return Value
     */
    public static double sgn(double value)
    {
        return value >= 0.0 ? 1.0 : -1.0;
    }

    /**
     *  Returns +1 if 'val' lies within range 'lowVal' - 'uppVal'
     *
     * @param  val     Description of the Parameter
     * @param  lowVal  Description of the Parameter
     * @param  uppVal  Description of the Parameter
     * @return         Description of the Return Value
     */
    public static double bound(double val, double lowVal, double uppVal)
    {
        return ((val >= lowVal) && (val <= uppVal)) ? 1.0 : 0.0;
    }

    /**
     *  Description of the Method
     *
     * @param  val     Description of the Parameter
     * @param  lowVal  Description of the Parameter
     * @param  uppVal  Description of the Parameter
     * @return         Description of the Return Value
     */
    public static double limit(double val, double lowVal, double uppVal)
    {
        if (val < lowVal)
            return lowVal;
```

```
        if (val > uppVal)
            return uppVal;
        return val;
    }

    /**
     *  Convert miles/hour to metres/sec
     *
     * @param  mph  Description of the Parameter
     * @return      Description of the Return Value
     */
    public static double mph_to_mps(double mph)
    {
        return 0.448 * mph;
    }

    /**
     *  Convert metres/sec to miles/hour
     *
     * @param  mps  Description of the Parameter
     * @return      Description of the Return Value
     */
    public static double mps_to_mph(double mps)
    {
        return 2.232 * mps;
    }

    /**
     *  Description of the Method
     *
     * @param  p  Description of the Parameter
     * @param  a  Description of the Parameter
     * @return    Description of the Return Value
     */
    public static Point rotate(Point p, double a)
    {
        Point pRotated = new Point();
        pRotated.x = (int) (Math.cos(a) * p.x - Math.sin(a) * p.y);
        pRotated.y = (int) (Math.sin(a) * p.x + Math.cos(a) * p.y);
        return pRotated;
    }
}

package vccs;

import java.awt.*;
import java.awt.event.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    22 February 2002
 */
public class About extends Panel
{
    /**
     *  Constructor for the About object
     */
    public About()
    {
        setLayout(new BorderLayout(10, 10));
        Label lblAbout = new Label("VCCS", Label.CENTER);
        Label lblAbout2 = new Label("Vehicle Cruise Control Simulator", Label.CENTER);
        Font font = new Font("Serif", Font.BOLD, 36);
        Font font2 = new Font("Serif", Font.BOLD, 12);
        lblAbout.setFont(font);
        lblAbout2.setFont(font2);
        add("North", lblAbout);
        add("Center", new ImageCanvas("caricon.gif"));
        add("South", lblAbout2);
    }


    /**
     *  Gets the insets attribute of the About object
     *
```

```
     * @return    The insets value
     */
    public Insets getInsets()
    {
        return new Insets(1, 1, 1, 1);
    }


    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
    }
}

package vccs;

import java.awt.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    25 February 2002
 */
public class ImageCanvas extends Canvas
{
    Image icon = null;

    /**
     *  Constructor for the ImageCanvas object
     *
     * @param  filename  Description of the Parameter
     */
    public ImageCanvas(String filename)
    {
        //Load Icon
        icon = Toolkit.getDefaultToolkit().getImage(filename);
        MediaTracker tracker = new MediaTracker(this);
        tracker.addImage(icon, 0);
        try
        {
            tracker.waitForAll();
        }
        catch (InterruptedException ie)
        {}
    }

    /**
     *  Description of the Method
     *
     * @param  g  Description of the Parameter
     */
    public void paint(Graphics g)
    {
        int width = getSize().width;
        int height = getSize().height;
        g.drawImage(icon, width / 2 - icon.getWidth(this) / 2, height / 2 -
icon.getHeight(this) / 2, this);
    }
}
```

Appendix 5 - JaM2 Nebula Source Code

```
import JaM2.*;

import java.awt.*;

/**
 * Adds a gDevice to the drawing canvas
 */
public class AddToCanvas implements JaM2.Operator
{
        static NetworkPanel panel;

        public static void setPanel(NetworkPanel panel)
        {
                AddToCanvas.panel = panel;
        }

    public Type f(ExprList args, Type toReturn)
    {
        //There are only ever two arguments (both Computers)
        Component deviceToAdd = (Component) args.getValueAt(0);
                panel.add(deviceToAdd);
        return (toReturn);
    }
}

import JaM2.*;

/**
 *  Description of the Class
 *
 * @author     tim
 * @created    22 February 2002
 */
public class Cable implements JaM2.TypeWithName
{
    /**
     *  Description of the Field
     */
    public final static int PATCH = 0;
    /**
     *  Description of the Field
     */
    public final static int CROSSOVER = 1;

    private ParameterSet parameters;

    int wiring_type = PATCH;

    Device from;       //Declaring these as Object Type means we can accept
    Device to;         //Either Hubs or Computers (Should really make Computer and Hub extend
a common Device class)

    /**
     *  Constructor for the Cable object
     */
    public Cable() { }

    /**
     *  Sets the jaMValue attribute of the Cable object
     *
     * @param  parameters  The new jaMValue value
     * @return             Description of the Return Value
     */
    public boolean setJaMValue(ParameterSet parameters)
    {
        this.parameters = parameters;

        wiring_type = parameters.getIntValue("wiring");
        from = (Device) parameters.getSubtypeValue("from");
        to = (Device) parameters.getSubtypeValue("to");

        return (true);
    }
```

```
    /**
     *  Gets the jaMValue attribute of the Cable object
     *
     * @return    The jaMValue value
     */
    public ParameterSet getJaMValue()
    {
        parameters.addParameter("wiring", wiring_type);
        parameters.addParameter("from","computer",from);
        parameters.addParameter("to","computer",to);

        return (parameters);
    }

    /**
     *  Gets the jaMTypeName attribute of the Cable object
     *
     * @return    The jaMTypeName value
     */
    public String getJaMTypeName()
    {
        return (parameters.getTypeName());
    }

    /**
     *  Description of the Method
     *
     * @return    Description of the Return Value
     */
    public boolean doJaMAction()
    {
        return (true);
    }

    String defn_name = "";

    public void setJaMDefinitionName(String defn_name)
    {
        this.defn_name = defn_name;
        }

        public String getJaMDefinitionName()
        {
                return defn_name;
        }

        public Device getFrom()
        {
                return from;
        }

        public Device getTo()
        {
                return to;
        }

    /**
     *  Description of the Method
     *
     * @return    Description of the Return Value
     */
    public String toString()
    {
        return ("A Cable with wiring " + (wiring_type == 0 ? "PATCH" : "CROSSOVER") + ",
connecting "+from+" to "+to);
    }

    public int getWiringType()
    {
        return wiring_type;
    }
}

import JaM2.*;

/**
```

```
 * Operator to join two Computers together with a Cable When joining two
 * computers you must use a CROSSOVER cable
 */
public class CableJoin implements JaM2.Operator
{
    public Type f(ExprList args, Type toReturn)
    {
        //There are only ever two arguments (both Computers)
        Device from = (Device) args.getValueAt(0);
        Device to = (Device) args.getValueAt(1);

        ParameterSet ps = toReturn.getJaMValue();
        ps.setParameter("from", "SUBTYPE", from);
        ps.setParameter("to", "SUBTYPE", to);

        if ((from instanceof Computer && to instanceof Computer) ||
                (from instanceof Hub && to instanceof Hub))
            ps.setParameter("wiring", Cable.CROSSOVER);
        else
            ps.setParameter("wiring", Cable.PATCH);

        toReturn.setJaMValue(ps);

        System.out.println("Joined a device called " + from + " to a computer called " + to +
".");

        return (toReturn);
    }
}
import JaM2.*;

public class Computer extends Device implements JaM2.TypeWithName
{
    String name = "(undefined)";

    public boolean setJaMValue(ParameterSet ps)
    {
        name = ps.getStringValue("name");
        parameters = ps;
        return (true);
    }

    public ParameterSet getJaMValue()
    {
        parameters.setParameter("name", name);
        return (parameters);
    }

    public String toString()
    {
        return (name);
    }

    /**
     * @see   Device
     */
    public java.awt.Dimension getSize()
    {
        return new java.awt.Dimension(55, 40);
    }

    /**
     * @see   Device
     */
    public void paint(java.awt.Graphics g, int x, int y)
    {
        g.setColor(java.awt.Color.black);
        g.drawRect(x + 12, y + 0, 31, 25);

        java.awt.Polygon p = new java.awt.Polygon();
        p.addPoint(x + 12, y + 27);
        p.addPoint(x + 42, y + 27);
        p.addPoint(x + 54, y + 39);
        p.addPoint(x, y + 39);
        g.drawPolygon(p);
    }
}
```

```
import JaM2.*;

public class Device implements JaM2.TypeWithName
{
    protected ParameterSet parameters;

    public boolean setJaMValue(ParameterSet ps)
    {
        parameters = ps;
        return (true);
    }

    public ParameterSet getJaMValue()
    {
        return (parameters);
    }

    public String getJaMTypeName()
    {
        return (parameters.getTypeName());
    }

    public boolean doJaMAction()
    {
        return (true);
    }

    String defn_name = "";

    public void setJaMDefinitionName(String defn_name)
    {
        this.defn_name = defn_name;
        }

        public String getJaMDefinitionName()
        {
                return defn_name;
        }

    /**
     * Draws the Glyph of the device on g at position x and y
     *
     * @param g  The Graphics object to draw the glyph on
     * @param x  X coordinate on g
     * @param y  Y coordinate on g
     */
    public void paint(java.awt.Graphics g, int x, int y) { }

    /**
     * Returns the size of the component glyph
     *
     * @return   Size of the Glyph
     */
    public java.awt.Dimension getSize()
    {
        return new java.awt.Dimension();
    }
}

import JaM2.*;

/**
 * Operator to draw a Cable
 */
public class DrawCable implements JaM2.Operator
{
    public Type f(ExprList args, Type toReturn)
    {
        Cable cable = (Cable) args.getValueAt(0);
        gDevice device1 = (gDevice) args.getValueAt(1);
        gDevice device2 = (gDevice) args.getValueAt(2);
        ParameterSet ps = toReturn.getJaMValue();
        ps.setParameter("cable", "SUBTYPE", cable);
        ps.setParameter("device1", "SUBTYPE", device1);
        ps.setParameter("device2", "SUBTYPE", device2);
        toReturn.setJaMValue(ps);
```

```
        return (toReturn);
    }
}

import JaM2.*;

public class Hub extends Device implements JaM2.TypeWithName
{
    public boolean setJaMValue(ParameterSet parameters)
    {
        this.parameters = parameters;
        return (true);
    }

    public String toString()
    {
        return ("Hub");
    }

    public java.awt.Dimension getSize()
    {
        return new java.awt.Dimension(50, 14);
    }

    public void paint(java.awt.Graphics g, int x, int y)
    {
        g.drawRect(x, y, 49, 13);

        g.drawOval(x + 9, y + 3, 7, 7);
        g.drawOval(x + 21, y + 3, 7, 7);
        g.drawOval(x + 33, y + 3, 7, 7);
    }
}

import JaM2.*;

public class Location implements JaM2.TypeWithName
{
    protected ParameterSet parameters;
    int x;
    int y;

    public boolean setJaMValue(ParameterSet ps)
    {
        parameters = ps;
        x = ps.getIntValue("x");
        y = ps.getIntValue("y");
        return (true);
    }

    public ParameterSet getJaMValue()
    {
        parameters.setParameter("x", x);
        parameters.setParameter("y", y);
        return (parameters);
    }

    public String getJaMTypeName()
    {
        return (parameters.getTypeName());
    }

    public boolean doJaMAction()
    {
        return (true);
    }

    String defn_name = "";

    public void setJaMDefinitionName(String defn_name)
    {
       this.defn_name = defn_name;
       }

       public String getJaMDefinitionName()
       {
              return defn_name;
```

```
        }

    public int getX()
    {
        return (x);
    }

    public int getY()
    {
        return (y);
    }

    public String toString()
    {
        return ("X: " + x + ", Y:" + y);
    }
}

import java.awt.*;
import java.util.*;

/**
 * A Canvas that draws Network components on it
 */
public class NetworkPanel extends Canvas
{
    //Stores all the components that are to be drawn on this canvas
    Vector vecComponents = new Vector();

    /**
     * Add a Component to this canvas
     *
     * @param c  Component to be added
     */
    public void add(Component c)
    {
        if (!vecComponents.contains(c))
                vecComponents.addElement(c);
    }

    /**
     * Remove a Component from this canvas
     *
     * @param c  Component to be removed
     */
    public void remove(Component c)
    {
        vecComponents.removeElement(c);
    }

    /**
     * Draw the components on the canvas
     */
    public void update(Graphics g)
    {
        paint(g);
    }

    /**
     * Draw the components on the canvas in the order they have been added to
     * the canvas
     */
    public void paint(Graphics g)
    {
        g.setColor(Color.white);
        g.fillRect(0, 0, getSize().width, getSize().height);

        for (int i = 0; i < vecComponents.size(); i++)
        {
            Component c = (Component) vecComponents.elementAt(i);
            c.paint(g);
        }
    }

    /**
     * Works out which component x and y is in, if x and y are within more than
     * one component then the first component that was added to the canvas is
```

```
     * returned.
     *
     * @return   The component that x and y is within
     */
    public Component getComponent(int x, int y)
    {
        for (int i = 0; i < vecComponents.size(); i++)
        {
            Component c = (Component) vecComponents.elementAt(i);
            if (c.getBounds().contains(x, y))
                return c;
        }

        return null;
    }
}

import JaM2.*;
import java.awt.*;

public class gCable extends Canvas implements JaM2.TypeWithName
{
    protected ParameterSet parameters;
    Cable cable;
    gDevice device1;
    gDevice device2;

    public boolean setJaMValue(ParameterSet ps)
    {
        parameters = ps;
        cable = (Cable) parameters.getSubtypeValue("cable");
        device1 = (gDevice) parameters.getSubtypeValue("device1");
        device2 = (gDevice) parameters.getSubtypeValue("device2");
        return true;
    }

    public ParameterSet getJaMValue()
    {
        parameters.setParameter("cable", cable);
        parameters.setParameter("device1", device1);
        parameters.setParameter("device2", device2);
        return parameters;
    }

    public boolean doJaMAction()
    {
        return true;
    }

    public String getJaMTypeName()
    {
        return parameters.getTypeName();
    }

    String defn_name = "";

    public void setJaMDefinitionName(String defn_name)
    {
       this.defn_name = defn_name;
       }

       public String getJaMDefinitionName()
       {
              return defn_name;
       }

    public String toString()
    {
        return "gCable : " + cable.toString();
    }

    /**
     * Returns a rectangle that the cable will fit within
     */
    public Rectangle getBounds()
    {
        Point p1 = device1.getCenter();
```

```
        Point p2 = device2.getCenter();

        return new Rectangle(p1.x, p1.y, Math.abs(p2.x - p1.x), Math.abs(p2.y - p1.y));
    }

    public void paint(Graphics g)
    {
        Point p1 = device1.getCenter();
        Point p2 = device2.getCenter();

        if (cable.getWiringType() == Cable.PATCH)
            g.setColor(Color.red);
        else
            g.setColor(Color.blue);

        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
}

import JaM2.*;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class gDevice extends Canvas implements JaM2.TypeWithName
{
    protected ParameterSet parameters;
    Location location;
    Device device;

    public boolean setJaMValue(ParameterSet ps)
    {
        parameters = ps;

        location = (Location) parameters.getSubtypeValue("location");
        device = (Device) parameters.getSubtypeValue("device");

        return true;
    }

    public ParameterSet getJaMValue()
    {
        parameters.setParameter("location", location);
        parameters.setParameter("device", (Device) device);
        return parameters;
    }

    public String getJaMTypeName()
    {
        String strType = parameters.getTypeName();
        return strType;
    }

    public boolean doJaMAction()
    {
        return true;
    }

    String defn_name = "";

    public void setJaMDefinitionName(String defn_name)
    {
       this.defn_name = defn_name;
       }

       public String getJaMDefinitionName()
       {
              return defn_name;
       }

    public String toString()
    {
        return "gDevice : " + location.toString() + "," + device.toString();
    }

    public Location getScreenLocation()
```

```
    {
        return location;
    }

    public Point getCenter()
    {
        Point p = new Point(location.getX() + getSize().width / 2, location.getY() +
getSize().height / 2);
        return p;
    }

    public Dimension getSize()
    {
        return device.getSize();
    }

    public Point getLocation(Point rv)
    {
        rv.x = location.getX();
        rv.y = location.getY();
        return rv;
    }

    public Rectangle getBounds()
    {
        return new Rectangle(location.getX(), location.getY(), device.getSize().width,
device.getSize().height);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        device.paint(g, location.getX(), location.getY());
    }
}

import JaM2.*;

/**
 * Operator to draw a Device
 */
public class DrawDevice implements JaM2.Operator
{
    public Type f(ExprList args, Type toReturn)
    {
        Device comp = (Device) args.getValueAt(0);
        Location loc = (Location) args.getValueAt(1);

        ParameterSet ps = toReturn.getJaMValue();
        ps.setParameter("location", "SUBTYPE", loc);
        ps.setParameter("device", "SUBTYPE", comp);
        toReturn.setJaMValue(ps);

        return (toReturn);
    }
}

import JaM2.*;
import JaM2.Base.*;
import JaM2.Parser.*;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class Network implements MouseListener, MouseMotionListener, ActionListener
{
    String rootPw = "secret";
    Script mainScript = new Script(rootPw);

    //Frame to display the network
    Frame frmNetwork;
    PopupMenu mnuPopup = new PopupMenu();
    TextArea txtInfo = new TextArea(3,3);

    MenuItem mnuAddComputer = new MenuItem("Add Computer");
```

```
MenuItem mnuAddHub = new MenuItem("Add Hub");

//A custom canvas to display the network components
NetworkPanel pnlNetwork;

//Coordinates of where a drag starts
int drag_x = 0;
int drag_y = 0;

//Component we are dragging
Component drag_component = null;

//Count of number of components we have added
//used to keep the definition names unique (1st starts at 10)
int object_num = 10;

public Network()
{
    //Set up the display
    pnlNetwork = new NetworkPanel();
    AddToCanvas.setPanel(pnlNetwork);
    pnlNetwork.addMouseListener(this);
    pnlNetwork.addMouseMotionListener(this);

    mnuAddComputer.addActionListener(this);
    mnuAddHub.addActionListener(this);

    mnuPopup.add(mnuAddComputer);
    mnuPopup.add(mnuAddHub);

    frmNetwork = new Frame("Nebula - JaM2");
    frmNetwork.setLayout(new BorderLayout());
           frmNetwork.add(mnuPopup);

           frmNetwork.add("Center",pnlNetwork);
    frmNetwork.add("South",txtInfo);
    txtInfo.setBackground(Color.lightGray);

    frmNetwork.setSize(600, 400);
    frmNetwork.setVisible(true);
    frmNetwork.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent evt)
            {
                System.exit(0);
            }
        });

    String addTypes = "";
    addTypes += "cleardefault;";
    addTypes += "addtype device \"Device\" top;";
    addTypes += "cleardefault;";
    addTypes += "addtype hub \"Hub\" device;";
    addTypes += "cleardefault;";
    addTypes += "addparameter name \"(default)\";";
    addTypes += "addtype computer \"Computer\" device;";
    addTypes += "cleardefault;";
    addTypes += "addparameter x 0;";
    addTypes += "addparameter y 0;";
    addTypes += "addtype location \"Location\" top;";
    addTypes += "cleardefault;";
    addTypes += "addparameter wiring " + Cable.PATCH + ";";
    addTypes += "addparameter from device {};";
    addTypes += "addparameter to device {};";
    addTypes += "addtype cable \"Cable\" top;";
    addTypes += "cleardefault;";
    addTypes += "addparameter location location {};";
    addTypes += "addparameter device device {};";
    addTypes += "addtype gdevice \"gDevice\" top;";
    addTypes += "cleardefault;";
    addTypes += "addparameter cable cable {};";
    addTypes += "addparameter device1 gdevice {};";
    addTypes += "addparameter device2 gdevice {};";
    addTypes += "addtype gcable \"gCable\" top;";

    String addOperators = "";
```

```
addOperators += "addtomap device device -> cable;";
addOperators += "addoperator join \"CableJoin\";";
addOperators += "clearmap;";
addOperators += "addtomap device location  -> gdevice;";
addOperators += "addoperator drawDevice \"DrawDevice\";";
addOperators += "clearmap;";
addOperators += "addtomap cable gdevice gdevice -> gcable;";
addOperators += "addoperator drawCable \"DrawCable\";";
addOperators += "clearmap;";
addOperators += "addtomap gdevice -> gdevice;";
addOperators += "addtomap gcable -> gcable;";
addOperators += "addoperator addToCanvas \"AddToCanvas\";";

String script = "";
script += "c1 is computer { name \"gem\" };";
script += "c2 is computer { name \"marble\" };";
script += "c3 is computer { name \"pitta\" };";
script += "c4 is computer { name \"stone\" };";
script += "h1 is hub {};";
script += "h2 is hub {};";
script += "cable1 is join( c1, h1 );";
script += "cable2 is join( c2, h1 );";
script += "cable3 is join( c3, h1 );";
script += "cable4 is join( h1, h2 );";
script += "cable5 is join( c4, h2 );";
script += "loc_c1 is location { x 25 y 50 };";
script += "loc_c2 is location { x 100 y 50 };";
script += "loc_c3 is location { x 175 y 50 };";
script += "loc_c4 is location { x 200 y 200 };";
script += "loc_h1 is location { x 100 y 150 };";
script += "loc_h2 is location { x 100 y 200 };";
script += "drawcomp1 is drawDevice(/c1, /loc_c1);";
script += "drawcomp2 is drawDevice(/c2, /loc_c2);";
script += "drawcomp3 is drawDevice(/c3, /loc_c3);";
script += "drawcomp4 is drawDevice(/c4, /loc_c4);";
script += "drawhub1 is drawDevice(/h1, /loc_h1);";
script += "drawhub2 is drawDevice(/h2, /loc_h2);";
script += "drawcable1 is drawCable(/cable1, /drawcomp1, /drawhub1);";
script += "drawcable2 is drawCable(/cable2, /drawcomp2, /drawhub1);";
script += "drawcable3 is drawCable(/cable3, /drawcomp3, /drawhub1);";
script += "drawcable4 is drawCable(/cable4, /drawhub1, /drawhub2);";
script += "drawcable5 is drawCable(/cable5, /drawcomp4, /drawhub2);";
script += "t1 is addToCanvas(drawcomp1);";
script += "t2 is addToCanvas(drawcomp2);";
script += "t3 is addToCanvas(drawcomp3);";
script += "t4 is addToCanvas(drawcomp4);";
script += "t5 is addToCanvas(drawhub1);";
script += "t6 is addToCanvas(drawhub2);";
script += "t7 is addToCanvas(drawcable1);";
script += "t8 is addToCanvas(drawcable2);";
script += "t9 is addToCanvas(drawcable3);";
script += "t10 is addToCanvas(drawcable4);";
script += "t11 is addToCanvas(drawcable5);";
script += "update;";

       System.out.println("Before Script:");

StringReader sr = new StringReader(addTypes + addOperators + script);

//Add the base data types
System.out.println("Registering Base Classes");
(new BaseUtils()).RegisterBaseClasses(mainScript, rootPw);

//Create a parser to parse our built-in script
System.out.println("Creating Parser for built-in input");
Parser parserBuiltIn = new Parser(new Yylex(sr));
parserBuiltIn.setUser("root", rootPw);
parserBuiltIn.setScript(mainScript);
parserBuiltIn.setOutputStream(System.out);
parserBuiltIn.setConnection(null);
try
{
    parserBuiltIn.parse();
}
catch (Exception e)
{
    System.err.println("Exception thrown during parsing:\n");
```

```
            System.err.println(e.getClass().getName() + ": " + e.getMessage());
        }

        //Draw the network for the first time
        pnlNetwork.repaint();

        //Create a parser to parse user input
        System.out.println("Creating Parser for user input");
        Parser p = new Parser(new Yylex(System.in));
        p.setUser("root", rootPw);
        p.setScript(mainScript);
        p.setOutputStream(System.out);
        try
        {
            p.parse();
        }
        catch (Exception e)
        {
            System.err.println("Exception thrown during parsing:\n");
            System.err.println(e.getClass().getName() + ": " + e.getMessage());
        }
    }

    public void mousePressed(MouseEvent evt)
    {
        if (evt.getButton() == MouseEvent.BUTTON3)
        {
               System.out.println("Popup Menu");
            mnuPopup.show(frmNetwork, evt.getX(), evt.getY());
            return;
        }

        //Store the coordinates of where we start dragging
        drag_x = evt.getX();
        drag_y = evt.getY();
        drag_component = pnlNetwork.getComponent(drag_x, drag_y);
    }

    public void mouseReleased(MouseEvent evt)
    {
        mouseDragged(evt);
        drag_component = null;
    }

    public void mouseClicked(MouseEvent evt) { }

    public void mouseEntered(MouseEvent evt) { }

    public void mouseExited(MouseEvent evt) { }

    public void mouseDragged(MouseEvent evt)
    {
        if (drag_component == null)
            return;

        //Now create a Location expression
        DefinitionSet ds = new DefinitionSet();
        JaMReturn r = mainScript.getSetableParameterSet("root", rootPw, "location");
        ParameterSet ps_loc1 = (ParameterSet) r.getValue();
        ps_loc1.setParameter("x", evt.getX() - (drag_x - drag_component.getBounds().x));
        ps_loc1.setParameter("y", evt.getY() - (drag_y - drag_component.getBounds().y));
        ExternalExpression ex_loc1 = ds.createExpression(ps_loc1);

        //Make a re-definition to the appropriate location
        Location location = ((gDevice) drag_component).getScreenLocation();
        String def = ((TypeWithName) location).getJaMDefinitionName();
        ds.defineDependency(def, ex_loc1);

        //update;
        r = mainScript.changeDefinitions("root", rootPw, ds);
        mainScript.waitForEvaluation();

        //Restart the drag from this location
        drag_x = evt.getX();
        drag_y = evt.getY();

        pnlNetwork.repaint();
```

```
    }

        Component currentDevice = null;

    public void mouseMoved(MouseEvent evt)
    {
        Component device = pnlNetwork.getComponent(evt.getX(),evt.getY());
        if (device instanceof gDevice)
                currentDevice = device;

                if (currentDevice != null)
                {
                String strInfo = "";
                String def = ((TypeWithName) currentDevice).getJaMDefinitionName();
                        JaMReturn r = mainScript.getDefinitionValue("root", rootPw, def);

                        JaM2.TypeWithName value = (TypeWithName) r.getValue();

                        strInfo = "Name :"+value.getJaMDefinitionName();
                        strInfo += "\nCurrent Value :"+value.toString();
                        r = mainScript.getExpression("root", rootPw, def);
                        strInfo += "\nCurrent Location Defintion :"+r.getValue().toString();

                        if (!strInfo.equals(txtInfo.getText()))
                                txtInfo.setText(strInfo);
                }
                else
                        txtInfo.setText("");
    }

    public void actionPerformed(ActionEvent evt)
    {
        String script = "";
        String cname = new String("c" + object_num);

        if (evt.getSource() == mnuAddComputer)
        {
            script += cname + " is computer { name \"gem\" };";
            script += "loc_" + cname + " is location { x 0 y 0 };";
            script += "drawcomp_" + cname + " is drawDevice(/" + cname + ", /loc_" + cname +
");";
            script += "add_" + cname + " is addToCanvas(drawcomp_" + cname + ");";
            script += "update;";
        }
        else if (evt.getSource() == mnuAddHub)
        {
            script += cname + " is hub { };";
            script += "loc_" + cname + " is location { x 0 y 0 };";
            script += "drawcomp_" + cname + " is drawDevice(/" + cname + ", /loc_" + cname +
");";
            script += "add_" + cname + " is addToCanvas(drawcomp_" + cname + ");";
            script += "update;";
        }
        else
            return;

        object_num++;

        StringReader sr = new StringReader(script);

        //Create a parser to parse our built-in script
        System.out.println("Creating Parser for built-in input");
        Parser parserBuiltIn = new Parser(new Yylex(sr));
        parserBuiltIn.setUser("root", rootPw);
        parserBuiltIn.setScript(mainScript);
        parserBuiltIn.setOutputStream(System.out);
        parserBuiltIn.setConnection(null);
        try
        {
            parserBuiltIn.parse();
        }
        catch (Exception e)
        {
            System.err.println("Exception thrown during parsing:\n");
            System.err.println(e.getClass().getName() + ": " + e.getMessage());
        }
```

```
            //Redraw the network
            pnlNetwork.repaint();
        }

        /**
         * The main program for the Nebula example
         *
         * @param args  The command line arguments
         */
        public static void main(String[] args)
        {
            new Network();
        }
}
```

Appendix 6 – Translated ADM script 'swap'

```
%Eden
func STR {
      auto s,i;
      if (type($1) == "char")
            return "'"//str($1)//"'";
      else if (type($1) == "string")
            return Quote($1);
      else if (type($1) == "list") {
            s = "[";
            for (i = 1; i <= $1#; i++)
                  s = (i != 1) ? s//","//STR($1[i]) : s//STR($1[i]);
            return s//"]";
      } else return str($1);
}

func Quote { para s;
    auto ret, i;
    if (type(s) != "string") { return str(s); }
    ret = "\"";
    for (i = 1; i <= s#; i++) {
        switch (s[i]) {
        case '"':
            ret = ret // "\\\"";
            break;
        case '\\':
            ret = ret // "\\\\";
            break;
        default:
            ret = ret // s[i];
            break;
        }
    }
    return ret // "\"";
}

func Quote0 { para s;
    auto ret, i;
    if (type(s) != "string") { return str(s); }
    ret = "";

    for (i = 1; i <= s#; i++) {
        switch (s[i]) {
        case '"':
            ret = ret // "\\\"";
            break;
        case '\\':
            ret = ret // "\\\\";
            break;
        default:
            ret = ret // s[i];
            break;
        }
    }
    return ret;
}
func QUOTE { return (type($1) == "string") ? Quote0(Quote($1)) :
STR($1); }
```

```
func Rand { return rand() % $1 + 1; }

TRUE = 1;
FALSE = 0;

/* E.g. sleep(5) = sleep for at least 5 seconds */
proc sleep { para period;
      auto start, current;
      start = ftime();
      for (current = ftime();
            (current[2] - start[2]) / 1000.0 + current[1] - start[1]
< period;
            current = ftime());
}

proc clocking : sysClock, stopClock {
    if (!stopClock && (stopTime < 0 || sysClock < stopTime)) {
      if (Pause > 0)
            sleep(Pause / 2.0);
      if (sysClock != -1) {
            todo("sysClock = -1;\n");
      } else {
            nextClock++;
            if (!Silent)
                  todo("writeln(\"time = \", nextClock);\n");
            todo("sysClock = nextClock;\n");
      }
    }
}

proc syncClock : sysClock { if (sysClock != -1) iClock = sysClock; }

proc startClk : startClock {
      if (sysClock == @) nextClock = sysClock = 0;
      stopClock = !startClock;
}

proc setClock { sysClock = nextClock = $1; }

srand(seed = time());
/* random seed remembered to enable repetition of the same simulation
*/

stopClock = 1;    /* set stopClock/startClock to stop/start system
clock */
stopTime = -1;    /* set stopTime to the system exit time */
Silent = 0; /* set to suppress showing of time */
Pause = 0;  /* minimum gap between two system clock pulses, in
seconds */

proc swap {
if (!Silent) writeln("instantiating swap()");
autocalc=0;
execute("
a = 1;
b = 2;
");
execute("
proc swap_action_1 : sysClock {
if (sysClock == -1) return;
```

```
      if (TRUE) {
            writeln(\"a is \",a,\" and b is \",b);
            todo(\"a = \"//STR(b)//\"; b = \"//STR(a)//\"; \");
      }
}
");
autocalc=1;
}

proc delete_swap {
if (!Silent) writeln("delete swap()");
execute("
forget(\"swap_action_1\");
b = @;
a = @;
");
}
swap();
```