Representing Memory Graphs
Interactive debugging using the Eclipse Framework

Luke Cartey
St. Catherine's College

May 14, 2008

Contents

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Memory Graphs . . . 1
- 1.3 Prior work . . . 2
- 1.4 Aims . . . 2

2 Definitions and Descriptions **3**
- 2.1 The Java language and the Java Virtual Machine . . . 3
- 2.2 Further Application . . . 3
- 2.3 Definitions . . . 3
  - 2.3.1 Simple Memory Graph . . . 4
  - 2.3.2 Adapted Memory Graph . . . 4
  - 2.3.3 Memory Slices . . . 4
  - 2.3.4 Object Depth . . . 5
  - 2.3.5 Object Ownership . . . 5
  - 2.3.6 Other Interesting Concepts . . . 5

3 Creating Memory Graphs in the Eclipse Development Framework **6**
- 3.1 Analysis and Design . . . 6
  - 3.1.1 Eclipse . . . 6
  - 3.1.2 Architecture . . . 6
  - 3.1.3 Design . . . 7
- 3.2 Implementation . . . 9
  - 3.2.1 Creating a memory graph . . . 9
  - 3.2.2 Comparing memory graphs . . . 9
  - 3.2.3 Primitive values . . . 11
  - 3.2.4 Model Interaction . . . 11

4 Making a memory graph more usable **12**
- 4.1 Focusing and Scaling . . . 12
- 4.2 Identifying Importance . . . 12
  - 4.2.1 Motivation . . . 12
  - 4.2.2 Implementation . . . 13
- 4.3 Folding . . . 13
  - 4.3.1 Example . . . 14

5 Testing and Development **17**
- 5.1 Code-based tests . . . 17
- 5.2 Empirical Testing . . . 17
- 5.3 Functional Tests . . . 17
- 5.4 Trac and Subversion . . . 17

6 Evaluation **20**
- 6.1 Test Programs . . . 20
  - 6.1.1 Tree . . . 20
  - 6.1.2 Composite . . . 22
- 6.2 Utility . . . 22

7 Conclusion **25**
7.1 Summary 25
7.2 Further Work 25

Abstract

The overall aim of the project was to recover, analyse and represent the memory graph from a running program, in such a way that other programs can use a public API to visualise the memory graph. It has achieved the following:

- Building an integrated plugin for an IDE that acquires and represents the memory graph of one or more running programs, in such a way that it can be used in conjunction with existing debugging tools.
- Exposing the representation to other plugins, through an API, with the intention of having the memory graph displayed visually.
- Further developing the idea to produce novel ways of reducing the output memory graph size by removing certain objects, according to a set of application rules.
- Investigating ways in which a memory graph might be focused, particularly through identifying subjectively important or relevant objects, providing some promising initial results.

Previous tools have been little more than experiments, testing the concept - this projects represents some of the first steps to update the methods for use with modern development environments in a generic fashion. In particular, the efforts to transform a memory graph by selectively removing certain nodes and the research into analysing important or relevant objects are areas of software development research that have not previously been investigated.

Chapter 1

Introduction

1.1 Purpose

The current state of debugging is, in general, a very static and textual affair. Whilst for small programs, the overhead of comprehending this representation is small, for larger programs it becomes increasingly tricky to visualise the state and structure of a running program. By making accessible this state and structure in such a way that it can be visualised, we can provide the programmer with a useful tool for comprehending a program. Programming, and, in particular, debugging, is a matter of holding many features simultaneously within your mind - by facilitating a graphical notation for displaying such features we can potentially increase the efficiency of the programmer.

Describing the representation of a running program is somewhat tricky. On the one hand, we wish to display as much information as possible, but on the other hand we wish to provide a simple visualisation that allows the user to comprehend the program as easily as possible. Selecting a balance between the two is a key feature of any potential solution.

One such solution is to introspect the state through instrumentation. Essentially, we append extra instructions at various points throughout the code, and use these to create a model of the running program at any given time. Introspection can occur manually or can occur through automated techniques, such as those that retrospectively modify the source - Aspect Orientated Programming, for example, or macros. Either way, introspection can prove to be useful for recovering data for representing specific algorithms in specific circumstances, but the need to define points to introspect limits the potential for a general, off-the-shelf solution. Instead, we choose to consider a more flexible approach, by considering precisely the complete state of memory of a running program - by accessing the heap of a given program directly. One particular representation of the memory is that of a memory graph.

1.2 Memory Graphs

Informally, a memory graph is a directed graph which represents structures stored in memory, with edges representing composition of nested structures. For example, in object-orientated languages, we might model objects as nodes and variables inside those objects as links to associated objects. This memory graph can be presented to the user, naturally, as a series of interlinked graphical objects. Initially, this may seem to be a diverting, but somewhat pointless exercise. There are, however, a number of practical uses for such graphs, that could aid in the debugging of a program, including:

- Visual comparison of a memory graph (or sub-section) before and after a certain line of code. We could use this to see the effects of a method-call - for example, if the program breaks at that point. By highlighting changes within the graph, we can see much more rapidly what has changed.
- We can also compare two separate runs of a program at the same breakpoint - if, for example, we know a bug has been introduced at a certain revision, we can identify how the data has changed between the two runs.

As well as aiding the debugging process, a memory graph can also clarify features of a program, particularly for programmers working on a new project. Such benefits may include:

- Inspection of the composition of an object or structure
- How two given objects are related in the memory graph.
- Identifying important objects within the system.

It is a well established fact that humans more readily understand data when displayed in a graphical form. (Diehl, 2007) Memory Graphs provide us with a way of visualising the data stored in a running program - in a way that can more readily transfer information about the in-depth workings of the current run of a program than a simple list of objects ever would.

However, the standard memory graph, whilst working well on small systems, can quickly become conceptually overcrowded as the complexity of the software in hand increases. This is unfortunate, since this kind of graphical structure is particularly valuable when attempting to debug systems where the conceptual overhead is high. So we also wish to find ways in which we might make the memory graph a more usable proposition in practice.

1.3 Prior work

The idea of developing memory graphs is not new. The DDD tool provides a graphical front end for the GDB debugger. However, it only displays limited views of the memory graph - each object has to be selectively displayed by the user, a tedious process for large graphs. Zimmermann and Zeller (2001) create memory graphs based upon programs written in C. This was created as a stand-alone tool, developed on top of a standard debugger, in this case GDB. By developing for C, they had to contend with representing a number of different data-structures.

Potanin et al. (2004) concentrated on the analysis of memory graphs, and their work will be useful in identifying further advancements to the concept. They were focused on the Java ecosystem, and utilised the HPROF (Heap Profiler) feature of the JVMPI (Java Virtual Machine Profiler Interface). This profiling tool produced a dump that represents the heap of the running program, that they then analysed. The analysis consisted of a number of different features from a variety of different practical Java programs. These features included the concept of unique objects, confined objects and object ownership.

1.4 Aims

The purpose of the project is, therefore, to produce a tool that will facilitate the practical use of memory graphs within software development. The intention is to produce a modern development tool for the Java language in the form of an integrated IDE plugin that will provide real-time access to memory graph related tools. Java was chosen as the base language due to its widespread popularity, relatively simple memory structure and vast array of comprehensive IDE's, with easily accessible debugging and extension mechanisms. By choosing integrated development environments, we can utilise the compact and interconnected nature to produce simple and generic solutions, that do not require external libraries such as HPROF or potentially incomplete analysis based methods.

The are a number of specific features that we will aim to implement. Firstly, we would like to be able to produce memory graphs at every suspension in the code. Thus, the state of the memory graph should match that of other debugging tools within the IDE. This should allow the comparison of states between two breakpoints. Identifying new and changed objects on each suspension is also an important feature to implement.

As well as representing basic memory graphs at key points in the debugging cycle, we would like to research and enable further developments to improve the practical use of such graphs, such as scaling and focusing. One of the primary mechanisms we intend to investigate, for the limiting or focusing a memory graph, is the identification of important or useful objects throughout the code. The idea of important objects is somewhat personal, which provides an interesting challenge. We will be dealing with a very much empirical approach to the problem, and, thus, we would like to investigate solutions that might provide a flexible or responsive solution. In particular, a simple approach that appeals to a best-fit heuristic - a heuristic that might be appropriate for a number of common cases - might be most suited as a initial route of investigation.

Chapter 2

Definitions and Descriptions

2.1 The Java language and the Java Virtual Machine

This project is focused on the Java language and the implementation provided by the Java Virtual Machine. There are a number of features of this framework that it is useful to describe before continuing. A basic knowledge of the structure of object orientated languages, and in particular Java is assumed.

The Java virtual machine deals in two distinct parent types for variables - **Primitive Types** and **Reference Types**. Primitive types refer to values directly referenced - the value is stored in the area of memory allocated to that particular instance of the type. These include in-built types such as numeric types like **int** and **float**, as well as **boolean**.

Reference types involve some degree of indirection - they represent a reference to the memory location holding the relevant instance. In the case of the JVM, references refer to objects created within the scope of the program. In particular, they allow multiple variables to point to the same memory location.

Primarily, we will be discussing reference types within this project, since they most naturally translate to a memory graph structure. However, we do discuss the adaption of the techniques to represent primitive types in Sections 3.2.3 and 4.3.

2.2 Further Application

Although the specifics given are related to Java and the JVM, many of the concepts here could be translated to a large number of different languages. In particular, many object-orientated languages have similar core concepts. C# has a similar system dealing with primitive types (called *value types* in the .NET framework) and reference types. It is, however, important to note that objects are not the only way to represent nested data structures - essentially, we only require a container for references to other containers, or pure data. C *structs* and ML *records* both provide an alternative way of structuring data - data, or references, are stored within the structure, creating a similar container.

2.3 Definitions

The aim of the overall project is to recover, analyse and represent the memory graph from a running program. Informally, a memory graph is a graph representing the memory currently in use by a running program. Each object within the system is represented as a node on the graph. Each variable inside an object is represented as an edge from the variables owner object to the object the variable refers to. In this way, we can create an interconnected graph, representing all structures currently in memory.

We will first give a simplified definition of a memory graph, to provide an understanding for the nature of the system. We also include formal definitions for a number of terms we considered or used through the project. We will then adapt this memory graph with improvements, specifically suited towards Java and our system. Justification and description for these improvements will be provided in Chapters 3 and 4. Although the concepts throughout are broadly language agnostic, the specifics given here are tailored to the Java language and virtual machine. One specific for this first definition is the way that we treat all variables as references, since this simplifies the graph.

2.3.1 Simple Memory Graph

The formal definition given below is a slightly adapted version of that described in Diehl (2007). We have also borrowed from Zimmermann and Zeller (2001), which describes an abstract memory graph for use with C. This is based upon an object orientated language. It is possible to generalise this concept further - to talk about data structures, say, and access paths between them - but this is beyond the scope of this project.

Definition 1: A memory graph

A memory graph is the tuple $(N, E, root)$ where N represents the nodes of the graph (the objects), E represents the edges (the variables) and $root$ represent the root of the graph.

Let S be the group of stack frames from the threads associated with this program. Thus, let V_S be the set of objects represented by the variables in the set of stack frames. Let V_o be the set of objects represented by the variables within object o.

We define the nodes and the edges of the graph as follows:

- $root$ is an arbitrarily created node
- N is the set of all nodes in E.
- $(root, o) \in E$, where $o \in V_S$,
- $(o_1, o_2) \in E$ iff $o_2 \in V_{o_1}$ and $\exists o.(o, o_1) \in E$

2.3.2 Adapted Memory Graph

At this stage, we make a number of adaptions. We add support for dealing with certain types differently. We add support for treating certain types as so called “primitives” - storing them as a separate set of edges. We also include rules for which variables we consider when traversing the existing structure. The folding section describes the mechanism in detail. We also consider the application of importance, by annotating the graph.

2.3.3 Memory Slices

Informally, a memory slice is described as a subgraph, or “slice”, of the memory graph. We can describe a number of different types of slices, based upon what we might find useful.

The first is a forward memory slice. This essentially represents the graph of all objects “reachable” from a given object. We consider a broad definition of reachable, ignoring scope modifiers and simply treating all variable links a traversable. This, we believe, is relevant enough for our purposes. This is useful to see the scope of a given object, as well as easily representing subgraphs in a sensible manner.

Definition 2: Forward Memory Slice

Let o be the object we wish to forward slice on, and M be the memory graph which we wish to slice on, where $M = (N, E, root)$. We can then define the forward memory slice to be a new memory graph of the form:

$$S_F(o) = (N_1, E_1, o)$$

$$\text{where } E_1 = \{(o_1, o_2) | o \rightarrow^* o_1 \text{ and } (o_1, o_2) \in E\}$$

$$\text{and } N_1 \text{ is the set of nodes in } E_1$$

(Diehl, 2007)

We can also define a backward memory slice in a similar fashion:

Definition 3: Backward Memory Slice

Let o be the object we wish to backward slice on, and M be the memory graph which we wish to slice on, where $M = (N, E, root)$. We can then define the backward memory slice to be a new memory graph of the form:

$$S_B(o) = (N_1, E_1, o)$$

$$\text{where } E_1 = \{(o_1, o_2) | o_2 \rightarrow^* o \text{ and } (o_1, o_2) \in E\}$$
$$\text{and } N_1 \text{ is the set of nodes in } E_1$$

(Diehl, 2007)

2.3.4 Object Depth

The depth of an object from a given scope is defined as the shortest number of edges we have to traverse to reach the object - the shortest path. We can define the current "scope" in a number of ways - however, with our current memory graph formulation we would consider it as the distance from the root node.

2.3.5 Object Ownership

An object is counted as the "owner" of another child object if and only if all paths from the child to the root node travel through the owner. Identifying important owners of objects throughout the graph might help in, say, collapsing certain parts of the graph that may be currently irrelevant, or may be summarised by that one owner object.

2.3.6 Other Interesting Concepts

There are a number of other concepts that may be of interest. In particular, concepts related to graph theory - including categorisation and clarification - may be beneficial in focusing or scaling. Strongly connected components are one particularly interesting area of investigation (Cormen et al. (2001) details the technique). Using this, we could attempt to identify parts of the memory graph that are reliant or connected, and thus help focus or hide certain parts of the graph. Whilst this project may not investigate such concepts, they might provide an interesting area for expansion and clarification.

Chapter 3

Creating Memory Graphs in the Eclipse Development Framework

3.1 Analysis and Design

3.1.1 Eclipse

Eclipse is an open-source community whose projects are aimed at creating an open platform for software development. The Eclipse platform provides an integrated development environment (IDE) that has comprehensive support for many languages, including Java, as well as advanced features such as integrated debugging and a complete plug-in development structure. It is also one of the most popular IDE's available today.

One of the features of the Eclipse framework that was attractive for developing this project was the accessibility of the debugging mechanism. This provides a standard API for accessing debugging information for the running programs within the IDE - information that can be accessed from other plugins. It includes access to the stack frames, the local variables in each stack frame, and the contents of each object. This provides clear benefits over more complex techniques - such as requiring external libraries to access the runtime heap or performing code based introspection to discover runtime data. It also allows tighter IDE integration, a feature that aids the ease with which it might be used.

There were a few more practical complexities that had to be considered in any design. Firstly, the Eclipse framework allows for multiple programs to debugged simultaneously. Each program, in Eclipse parlance, is described as a "target". Each target could include a number of threads running at any one time. We had to include a system for identifying each separate target, and devise a way to represent this information to clients.

Each **DebugTarget** represents a collection of threads, themselves representing a collection of stack frames. The stack frames represent the calling history of the thread, and allows the identification of all objects currently within the scope of the running thread. A stack frame provides access to the variables in that stack frame. Each variable contains a link to the value the variable represents and some contextual information, such as the meaningful identifier given to the variable by the programmer. Recovering the value of the variable facilitates the recursive identification of new variables, and thus the memory graph.

3.1.2 Architecture

One feature the debugging API does not provide is a direct mapping to our intended representation - a memory graph. The project itself was therefore one of two halves: firstly, a plugin that exposed the debug mechanism as a memory graph (hereafter called the DebugModel plugin), and secondly, a plugin that provided a visual representation of that graph. This report describes the former, and so only passing mention will be made about the visual representations.

We decided to use the MVC (Model-View-Controller) framework - segregating the construction of the model from the view in such a way that we would be able to develop each independently, and so that the model itself is reusable with other views. A diagram showing our overall architecture is included in Figure 3.1. This allows far more flexibility than any method that combined the debugging section with the view itself. The uncoupled nature of the two sections will allow further development on different forms of view at a later point in time.

Another benefit of implementing loose coupling between the view and the model is that a number of views can be used simultaneously. This would potentially allow the development of specialist views, as well as general purpose interfaces. This might include views tailored to particular contexts - or highlighting specific areas - or adapted for particular user interface mechanisms. This will potentially increase the utility of the DebugModel.

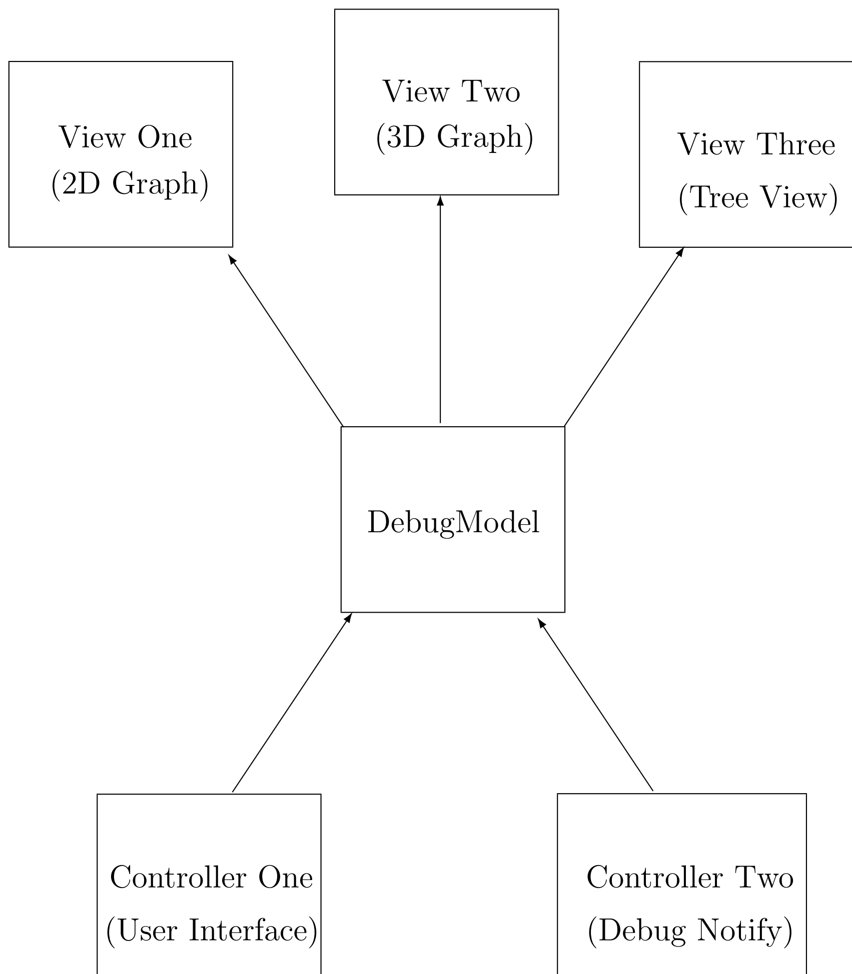

Figure 3.1: MVC Architecture used in the system. Includes some examples of the kinds of view and controller that would interact with the DebugModel.

3.1.3 Design

The brief, therefore, was to build a plugin that would encapsulate the existing debug framework, exposing a memory graph via an API that would allow any view to register, and thereafter get notification updates.

A number of adaptions were necessary due to the nature of the pre-existing Eclipse debug framework. The API provides no mechanism to discover objects being instantiated or garbage collected, and so we had to rely on querying the API at regular intervals. Since the likely use case is similar to other debugging tools, we considered that it would also be used during suspension - caused, perhaps, by something like a breakpoint. However, we acknowledge that a more complete solution might involve monitoring the heap in a more direct manner, which would improve the efficiency of the analysis.

The design of the memory graph adapter therefore became clear. We would write a plugin that stored a memory graph, which would updated when notification of suspension arrived from the Eclipse debugger. Our aim throughout was to provide a flexible framework for memory graph creation, that could then be adapted to include further enhancements to the original concept.

The chosen architecture is shown in the simplified UML diagram (Figure 3.2). We represent each target as a separate memory graph. Each memory graph is wrapped in a container, defined as a **DebugModel**. Each DebugModel consists of a number of DebugObjects, each of which represents an object within the scope of the threads of

that target. The DebugModel is responsible for updating itself - given a set of references to the current stack frames of the threads of the target, it rebuilds the memory graph (in the *updateModel(..)* method). *updateModel(..)* uses the private utility method *addObject()* to add an object based on the value returned from the Eclipse Debugger (an *IJavaValue*). DebugModel also provides access to the memory graph directly, via *getObjects()* and *getObject(..)* - the latter working on an IJavaValue taken from the debugger to provide a memory graph node - a **DebugObject**.

DebugObjects provide access to the nodes of the memory graph. As well as providing access to the raw debugging objects, via *getValue()*, we also provide direct access to the edges available to/from this object, through *objectLinks()* and *backLinks()*. The DebugModel stores the DebugObjects within its particular purview - any running threads within the target it represents. As well as providing various methods of accessing these objects, it also provides utility links for discovering information about the model as a whole - such as the changes from the last debugging suspension point.

The glue for the system is the DebugModelContainer, which is a singleton (see Gamma et al. (1995) for details) that registers itself with the Eclipse DebugPlugin. It deals with incoming messages from the DebugPlugin, via the *handleDebugEvents(events)* method. This then updates the relevant models, via the *updateModel(..)* method on each model, and then notifies any interested parties with details of the updated graphs. It includes a listener interface, such that an interested party can register itself via the *addListener*(..) method, on the singleton instance. It also provides the point of interface between the controllers and the memory graphs themselves.

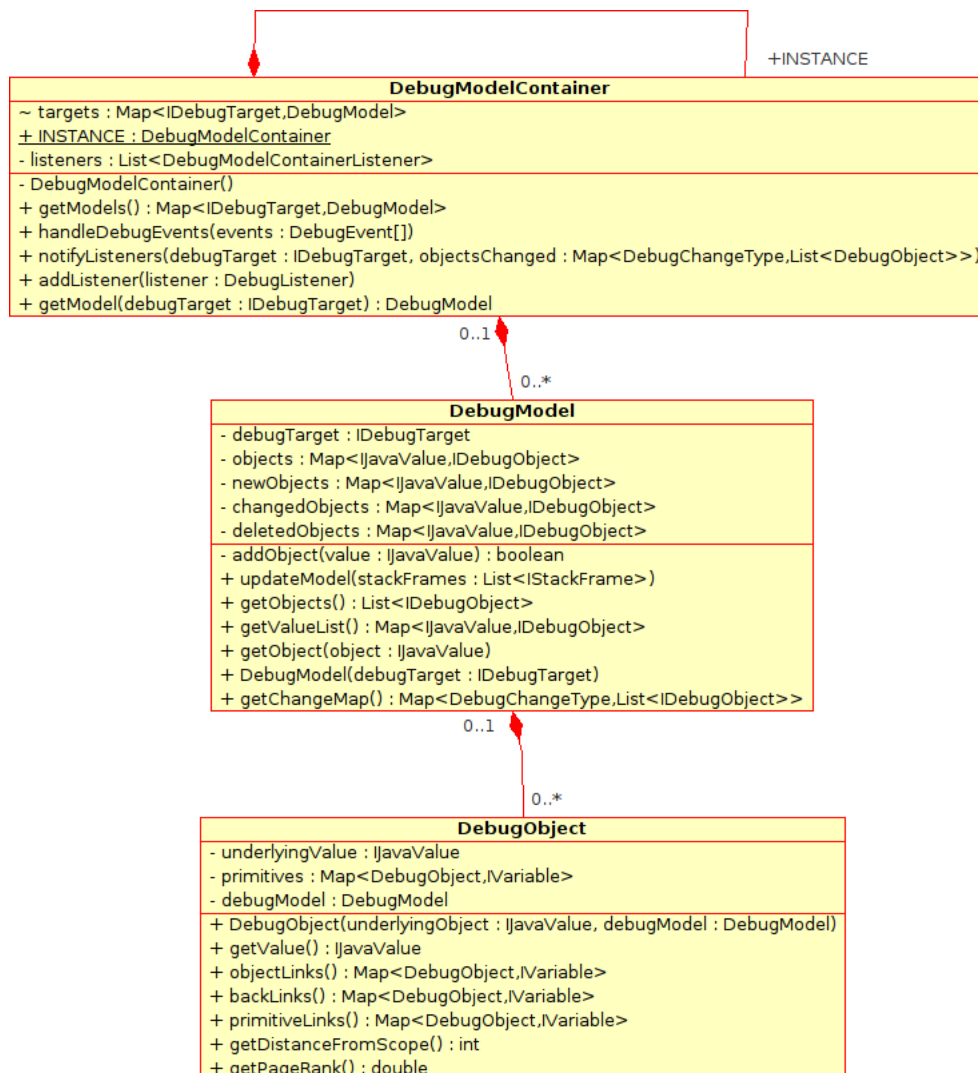

Figure 3.2: UML diagram representing the class design for the DebugModel

3.2 Implementation

3.2.1 Creating a memory graph

The Eclipse API provides a mechanism where we can register for notifications of the change of the program state. These include notifications of suspension, termination and continuation. This provided our DebugModel plugin with a notification every time the program was suspended. Upon suspension, we can then "scrub" the API to produce a new memory graph.

This involved searching through each stack frame in order, identifying variables within the stack frame, and adding each object to our model. The model is then produced recursively - each object, when added, recursively adds the objects pointed to by its own instance variables. The model then records the link between the object holding the variable, and the object that was pointed at. Figure 3.3 provides a simple pseudocode description of the algorithm.

```
def createNewModel(stackframes):
   model = new Model()
   foreach stack frame in stackframes
      foreach variable in stackframe
         model.addObject(variable.getObject)

class model
   objects = List<objects>
   links = List<(object,object)>

   def addObject(object):
      if (!model.contains(object)):
         foreach variable in object.getVariables
            addObject(variable.getObject)
            links.add(object,variable.getObject)
```

Figure 3.3: Simplified pseudocode algorithm for creating a memory graph

3.2.2 Comparing memory graphs

At this stage we have produced a single model for a given suspension point. However, we would like the Debug-Model to provide a new model with each further suspension point. As well as providing this new model, we would also like information about how the model has changed between each step. Clients of the DebugModel will find it useful to be notified of which objects have changed, which have not, and which were added in the last step. This type of information is certainly something a view might like to display to the user - for example, it could be used to highlight, or even only display the objects that have changed. This allows a user to visually compare the state before and after a function call.

To formalise each consideration we define:

- the set of **new objects** as those that did not previously exist in memory.
- the set of **changed objects** as those that already existed, but the variables of that object have changed. As the variables represent the state of the object, this will represent the state of the object changing. Essentially, this represents a subgraph (or a number of unconnected sub-graphs) of the original graph, including only those objects that have changed.
- the set of **old objects** as those that already existed, but no variables have changed, and who continue to exist.
- the set of **deleted objects** as those that no longer exist in our memory graph.

There are a number of techniques for calculating this information. We can build our new model, as described above, and then compare the models step by step - marking off each object against the old model as we go. This

is straight forward, but inefficient as we increase the size of the graph.

A second method would be to simply update the model in place. By comparing each object we find in our search of the API's against the existing model, we can calculate and store the relevant sets of data. This provides a far more efficient mechanism for updating the model, as shown in Figure 3.4. Essentially, we call our update method on the model object, which in turn stores the previous list of objects, adds new objects one at a time, comparing them with the old model.

```
def updateModel(model,stackframes):
   model.update(stackframes)

class model
   // stores all objects
   objects = List<objects>
   deletedObjects = List<objects>
   changedObjects = List<objects>
   newObjects = List<objects>
   links = List<(object,object)>

   def addObject(object):
      // Calculate whether this is an existing, changed
      // or new object
      if (object is in deleted):
         if (object is not identical to deleted):
            changedObjects += object

         deleted -= object
      else:
         newObjects += object

      // Consider reciprocal links
      if (!model.contains(object)):
         foreach variable in object.getVariables
            addObject(variable.getObject)
            links.add(object,variable.getObject)

   def updateModel(stackframes):
      move objects to deletedObjects
      changedObjects = new List<objects>
      newObjects = new List<objects>

      // Check each stack frame in scope
      foreach stackframe in stackframes
         foreach variable in stackframe
            addObject(variable.getObject)
```

Figure 3.4: Simplified pseudocode algorithm for creating a new memory graph by comparing to an existing memory graph

A third possible mechanism might be to store only the iterative values for each step. By storing only new, changed and deleted objects at each step - together defining a changeset - we can build the current model by combining each atomic changeset from the start. Storing each of these changesets would be possible, but would come with some technical challenges. In particular, efficiency might become an issue as we delved further into the program - building combined graphs would become computationally complex. However, it would allow for composite changes to be created (simply by combining the changesets of each suspension), which would improve the ease with which we could compare different points in the program.

We have chosen to implement method two, primarily since it provided an adequate balance between performance and flexibility, whilst still being relatively simple to code. However, further research might consider the third algorithm to be either more flexible or indeed have improved performance for certain operations. Once the system is fully functioning, empirical tests might consider benchmarking the various mechanisms against large or common programs, in addition to some calculations on the nature of the efficiency of each operation.

3.2.3 Primitive values

The Java Virtual Machine (JVM) provides two distinct methods of representing values, as mentioned in Chapter 2. Values can be represented as objects, stored on the heap, and referred to by references or stored by value, in situ. The code we have given above only deals with objects - we do not deal with the case of primitive values.

For our initial construction, we decided to ignore primitive values, to produce a simplified initial program. After producing an initial solution, we expanded it to include primitive values. We implemented a mechanism whereby we treated primitives simply as objects with no variables. However, this did have its downsides - each number, character or other primitive used in the program was represented as an object and, in a large program, many may be used. This creates a rather more complex graph than previously, with many unhelpful links. In particular, if two objects have the primitive "2" as a variable, both will now include a link to the number "2". Intuitively, we would expect this to be unnecessary - whilst we would like to be able to represent values, we maybe do not want to consider them as part of the complete memory graph. It is useful to note that this problem is not necessarily limited to primitives - primitive wrappers (such as the Integer, Double etc. classes) also have this issue.

Our initial solution was to provide a configuration option, allowing the user to decide whether to view primitives or not. However, we developed a number of others solutions to this problem, which we describe in detail in the next chapter.

3.2.4 Model Interaction

All this information is no good in isolation. We must provide a mechanism to allow any view to receive a notification when the model has been updated. For this, we use the Observer pattern (see Gamma et al. (1995)), which allows any number of views to register with the model, and thereafter receive updates. This is ideal, as we want to be able to run multiple different views from the same model.

We also provide interaction through a number of different controllers. Each view may implement a controller specific to that view. By interacting with each model, the view can update the state of the model on the fly. This might include functions for limiting the graph size, such as those described in the next chapter. We also provide a preference controller, which allows global configuration changes - settings which will apply across all models. For this we have used the Eclipse in-built preference panel features. By providing configuration options in a conventional place, we improve the usability for the user.

Chapter 4

Making a memory graph more usable

As we have previously commented, there are a number of usability problems with the existing representation of memory graphs. One of the primary problems with the method described so far is with the sheer size of the graph space involved in anything more than a trivial program. Even moderately sized programs will have thousands of objects - and larger programs will range into the hundreds of thousands of objects. (Potanin et al., 2004)

A secondary problem is that of lack of clarity amongst the displayed objects. Our graph is easily muddled by primitive values, library data structures and other language complexities. Ideally, we would like to allow the user to abstract away from these complexities, and provide a clear notion of the intention of the program - an intention that might otherwise be obscured by the quirks of the language.

It is important to note that there is no "silver bullet" when considering improvements to the performance or usability of a memory graph - there is no single solution to solve either of these problems. The problems primarily span from the complexity of the source of the information - the program itself. Attempting to place order on top of a complex graph, representing many nodes, will always be conceptually challenging. Therefore, what we must provide is a number of smaller changes that, when combined, may improve the usability. In particular, we wish to reduce the complexity by aiding the user in ordering and focusing the vast amounts of data.

4.1 Focusing and Scaling

One of the easiest ways to tackle the vast-space problem is to minimise the portion of the graph you wish to visualise. Essentially, we are trying to manage the users screen estate, either by limiting the focus, or by drawing the eye to certain objects. We can do this in a number of ways. Either we can reduce the graph space (scaling) - either automatically, or on user input - or we can identify objects the user might want to look at (focusing).

A simple method would be to introduce something like a depth traversal limit, where we only traverse so many objects. This, however, is a quite imprecise measure - what if we remove important objects? Another way might be to utilise specific slices, allowing the user to select an object, and perform a forward-slice or backward-slice. (See 2.3.3 for definitions of forward and backward slicing). This would certainly provide a useful focusing mechanism. But how might the user identify which objects they wish to focus on?

A simple method we implemented was to identify recently changed objects. The DebugModel provides each view with a list of deleted/changed/new objects on each iteration. These are undoubtably helpful - if we have created a breakpoint at a certain line, then we may be interested in seeing the effects of that line, and recovering the changed/new objects will certainly aid that. The view can then relay these to a user, allowing them to identify objects they are likely to want to focus on.

4.2 Identifying Importance

4.2.1 Motivation

So far we have only described methods that either require naive limitations or specific user input. Thus, we would like some method for identifying relevant parts of a program, which in turn allow us to expand and highlight useful sections for the user.

It can be assumed that not all objects are equal in importance. The definition of importance is somewhat vague and empirical, since we are relying on human judgement. It is sufficient, for our needs at this stage, to define an important object as one the user is likely to want to inspect. Due to this personal nature of importance, we would also like to be able to configure the concept for different users. In essence, we wish to provide a heuristic of some description - a tool that might help the user navigate vast or complex memory graphs in a simple and quick fashion. As such, our aim is not so much to provide a thoroughly accurate measure, but to simply provide a best-fit metric for a number of common use-cases.

4.2.2 Implementation

We investigated a number of factors when we were determining relevant objects. A number were directly related to the graph like nature of the model - including links to and from objects, as well as the relevant importance of those objects we link to and from. Depth from the current scope is another, as well as concepts of ownership and confinement (as described in 2.3.5).

One algorithm which uses the first two factors (links and their relative importance) is called PageRank, developed by Larry Page and Sergey Brin, founders of Google (Brin and Page, 1998). Whilst it may initially appear that web-search and memory graphs have little in common, they both deal with interconnected graphs. Each has to identify important objects, purely based on the links of the web. And so we have adapted the PageRank algorithm, originally designed for indexing the relative importance of web pages, to identify the importance of objects.

Our first attempts used PageRank in its original setting. PageRank essentially gives each page in the web a score - its PageRank - that represents the likelihood of a random web-surfer arriving at that particular page. We initially set the probability distribution across the pages as equal. We then perform an iterative evaluation of the probabilities until we reach convergence. We consider each page in turn, consider the links to it, and the current importance of those pages it links to, based on the following formula:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}$$

Where B_u are the pages that link to u, and N_v is the number of links from v. Essentially, we distribute an objects importance amongst all its links equally. See Page et al. (1998) for a more detailed analysis of the technique.

In our attempts we consider the PageRank of an object to be the chance of a random thread visiting that object. Whilst this is slightly more contrived than in the context of the web, we still believe that it will provide some useful statistics.

The problem with PageRank in our particular situation was that important objects commonly held references to other objects, but let very few objects hold references to themselves. This, in turn, lead to a small PageRank score, since PageRank relies on the importance of objects that link to the object. We decided to adapt the algorithm to represent this - instead of considering objects linking *to* the object, we consider the link *from* this object to others. Our intuition behind this was thus: for an object to allow a reference of itself to be given to other objects, it must consider itself an important object. This adaption appeared to work well in practice, and, in particular, seemed suited to many tree based data structures. The success of this technique is discussed in Chapter 6.

4.3 Folding

We encountered a number of problems when building and traversing memory graphs naively. Traversing primitive values created over-crowded and unclear graphs. Traversing in-built data-structures often revealed much detail that the user might wish to be obscured.

We also considered that the graph was, in many ways, inflexible in its approach to displaying algorithms. As a general, rather than specific, tool, we would like to provide visualisations for a wide range of different types of common algorithms and data-structures. Whilst we still wanted to keep the graph approach, as it provides an excellent, general tool for visualisation, we wanted to improve the nature of the graph.

We developed a general solution to these problems, one we described as **Folding**. We decided to classify each object, depending on its specific role within the program, and thus the memory graph. Classification of objects is in one of the following five categories:

- **Recurse -** a standard object. Traversed as described in the original algorithm.

- **Ignore** - an object we always ignore. Never added to the memory graph, and never traversed - so, in particular, variables within the object are never considered. Essentially removes the object from the memory graph altogether, along with any objects only accessible from this object.
- **NoRecurse** - an object we add to the memory graph, but never traverse. Examples include wrappers for the primitive types - such as *Integer* and *Double* - as well as the *String* class (since all include implementation details that are generally unnecessary to view).
- **Primitives** - operationally, identical to **NoRecurse** objects. However, separate classification allows them to be treated differently by the model and therefore the view. Will allow the view to only display them in certain circumstances, for example.
- **Fold** - This class allows us to ignore certain children of the object, considering instead only the grandchildren from that child. Each link from the child is treated as a link from the current object. This essentially has the effect of removing the child object from the memory graph, based upon the type of the parent and the child. This definition is recursive, so if a child is defined as "folded", it is folded into the parent fold object. In this way, the parent object might include links to great-grandchildren, or even deeper. By default we fold every child, but we also provide a mechanism for describing which children to fold based upon their types.

In practice, these categories allow us to transform the memory graph as we are creating it, with each category having its own transformation.

We decided to model the categorisation of objects as application rules, based on their type - if the type of the current object matches a given rule, we give it the matching category. Rules are of the form:

```
foo.bar.typetomatch -> [RECURSE|IGNORE|WRAPPERS|PRIMITIVES|FOLD(SEQ types)]
```

Rules are considered one at a time, and the first matching rule is chosen. Section 4.3.1 describes a worked example that considers LinkedLists. and gives some example rules. As well as these type-based rules, we also consider some in-built optimisations. This includes the way we treat primitives, in particular. A basic, pseudocode, definition of type folding is given in Figure 4.1. Essentially, for each object we calculate a type category, and use this to decide whether we need to recurse. We then consider each child in turn, based upon the type of the parent and type of the child, folding where appropriate.

To provide the most flexible solution, we decided to make this configurable. We used the standard Eclipse property mechanism to store and retrieve configurations. We provide some built-in configurations for dealing with a number of standard Java types - and in particular Java utility libraries, such as LinkedList. As well as these standard configurations, we provided the user with an Eclipse preference pane, as described in the previous chapter, for adding new type folding rules, and modifying existing ones.

4.3.1 Example

The abstract application rule definition of LinkedList is given below, as an example of this kind of type folding.

```
java.util.LinkedList<E> -> FOLD
java.util.LinkedList$Entry<E> -> FOLD(java.util.LinkedList$Entry<E>)
```

Working through the example might clarify the process. If we have a LinkedList that represents an abstract list of three items: [$Object1$,$Object2$,$Object3$], this will be represented in memory by the memory graph given in Figure 4.2.

Each item in the list is represented by a LinkedList$Entry⟨E⟩ object, that in turn points to the next Entry item in the list. However, we wish to abstract away this messy detail, and so we use folding to produce a memory graph of the form shown in Figure 4.3.

We have removed each LinkedList$Entry variable, and folded each object, to link to the LinkedList⟨E⟩ object. We do this in a recursive fashion - first LinkedList⟨E⟩ calculates its type, in this case "Fold". Since fold requires to ignore any children, we take each child in turn, and ask it only for its children. In this case, we look at the first LinkedList$Entry⟨E⟩. It identifies itself as a "FOLD(java.util.LinkedList$Entry⟨E⟩)", which means it folds only on entries of the given type. So, in this case, it creates a link between the root and $Object1$, and then recurses to look at the next entry object, still with the same root. It recurses like so until it reaches a no-fold object, when it

Pseudocode Explanation

We firstly modify the *addObject(..)* routine to identify a category, and extract any code dealing with recursively identifying new objects to a new method, *calculateLinks(..)*. This facilitates the identification of folded types.

```
def addObject(object):
   ...
   // We extract the type category of the object
   // Initially a simple pattern matching against a series of rules
   // But, in future could include
   typeCategory = getTypeCategory(object)
   ...
   // Identify whether this object is in a recursable type category.
   if (typeCategory == RECURSE || typeCategory == FOLD):
      calculateLinks(object, typeCategory, object.getVariables())
```

calculateLinks(..) is a utility method which allows the algorithm to remove an object, and recurse considering parents of the object as the new parents for the children of the removed object. Once we have identified an object to recurse over, we call the *calculateLinks(..)* method.

```
def calculateLinks(rootObject, typeCategory, variables):
   // Check each child in turn
   for object in variables:
      // Acquire a child category
      childTypeCategory = getTypeCategory(object)
      // If we fold over this pair of child and parent types
      // e.g. if we wish to remove the link to this child,
      //  and replace it with links to the grandchildren
      if (typeCategory == FOLD && childTypeCategory in typeCategory.typesToFoldList()):
         // We effectively ignore this child, and treat its children
         // as variables of the root object, but with the category of the child object
         calculateLinks(rootObject, childTypeCategory, object.getVariables())
      else:
         // Either the type category is RECURSE, or we don't fold on this type
         addObject(object)
         // We treat primitives as a different kind of link on the graph
         if (childTypeCategory == PRIMITIVE):
            addPrimitiveLink(rootObject,object)
         else:
            addLink(rootObject,object)
```

Figure 4.1: Simplified pseudocode algorithm for type folding

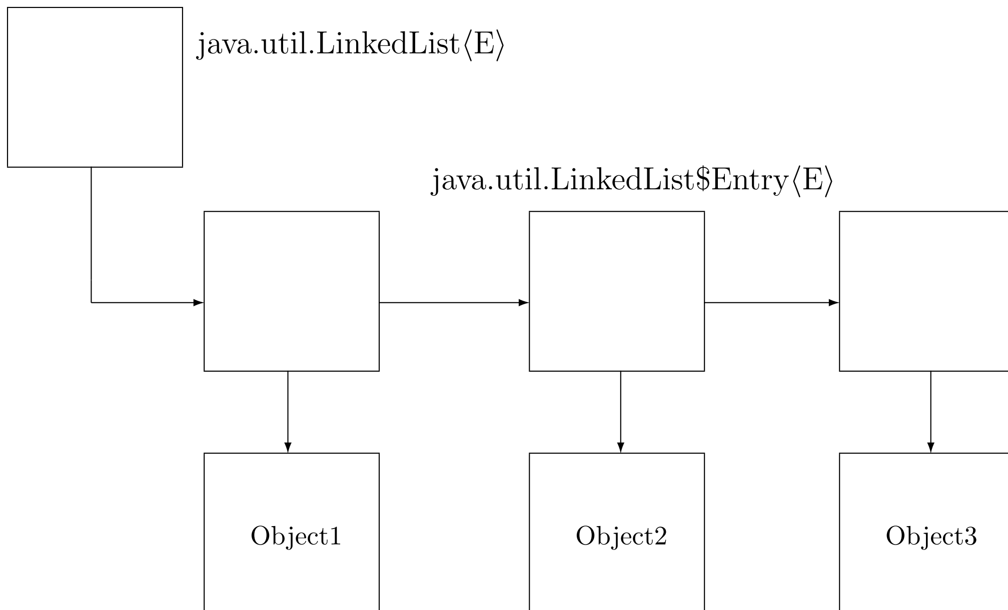

Figure 4.2: LinkedList before folding

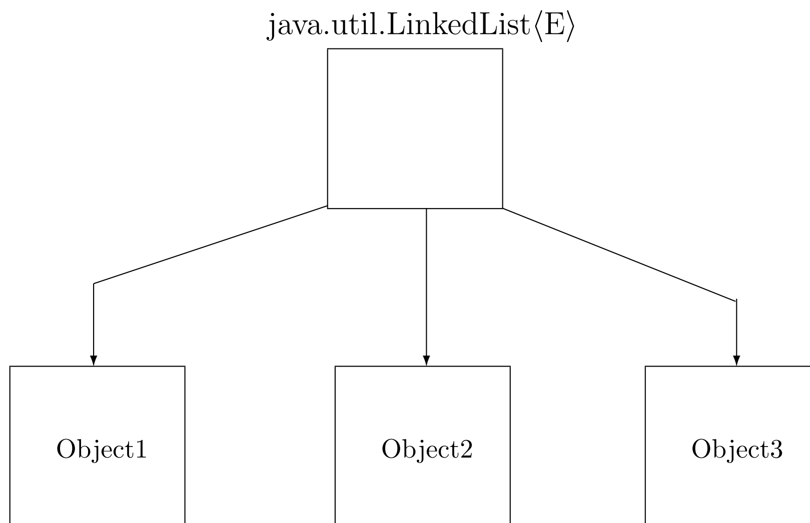

Figure 4.3: LinkedList after folding

returns the list of objects found.

This feature provides a useful mechanism for abstracting away from the details of the implementation of the LinkedList, whilst providing the important semantic content of the object.

Chapter 5

Testing and Development

5.1 Code-based tests

Unit tests were developed for each module, as and when it was written. A unit test attempts to stress a particular class or unit, through the creation of a series of individual tests. Each might test a method or a common usage of the class. This is a challenging procedure when dealing with an external API, such as the debug plugin, since there are few simple ways to feed real input into the system. As such, we utilised test data injected into the DebugModel to run tests.

Whilst a unit test cannot prove the absence of a bug, it can prove the existence of bugs. Therefore, we aimed to create tests that strained the system - tests that use the so-called edge cases to profitably find bugs within the system. As well as that, we aimed to test the common use cases of the system, thus providing some confidence that the system works as expected.

As much of our development was iterative, we made heavy use of refactorings within the system. As such, unit tests provide a useful measure for considering whether a refactoring had introduced any new bugs.

Another tool we made use of was FindBugs. FindBugs is an Eclipse plugin that performs static analysis on your code base, along similar lines to a compiler. However, it performs more detailed semantic analysis of the code to discover either bugs or areas that have the potential to lead to bugs. This tool provides another layer of automated testing, one that provides an extra safety net for identifying bugs.

5.2 Empirical Testing

Providing visual feedback during development is vital. To ensure proper separation of features, and as part of the independent testing of the framework, we provided a simple view based upon the Eclipse tree structure. This allowed some visual feedback on the state of the model during development. We include an example screenshot in Figure 5.1. It represents each object in the memory graph, with its children in the graph. Whilst it is only a simple representation, it does provide a basic interactive feature for inspecting the current state of the model - particularly useful for debugging and development.

5.3 Functional Tests

As well as testing using unit tests, we also provided some functional test cases. These included example programs that represented use cases for the software that we had previously considered - including the representation of some specific data structures, such as trees, LinkedLists and other, more complicated, structures. Further description of the programs we used to test the system is included in the Evaluation chapter.

5.4 Trac and Subversion

An important part of the development was the use of a number of tools designed to aid software development. The first, a version control system, provides many benefits. All files relating to the project were stored in the version control system. This allowed any developer to perform a clean checkout of the whole project during development, and have it “just work”. It also allowed regression back to previous versions if a bug was found. Another feature

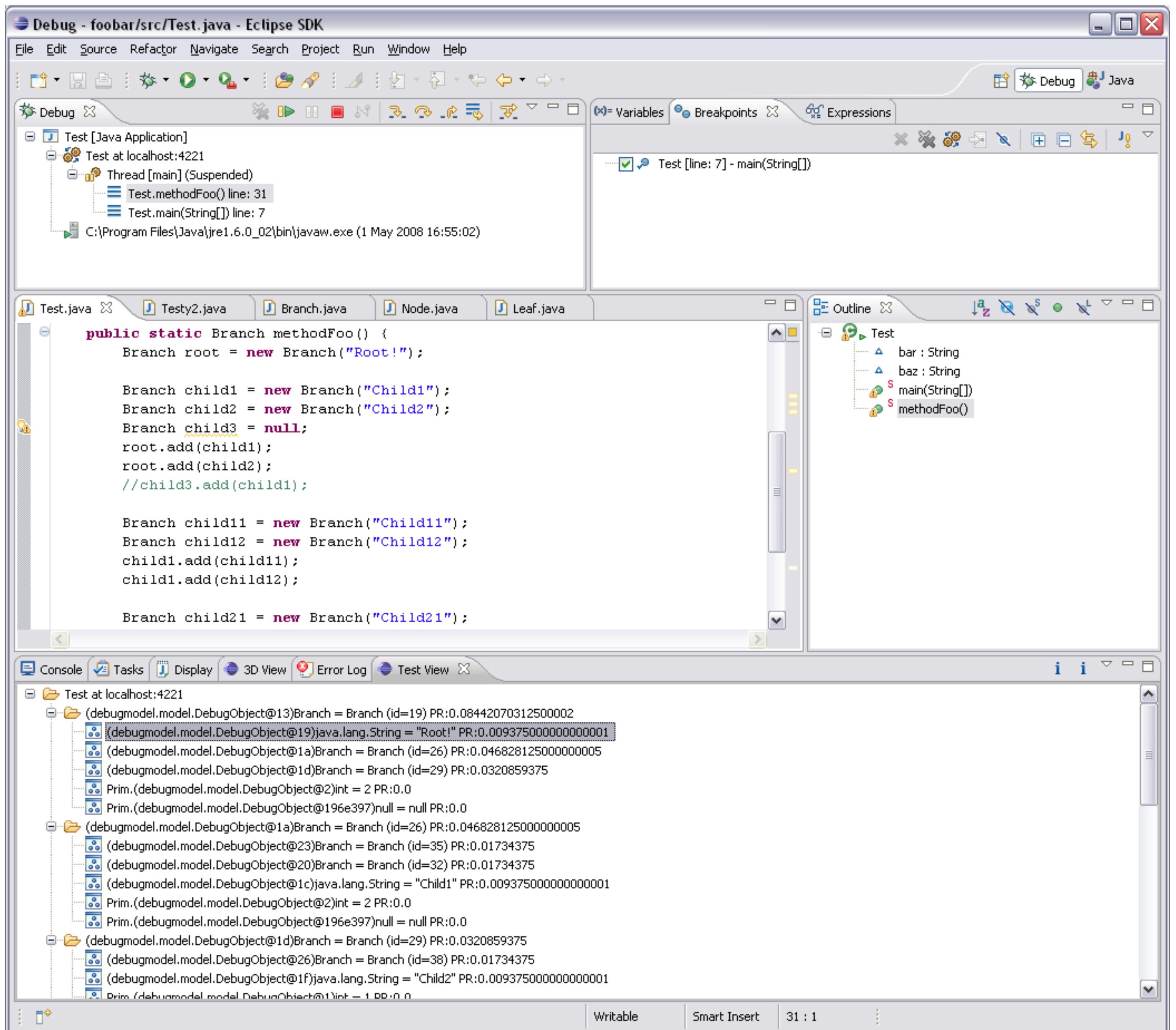

Figure 5.1: Screenshot displaying the simple tree-based memory graph view.

was to identify the state of the software by version number. This allowed bugs to be filed against a particular version, to aid in the correction of that bug. This was particular important, as other people were developing against the API provided by the DebugModel. Our chosen version control system was Subversion, which applies a client-server mechanism, allowing the code to be checked out on any number of machines.

We also used Trac, a combined bug-reporting, wiki and version control interaction system. This allowed anybody to file bugs against any part of the project, as well as providing an area to document the API for the DebugModel, as well as documenting feature ideas, bugs, setup information and other useful information. It also provided task management, which aided in communicating information about when features might be completed.

Chapter 6

Evaluation

6.1 Test Programs

As part of the development of the software, we created a number of test programs, designed to explore the utility of the software. Many of the hand-crafted programs took inspiration from common design patterns, particularly those from the Gang of Four book (Gamma et al., 1995). Patterns provide a rough estimate for templates that are often seen, or constructed in code - they are designed to represent solutions to commonly seen problems. As such, they provide an ideal first stop for a series of test data and control structures. A number are described below as a way of illustrating the evaluation of the techniques that were used.

Screenshots displaying 3D models are taken using a 3D memory graph viewer created by Darius Bradbury, and shown for illustrative purposes only.

6.1.1 Tree

Many programs rely on some form of tree as a core data structure. They are both widespread and useful in many contexts, so it became an obvious choice for the contents of a test program for the system. The program creates a standard tree structure. It includes an interface, **Node**, which is implemented by **Branch** and **Leaf**. A branch may have a number of children, all of which must be **Nodes**. **Leaf**'s have no children. Both **Branch** and **Leaf** contain a String representing the name of that node.

Using this construction, a test program of the following form was created:

```
01 public static Node methodCreateTree() {
02         Node root = new Branch("Root!");
03
04         Node child1 = new Branch("Child1");
05         Node child2 = new Branch("Child2");
06         root.add(child1);
07         root.add(child2);
08
09         Node child11 = new Leaf("Child11");
10         Node child12 = new Leaf("Child12");
11         child1.add(child11);
12         child1.add(child12);
13
14         Node child21 = new Leaf("Child21");
15         Node child22 = new Leaf("Child22");
16         child2.add(child21);
17         child2.add(child22);
18
19         return root;
20 }
```

This creates a simple tree structure, with a root ("Root!") with two children ("Child1" and "Child2"), each with their own two children ("Childx1" and "Childx2"). The standard memory graph for this program, at the point we return the root (line 19), looks something like the graph given in Figure 6.1. It is important to note how

the messy implementation details required by the implementation of Node reduces the clarity of the graph. In particular we see the structure, in this case an array of Nodes, used to store the list of child objects.

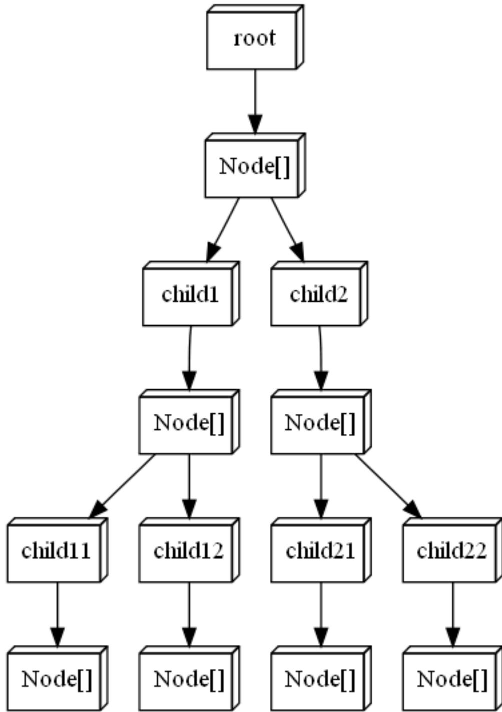

Figure 6.1: Initial, naive memory graph produced from the tree program.

This is the standard memory graph, without the added features detailed in Chapter 4. When we add folding functionality, we can remove the untidy implementation details. As a first step we can remove the implementation of the LinkedList, and have all the objects in the list as children of the original LinkedList object. This is part of the default configuration of the system. However, we can provide some application specific configurations. For example, in this case we may only wish to be informed about the overall structure of the tree, rather than the specifics. So, we can configure the system to ignore LinkedLists as children of **Branch**es altogether. In this way, we can display purely the structure of the tree, through only the constructed classes. The transformed graph will look rather more like the Figure 6.2.

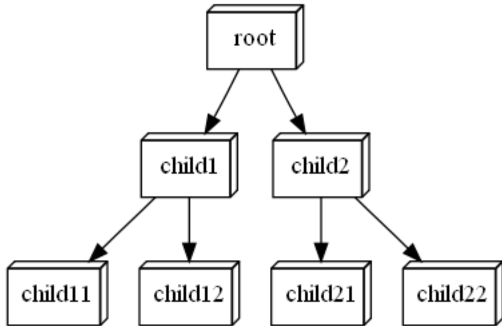

Figure 6.2: Folded memory graph produced from the tree program.

A final feature to discuss is the value of the importance metric for this test structure. The relative scores calculated for the given tree are given in Table 6.1. It easily identifies the root node as the most important in this case - a reasonably natural assumption, given the tree in question. Whilst it appears to produce reasonable data for this kind of graph, it is questionable whether a tree on this scale provides a large enough test set to challenge the system.

Object	Importance
Root	0.372915
Child1	0.180120
Child2	0.180120
Child11	0.066711
Child12	0.066711
Child21	0.066711
Child22	0.066711

Table 6.1: Table representing the importance of various objects in the folded tree example.

6.1.2 Composite

The tree test program is a useful benchmark for measuring the utility of the system. In particular, it has a wide range of adaptability. Many programs use the concept of tree type data-structures - in particular, the representation of part-whole hierarchies. The Composite pattern (see Gamma et al. (1995) for a detailed description) is a classic example of a pattern that utilises tree structures. It is commonly used within GUI frameworks, such as Swing, as well as a broad range of other domains.

Briefly, the pattern consists of a series of *Components*. A component can either be a *Composite*, in which case it stores a list of other components contained within itself, or a *Leaf*, which is a terminating object. For a test program, we have modeled a simple HTML document as a composite structure (with tags such as ⟨html⟩, ⟨ul⟩ and ⟨body⟩ modeled as containers, and tags such as ⟨br⟩ modeled as leafs). This example is designed to represent an easy to understand representation of the Composite pattern, and does no necessarily reflect either correct HTML or the best way to model it.

I modeled the following page as a composite:

```
<html>
<head>
<title>Title</title>
</head>
<body>
<h1>Title of the Page</h1>
<ul>
<li>List Item 1</li>
<li>List Item 2</li>
<li>Click <a href=''http://www.google.co.uk''>here</a> for google</li>
</ul>
<p>More Information</p>
</body>
</html>
```

After running this program, we identified a number of optimisations we could make with regard to the tree structure. Again, we could "fold" away the implementation of each node, by removing data structures such as lists from the graph. This left us with a pure composite tree that represents only the data of the program. We achieved this by applying a simple textual regular expression - removing all lists that were children of any objects belonging to the package "tags".

Using a 3D view to display the page structure, we produce this overall graph (Figure 6.3), when focused on the HTML element tag. It provides a nice, simplified view of the data-structure, stripped of unnecessary information, and providing clarity to the previous busy graph.

6.2 Utility

Essentially, by creating a memory graph, we mainly aim to deal with composition. **Composition** can be described as a data structure that is *composed* of a number of other structures. We can compose items in two separate ways - in a parent-child relationship, or a peer-peer relationship. The test programs dealing with trees and the composite pattern are classic examples of parent-child relationships. Our particular composite example deals with something

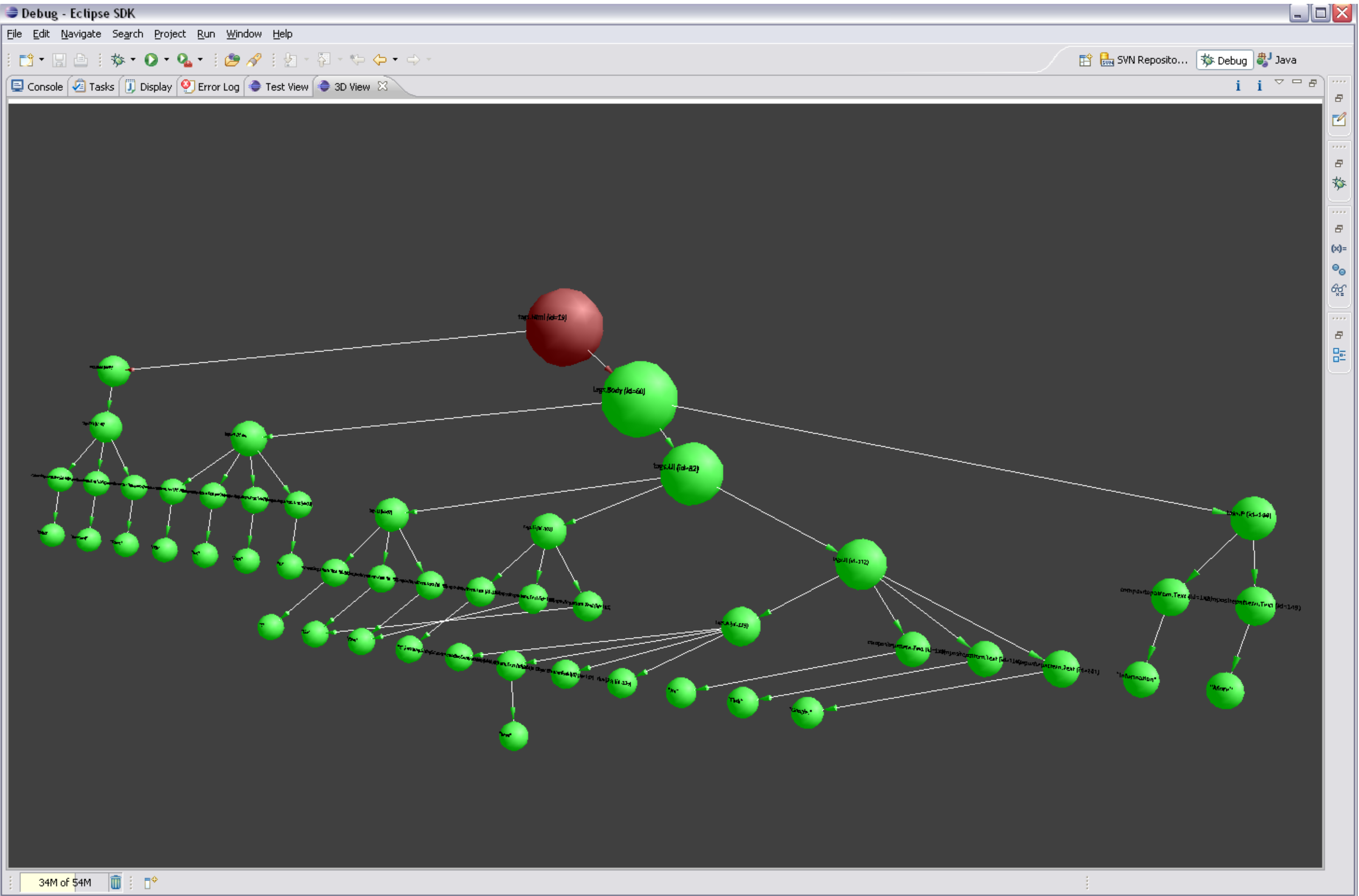

Figure 6.3: 3D graph representing HTML page

that is, in particular, a part-whole relationship.

The folding techniques provide a clear and useful way of extracting information from a busy graph - something that is adequately displayed by both example programs. It is fairly clear that folding techniques certainly provide benefits on graphs of this scale, and would be expected to provide even greater benefit on larger graphs. In particular, any tool that reduces the quantity of objects within the system without reducing the clarity of the representation will always prove to be an excellent tool.

The relative importance of objects has proven to be somewhat trickier to analyse. Whilst the importance metric appears to produce sensible results for the small test-programs provided, we have little evidence about the relevance of such a metric over a larger system. We aimed to provide a useful heuristic that might guide the user to important part of the programs, and as such this appears to provide such a solution. Whilst it may not fit all solutions, we suggest that it provides a sensible first step for this type of use. However, further development, and, in particular, further analysis, might reveal far better (if more complex) heuristics.

To some extent, the utility and functionality of all these features will depend on the particular implementation of the View that is utilising the data. The intention when providing access to these memory graphs is to provide an API such that writing intuitive and flexible interfaces to memory graphs, in a simple fashion, will become possible. The techniques above appear to provide functionality which will certainly have some utility in the creation of such interfaces.

Chapter 7

Conclusion

7.1 Summary

In conclusion, the project has achieved the majority of the aims that were set out in the original brief.

A comprehensive plugin for the Eclipse framework has been created, that integrates with the existing debugging facilities to provide a tool for accessing and manipulating the memory graphs of a running program. This includes the creation of an accessible API, which allows any client to create a suitable view for the purposes of displaying and interacting with the memory graph.

Furthermore, there have been a number of extensions to the concept developed, that provide some flexible and adaptable ways of focusing, scaling and manipulating memory graphs. We have completed some initial research on the identification of important objects, and whilst the work has been far from exhaustive, methods have been developed which seem to provide some utility in a number of cases. Further mechanisms have been developed for obscuring, hiding or removing certain parts of a memory graph through a method we have described as **Folding**. This folding not only provides some utility for removing standard classes, it also provides extensive user customisation. This kind of customisation seems a sensible, and necessary next step - allowing flexible algorithm visualisation, and making the whole concept more usable.

The overall design chosen for the system appears to have been successful. The MVC architecture chosen provides a clean and loosely coupled interface to the memory graph. The overall structure has also provided a flexible system for adapting to the various extensions that were developed over the course of the project. The design of the DebugModel itself has proven to be both functional and usable. However, there are cases where the design could be improved with hindsight. The extension constructions, such as folding, and comparation of memory graphs, have added much complexity to the procedure of scrubbing the heap for the memory graph. A further design revision might delegate more work to utility classes - perhaps even providing an extension system of its own to sufficiently separate the creation of the memory graph itself from the extra information required.

The development techniques used for this project also were, on the whole, successful. The use of comprehensive wiki and feature/bug tracking facilities were certainly vital in ensuring that this sort of project, where others depend directly on the code being written, completed successfully. The use of comprehensive testing and use of automated tools throughout also helped provide an accurate and working system.

7.2 Further Work

As with any project of this sort, there is a vast array of further research that might be considered. There were many ideas which we never got a chance to implement during the project, which we would very much like to have tested.

We would have liked to have performed further research on rating the importance of objects. Although the metrics that were developed seemed to provide a useful heuristic for identifying important objects, the work was far from exhaustive. Further empirical evidence would clearly be required in this area - researching production quality code to help identify further metrics would clearly provide a more robust solution. In particular, developing experiments against common code bases utilising human test subjects would provide a useful starting point. We might, for example, give a memory graph to a number of test subjects and ask them to rank the importance of

each object with a score. We can them compare that to the results given by the algorithm, perhaps attempting to develop a measure for representing the quality of a given ranking. This is also an area where user-specified settings might also provide a useful service - since the idea of importance is largely preferential.

Further research on other areas of graph analysis, such as the possibility of compressing graphs with important ownership nodes, or even attempting to classify and cluster objects from different parts of the program, would be a fruitful course of action. In particular, any analysis that could help order or classify areas of the graph would certainly provide a useful area of future extension.

Another area we would have liked to investigate is the implementation of type hierarchy comparison within folding, in addition to standard string based type matching. This would allow a far more complete system for customisation, as well as providing flexibility in the definition of application rules. However, this is more complex than it may at first seem. The Eclipse debugger provides only text based responses to questions of an objects type. Since this may return a type that is not in scope of the debug plugin, we would have to interface with the original process to query the type hierarchy. A simpler, but possibly as useful function would be regular expression matching within the type folding application rules.

On a more technical side, a deeper integration with the Eclipse model itself would certainly provide added value. In particular, source code integration would provide definite benefits for the usability of the program. This might include links back to object declarations or class definitions, or highlighting places where the object is thought to be used. Another interesting technique would be to allow the model to be adaptable. The Eclipse Debug plugin provides support for this kind of modification of reference values in place, so would be a possible extension.

Further support for some of the interesting graph manipulation techniques within the debug model itself would be a useful mechanism. As well as providing more intuitive access to complicated memory slices, for example, we would like to be able to transform and manipulate the graph to provide useful output. We would also like the user, via the API, to be able to thoroughly search the graph space in order to locate a specific object - including search by type, as well as be criteria. Another suggested feature would be to provide "time machine" functionality - backward and forward movements in time. Eclipse debugging does not support this (unlike, for example, the OCaml debugger, which provides tools for rewinding and replaying code paths), however it is a feature that could be reasonably added to the DebugModel plugin. However, this would be limited to tools based upon our code, so the movement through time would not effect in-built debugging tools.

There was little chance to test the program on many different programs in the wild. As such, we have had, as of yet, very little realistic feedback on how well this kind of visualisation helps practical software development, debugging and analysis. There are a wide variety of programs written in Java, and, as such, we would like to identify common patterns in object allocation and ordering amongst them. Such real, empirical data would provide much important information on the scope for further development.

Finally, we would like to provide support for other languages within the Eclipse framework. We would envision this process occurring by splitting the DebugModel into language agnostic and language specific extensions.

Bibliography

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001. ISBN 0-262-03293-7, 0-07-013151-1.

Stephen Diehl. *Software Visualisation*. Springer, Boston, MA, USA, 2007. ISBN 9783540465041.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201633612.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency - Practice and Experience*, 16(7):671–687, 2004.

Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204, 2001.

Further References

As well as those reference mentioned in the bibliography, I also made use of a number of other sources, many of which are listed below. This page is intended to act as a guide for particular tools, including Eclipse references.

As part of our project, we also used a wiki. The wiki is available at http://wlodge.co.uk/projects/visualiser, and contains further technical information, as well as initial design and development considerations.

Eclipse Framework - The homepage for the Eclipse framework, the IDE which we built our plugin for - http://www.eclipse.org/
FindBugs - Tool designed to aid programmers by identifying bugs in Java programs through use of static analysis - http://findbugs.sourceforge.net/
Insight - An Eclipse plugin that performs 2D visualisations of the memory. Essentially DDD for Eclipse. - http://people.cis.ksu.edu/ clt3955/projects.php
JUnit - A Java library for writing unit tests. Integrates into the Eclipse IDE - http://www.junit.org/
EasyMock - A library that allows you to "mock" certain classes to ensure you're fully testing only one unit at a time. - http://www.easymock.org/

Appendix

Code Listing

I have provided a representative sample of some of the code from my project as supporting information for my project. I include the primary three classes, as described in the design section, and also include a sample of unit tests.

Codelisting for DebugModelContainer.java

```
/**
 * A container for a number of DebugModels. Designed to propagate
 * messages from the DebugPlugin into the various models, and
 * notify the listeners.
 * @author luke
 *
 */
public class DebugModelContainer implements IDebugModelContainer,
    IDebugEventSetListener {

  /**
   * Logger
   */
  private Logger logger = Logger.getLogger(DebugModelContainer.class);

  /**
   * Each debug target has its own DebugModel.
   */
  Map<IDebugTarget, IDebugModel> targets =
      new HashMap<IDebugTarget, IDebugModel>();

  /**
   * Only allow our singleton instance.
   */
  public static final DebugModelContainer INSTANCE =
      new DebugModelContainer();

  private DebugModelContainer() {
    // Register our interest in DebugEvents.
    DebugPlugin.getDefault().addDebugEventListener(this);
  }

  public Map<IDebugTarget, IDebugModel> getModels() {
    // We clone the HashMap here so no-one else can make changes.
    return new HashMap<IDebugTarget, IDebugModel>(targets);
  }

  public IDebugModel getModel(IDebugTarget debugTarget) {
    return targets.get(debugTarget);
  }

  public void handleDebugEvents(DebugEvent[] events) {
    logger.debug("We received these debug events:");
```

```
boolean updateModel = false;

for (DebugEvent debugEvent : events) {
  logger.debug("Type:" + debugEvent.getKind() + " Source: "
      + debugEvent.getSource().toString());

  // We only want to update the model on a suspend event or the terminate event
  // We get weird results when we try to update on a return, and it's a little
  // pointless anyway.
  if (debugEvent.getKind() == DebugEvent.TERMINATE) {
    updateModel = true;
  } else if (debugEvent.getKind() == DebugEvent.SUSPEND) {
    updateModel = true;
    break;
  }
}

if (updateModel) {

  /**
   * Check each launch, each target in each launch, each thread in each target.
   * Then pass this information to the appropriate model.
   */
  ILaunch[] launches =
      DebugPlugin.getDefault().getLaunchManager().getLaunches();
  for (ILaunch launch : launches) {
    IDebugTarget[] debugTargets = launch.getDebugTargets();
    for (IDebugTarget debugTarget : debugTargets) {
      try {
        IThread[] threads = debugTarget.getThreads();

        IDebugModel debugModel;
        if (targets.containsKey(debugTarget)) {
          debugModel = targets.get(debugTarget);
        } else {
          debugModel = new DebugModel(debugTarget);
          targets.put(debugTarget, debugModel);
        }

        // We combine all the stack frames for one model, so we can
        // pass them to the model in one go. This helps to know when
        // we can calculate changesets in the model and perform pagerank calculations.
        LinkedList<IStackFrame> stackFrames =
            new LinkedList<IStackFrame>();
        for (IThread thread : threads) {
          stackFrames.addAll(Arrays.asList(thread
              .getStackFrames()));
        }

        logger.debug("Found " + stackFrames.size()
            + " stack frames.");

        debugModel.updateModel(stackFrames);

        // After the model has been completely updated, we perform PageRank caculations
        debugModel.calculatePageRank();

        // The DebugModel calculates the changeset for us
        // (including deleted/updated/new objects)
        Map<DebugChangeType, List<IDebugObject>> changeMap =
            debugModel.getChangeMap();

        logger.info("Update called on DebugTarget:"
            + debugTarget.getName());
```

```
            notifyListeners(debugTarget, changeMap);

          } catch (DebugException e) {
            logger.error("Couldn't rebuild the model.", e);
          }
        }
      }
    }

  }

  /**
   * Listener framework.
   */

  private List<DebugModelContainerListener> listeners =
      new LinkedList<DebugModelContainerListener>();

  public void notifyListeners(IDebugTarget debugTarget,
      Map<DebugChangeType, List<IDebugObject>> objectsChanged) {
    for (DebugModelContainerListener listener : listeners) {
      listener.updateDebugModel(debugTarget, objectsChanged);
    }
  }

  public void addListener(DebugModelContainerListener listener) {
    listeners.add(listener);
  }
}
```

Codelisting for DebugModel.java

```
public class DebugModel implements IDebugModel {

  /**
   * This is the debug target that this model represents
   */
  private final IDebugTarget debugTarget;

  /**
   * Allows us to pass off the dealing with types.
   */
  private static final ITypeHandler typeHandler = new TypeHandler();

  /**
   * This is representing all the objects in this model
   */
  private Map<IJavaValue, IDebugObject> objects =
      new HashMap<IJavaValue, IDebugObject>();

  /**
   * All the values that were new on the last step
   */
  private Map<IJavaValue, IDebugObject> newObjects =
      new HashMap<IJavaValue, IDebugObject>();

  /**
   * All the values changed on the last step
   */
  private Map<IJavaValue, IDebugObject> changedObjects =
      new HashMap<IJavaValue, IDebugObject>();

  /**
```

```java
 * All the values deleted on the last step
 */
private Map<IJavaValue, IDebugObject> deletedObjects =
    new HashMap<IJavaValue, IDebugObject>();

/**
 * Logger
 */
private Logger logger = Logger.getLogger(DebugModel.class);

/**
 * Represents links to objects
 *
 * @param debugTarget
 */
private PageLinks pageLinks = new PageLinks();

/**
 * Represents deleted links to objects
 *
 * @param debugTarget
 */
private PageLinks deletedPageLinks = new PageLinks();

public DebugModel(IDebugTarget debugTarget) {
  this.debugTarget = debugTarget;
}

public Map<IJavaValue, IDebugObject> getValueList() {
  return objects;
}

/**
 * We add a value to our model. If we have previously seen and registered
 * this value, we simply ignore it (returning true to state that we've
 * already registered it).
 *
 * If this value does not belong to the debug target this model represents,
 * we should throw an error
 *
 * @param value
 * @throws WrongDebugTarget
 * @throws DebugException
 * @return boolean true iff the object is in the model (either having been
 *   added, or already existing).
 */
private boolean addObject(IJavaValue value)
    throws WrongDebugTarget, DebugException {
  // Sanity check, we only represent one debug target
  if (value.getDebugTarget().equals(debugTarget)) {

    // We ignore it if we've seen it before - we've already registered
    // it
    if (!objects.containsKey(value)) {

      // We check whether we treat this object specially - see
      // TypeHandling in the wiki.
      ITypeCategory typeCategory =
          typeHandler.getTypeCategory(value);

      if (typeCategory.getTypeCategory() == TypeCategory.IGNORE)
        return false;

      // Check the deleted model for the debug object, then update it
      IDebugObject debugObject;
```

```
      if (deletedObjects.containsKey(value)) {
        debugObject = deletedObjects.remove(value);
        // Unnecessary updating
        debugObject.updateDebugModel(this);
      } else {
        // Didn't previously exist, therefore this is a shiny new
        // object
        debugObject = new DebugObject(value, this);
        newObjects.put(value, debugObject);
      }

      objects.put(value, debugObject);

      if (typeCategory.getTypeCategory() == TypeCategory.RECURSE
          || typeCategory.getTypeCategory() == TypeCategory.FOLD) {
        // We now want to consider objects that this has referenced
        List<IVariable> newVariables = new LinkedList<IVariable>();
        Map<IDebugObject, IVariable> primitives =
            new HashMap<IDebugObject, IVariable>();
        List<IJavaValue> foldingVariables =
            new LinkedList<IJavaValue>();
        foldingVariables.add(value);
        calculateLinks(value, typeCategory, debugObject,
            newVariables, primitives, foldingVariables);

        if (!newObjects.containsKey(value)) {
          Map<IDebugObject, IVariable> deletedFromLinks =
              deletedPageLinks.getFromLinks(debugObject);
          Map<IDebugObject, IVariable> currentFromLinks =
              pageLinks.getFromLinks(debugObject);
          if (!match(currentFromLinks, deletedFromLinks)) {
            changedObjects.put(value, debugObject);
          }
        }

        debugObject.addPrimitives(primitives);
      }
    }
    return true;
  } else {
    // Debug targets don't match, throw error.
    throw new WrongDebugTarget(debugTarget, value.getDebugTarget());
  }
}

private boolean match(Map<IDebugObject, IVariable> deletedLinks,
    Map<IDebugObject, IVariable> links) {
  if (deletedLinks == links) {
    return true;
  } else if (deletedLinks == null || links == null) {
    return false;
  } else {
    return (deletedLinks.keySet().equals(links.keySet()));
  }
}

/**
 *
 * @param value
 * @param typeCategory
 * @param debugObject
 * @param newVariables
 * @param primitives
 * @param foldingVariables
```

```
 * @throws DebugException
 * @throws WrongDebugTarget
 */
private void calculateLinks(IJavaValue value,
    ITypeCategory typeCategory, IDebugObject debugObject,
    List<IVariable> newVariables,
    Map<IDebugObject, IVariable> primitives,
    List<IJavaValue> foldingVariables) throws DebugException,
    WrongDebugTarget {
  // We want to ensure we have complete coverage of all objects
  // So we need to consider all the objects that may only be
  // referenced in this object.
  for (IVariable variable : value.getVariables()) {
    IValue childValue = variable.getValue();
    if (childValue instanceof IJavaValue) {
      IJavaValue childJavaValue = (IJavaValue) childValue;
      // We consider the type of each object before adding it to the
      // graph
      ITypeCategory childTypeCategory =
          typeHandler.getTypeCategory(childJavaValue);
      if (childTypeCategory.getTypeCategory() == TypeCategory.PRIMITIVE) {
        primitives.put(new DebugObject(childJavaValue, this),
            variable);
        // Should primitives count towards pagerank?
      } else if (typeCategory.getTypeCategory() == TypeCategory.FOLD
          && typeCategory.contains(childValue
              .getReferenceTypeName())) {
        if (!foldingVariables.contains(childJavaValue)) {
          foldingVariables.add(childJavaValue);
          calculateLinks(childJavaValue, childTypeCategory,
              debugObject, newVariables, primitives,
              foldingVariables);
        }
      } else if (addObject(childJavaValue)) {
        newVariables.add(variable);
        pageLinks.addLink(debugObject, objects.get(childJavaValue),
            variable);
      }
    }
  }
}

@Override
public void updateModel(List<IStackFrame> stackFrames) {
  deletedObjects = objects;
  objects = new HashMap<IJavaValue, IDebugObject>();
  deletedPageLinks = pageLinks;
  pageLinks = new PageLinks();
  changedObjects = new HashMap<IJavaValue, IDebugObject>();
  newObjects = new HashMap<IJavaValue, IDebugObject>();

  try {
    for (IStackFrame stackFrame : stackFrames) {
      IVariable[] variables = stackFrame.getVariables();
      for (IVariable variable : variables) {
        IValue value = variable.getValue();
        logger.debug("Found a variable, " + variable.getName());
        if (value instanceof IJavaValue) {
          if (addObject((IJavaValue) value)) {
            logger.debug("Added " + value.getValueString()
                + " as a value (primitive or otherwise).");
          }
        }
      }
    }
```

```
    } catch (DebugException e) {
      logger.error("Couldn't rebuild the model.", e);
    } catch (WrongDebugTarget e) {
      logger
          .error(
              "We've attempted to attach an object to the wrong DebugTarget.",
              e);
    }

  }

  public void calculatePageRank() {
    double d = 0.85;

    double N = objects.values().size();

    // Initialise values
    for (IDebugObject object : objects.values()) {
      object.setPageRank(1 / N);
    }

    // We iterate a number of times until we assume we reach some sort of
    // convergence.
    for (int i = 0; i < 50; i++) {
      for (IDebugObject object : objects.values()) {
        double newPageRank = 0;
        if (pageLinks.getFromLinks(object) != null) {
          for (IDebugObject link : pageLinks.getFromLinks(object)
              .keySet()) {
            newPageRank +=
                (link.getPageRank() / pageLinks.getToLinks(link)
                    .size());
          }
        }
        object.setPageRank(((1 - d) / N) + (d * newPageRank));
      }
    }
  }

  public IDebugObject getObject(IJavaValue object) {
    return objects.get(object);
  }

  /**
   * Number of links from other objects to this object (not including
   * primitives)
   */
  public Map<IDebugObject, IVariable> linksToObject(
      IDebugObject object) {
    return pageLinks.getToLinks(object);
  }

  /**
   * Number of links from this object to other objects (not including
   * primitives)
   */
  public Map<IDebugObject, IVariable> linksFromObject(
      IDebugObject object) {
    return pageLinks.getFromLinks(object);
  }

  public List<IDebugObject> getObjects() {
    return new LinkedList<IDebugObject>(objects.values());
  }
```

```
  @Override
  public Map<DebugChangeType, List<IDebugObject>> getChangeMap() {
    HashMap<DebugChangeType, List<IDebugObject>> changeMap =
        new HashMap<DebugChangeType, List<IDebugObject>>();

    // Simply return the current changeset
    changeMap.put(DebugChangeType.DELETED,
        new LinkedList<IDebugObject>(deletedObjects.values()));
    changeMap.put(DebugChangeType.CHANGED,
        new LinkedList<IDebugObject>(changedObjects.values()));
    changeMap.put(DebugChangeType.CREATED,
        new LinkedList<IDebugObject>(newObjects.values()));

    return changeMap;
  }
}
```

Codelisting for DebugObject.java

```
package debugmodel.model;

import java.util.HashMap;
import java.util.Map;

import org.eclipse.debug.core.model.IVariable;
import org.eclipse.jdt.debug.core.IJavaValue;

import debugmodel.exceptions.NullLinkException;
import debugmodel.interfaces.IDebugModel;
import debugmodel.interfaces.IDebugObject;

/**
 * Standard implementation of IDebugObject.
 * Simply a wrapper around the underlying java object.
 * @author luke
 *
 */
public class DebugObject implements IDebugObject {

  /**
   * The object supporting this class.
   */
  private final IJavaValue underlyingValue;

  /**
   * Primitive objects stored under this objects
   */
  private Map<IDebugObject, IVariable> primitives =
      new HashMap<IDebugObject, IVariable>();

  /**
   * Initial pagerank
   */
  private double pageRank = 0.0;

  /**
   * The debug model that created us.
   */
  private IDebugModel debugModel;

  public DebugObject(IJavaValue underlyingObject,
      IDebugModel debugModel) {
    this.underlyingValue = underlyingObject;
    this.debugModel = debugModel;
```

```java
}

public IJavaValue getValue() {
  return underlyingValue;
}

public Map<IDebugObject, IVariable> objectLinks()
    throws NullLinkException {
  Map<IDebugObject, IVariable> variableObjects =
      debugModel.linksFromObject(this);

  if (variableObjects == null) {
    variableObjects = new HashMap<IDebugObject, IVariable>();
  }

  return variableObjects;
}

public Map<IDebugObject, IVariable> backLinks()
    throws NullLinkException {
  Map<IDebugObject, IVariable> variableObjects =
      debugModel.linksToObject(this);

  if (variableObjects == null) {
    variableObjects = new HashMap<IDebugObject, IVariable>();
  }

  return variableObjects;
}

public Map<IDebugObject, IVariable> primitiveLinks()
    throws NullLinkException {
  return primitives;
}

public void addPrimitives(Map<IDebugObject, IVariable> primitives) {
  this.primitives = primitives;
}

public void updateDebugModel(IDebugModel debugModel) {
  this.debugModel = debugModel;
}

/**
 * @returns true iff the underlying objects are the same.
 */
@Override
public boolean equals(Object obj) {
  if (obj instanceof DebugObject) {
    DebugObject debugObject = (DebugObject) obj;
    return (debugObject.getValue().equals(this.getValue()));
  } else {
    return false;
  }
}

@Override
public int hashCode() {
  return underlyingValue.hashCode();
}

public void setPageRank(double pageRank) {
  this.pageRank = pageRank;
}
```

```
  public int linksFromObject() {
    return debugModel.linksFromObject(this).size();
  }

  public int linksToObject() {
    return debugModel.linksToObject(this).size();
  }

  public double getPageRank() {
    return pageRank;
  }
}
```

Codelisting for one test from TestDebugObject.java

```
public class TestDebugObject extends TestCase {

  ...

  /**
   * A simple test of checking object links.
   *
   * @throws NullLinkException
   */
  public void testGetLinks() throws DebugException, NullLinkException {
    final IJavaObject underlyingObject =
        createMock(IJavaObject.class);

    final IVariable linkOne = createMock(IVariable.class);
    final IJavaObject underlyinglinkOne =
        createMock(IJavaObject.class);
    final IVariable linkTwo = createMock(IVariable.class);
    final IJavaObject underlyinglinkTwo =
        createMock(IJavaObject.class);

    final IDebugModel debugModel = createMock(IDebugModel.class);

    DebugObject debugObject =
        new DebugObject(underlyingObject, debugModel);

    DebugObject debugObjectOne =
        new DebugObject(underlyinglinkOne, debugModel);
    DebugObject debugObjectTwo =
        new DebugObject(underlyinglinkTwo, debugModel);

    Map<IDebugObject, IVariable> objectList =
        new HashMap<IDebugObject, IVariable>();
    objectList.put(debugObjectOne, linkOne);
    objectList.put(debugObjectTwo, linkTwo);
    expect(debugModel.linksFromObject(debugObject)).andReturn(
        objectList);

    expect(linkOne.getValue()).andReturn(underlyinglinkOne);
    expect(linkTwo.getValue()).andReturn(underlyinglinkTwo);

    expect(debugModel.getObject(underlyinglinkOne)).andReturn(
        debugObjectOne);
    expect(debugModel.getObject(underlyinglinkTwo)).andReturn(
        debugObjectTwo);

    replay(underlyingObject);
    replay(linkOne);
    replay(linkTwo);
    replay(debugModel);
```

```
    Set<IDebugObject> objectLinks =
        debugObject.objectLinks().keySet();
    assertEquals(2, objectLinks.size());
    assertTrue(objectLinks.contains(debugObjectOne));
    assertTrue(objectLinks.contains(debugObjectTwo));

    boolean one = false, two = false;
    for (IDebugObject object : objectLinks) {
      if (object.equals(debugObjectOne)) {
        assertEquals(underlyinglinkOne, object.getValue());
        one = true;
      } else if (object.equals(debugObjectTwo)) {
        assertEquals(underlyinglinkTwo, object.getValue());
        two = true;
      } else {
        fail("We've found an item in the object list that we didn't put in.");
      }
    }
    assertEquals(2, objectLinks.size());
    assertTrue("Object one was not found", one);
    assertTrue("Object two was not found", two);
  }

  ...
}
```