Project Report

Eclipse as a Teaching Platform for Kenya

Thomas Timbul
2005

Supervisor	Robert Chatley
Second Marker	Susan Eisenbach

PREFACE

A paper describing the wok done in this project (reference [RCTT]) has been accepted for publication in the proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005) in the Research Demonstrations track.

The work was the recipient of an award under the IBM Eclipse innovation award scheme.

All project data, such as source code, binary distributions, manuals and this report are available for download from http://www.doc.ic.ac.uk/~tt101/kenya/

ABSTRACT

This document introduces an integration of Kenya into the Eclipse IDE as a **plug-in** in order to provide not only the introduction to Java that Kenya already provides, but also to introduce professional features available in Eclipse and thus to enable a smooth transition to the use of professional development tools that are supported by the same environment.

To enhance the pedagogic strength of Kenya by encouraging good programming style, a style guide is integrated, which makes use of various existing platform capabilities to ultimately allow the tool to give automated advice on particular code patterns that can in general be branded as ‘bad style’ or so-called ‘code smells’ and to support and guide the user through the resolution of such common mistakes step-by-step.

ACKNOWLEDGEMENTS

I would like to acknowledge my supervisor, **Robert Chatley**, for his great initial work on the Kenya language.

I also want to thank my second marker, **Susan Eisenbach,** for leading me into the style checking problems in the first place.

Thanks go to **Tristan Allwood** and **Matthew Sackman** for their efforts on Kenya v4.

Finally I should mention **IBM**, the rest of the **Eclipse Consortium** and the **Eclipse community** for creating an excellent development platform and giving me the tools...

CONTENTS

1 INTRODUCTION 1

1.1 Overview 1

1.2 KenyaEclipse 1

1.3 Highlights 3

1.4 Document Structure 3

2 BACKGROUND 5

2.1 Kenya 5

2.2 Kenya IDE 5

2.3 Code smells - style guidance 6

2.3.1 Metrics based refactoring 6

2.3.2 Anti-pattern detection 7

2.3.3 The novice's favourite mistakes 7

2.3.4 Summary 7

2.4 Eclipse IDE 8

2.4.1 Eclipse Plug-in Architecture 8

2.4.2 Platform subsystems 9

2.4.3 Eclipse SDK features 9

2.4.4 Plug-in Development 10

2.4.5 Build your own 13

2.4.6 Existing style checking tools 13

2.4.7 Existing teaching environments or platforms 14

2.5 Summary 15

3 REQUIREMENTS AND SPECIFICATION 17

3.1 Integration of Kenya with Eclipse 18

3.1.1 Kenya perspective 18

3.1.2 Addition of the Kenya Navigator view 18

3.1.3 Addition of a basic Kenya/Java editor 18

3.1.4 Association of the editor with Kenya files 19

3.1.5 Addition of Kenya projects and the Kenya Nature 19

3.1.6 Addition of a Wizard for creating Kenya Projects 19

3.1.7 Icons/banners/images 19

3.1.8 Syntax highlighting 20

3.1.9 Translation from Kenya to Java 20

3.1.10 Errors shown in Problem view and as annotations 20

3.1.11 Hovers for the annotations 21

3.1.12 Addition of a Wizard to create new Kenya files 21

3.1.13 Addition of run support for Kenya files 21

3.1.14 Integration of the debugger 21

3.1.15 Ability to save Java files separately 22

3.1.16 Addition of an editor action menu 22

3.1.17 Addition of a preferences panel 22

3.2 KenyaEclipse Feature Extension 23

3.2.1 Style guidance module 23

3.2.2 Code Assist 23

3.2.3 Reference highlighting 24

3.2.4 Refactoring processor 24

3.2.5 Refactoring: renaming 24

3.2.6 Quick fix proposals (Style) 25

3.2.7 Quick fix proposals (Code) 25

3.2.8 Code collapse 25

3.2.9 Help System Integration 25

4 PREPARATION AND DESIGN 27
4.1 Choice of Eclipse 27
4.2 Preparation 27
4.3 Kenya v4 design modifications 27
4.4 Overall Plug-in package structure 29
5 IMPLEMENTATION 31
5.1 KenyaEclipse Architecture 31
5.1.1 Editor architecture 31
5.1.2 Run/Debug support 35
5.2 Advanced editor features 37
5.2.1 Occurrence Highlighting 37
5.2.2 Code Completion 38
5.2.3 Source Editing Features 39
5.2.4 Refactoring - Renaming 39
5.3 Style Guidance Module (SGM) 39
5.3.1 Independence 39
5.3.2 Extensibility 39
5.3.3 Configurability 40
5.3.4 Efficiency 41
5.3.5 Assistance 41
5.3.6 SGM implementation 42
6 EVALUATION 45
6.1 Quality Assurance 45
6.2 Testing 45
6.3 Results 46
6.3.1 New project and file wizards 46
6.3.2 Compiler 46
6.3.3 Run support 47
6.3.4 Debug support 47
6.3.5 Occurrence Highlighting 47
6.3.6 Code Completion 47
6.3.7 Source Editing Features 48
6.3.8 Refactorings 48
6.3.9 Style guidance module 48
6.4 Unresolved Issues 49
6.4.1 Debugger 49
6.4.2 Binding resolution 49
7 CONCLUSIONS 51
7.1 AST analysis and modification 51
7.2 Plug-in creation 51
7.3 Code Design 51
7.4 Criticism of the KenyaEclipse approach 51
7.5 Benefits of the KenyaEclipse approach 51
7.6 Future Work 52
7.6.1 Missing features 52
7.6.2 Eclipse as a Teaching platform for Kenya 52
8 BIBLIOGRAPHY 53
9 APPENDIX 55

FIGURES

FIGURE 1.2.1 – RESOLUTION EXAMPLE FOR BOOLEAN REDUCTION 2
FIGURE 1.2.2 – RESOLUTION EXAMPLE FOR OMITTED BREAK STATEMENT 2
FIGURE 2.1.1 - EXAMPLE KENYA PROGRAM 5
FIGURE 2.4.1 - SUBSYSTEMS IN ECLIPSE AND PLUG-IN ARCHITECTURE [E-PDG] 8
FIGURE 2.4.2 - MANIFEST FILE CONTENTS FOR ABOVE EXAMPLES 13
FIGURE 4.3.1 - HIGH COUPLING IN KENYA V4 CODE BEFORE REFACTORING 28
FIGURE 4.3.2 - DECREASED COUPLING IN THE KENYA V4 GUI AFTER REFACTORING 29
FIGURE 5.1.1 - KENYAECLIPSE FEATURES BASED ON EXISTING COMPONENTS 31
FIGURE 5.1.2 – MULTIEDITOR COMPONENT 32
FIGURE 5.1.3 – COMPONENT INTERACTION DURING COMPILATION 32
FIGURE 5.1.4 – PROBLEMS VIEW 32
FIGURE 5.1.5 - OVERALL COMPONENT INTERACTION DURING/AFTER COMPILATION 33
FIGURE 5.1.6 – CODE TO INVOKE STYLE CHECKING 34
FIGURE 5.1.7 - EXTENSION POINT MANAGEMENT CODE 34
FIGURE 5.1.8 - LAUNCH SHORTCUTS 35
FIGURE 5.1.9 - LAUNCH CONFIGURATION DIALOG 35
FIGURE 5.1.10 - CORE ECLIPSE DEBUG MODEL ARCHITECTURE 36
FIGURE 5.3.1 - EXTENSION POINT DECLARATIONS IN KENYAECLIPSE 39
FIGURE 5.3.2 - EXTENSION HIERARCHY; EXAMPLE WITH 3 STYLECHECKERS 40
FIGURE 5.3.3 - STYLE GUIDANCE MODULE PROPERTY PAGE 41
FIGURE 5.3.4 - SIMPLIFIED IMPLEMENTATION OF THE STYLE ANALYSIS JOB 42

1 INTRODUCTION

1.1 Overview

Teaching a new programming language such as Java is a challenging task because of the many features that exist in such a language. Many concepts are too complicated to explain at first and thus a cut-down approach is needed.

The Kenya 'teaching language' (see section 2.1) solves this by presenting students with a less complex selection of features and abstracting away from concepts that a novice programmer might not understand immediately.

While knowledge of the language is important, it cannot be denied that the most proficient and efficient programming has come about with the development of tools, such as integrated development environments (IDEs), that allow the programmer to quickly manipulate code and to automate repetitive tasks.

Although this is becoming common knowledge among students, many computing **graduates use notepad to write code and still make beginner's mistakes** because they lacked feedback from when they first learned programming. Provided they look at them, many students still **don't learn from the scribbles on their lab exercise printouts**.

It is desirable that students not only learn programming concepts, but also to make use of the technology provided by professional IDEs to be able to program effectively and at the same time harness its functionality to learn correctly.

Most IDEs, however, have

a) a steep learning curve
b) no or little feedback about mistakes, especially for beginners

This report documents the design and implementation of a plugin for the Eclipse IDE, which

a) smoothes the learning curve for Java by using the Kenya teaching language and eases the user into developing with advanced tools
b) provides automated feedback and guidance for common mistakes through an extensible, configurable and pluggable module

1.2 KenyaEclipse

The next version of the Kenya IDE (see section 2.2) integrates with the Eclipse IDE as a plugin to provide a smooth, consistent interface and an effective training-ground for both Java and Eclipse. It makes use of the platform's capabilities to realise (in addition to previously available features in Kenya) the following functionalities:

- Code completion
- Entity (variables, methods, classes) occurrence highlighting
- Source editing tools (toggle comment, correct indentation, etc.)
- Refactoring (entity renaming)

Students are introduced to these more advanced features early on and promoted to make use of programming tools that are designed to make a programmer's job easier.

The main concern, however, lies with the students' style of programming. A number of bad style patterns have been singled out to be the most commonly made mistakes by beginners. To aid programming tutors, KenyaEclipse contains a Style Guidance Module that allows for addition and removal, simple configuration of checks for style patterns and makes it possible to generate automated advice on style as the user types.

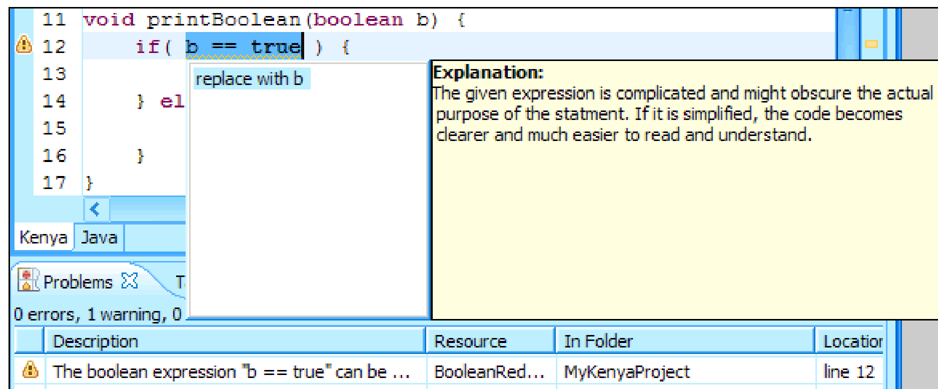

Figure 1.2.1 - Resolution example for boolean reduction

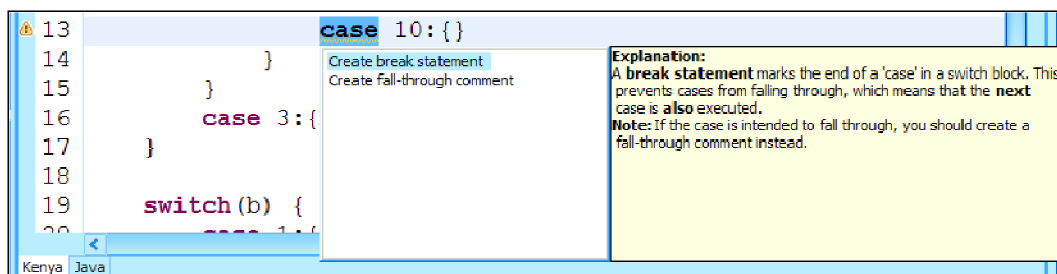

Figure 1.2.2 - Resolution example for omitted break statement

Comprehensive explanations and graphical hints on the user interface as the ones shown in Figures 1.2.1 and 1.2.2 above will help students improve their coding technique and correct mistakes 'as soon as they appear', as opposed to being notified through reduced grades. In the learning process, this removes the ability to learn discouraged practices without being challenged about it.

1.3 Highlights

While the entire document contains most of the knowledge embedded in KenyaEclipse (the remainder is in the code), there are various sections that should be of particular interest to the reader:

(section)	(page)	
2.1	- 5 -	**introduction to the Kenya teaching language**
2.2	- 5 -	introduction to the Kenya IDE
2.3	- 6 -	background of style checking
3	- 17 -	the original feature specifications
5	- 31 -	implementation details for the most intricate features
5.3	- 39 -	**Implementation of the Style Guidance Module**
6.3	- 46 -	Results and comments on the implementation
6.4	- 49 -	Unresolved issues
7	- 51 -	**Conclusions and criticism of the overall approach**
7.6	- 52 -	**Future ideas for creating a true teaching platform**

1.4 Document Structure

This document is divided into 7 chapters, each containing a number of sub-sections.

While chapter 2 (pages 5-15) sets the scene by introducing relevant technology, chapter 3 (pages 17-25) gives the initial specification to which the project was undertaken. Details of the actual implementation can be found in chapter 5 (pages 31-39).

Chapters 6 (pages 45-49) and 7 (pages 51-52) top off by giving detailed analysis of the achievements of the project and conclusions that may be drawn from it and finally pose interesting ways of attempting to create an even better teaching platform.

Throughout this document, the ‘Eclipse IDE’ is freely referred to as ‘Eclipse’. ‘Kenya’ may refer to the language ‘Kenya’ or freely to the ‘Kenya IDE’ unless there may be ambiguity, in which case the difference shall be made clear.

2 BACKGROUND

2.1 Kenya

Kenya is a programming language that was developed by Robert Chatley at Imperial College in 2000-2001. Kenya serves to simplify learning Java in a university environment. It captures the basic features of Java, but abstracts away from its object-oriented nature by removing the need to worry about various language features that would impede the student's ability to comprehend, such as packages, classes, class imports or access modifiers. This enables the first time programmer to concentrate more on the program flow and the algorithmic side of programming while also being introduced to the general syntax of Java.

An example Kenya program that prints a String, depending on its arguments, is given in Figure 2.1.1.

```
const String hi = "Hello World";

void main(String[] args) {

  if(args.length>0) {
    if(args[0]== "yes") {
      println(hi);
    } else {
      println("I am not supposed to print this" +
              ", but I will anyway: " + hi);
    }
  }
}
```

Figure 2.1.1 - Example Kenya program

Kenya has been successfully used for teaching at Imperial College since it was introduced to the computing laboratories and greatly eases the process of getting used to Java. It is also accepted by the students as it does not merely serve as a teaching language without a purpose beyond the course, but rather does it lead on to the full blown feature set of Java which is then an extension to already acquired knowledge when the syntax of the language is not an issue anymore.

2.2 Kenya IDE

The original version of this editor was written by Robert Chatley. It consisted of a basic text editor with the ability to translate written Kenya code into valid Java code, which could be displayed on the screen and executed from within the application. Users can thus see the correspondence between the two languages.

Since then, the Kenya IDE has been maintained and upgraded over the years and has now reached version 4, when it was completely rewritten during the summer period of 2004. Matthew Sackman and Tristan Allwood created a more aesthetically pleasing version of the editor. This new version supports stepwise debugging and breakpoints (added in Kenya v3), various features from Java 1.5 as well as error highlighting and more user-friendly error comments.

This version was rebuilt on top of the Standard Widget Toolkit (SWT), which is the same graphical toolkit that Eclipse uses, instead of the previously used *Swing* framework.

SWT features a platform independent API while providing tight coupling with the native window system. This means that the application looks like a native application, thus providing responsiveness and familiar behaviour, without developer's special action.

2.3 Code smells - style guidance

The most common problem that programming tutors face is not that of non-working code, but rather the fact that some students insist on using a programming style that might be considered awkward or, in particular cases, simply 'bad'.

The IDE should ideally act as a guide and tutor which can point these 'code smells' out to the student and assist her in the resolution of the potential problems. Code smells as described by Fowler and Beck are not necessarily mistakes, but an indication of some structural problems in the code that could give rise to other problems that have a more significant impact. [R-MF]

There are mainly two different ways in which these smells can be detected. Both methods lead to the basic concept of **refactoring**, which is the transformation of code in a way that

a. is meaning preserving
b. minimises the chance of introducing bugs
c. makes the code cleaner and increases readability

[SE4, R-MF]

2.3.1 Metrics based refactoring

Metrics are essentially measures that are supposed to indicate good or bad style in a program, such as number of lines in a class, the length of method bodies or the number of parameters passed to constructors and methods. More complicated examples would be the calculation of the 'Lack of Cohesion of Methods' or the 'Specialisation Index'.

Distance based cohesion is described by Simon, Steinbruckner, and Lewerentz in their paper on Metrics based refactoring [R-SSL]. Many of the particular metrics can be found in the CheckStyle plug-in for Eclipse [CS].

These metrics provide an indication of there something being wrong and can serve as a guide to improve code and thus reduce the number of mistakes that might already be present or be introduced by someone (including the author) editing the code at a later stage.

Metrics are usually a quantifiable measure of some sort. The length of method bodies for example could indicate that the method in question does too much and might be split into smaller functions to improve readability and separation of concerns.

Code smells identified by metrics can be calculated easily, but it is often very hard to give a specific solution to the problem [R-GC].

Because of the size of the average Kenya program and the guidance received by tutors and lecturers, this approach may not provide a lot of material for style guidance, especially since the more advanced checks are not applicable to Kenya at all (problems related to multiple classes for example).

2.3.2 Anti-pattern detection

Anti-patterns describe the usage of certain code patterns that are generally recognised as being 'bad' coding style (thus *anti*-pattern). A numerical approach cannot be used here, but rather the structure of the program needs to be analysed for the existence of such patterns. The more advanced refactorings that are described in Fowler's book are based on the detection and correction of their cause.

Grant and Cordy's approach [R-GC] deals with some form of pattern detection in the source code and ways of suggesting improvements. One way of detecting patterns is by analysing the Abstract Syntax Tree (AST) representation of the program itself.

Most of the problematic patterns are much harder to detect as they are not based on some calculation. However, as opposed to metrics based problems, bad code patterns can often be remedied by a simple yet effective transformation.

2.3.3 The novice's favourite mistakes

Since we are dealing with novice programmers, it is important to see that a lot of the smells and matching refactorings in the book and proposed by Ziring [NJP] do not actually apply to the user of Kenya as it does not have the same features as Java. The Kenya compiler also flags many things that are permitted syntactically in Java, but could cause unexpected runtime behaviour.

However, a beginner can make various mistakes when first working with Kenya/Java and might not own the intuition that would lead him to his mistakes and allow a better correction than the tool [R-MF].

The Style guidance module proposed herein should therefore address these 'favourite mistakes' and alert the user to their presence. The focus should be on small things that go wrong easily, but tend to take a long time to track down, but broader issues that play a role in business could also be catered for. The aim is then to guide the user through the resolution as an effective way of learning how to avoid those situations.

2.3.4 Summary

There are different ways of detecting bad code. Not all the bad style patterns that are generally recognised as such apply to users of Kenya because of its simplicity, so the list of classic mistakes that one could attempt to tackle in Kenya might include:

- Over-complicating boolean expressions
- Usage of 'magic Strings, chars or numbers'
- Creating infinite loops
- Forgetting to `break` cases in a switch
- Returning temporary variables
- Forgetting to initialise members of an Object array
- Referencing nonexistent Array indexes
- Forgetting that Strings are immutable
- Metrics based problems: method length, number of method parameters

Appendix A contains a full list, detailed descriptions and examples of bad style patterns, while Appendix B addresses those actually implemented in KenyaEclipse.

2.4 Eclipse IDE

The following is an excerpt from the [E-PDG] Eclipse Platform Plug-In Developer Guide:

“Eclipse is a platform that has been designed from the ground up for building integrated web and application development tooling. By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a **plug-in** model.”

2.4.1 Eclipse Plug-in Architecture

[E-PDG]

The Eclipse platform is structured around the concept of **plug-ins**. Plug-ins are structured bundles of code and/or data that contribute function to the system. Function can be contributed in the form of code libraries (Java classes with public API), platform **extensions**, or even documentation. Plug-ins can define **extension points**, well-defined places where other plug-ins can add functionality.

Each subsystem in the platform is itself structured as a set of plug-ins that implement some key function. Some plug-ins add visible features to the platform using the extension model. Others supply class libraries that can be used to implement system extensions.

The Eclipse SDK includes the basic platform plus two major tools that are useful for plug-in development. The Java development tools (JDT) implement a full featured Java development environment. The Plug-in Developer Environment (PDE) adds specialized tools that streamline the development of plug-ins and extensions.

Figure 2.4.1 shows the Eclipse SDK with the Platform and its subcomponents and how these two major plug-ins hook into it.

These tools not only serve a useful purpose, but also provide a great example of how new tools can be added to the platform by building plug-ins that extend the system.

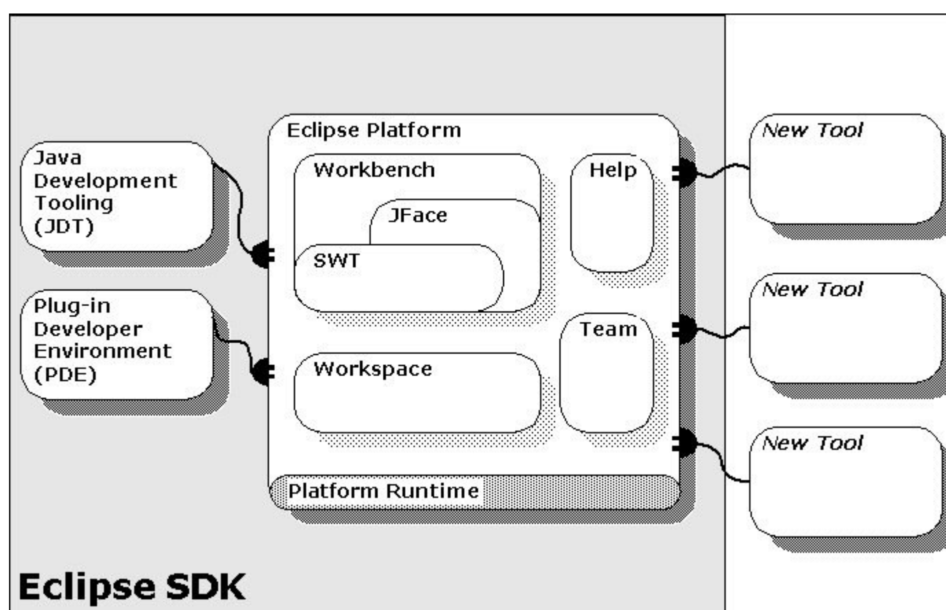

Figure 2.4.1 - Subsystems in Eclipse and plug-in architecture [E-PDG]

2.4.2 Platform subsystems

Figure 2.4.1 illustrates the sub-components (subsystems) of the Eclipse platform. Each of these components is contributed as a group of plug-ins, so-called features, and provides common functionality that is available for extension by any plug-in such as the JDT or PDE. The individual plug-ins hook into the same basic extension points that third party developers would use.
The purpose of each of these subsystems is detailed in Table 2.4.1.

2.4.2.1 Resource management (workspace)	Defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.
2.4.2.2 Workbench UI	Implements the user cockpit for navigating the platform. It defines extension points for adding UI components such as views or menu actions. It supplies additional toolkits (JFace and SWT) for building user interfaces. The UI services are structured so that a subset of the UI plug-ins can be used to build rich client applications that are independent of the resource management and workspace model. IDE-centric plug-ins define additional function for navigating and manipulating resources.
2.4.2.3 Help system	Defines extension points for plug-ins to provide help or other documentation as browsable books.
2.4.2.4 Team support	Defines a team programming model for managing and versioning resources.
2.4.2.5 Debug support	Defines a language independent debug model and UI classes for building debuggers and launchers.
2.4.2.6 Other utilities	Other utility plug-ins supply function such as searching and comparing resources, performing builds using XML configuration files, and dynamically updating the platform from a server.

Table 2.4.1 - Functionality provided by subsystems of the Eclipse platform [E-PDG]

2.4.3 Eclipse SDK features

Subsequently, I will not distinguish between features or plug-ins in terms of naming and refer to features simply as ‘plug-ins’ if no ambiguity arises.

The two SDK features shown in Figure 2.4.1 are the JDT (Java Development Tools) and the PDE (Plug-in Development Environment).

2.4.3.1 Java Development Tooling (JDT)

This feature adds Java editing capability to the platform. The implementation includes a fully-featured Java development environment and provides specialised features for handling, manipulating, compiling, executing and debugging Java code. It defines its own extension points that enable tool developers to use Java capability in their plug-ins. An example would be the ability to use the Java Model that defines the package, field and method definitions of a Java project.

2.4.3.2 Plug-in Development Environment (PDE)

This feature provides convenience for creating plug-ins, a task that is filled with repetitive and laborious actions. It greatly simplifies creation, manipulation, debugging and deployment of plug-ins by automating some of the processes involved as well as presenting plug-in information at a much higher level than its XML meta file (see section 2.4.4.2).

The meta file stores details about the plug-in properties such as the used extension points, external libraries, build configuration and deployment options. Manually editing this file would be error prone and tedious, thus using PDE is much preferred.

The outlined architecture will be developed using the PDE, allowing it to be built within Eclipse itself. From version 3.0 onwards PDE allows the launch of a separate workbench to test and debug the plug-in under development. The development code is seamlessly and automatically deployed into the new environment and can be used immediately.

2.4.4 Plug-in Development

[E-PDE]

Because of the way plug-ins are loaded, they can be extremely flexible. However they must follow a particular structure so that the platform core can detect and load them properly.

First of all a plug-in has to be deployed in a certain manner (outside the development environment). The plug-in is delivered inside a named directory. Typically this contains the name of the plug-in itself as well as its version number. This directory is placed inside the `plugins` folder in the installation directory of Eclipse so that it can be detected and executed.

A plug-in consists of the following items:

- JAR librar(ies) containing compiled Java code
- Plug-in descriptor file (`plugin.xml`)
- Manifest file (`META-INF/manifest.mf`)
- Other external resources

2.4.4.1 JAR librar(ies) containing compiled Java code

A JAR library or archive is used to package the code required to run the plug-in. It is not compulsory to include code and there are examples of plug-ins that do not have any compiled code attached to them. On the other hand it is possible to supply more than one archive containing code.

2.4.4.2 Plug-in descriptor file (plugin.xml)

The plugin.xml file provides Meta level information to the platform as to what existing extensions points the plug-in uses, and what new extension points it declares, and thus how it is to be integrated into the platform. Other plug-ins can then for example access the newly defined extension points and use them to build other functionality.

The content also describes how and when code is run. Certain extensions can be specified to only act under certain conditions and act on particular items.

The descriptor file is a necessary part of a plug-in. It is formatted as an XML file, however the PDE creates a series of views that allow editing the file at a higher, graphical level as mentioned in section 2.4.3.2.

The file divides into various sections, which are detailed below.

2.4.4.2.1 Header

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
  id="org.eclipse.jdt.ui"
  name="Java Development Tools UI"
  version="3.0.1"
  provider-name="Eclipse.org"
  class="org.eclipse.jdt.internal.ui.JavaPlugin">
```

The plug-in is identified by the platform through a fully qualified, unique id, while the user is presented with a readable name. The header may also specify the version of the plug-in. The most important attribute is probably the `class` attribute. It references a class in the JAR library bundled with the plug-in. This class contains the code that will activate the plug-in and its functionality.

2.4.4.2.2 Runtime

```
<runtime>
  <library name="jdt.jar">
    <export name="*"/>
  </library>
</runtime>
```

The runtime declaration defines how the code is packaged into JAR archives (libraries). In this particular case a single archive is used to package the entire plug-in contents.

2.4.4.2.3 Dependencies

```
<requires>
  <import plugin="org.eclipse.ui"/>
  <import plugin="org.eclipse.ui.console"/>
  <import plugin="org.eclipse.help"/>
  <import plugin="org.eclipse.core.expressions"/>
  <import plugin="org.eclipse.core.resources"/>
  <import plugin="org.eclipse.jdt.core"/>
  […]
</requires>
```

This section contains information about the dependency on other plug-ins. Note that this is different from the individual extension points. Each reference here is to the ID of another plugin which was declared in the header as described above.

2.4.4.2.4 Extension point definition

```
<extension-point
  id="javaEditorTextHovers"
  name="%javaEditorTextHoversName"
  schema="schema/javaEditorTextHovers.exsd"
/>
```

The extension point declared would conform to the named extension-point schema (`.exsd`). I will not go into the details of this, as it is beyond the scope of this project. For more information on extension point definitions, you might refer to [E-PDE]. Extension points can be used by other plug-ins as well as by the declaring plug-in itself.

2.4.4.2.5 Extension definition

```
<extension point="org.eclipse.ui.perspectives">
  <perspective
    name="Java"
    icon="icons/full/eview16/jperspective.gif"
    class="org.eclipse.jdt.internal.ui.JavaPerspectiveFactory"
    id="org.eclipse.jdt.ui.JavaPerspective">
  </perspective>
  […]
</extension>
```

This is the 'meat' of the plug-in description. It contains information about the functionality that this plug-in provides in the form of extensions. Extending a different extension point may require different information to be supplied in this tag. In the example a perspective is added (namely the Java perspective) by extending the **org.eclipse.ui.perspectives** extension point. The perspective receives an `icon`, which is shown in the workbench and a `class` attribute which defines the class that is instantiated and called when the perspective is activated.

2.4.4.3 Manifest file (META-INF/manifest.mf)

The manifest file is automatically generated and is empty apart from the first line if all the above is declared inside the plugin.xml file. It is however possible to shift the higher level meta data into the manifest file and is in fact the standard method from Eclipse 3 onwards. This means that the only data declared in the plug-in descriptor are the actual extension and extension-point declarations.

All the 'administrative' data would instead reside in the manifest. The above information could thus be written as shown in Figure 2.4.2. The correspondence between items should be quite clear.

```
Manifest-Version: 1.0
Bundle-Name: Java Development Tools UI
Bundle-SymbolicName: org.eclipse.jdt.ui
Bundle-Version: 3.0.1
Bundle-ClassPath: jdt.jar
Bundle-Activator: org.eclipse.jdt.internal.ui.JavaPlugin
Require-Bundle:
 org.eclipse.ui,
 org.eclipse.ui.console,
 org.eclipse.help,
 org.eclipse.core.expressions,
 org.eclipse.core.runtime,
 org.eclipse.jdt.core,
 […]
Bundle-Vendor: Eclipse.org
```

Figure 2.4.2 - Manifest file contents for above examples

2.4.4.4 Other external resources

It is common to include other resources in the plug-in. This could be HTML help pages, icons, images or other multi-media content, but could also be binary files that are required for plug-in execution.

2.4.5 Build your own

There are various points to note when building any kind of plug-in for Eclipse.

- Decide on the kind of integration between tool and platform
- Identify extension points that can/need to be used to supply the desired functionality
- Implement functionality using these points and according to their definition
- Create the plug-in descriptor and manifest files that make the plug-in accessible by the platform

2.4.6 Existing style checking tools

2.4.6.1 EclipseMetrics

http://metrics.sourceforge.net

This Eclipse plug-in calculates various metrics after a Java project compiles and displays them in a table view on the screen. The plug-in can be configured to respect 'safe' value ranges for each metric in the sense that that range will not be marked with a warning. The tool also allows the user to export the collected data to XML.

Another nice thing that is included in this tool is the dependency graph viewer. This view creates a graphical representation of the package or class interdependency in the Java project that is examined. It displays code tangles and highly coupled code in different colours so that the user is made aware of the situation.

This tool provides purely metrics based checking and does not offer any kind of assistance for improving the code design. However, for most of the provided metrics, some of the refactorings built into the JDT would suffice to improve the design.

2.4.6.2 Checkstyle for Eclipse

http://checkstyle.sourceforge.net

"Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard." [CS]

This tool is configurable to check with a particular coding standard. The checks do not only involve metrics, but also comment placement, indentation, variable naming conventions, whitespace and the organisation of import statements.

Checkstyle also supports some more advanced techniques and is capable of detecting duplicated code, obsolete braces and various other patterns.

Again, there is currently no facility to automatically correct the findings.

2.4.7 Existing teaching environments or platforms

2.4.7.1 GILD (Groupware enabled Integrated Learning and Development)

[GILD, GILD1]

GILD is being developed at the University of Victoria in California. It has been awarded an Eclipse grant and is partially worked on by IBM developers.

Based on Eclipse, GILD is geared towards support for teaching programming from within a single environment, as opposed to traditional tools, which "tend to place the students (and indeed the instructors) outside the domain of the development environment and instead trap them in a Web browser or in a presentation tool".

It aims to be easy to adopt for instructors by allowing them to, for instance, edit web material from within its own context and dynamically updating the corresponding web pages with typed code, thus, in this scenario, avoiding syntactical errors easily introduced into course material, which subsequently causes students much grief.

There is also integrated support for marking labwork and the ability to communicate between students and supervisors using a messaging system. The featured tool 'Dr. Java' allows dynamic execution of single Java statements to create better understanding of code.

GILD is an Eclipse plug-in, tightly integrated with JDT, and is deployed via an installer that includes both the Eclipse platform and a Java JVM and is thus easy to setup.

There seems to not be a great deal of support for helping individuals (other than those on a specific course by the university) learn programming. All pedagogical support is integrated using tooltips, but no solution paths are offered.

2.4.7.2 Class Compass

[CC]

Although Class Compass is not strictly a development tool, it is aimed at student-teacher relationship and the problems that arise from lack of communication.

Class Compass introduces “a set of easy to use online tools for teachers and students to interact on a one to one basis, promote learning equality and explore the bounds of their imaginations and capabilities”.

It essentially aims to encourage student-teacher as well as student-student communication to aid the learning process. It allows participants to create a distinct web-presence for themselves, where they may share knowledge through galleries, forums and even file-sharing.

2.5 Summary

Although tools exist for Java to alert to style issues and other errors, they do not offer the ability to automate the correction of these problems. They are not aimed at programming beginners.

There are various different platforms with the goal of improving the quality of teaching.

The GILD tool is aimed directly at students learning programming with Java and focuses much on the interaction between instructors and students, although limited automated feedback is available.

The Class Compass set of tools has its client base in schools rather than universities and does not have a specific subject attached. It seeks to improve communication by letting students and teachers share course-related information and knowledge over the internet.

Eclipse provides the necessary infrastructure to extend the IDEs functionality so that it can handle Kenya file creation, editing, running and debugging, as well as allowing the integration of a system to (partially) automate style error detection and correction.

The complete list of features is detailed in the specification in chapter 3 together with the extension points that they need to use for integration with Eclipse.

3 REQUIREMENTS AND SPECIFICATION

This section is intended to clearly lay out the specification for the implementation of the plug-in in terms of its requirements as well as the user interface (UI) design and the features that are to be implemented.

The implementation consists of a plug-in to guide users in the use of the programming language Kenya and aids teaching of the language by providing a clear user interface that provides assistance for errors as well as in stylistic matters.

The project can essentially be split into two main phases:

I **Integration of Kenya with Eclipse**
This part is concerned with implementing all the features of the current Kenya IDE as a plug-in in Eclipse.

II **KenyaEclipse Feature Extension**
After creating an extensible base within Eclipse, new features can be added, such as the proposed style guidance module or code completion proposals.

All items are intended to be basic project deliverables unless stated otherwise.

Time plans for both parts can be found in Appendix E.

3.1 Integration of Kenya with Eclipse

The implementation requires the addition and implementation of the following items.

Some of these correspond directly to existing elements while others may be an adaptation to do it 'the Eclipse way'. There are also some items that do not yet exist in Kenya. These are basic, but are not essential and are thus marked as extension features.

3.1.1 Kenya perspective

Description:	This should simply provide a customised selection of views that are directly related to working with Kenya. Essentially this encapsulates Kenya functionality in its own context.	
Implementation:	Extension point	**org.eclipse.ui.perspectives**
Testing:	« Perspective is selectable in the 'open perspective' menu. »	

This item also serves as a 'Get-To-Know-Eclipse'.

3.1.2 Addition of the Kenya Navigator view

Description:	The Kenya Navigator is the counterpart to the Package Explorer in JDT. It should have the ability to hide files that are not relevant to a Kenya user. In particular Kenya files (file ending 'k') and Java files (file ending 'java') should be visible. It is debatable whether only Kenya projects (see point 0 below) should be visible in order to simplify the view further.	
Implementation:	Extension points	**org.eclipse.ui.views** **org.eclipse.ui.resourceFilters**
Testing:	« The Kenya Navigator is selectable from the 'open view' menu. It displays the resource tree, hiding any projects and files that are not relevant to Kenya. It otherwise performs exactly like the basic Resource Navigator. »	

This item also serves as a 'Get-To-Know-Eclipse'.

3.1.3 Addition of a basic Kenya/Java editor

Description:	The basic editor should have two tabs: one for editing Kenya code and the other to inspect the resulting Java code.	
Implementation:	Extension point	**org.eclipse.ui.editors**
Testing:	« The editor can be used for typing and saving. It supports all basic edit functions such as copy, paste, undo, etc. »	

3.1.4 Association of the editor with Kenya files

Description: The editor should by default be activated and used for Kenya files (file ending 'k') and not for any other file type.

Implementation: This is part of the extension definition for the basic editor

Testing: « The editor opens as the default editor whenever a Kenya file is double clicked in the navigator. »

3.1.5 Addition of Kenya projects and the Kenya Nature

Description: A nature is added to a project to tag it as special. Java projects are tagged with the JavaNature so that the platform knows to use the Java perspective when dealing with these projects.
In this case we would like Kenya projects to be associated with our Kenya perspective. In order to do that we define our own nature.
Note that the existence of a 'Kenya project' is thus just a concept. The only thing that really exists is a basic project, however it has been marked as a 'Kenya project' by using the Kenya nature.

Implementation: Extension point **org.eclipse.core.resources.natures**

Testing: « This item cannot be tested on its own. »

3.1.6 Addition of a Wizard for creating Kenya Projects

Description: There is no way that a user can edit files unless she first creates a project in the workspace. This wizard is responsible for creating a project and adding the Kenya nature to it.

Implementation: Extension point **org.eclipse.ui.newWizards**

Testing: « The wizard is selectable from the 'new' menu. It creates a new project with the Kenya nature (see above) which is then visible in the navigator. »

3.1.7 Icons/banners/images

Description: Eclipse uses icons and images where appropriate to decorate visible items on the screen. The items used have consistent meaning throughout the platform. In order to make KenyaEclipse look consistent with the rest of the application, icons are necessary to decorate the functional items described in this section.

Implementation: Requires the use of some graphical tool to create artwork that is similar to that used in JDT.

Testing: « Each of the above, where applicable, has artwork associated with it, that is visible on the screen and is consistent with the artwork used by other Eclipse components. »

3.1.8 Syntax highlighting

Description: Syntax highlighting is one of the main purposes of using a specialised IDE in the first place. Language keywords are highlighted in a particular colour to visually represent their meaning in the code and to make them stand out.

Implementation: It may be possible to reuse the methods used in the current Kenya implementation. Otherwise, there are methods available in the editor implementation that allow a document to be partitioned and each partition to be associated with a highlighter.

Testing: « All test programs are correctly highlighted with respect to language keywords, operators and constants. »

3.1.9 Translation from Kenya to Java

Description: The editor contains two parts: the Kenya source editor and the Java source viewer. The Kenya code needs to be compiled into equivalent Java code periodically to keep the Java view updated.

The current implementation uses a one second timeout after which parsing starts. If the user types during this time, the timeout is reset. The same approach should be used in KenyaEclipse.

Implementation: This may require alterations to the compilation system in Kenya so that existing functionality can be reused.

Testing: « All test programs are translated in exactly the same way as they are in the standalone Kenya IDE that is installed in the computing laboratories at Imperial College. »

3.1.10 Errors shown in Problem view and as annotations

Description: The Kenya IDE displays errors and warnings in the lower part of the screen. A click reveals the location of the error and squiggles under the code also alert the user to their presence.

The JDT has a similar feature in that errors are shown in the Problems view and rulers at both sides of the editor window signal their presence in the document. Squiggles are also used here.

The KenyaEclipse implementation should provide the same feature and give the ability to display errors in the same way that the JDT does.

Implementation: Requires the use of Markers (**org.eclipse.core.resources.markers**) and the AnnotationModel (provided by the editor implementation).

Testing: « For each error or warning in the code, there is an entry in the Problem view containing the error message and its location in the code. Furthermore, for each error or warning there is a graphical indication in the editor, both on the rulers at the sides as well as squiggles in the text. »

3.1.11 Hovers for the annotations

Description: When the user hovers over an annotation marker a hover text should summarise the error message(s) to save her from having to look into the Problem view for a description.

Implementation: There is no extension point for this, but since the JDT has this functionality, there will be some existing mechanism to facilitate this.

Testing: « When hovering over an annotation on one of the rulers, a tool-tip like hover appears containing the error message(s) belonging to the annotation (error) in question. »

3.1.12 Addition of a Wizard to create new Kenya files

Description: In Eclipse files are created in the workspace before they are edited (unlike in many other editors). This wizard is intended to create a simple Kenya source code file.

To facilitate code creation, the user should be able to select whether he wishes to create a `main` method (with or without arguments), since every Kenya program requires one to be present.

Implementation: Extension point **org.eclipse.ui.newWizards**

Testing: « The wizard is selectable from the 'new' menu and on finishing creates a file in the specified directory, with the specified name and containing a main method as chosen by the user. »

3.1.13 Addition of run support for Kenya files

Description: One of the main features of Kenya is the ability to compile and launch the program from within the IDE. The JDT also offers this capability for Java.

In the KenyaEclipse implementation, launching should also be possible in the same way that it is in JDT. This includes being able to configure application parameters, working directory and so on.

Implementation: Extension point **org.eclipse.debug.core.launchConfigurationTypes**

Testing: « It is possible to create a launch configuration that will execute the selected Kenya program in the same way as the Kenya IDE does. »

3.1.14 Integration of the debugger

Description: Similar to run support, debugging is another main advantage of using an IDE. Eclipse offers a debugging framework which should be used to incorporate the debugger to present a consistent interface

Implementation: Various extension points in **org.eclipse.debug.core**

Testing: « It is possible to launch the run configuration in debug mode such that detailed inspection of the execution is possible in a similar way as it is in the Kenya IDE. »

3.1.15 Ability to save Java files separately

Description: The user should be able to save the contents of the Java editor (if it contains valid code) to be able to use it elsewhere.

Implementation: This could be trivial or complicated depending on how complicated the translation mechanism is designed. It might be done as an export wizard or as a 'Save as' option. Alternatively, the Java file might be created during translation from Kenya to Java.

This is an extension feature

Testing: « The Java editor's (valid) contents can be saved to a file by some means. »

3.1.16 Addition of an editor action menu

Description: Add all the Kenya editor specific actions to a menu, so that they can be selected and run by the user.

Implementation: Extension point **org.eclipse.ui.actionSets**

This is an extension feature

Testing: « Each action is selectable from the menu when it is appropriate for that action to be enabled. Each action performs the operation it was designed to do. »

3.1.17 Addition of a preferences panel

Description: To stay consistent with existing Eclipse plug-ins, KenyaEclipse should be as configurable as possible. The preference panel simply visualises the configurable options.

Implementation: Extension point **org.eclipse.ui.preferencePanels**

This is an extension feature

Testing: « The panel contains all configurable options that are relevant to KenyaEclipse in a suitable layout. »

3.2 KenyaEclipse Feature Extension

This section details additional features which do not yet exist in the current Kenya IDE, but will be included in order to provide some of the advanced functionality that JDT has to offer, as well as the style guidance module to improve the ability to teach using this tool and its effect.

The items in this section are not to be confused with the extension part of the project. Most notably the inclusion of the style guidance module is a basic product feature. Project extension features are specially marked as such.

3.2.1 Style guidance module

Description: The style guidance module is designed to detect predefined code smells at compile time and alert the user to their presence.

Implementation: This item requires the use of various extension points. There may be requirements for marker types different from the 'regular' problem markers. Separate preferences should also be available to enable or disable checks.

The implementation should be generic enough to allow other developers to implement and add other checks that can be detected by this module.

An idea would be to create the module as a separate plug-in which defines extension points to easily add code smell definitions.

Testing: « For each code smell described in Appendix A, the smell is detected and highlighted in the editor. Individual checks can be enabled or disabled using preferences. »

3.2.2 Code Assist

Description: Code assist gives the user the ability to choose keywords or variables that she wishes to insert from a table when a particular key combination is pressed. The help provided is context sensitive and provides only legal choices. This feature allows the user to develop more rapidly by not having to type as much.
Furthermore, the user can query the legal input at the current position in the editor, simply by triggering the code assist action.

Implementation: JDT implements this feature. There is also an editor example available that explores this feature.

This is an extension feature

Testing: « In different syntactical positions in the code, triggering the code-assist will render a table listing valid input at that position such as available method calls, variables or keywords. »

3.2.3 Reference highlighting

Description: The intent is to graphically highlight all references to a particular variable or method that the user selects in the code. This allows a quick visual overview of places that could be affected by changing the variable or method.

Implementation: This could be done using graphical methods that are provided by the SWT widget representing the text in the editor. It also needs a way of determining the scope of a variable to prevent false positives.

Testing: « Enabling this feature will cause all references to types, methods or variables to be highlighted in the editor when such is selected by the user. Variable highlighting occurs with consideration of the variable's scope. »

3.2.4 Refactoring processor

Description: Refactoring allows users to modify the representation of the code without changing its semantics. To make refactorings possible, there needs to be some kind of component to enable this process.

This feature would be invisible to the user as such, but it allows the implementation of more advanced components, such as the next three listed below.

Implementation: This kind of functionality exists within the Eclipse feature **org.eclipse.ltk.** Hopefully the provided components can be used to implement this component, which would store a representation of Kenya code and provide methods to manipulate this code.

Testing: See: Refactoring: renaming

3.2.5 Refactoring: renaming

Description: One of the basic refactorings that can be applied is renaming of variables or methods. All occurrences of the particular variable or method at appropriate scope levels would need to be renamed.

This feature is mainly intended to give Kenya users the idea that the IDE can be used to automate certain tasks that would be tedious for the user to do herself.

There is a strong dependency on the Reference Highlighting feature, which would provide the necessary ability to inspect scope levels and find variable/method occurrences.

Implementation: Making use of the Refactoring processor from above, an addition to the editor's context menu should allow selection and execution of the refactoring.
The Kenya Navigator should be changed such that renaming prompts for a type name without file extension (for Kenya files).

Testing: « The implementation correctly renames variables and methods and their references, while respecting variables' scope. »

3.2.6 Quick fix proposals (Style)

Description: The quick fix system allows users to select from a table a course of action that is to be taken in order to fix errors in the code. In particular this feature is intended to give solutions to code smells that were picked up by the style guidance module.

Implementation: Extension point **org.eclipse.ui.quickFixProcessors**

Testing: « For selected (or all if possible) code smells described in Appendix A it is possible to have the platform suggest a way of solving the problem at hand. If a quick fix solution exists, it should be possible to have the suggestion carried out automatically by selecting the appropriate command. These changes must be meaning preserving transformations. »

3.2.7 Quick fix proposals (Code)

Description: It should not only be possible to automatically fix style problems, but programming errors should also be catered for. The implementation of this feature would provide suggestions to fix compilation errors in the code.

Implementation: Extension point **org.eclipse.ui.quickFixProcessors**

This is an extension feature

Testing: « For as many compilation errors as possible there is the ability to have the platform perform transformations on the code such that the error disappears. »

3.2.8 Code collapse

Description: To reduce clutter on the screen and increase readability, this feature allows the user to collapse method bodies and scope levels, so that she can focus on a particular section in the code.

This is an extension feature

Implementation: An example editor is available, which explores this feature.

Testing: « The implementation correctly recognises and marks collapsible regions. These can be collapsed by clicking on a mark on the vertical ruler of the editor. »

3.2.9 Help System Integration

Description: The current manual is available on the Kenya website. Integration with Eclipse's help system provides consistency and of course the ability to reference the manual offline from within the same platform.

Implementation: Extension points in the **org.eclipse.help** plugin

Testing: « Accessible through the help menu. All pages are complete and display properly. »

4 PREPARATION AND DESIGN

This section details some of the design of *KenyaEclipse* before going into more technical details that are provided in the Implementation section.

4.1 Choice of Eclipse

Why was Eclipse chosen for this project and not, for instance, NetBeans, JBuilder or IntelliJ? There is in fact no technical answer to this question.

One first criterion must naturally be the ability to add functionality to a platform. JBuilder for example is a commercial product and not easily accessible for a plug-in developer. Other than that, most current IDEs are pretty even in terms of features and user interface, although there may not be such a high level of extensibility. [TIDE]

So why choose Eclipse? This is probably because of the author's own familiarity with the platform and lack of experience with any other, and possibly the overwhelming level of interest in it by multiple parties. Another factor may be the knowledge that development tools for plug-ins are readily available and used by the platform developers themselves.

In fact, any other IDE could have been chosen based on its features. A question might be whether (for example) the name 'Kenya(Net)Beans' or 'IntelliK(enya)' sounds better than 'KenyaEclipse'.

4.2 Preparation

As explained in the background about Eclipse, a plug-in consists of a plug-in descriptor file, a manifest, the compiled code and various other resources.

The Eclipse version used for building the plug-in is Eclipse 3.0.2, and the aim is to build a plug-in that is **at least** compatible with this current version and hopefully as well with the upcoming release of Eclipse 3.1.

Using the Plugin Development Environment (PDE) in Eclipse, both the descriptor (plugin.xml) and the manifest are created automatically. The graphical interface allows both files to be edited through a tree representation to which nodes representing various extensions can be added or removed and their attributes set through simple text fields. The manifest contents are similarly edited through the interface provided.

For coding, the standard JDT plug-in is used. The current Kenya version's source code is obtained from the previous developers via CVS (Concurrent Version System).

An own CVS repository is set up in order to be able to transfer code between different locations. It also provides a convenient backup mechanism in case of technical failure.

4.3 Kenya v4 design modifications

The current design of the Kenya editor was one of the issues that had to be addressed. All functionality was crucially centred on the `EditingWindow` class, which included a lot of explicit and implicit dependencies on this class. Some of the core functionality

such as the translation from Kenya to Java code was partially wired into this class and concerns not separated appropriately.

This result emerged from an analysis of the code that was started in order to determine the inside workings of the code and to identify specific components that would be useful in future and that were crucial milestones in the development of *KenyaEclipse*, such as the translation and the run/debug support.

Part of the analysis was done by using the EclipseMetrics plug-in (see section 2.4.6.1), which allows the generation of dependency graphs between packages and individual classes. This provided a starting point as to where to start the process of decoupling the individual components in the original Kenya program.

As a result, original Kenya has been split into two independent components, which used to be one: The KenyaCore, which comprises of the Kenya compiler and the command line tool for batch processing Kenya source files, and the KenyaGUI, which is the GUI version of the tool, including the graphical editor with debugger.

Figure 4.3.1 shows the high coupling between the `EditingWindow` class on the left and the core Kenya functionality on the right (around the `Kenya` class). The result is shown in Figure 4.3.2, which shows decreased coupling around the `EditingWindow` class after reorganisation of the code and restructuring into a separate component.

The main revision to the KenyaCore, which is not shown, consists of the creation of the 'MediatorService', which centralises the parsing and compilation of code to avoid tangling and to fully encapsulate the compiler from the remainder of the program.

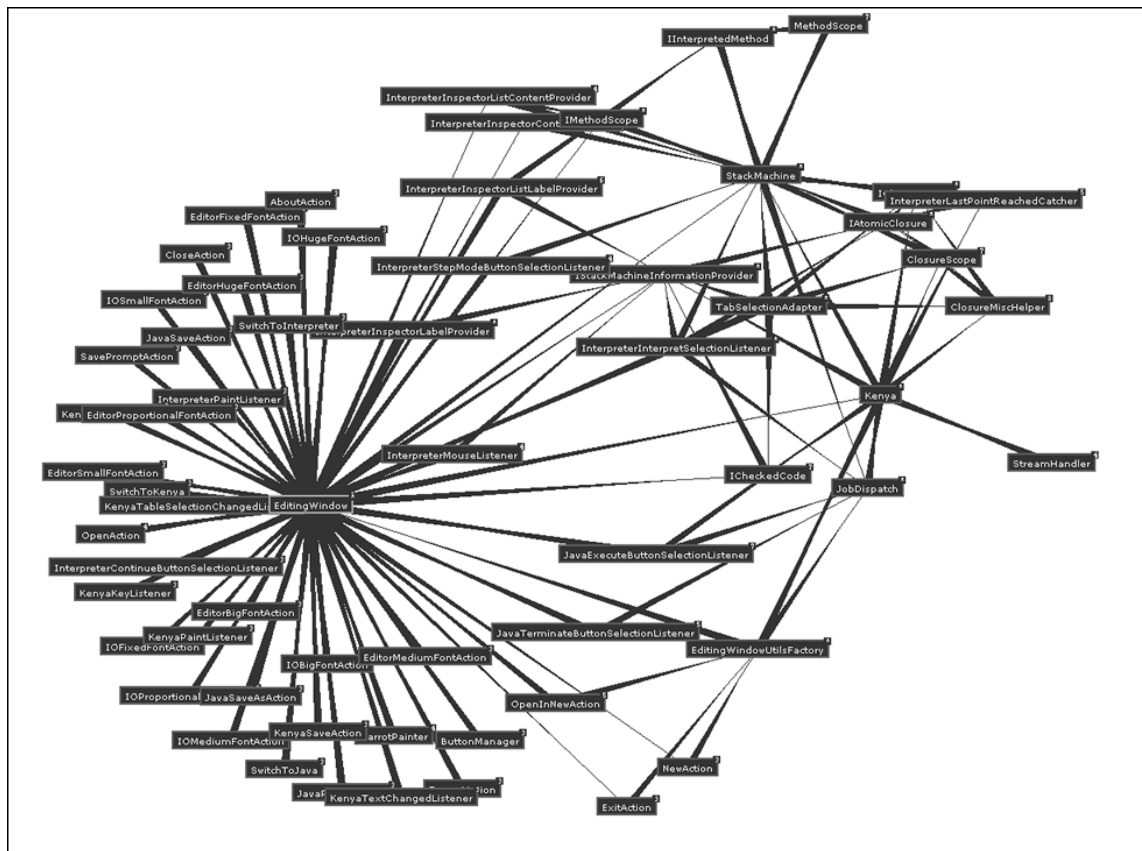

Figure 4.3.1 - High Coupling in Kenya v4 code before Refactoring

Both figures show high coupling by shorter lines and darker colour.

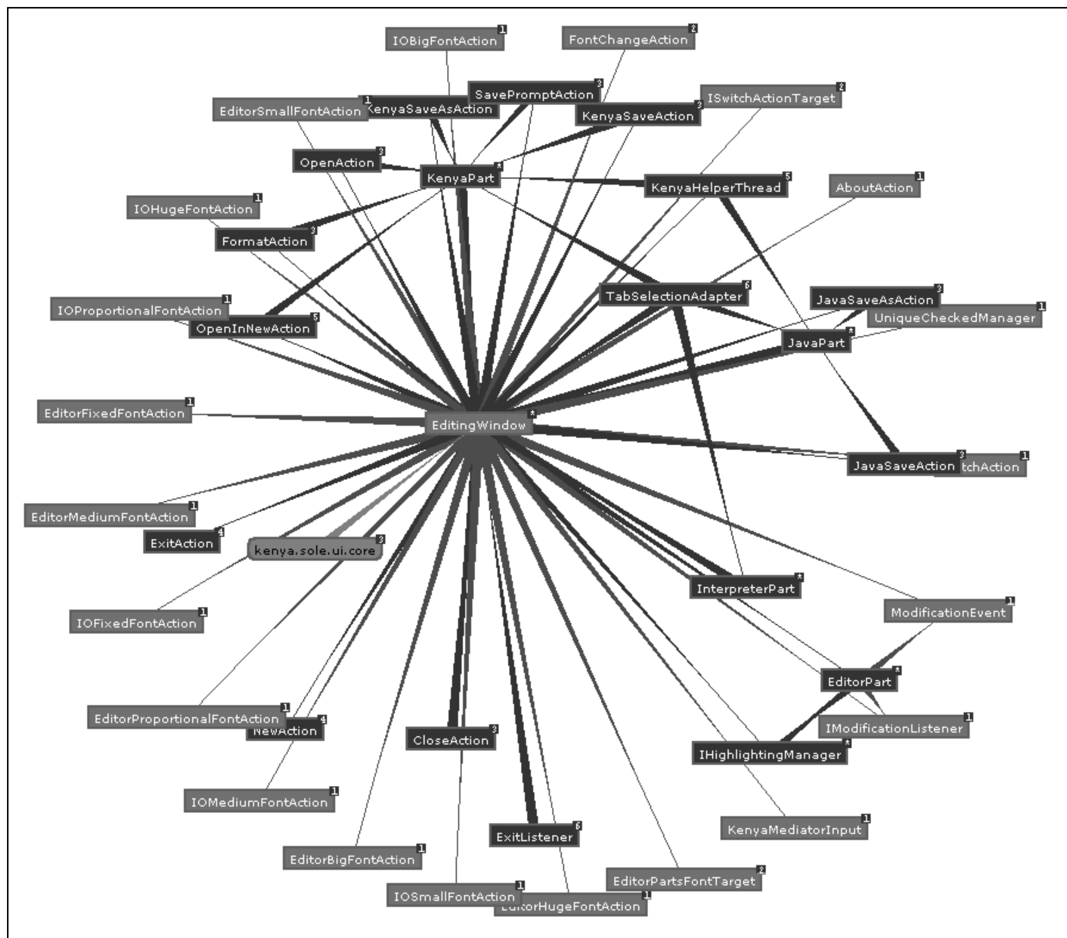

Figure 4.3.2 - Decreased Coupling in the Kenya v4 GUI after refactoring

4.4 Overall Plug-in package structure

KenyaEclipse is composed of multiple components that are to a great extent independent of each other. This independence is not achieved by the code itself, but by the Eclipse architecture which abstracts away from internal implementation issues and provides a generic, consistent framework.

There are by enlarge 5 major component or component groups involved in this project, which have already been mentioned to some extent in the specification:

- *KenyaCore*
- *KenyaEclipse* core components
- *KenyaEclipse* Kenya/Java editor component
- *KenyaEclipse* Run/Debugging component
- *KenyaEclipse* StyleChecker component

One idea is to separate each of these into individual Eclipse plug-ins and deploying this group of plug-ins as a so-called *Feature*. This approach would present a very high level of maintainability and a well defined hierarchy of dependence between the parts.

In practice, there were problems with Eclipse not being able to find the other plug-ins on the classpath when trying to achieve a physical separation of components. It was therefore decided that there would only be a logical separation by package naming, while keeping all components within a single plugin.

5 IMPLEMENTATION

This chapter describes the overall implementation and architecture of KenyaEclipse. The focus is mainly on technical aspects and the integration with Eclipse. For a functional overview, please consider reading the user manual provided in Appendix D. Mark-up examples in the plugin.xml plug-in descriptor file have already been given in the Background section and will not be given here.

5.1 KenyaEclipse Architecture

The following figure represents a high level overview of how KenyaEclipse emerges from existing extension points and the features it provides:

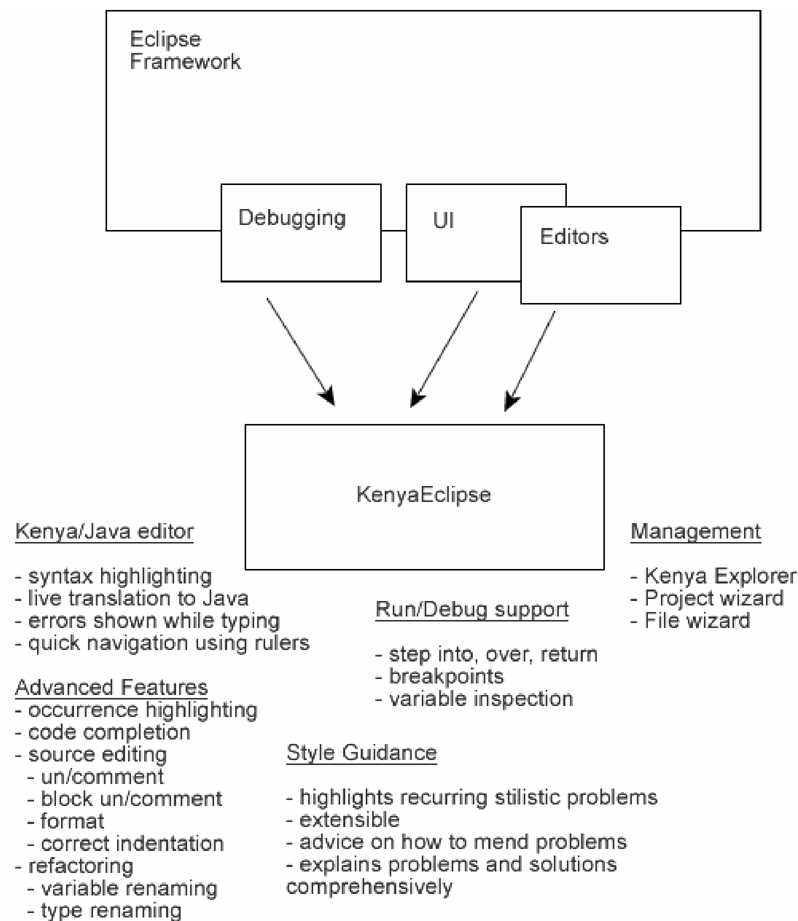

Figure 5.1.1 - KenyaEclipse features based on existing components

5.1.1 Editor architecture

The Kenya/Java editor (Multieditor) is implemented as an extension to the editors extension point. It is a combination of two independent editors that are integrated as tabs, one being the fully fledged Kenya editor, the other just a simple text viewing utility for Java code. A screenshot of the Multieditor component is shown in Figure 5.1.2.

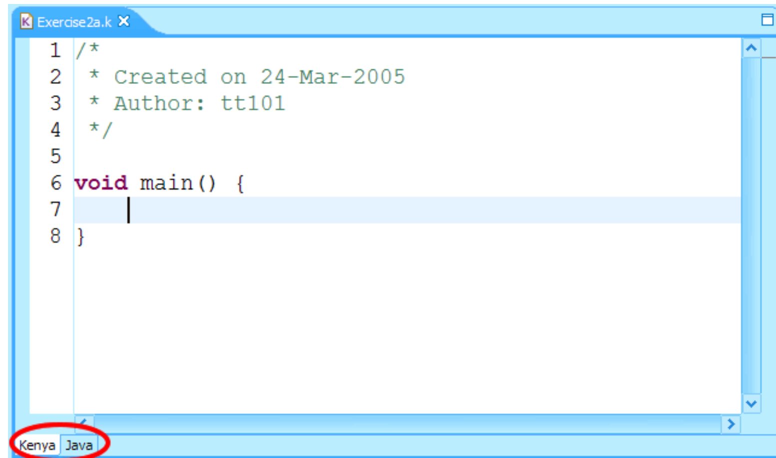

Figure 5.1.2 - Multieditor component

A background thread synchronises between the Kenya code that is typed and the Java view. When the user types, a timeout is triggered and reset while the typing run continues. Once the user pauses for long enough for the timeout to expire, the Kenya source code is passed on to the MediatorService (see previous chapter). This then starts the translation in a separate background thread and passes the result back to the editor, which can in turn update the Java editor part. This interaction is illustrated in Figure 5.1.3 as a UML like sequence diagram.

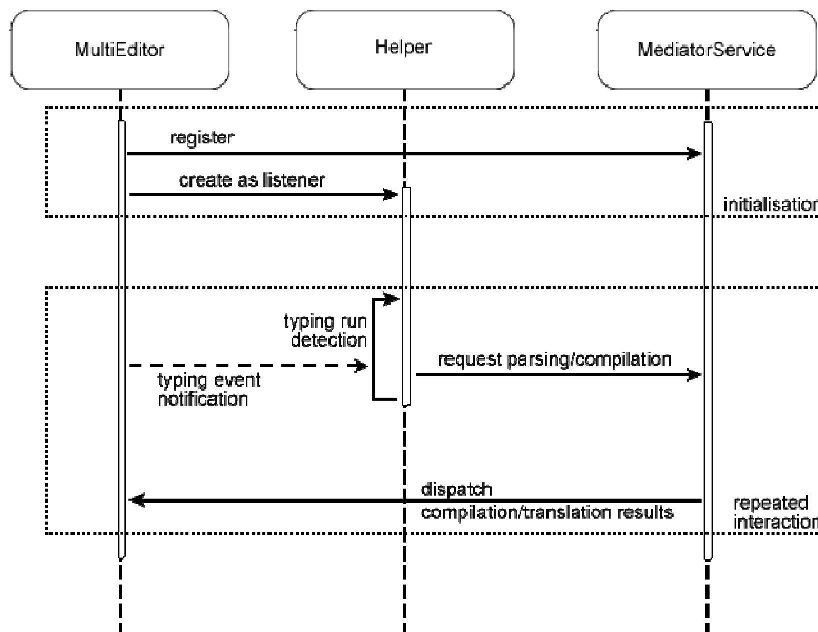

Figure 5.1.3 - Component interaction during compilation

Compiler generated errors and warnings appear in the Problem View (Figure 5.1.4), which is displayed below the editor.

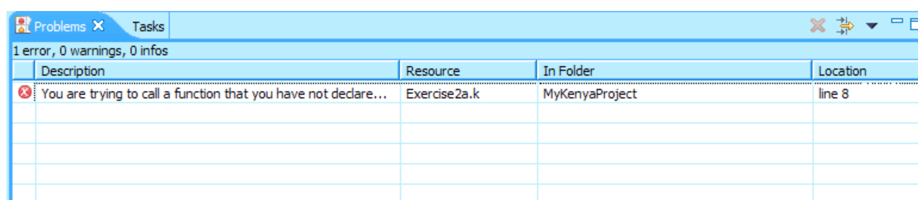

Figure 5.1.4 - Problems View

5.1.1.1 Analysis after compilation

The postBuildAnalyserFactories extension point has been created to allow clients to add a generic analyser that may examine the Kenya or Java code that is produced each time the source code is parsed and compiled.

```
<extension-point
  id="postBuildAnalyserFactories"
  name="Post-Build Analyser Factories"
  schema="schema/postBuildAnalyserFactories.exsd"/>
```

By using this extension point, factories that create code analysers based on configuration data passed from the editor can be added. The overall interaction between the components is shown in Figure 5.1.5.

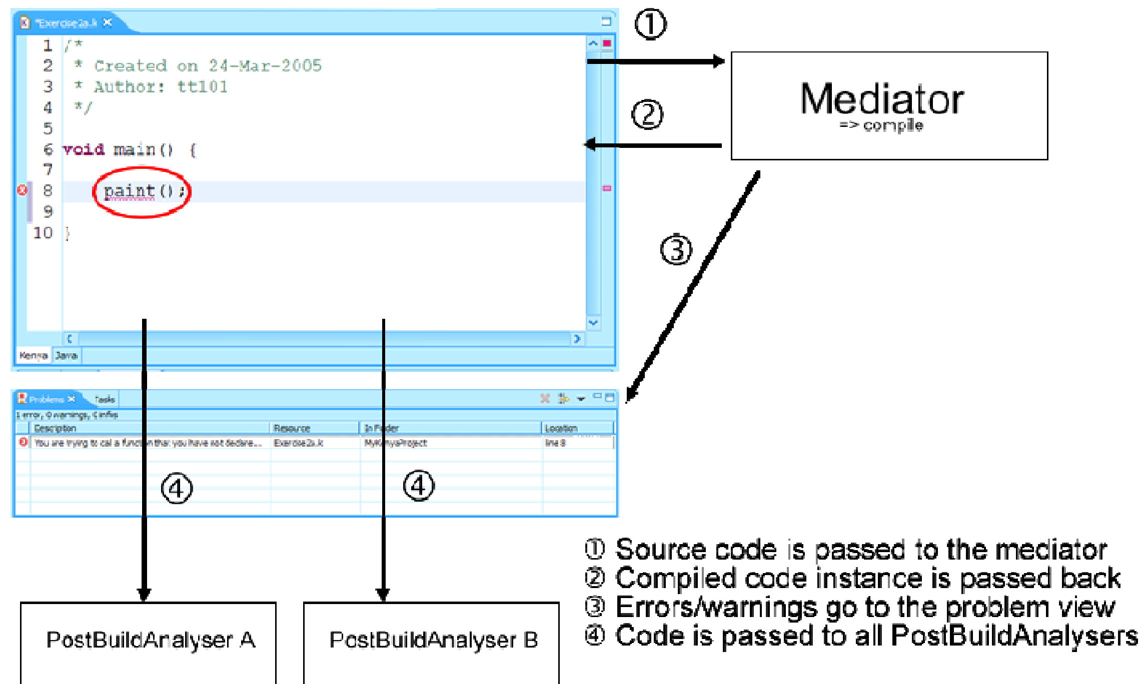

Figure 5.1.5 - Overall Component interaction during/after compilation

This extension point is used by KenyaEclipse itself to hook the style checking framework into the mechanism (see section 5.3).

The implementation of the extension point takes care of dynamically loading analysers and passing them the new code each time it is compiled. There is no direct dependency on any extension to exist. In fact, if no extension is present, then the editor continues normal operation.

The code fragment used to invoke style checking is given in Figure 5.1.6. Most important is the call to `KenyaPlugin.getDefault().getPBAnalyserFactories()`. The main plug-in class takes care of the actual management, code for which is shown in Figure 5.1.7. The extension point is initialised only once, but new instances are created for each invocation of the method.

```
[ ICheckedCode cc; IJavaCode jc; ] (passed as parameters)

//buildext: get all post build analysers..
Collection pbas = KenyaPlugin.getDefault().getPBAnalyserFactories();

//create and call each registered analyser
for(Iterator iter = pbas.iterator(); iter.hasNext();) {
  IKenyaPostBuildAnalyserFactory element
    = (IKenyaPostBuildAnalyserFactory)iter.next();
  //create configured with current file
  AbstractKenyaPostBuildAnalyser rec
    = element.createAnalyser(( (IFileEditorInput)getEditorInput() ).getFile());

  if(rec!=null) {
    if(cc!=null) {
      rec.setCheckedCode(cc);
    }
    if(jc!=null) {
      rec.setJavaCode(jc);
    }
  }
}
```

Figure 5.1.6 - Code to invoke style checking

```
/**
 * Return a collection of all registered IKenyaPostBuildAnalysers
 */
public Collection getPBAnalyserFactories() {
  if (fPostBuildAnalyserMap == null) {
    initializePostBuildanalyserMap(); //once only
  }
  Collection c = fPostBuildAnalyserMap.values();

  ArrayList l = new ArrayList(c.size());
  for(Iterator iter = c.iterator(); iter.hasNext();) {
    IConfigurationElement element = (IConfigurationElement)iter.next();
    try {
      Object ext = element.createExecutableExtension("class");
      l.add(ext);
    } catch(CoreException e) {
      //ignore, try next
    }
  }
  return l;
}

/**
 * initializes the HashMap containing IKenyaPostBuildAnalysers
 * from the extension registry
 */
protected void initializePostBuildAnalyserMap() {
  fPostBuildAnalyserMap = new HashMap(10);
  //load the extensions for the extension point
  IExtensionPoint extensionPoint
    = Platform.getExtensionRegistry().getExtensionPoint(
       getPluginId(), KenyaConstants.EXTENSION_POINT_PB_ANALYSERS);
  IConfigurationElement[] infos
    = extensionPoint.getConfigurationElements();
  for (int i = 0; i < infos.length; i++) {
    String id = infos[i].getAttribute("id");
    fPostBuildAnalyserMap.put(id, infos[i]);
  }
}
```

Figure 5.1.7 - Extension point management code

5.1.2 Run/Debug support

Running and debugging is enabled by using the LaunchDelegates extension point defined by the eclipse debug plug-in. Through launch shortcuts and configuration dialogs (see Figure 5.1.8 and Figure 5.1.9), the delegate is invoked and performs the necessary steps to initialise a corresponding run or debug process.

Both running and debugging in KenyaEclipse is modelled by 'virtual' processes derived from `java.lang.process`. While the process of the 'run' type wraps around the execution of an actual external process (command line `java`), the 'debug' type is a wrapper for the execution of a Kenya StackMachine instance.

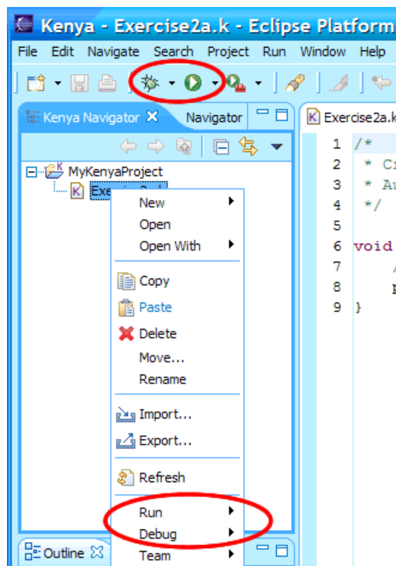

Figure 5.1.8 - Launch shortcuts

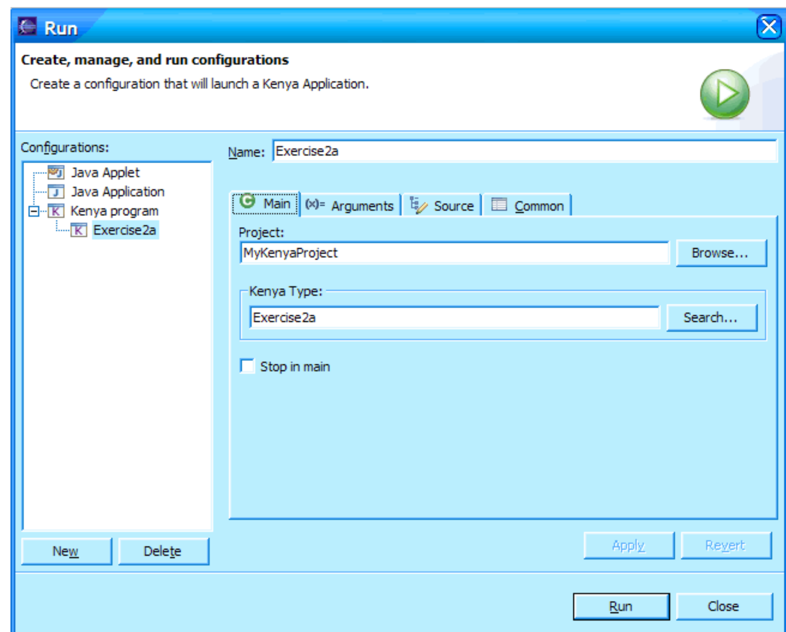

Figure 5.1.9 - Launch configuration dialog

Run

The implementation of 'run' is quite straight forward and code from the previous Kenya version was reused to invoke the `java` executable on the translated Java code. In order to make the usage of KenyaNIO (for reading from stdin in Kenya) possible, the corresponding jar archive is located within the plug-in codebase and its path appended to the system property `java.class.path`, where it is expected by the Kenya v4 code.

Debug

The 'debug' implementation is much more complicated because of the dissimilarity between the StackMachine implementation and the generic debugging framework architecture defined in `org.eclipse.debug`.

The StackMachine (SM) was introduced with the debugging component in Kenya v4. Its execution is built as a set of closures during compilation. It was not designed to handle external inspection (other than by Kenya v4) and is thus, for this purpose, a very 'black-box' design.

On the other hand, the nature of debugging in Eclipse allows white-box testing of the code. The SM implementation makes it difficult to extract the required information about variables, execution point and the *artifacts* that are required by the framework. (A relevant extract about Artifacts is provided below.)

At first this caused a large disruption in the schedule, because originally only a week had been allocated to integrate the debugger. Subsequently the plans were changed and some features moved to a low priority. Although supposed to have been completed by the end of December, integration and bugfixing actually continued into January.

The implementation was mainly made possible by firstly using a wrapper immediately around the SM and by relaxing some of the SM access methods and secondly by using a bridge between the DebugTarget (main access class to the debug model) and the wrapper. When data from the SM is required by artifacts, this is relayed over the bridge and the result returned from the wrapper. This bridge pattern is mainly used to avoid downcasting from the generic launch object returned from the debugging framework when the executing process is created, but also to keep the interface between SM and DebugTarget as small as possible.

Although this solved the problem of obtaining access to the SM contents in the first place, the debug component in KenyaEclipse is not entirely bug-free. Details of remaining debugger related issues are covered in section 6.4.1.

Artifacts [E-PDG]

The model includes classes that represent different artifacts in a program under debug. All of the artifacts implement *IDebugElement* in addition to their own interfaces. The model includes definitions for the following artifacts (irrelevant entries omitted):

- Debug targets - debuggable execution context, such as a process or virtual machine
- Stack frames - execution context in a suspended thread containing local variables and arguments
- Threads - sequential flow of execution in a debug target containing stack frames
- Values - the value of a variable
- Variables - a visible data structure in a stack frame or value

The overall core debug model architecture is shown in Figure 5.1.10.

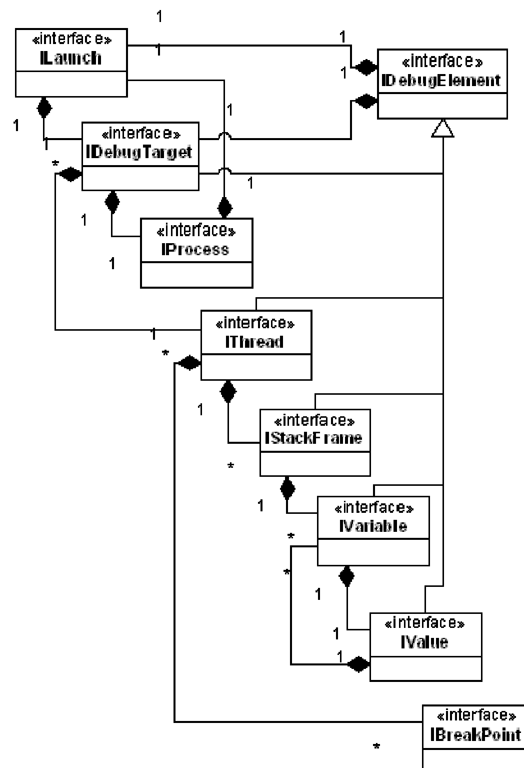

Figure 5.1.10 - Core Eclipse debug model architecture

Breakpoints

Breakpoints are implemented using the SM concept of a `PointListener`. This listener is registered with the SM on start-up and receives event notification whenever the SM reaches the next step in its execution. By keeping track of set breakpoints' locations, the `DebugTarget` can decide whether or not to suspend the SM based on the location information it has been given through the listener. The `PointListener` is also used to suspend after a step command from the user by discovering the step end.

5.2 Advanced editor features

Most of the features in this section are implemented in the JDT in some way or other. The aim is to emulate these features as closely as possible for consistency, even to the extent of copying existing code and adapting its functionality to KenyaEclipse's purposes. This is the best way of introducing Kenya users to these features.

5.2.1 Occurrence Highlighting

In the JDT version of this feature the very first step is to obtain a so-called variable binding, which is created by the compiler and contains information (amongst other things) about the type and name of any named entity, such as methods, classes and variables. Each named entity and each of its occurrences share a unique binding. This means that in JDT highlighting is 100% accurate.

Unfortunately, in KenyaEclipse, binding information is not maintained by the compiler and since there was no intent to change the core classes, bindings in KenyaEclipse have to be resolved dynamically.

Although this means potential for great disaster, the algorithm for doing this has matured through trial and error and now caters for (almost) all possibilities. Resolution is invoked every time the selection in the editor changes to an other named entity while occurrence highlighting is enabled.

There are multiple steps in the resolution algorithm:

1. Determine position of selection in the document
2. Find the AST node immediately surrounding that position
3. Determine whether the position is inside a method body or not
4. if inside a method body and the selection is that of a variable name, traverse the method's node in the AST and search for a local variable declaration with that name BEFORE the selection. If not found, continue by searching constants
5. if the selection is that of a class identifier, search all class declarations to find the correct one
6. if the selection is that of a method name, search all method declarations to find the correct one.
7. The declaration found represents the binding of the selected name

For the highlighting, the entire AST is searched again, and for each variable, class and method name, its binding is determined as in steps 1-7 above. If the binding matches, this is an occurrence of the selection, and it is added to a list together with its position in the document.

Annotations are then created which dynamically highlight these locations. (The yellow highlighting background was taken over from JDT for consistency.)

Annotations are a feature of the Eclipse platform and JFace, while the code to create them was freely available by example from JDT. The main alteration is the resolution algorithm which is absolutely necessary for highlighting to work at all and its correctness determines the correctness of the highlighting.

We can argue why there will be no false positives or wrong highlighting when using the resolution algorithm as above, provided that the code it examines has compiled successfully (if not, a standard solution is to not highlight anything).

There are two possibilities in which the 'wrong thing' might be highlighted, those are

a) shadowed constants
b) overloaded methods

a) In step 4, the algorithm first searches the method parameters and body for the corresponding variable declaration. This means that scope is always preserved. If the selection occurs before the declaration of a variable with the same name, then by the definition of step 4, that local declaration will be ignored.
b) The algorithm can distinguish between methods that have a different number of arguments. If two or more methods have the same number of arguments, they are distinct in their arguments' types, but unfortunately this is not easily done in Kenya without efforts to dynamically type the program (since access to that data is not available). If this particular case is detected, the highlighter will not highlight anything. This has been deemed appropriate so as to not confuse the user (see section 6.3.3).

5.2.2 Code Completion

The code completion feature is based on regular expression matching and limited analysis of the program AST.
The AST is used to determine what variables, methods and constants are available to be used at the current location in the program.
Regular expressions are used to determine what the user is currently typing, an assignment or a method call for example. This is important as already typed letters count as a prefix to the completion and non-matches are eliminated as the user types. The amount of typed text is essentially determined through the use of regular expressions and patterns.
The associated feature of parameter assistance is very similar and uses a similar technique. By determining all methods that share the same name, the user can be offered a choice of these methods and the associated required parameters for the call can be displayed as an overlay.

Code examples for both features are available from JDT, but because of the difference in compilation and AST, a complete rewrite was necessary to adapt to Kenya. This proved to be fairly difficult because of the inability to use a complete AST and partially because of the previously mentioned problems with resolving variable bindings and types (see also section 5.2.1).

A more sophisticated and possibly more reliable implementation might seek to use the AST only, however, using regular expressions was considered a simpler way of performing the same task.

5.2.3 Source Editing Features

The features that were implemented (the ones listed except the 'format' action), are all purely text based transformations that do not require the availability of an AST or similar. This means that the code could be taken over from the present JDT implementation of the same features with only little adaptation.

5.2.4 Refactoring - Renaming

The only refactoring relevant enough to Kenya is that of entity renaming. Entities include variables, methods, classes and constants (plus fields in Java). The rename action allows the user to rename such an entity after selecting it or placing the cursor in it in the Kenya editor.
She will then be presented with an input dialog prompting for a new name. The dialog validates the input and disallows, for example, reserved keywords and names already in use, as well as warning of discouraged names, such as class names starting on lower-case letters. This is similar to how the new file wizard operates and in fact uses the same validation code. The dialog can not be submitted unless the input is error-free.

The refactoring operation itself is based on the dynamic binding resolution also used by occurrence highlighting (see section 5.2.1). It is therefore affected by the same problems associated with that algorithm. The solution to these is the disablement of the action in the first place. The action is only enabled by the editor if the current selection can be resolved unambiguously, so that all occurrences can be renamed reliably (see definition of refactoring in section 2.3).

5.3 Style Guidance Module (SGM)

The implementation was aimed to provide the following attributes:

- Extensibility: easy addition or removal of style checks
- Configurability: ability to enable/disable/configure individual style checks
- Independence: KenyaEclipse must work without the style module
- Efficiency: calculations must be done in the background, not block the GUI
- Assistance: Allow users to apply automatic correction if possible or applicable

While the next few sections address each of these points in turn, section 5.3.6 contains a deeper overview of the module's implementation and will further clarify these points.

5.3.1 Independence

Independence is provided through the use of the postBuildAnalyserFactories extension point (see section 5.1.1) to link the style checking framework into the editor's compilation (and translation) mechanism.

5.3.2 Extensibility

Another extension point allows the addition of Style Checkers into the framework:

```
<extension-point
  id="styleCheckers"
  name="Kenya Style Checkers"
  schema="schema/styleCheckers.exsd"/>
```

Figure 5.3.1 - Extension point declarations in KenyaEclipse

The second extension point is managed by the style checking framework rather than the editor. It allows clients to add a style checker to the framework, which will then dynamically be instantiated, configured and invoked by the style checking framework.

A visualisation of this extension hierarchy is shown in Figure 5.3.2 below.

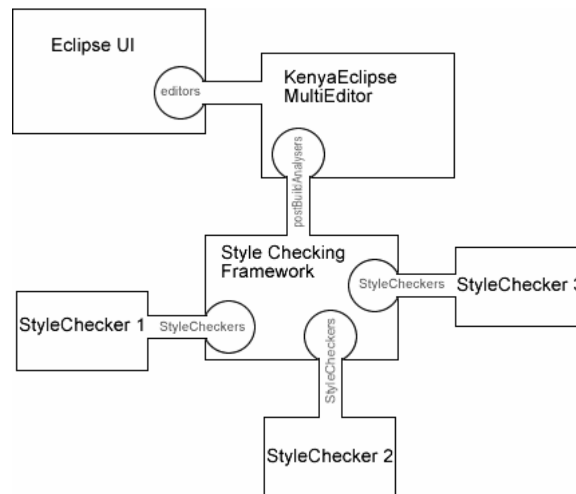

Figure 5.3.2 - Extension hierarchy; example with 3 StyleCheckers

It is very easy to add a new StyleChecker using this extension point. The following details are required:

- **class** - name of a class implementing the IStyleChecker interface
- **ID** - unique ID for the StyleChecker to be recognised by the platform
- **name** - short description for display on the screen
- **defEnabled** - whether or not the check should be enabled by default on all files
- list of custom attributes for reusable StyleCheckers

The Developer's Guide provided in Appendix C gives a detailed description of how to define and implement a StyleChecker.

5.3.3 Configurability

There are essentially two kinds of configurability.

Configurability for the *developer* is achieved by allowing custom attributes in the declaration of StyleCheckers that change behaviour of potentially (through these attributes) reusable checks.

Configurability for the *user* is achieved by associating a style property page with each file. The page is available to all files associated with Kenya. It allows the user to enable or disable individual StyleCheckers as well as the entire module on a file-by-file basis.

Figure 5.3.3 shows the implemented property page. The view shown is based on the default configuration of the module with the enabled state of the checkers as shown. The labels are taken from the *'name'* attribute as described previously.

Particular attention is drawn to the *'check for method length > X'* as a good example of the declaration requirements (see above). This is a reusable StyleChecker that has been declared with a different unique *ID* thrice. Two declarations have the *'defEnabled'* flag set to `false` and thus by default they are not checked on the page. Use of *custom attributes* is also made by providing each declaration with a value for the maximum allowable length of methods, X, that it should react to.

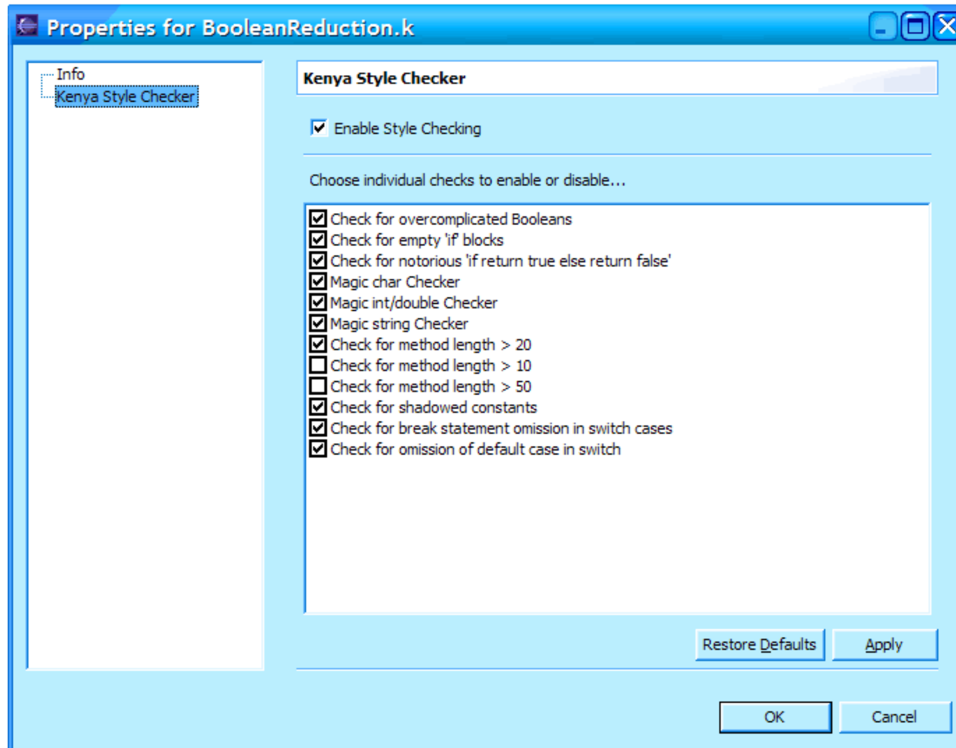

Figure 5.3.3 - Style guidance module property page

5.3.4 Efficiency

While ultimately each StyleChecker carries and is responsible for its own performance penalties, the basic framework takes care that each run (of the module) is performed in a separate thread. This allows checking to be carried out asynchronously and the user interface being unaffected by slow running checks.

An additional measure are the predefined methods defined in the abstract base class of StyleCheckers (`AbstractStyleChecker`) that take care of creating the annotations in the relevant document, which is also done asynchronously through the platforms Job architecture. Although subclasses can circumvent this mechanism, it would be inconvenient, if not more difficult to achieve the same result.

Effectively, StyleCheckers are executed over 2 threads that are separate from the platforms main thread and the UI thread which does not necessarily yield better performance, but does help to keep the system responsive.

5.3.5 Assistance

In its implementation, a StyleChecker may update a provided map that is subsequently returned to the module upon completion of the run. The map contains the details required to provide an interactive solution proposal to the user. If the map is not completed, then there will simply be no solution proposal available for the problems detected by that particular checker.

5.3.6 SGM implementation

During any typing run, certain steps are first performed before actual checking begins:

1. Multieditor calls `StyleCheckingFactory.createAnalyser` with the file to be checked as argument. This is already done from a separate thread so that there is no influence on the editor's other activities.
2. The factory retrieves all enabled StyleCheckers (if any) for the file from the preferences, instantiating and configuring them as required, and returns a `PostBuildAnalyser` configured to run those checks. The mechanism for this is almost identical to the way in which the core plug-in class handles postBuildAnalysers, as seen in section 5.1.1.1.
   If the module is disabled for the file, it will return an empty analyser instead.
3. The editor calls the analysing methods first with the Kenya, then with the Java code object as argument.
4. Since the StyleAnalyser is only concerned with Kenya code, the call with Java code is simply ignored.

Actual checking is then performed as a system job running through the Eclipse scheduler. A job can be cancelled by the user at any time using the Progress view, which is updated through a monitor object maintained by the job itself.

The contents of this job can be summarised as in the simplified code excerpt below (some monitor calls have been left out):

```
for(int i = 0; i < checkers.length; i++) {                              //1
  if(monitor.isCanceled() || !fFile.isAccessible()) {                   //2
    //Job is cancelled
    return Status.CANCEL_STATUS;
  }
  IStyleChecker c = checkers[i];
  monitor.subTask("Executing - "                                        //3
     + c.getDescriptor().getName());
  try {
    c.performCheck(code, fFile);                                        //4
    KenyaStyleManager.getResolutionManager()                            //5
       .addResolutions(fFile, c.getMarkerResolutionMap());
  } catch(Exception e) {
    //dont let one of them ruin everything, but log the error
    KenyaPlugin.log("Error while executing StyleChecker \""             //6
         + c.getDescriptor().getName() + "\": " + e, e);
  } finally {
    monitor.worked(1);                                                  //7
  }
}
```

Figure 5.3.4 - Simplified implementation of the style analysis job

Comments:

1. `checkers` contains instances of the `StyleChecker`s that are to be executed
2. `monitor` is a `ProgressMonitor` that displays progress and allows user to cancel
3. Subtasks of the job may be displayed with their own name
4. Call the main checking function
5. Any resolution (assistance) proposals are obtained from the checker and cached by the `StyleResolutionManager`
6. Any exceptions that occur during the execution of a `StyleChecker` are logged by the plugin. This implementation prevents broken `StyleChecker`s from affecting others and prevents platform runtime exceptions from being generated and thrown back at the user. The central log is, however, easily accessible.
7. The progress displayed by the monitor is advanced by one unit of work. The total work was specified in advance, in this case the number of `StyleChecker`s to be executed.

5.3.6.1 StyleChecker implementations

StyleCheckers can generally be implemented as three phases:

- Broad search
- Narrow search
- Detail definition & resolution generation

(Broad search)
A style pattern generally applies to certain constructs. The `BooleanReduction` checker for instance applies specifically to boolean expressions. By using a generic visitor over the AST of the Kenya program, all nodes representing boolean expressions can be picked out.

(Narrow search)
The results are then subjected to finer criteria, in the `BooleanReduction` case this is the ability to reduce the expression. This can be done by applying another, finer grained, visitor to each node and keeping only the ones that explicitly 'violate'.

(Detail definition & resolution generation)
The remainder is a list of all 'offenders'. Based on their position, an automatic resolution can be generated (if applicable) and the internal map of resolutions updated. The position is used to create a marker, which is then interpreted by the editor component to create an annotation.

Examples of implementation are given in the Developer's Guide (Appendix C).

6 EVALUATION

6.1 Quality Assurance

The tool that is being developed will be deployed in a 'mission critical' environment: the teaching laboratories that are used by computing students in order to learn programming. It is thus vital to provide them with stable, well performing tools that are resistant to any form of 'mishandling' by inexperienced students. At the same time, the tool should provide the clearest possible messages when errors do occur.

Therefore, while testing components, extra care has to be taken to make sure they perform in the target environment and can handle all kinds of erroneous user input.

At a lower level, all Exceptions that could be thrown must be caught, logged and, in severe cases, may be reported to the user through a clear message on the screen. This message could contain advice on how to deal with the problem that just occurred.

6.2 Testing

JDT comes with full debug support, while PDE allows the launch of a second instance of Eclipse. This combination allows the plug-in to be tested and debugged from within the same environment. The debugger also allows the so-called 'hot replacement' of the code if it was changed in a compatible fashion. This means that if for example the body of a method has changed, this change is reflected the next time this method is run. This feature saves valuable time when testing the application because a restart of the debugger is not always necessary (see also section 2.4.3.2).

There are three parts of testing to ensure the correct functionality of the plug-in:

- **Systematic single component testing**
  This is done each time a component is added to the plug-in in order to make sure that it behaves as intended.

- **Integration testing**
  During and after creating each component it has to be tested to make sure that its addition to the system does not cause conflicts with other parts of the application.

- **Simulated user testing**
  this comes after implementation of various features that allow the basic usage of the tool. The idea is to play a real user and to (almost) randomly perform actions and invoke functionality in order to test the robustness of the plug-in.

For semi-detailed feature/component tests, please review implementation requirements given in chapter 3.

409 test programs (kindly provided by members of staff) were available for random tests of all features. Addition test files were written to examine StyleChecker functionality.

A very small number of first and fourth year students agreed to test and comment on the usability and functionality of KenyaEclipse, although this did not yield significant results (not enough data).

6.3 Results

Individual feature tests do in this case probably not reflect on the success of the project very well. Respecting the quality assurance statement above, features were in the end left out if they could not be made to work to a high degree of satisfaction, rather than leaving non-working components or half-finished items.

The compiler and associated parts had been tested previously and were not changed during the course of this project. The core Kenya components are imported as an independent library and thus were not checked explicitly for bugs.

The basic components (editor, compiler, run, debug) were tested to show the same behaviour as the previous version of Kenya did for these features, however still some problems occurred.

The following sections will go through the most important features, comment on the implementation and argue how they have affected the effectiveness of the tool (positively or negatively). For functional results, you may read the user manual, which is contained in Appendix D, which also includes relevant screenshots. This part is only aimed to discuss the advantages and disadvantages of the functionality.

6.3.1 New project and file wizards

The project wizard is very easy to use, because it only asks for a project name. The disadvantage is that it is not possible to create a project outside of the workspace, although it can be moved manually later.

The file wizard provides the ability to create a main method, which has to be present in every Kenya file. This is a step forward as it removes the need to retype a main method each time a file is created. It also introduces the use of wizards to create classes, so that the user becomes familiar with the concept and is not suddenly overwhelmed with the amount of automatically generated code in JDT later on.

Overall the use of wizards is a step in the right direction, but the project wizard may have been over-simplified. Students confirmed that it had been tedious to write a main method over and over again.

6.3.2 Compiler

Compilation occurs when the user stops typing for a short period of time. All compilation errors can dynamically appear in the Problem view as was the case in the previous version of Kenya.

A nice envisaged feature was the ability to create solutions to compilation problems as is the case in JDT. This would have required significant changes to the compiler and was in the end left out of the project. If the compiler marked problems with a unique ID, then these could later be used to provide a solution to each error.

One problem was the choice of the time that the user needs to stop typing for the compiler to be triggered. The number used in the end was 1000msec. It is usually the case that during a typing run, the user may pause briefly, and parsing should not occur at that moment. On the other it should not take too long after the user actually finishes, that he can see the result on screen, so a trade-off definitely exists.

A future solution might be to incorporate the timing into a preference page.

6.3.3 Run support

The users can run their applications the same way in which Java programs are run in JDT. The introduction is good, however the way in which arguments are passed to the applications is awkward. The user first has to open the run configuration dialog and then navigate to the appropriate page to change arguments. Eclipse is here more bent on preconfigured input that does not change over multiple launches. GILD (section 2.4.7) solves this problem by introducing their own toolbar and individual launch icons and dialogs that support entering arguments prior to launch without using the launch configuration dialog as in KenyaEclipse.

For Kenya users it would be better to have a faster way of changing runtime arguments as is the case in Kenya v4, where this is all done on the Java page in the editor.

The console on the other hand is very similar and provides stdin, stdout and stderr stream 'output' in different colouring. It was even possible to implement the 'insert EOF' button as an extension to the console view, without which testing input based programs from within Eclipse would not be possible.

6.3.4 Debug support

Suffers from the same 'argument problem' as running.
The way in which breakpoints are created has become more intuitive. It was not obvious in Kenya v4 that double clicking in the code area will add a 'carrot' to indicate a breakpoint and its exact position of appearance was not clear either. Some students did not even know that this feature existed.

Unfortunately, again, the underlying architecture presented some problems. The Kenya StackMachine used for debugging has a fundamentally different architecture to the debugging framework in Eclipse. It was very difficult to minimise the number of problems and there are various issues that are still unresolved (see section 6.4).

6.3.5 Occurrence Highlighting

The main thing to avoid is highlighting the wrong thing, i.e. false positives. KenyaEclipse has to resolve the binding of an entity dynamically without access to class tables. This makes it hard to guarantee correctness, however, the algorithm used (see section 5.2.1) provides correct resolution, provided three conditions hold:

- code does not contain compilation errors
- selection is ***not*** a Java 1.5 features (such as an enumeration)
- selection is ***not*** a method that is overloaded with the same number of arguments

By improving the compiler and the available information, the algorithm could be simplified and its reliability, as well as performance, much improved (as well as catering for the above situations). The AST also plays a role through the way in which names are used.

In the above cases, occurrence highlighting was forced to be switched off (Java 1.5) or does not highlight anything (overloaded methods, compilation errors). This is a shortcoming that could (will hopefully) be addressed in the future.

6.3.6 Code Completion

The code completion feature was very tricky to implement as it has to work with a program that cannot be compiled (if it works, you do not need to complete anything).

The AST used for the completion (to find available method calls, etc.) is based on the last successful compilation, which means that completion does not work if a file is opened that contains compilation errors (as no such compilation is then available).
This severely impacts the usefulness of the completion tool as it is supposed to be one of the big helpers with broken code.
However, overall the addition of this feature is a great help and various students in the MEng 4 class agreed that "it would have been much easier with this…", mainly due to the fact that it displays all available pre-defined methods which are not well-documented in the Kenya manual. Students who had not used Kenya, but had experience with IDEs were delighted by this addition.

6.3.7 Source Editing Features

There were no problems with these features as they are purely text based and the code was already available from JDT. One feature, format, had to be left out because of its complexity. The formatting feature is responsible for formatting arbitrary code based on a certain template. In JDT this is done by examining the AST and laying out the corresponding code on the screen. This proved difficult in KenyaEclipse in the face of non-compiling code and an apparent lack of time in the end.

The working features remove part of the tediousness of coding and debugging code by allowing fast alternatives to commenting (uncommenting) entire chunks of code or correcting the indentation of the entire document, that otherwise might be done by hand (or not at all).

6.3.8 Refactorings

Apart from bad style, variable naming is also an issue that is frequently mentioned by programming tutors. The rename refactoring allows students to give entities a more meaningful name to please their tutors, even if they have used a single-letter name before, just for the simplicity of typing it. Together with the code completion feature, there is now little excuse for using short, indescriptive acronyms instead of readable names.

Renaming suffers from the same problems as occurrence highlighting since the same mechanism is used to determine what has to be renamed. In the cases where renaming is not deemed to be safe, the action is disabled. In JDT renaming is possible, although the tool issues a warning about possible errors that may arise. JDT also offers a preview of the changes that refactorings will introduce. This is made possible by a refactoring plug-in that maintains all relevant functionality. The use of this plug-in was not deemed necessary by KenyaEclipse since it would have required the use of unfamiliar API for a relatively small feature, but it could be considered in the future, since that would allow for a more consistent design.

6.3.9 Style guidance module

The most important addition in terms of features has turned out to work quite nicely. It is possible to easily add new checks and configuration can be done on a per-file basis (enabling and disabling individual checks or the entire module). There are no failures (crashes) if individual checks do not work properly and overall the implementation is very stable.

However, there are still more things that can be done. For example, two metrics based checkers have been included, one for the length of methods and another for the number of parameters passed to a method. Each must be configured with the number of lines or

parameters that is considered as 'the maximum allowed before a warning is issued'. Currently this can be done by adding a 'custom attribute' to the style checker's declaration in the plugin.xml file. This is inconvenient as it cannot be easily changed after deployment. It should be possible to configure this kind of information either globally through the main preferences, but also on a file-by-file basis, and this is something that should definitely be addressed in the future.

Further, resolutions (assistance) proposals for each problem are cached by the `StyleResolutionManager`. There is no persistence across executions of the platform. This means that while in JDT it is possible to 'cure' problems directly from the problem view, this only works for Kenya if the file has been opened during the current execution, since resolutions are generated during analysis, but not dynamically based on the problem. This could be addressed by introducing a separation of detection and correction.

6.4 Unresolved Issues

Various problems were noticed, but could not be overcome.

6.4.1 Debugger

When using the step functions to examine the detailed execution, the current execution point is highlighted in the editor. Sometimes, the debugger may lose focus and not update the highlighting accordingly and a further click on the executing thread or stackframe is required to perform that update. To reduce the amount of times this happens, caching of program state was introduced. It allows the debugger to return the same instance of a stackframe, which allows the UI to not lose focus. On the other hand, this has disadvantages when it comes to debugging recursive code.

In all cases where recursive method calls were used, the debugger had problems with updating the list of stackframes in the program if a breakpoint was set inside a recursive method. This is mainly due to the caching above, although this can also be fixed by clicking on the relevant stackframe multiple times. However, there is a strong conflict between the 'loss of focus' issue and the 'recursive method call' issue.

Almost all of the test files contain some kind of loop (for or while). If a breakpoint is set inside the loop, it can happen that the debugger does not suspend during all iterations. This is a synchronisation issue between the StackMachine and the debugging framework and could not be resolved, unless the StackMachine were to be rewritten to conform to the Eclipse debugging architecture.

Sometimes, the debugger can leave the highlighting of current execution point in the editor, which will then look quite ugly (with coloured bits all over the place). This was observed to happen exactly each time that the debugging view 'loses focus'. So the two problems are clearly related and resolving the first will probably also clear this one.

6.4.2 Binding resolution

This is essential for both occurrence highlighting and renaming. The current algorithm does not fully handle Java 1.5, overloaded methods with same number of arguments nor code with compilation errors.
A possible alternative would be to modify the grammar (AST) and compiler in such a way that all required information is available from the compiled code object. The other possibility is to improve the algorithm, which may be quite hard due to its currently already complex nature (in the implementation).

7 CONCLUSIONS

7.1 AST analysis and modification

It is not a trivial task to analyse code, even if an AST is available. Static analysis can be extremely expensive and mind-boggling, as is proven by the code used for the dynamic binding resolution algorithm (described in section 6.3.5).

7.2 Plug-in creation

Eclipse provides many great features including the ability to write and test plug-ins from within its own environment. Using XML mark-up to define extension points and extensions is a clever way of managing metadata and creating associations and allows developers to easily create very extensible solutions by making use of this 'built-in' extension mechanism.

While Eclipse itself is not totally bug-free, its community drives the development of a stable solution. Keeping up to date and frequenting forums and newsgroups is a must.

7.3 Code Design

In a project such as this it is important to keep the codebase maintainable. Significant time was spent to decouple various components, only to reuse one of them. However, the resulting narrow interface to the mediator (compiler) was surely worth the effort. Componentisation is not always easily achieved, especially as the amount of code grows.

7.4 Criticism of the KenyaEclipse approach

- Students should still be taught about the **basics, like command line compiling**, so they are not stranded without the (an) IDE.
- related to the above is **automatically generated code**: if even beginners have 'everything' generated for them (rather than writing themselves), are they able to reproduce it elsewhere? ("public static void main(String[] args)" for instance)
- **The choice of Eclipse** (see section 4.1) as the base platform has many consequences. Independence is diminished and the project becomes bound to a particular platform, although open-source. Students should be offered the choice of any platform that suits them, although they may limit themselves to using Eclipse because the university seems to advocate its use, while other tools may be better suited for particular tasks.

7.5 Benefits of the KenyaEclipse approach

- Students get to know the **amenities** that modern IDEs have to offer.
- Integrating with a professional toolkit ensures **realism**: Students feel they are learning something worthwhile, not something they will forget after the exam.
- Focus is student support, not instructors convenience (as in GILD, see 2.4.7.1)

7.6 Future Work

The previous sections have already touched on various issues arising from the KenyaEclipse implementation. There are few distinct ways in which the work could be improved beyond correcting the issues mentioned in section 6.4 and that may help creating a true teaching platform (maybe not only for Kenya).

7.6.1 Missing features

- **Global Kenya preferences** - there is currently no mechanism for changing any preferences in KenyaEclipse. Although the infrastructure for preferences are in place and being used internally, there is currently no design for letting the user change these. By using the preferences extension point, a configuration page may be added that allows for user configuration, including global preferences for the style guidance module.

- **Correction proposals for compilation errors** - While this was not possible for technical reasons, this would provide the students with help for correcting more severe mistakes than style errors.

- **Previously planned features** - Some other planned extension features could not be included, such as folding code regions and separate saving of Java files. The latter might be done easily by adding to the ‘export’ context menu.

- **Support for Java 1.5 features** - As mentioned, some features do not fully support Java 1.5. Although it has just been introduced, its use may become more widespread rapidly and may be taught extensively to the first year.

- **Bad style ‘templates’** - Rather than using the current implementation, a generic pattern matching algorithm could be developed in which patterns to detect are defined similar to a language’s grammar. This might even allow for checking in various languages without need for much adaptation. Also fix patterns like `if(cond) doX(); else if(cond2) doX();` to: `if(cond||cond2) doX();`.

- **StyleChecker hierarchy** - Better performance could be achieved by combining the ‘first phase’ of StyleChecker implementations. If, for instance, many style checks are concerned with conditionals (‘if’ statements) then it would be beneficial if the code was searched for those expressions only once.

7.6.2 Eclipse as a Teaching platform for Kenya

- **Explicit Supervisor control** - Allow supervisors to change settings for particular workstations or based on user information (login name for instance). These settings would include forcing certain style checks to be enabled or disabled or changing the enablement of certain features (such as code completion), thereby allowing a more gradual and **selective introduction of features** to the students.

- **Modular language features** - Designing Kenya in such a way that particular features could be enabled or disabled at will (for instance: add or remove the ability to cast classes, add or remove the ability to use packages, etc.) would create a **true teaching language** that can be adapted to the level at which the students are familiar with the language concepts and blend extremely well with ‘explicit supervisor control’. Unfortunately this could prove difficult to implement.

8 BIBLIOGRAPHY

[CC] *Class Compass*
Mervis Learning Designs, Ltd.
http://www.classcompass.com
http://www.mervlink.com

[CS] *Eclipse Checkstyle Plug-in reference*
Oliver Burn
http://checkstyle.sourceforge.net

[E-PDE] *Eclipse Plug-in Development Environment Guide*
IBM
http://www.eclipse.org (Documentation 3.0)

[E-NG] *Eclipse Corner Newsgroups*
news://news.eclipse.org/eclipse.platform

[E-PDG] *Eclipse Platform Plugin Developer Guide*
IBM
http://www.eclipse.org (Documentation 3.0)

[GILD] *GILD: Groupware enabled Integrated Learning and Development*
Computer Science Department, University of Victoria, California
http://gild.cs.uvic.ca

[GILD1] *Adopting GILD: An Integrated Learning and Development Environment for Programming*
Margaret-Anne Storey et al.
http://gild.cs.uvic.ca/docs.html

[K] *Kenya*
Technical Report by Tristan Allwood, Robert Chatley and Matthew Sackman
Imperial College London, 2004
http://www.doc.ic.ac.uk/kenya

[NJP] *Novice Java Programmers' Favourite Mistakes*
Neal Ziring
http://users.erols.com/ziring/java-npm.html

[RC] *Java for Beginners*
Technical report by Robert Chatley
Imperial College London, 2001
http://chatley.com/kenya/thesis

[RCTT] *Learning to Program in Eclipse*
Robert Chatley and Thomas Timbul
Imperial College London, 2005
Included in Appendix F

[R-GC] *Automated Code Smell Detection and Refactoring by Source Transformation*
Scott Grant and James R. Cordy
IEEE Working Conference on Reverse Engineering, 2003

[R-SSL] *Metrics Based Refactoring*
(in CSMR, pages 30-38)
Frank Simon, Frank Steinbrückner, Claus Lewerentz
(Software Systems Engineering Research Group)
Technical University Cottbus, Germany, 2001

[R-MF] *Refactoring: improving the design of existing code*
Martin Fowler
Addison-Wesley Longman Publishing Co., Inc., 2000

[SE4] *Software Engineering - Environments*
Teaching Material by Susan Eisenbach
Imperial College London, 2004
http://www.doc.ic.ac.uk/~sue/475/

[SUE] *Kenya notes*
Teaching material
Susan Eisenbach
Imperial College London, 2004
http://www.doc.ic.ac.uk/~sue/121/index.html

[TIDE] *Top 7 Java Integrated Development Environments (IDE)*
Kevin Taylor
About.com
http://java.about.com/od/idesandeditors

[TJE] *Top 5 Java Editors for Beginning Programmers*
Kevin Taylor
About.com
http://java.about.com/od/idesandeditors

9 APPENDIX