"The candidate confirms that the work submitted is his/her own and the appropriate credit has been given where reference has been made to the work of others".
1. SUMMARY

The main objective of the project is to create a multi-platform application whose primary function is to generate UML class diagrams from Java classes. Other more general objectives included exploring the features provided by UML and CASE tools and the improvement of my Java programming abilities especially in the areas covered by the Java Reflection and Graphics APIs, with the aim of increasing my understanding in these areas.

To achieve these objectives I built an application, the UML Reverse Engineer (URE) using Java that was able to analyze Java classes in the form of Java byte code and optionally Java source code (text). The URE is able to reverse engineer a UML class diagram from the Java classes by utilizing tools available from the Reflection package to analyze the Java byte code and a rudimentary text-searching algorithm to extract pertinent information from the Java source code.

The URE also has a Graphical User Interface (GUI), which displays the reconstructed UML class diagram. Features of the GUI included a mouse function that allows users to reposition individual classes around the screen to aid clarity and a comprehensive menu that avails users with the ability to load and save class diagrams from and to files on disk just to mention a few. The GUI and the ability to analyze Java classes together form the core of the URE. An extra feature not in the minimum requirements of the project is a layout manager that is able to automatically reposition the class diagram elements so that there are the fewest number of line crossings and overlapping elements i.e. finds the optimum layout for the class diagram displayed in the GUI.

The core part of the URE can run on any platform that supports the Java 2 platform. (JDK 1.3 and above recommended.) It has been tested on both windowsNT/2000 and Linux Red Hat 7.0.

The layout manager currently runs only in the windows environment, but could in the future be extended to include the Linux. This said, the layout manager can still work on any system has an Internet connection as the platform dependent part of the layout manager can be run remotely from a website.
2. ACKNOWLEDGMENT

I would like to thank supervisor, Dr Nick Efford for his help and advice throughout the Project.
3. CONTENTS

1. SUMMARY .................................................................................................................. 2

2. ACKNOWLEDGMENT ................................................................................................. 3

3. CONTENTS .................................................................................................................. 4

4. INTRODUCTION ......................................................................................................... 6

5. BACKGROUND .......................................................................................................... 7
  5.1 The Unified Modeling Language (UML) ................................................................. 7
    5.1.1 Definition: .......................................................................................................... 7
    5.1.2 Goals of UML ................................................................................................... 8
    5.1.3 Class Diagrams ................................................................................................. 9
    Notation used in class representations ..................................................................... 10
    Class diagram Relationships .................................................................................... 11
  5.2 Java .......................................................................................................................... 14
    5.2.1 Why Java was chosen ...................................................................................... 14
    5.2.2 The Java Graphics API(s) ............................................................................... 14
    5.2.3 The Java Reflection API - Overview ............................................................... 15
    5.3.4 Reflection Model .............................................................................................. 15

6. IMPLEMENTATION .................................................................................................... 17
  6.1. Reflection – gathering information. ................................................................. 18
  6.2. Relationships – establishing links ................................................................... 19
    6.3.a. Generalizations and realizations ................................................................. 19
    6.3.b. Associations ................................................................................................ 19
  6.3. The Graphical User Interface (GUI) ................................................................. 19
    6.3 a. Drawing Area ............................................................................................... 20
      Drawing the classes: ........................................................................................... 20
      Drawing relationships: ....................................................................................... 20
    6.3 b. Dialog boxes ................................................................................................. 21
    6.3 c. Main Menu .................................................................................................. 21
    6.3 d. Layout Manager .......................................................................................... 22

7. EVALUATION AND TESTING ................................................................................. 25
  7.1 Evaluation criteria: ............................................................................................ 25
  7.2 Testing .................................................................................................................... 25
  7.3 Results ..................................................................................................................... 27

8. LIMITATIONS .......................................................................................................... 27

9. FURTHER WORK ....................................................................................................... 28
10. CONCLUSION ..............................................................28

11. REFERENCES ............................................................30

12. APPENDICES ............................................................31

Appendix A – Reflection on the project ..............................................31
Appendix B – copy of Objectives and Deliverable form .........................32
Appendix C - marking scheme, header sheet and assessors comments for the marked Interim Report .................................................................34
Appendix D – Screen shots of the URE’s menu ........................................36
Appendix E – screen shots of the URE’s dialog boxes in action .................39
Appendix F – screenshots of the layout manager in action .......................40
Appendix G - Miscellaneous screenshots ...............................................41
Appendix H - Dummy code used to test the URE .................................43
  Dummy code #1 ........................................................................43
  Dummy code #2 ........................................................................43
Appendix I – testing ........................................................................45
Appendix J - Reflection and Relationships ..........................................48
Appendix K – Evolution of UML ........................................................49
4. INTRODUCTION

UML, which stands for Unified Modelling Language is a language for designing an object orientated model to describe the system. It was the brainchild of Grady Booch, James Rumbaugh and Ivor Jacobson. Its inception was in late 1995 and was essentially an amalgamation of object orientated modeling techniques developed by its original founders. In normal use the UML is used to aid developers in the planning stage through to the actual implementation and maintenance stages. This progression of software engineering is known as forward engineering. Conversely the project aims to reverse engineer Java classes.

4.1 Objectives

- Learn about UML and learn how it is used for visual modelling and code generation.
- Improve my Java programming skills especially in using the Java Graphics and Reflection APIs.
- Develop a Java software prototype that analyzes Java byte code and generates UML class diagrams from it.

4.2 Deliverables

a) A project report
b) A software prototype

4.3 Minimum requirements

- An application that reads Java classes provided to it and extracts information such attributes and local methods.
- A Graphical User Interface (GUI) that displays the extracted information as a Class diagram.

4.4 Current State of the art

There are many commercial visual modelling tools that have reverse engineering tools built in. Examples of these include Rational Rose and MagicDraw. Rational Rose is a product from Rational, a company founded by the Booch et al. It provides a tool to reverse engineer Java code in one of its menus. MagicDraw is a visual UML modeling and CASE tool with teamwork support. MagicDraw supports the latest standards of Java, UML and XMI and provides full round-trip support for Java programming languages. MagicDraw also provides reverse engineering facilities.
5. BACKGROUND

5.1 The Unified Modeling Language (UML).

5.1.1 Definition:
The Object Management Group (OMG) ’s specification defines UML as:

"A graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems.”

UML is an open standard and as such anyone can download the complete UML specification from OMG ’s Web site free of charge (http://www.omg.org). OMG controls the process of maintaining and extending UML. Other companies can and have joined the standards organization and have contributed to the development process of UML now in UML 1.4 (see Appendix K for evolution of UML).

UML represents a collection of proven successful best engineering practices that have in the modeling of large and complex systems.
The Unified Modelling language (UML) is, as the name suggests, a unification of a number of earlier object-oriented modelling languages. The three principal designers of UML had each previously published their own methods, and the explicit purpose of UML was to integrate the insights of these three methods. This integration was aided by the common framework, in the form of the object model, shared by the original methods. (Mark Priestly, Practical Object-Oriented Design with UML.)

UML It is the product of the amalgamation of the modeling languages found in the Booch, OOSE/Jacobson, OMT and other methods. The UML standard covers disciplines such as business modeling, requirements management, analysis & design, programming and testing.
Prior to the inception of the UML, there was no universally accepted modeling language. Users had to choose from a plethora of similar modeling languages with slightly different approaches to modeling. It turns out that most of these modeling languages shared a set of commonly accepted concepts the differences were due to the slightly different notations they used. The many approaches used had the effect of fragmenting the object oriented industry, as a result many new users were put off from learning and using visual modelling techniques. Developers wanted the software industry to agree on one broadly supported modeling language that was suited to general-purpose use i.e. they wanted a lingua franca for modelling software.

UML defines a language only, and as such provides no description or recommendations relating to the process of development (Mark Priestly, Practical Object-Oriented Design with UML.)

Software professionals to create virtual models of software systems they plan to build use the Unified Modeling Language (UML) especially when they want to model large complex software applications. They do this for the same reason people hire architects to design buildings before actually building them e.g. it is easier and less expensive to make corrections and alterations on the models than it is to make changes after implementing the systems thus in the long run, a well designed and documented
system saves both time and money. UML enables developers to plan their software better by making them consider their alternatives, select the best option, work out details, and achieve agreement before they start the building the application. As mentioned earlier it is much less costly to use a model than it is to modify a building or a software application after it has been assembled. A secondary but no less important advantage of using UML in the development process is the documentation generated. This is important as typically 90% of the costs involved in large applications are as a result of changes, extensions, and other maintenance to the software systems after they have been built. The UML documentation with its innate clarity is able to not only direct these changes but also document them in case they have to be revised at a future date.

5.1.2 Goals of UML

The primary goals in the design of the UML were as follows:
1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.

Chris Kobryn, document omg/01-03-02 updated UML lecture 1, March 2001 version.

![Diagram of UML views]

Figure 1: The structure of UML software systems known as the 4+1 view model.

By amalgamating all five views in figure 1, it is possible to model the complete system, but for particular purposes it may be sufficient to consider only the information contained in just a few of the views leaving out some of the views. The use case view defines how the system interacts with the external world. It is particular interest to end users, analysts and testers of the system. It defines the requirements of the system and plays a central role as it constrains the other views and is often said to drive the development process.
The *design view* describes the logical structures that support the functional requirements of the system as expressed in the use case view. It consists of definitions of program components, mainly classes together with any specifications of the data they hold, and their behaviours and interactions. A class diagrams is an example of a design view.

The *implementation view* describes the physical components from which the system is to be constructed. They are different from logical components and include things like executables files, code libraries and databases. This information is particularly useful in configuration management and system integration.

The *process view* deals with issues of concurrency within the system i.e. how processes and events in the system are ordered with respect to each other. The *deployment view* deals with how the physical components of the system are distributed in the physical environment e.g. distribution over a computer network.

<table>
<thead>
<tr>
<th>Diagram</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Use case diagram</td>
<td>Use case view</td>
</tr>
<tr>
<td>2. Class diagram</td>
<td>Design view</td>
</tr>
<tr>
<td>3. Object diagram</td>
<td>Use case and design views</td>
</tr>
<tr>
<td>4. Sequence diagram</td>
<td>Use case and design views</td>
</tr>
<tr>
<td>5. Collaboration diagram</td>
<td>Use case and design views</td>
</tr>
<tr>
<td>6. State-chart diagram</td>
<td>Design view</td>
</tr>
<tr>
<td>7. Activity diagram</td>
<td>Design view</td>
</tr>
<tr>
<td>8. Component diagram</td>
<td>Implementation view</td>
</tr>
<tr>
<td>9. Deployment diagram</td>
<td>Deployment view</td>
</tr>
</tbody>
</table>

Table 1: UML diagram types.

Information contained in a UML model is usually represented in graphical form using the diagram types in Table 1.

5.1.3 **Class Diagrams**

Classes categorise the objects that exist in the system and define their shared properties. Class diagrams show the classes and variety of relationships between the classes.

```plaintext
classA

- attribute01
+ attribute02 : Boolean {final}
# attribute03 : Integer

- operation01() { throws exception }
# operation02(String, int)
+ operation03(boolean) : String
```

Figure 2: Basic notation for classes.
Construct | Description | Syntax
--- | --- | ---
class | A class describes of a set of objects that share the same attributes, operations, methods, relationships and semantics. In class-based object oriented languages like Java, every object belongs to a class. |
interface | An interface defines features in particular operations that other modules depend on i.e. a named set of operations. They are useful as they mean a module implementing a well-defined interface can change internally without necessitating extensive changes elsewhere in the system. | `{interface}`
constraint | This is a condition that has to be satisfied for correct implementation of the software e.g. student studies alongside an association between a student object and a course object. | `{constraint}`

Table 2: Some of the elements in a class diagram.

**Notation used in class representations**

In class-based object oriented languages like Java, every object belongs to at least one class. UML class objects contain the name, attributes and operations of the class (see figure 2 above). Operations can be defined as the ways in which objects may interact i.e. behaviours of objects. Attributes on the other hand describe the data contained in an object of the class.

a. Attributes

**syntax:**

```
[visibility]  [type]  name  [{properties}]
```

- **[visibility]** Symbol ‘+’, ‘-‘ and ‘#’ are used to represent the access levels of the attributes. The symbols represent public, private and protected respectively. However during design, access levels are hardly used due to the fact that access levels are language dependent and are not supported by all programming languages e.g. attribute01 in figure 2 has prefix ‘-‘, signifying it has a private visibility.

- **[type]** This is the type of the attribute e.g. attribute03 in figure 2 is of type Integer.

- **[{properties}]** Consists of a list of the constraints on the attribute e.g. attribute02 has a property of final which in Java means that the attribute has the constraint of not being able to assigned new values after the initial declaration.

Table 3: Brief description of attribute syntax.
b. Operations

**syntax:**

\[
[\text{visibility}] \ [\text{return-type}] \ \text{name} \ ((\text{parameters})) \ [(\text{properties})]
\]

<table>
<thead>
<tr>
<th>[visibility]</th>
<th>Same as in visibility of attributes above i.e. represent the access levels of the attributes e.g. operation02 in figure 2 has prefix ‘#’, signifying it has a protected visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(parameters)]</td>
<td>This is made up of all the types of the arguments/messages passed to this operation separated by a comma e.g. operation02 in figure 2 has two arguments passed to it, the first is of type String while the second is of type int.</td>
</tr>
<tr>
<td>[: return-type]</td>
<td>This is the type of message that is returned by this operation. e.g. operation03 in figure 2 has a String return-type.</td>
</tr>
<tr>
<td>[{properties}]</td>
<td>Consists of a list of the constraints on the attribute e.g. operation01 throws an exception. This is important, as an object utilizing this operation needs to know that it has to handle this exception.</td>
</tr>
</tbody>
</table>

Table 4: brief description of operations syntax.

**Class diagram Relationships**

In UML, classes can be connected/linked by means of associations, generalization or aggregation among others.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>association</td>
<td>Shows that there is a relationship between two or more instances of classes. The links between the classes can be considered as instances of association.</td>
<td></td>
</tr>
<tr>
<td>aggregation and composition</td>
<td>A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.</td>
<td></td>
</tr>
<tr>
<td>generalization</td>
<td>a taxonomic relationship between a more general and a more specific element.</td>
<td></td>
</tr>
<tr>
<td>dependency</td>
<td>a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).</td>
<td></td>
</tr>
<tr>
<td>realization</td>
<td>a relationship between a specification and its implementation.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Some of the Relationships in a class diagram.
Associations
Associations express the relationship between classes i.e. an association between two classes implies links between instances of the classes. e.g. class A and class B are associated if an instance of class A knows about an instance of class B and or vice versa i.e. if
- A class A object contains an instance of class B
- A class A object has an attribute whose values include one or more instances of class B objects.
- A class A object receives a message with a class B object as one of its values.
- A class A object sends messages to a class B object.

![Diagram of Employee and Company association](image)

Figure 3: A simple example of an association between classes.

An open arrowhead at the tip of the association line is sometimes used to show the direction (navigability) of the association. An association line without any indication of direction can generally be assumed to be same as a double-headed association line i.e. it implies a two-way association where both classes know about each other.

Aggregation and composition
As mentioned in table 5, aggregation and composition are kinds of associations. In addition to showing that two classes are associated, they show that an object of one class is part of another class object. The difference between aggregation and composition is composition is a special kind of aggregation with extra restrictions. In composition, the whole has very strong links with its parts e.g. if the whole part is deleted the all its parts are also deleted. This means that a part cannot be have more than one whole. (Take part to be the object that makes up part of another object and that the whole is the object made up one or more parts.)

![Diagram of Course and Module aggregation](image)

figure 4: an aggregation, ten or more Modules make up one or more Courses.

![Diagram of Chessboard and Square composition](image)

figure 5: a composition, sixty four Squares(parts) make one Chessboard(whole).

Generalisation

![Diagram of Animal and Dog generalisation](image)

Figure 6: an example of a generalisation.
This is because conceptually every dog is an animal. Generalization can be loosely seen as inheritance, where one class inherits (in the case of Java extends) attributes and behaviors of another class.

**Realization**

```java
<<interface Marks>>
+ getMarks(): int

Module
- marks: int
+ getMarks(): int
```

Figure 7: An example of a realization. Module implements interface Marks.
5.2 Java

The Java programming language is now in the second version of its specification (Java 2) and the forth incarnation of its Java development kit (JDK 1.3). It was created in 1991 by Sun Microsystems, and was initially designed as a platform independent language that could be used on many different systems from personal computers to consumer electronic devices like videos. They also wanted to remove the need for full CPU-specific compilers for each OS platform. With the increased popularity of the World Wide Web, the initially concept ideals were altered in 1995. Sun still kept the platform independent concept of the language, as they saw that improving interactivity of client side programs would still require programs to have cross platform portability. Currently Java is aimed more at personal computer architectures; the consumer electronic devices part of the initial concept has been scaled down.

The Java compiler produces byte code files, which have a .class extension instead of the more conventional binary executable files that might be produced by a C++ compiler for example. These .class files can be viewed as versions of the source code optimized for execution by the Java Virtual Machine (JVM). At execution the Java byte code is interpreted by a JVM that has been designed to run on a specific platform, so the same byte code can run on any operating system with a JVM. This means that whilst the compiled Java programs are platform independent, the JVM is platform specific. But taking all this into account, the overall cost involved in creating the JVM for each type of system substantially less than the cost of implementing different programs for each platform for all the software produced. The main limiting factor is the fact that Java interprets its code instead of producing binary executables that directly interface with the operating system. This is because there are significant performance issues associated with this type of interpretation especially in computationally expensive applications and in graphically environments. The new release of Java 2 and the JDK 1.3, has improved performance especially the graphics part but the issue of runtime performance still remains.

5.2.1 Why Java was chosen.
As Java is an Object Oriented language i.e. it like UML follows the object oriented paradigm, this means that it has many useful features that aid developers to produce better software; features include its inherent security and robustness, better scalability, readability and maintainability. The take up of Java was smoothed by its close syntax similarity to C++, which was already at the time of developing Java established as an industry standard especially in the object oriented arena. However there are some major differences between the languages such as automatic garbage collection and inbuilt multithreading provided by Java but absent from C++. The most important difference especially pertaining to this project is the fact that Java provides reflection and C++ does not.

5.2.2 The Java Graphics API(s).
Java2 introduced the java.swing API, which is made up of graphical interfaces that are mainly improvements on previous Java 1 interfaces. Many of the classes are extensions of the Abstract Windows Toolkit API (java.awt), while some are
completely new as the foundations of \texttt{java.swing} are \texttt{java.awt} and the new Java2D toolkit. With the advent of Java 2 there has been the incorporation of the Swing components into the main Java API. The Java 2 API still includes the AWT classes because many of the Swing classes depend on them e.g. \texttt{java.event.awt.MouseListener}.

### 5.2.3 The Java Reflection API - Overview

“Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions. The API accommodates applications that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.”

Java\textsuperscript{TM} 2 SDK, Standard Edition Documentation.

The Java(TM) Core Reflection API provides a small, type-safe, and secure API that supports introspection about the classes and objects in the current JVM. If permitted by security policy, the API can be used to:

- a) Construct new class instances and new arrays
- b) Access and modify fields of objects and classes
- c) Invoke methods on objects and classes
- d) Access and modify elements of arrays

Java\textsuperscript{TM} 2 SDK, Standard Edition Documentation describes three objects of type class \texttt{Field}, \texttt{class Method}, and \texttt{class Constructor} at the core of the Reflection API. These objects can be utilized to provide the reflective information about the underlying attributes, members and constructors respectively. Reflection makes use of methods in class \texttt{Class} to construct new instances of the \texttt{Field}, \texttt{Method}, and \texttt{Constructor} classes. The utility class \texttt{Modifier} can then be used in the decoding of the Java language modifier information in the reflective information e.g. the return type of a method.

### 5.3.4 Reflection Model

The three classes \texttt{Field}, \texttt{Method}, and \texttt{Constructor} are \texttt{final}. Only the Java Virtual Machine may create instances of these classes; these objects are used to manipulate the underlying objects; that is, to:

- get reflective information about the underlying member or constructor
- get and set field values
- invoke methods on objects or classes
- create new instances of classes

Only the first two of these features is applicable to this project. The other features are also useful especially when using Java Beans where situations might arise that require a class to be created during runtime.

**Field Objects**

The \texttt{Field} objects represent reflected fields. The fields may be static class variables or none static instance variables. Methods of class \texttt{Field} can be used to obtain the name and type of the underlying field.
Method Objects
Methods in a class may be abstract, instances, or static classes. Class Method provides methods that can be used to obtain the parameter types, the return type as well as checking if the methods throw any exceptions.

Constructor Objects
Methods of class Constructor are used to obtain the formal parameter types and the checked exception types of the constructors of the class object being inspected.

Modifier Class.
The Modifier class is a static class. It provides methods to decode Java language modifiers for classes and members. The language modifiers are encoded as integers as defined by the JVM Specification encoding constants

Member Interface
Field, Method and Constructor classes implement the Member interface. Methods provided by class Member are utilized to extract identifying information. The identifying information consists of the name of the class or interface that declared the member, the name of the member itself and the check whether the objects are public, protected, abstract and synchronized.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Constructor provides information about, and access to, a single constructor for a class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>A Field provides information about, and dynamic access to, a single field of a class or an interface.</td>
</tr>
<tr>
<td>Method</td>
<td>A Method provides information about, and access to, a single method on a class or interface.</td>
</tr>
<tr>
<td>Modifier</td>
<td>The Modifier class provides static methods and constants to decode class and member access modifiers.</td>
</tr>
</tbody>
</table>

Table 6: Summary of Reflection API’s primary methods as specified in Sun’s Java documentation.

Java documents states that Reflection allows runtime access to information about the fields, methods and constructors of loaded classes. Reflection can also be used to operate on the underlying counterpart objects, within security restrictions. Classes in the reflection package along with java.lang.Class are sometimes used by services such as Object Serialization and Java Beans that sometimes need access to information about objects during execution. Reflection is also used by applications such as debuggers, interpreters, and class browsers.
6. IMPLEMENTATION

Gather information to be used to construct Class Diagram

1. Reflection

Extract information about the classes provided using the tools provided by the Java Reflection API.

2. Relationships

Use the information gathered in part 1 to establish the relationships between the classes.

3. The Graphical User Interface (GUI).

Used to display the class diagram. Also provides functionality to control how it is displayed.

a. Drawing Area

Provides the main interface where the user can view the class diagram.

b. Dialog Boxes

Used to display the details of individual class objects.

c. Menu

Provides the user with standard menu functions like Open, save and Exit as well none standard functions to enable the user to control what appears in the drawing area. Also uses the layout manager.

d. Layout manager

Used to display the class diagram that has been auto-arranged using the dot program.

Figure 8: Design of the final URE.
The implementation of the UML Reverse Engineer (URE) can be broken down into three parts.

1. Reflection – This involves the gathering of information from the java .class files.
2. Relationships – This looks at the information gathered and tries to find any inter-relationships.
3. The Graphical User Interface (GUI) – The visual components of the URE.

(See figure 8 on the previous page for the design of the overall system. Figure 11, page 25 shows the class diagram. Appendix J has a figure showing how reflection and especially relationships are actually implemented)


This takes in the name of the class being inspected (as given in the command line arguments) and creates and instance of the class, provided the class exists in the java class path. The URE then calls the inspect method in class ClassInspector. Inspect() is able to examine the class, extract information from it using reflection. The information is then stored in the Vector v. The process is repeated for all the classes in the command line arguments.

```java
ClassInspector obj_cls = new ClassInspector();
Vector data = new Vector();
Class cls = Class.forName(name_cls);
data = obj_cls.inspect(cls);
...
// accumulate all the data for all classes into one Vector v.
for (int i = 0; i < data.size(); i++) {
    ClassInfo info = (ClassInfo) data.elementAt(i);
    v.addElement(info);
}
```

ClassInspector.inspect() with the help of other methods in ClassInspector gets the class’s name, fields (attributes), constructors (operations part 1), methods (operations part 2), interfaces (realization), the package the class belongs to and finally its superclass.

```java
// method: ClassInspector.inspect()
public Vector Inspect(Class c) {
    ClassInfo = new ClassInfo();
    classInfo.cls_name = c.getName(); // get name
    findFields(c); // get attributes
    findConstructors(c); // get operations part 1
    findMethods(c); // get operations part 2
    findInterfaceNames(c); // get realization
    findPackage(c); // get the package this class
    // belongs to.
    Class super_cls = c.getSuperclass();
    if (super_cls != null) // Call the method
        return Inspect(super_cls); // recursively until
        // there are no more
        // superclasses.
```
6.2. Relationships – establishing links

6.3.a. Generalizations and realizations.

The second stage is to use the information gathered in reflection above and uses it to establish the relationships between the classes found. Each ClassInfo object stores the name of its super class (generalization) and the names of the interfaces it implements (realizations). It is possible by searching the main Vector v to establish how the classes are interrelated. This proved more difficult to implement than it first appeared, the solution I finally used was to search the Vector v for classes whose names match those in the set of generalisations and set of realisations in stored in the particular ClassInfo object whose relationships are being ascertained and store its position in Vector v in the Class object e.g.

If v.elementAt(2) is the super class of v.elementAt(5), then v.elementAt(5) adds the number 2 to is set of generalizations.

This enables the URE to know that the object at position 5 in Vector v is related to the object at position 2 in Vector v and that their relationship is a generalization (realization relationships are stored in a set of interfaces as opposed to a set of generalizations). This becomes particularly handy when it comes to determining links and the location of the objects they are linking are.

6.3.b. Associations.

The case of associations is slightly different. It is possible to determine if class A is associated with class B by searching through the attributes (fields) and operations (methods) of class B for occurrences of objects or messages that are of type class A. This works in cases where the programmer has decided to make the associated object an attribute or message (return type or pass-by value of a method) of the class. However UML does not restrict associations to instances of one class that have been declared as attributes or form part of the messaging of another object class, it only requires that one or both of the objects know about each other. This means that the object instance could be declared anywhere in the code. To overcome this problem the URE has the ability to read in the actual source code and perform a simple text search for looking for occurrences of the classes involved in the source code. In both cases if it finds an instance of a class e.g. there is an instance of class A in class B, the URE adds the name of class A to the set of associations in the ClassInfo object pertaining to class B. The association relationships are then enumerated exploiting the same method as described above for generalizations and realizations relationships, for the same reasons.

6.3. The Graphical User Interface (GUI)

The GUI consists of:

a. A drawing area
b. Dialog boxes
c. A menu, and
d. The Layout Manager.
6.3 a. Drawing Area

Drawing the classes:

Design possibilities:
   a. Use internal frames
   b. Draw the classes and all their details in the same area.
   c. Draw rectangles representing the classes and have the details of individual classes shown in a dialog box on demand.

Using internal frames was considered because internal frames allow the placing of text directly in them as the information was already in text format this seem a viable option. Internal frames also have the added advantage of built-in event handling which allows them to be dragged around and using the mouse with hardly any addition code. But having built an application using internal frames it became evident that the display was not very user friendly. This was because whilst internal frames are adequate for displaying one or two classes when it came to several classes the display became very overcrowded and too unwieldy to use.

Drawing the classes and all their details as shown in figure 2 in page 9 also proved to be a dead end for similar reasons to using internal frames i.e. It would have required an extremely large area to display just a few classes making it difficult for the user to navigate around, not to mention denying the user an overall view of the systems class diagram especially if the classes being displayed have long lists of attributes and operations.

Having explored the first two options and found them wanting, Option (c) to Draw rectangles representing the classes and have the details of individual classes shown in dialog boxes on demand was chosen as it provided the best user-friendly interface and got around the problem of long classes gracefully by using a tree and scrollbars in the dialog box.

Drawing relationships:

When the relationships have been established (see 2. Relationships – establishing links above). The URE creates a new Link object for each relationship. The link object contains references to the two classes it links. Using these references it is able to determine the type of relationship, which determines the type of link it will draw between the classes. The Link objects are also able to keep tabs on the location of the objects they are linking as well as their orientation relative to each other so that it knows how to draw itself i.e. its starting point and its terminating point.

Drawing the complete class diagram with classes and relationships:

The class diagram was constructed and displayed by class DrawingPanel that extended javax.swing.Jpanel. The drawing area makes use of double buffering which greatly reduces flickering during repainting it does this by first drawing the entire area on a buffered image in memory and when the drawing is complete swapping the current screen image with the buffered image. The reason for using double buffering was to reduce flickering during when the class objects were being dragged across the screen.
The drawing area is used to draw the class diagram, which basically consists of rectangles labelled according to the class they represent and the linking lines between them that show how they are interrelated. The user can reposition individual rectangles by dragging them using the mouse (any button) to new positions. The user can also click on a rectangle; doing this brings up a dialog box with the details of the particular class that the rectangle the clicked represents. (See 3 b. dialog boxes below)

The lines between the rectangles are able to sense the orientation of the class rectangles that they link and redraw themselves to reflect this. For example, if initially the link was drawn from the bottom of classA to top of classB when classA was above classB. If classA is repositioned such that it is now below classB then the link is able to detect this change and now draws itself from the bottom of classB to the top of classA e.g. figure 9 below.

![Figure 9: Dynamic link lines.](image)

6.3 b. Dialog boxes

Each ClassInfo object that goes to make up the class diagram has the ability to draw a JDialoag box that contains a tree made up of the details of the class whose information is held by that instance of ClassInfo. A dialog box is displayed when a user clicks on a rectangle (in the class diagram) representing a class. The click could occur in either the main drawing area or in the layout frame (part of the layout manager, see below). Only one dialog box of an instance of ClassInfo can exist at any one time.

Each occurrence of ClassInfo contains its own dialog box. The tree in the dialog box is filled with selected information in the ClassInfo object. The information in the tree can be portioned as follows:

```
<< class name >>
<< attributes >>    - if any.
<< operations >>    - if any.
associations / contains - if any.
generalizations / super-class - if any.
realizations / interfaces - if any
```

(See figures in Appendix E that shows a dialog box in action).

6.3 c. Main Menu.

```
-> File
  Open
```
Used to open class diagrams that had previously been saved as a disk file.

Save
Used to save the current class diagram to disk.

Print
Used to print the current class diagram. Allows the user to choose the print format.

Exit
Closes the UML Reverse Engineer.

-> Classes
Filled with all the classes that make up the current class diagram. It allows the user to show or hide individual classes using the provided checkboxes.

-> Details
Filled initially with preset values which allow the user to show or hide groups of objects e.g. user can hide all the classes whose name begins with java by un-checking the check box next to menu item “java”. Additional values/menu items are added for each package found in the class diagram.

-> Levels
Shows the number of levels in the current class diagram. The user can use show or hide groups of objects by setting the maximum visible level using the radio buttons provided in the menu.

-> Layout
Used to open a new layout frame. Allows the user to choose which program to use when generating the layout. The initially program options where ‘dot’ and ‘neato’ programs, but ‘neato’ was later dropped as it was unable to layout the class diagram in a clear fashion. The user is also able to choose the orientation of the class diagram i.e. top to bottom or left to right the actual elements in the diagram are always right side up.

-> Help
Opens up a dialog box with instruction on how to use the program.

(Appendix D has screenshots of the URE’s menus)

6.3 d. Layout Manager.

Problem:

After drawing the first couple or so test class diagrams, it became evident that a layout manager was needed. This was because despite the ability to move and rearrange the class objects, the class diagrams with every additional class object become increasingly more complex, interconnected, interweaved web of class objects and links. Eventually they become completely unmanageable, as the user would have to spend a very long time to trying to untangle these complex webs.

The solution was to have a layout manager that untangles the class diagrams on behalf of the user. However having looked at developing such a manager from scratch for
the URE, it became apparent that this was way beyond the scope of the project and would have taken more time than was available for the project.  

**Solution:**

With more research and the guidance from my supervisor Dr. Nick Efford, a compromise was found in the shape of “Grappa the Java Graph Package” from AT&T labs-Research. Grappa was chosen primarily because it is written in Java and could therefore be more easily incorporated into the URE as it was then.

“Grappa is a Java graph drawing package that simplifies the inclusion of graph display and manipulation capabilities within Java applications and applets. It has a good number of useful features built into it, but is also extensible. Grappa can be thought of as a port of a subset of Graph to Java. The current version of Grappa was built using the Java Development Kit (JDK) version 1.2 and takes advantage of recent improvements in the JDK such as Java 2D and Swing.”

AT&T labs-Research, last visited 26/08/01.

**Integrating Grappa to the URE:**

By modifying the some of the code that comes with grappa I was able to amalgamate grappa with the URE.

![Diagram](image)

**Figure 10:** Shows how the grappa layout program ‘dot.exe’ is used to layout the class diagram

**Utilising the grappa enabled layout manager:**

Using the main menu’s Layout options the user is able to rearrange the positions of the objects in the class diagram such that there the fewest possible line crossings and overlaps i.e. the best/optimal layout. The rearranged diagram is the placed in a new window frame, the layout frame that has its own menu.
The best laying-out is done with the assistance of the dot program ‘dot.exe’. The program uses heuristics to find the optimum arrangement for the objects making up the class diagram.

The new rearranged diagram retains the ability to bring up the details of a class by clicking on a rectangle with the name of the class whose details you wish to view. It loses the ability to manually reposition the objects using the mouse although this should not be a problem as the diagram is already at its optimum layout and probably does not need rearranging. (See Appendix F for screen shots of the layout manager in action)

Supplementary features include the ability to zoom into particular areas of interest by dragging a bounding box over the area, right clicking the mouse and selecting zoom to sweep on the pop-up menu that appears or simply zooming in and out of the diagram as a whole also using the pop-up menu. The bounding box can also be used to select a group of objects which can then be deleted from that diagram using delete option in the popup menu. The layout frame has a print menu that provides the user with the ability to print out the class diagram in that frame. It also has a refresh menu that allows the user to redraw the diagram thereby maintaining the diagram at its optimal layout. Currently this is done manually but might in the future be automated so that the diagram is always at its optimal layout by taking into account of any changes made to the diagram.

The layout frame was the last part of the program to be added and as such is still under construction as there are still some latent bugs that are yet to be abrogated. This can be best demonstrated by the fact that the dialog boxes don’t always appear when a rectangle on the class diagram is clicked.

![Figure 11: Class Diagram of the URE.](image-url)
7. EVALUATION and TESTING

7.1 Evaluation criteria:

a. Is the solution able to generate UML class diagrams if provided with java class files?
b. If so, is does the diagram conform to UML standards i.e. does the class diagram text satisfy UML syntax requirements and do the drawn elements in the class diagram also meet UML class diagram standards.
c. Is the solution platform independent i.e. how portable is it.
d. How user friendly is the solution.

![Diagram](image)

Figure 12: Proposed method for evaluating the efficacy of The URE.

7.2 Testing

To test if these criteria are met by the URE, the method described in the figure below was devised as an objective way assessing whether the URE was able to reconstruct a class diagram from Java byte code. By comparing the original class diagram with the one reconstructed by the URE it will be possible to ascertain how well the URE was able to reconstruct the UML class diagram.

![Diagram](image)

Figure 12: The original simple class diagram for the dummy classes #1. (Source code in Appendix H Dummy Code #1.)
Figure 13: The class diagram as reconstructed by the URE including the dialog boxes, compare with figure 12 above. (See Appendix I - testing for better close-up screenshots.)

Figure 14: The original simple class diagram for the dummy classes #2. (Source code in Appendix H Dummy code #2)
7.3 Results

By looking at the class diagrams and dialog boxes produced by the URE, it is possible to confirm that the class diagram and UML text generated do conform to UML standards.

The URE is not fully platform independent this is because whilst the URE is entirely written in Java making it platform independent, the URE’s layout manager uses programs the are platform dependent i.e. ‘dot.exe’. However these programs can run from a website thereby maintaining the URE’s platform independence for systems with Internet connections.

The system is very user-friendly. This is because The URE has a simple to use (difficult to build) menu that is clearly labelled and is intuitive to use. It has a main display that allows the user to interact with the class diagram.

8. LIMITATIONS

The main limitation of the URE is that it is limited to the Java language as it relies on Java’s reflection abilities to perform its tasks. This means it would be difficult to extend it to other object based languages like C++.

Using just reflection techniques it is not possible to reconstruct the original class diagram completely as some of the information are simply not available from the compiled java classes. While some of the information can be extracted from the source code e.g. when looking for associations within methods, a good deal of the information is not available is also not recoverable from the source code. As a result
the reconstructed class diagram may not always completely match the original class diagram.
A good number of these discrepancies are due to the non-availability of the information in the implemented solutions. This is because some of the information e.g. composition is conceptual information and does not have a set unambiguous approach of implementation.
The URE can be very demanding on a system’s memory and CPU speed. Especially when analysing class diagrams with a very large number of elements in it thus it is not suited to systems with slow processors and low memory.

9. FURTHER WORK
There are just two known small bugs that are yet to ironed out. The first being, arrays are not displayed properly especially when they are multidimensional arrays. This is because when using reflection arrays have special prefixes and suffixes at either end of their names. These prefixes and suffixes have proved difficult to automatically interpret for multidimensional arrays. The second is in the layout frame of the layout manager. The layout frame does not always respond correctly to clicks especially when trying to open dialog boxes.
The layout manager is platform-dependent as the graph layout program it uses only works in the windows environment. A challenge for the future would be to build wholly in Java a graph layout program thus maintaining the platform independence of the URE.
The URE’s user-friendliness could be improved by allowing the user to select Java .class (files to be reverse engineered) using a graphical file browser, this would make it easier for the user to select multiple files. This will make it easier to analyze large numbers of files as the user would not have to type the in at the command line.

10. CONCLUSION
The primary objective of the project was to develop an application that is able to reverse engineer a UML class diagram from any Java Classes. The UML Reverse Engineer (URE) was built to achieve this objective. The URE was successful in satisfying the primary objective by taking in any compiled Java classes in byte code file format and reverse engineering a UML class diagram from it. It did this by using the tools provides by the Java reflection API to extract pertinent information from the Java classes, translating and interpreting the information so that a UML class diagram could be constructed from the information gathered. The URE then made use of the Java Graphics API to build the URE’s Graphical User Interface (GUI), which was then used to display the class diagram in a highly interactive and user-friendly fashion. The URE’s GUI is packed with many user-friendly features like a comprehensive menu that not only allows the users to change how the class diagram is displayed but it also updates itself so that the user has feed back on the current state of the URE. The URE’s class diagram is also interactive and allows the user to reposition individual elements in the diagram.
The URE goes beyond it scope by providing a layout manager that is able to automatically arrange the elements in the class diagram to achieve the best possible view. The URE also provides printing, saving and loading facilities for the class diagram among other ftures.
As mentioned in the limitations, the URE is limited to only one language Java, this is because Java provides reflection tools without which this project would not be possible as alternatives like developing a Java language parser to parse the Java source code to extract the information required would have been more difficult and thus would have taken more time than was available to implement.
Overall the URE was able to reconstruct a reasonable class diagram given the information available to it. This is because UML diagrams are usually used to first design the software before building it (forward engineered). This means that they contain conceptual information for aid clarity in the documentation generated that are not necessarily present after the software has been built. It is therefore possible to come to the conclusion that because UML diagrams are forward engineered they are not necessarily fully reversible. However the URE proves that UML class diagrams are at least partially reversible.
11. REFERENCES


Deitel, H M, Deitel, P J (1999), Java How to Program, Prentice Hall.

Flanagan David (1999), Java in a nutshell: A Desktop Quick Reference, O’REILLY.


AT&T Labs Research, Grappa

Sun Microsystems Inc, Java™ 2 SDK, Standard Edition Documentation


12. APPENDICES

Appendix A – Reflection on the project.

The project involved considerable programming in Java. I found this aspect of the project challenging, educational and thoroughly enjoyable. Prior to the project I had very little programming experience using Java that I had comparatively recently started using at the start of the MSc course. One of my reasons of choosing this project was to extend my knowledge of Java in a software development environment. I feel that the project helped increase my understanding of Java in not only the Java Reflection and Graphics API(s) but also the Java language as a whole. The project was my first contact with UML this meant that I had to scale a steep learning curve to gain sufficient insight into UML so that I could complete the project. The approach used in developing the solution was to build numerous mini-applications / prototypes to explore how certain parts of the system could be built. Spiral development process that best fits the development process the project used. The prototypes were continuously tested and refined to produce the final solution. The solution itself (the URE) was able to perform the primary tasks that were set out at the beginning of project. By integrating the layout manager, I was able to go beyond the primary objectives of the project. During the project I encountered many challenges like how to determine relationships and then make sure the objects that are related are able to know to whom they are related and the type of relationship. Problems like this helped test and improve my problem solving aptitude especially in the programming aspects of the project. To make the layout manager work on Linux machines required the installation of third party proprietary software and editing some shell scripts. Unfortunately I was unfamiliar with this type of installation and I was therefore unable to make the layout manager work on Linux machines. Overall I feel that the project was a success as I was able to meet both the projects and my personal objectives.
Appendix B – copy of Objectives and Deliverable form.

School of Computer Studies

MSC PROJECT OBJECTIVES AND DELIVERABLES

This form must be completed by the student with the agreement of the supervisor of each project, and submitted to the MSc project co-ordinator (Mrs A. Roberts) by 22nd March 2001. A copy should be given to the supervisor and a copy retained by the student. Amendments to the agreed objectives and deliverables may be made by agreement between the student and the supervisor during the project. Any such revision should be noted on this form. At the end of the project, a copy of this form must be included in the Project Report as an Appendix.

Student: 
Programme of Study: 
Supervisor: 
Title of project: 

External Organisation*: 

*if applicable

AGREED MARKING SCHEME

<table>
<thead>
<tr>
<th>Understand the Problem</th>
<th>Produce a Solution*</th>
<th>Evaluation</th>
<th>Write-Up</th>
<th>Appendix A</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>40</td>
<td>20</td>
<td>15</td>
<td>5</td>
<td>100</td>
</tr>
</tbody>
</table>

* This category includes Professionalism (see handbook)

OVERALL OBJECTIVES (continue overleaf if necessary):

• Learning about UML and how it is used for visual modelling and code generation.

• Explore the features provided by modern C++ tools.

• Improve basic programming skills and the ability to produce well-documented and robust reports.

DELIVERABLE(S):

1. A project report.

Continued over e
Agreed objectives and deliverables (continued):

1. Develop using Java a software prototype that generates code from UML-class diagrams.

Amendments to agreed objectives and deliverables:

<table>
<thead>
<tr>
<th>Date</th>
<th>Amendment</th>
</tr>
</thead>
</table>
Appendix C - marking scheme, header sheet and assessors comments for the marked Interim Report

School of Computer Studies

MSC PROJECT INTERIM REPORT

All MSc students must submit an interim report on their project to the MSc project co-ordinator (Mrs. A. Roberts) by Tuesday 1st May 2001. Note that it may require two or three iterations to agree a suitable report with your supervisor, so you should let him/her have an initial draft well in advance of the deadline. The report should be a maximum of 10 pages long and be attached to this header sheet. It should include:

- the objectives, deliverables and agreed marking scheme
- resources required
- progress report and project schedule
- proposed research methods
- a draft chapter on the literature review and/or an evaluation of tools/techniques
- the WWW document link for the project log to date

The report will be commented upon both by the supervisor and the assessor in order to provide you with feedback on your approach and progress so far.

The submission of this Interim Report is a pre-requisite for proceeding to the main phase of the project.

<table>
<thead>
<tr>
<th>Student:</th>
<th>PETER N. MUCHOGU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programme of Study:</td>
<td>MSc Distributed Multimedia Systems (DMS).</td>
</tr>
<tr>
<td>Title of project:</td>
<td>UML and CASE Tools</td>
</tr>
<tr>
<td>Supervisor:</td>
<td>Dr. NICK EFFORD</td>
</tr>
<tr>
<td>External Company (if appropriate):</td>
<td></td>
</tr>
</tbody>
</table>

AGREED MARKING SCHEME

<table>
<thead>
<tr>
<th>Understand the problem</th>
<th>Produce a solution</th>
<th>Evaluation</th>
<th>Write-up</th>
<th>Appendix A</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>40</td>
<td>20</td>
<td>15</td>
<td>5</td>
<td>100</td>
</tr>
</tbody>
</table>

* This includes professionalism (for details of what is included please see handbook)

Signature of student: [Signature]
Date: 1/05/2001

Supervisor’s and Assessor’s comments overleaf

Peter Muchogu MSc in Distributed Multimedia Systems 34
Supervisor's comments on the Interim Report

Clear objectives.
A reasonable range of aims has been
covered.
The "basic concepts" in less than 1 page,
length, is clearly inadequate.

Assessor's comments on the Interim Report
First apologies on behalf of the School, for
the late return of the form, which I should
only receive the last couple of days.
The objectives often quite clear, but
there needs to be a good evaluation
methodology in order to measure how well
they have been met. (Are you going to be able
to apply your tool to programs of arbitrary length
and complexity? Alternatively, when confident
you set yourself?)

It is a pity that the report is so short.
This makes it difficult to give any more
guidance on style or content.
Appendix D – Screen shots of the URE’s menu

Figure d1: showing the App menu.

Figure d2: showing the classes’ menu filled with all the classes in the current class diagram.
Figure d3: Screen shot showing the Details menu filled with the packages that the classes in the current class diagram fall under.

Figure d4: Screen shot showing the Levels menu. The levels range from zero to the maximum level found, usually the level of java.lang.Object. The radio buttons are used to hide or show levels. They also indicate the maximum visible level.
Figure d5: screenshot of the layout frame. It enables the user to auto arrange the class diagram using the Grappa API in conjunction with the dot program.

Figure d6: screenshot of the help menu. When Readme is clicked, a dialog box containing text is displayed. The text has instruction on how to use the URE.
Appendix E – screen shots of the URE’s dialog boxes in action.

Figure e1: A screenshot of the URE showing the class diagram and two dialog boxes filled with information about two of the classes in the class diagram.

Figure e2: screenshot showing a close-up of a dialog box displaying the attributes and methods of a class ChessPiece.
Appendix F – screenshots of the layout manager in action.

Figure f1: screenshot showing the URE class diagram in the main drawing area.

Figure f2: screenshot showing the result of using Grappa-dot layout manager. (The class diagram is the grappa version of the class diagram in figure d1 above.)
Appendix G - Miscellaneous screenshots.

Figure g1: Using the Detail menu to hide java classes.

Figure g2: Starting the layout manager using the main menu.
Figure g3: The layout manager’s layout frame.

Figure g4

Figure g5

Figure g6

Figures g4, g5 and g6 shows deletion of class diagram elements in the layout frame.
Appendix H - Dummy code used to test the URE

Dummy code #1

```
// file name : ClassA.java
import java.util.Vector;
import javax.swing.*;

public class ClassA extends JFrame {
    private Vector v = new Vector();
    public ClassA (String n) {
        super(n);
    }
    protected void setTest(Vector v01) {
        v = v01;
    }
}

// file name : ClassB.java
public class ClassB extends ClassA implements InterfaceA {
    ClassB() {
        super("new");
    }
    public void draw () {
    }
    public void setLink (ClassC classC) {
        ClassC link = classC;
    }
}

// file name : ClassC.java
public class ClassC {
    ClassC () {
    }
    public void test (){
        System.out.println("in class C");
    }
}
```

Dummy code #2

```
// file name : Node.java
public abstract class Node {
    private String name;
    protected Node (String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
    public Node getChild(int i) {
        return null;
    }
    public void adopt(Node n) {
    }
}

// file name : Directory.java
```
```java
import java.util.Vector;
public class Directory extends Node {
    private Vector nodes;
    public Directory (String n) {
        super(n);
    }
    public Node getChild(int i) {
        return i < nodes.size()?
            (Node)nodes.elementAt(i): null;
    }
    public void adopt(Node n) {
        nodes.addElement(n);
    }
}
// file name : File.java
public class File extends Node {
    public File(String name) {
        super(name);
    }
}
// file name : Link.java
public class Link extends Node {
    Node subject;
    public Link (Node n) {
        super(n.getName());
        subject = n;
    }
    public String getName() {
        return subject.getName();
    }
    public Node getChild(int i) {
        return subject.getChild(i);
    }
    public void adopt(Node n) {
        subject.adopt(n);
    }
}
```
Appendix I – testing

Figure i1: Class diagram for dummy code # 1.

Figure i2: ClassA dialog box.

Figure i3: ClassB dialog box.
Figure i4: ClassC dialog box.

Figure i5: InterfaceA dialog box.

Figure i6: Class diagram for dummy code # 2 drawn in the layout frame.

Figure i7: class File dialog box.
Figure i8: class Node dialog box.

Figure i9: class Directory dialog box.

Figure i10: class Link dialog box.
Appendix J - Reflection and Relationships

Get Class Information using reflection.
*Output*: Vector data containing all the Class and all its super classes up to Object. Information on each class is first stored in UML notation in ClassInfo objects, which are the stored in data.
*Method*: `getClassInfo()`

Get interfaces of each of the ClassInfo Objects also using `getClassInfo()`.
*Output*: Get and add interface objects to Vector v.
*Method*: `getInterfaces()`

Remove duplicate ClassInfo objects
*Output*: Filtered Vector v.
*Method*: `filterClassInfo`

Get associations between the objects. Searches attributes and methods and finally the source code if it is available of classes for associations.
*Output*: Vector setOfAssociations
*Method*: `getAssociations`

Remove and duplicate associations
*Output*: Filtered Vector setOfAssociations.
*Method*: `filterAssoc()`

Find generalisations and realizations. Enumerate all the relationships including associations.
*Output*: Vectors setOfRelate, setOfInterfaces and SetOfAssociations. All these are then added to ClassInfo Objects in Vector V.
*Method*: `setRelationships`

Create the links between objects.
*Output*: Vector setOfLinks used to draw the lines between objects.
*Method*: `setLinks`
Appendix K – Evolution of UML

Figure g1: OMG-UML evolution. (from Document omg/01-03-02 (Updated UML Lecture 1, March 2001 version) by Mr. Cris Kobryn)