JClone: Syntax tree based clone detection for Java
Abstract

An unavoidable amount of money is spent on maintaining existing software systems today. Software maintenance cost generally higher than development cost of the system therefore lowering maintenance cost is highly appreciated in software industry.

A significant part of maintenance activities is related to repeating the investigation of problems and applying repeated solutions several times. A software system may contain a common bug in several different places and it might take extra effort and time to fix all existences of this bug. This operation commonly increases the cost of Software Maintenance Activities.

Detecting duplicate code fragments can significantly decrease the time and effort therefore the maintenance cost. Clone code detection can be achieved via analyzing the source code of given software system. An abstract syntax tree based clone detector for java systems is designed and implemented through this study.

This master thesis examines a software engineering process to create an abstract syntax tree based clone detector for the projects implemented in Java programming language.

Keywords: clone detection, abstract syntax tree, java, eclipse, software, maintenance
# Table of Contents

1. **INTRODUCTION** ........................................................................................................ 1
   1.1. **PROBLEM** .................................................................................................................. 1
   1.2. **GOALS** ....................................................................................................................... 2
   1.3. **CRITERIA** ................................................................................................................... 3
       1.3.1. **Restrictions** ........................................................................................................ 3
   1.4. **MOTIVATION** ............................................................................................................ 4
   1.5. **OUTLINE** ................................................................................................................ 4

2. **BACKGROUND** .............................................................................................................. 5
   2.1. **EVOLUTION OF SOFTWARE SYSTEMS** ................................................................. 5
   2.2. **MAINTAINING SOFTWARE SYSTEMS** ..................................................................... 6
   2.3. **ABSTRACT SYNTAX TREE** ...................................................................................... 7
   2.4. **ECLIPSE** .................................................................................................................. 8
       2.4.1. **Eclipse JDT** ...................................................................................................... 8
       2.4.2. **Eclipse PDE** .................................................................................................... 8

3. **REQUIREMENTS AND ANALYSIS** .............................................................................. 10
   3.1. **VISION** .................................................................................................................... 10
   3.2. **ACTORS** ................................................................................................................... 10
   3.3. **FEATURES** ............................................................................................................. 11
   3.4. **USE-CASES** ............................................................................................................ 12
       3.4.1. **Use-Case Model** ............................................................................................ 12
       3.4.2. **Detailed Use-Case Information** ...................................................................... 13
   3.5. **REQUIREMENTS** .................................................................................................... 17

4. **ARCHITECTURE** ............................................................................................................. 19
   4.1. **ARCHITECTURE OF ECLIPSE** .............................................................................. 19
   4.2. **ECLIPSE PLUG-IN MODEL** .................................................................................... 19
   4.3. **JCLONE ARCHITECTURE** ....................................................................................... 21
       4.3.1. **CloneDetector** ............................................................................................... 22
       4.3.2. **CloneContainer** ............................................................................................. 22
       4.3.3. **Visualizer** ....................................................................................................... 22
       4.3.4. **TreeMapViewer** .............................................................................................. 22
       4.3.5. **PrefuseTreeMap** ............................................................................................. 22

5. **DESIGN AND IMPLEMENTATION** ............................................................................ 23
   5.1. **UML CLASS DIAGRAM** ......................................................................................... 23
       5.1.1. **CloneDetector class** ....................................................................................... 25
       5.1.2. **ICompilationUnit Interface** ............................................................................. 25
       5.1.3. **VisitCompUnit Class** ..................................................................................... 26
       5.1.4. **ASTVisitor Abstract Class** ............................................................................ 27
       5.1.5. **GMLVisitor Class** .......................................................................................... 27
       5.1.6. **CloneVisitor Class** ........................................................................................ 27
       5.1.7. **CloneContainer Class** .................................................................................. 28
       5.1.8. **ModifyClones Interface** ................................................................................. 29
       5.1.9. **AccessClones Interface** .................................................................................. 29
       5.1.10. **ViewerWrapper Class** ................................................................................. 29
       5.1.11. **DefaultTreeView Class** ............................................................................... 30
       5.1.12. **DataSetInterface Interface** ......................................................................... 30
       5.1.13. **ViewerModifier Interface** ............................................................................ 30
       5.1.14. **GraphInterface Interface** ............................................................................. 30
       5.1.15. **ViewPart Abstract Class** .............................................................................. 30
       5.1.16. **CodeCloneGraphView Class** ........................................................................ 30
       5.1.17. **ExternalTreeMapView Class** ........................................................................ 30
       5.1.18. **MainStart Class** ........................................................................................... 31
       5.1.19. **JApplet Class** .................................................................................................. 31
   5.2. **IMPLEMENTATION ASPECTS OF JCLONE** .............................................................. 31
       5.2.1. **Eclipse Plug-in and plugin.xml** ....................................................................... 31
List Of Tables

TABLE 3.1 Feature List for JClone ................................................................. 12
TABLE 3.2 Use-Case UC1: Select Projects ....................................................... 14
TABLE 3.3 Use-Case UC2: Select Classes ......................................................... 14
TABLE 3.4 Use-Case UC3: Analyze Projects ..................................................... 15
TABLE 3.5 Use-Case UC4: Analyze Classes ..................................................... 15
TABLE 3.6 Use-Case UC5: Detect Clones ........................................................ 16
TABLE 3.7 Use-Case UC6: Display Results ...................................................... 16
TABLE 3.8 Use-Case UC7: Calculate Similarity .............................................. 17
TABLE 3.9 List of Requirements for JClone ..................................................... 18
List Of Figures

Figure 2.1 Waterfall Model
Figure 2.2 Rational Unified Process
Figure 2.3 Abstract Syntax Tree of example syntax
Figure 3.1 Use-Case Model for JClone
Figure 4.1 Architecture of Eclipse (Ventribe 2006)
Figure 4.2 An Example of ‘plugin.xml’ file
Figure 4.3 An example MANIFEST.MF file
Figure 4.4 Component Diagram for JClone
Figure 5.1 UML class diagram for JClone
Figure 5.2 CloneDetector Class
Figure 5.3 Selecting Java files to analyze
Figure 5.4 VisitCompUnit Class
Figure 5.5 Justified implementation of VisitCompUnit class
Figure 5.6 CloneVisitor and Singleton CloneContainer
Figure 5.7 Justified implementation for CloneVisitor class
Figure 5.8 CloneContainer class and singleton association
Figure 5.9 Simplified implementation of singleton class
Figure 5.10 Existing methods of ModifyClones interface
Figure 5.11 AccessClones interface
Figure 5.12 ViewerModifier interface
Figure 5.13 Significant code fragment for Swing integration
Figure 5.14 The plugin.xml for JClone
Figure 5.15 List of depending Eclipse components
Figure 5.16 External dependencies of JClone
Figure 6.1 Type-2 clones which are copied and modified from each other
Figure 6.2 Menu item for selected projects to start clone detection
Figure 6.3 Additional views to see the clone information
Figure 6.4 TreeMap integrated within Eclipse
Figure A.1 pom.xml of JClone for Maven
Figure A.2 JClone menu item
Figure A.3 Swing TreeMap implementation integrated in Eclipse
Figure A.4 JClone Code Clone Graph View
**Glossary**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Window Toolkit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Developer Tools</td>
</tr>
<tr>
<td>JLS</td>
<td>Java Language Specification</td>
</tr>
<tr>
<td>PDE</td>
<td>Plug-in Development Environment</td>
</tr>
<tr>
<td>POM</td>
<td>Project Object Model</td>
</tr>
<tr>
<td>RCP</td>
<td>Rich Client Platform</td>
</tr>
<tr>
<td>SWT</td>
<td>Standard Widget Toolkit</td>
</tr>
<tr>
<td>UC</td>
<td>Use Case</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
1. Introduction

Software Engineering Lifecycle does not end with the accomplishment of the software development phase. Generally, software systems are designed to be durable, long-term solutions. As a matter of fact, durability of software systems should be addressed by reliability and maintainability attributes. Both Software Reliability and Software Maintainability attributes are described among the main characteristics of the ISO 9126 Software Quality Model.

Software systems often tend to adapt changes against environmental requirements. Maintenance activities follow after development and installation of a software system while it is being operated. According to previous studies and statistics, more than 60% of software cost is spent on maintenance activities (Granja-Alvarez & Barranco-Garcia 1997, p.1). For most software projects the cost for maintenance is higher than the cost for developing the actual software.

International Standards Organization (ISO) suggests The Software Maintenance Process as a standard for maintenance activities of a software system. In ISO 14764 Software maintenance is addressed by “the totality of activities required providing cost-effective support to a software system” (ISO/IEC 2006); therefore, reducing the cost of software maintenance activities is one of the most important aspects in the Software Maintenance Process. However, ISO’s Software Maintenance Process does not include operational activates of a software system.

Evolution of software systems might result in source code duplication. Most of the time, the system implementers do not take notice of these duplicate code fractions. Exact or near-miss clones of software code may appear due to programming behaviors, design patterns, copy/paste programming etc. In addition, duplicate code fragments might result in some disadvantages from the viewpoint of software quality. Encapsulation addresses better maintainability; therefore, in order to improve maintainability of the software system, detecting duplicate code fragments is quite important. Sometimes duplicate code might contain bugs and when it comes to fixing bugs it would be critical to detect duplicate fragments to be able to fix bugs in all occurrences.(Li et al. 2006)

Previous studies show that between 7% and 23% of software systems contain duplicate code fragments (Roy et al. 2009). Analyzing software systems and detecting code clones is a challenging task but have important improvements on maintainability, which leads to efficiency on long-term software costs.

1.1. Problem

Due to implementation behaviors of software programmers, similar code fragments may appear in different parts of the software system. Sometimes, programmers intentionally copy/paste the code fragment and use it either with some modification or without any modifications in order to implement different parts of software system. Most of the time, programmers prefer to reuse the code for well-known solutions. This behavior leads the programmer to copy proven code fragment and paste into another place in the implemented code of software system. Programmers may achieve time efficiency with this approach but on the other hand they might lose control of implemented software system.

The problem could be addresses as defining and analyzing code clones on large-scale Java projects by the help of designing and creating a tool for developers which has ability to operate in integrated development environment.

The purpose of this thesis is detection of code clones in Java projects and visualizing clone information in a significant way.
Code clones may unintentionally be injected by the programmers due to the fact that they need to comply to some well know solutions or best practices like exception handling, user interface design etc. In such cases code clones “may carry domain knowledge” or “indicate presence of an aspect”. Fixing the bugs efficiently, in the code fragments, affect system functionality if there are several code fragments that are addressed by the clone detector as code clones.

According to (Roy et al. 2009) there are four different types of code clones. The first type of clones is an exact copy of a code fragments except white spaces and comments. The second type code clones, which are syntactically equal, contain small modifications such as renaming variables and methods. The third type code clones are copied code fragments with slightly syntactical differences that may contain extra statements etc. The last type of code clones consist of functionally equivalent but implementation wise different code fragments.

Detection of code fragments, which constitute code clones, is a non-trivial task because it is neither predictable nor obvious under what condition the code clones may appear. As a result, detailed analysis of software code is a must to be able to detect duplicate code.

Previous studies presented several different approaches for code clone detection. Some of those methods uses text based searching and matching; some of them focuses on lexical analysis and token based matching; some techniques are based on abstract syntax tree analysis to verify syntactical equivalence.

This master thesis focuses on abstract syntax tree based code clone detection for large-scale Java projects. From the developer point of view, processing speed of code clone analysis is an important aspect due to cost and efficiency requirements. Especially when it comes to enterprise products and large-scale projects clone code analysis operation could be very complex. In order to decrease time complexity, hashing technique of abstract syntax trees is used, which had been examined by (Baxter et al. 1998).

Another important aspect for the developers is having tools that are accessible inside from an Integrated Development Environment (IDE). Code clone detection tools, which lack of IDE support, are usually not preferred by software developers. Most of the software developers do not want to use an external tool to inspect their software code because generally, code analysis operations depend on software dependencies and external tools cost extra time to configure.

Several tools are using AST based clone detection but none of them includes native support of IDE for AST representation. Since most of integrated development environments are already able to compile software code; therefore, they already have ability to create AST. General approach was selecting or creating a third party AST representation which may cause several issues when it comes to sustain future changes on language specification, future updates on code clone detection tool etc.

1.2. Goals
The main goal of this thesis is applying the theory of the clone detection and to create a prototype that implements AST based code clone detection on Java projects.

The implementation should also be able to visualize duplicate code fragments in a certain way that helps user to distinguish code fragments. Colorizing the duplicate code fragments using the same color codes could be an option as well as linking the code clones which would make it easier for the user to see the similarity.

Another way to visualize results could be using the treemaps. Using the treemaps in an efficient way to visualize code clones is explained by (Pan & Wang 2009).
Integrating this approach or implementing a similar solution would be successful enough to solve addressed problem.

In order to support up-to-date Java programs, the latest version of the Java Language Specification should be valid for JClone, which means that the tool should be able to analyze and find clones on all valid Java projects. When it comes to real world examples it is neither acceptable nor usable to support only a subset of Java language.

1.3. Criteria

Creating an Eclipse plug-in to detect code clones is the main outcome of this master thesis. Evaluation of the Eclipse platform among others will be explained in the following chapters. The JClone, as an Eclipse plug-in, should have abilities to analyze different projects and display similarities, which means code clones, between selected projects.

Analyzing large-scale Java projects in an efficient time and detecting code clones is addressed as mandatory goals of this master thesis. In view of the fact that code clone detection could be very complex task; therefore, it is a must to use an efficient algorithm and data structures that reduce time complexity of the process. The tool that we are going to create will be named JClone.

From a user point of view, it is important to integrate the code clone detector within the IDE. This ability provides more effective usability for the software developers.

Presentation of code clones and visualization of results are other important aspects from a user point of view. The system should be able to visualize the duplicate code in a user-friendly interface. Perceiving code fragments that construct code clones should be easy for software developers.

Visualization should be easily distinguishable by the users; in other words the user of this system, the software developer, should be able to browse the code clones and relations between them.

By the help of the Eclipse Plug-in architecture different components should be designed to detect clones and visualize results. Several options should be available in case the system needs modification on future releases.

The decomposition of a designed system should give opportunities to extend or change the components of the system. The detection of clones and visualization of the results should be encapsulated by different components in order to provide a flexible system design.

JClone should contribute on improvement of maintenance activities of enterprise level Java projects.

1.3.1. Restrictions

JClone utilizes an AST based code clone detection method. AST based detection techniques are dependent on the programming language. In other words, AST based techniques have a programming language constraint in opposition to text-based techniques operating on any programming language. In our case, the Java programming language is the desired language by JClone to detect duplicate code fragments.

Java code and all necessary dependencies should be resolved before the code clone analysis begins. JClone cannot analyze or detect code fragments on the projects that have compilation errors.

Detecting Type-3 code clones, which may introduce extra logic on type-2 clones’ minimum granularity level, is not in the interest area of this master thesis. In other words, JClone will be able to detect only type-1 and type-2 code clones. (Roy et al. 2009, p.472)
1.4. Motivation
Maintenance cost of software systems is generally higher than implementation cost. According to (Brooks 1975) more than 90% of software cost belongs to software maintenance activities; furthermore, it is unavoidable to skip maintenance activities if it is desired to achieve the goal of having a successful system. According to international standard of Software Maintenance Process “The maintainer shall conduct analysis and determine which documentation, software units and versions thereof need to be modified. These shall be documented.” (ISO/IEC 2008, pt.6.4.10.3.3.1) JClone is a tool that is intended to address software units, which consist of duplicate code fragments and are important to determine while applying maintenance activities successfully.

According to (Roy et al. 2009) between 7% up to 23% of software systems contain clone code. These software code fragments may sometimes be injected intentionally but most of the times the duplicate code fragments contain domain knowledge.

Code clone detection could be useful in many ways such as for the purpose of plagiarism detection, aspect mining as well as well as decreasing the cost of software maintenance activities. Detection of duplicate code fragments increases understandability of software systems and may help system maintainers to increase code quality of the existing system.

According to (Lehman 1985) “The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.” Consequently, detection of duplicate code fragments leads to efficiency on the Software Maintenance Process and decreases maintenance costs while increasing software code quality.

1.5. Outline
The structure of this paper is addressed as follows; Chapter 2 provides essential background information about the problem and goal domain while presenting code clone detection techniques, Eclipse and the plug-in architecture. Chapter 3 clarifies the actors and features for JClone, this chapter also provides requirements specification. Chapter 4 mainly gives information about the architecture of JClone and Eclipse. Chapter 5 describes the design and implementation of JClone. Furthermore, this chapter provides detailed information about the software components, packages and classes that are parts of JClone. In Chapter 6 an evaluation of the suggested solution is presented. Chapter 7 gives information about results of the related work and future work.
2. Background

In order to understand the idea behind syntax tree based clone detection, brief information about underlying concepts is necessary. This section starts with definitions of clone fragment and code clone and then explains the important concepts.

Definition of clone fragment by (Roy et al. 2009, p.471):

*A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin–end block, or sequence of statements. A CF is identified by its file name and begin–end line numbers in the original code base and is denoted as a triple (CF.FileName, CF.BeginLine, CF.EndLine).*

Based on this clone fragment definition there we can define the notion of a code clone definition made by (Roy et al. 2009):

*A code fragment CF2 is a clone of another code fragment CF1 if they are similar by some given definition of similarity, that is, \( f(CF1) = f(CF2) \) where \( f \) is the similarity function (see clone types below). Two fragments that are similar to each other form a clone pair (CF1, CF2), and when many fragments are similar, they form a clone class or clone group.*

2.1. Evolution of Software Systems

Every Software System needs a change at some point no matter it requires minor corrections or larger modifications. The term evolution can be described as “a process of continuous change from a lower, simpler, or worse to a higher, more complex or better state.” (Williams & Carver 2009) The evolution of Software Systems is a necessity in order to meet the changing and expanding needs of system users. Software systems are also changed as a response to reorganization of the company that is using the system. Other reasons for the changes are to correct existing errors in the original application or due to newly introduced computers that have to work with the older system. Furthermore, the external environment of the system may have been altered, which requires a change in the software system itself (Sommerville 2007, p.33).

![Figure 2.1 Waterfall Model](image)

Software evolution is often a fundamental part of a software process; there are many kinds of software processes such as the basic waterfall model, as in Figure 2.1, or the
complex Rational Unified Process (RUP), which is illustrated by Figure 2.2, where
software evolution is a recognized part of the model. Evolution is often, when it is
included in the model, described in the last phase of any given process. It serves to
ensure that the software fits the needs of the customer that can change over
time.(Sommerville 2007, p.64)

![Figure 2.2 Rational Unified Process](image)

One of the main reasons for software evolution is the inherent flexibility of a
software application. In comparison to hardware, software is less expensive to make
changes to during or after the development of the system. (Sommerville 2007, p.81) But
on the other hand, in comparison to development activities, it is more expensive to
make changes on maintenance phase of a system.

2.2. Maintaining Software Systems

One major part of software evolution is software maintenance. According to Horch, the
activity “can best be viewed and treated as an extension or repetition of the
development process.”(Horch 2003, p.20)

Although software maintenance is a recognized part of software engineering, it has
not always been a popular activity. While software development historically has been
seen as an artistic and inspiring act, the maintenance of a system has been labeled as
dull or less interesting. A contribution factor to this point of view may also be that the
cost of maintenance often is higher than the original cost for developing the software
system. Although maintainability has had this viewpoint in the past, it is becoming more
evident to maintain a software system and the divergence of development and
maintenance is becoming less important (Sommerville 2007). According to Grubb &
Takang, the value of good software maintenance skills is being increasingly asked for
(Grubb 2003, p.7).

There are two main activities connected to software maintenance; the first activity
involves correcting the errors that were discovered after the system has been installed
that were not found during the testing phase and the second activity concerns changing
the software after it has been deployed to fulfill new or modified requirements.(Horch
2003, p.21)
The maintenance process is an important part of the framework of software maintenance where one has to analyze elements such as programming practice, changed requirements of the software and error detection. The first point addresses the variation in programming practice. The use of operations that sets a certain program structure can vary between programmers and/or organizations. Without any consistency, problems could arise. Obtaining changed requirements means understanding the changes that are to be made. Changed requirements are as stated before one of the main activities in maintaining software; this is due to the fact that it can be rather difficult for a user to describe to a programmer or an analyst in words what is desired of the future software system that is to be developed. This is often referred to as the so called “information gap” between the user of the system and the programmer who develops the application.

Error detection is another fundamental part of the maintenance process. There are no software systems without any errors; software will always contain defects that are either detected in the testing phase or during maintenance. Generally, the later an error is found in the life cycle of an application, the more it costs to fix it. (Grubb 2003, p.24)

Software maintenance has gone from being an almost invisible activity in the middle of the 20th century to a well-recognized process of software engineering in the beginning of the 21st century. According to Grubb & Takang, software maintenance will be an advanced major driver in the future of the software industry. (Grubb 2003, p.313)

2.3. Abstract Syntax Tree

An Abstract Syntax Tree (AST) can be described as “a tree representation of the abstract (simplified) syntactic structure of source code written in a certain programming language”. The syntax tree is referred to as an abstract visualization as the tree does not show all the details that exist in the real syntax. A node in syntax tree represents a statement or at least a meaningful piece of source code. (Wikipedia 2009a)

To give an example, the tree below depicts an abstract syntax tree for the following concrete syntax:

![Abstract Syntax Tree of example syntax](image)
Concrete syntax is usually understandable for humans while abstract syntax trees are more useful for computers. (Friedman 2008, p.51)

2.4. Eclipse
Eclipse is an environment for developing software that has an Integrated Development Environment (IDE), which is a comprehensive tool for developers in the creation of software. Eclipse also has an extended plug-in system and supports multiple languages such as C, C++, COBOL, Python, Perl, PHP, and others even though the preferred language is Java. Eclipse is mainly written in the Java language and consists of a small run-time kernel; apart from this kernel, the rest of the components in Eclipse are plug-ins. (Wikipedia 2009b)

IBM founded the Eclipse Project in November 2001 as an open source community. The main focus of the project is aiming to build a platform for open development that consists of frameworks, tools and runtimes for the construction, deployment and management of software during its life cycle. The Eclipse Foundation hosts the projects of Eclipse in a non-profit manner in order to keep the community vendor neutral and open for everyone. The community is run by organizations and individuals coming from a wide range of different areas of the software industry. (Eclipse.org 2009a)

The members financially support the Foundation of Eclipse with annual fees. The committers of Eclipse are usually either working independently or for organizations; the people who are working individually for Eclipse are programmers that work voluntarily on a specific open source project. (Eclipse.org 2009a)

2.4.1. Eclipse JDT
The Eclipse project JDT, which stands for Java Development Tools, gives the tool plug-ins for a Java IDE that supports Java application development. It is further described as adding, “a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools.” The JDT project gives Eclipse the possibility to work for itself as a development environment.

The components of JDT can be seen in the figure below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT</td>
<td>Java 5.0 annotation processing infrastructure</td>
</tr>
<tr>
<td>Core</td>
<td>Java IDE headless infrastructure</td>
</tr>
<tr>
<td>Debug</td>
<td>Debug support for Java</td>
</tr>
<tr>
<td>Text</td>
<td>Java editing support</td>
</tr>
<tr>
<td>UI</td>
<td>Java IDE user interface</td>
</tr>
</tbody>
</table>

The project of Eclipse JDT consists of several components where each part functions like a project on its own with committers, bug categories and mailing lists belonging to each component. (Eclipse.org 2009b)

2.4.2. Eclipse PDE
The Eclipse PDE stands for Plug-in Development Environment and gives the programmer tools for creating, developing, testing, debugging, building and deploying fragments, features and plug-ins of Eclipse. Furthermore, Eclipse PDE gives tools for OSGi, making it a well-suited environment for component programming.

The components of PDE can be viewed in the following figure. (Eclipse.org 2009c)
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE Build</td>
<td>Ant based tools and scripts to automate build processes</td>
</tr>
<tr>
<td>PDE UI</td>
<td>Models, builders, editors and more to facilitate plug-in development in the Eclipse IDE</td>
</tr>
<tr>
<td>PDE API Tools</td>
<td>Eclipse IDE and build process integrated tooling to maintain API</td>
</tr>
<tr>
<td>PDE Incubator</td>
<td>Development of new tools that are not ready to be added to the Eclipse SDK</td>
</tr>
</tbody>
</table>
3. Requirements and Analysis

This chapter gives fundamental information about the requirements of JClone. The first subsection of this chapter gives information about system actors, which is necessary from a design point of view. Following sub-sections describe use-case analysis and recommend a use-case model as a solution. At the end of this chapter formal requirements are addressed and detailed use-cases are created.

3.1. Vision

This subsection describes the system from an end-user point of view.

As a software engineer I am working with maintenance activities of enterprise level legacy Java Systems and I often make changes on software code to be able to fix reported bugs. Most of the time it is trivial to find the reported bug with bug reports but at the same time it is a non-trivial task to reach non-reported system bugs caused by copy/paste programming. I want to be able to analyze the software code of an existing software system to detect duplicated bugs caused by duplicate code fragments.

As a software engineer I am responsible of the general code quality of the implemented software systems. I want to analyze implementation against copy/paste programming activities. I want to address duplicate code fragments in order to check whether they contain domain knowledge or not. I want to analyze the implementation and detect code clones in order to achieve better encapsulation rate and increase code quality.

As a Computer Science teacher I hand out several programming tasks to my students in order to examine their adequacy in programming classes. Even though it is not very common for students to refer each other’s solutions I want to be aware of the similarity proportion between solutions of different students. I want to be able to compare at least two different solutions and see possible duplicate code fragments even if duplicate code fragments have modified variable or method names. I want to be able to see clone rate for different classes in a solution or among all given solutions.

As a Software Engineer or teacher I need a tool, which is bundled to my existing software development environment. I do not want to deal with external tools to detect duplicate code fragments, which will be more time consuming. This is due to the fact that, most of the time, the external tools have to resolve dependencies of my software projects and need extra effort to be able to accomplish the given tasks. I do not want to spend extra time on configuration just to enable analyzability of a given project.

As a future developer of this system I need it to be developed in a component-based architecture in order to extend or enable some functionality easily in the future. I want to be able to change different modules of the system without any difficulty, in order to achieve different views on the result set. More specifically, I want to be able to change how the system presents the results easily or create information export tools in an optimal way.

3.2. Actors

Actors are important to be able to identify use cases and cerate a requirements specification from the vision document.

Several actors can be derived from the vision document, which is described in the previous subsection:

The first actor is the end-user who is intended to detect code clones due to several different reasons. In other words Software Engineers, Software Maintainers, Programmers or teachers can be addressed as the end-user actor of the system. The end
user is the actor who wants to use JClone in order to detect code clones or see similarity rates between different projects or classes.

The second actor is the Eclipse Integrated Development Environment. Eclipse IDE is responsible for deployment and execution of JClone. At the same time it provides a working environment for the end-user who interacts with JClone.

The third actor is the JClone itself, which interacts with the end-user and Eclipse to since it is in need of performing various operations.

The last actor of JClone is the software maintainer who is intended to maintain the system for future releases or extend the functionality to achieve some other goals.

3.3. Features

This subsection describes and gives detailed information about features of JClone. The features listed below in Table 3.1 are obtained from the problem domain and will be used to define use cases and formal requirements.

<table>
<thead>
<tr>
<th>Feature ID</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature 1</td>
<td>Capability to be bundled in Eclipse IDE</td>
</tr>
<tr>
<td></td>
<td>JClone should be designed as an Eclipse plug-in which enables the end-user to analyze the software code directly inside from Eclipse IDE</td>
</tr>
<tr>
<td>Feature 2</td>
<td>Capability to analyze selected Java classes</td>
</tr>
<tr>
<td></td>
<td>JClone should have the possibility to analyze a certain number of selected Java files. This is important since the user may want to analyze just one or several classes rather than analyzing the whole project each time.</td>
</tr>
<tr>
<td>Feature 3</td>
<td>Capability to analyze selected projects in workspace</td>
</tr>
<tr>
<td></td>
<td>JClone should be able to analyze selected Java projects in order to detect duplicate code fragments.</td>
</tr>
<tr>
<td>Feature 4</td>
<td>Capability to calculate similarity rates of classes</td>
</tr>
<tr>
<td></td>
<td>JClone should include a feature to calculate similarity rates between individual classes. In order to achieve this goal JClone should keep track of equal AST nodes.</td>
</tr>
<tr>
<td>Feature 5</td>
<td>Capability to visualize code clones</td>
</tr>
<tr>
<td></td>
<td>JClone should be able to visualize the duplicate code fragments in a way that helps the end-user to distinguish duplicate code fragments.</td>
</tr>
<tr>
<td>Feature 6</td>
<td>Capability to locate duplicate code fragments</td>
</tr>
<tr>
<td></td>
<td>JClone should be able to look through code clones for editing purposes.</td>
</tr>
<tr>
<td>Feature 7</td>
<td>Capability to analyze projects in a time-efficient way</td>
</tr>
</tbody>
</table>

3.4. System Architecture

The architecture of JClone is designed in a way that it can be integrated into Eclipse IDE. The system consists of three main components: user interface, core functionality, and communication with Eclipse IDE. The user interface is responsible for receiving user inputs and displaying results. The core functionality is responsible for performing code analysis and similarity calculations. The communication component is responsible for sending and receiving data between the user interface and the core functionality.
JClone should make it possible for the end-user to analyze big scale Java projects in a useful time fragment. In order to achieve efficient processing time it should implement a fast algorithm, which enables hashing the AST nodes. JClone should use concurrent programming techniques and activate multi thread calculation in order to achieve faster calculation.

Table 3.1 Feature List for JClone

3.4. Use-Cases
This section is intended to give information about the use-case model in JClone. Since there is a need of giving, a clearer perspective of the uses-cases, a use-case model is firstly created first. Afterwards, each use case is explained in detail.

3.4.1. Use-Case Model
This subsection describes the use-case model of JClone. Figure 3.1 explains how the end-user, JClone and Eclipse platform interact with each other in order to perform different kind of operations. This use-case model, shown in Figure 3.1, is derived from problem domain analysis which is done to make it fit into the feature set.
3.4.2. Detailed Use-Case Information

This subsection focuses on giving more details about, all mentioned, use-cases of JClone. Each use case have the following properties: Brief Description, Triggering Event, Flow Of Events (Basic & Alternative), Pre-conditions, Post-conditions, Extension points and Actors involved.

Table 3.2 gives detailed information about the use-case Select Projects.
Use-Case UC1: Select Projects

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Select one or more projects to analyze with JClone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggering event</td>
<td>The end-user selects a project with mouse pointer</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Basic: The end-user clicks on a Java project from the Eclipse projects list. The end-user right clicks on the selection and chooses option analyze from the JClone Menu. Alternative: The end-user selects multiple projects by pressing the CTRL key before selecting the projects.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The Java project should be ready to compile which means that it should not contain any syntax errors or missing dependencies</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>JClone is ready to work with the selected project input</td>
</tr>
<tr>
<td>Extension Points</td>
<td>-</td>
</tr>
<tr>
<td>Actors</td>
<td>The end-user, JClone</td>
</tr>
</tbody>
</table>

Table 3.2 Use-Case UC1: Select Projects

Table 3.3 explains the use case Select Classes.

Use-Case UC2: Select Classes

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Select one or more Java classes to analyze with JClone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggering event</td>
<td>The end-user selects a Java class file with mouse pointer</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Basic: The end-user clicks on a Java class file from Eclipse projects list. The end-user right clicks on the selection and chooses the option analyze from the JClone Menu. Alternative: The end-user selects multiple Java class files by pressing the CTRL key before clicking on the projects.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The Java classes should be ready to compile which means that, they should not contain any syntax errors or missing dependencies</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>JClone is ready to work with the selected Java files input</td>
</tr>
<tr>
<td>Extension Points</td>
<td>-</td>
</tr>
<tr>
<td>Actors</td>
<td>The end-user, JClone</td>
</tr>
</tbody>
</table>

Table 3.3 Use-Case UC2: Select Classes

Table 3.4 and Table 3.5 gives detailed information about the use-cases for analyze operations.
## Use-Case UC3: Analyze Projects

### Brief Description
Analyze selected Java Projects to find code clones

### Triggering event
The end-user clicks analyze on the action option from the JClone menu for selected projects

### Flow of Events

<table>
<thead>
<tr>
<th>Basic</th>
<th>JClone creates AST for the Java classes of the selected projects. JClone runs code clone detection algorithm over generated AST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative</td>
<td></td>
</tr>
</tbody>
</table>

### Pre-conditions
Projects are already selected by the end-user

### Post-conditions
Code clone information is generated.

### Extension Points
Includes: Select Projects (UC1), Inherits: Detect Clones

### Actors
The end-user

---

## Use-Case UC4: Analyze Classes

### Brief Description
Analyze selected Java Classes to find code clones

### Triggering event
The end-user clicks on the analyze action option from the JClone menu for selected classes

### Flow of Events

<table>
<thead>
<tr>
<th>Basic</th>
<th>JClone creates AST for the selected Java classes. JClone runs code clone detection algorithm over generated AST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative</td>
<td></td>
</tr>
</tbody>
</table>

### Pre-conditions
Classes are already selected by the end-user

### Post-conditions
Code clone information is generated

### Extension Points
Includes: Select Classes (UC2), Inherits: Detect Clones

### Actors
The end-user

---

There is Table 3.6 below, which explains detecting clones use case.
### Use-Case UC5: Detect Clones

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Detect Clones by running clone detection algorithm over ASTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggering event</td>
<td>Analyze Projects/Classes Use cases</td>
</tr>
</tbody>
</table>
| Flow of Events    | **Basic**  
|                   | JClone visits each AST node to find similar nodes  
|                   | JClone records matching AST nodes in Clone Container       |
|                   | **Alternative**                                            |
| Pre-conditions    | Analyze action is performed for projects or classes         |
| Post-conditions   |                                                            |
| Extension Points  | Extends: Display Results (if clones exist),                |
|                   | Includes: Calculate Similarity                             |
| Actors            | JClone                                                     |

**Table 3.6 Use-Case UC5: Detect Clones**

Sixth use-case in Table 3.7 expresses how to display results.

### Use-Case UC6: Display Results

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Visualize Detected Code clones</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggering event</td>
<td>Detect Clones Use case or The end-user enables the visualization</td>
</tr>
</tbody>
</table>
| Flow of Events    | **Basic**  
|                   | The viewer navigates through the code clone container in order to visualize the results  
|                   | Draw an understandable graphical representation of Code clone information |
|                   | **Alternative**  
|                   | The user can change the layout  
|                   | The user can navigate through source of selected code clones |
| Pre-conditions    | Clone detection operation is accomplished |
| Post-conditions   | Code clone information is visible and navigable |
| Extension Points  | - |
| Actors            | JClone, The end-user |

**Table 3.7 Use-Case UC6: Display Results**

Table 3.8 shows the use case of similarity calculation.
**Use-Case UC7**

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Calculate Similarity</th>
<th>Feature 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>JClone calculates a similarity rate based on equal AST node count</td>
<td>Detect Clones</td>
<td></td>
</tr>
</tbody>
</table>

**Triggering event**

- Detect Clones

**Flow of Events**

- **Basic**
  - JClone keeps track of similar node number on each Java class
  - JClone calculates the similarity rate based on similar node number

- **Alternative**

**Pre-conditions**

- Code clone detection started

**Post-conditions**

- Java classes can be related to each other according to similarity rates

**Extension Points**

- -

**Actors**

- JClone

---

### 3.5. Requirements

In this subsection software requirements are derived through use cases and the vision document. There are some different priority levels between the defined requirements in order to complete most desirable system. The implementation of the requirements takes action according to its precedence.

Table 3.9 introduces requirements for JClone and establishes three different priority levels: Essential represents the most important functionality, which is a must, Desirable represents the functionality that should be there in order to increase productivity. Lastly, the priority level, Optional represents the functionality that increases usability.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. JClone should be implemented as an Eclipse plug-in in order to increase usability of the code clone detection operation.</td>
<td>Essential</td>
</tr>
<tr>
<td>2. JClone should enable new menu items in order to start the code clone detection process on selected Java projects or Java source files.</td>
<td>Essential</td>
</tr>
<tr>
<td>3. JClone can analyze Java projects which enables the detection of code clones in selected projects</td>
<td>Essential</td>
</tr>
<tr>
<td>3.1. JClone can detect code clones in different Java projects.</td>
<td>Desirable</td>
</tr>
<tr>
<td>3.2. JClone can detect code clones in selected Java classes.</td>
<td>Essential</td>
</tr>
<tr>
<td>4. JClone uses Abstract Syntax Tree framework provided by Eclipse JDT in order to provide easy adaptation for future releases.</td>
<td>Essential</td>
</tr>
<tr>
<td>5. JClone saves the clone information among multiple analyze sessions in order to enable easy to use analyze scenarios.</td>
<td>Desirable</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6.</td>
<td>JClone calculates similarity rates for each Java source file or ICompilationUnit</td>
</tr>
<tr>
<td>6.1</td>
<td>JClone keeps track of similar node count on each compilation unit</td>
</tr>
<tr>
<td>7.</td>
<td>JClone normalizes the source code in a way that type-2 code clones can be detected.</td>
</tr>
<tr>
<td>7.1</td>
<td>JClone normalizes local variable and field declarations</td>
</tr>
<tr>
<td>7.2</td>
<td>JClone normalizes method declarations</td>
</tr>
<tr>
<td>7.3</td>
<td>JClone normalizes loops</td>
</tr>
<tr>
<td>8.</td>
<td>JClone is fast enough to analyze big-scale Java projects</td>
</tr>
<tr>
<td>8.1</td>
<td>JClone uses hash code based matching which gives the possibility of fast code clone detection</td>
</tr>
<tr>
<td>8.2</td>
<td>JClone enables multi threading, so that code clones can be computed in a shorter time.</td>
</tr>
<tr>
<td>9.</td>
<td>JClone is able to visualize code clone information</td>
</tr>
<tr>
<td>10.</td>
<td>JClone has functionality to navigate selected duplicate code fragments</td>
</tr>
</tbody>
</table>

**Table 3.9 List of Requirements for JClone**
4. Architecture

JClone should provide its functionality inside Eclipse IDE; furthermore it should be fully integrated to the Eclipse plug-in architecture. This need has already been expressed by one of the essential requirements in the previous chapter.

On the other hand JClone should have an extensible architecture in order to provide the result set, the code clone information, among different components. For instance it should be easy to integrate another component to visualize the code clones or it should be easy to export the data in XML format. Moreover, JClone should introduce a well-known interface to make the result set available among many other components.

This chapter starts with a basic understanding of the Eclipse plug-in architecture, which is followed by the architectural design of JClone and introduces a relationship between two main components: Eclipse and JClone.

4.1. Architecture of Eclipse

Eclipse Rich Client Platform (RCP) consists of different components called plug-in. This architecture can be extended in many ways by the help of provided extension points. New plug-ins and features can contribute the Eclipse RCP in a way that improves the overall functionality. Since Eclipse IDE is also a specified set of specialized plug-ins over Eclipse RCP, Eclipse IDE provides various points of extensions that enable extra functionality on top of existing architecture. New tools or plug-ins like JDT or PDE extends platform over these extension points. Each component in Eclipse is a plug-in except a small runtime component. Therefore, each existing component or new integrated component are equal in terms of environment. But on the other hand, various plug-ins may have several dependencies on each other. Figure 4.1 shows the Architecture of Eclipse.

4.2. Eclipse Plug-in Model

Eclipse IDE can successfully be extended with an optimal use of provided extension points. The provided extension points of Eclipse are defined interfaces for new plug-ins to interact with the Eclipse IDE. Eclipse installation directory contains a folder named
‘plugins’ to install new Eclipse plug-ins. This directory also contains native Eclipse components, which are plug-ins too.

A contained ‘plugin.xml’ file defines the Eclipse plug-in and connection points to IDE. And each ‘plugin.xml’ file is connected to a ‘MANIFEST.MF’ file, which defines identical information for a specific plug-in. The Eclipse platform integrates and enables each plug-in in ‘plugins’ according to those configuration files explained. An example of ‘plugin.xml’ is shown at Figure 4.2

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Eclipse version="3.2">
<plugin>
  <extension
    id="sampleApplication"
    point="org.Eclipse.core.runtime.applications">
    <application>
      <run
        class="sampleplugin.Application">
      </run>
    </application>
  </extension>
</plugin>
```

**Figure 4.2 An Example of ‘plugin.xml’ file**

In the plug-in manifest, the ‘plugin.xml’ file utilizes an extension point named `org.Eclipse.core.runtime.applications` to extend Eclipse platform. And according to this extension point `sampleplugin.Application` class will be executed whenever the plug-in is activated.

In addition to ‘plugin.xml’ file the ‘MANIFEST.MF’ file contains package names and dependencies for a plug-in. Figure 4.3 gives an example for the ‘MANIFEST.MF’ file.

```text
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Sample
Bundle-SymbolicName: Sample; singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: sample.Activator
Require-Bundle: org.eclipse.ui,
  org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

**Figure 4.3 An example MANIFEST.MF file**
4.3. JClone Architecture

The architecture of JClone contains different modules to accomplish different operations requested by the user. On the other hand JClone is a module itself and interacts with other modules like the Eclipse Platform and PrefuseTreeMap Viewer, in regards to Eclipse plug-in architecture. Figure 4.4 visualizes the architecture of JClone, components of the whole system and sub components of JClone. Each component exchanges data by the help of defined interfaces and those interfaces can be used by other components too.

![Main Component Diagram](image)

Figure 4.4 Component Diagram for JClone

JClone consist of four sub-components and each sub-component realizes or utilizes several interfaces to communicate the others. However, this suggested architectural design could be modified in order to extend the system in the future. Existing architectural design consist of CloneDetector, CloneContainer, Visualizer and TreeMapViewer components.
4.3.1. CloneDetector
Clone detector can be called as the core component for JClone since, it utilizes Eclipse Java Developer Tools (JDT) component to access AST for selected Java files. JClone is also effective on normalizing the Java code and comparing given AST nodes in order to identify duplicate code fragments. CloneDetector implements multi-thread concurrent programming technique on visiting operation of different java classes. In that way it is expected that the clone detection operation be accelerated while utilizing CPU better.

CloneDetector uses Eclipse JDT in away that gives access to the AST. On the other hand stores significant information to the singleton CloneContainer component over ModifyClones interface.

4.3.2. CloneContainer
The CloneContainer component is designed to be singleton that is accessed by CloneDetector component to be modified and realizes AccessClones interface in respect to serve the clone information to the other components. It is important that this component to be thread safe since multi thread programming may result some data inconsistencies.

4.3.3. Visualizer
The Visualizer component is responsible to display the code clones in a way that makes it is easy to distinguish for the user. It is important to visualize relationships among duplicate code fragments. The suggested solution to achieve this goal is using same colors or adding edges that reflect connections between duplicate code fragments.

4.3.4. TreeMapViewer
TreeMapViewer component is a ‘glue-component’ that enables integration between Eclipse Architecture and an external tool called PrefuseTreeMap. Eclipse is implemented in SWT but on the other hand PrefuseTreeMap is using Swing as Graphical User Interface (GUI) API. The TreeMapViewer component is responsible for utilizing SWT to AWT interface and integrating a Swing based application into the Eclipse IDE. This sub component may introduce some extra logic in respect to achieve a successful integration.

4.3.5. PrefuseTreeMap
This component is an external tool and a proven study to visualize code clones which has been introduced by (Pan & Wang 2009).

The architectural view of JClone and depending systems has been explained in this section. It is quite important to have a modular architectural design in order to achieve well-defined system and high quality software code. A well-defined software system has also advantages on maintenance activities later on.
5. Design and Implementation

The previous chapter was intended to describe JClone and the related components as well as sub components of JClone itself. This chapter focuses on details of the big picture and gives information about the class design with the help of UML class diagram. Figure 5.1 gives information about the class diagram of JClone. Each class is explained in detail afterwards.

5.1. UML Class Diagram

UML class diagram is created according to use-case model and requirements list in respect to object oriented design principles. As an outcome the static object view of the desired system is created. Furthermore the UML class diagram of JClone utilizes architectural design, which is shown on Figure 4.4, with respect to component-based software engineering.

CloneDetector component consist of:
- CloneDetector,
- CloneVisitor,
- GMLVisitor,
- VisitCompUnit; classes.

The second component, CloneContainer, contains listed classes below:
- CloneContainer,
- ViewerModifier,
- ViewerWrapper.

The third component is Visualizer, which displays the clone information as a graph is implemented by a single class: CodeCloneGraphView.

The last component is TreeMapViewer and consist of the ExternalTreeMapView class, which has ability to integrate swing based tree map tool into the SWT based Eclipse View.

Many other classes and interfaces are used to integrate the components to each other. More information about the implementation details, including connection details about the used APIs, exists in the following pages.

The UML class diagram of JClone in Figure 5.1 presents static object structure of the desired code clone detector and the following subsections focus on details of each specific class.
Figure 5.1 UML class diagram for JClone
5.1.1. CloneDetector class

The CloneDetector class is the core component of the system. Main responsibility of this class is creating and running threads that execute VisitCompUnit on each ICompilationUnit object provided by Eclipse JDT. As in presented in Figure 5.2 this class has two private attributes. One for the list of ICompilationUnit instances and the other one is for a list of VisitCompUnit threads.

Iterating into java files for selected projects in order to start code clone analysis on them for selected project is shown in Figure 5.3.

```java
for (Iterator i = selectedFiles.iterator(); i.hasNext();)
    Object s = i.next();
    if (s instanceof IJavaProject) {
        IJavaProject proj = (IJavaProject) s;
        for (IPackageFragment p : proj.getPackageFragments()) {
            for (ICompilationUnit c : p.getCompilationUnits())
                startCompilationUnit(c);
        }
    } else if (s instanceof ICompilationUnit) {
        ICompilationUnit c = (ICompilationUnit) s;
        startCompilationUnit(c);
    }
}
...

private void startCompilationUnit(ICompilationUnit c) {
    Thread t = new Thread(new VisitCompUnit(c));
    threads.add(t);
}
```

5.1.2. ICompilationUnit Interface

This interface is provided by Eclipse JDT API and represents the Java source files that are going to be used for code clone analysis.

Each Java project in Eclipse Workspace is implements IJavaProject interface which contains IPackageFragment instances to represent Java packages. And each package fragment represents Java classes with an object instance of ICompilationUnit.
5.1.3. VisitCompUnit Class

*VisitCompUnit* class is designed to analyze a given Java file with respect to code clone detection operation. This class interacts with Eclipse JDT component in order to access AST of given Java file and employs a *CloneVisitor* object for normalizing the software code and gathering code clone information.

*VisitCompUnit* class implements *Runnable* interface because code clone detection operation can be triggered with many threads from *CloneDetector*. This means every *VisitCompUnit* object runs a single thread for a single Java class to detect code clones. On the other hand *CloneContainer* is a singleton object with synchronized methods to prevent data inconsistencies might be caused by multiple thread access.

As shown in Figure 5.4 and Figure 5.1, *VisitCompUnit* has a composition relationship with *CloneVisitor*. On the other hand this class establishes bi-directional associations towards *ICompilationUnit* and *ViewerWrapper*.

![Figure 5.4 VisitCompUnit Class](image)

On the other hand this class utilizes *GMLVisitor*, which prepares clone data for *ExternalTreeMapView*.

Figure 5.5 contains information about a justified implementation of *VisitCompUnit*.

```java
class VisitCompUnit implements Runnable {
    public VisitCompUnit(ICompilationUnit cls) {
        this.cls = cls;
        parser = ASTParser.newParser(AST.JLS3);
        parser.setKind(ASTParser.K_COMPILATION_UNIT);
        parser.setResolveBindings(true);
        parser.setSource(cls);
    }
    public void run() {
        ASTNode b = parser.createAST(null);
        CloneVisitor cVisitor = new CloneVisitor();
        b.accept(cVisitor);
        GMLVisitor gVisitor = new GMLVisitor();
        b.accept(gVisitor);
        DirectedGraphInterface g1=(DirectedGraphInterface)
            gVisitor.getTreeview().getGraph();
        viewerWrp.addGraph(g1, cVisitor.getRatio());
    }
}
```

![Figure 5.5 Justified implementation of VisitCompUnit class](image)
5.1.4. ASTVisitor Abstract Class

ASTVisitor class is an abstract class which is provided by Eclipse JDT API and its main functionality is traversing the given abstract syntax tree. This class has separate visit methods for different types of ASTNode objects.

5.1.5. GMLVisitor Class

GMLVisitor class extends the visitor functionality and adds some significant logic to prepare code clone information for ViewerWrapper class. Briefly, this class creates a DefaultTreeView instance for representing normalized abstract syntax tree. DefaultTreeView class is provided by grail library and it is used as an input to the TreeMapViewer. Furthermore this class has the responsibility for calculating the similarities of given Java classes.

5.1.6. CloneVisitor Class

CloneVisitor is the core component for normalizing Java code. By default the normalization contains variable and method names but it could be extended to classes, types or loops. As it is visible in Figure 5.6, CloneVisitor has a relationship of ‘bi-directional association’ towards CloneContainer that is a class with implementation of singleton design pattern to support shared data access.

![Figure 5.6 CloneVisitor and Singleton CloneContainer](not for commercial use)

Figure 5.6 briefly explains the implementation logic of CloneVisitor. Even though this figure does not contain all visit methods for AST nodes, the most important three methods for Clone visitor is listed in Figure 5.7.

‘preVisit’ method is called before visit method for each ASTNode; therefore CloneDetector utilizes this method in a way that helps to save existing state of ASTNode as a property on object. After ‘visit’ method and consequent ‘postVisit’ method, software code is normalized.

‘postVisit’ methods checks node type and executes ‘checkClones’ method if it is a desired type of node.

‘checkClones’ has the main responsibility for saving the node to the CloneContainer and setting a color code property for the node. This color information is used at visualization phase later on.
5.1.7. CloneContainer Class

CloneContainer class is a wrapper for a specific map implementation, which holds string representations of the nodes as key and a list of clones for each string representation. String representations are in normalized form, which means type-2 code clones are detectable with this implementation. Clone code has a self-association, which is displayed in Figure 5.8, and it implements singleton design pattern.

```java
private void checkClone(ASTNode node) {
    boolean hit=!CLONES.put(node.toString(), node);
    node.setProperty("colorCode", "+
                        Integer.toHexString(node.toString().hashCode()));
}

public void preVisit(ASTNode node) {
    node.setProperty("code", node.toString());
    node.setProperty("root",
                    ((CompilationUnit)node.getRoot()).getJavaElement().getElementName());
    super.preVisit(node);
}

public void postVisit(ASTNode node) {
    super.postVisit(node);
    if (node instanceof Statement || node instanceof MethodDeclaration ||
        node instanceof TypeDeclaration || node instanceof CompilationUnit) {
        this.checkClone(node);
    }
}
```

According to singleton design pattern, visibility for the constructor of class should be private and object should contain a static instance. On the other hand object access is made with a static ‘getInstance’ method, which utilizes the constructor inside. A simplified singleton implementation can be seen on Figure 5.9.
public class CloneContainer {
    private static CloneContainer singleInstance;
    private CloneContainer() {
    }
    public static CloneContainer getInstance() {
        if (singleInstance == null) {
            singleInstance = new CloneContainer();
        }
        return singleInstance;
    }
}

Figure 5.9 Simplified implementation of singleton class

5.1.8. ModifyClones Interface
This interface is created to represent the operations that modify code clone information in CloneContainer. Figure 5.10 below gives specific information about ModifyClones interface.

public interface ModifyClones {
    public boolean put(String key, ASTNode value);
    public List<ASTNode> remove(Object key);
    public void clear();
}

Figure 5.10 Existing methods of ModifyClones interface

5.1.9. AccessClones Interface
The AccessClones interface is designed to stand for the operations that read the clone information. Figure 5.11 gives information about the methods of this interface.

public interface AccessClones {
    public boolean containsKey(Object key);
    public Set<Entry<String, List<ASTNode>>> entrySet();
    public List<ASTNode> get(Object key);
    public boolean isEmpty();
    public Set<String> keySet();
    public int size();
    public Collection<List<ASTNode>> values();
}

Figure 5.11 AccessClones Interface

5.1.10. ViewerWrapper Class
This class implements DataSetInterface and represents needed data is required by TreeMapViewer. The ViewerWrapper Class is also a singleton, which contains self-
reference and private constructor with a public static getter for the instance. An example implementation of singleton pattern was given on Figure 5.9. As it is obviously seen in Figure 5.1 this class has an association with ExternalTreeMapView class to supply needed data.

5.1.11. DefaultTreeView Class
This class is supplied by grail graph library to represent and enable AST ready for ExternalTreeMapView class. ViewerWrapper uses this class.

5.1.12. DataSetInterface Interface
DataSetInterface is the needed interface for making the clone information ready for ExternalTreeMapView. This interface was defined in the master thesis by (Pan & Wang 2009).

5.1.13. ViewerModifier Interface
ViewerModifier interface consist of a set of methods, which is used to modify clone information for ViewerWrapper. These methods and the interface is available at Figure 5.12.

```
public interface ViewerModifier {
    public void addGraph(GraphInterface graph, double d);
    public void addNode(Object o, DirectedNodeInterface n);
}
```

Figure 5.12 ViewerModifier interface

5.1.14. GraphInterface Interface
GraphInterface is the interface for the expected form of DefaultTreeView. In other words the DefaultTreeView class implements GraphInterface and TreeMapViewer expects an instance of GraphInterface.

5.1.15. ViewPart Abstract Class
This abstract class is supplied by Eclipse PDE and utilized by jClone in order to integrate two new views into Eclipse user interface.

5.1.16. CodeCloneGraphView Class
This class extends ViewPart class and creates a graph viewer, which is used to visualize the raw data in CloneContainer. CodeCloneGraphView class implements a GraphViewer object, which is provided by Eclipse zest framework and utilizes this object to display code clone information.

This class owns a couple of classes that wrap the data in CloneContainer in a way that makes it available for GraphViewer object with respect to Model View Control (MVC) design pattern.

5.1.17. ExternalTreeMapView Class
This class is also extends the ViewPart class and integrates a Swing based TreeMap component into the view. Uses SWT to AWT Bridge in order to achieve a successful integration. This class has an association relationship to access ViewerWrapper to get the GraphInterface object and passes it to the MainStart class, which is the key component for the Swing based tree map implementation. Figure 5.13 Explains the
significant code fragment that enables the Swing component in SWT composite element.

```java
Composite swt = new Composite(folder,
SWT.BORDER_DOT|SWT.EMBEDDED | SWT.NO_BACKGROUND);
swt.setLayout(new GridLayout());
IPath p=ResourcesPlugin.getWorkspace().getRoot().getLocation();
Frame frame = SWT_AWT.new_Frame(swt);
frame.add(new MainStart("",
"name",p.toString()+"/",ViewerWrapper.getInstance().getTrees(
)));
```

*Figure 5.13 Significant code fragment for Swing integration*

5.1.18. MainStart Class
MainStart class is the start point of the external tree map viewer. This is a product of (Pan & Wang 2009) but the author of JClone included many contributions and bug fixes and the source code is modified in respect to the needs of integration procedure with JClone and Eclipse IDE.

5.1.19. JApplet Class
JApplet is one of the top-level containers in Swing UI library.

5.2. Implementation Aspects of JClone
This section is responsible for presenting useful information about the implementation details of JClone. JClone is developed as an Eclipse plug-in and utilized many other components on development phase. Even though some of these components are not critical or required from the runtime viewpoint of system, it is very useful to know about implementation details when it comes to maintenance activities.

5.2.1. Eclipse Plug-in and plugin.xml
JClone is designed to be a plug-in for Eclipse due to requirements and one of the most significant parts of this implementation is the ‘plugin.xml’ file. This file is responsible for defining the utilization of the Eclipse extension points. The ‘plugin.xml’ file of JClone is available below on Figure 5.14.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
  <extension
    point="org.eclipse.ui.popupMenus">
    <objectContribution
      objectClass="org.eclipse.jdt.core.IJavaProject"
      id="org.bahtiyar.jClone.plugin.contribution1">
```
<menu
    label="jClone"
    path="additions"
    id="org.bahtiyar.jClone.plugin.menu1">
    <separator
        name="group1">
    </separator>
</menu>

<action
    label="Analyze"
    class="org.bahtiyar.jclone.plugin.popup.actions.AnalyzeAction"
    menubarPath="org.bahtiyar.jClone.plugin.menu1/group1"
    enablesFor="1"
    id="org.bahtiyar.jClone.plugin.newAction">
</action>

<extension
    point="org.eclipse.ui.views">
    <category
        id="org.bahtiyar.jClone.plugin"
        name="jClone">
    </category>
    <view
        category="org.bahtiyar.jClone.plugin"
        class="org.bahtiyar.jclone.plugin.views.CodeCloneGraphView"
        icon="icons/sample.gif"
        id="org.bahtiyar.jclone.plugin.views.CodeCloneGraphView"
        name="Java Clone Code Graph Viewer">
    </view>
    <view
        category="org.bahtiyar.jClone.plugin"
        class="org.bahtiyar.jclone.plugin.views.ExternalTreeMapView"
        icon="icons/sample.gif"
        id="org.bahtiyar.jclone.plugin.views.ExternalTreeMapView"
        name="External TreeMap">
    </view>
</extension>

<extension
    point="org.eclipse.ui.perspectiveExtensions"
Several extension points of Eclipse are utilized in order to implement jClone. The first one is `org.eclipse.ui.popupMenus`, which is created to implement a pop up menu item named JClone that appears when a user right clicks on the selected project.

Second extension point, `org.eclipse.ui.views`, is used to implement `CodeCloneGraphView` and `ExternalTreeMapView` views.

The third extension point, `org.eclipse.ui.perspectiveExtensions`, is used to integrate created views into Java perspective of Eclipse IDE.

### 5.2.2. Plug-in dependencies

JClone plug-in has several dependencies on some other Eclipse plug-ins as well as external system. Figure 5.15 gives a list of depending Eclipse components.
5.2.3. **Maven and pom.xml**

Maven is a software tool that supports project management and build automation. JClone takes advantage of maven to manage its dependencies and build process. The ‘pom.xml’ file in the project root directory supplies maven configuration. Configuration details can be accessed through Figure A.1 pom.xml of JClone for Maven.

Figure 5.16 displays graphical representation of external dependencies for JClone plug-in, which is a dependency graph created by maven. Maven manages these external dependencies.

JClone is dependent to viewer, which is implemented by (Pan & Wang 2009), and called org.bahtiyar.jCloneviewer here since it consist of just source code and did not have a project form. Author of JClone is modified source code and fixed some bugs and created a Maven Eclipse project with dependency definitions etc. But still all credits about the viewer are given to the authors of original work, (Pan & Wang 2009).
6. Evaluation

JClone plug-in is created to implement an AST based Java code clone detector as an Eclipse plug-in in order to overcome some difficulties for the software engineers such as supplying extra frameworks, putting extra effort on configuration or skipping useful analysis operations while working with external tools. One of the first goals of this master thesis was to create such a tool that makes it easy and useful for the software engineer to analyze the software code in respect to AST based code clone detection. This goal has been reached by created tool, the JClone Eclipse plug-in.

The goal for supporting the latest version of the Java Language Specification (JLS) has been reached by the native support of Eclipse JDT API. Since Eclipse project owns a Java compiler, latest JLS, by the date of this study JLS3, is provided to represent ASTs for selected Java files.

However there is no empirical data about the speed and performance of JClone yet, the analyze operations made by the author has been succeeded in reasonable time frames for real time projects. Especially due to hashing algorithm and multi thread concurrent programming model, normalizing software code and creating code clone information is usually done in short time frames. However, external tools for visualizing code clones may cause some performance problems due to two reasons.

The first reason that might cause some performance issues is the external tree map viewer that is created by (Pan & Wang 2009) creates xml formations of given dataset interface. This is an expensive operation for big java projects and takes a lot of processing time to complete. The user of this system might have latencies during the launch operation of this integrated external tree map viewer.

The second reason is the external tree map viewer is originally implemented in Swing and integrated into Eclipse by the help of SWT to AWT Bridge. This implementation preferred not to cause external dependencies through the code clone analysis, detection and visualization process but on the other hand may cause some inconsistencies. In other words, it is expected to work but should be tested more to be able to say it is bullet proof.

On the other hand CodeCloneGraphView presents useful information on detected software clones. Figure 6.1 Gives a simplified example view for given code clones of type-2 classes.

![Figure 6.1 Type-2 clones which are copied and modified from each other.](image)

This visualization codes colors for each clone and links the clones to each other in that way the user of the system easily distinguish the code fragments that compose a code clone.

The JClone Eclipse plug-in is designed with respect to component based software design principles and contains interfaces to operate over. As a result it would not be a problem to introduce new functionality or modifying existing ones.
As it is shown in Figure A.2 and partially in Figure 6.2, an edition has been made to the menu items, which is used to start clone detection process when it is clicked on. This new menu item is available for one or more selected projects.

![Figure 6.2 Menu item for selected projects to start clone detection](image1)

JClone introduces two new views to visualize the results and they can be activated through the eclipse Show View menu as showed in Figure 6.3.

![Figure 6.3 Additional views to see the clone information](image2)

External TreeMap view is in fact an AWT/Swing based tool, which is integrated to Eclipse environment by the help of SWT to Swing bridge. Figure 6.4 and Figure A.3 shows example output for this view.

Java Clone Code Graph Viewer is the view that renders linked and color encoded clone information as in Figure 6.1. An example of the view can be explicitly seen in Figure A.4.
The list of met criteria can be summarized like:

- An eclipse plug in was created to detect code clones in java projects.
- JClone can analyze large-scale Java projects in reasonable rime frames and can detect code clones by the help of fast syntax tree based method and multi thread design.
- JClone can be integrated to Eclipse IDE to increase usability.
- JClone introduces multiple interfaces that visualize clone information in an easily distinguishable way.
- While complying eclipse plug-in model by nature it is possible to extend the system by introducing new components.
- While presenting the code clone information JClone can ease the maintenance activities and contribute improvement of java projects.

Since all the criteria has been met, it obvious that the goal of implementing a syntax tree based code clone detector prototype has also been reached.
7. Conclusion and Future Work

This study has investigated the importance of code clone detection from the software maintenance point of view and suggested an applicable solution to the problem. Following subsections explains the conclusion and the future work.

7.1. Conclusion

As it is stated in the first chapter the problem was detection of code clones in order to help lowering the cost of software maintenance process while increasing internal code quality.

In this thesis, first of all, the problem domain is studied and the relations between duplicated code fragments and software maintenance process are revealed. According to expected benefits and implied importance of code clone detection, a valid solution is selected among several techniques. As an outcome JClone Eclipse plug-in, AST base Java code clone detector is created with respect to software engineering principles.

As it is already mentioned in the previous chapter JClone Eclipse plug-in fulfills most of the defined requirements and defines a valid method of development and implementation for such a system. It introduces several internal and external tools in order to reach goals from different ways and succeed on its predefined goals.

As we have already discussed at the end of previous chapter all goals are achieved and the criteria met as a result we can imply that the problem is solved.

7.2. Future Work

However the criteria met and the problem is solved in order to use this system in production its documentation should be completed. The user manuals and developer documents should be created before going production.

On the other hand it would be useful to improve user experience by introducing several shortcuts and more views on the clone information. Editing code clones and grouping of code clones can be implemented.

Several test cases should be designed and applied in order to test the complete functionality of the system. Even though it is a complete system test cases should be applied before going in production.

As well as suggested visualization techniques, new approaches and alternative clone visualization methods should be investigated. On the other hand improvements on the suggested system can be introduced in a way that implements tree maps natively in SWT.
8. References


ISO/IEC, 2006. ISO14764 Software Engineering -- Software Lifecycle Processes -- Maintenance, New York, NY :: Institute of Electrical and Electronics Engineers,


A. Appendices

A.1. Pom.xml for Maven

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.bahtiyar</groupId>
  <artifactId>org.bahtiyar.jClone.plugin</artifactId>
  <packaging>eclipse-plugin</packaging>
  <name>org.bahtiyar.jClone.plugin</name>
  <version>0.0.1</version>
  <description>Java Clone Code Detection Eclipse Plugin</description>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-eclipse-plugin</artifactId>
        <configuration>
          <pde>true</pde>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>process-sources</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>${basedir}</outputDirectory>
              <overWriteReleases>false</overWriteReleases>
              <overWriteSnapshots>false</overWriteSnapshots>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```
<overWriteIfNewer>true</overWriteIfNewer>
   </configuration>
   </execution>
   </executions>
   </plugin>

   <plugin>
     <artifactId>maven-clean-plugin</artifactId>
     <configuration>
       <filesets>
       <fileset>
         <directory>${basedir}</directory>
         <includes>
         <include>*.jar</include>
         </includes>
       </fileset>
     </filesets>
   </configuration>
   </plugin>

   <plugin>
     <groupId>org.apache.geronimo.devtools</groupId>
     <artifactId>maven-eclipsepde-plugin</artifactId>
     <version>1.0</version>
   </plugin>

   </plugins>
   </build>
   <dependencies>
     <dependency>
       <groupId>org.bahtiyar</groupId>
       <artifactId>org.bahtiyar.jClone.viewer</artifactId>
       <version>0.0.1-SNAPSHOT</version>
       <scope>runtime</scope>
     </dependency>
   </dependencies>
Figure A.1 pom.xml of JClone for Maven
A.2. Screenshots

Figure A.2 JClone menu item
Figure A.3 Swing TreeMap implementation integrated in Eclipse

Figure A.4 JClone Code Clone Graph View