Tool-assisted Code Splitting for GWT

Alan Hicks

A thesis submitted in part fulfilment of the

MSc (hons) in Advanced Software Engineering

Supervisor: Dr. Joseph Kiniry

UCD School of Computer Science and Informatics

College of Engineering Mathematical and Physical Sciences

University College Dublin

May 25, 2010

Table of Contents

Glossary of Acronyms

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

AST Abstract Syntax Tree

CSS Cascading Style Sheets

DHTML Dynamic Hyper Text Markup Language

DOM Document Object Model

GUI Graphical User Interface

GWT Google Web Toolkit

HTML Hyper Text Markup Language

IDE Integrated Development Environment

JSR Java Specification Requests

MSDN Microsoft Development Network

SDK Software Development Kit

SOYC Story of Your Compile

XML Extensible Markup Language

YUI Yahoo! User Interface Library

Abstract

This research focuses on applying automated techniques for code splitting in Google Web Toolkit (GWT) [1] applications. Code splitting is one of the advanced features in the 2.0 release of GWT and aims to speed up the compiled application by separating the code into individual download segments. Segments are downloaded only when required by the end user. Code splitting is a useful feature because as applications get more complex and have a bigger footprint, it is necessary to find ways to speed up the response time for the user.

GWT allows developers to build JavaScript applications using the Java language. The toolkit converts Java code into JavaScript. GWT abstracts from the low-level browser quirks of JavaScript development. The comprehensive set of features provided by GWT represents a viable alternative to JavaScript programming.

Over the last several years GWT has become increasingly popular. The resulting tools, which have been developed by the GWT community, are proof of this popularity. One example of such a tool is the GWT Eclipse Plugin which enables simple compilation and deployment of GWT-based applications within the Eclipse environment.

Despite advances in GWT, such as compiler optimisations, a potential bottleneck at application start up is still relatively common. The cause of this bottleneck is the need to download the entire JavaScript application. This methodology represents a poor use of resources, as the end user may not require the full feature set of the application. The GWT team have introduced, in an effort to alleviate this bottleneck, a technique called code splitting. Code splitting gives developers an API to support splitting their application up into chunks that are loaded on demand, as triggered by subsequent events. The use of code splitting separates the download impact over the entire user's session and thus is intended to reduce the number of download bottlenecks. An additional benefit is that the user will download only the code they require for the features of the application they use.

This research extends the code splitting technique by providing a mechanism that helps objectively identify code segments for code splitting. This analysis is based upon static code analysis, feature annotations and/or recorded user usage. This assists the user as their required downloads will be staggered according to the architecture of the application and the application will feel more like a responsive desktop application rather than a web application.

An Eclipse plugin is utilised, the intention of which is to implement this advancement of code splitting techniques. The plugin analyses the application source code and quantitative data and displays graphical information to the developer. The plugin thus highlights to the developer where code splitting should take place.

To demonstrate the concepts proposed in this research a comparison is made between a code-splitting sample application produced by the GWT team and the same application split using the techniques developed in this research. The research illustrates that it is not only possible to automatically apply code splitting, but that the concepts proposed produce positive results, as can be seen in a reduction in the initial JavaScript code load of the sample application by 65%.

Acknowledgments

I would like to thank my dissertation supervisor, Dr. Joseph Kiniry, who has been enthusiastic, encouraging and unfailingly positive during the duration of this project.

Julien Charles and Fintan Fairmichael for their pointers on Eclipse plugin development.

My work colleagues at CarTrawler, especially Bobby, Gillian and Mike for their support over the last two years.

Andrew and Clare for the detailed proof reading and writing advice.

My parents and family for their constant support and encouragement.

And finally, and especially, Clare (*again*) for all her support and understanding of all the late nights and long weekends spent researching.

Chapter 1: **Introduction**

The focus of this project is to examine code splitting, one of the advanced features in the 2.0 release of Google Web Toolkit (GWT) [1]. Code splitting is a technique that aims to speed up the compiled application by separating the code into separate download segments. These segments are downloaded only when required by the end user. Code splitting is a useful feature because as applications get more complex and have a bigger footprint, it is necessary to find ways to speed up the response time for the user.

"GWT is a development toolkit for building and optimising complex browser-based applications. Its goal is to enable productive development of high-performance web applications without the developer having to be an expert in browser quirks, XMLHttpRequest, and JavaScript."[1]

GWT, since its 1.0 release in 2006, has gained a wide community base. It was open sourced during its 1.0 release which was key to gaining the support and interest of the community. The adoption of the toolkit internally in Google has spread confidence in the project for use by others in the construction of complex web applications. Google has most recently used GWT to construct its Wave platform client [2, 3].

At its core, GWT enables developers to focus on the business problems of an application, rather than constantly battling with the numerous browser quirks that plague a JavaScript developer. An application written in Java is converted by GWT into JavaScript that is portable across all major web browsers.

Code Splitting Example

Taking the example of a simple web mail application built in JavaScript, the web mail application allows the user to read and compose email. Traditionally, in a JavaScript application the entire application is downloaded by the user before they can either read or write an email. This is acceptable if the user will read and compose an email. If the user only wishes to read an email they are still required to download the code that enables them to compose an email. This is inefficient both in terms of a user's time and server bandwidth. With code splitting, the application is written so that the read and compose features are separated and only downloaded as required. So, if a user reads an email, only the code required to read an email is downloaded. This technique provides a much quicker response and a more desktop-like feel to the web application experience for the user. As an aside, it also reduces the server bandwidth usage, which benefits both the end user and the provider of the code.

There are, however, complex issues in applying this technique, even in the more simple examples such as: where and how to split; shared code handling and which split to download.

The complexities mentioned are some of the reasons why currently all code splits are developer-driven. It is up to the developer to decide and implement the splits in the code. GWT provides some code analysis tools to aid a developer, these tools are only available after the code has been compiled. The tools provide information based on the compile process and do not make recommendations on how and where to split the code.

This project assists developers in providing automated code splits. This is achieved using a combination of three algorithms: static analysis; feature annotations and user usage patterns.

Research like this is difficult to test and verify. In the GWT 2.0 release Google re-factored one of the example applications to take advantage of code splitting. To gather results on the success of the research carried out, the original unmodified Google application is split using the algorithms developed. This split application is then compared to the application manually split by Google.

It is outside the scope of this project to explain the complex underlying compiler details of how code splitting is achieved at the compiler level and, subsequently, at the end user level with regards the timing of the downloaded splits. The primary focus of this research is to look at ways to assist in the splitting of a given codebase.

The research in this project looks at a combination of splitting algorithms. These algorithms, static analysis; feature annotation and user usage are detailed later in Chapter 3.

It is possible to use any combination of the algorithms to apply automatic code splitting. This allows automated splitting to be flexible depending on the data available. This means that a codebase without feature annotations or user usage data can take advantage of automated code splitting. However, the more analysis information available, the more accurate the suggested splits are likely to be.

The project compiles complex matrices and a method for displaying all of this information to the developer is required. It is necessary to express clearly to the developer, reasons for split suggestions.

The GUI for the analysis tool is in the form of an Eclipse plugin. All of the analysis is carried by a standalone program which reports back to the Eclipse plugin, this means that porting the tool to another IDE/system is a simple task.

1.0.1 Overview of Research Components

To gather all of the requirements necessary for this research, a set of components was required to be developed. An overview of how these components work together is provided in Figure 1.1. Later in chapters 4 and 5 these components are detailed. The base of the system is CodeInspect, this is responsible for parsing, analysing and splitting the GWT project source. GWTTrack is a module developed to allow user usage data logging to be added to a GWT project. An Eclipse plugin acts as the control centre for these systems. The diagram illustrates the flow of information between the components.
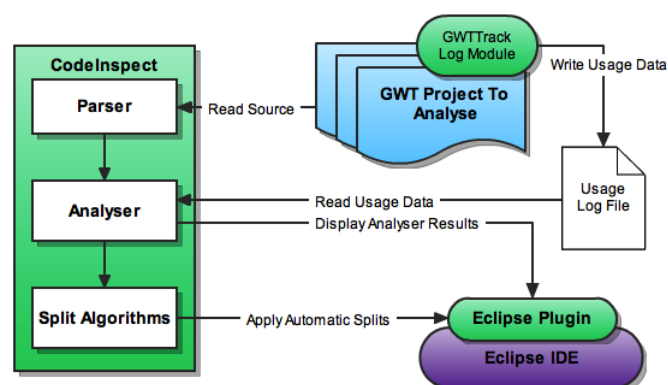

Figure 1.1: Overview of System

Chapter 2: **Background Research**

_____

This chapter presents the different technologies researched during the course of this study, included with each is a brief description. Also examined are the reasons why code splitting is important when building complex web applications. The theory behind code splitting techniques is then explored. The chapter concludes with the a look at similar research relating to code splitting.

2.1 Web 2.0 Applications

There is an increased focus on web applications rather than static websites in internet usage. The internet is dominated by access to complex applications such as email clients and office applications such as document processors and spreadsheets. These applications are known as Web 2.0 and can be characterised by the technologies that enable them. A Web 2.0 application is almost always constructed using advanced JavaScript, DHTML and CSS technologies. Collectively these technologies are known as AJAX (Asynchronous JavaScript and XML).

AJAX allows applications to run on the internet and be as responsive as traditional desktop applications. Examples of some high profile Web 2.0 applications are GMail, Google Docs and Live Maps. While the listed applications can be called pure AJAX applications, AJAX finds its way into nearly every new web site, even if the web site cannot be classed as an application.

It is clear that the advancement in AJAX and web technologies is driving application development onto the internet.

The success of these applications relies on the responsiveness of the code that powers them. Unfortunately, as these applications increase their feature set, the code required is also increased. As the code must run on the client web browser, there is normally a large initial download footprint. This initial download contains all of the application code. In today's AJAX applications the largest bottleneck for the user is the application code download.

2.2 Google Web Toolkit

JavaScript is a very powerful language designed to run within a client web browser. JavaScript combined with DHTML and CSS forms the AJAX set of technologies. This set of technologies enable the creation of very powerful web applications. These are a complex set of technologies to master and because the clients(web browsers) for these technologies are created and maintained by different vendors it is not sufficient to simply write the code. Each web browser and browser version has a varying level of support for AJAX and can interpret the code differently. These idiosyncrasies are known as browser quirks. A typical AJAX developer will spend a considerable amount of time browser testing and kludging code to work across

all browsers they wish to support.

To alleviate the issues surrounding browser quirks a number of frameworks have been developed. A list of some common JavaScript based frameworks are set out below:

- Yahoo! User Interface Library (YUI)
  YUI is a framework from Yahoo! that provides prebuilt JavaScript and CSS components for reuse in web applications. The YUI Library is widely used by many web applications not only those applications built by Yahoo! [4].

- Prototype
  Prototype was one of the first JavaScript frameworks to emerge that provides a toolkit for building AJAX enabled applications. Prototype focuses on a class-driven development model [5].

- jQuery
  jQuery is a JavaScript library that provides both a low-level library and a User Interface library of production ready components. jQuery is becoming increasing popular because of its simple syntax and full featured pre-built components [6].

Another approach, which has been developed to alleviate browser quirks and speed AJAX development, was released by Google in 2006. Google's approach is to allow a developer to write code in a structured programming language (Java) and it then provides a compiler that converts the Java code into JavaScript. Built into the GWT compiler is a set of rules for each web browser. The GWT compiler then creates a different version of the JavaScript for each major web browser. This process virtually eliminates the browser quirks issues.

GWT has many other advantages over writing raw JavaScript code aside from the issue of browser quirks. GWT converts Java code, as a result, the developer has the option to use any Java IDE. As Java is a very popular and accessible language there are a lot more development tools that can be leveraged. GWT also allows a developer to debug an application and step through the code. Java is a type-safe language; it enforces good practice when writing code. This translates well as type-safe checking can be carried out by the GWT compiler which will detect errors at compile time rather than at runtime as would be the case with a raw JavaScript development cycle.

2.2.1   Dead-For-Now

The GWT compiler applies a dead-for-now (DFN) approach when compiling Java code into JavaScript. DFN will determine and remove any Java code which is not called. This is valuable as it allows the Java code to be kept clean and maintainable while not compromising on the end performance of the JavaScript.

2.2.2   GWT Compile Report

The Compile Report [B.1] gives the developer feedback on the GWT compile process and is a valuable tool as it will breakdown for the developer the different dependancies required in a particular fragment of code. The results of the Compile Report are output in XML format. A visual report view can then be applied to the XML which will generate a simple set of HTML pages that allows for easier browsing of the results.

The Compile Report is particularly useful when code splitting is added to the application. Due to the complexities of code splitting it is difficult to know what code is included in a split. The compile report gives a detailed breakdown of a spilt including the size of the split fragment and reasons for the size. This information can then be used by the developer to refactor their code to make the splits more efficient.

The beta version of the Compile Report was known as Story of Your Compile (SOYC). Since the beta and the official release of GWT 2.0, apart from the name, only minor GUI improvements have been made.

2.3   Code Splitting

AJAX applications are growing in complexity and functionality. This growth is unlikely to reverse as the trend in moving applications to the internet continues. The result of this growth in application size causes a problem for the users of such applications. As the applications grow, the size of the code required grows, which increases the client download time. This provides a very poor experience for the user. The increasing trend of the usage of smart-phone devices on mobile phone networks is also now a primary concern for application vendors. This is because the download speeds are significantly slower on mobile phone networks.

As a result of the download bottleneck associated with AJAX applications a technique was introduced in GWT to allow a developer to insert code splits. The GWT compiler will, based on the split calls, fragment the code to produce separate JavaScript downloads. The fragments will be downloaded only when required by the client [7].

While the API for code splitting is simple, the process of identifying what and how to split is more complex. The current approach to code splitting is somewhat of a trial and error procedure. A split point can be inserted anywhere in the program code but when the GWT Compiler calculates the split fragment it will include all associated code in the split. This is the only way that a split can function but it can lead to inefficient splits that are only discovered after the GWT Compile Report has been reviewed.

For example, if the code shown in Listing 2.1 was compiled; the split defined would pull in the dependancies required to run the code contained in the split. So, in this case, the split would contain the code for the java.util.Date package which could lead to a larger than expected split.

Listing 2.1: Code Splitting Dependancy Example

```java
// Sample domain class
import java.util.Date;
class MyDate {
  final private static Date now = new Date();
  public static Date getNow() {
    return this.now;
  }
}

// Sample usage of domain class
class MyApp {
  ...
  // Split Point
  GWT.runAsync(new RunAsyncCallback() {
    public void onSuccess() {
      MyDate.getNow();
```

```
        ...
    }
    ...
  });
  ...
}
```

2.4   Eclipse IDE

The Eclipse IDE [8] is an open source multi purpose development environment. The IDE is primarily used by the Java community. The open source nature of Eclipse and the extensive plugin architecture it supports makes the perfect environment for plugin development. It is clear that the success of the proposed solutions relies on getting the information to the developer in a simple and responsive manner. The Eclipse environment allows for this.

An Eclipse plugin is used as the GUI for this project. The GUI is responsible for providing feedback to the developer as to where splits should be added to the code.

2.5   Java Abstract Syntax Tree

An abstract syntax tree (AST) is a semantic representation of a programs code. Each program structure is represented by a difference node type. This allows for programatic parsing of source code.

The Java Compiler API produces an AST representation of the Java code. The AST provides node based access to the different elements of the Java compilation unit. Each node in the tree is assigned a Kind depending on the the element it is representing. For example method invocations are represented by nodes of type com.sun.tools.javac.tree.JCTree.JCMethodInvocation.

2.6   Java Compiler API

The Java Compiler API is an API released in the Java SDK version 6 as defined in the JSR 199 [9]. This API allows for runtime compilation of Java code. The API is very powerful as it allows running Java programs to compile Java code dynamically at runtime. In this research the API is used not to compile and run Java code but to compile and analyse Java code. During the compile process the Java Compiler API creates an AST of the code being compiled. This is useful because the AST can then be analysed. The compilation ensures that what is being analysed is valid as per the Java compiler rules, which means time is not wasted analysing broken code. The Java Compiler API allows us to create a portable parser that runs on any system using Java SDK 6. Other options for constructing an AST would have been to modify the Javac compiler source, but this would have not enabled a portable parser to be built. As the Java Compiler API is a new feature of the Java SDK and is not widely used, documentation is rare. Heavy use was made of the Javadoc pages [10].

2.7 Related Work

Microsoft MSDN DevLabs have carried out similar research in the form of Doloto "an optimization tool for Web 2.0 applications". Doloto is a tool that analyses user usage patterns for a given JavaScript application. Based on the usage patterns analysed, code splits are recommended. Doloto analyses a web application at the JavaScript function level. Doloto generates a set of code split recommendations for the usage patterns, the result is known as an access profile [11].

The Doloto process consists of two phases, training and rewriting. The training phase is concerned with recording the usage of the application at a JavaScript function level. This is achieved by capturing requests with the usage of a local proxy that instruments the JavaScript calls. Typically the training phase is carried out on a single computer with a predefined workload. The workload is manually determined by the user and should represent the most common path though the application.

Based on the information gathered during the training phase the rewriting generates a new set of JavaScript files. These files contain a code spitted version of the application tuned to the usage patterns analysed. Using the generated files the application will run faster, as only the code required will be downloaded. The generated files contain the optimised split code. The mechanism that Doloto uses to enable code splitting is stub based. Doloto generates empty stubs for the JavaScript methods and downloads the required code lazily as required [11].

"The ultimate goal of Doloto is to automate the process of optimal code decomposition. The decomposition process automatically handles language issues such as closures and scoping, as a result developers no longer have to manually maintain the decomposed version of the application as it changes." [11]

The research carried out in this project has the same goal as the Doloto project; to speed up AJAX applications by limiting the code download to only the required fragments. The approach of this research differs. The Doloto project does not attempt to statically analyse the application structures to preform code splitting, it relies solely on the information gathered during the training phase to determine code splits. The training phase is only carried out on fixed workloads. This can lead to biased results.

This research project explorers code splitting using static analysis; feature meta-data and user usage data. The user usage data utilised in this research is equivalent to the training stage of the Doloto project, however usage data can be gathered while the application is live, this has the benefit of analysing both common and uncommon usage patterns of an application. The Doloto project requires a training phase to calculate code splits, training is something that can only be applied to a complete application. Doloto does not attempt to calculate code splitting until the application has been completed. Whereas, the approach in this project is to provide the developer with code splitting feedback during development via static analysis and also when the application is complete. It is proposed that this provides more informative and thus better code splits.

Chapter 3: **High Level Methodology**

The goal of the research is to develop a set of tools bundled as an Eclipse plugin that will aid developers in splitting GWT applications in an efficient and meaningful way. The basic principle of the research is to devise a number of code splitting algorithms that can be applied to a given code base. The three algorithms proposed;

- Static Analysis

- Feature Annotation

- User Usage

The code splitting algorithms are applied at a method level. The static analysis algorithm relies solely on the constructs within the codebase. The feature annotation algorithm takes into account additional code comment information added by developers throughout the code. The user usage algorithm applies a log file of user application usage data.

To apply any of the developed algorithms to the code base a parser has been developed to analyse the code structures and associated metadata, if any. For the user usage data a GWT module was developed to allow for usage logging. These two systems are referred to as CodeInspect and GWTTrack respectively. The details of their construction are discussed in Chapter 4.

3.1 Overview of GWT Compiler Splitting Code

Code splitting in GWT is a complex operation for the compiler. The compiler breaks the code into separate chunks based on the splits applied to the code. Figure 3.1 provides a graphical view of how the compiler separates the code. A number of files are created, cache.html, 1.cache.js, 2.cache.js, 3.cache.js and leftovers.js. The cache.html is the initial loading file for the GWT app. A file is created for each split point numbered 1.cache.js to N.cache.js and another file called leftovers.js. The compiler, where possible, adds the code for each split to the X.cache.js file, however, if there is shared code between two or more splits then it is added to the leftovers.js file which is downloaded when the first split point is reached.
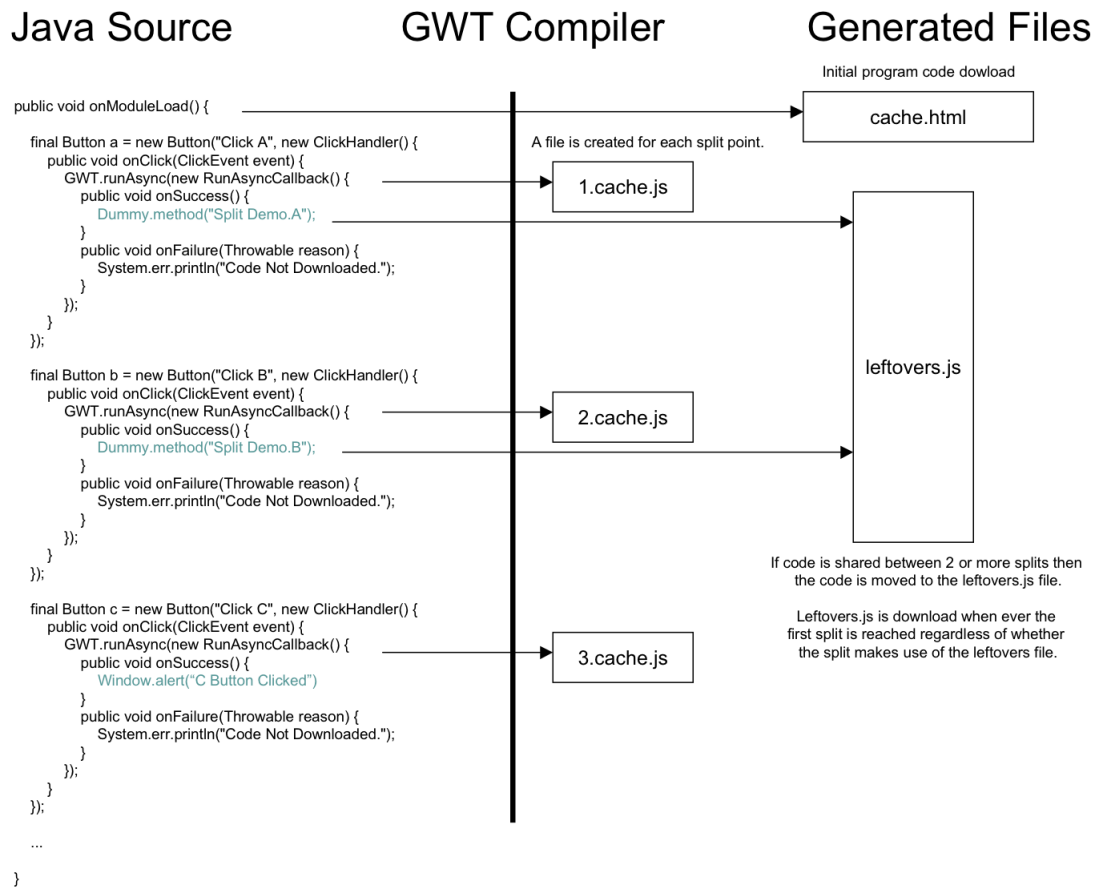
Figure 3.1: Overview of how the GWT Complier handles code splitting

3.2 Code Splitting Algorithms

Static Analysis

The codebase of the application is statically analysed to determine the code structure and to attempt to isolate the various pieces that can be separated. The AST of the code is analysed and code relationships are built up. Based on these relationships, calculations are carried out to construct logical splits which are possible. With static analysis the applications context is not clear, therefore it is impossible to recommend splits based on functionalities defined by the program code. During this phase of analysis, the application can return possible splits based only on the code relationships, such as separate code based on inherits.

The principle logic of the algorithm can be seen below:

$$i \forall Class$$
$$j \forall Methods$$
$$if(NumberCalls_{ij} = 1 \&\& MethodSize_{ij} > 0)$$
$$\quad Split_{ij} = true$$
$$end$$

Java implementation of the static analysis algorithm shown in Listing 3.1.

Listing 3.1: CodeInspectResult.java: Static Analysis Algorithm

```java
public final List<DerivedSplit> staticAnalysisBasicSplits() {

  List<DerivedSplit> splits = new ArrayList<DerivedSplit>();

  for (Entry<String, GSClass> entry : gsClasses.entrySet()) {
    GSClass c = entry.getValue();
    for (GSMethod m : c.getMethods()) {
      if (!m.isDeadCode()) {
        if (m.calls.size() == 1 && !m.codeGenerated && m.totalSize() > 0) {
          DerivedSplit split = new DerivedSplit();
          Marker marker = m.callMarkers.get(m.name);
          split.marker = marker;
          splits.add(split);
        }
      }
    }
  }
  return splits;
}
```

Feature Annotation

The analysis tool constructed for the project allows the developer to add some simple meta-data which can identify a method as being part of a feature. The meaning of a feature in this context is a logical grouping of code. The code alone cannot provide the context of the application, as a result metadata can be added to allow the analysis tool some insight into the business view of the application. Feature annotations can be added to any method in the program by adding an annotated comment to the code.

Listing 3.2: Feature Annotation Example

```java
/** @feature ReadEmail */
public void openEmail() {
  ...
}


/** @feature ReadEmail, WriteEmail */
public void commonMethod() {
  ...
}
```

The feature annotation accepts multiple feature names. This allows a particular method to be part of two or more features. Feature names are case sensitive. It is not required that all methods of a feature are tagged. Based on the feature tag, the analysis tool will scan the code and associate all linking code to that feature. This means that it is possible to use the feature tag at a high level in the code, ideally at the point of the initial call. Based on the initial method call, the analysis tool finds all associated code and recommends a split.

User Usage

The most accurate method of splitting a program is based on the actual usage of the application. Gathering user usage data is key to working out how a user interacts with the various sections of the code. The analysis tool developed takes user usage feedback and maps

it onto the code to calculate useful splits based on the usage. It is suggested that this type of analysis provides the most accurate splits of an application.

To track a users usage, comment are added to the applications code which record the application usage to a file on the server. Live commenting is possible when running in an IDE environment. Usage recording is added to the code by making calls to a custom built GWT Module.

Listing 3.3: User Usage Logging Example

```
/** @feature ReadEmail */
public void openEmail() {
  GWTTrack.log("openEmail");
  ...
}
```

3.3   User Usage Data

The user usage data log provides a detailed account of a user's actions for a web application. Logging statements are included in the GWT source code. These log statements at runtime record the users actions performed. Actions are recorded with a timestamp, client session ID and the string message as defined in the log call. Due to client-side scripting restrictions, to log the information it is sent to a server-side process, which records the data in a central file. JavaScript, because it runs on the client web browser, does not allow file writing. Having a central storage point also has advantages because it records in one place all user actions of the application.

The data file is loaded and parsed at a later date to build up a picture of common usage patterns for the application. Taking the mail application example, it can be established from the usage log that 90% of users who read 10 emails end up composing an email. Based on this data, not only does the information provide details on the parts of the application to split, but also information on which splits to preload. It is possible to build into the application, a rule, that if a user reads 10 emails, the compose email code is prefetched. This type of pre-emptive code loading impacts positively on the consumer satisfaction of an application.

Listing 3.4: Pre-emptive Code Loading Example

```
public void composeEmail(final int emailReadCount {
  if (emailReadCount >= 10) {
  GWT.runAsync(new RunAsyncCallback() {
    public void onSuccess() {
      ...
    }
    ...
  });
}
```

Chapter 4: **Detailed Design and Implementation**

To apply the algorithms that have been developed, the source code of the program must be broken down and analysed. To parse the code, the new Java Compiler API that ships with Java SDK 6 is used. Other options for this would be to modify the source code of the Javac compiler but for the needs of this project that was not a viable option.

The parser written, scans the code using various visitors and registers its findings with a static set of list objects. Each method and method invocation is broken down and the call stack is calculated. The parser built has been dubbed CodeInspect.

4.1 CodeInspect

4.1.1 Parser and Analyser

The CodeInspect parser is the main body of code written for this project. It takes as a parameter the project source directory, parses the code and builds up a model of the relationships in the code. While the parser focuses on GWT projects it can be used to analyse any Java project, the only requirement, is that the jar files required at runtime for the project are stored somewhere in the project root directory so they can be accessed by the parser.

CodeInspect is built on top of the Java Compiler API which itself utilises the visitor pattern. This is useful as new visitors can be registered and enacted on the code. The controlling class in the project is the CodeInspector.java. This class wraps the complexities of the Java Compiler API and sets out the calling sequence for the the registered visitors.

CodeInspector has one public method call as shown in Listing 4.1.

Listing 4.1: AnalyzeCode Method Signature

```
/**
 * @param directoryPath       Source code file path
 * @param visitors            List of AST visitors
 * @return CodeInspectResult  Result of analysis
 */
public static CodeInspectResult analyzeCode(final String directoryPath,
        final List<CodeVisitorBase> visitors)
```

The parser contains four core visitors that look at the main structure of the program; class declarations/invocations and method declarations/invocations. The order in which these are run is important and so the calling of these is controlled by the analyzeCode method as shown in Listing 4.2. The analyzeCode method allows a list of ad-hoc visitors to be passed at runtime. These visitors are applied after the core visitors. For this project, two visitors are passed; FeatureCodeVisitor and TrackCodeVisitor. These visitors analyse feature comments and the GWTTrack calls and are discussed later.
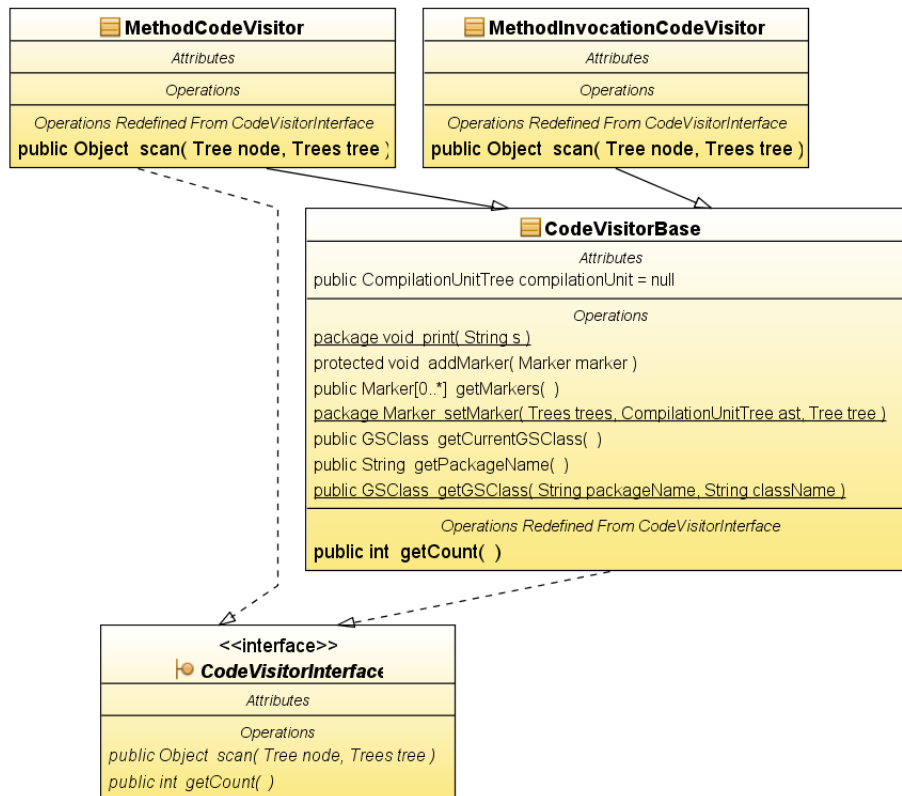
Listing 4.2: AnalyzeCode Visitor Call Order

Figure 4.1: Subsection of CodeInspect Architecture, Focusing on Visitor Objects

```java
// Workout class/method structure phase 1 : delcarations
for (CompilationUnitTree ast : asts) {
  ClassCodeVisitor classVisitor = new ClassCodeVisitor();
  classVisitor.compilationUnit = ast;
  classVisitor.scan(ast, trees);

  MethodCodeVisitor methodVisitor = new MethodCodeVisitor();
  methodVisitor.compilationUnit = ast;
  methodVisitor.scan(ast, trees);
}

// Workout class/method structure phase 2 : invocations
for (CompilationUnitTree ast : asts) {
  MethodInvocationCodeVisitor invoVisitor = new MethodInvocationCodeVisitor();
  invoVisitor.compilationUnit = ast;
  invoVisitor.scan(ast, trees);

  ClassInitCodeVisitor classInitVisitor = new ClassInitCodeVisitor();
  classInitVisitor.compilationUnit = ast;
  classInitVisitor.scan(ast, trees);
}

// Run users Visitors
for (CompilationUnitTree ast : asts) {
  for (CodeVisitorBase v : visitors) {
    v.compilationUnit = ast;
    v.scan(ast, trees);
  }
}
```

The code analysis produces a set of GSClasses and GSMethods. These are representations of the structures from the Java code being analysed. Stored in these objects are the details of a single class/method and their call hierarchy. This information is used when applying the code splitting algorithms.

4.1.2 Applying Derived Splits

Once the CodeInspect parser has completed, the code is split based on the code splitting algorithms. The algorithms calculate a set of splits and these are applied to the source code. A utility finds the line where the split is to be inserted and surrounds with the standard GWT split declaration code as shown in Listing 4.3.

Listing 4.3: Template Code Inserted To Apply Split

```
// SPLIT_HEADER
/** start: gwt-splitter defined */
GWT.runAsync(new RunAsyncCallback() {
  public void onSuccess() {

    ... Line To Split ...

//SPLIT_FOOTER
  }

  public void onFailure(Throwable reason) {
    // TODO: Error Handling
  }
});
/** end: gwt-splitter defined */
```

4.2 Feature Annotating

Feature annotating functions by associating metadata with a section of code. A feature tag is placed above a method call. This allows a developer to associate some context to the program code. Once a method has been declared as part of a feature, all methods associated with that method will be grouped as being part of that feature. This CodeInspect parser has a visitor (FeatureVisitor) that will find and group these associations. A single method can be defined as being used by 1..n features.

The FeatureVisitor looks for the feature tags and links them with method declared below the tag. The structure of the feature tag is shown in Listing 4.4.

Listing 4.4: Feature Annotation Examples

```
/** @feature ReadEmail */
public void openEmail() {
  ...
}


/** @feature ReadEmail, WriteEmail */
public void commonMethod() {
  ...
}
```

The Javac compiler removes all comments aside from Javadoc ones, the feature tags must use /** */ and be placed just before the method declaration.

Listing 4.5: FeatureVisitor.java: Feature Visitor Code Block

```java
public final Object scan(final Tree node, final Trees tree) {
  ...
  JCCompilationUnit unit = (JCCompilationUnit) node;

  // Load comments for current class
  Map<JCTree, String> comments = unit.docComments;

  for (Entry<JCTree, String> entry : comments.entrySet()) {
    String[] names = Feature.featureName(entry.getValue());
    if (names != null) {
      // One method can be used in multiple features
      for (String name : names) {
        name = name.trim();
        Feature feature = Feature.findIn(features, name);

        if (feature == null) { // add to features
          feature = new Feature(name);
          features.put(name, feature);
        }

        // Find method associated with Feature
        JCTree nodeBlock = entry.getKey();
        if (nodeBlock instanceof JCMethodDecl) {
          // Find method and add to Feature
          JCMethodDecl decl = (JCMethodDecl) nodeBlock;
          ...
          // Record method
          feature.addMethod(method);
        }
      }
    }
  }
  ...
}
```

4.3   GWTTrack

To enable user usage data to be tracked and logged, a way of logging the GWT application is required. To achieve this, a GWT module was developed which can be plugged in and utilised by a GWT application. A GWT module is a specially packaged GWT project that allows for reuse by another GWT application, in a similar way to the manner in which Java functionality can be separated into jar files and included in other Java projects.

The logging module allows a GWT developer to insert simple API calls that accept a string as a parameter. The location of the track call is important as it is used by the CodeInspect TrackVisitor to associate the logging call with the method. This allows the CodeInspect Parser to map the log calls to the methods they are called in. When the log call placement data is compared to the data in the user usage log file, a picture of user's activity at the method level is constructed. Using this information, details on splits can be calculated along with pre-emptive split fragment loading.
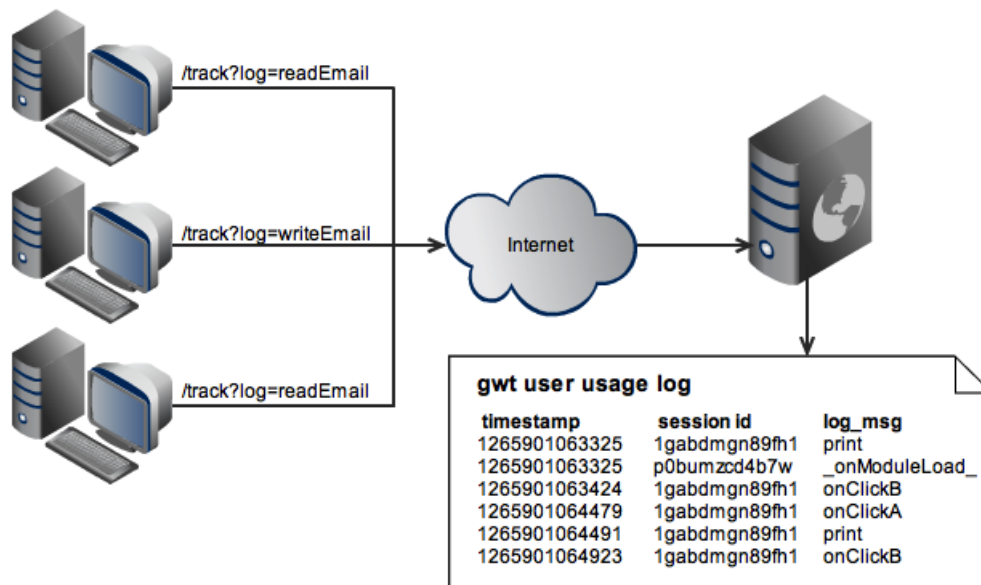
Figure 4.2: Flow of Logging Information From a Client PC to the User Usage Log

4.4 Eclipse Plugin

An Eclipse plugin has been developed to provide a code splitting control centre for the
developer. The plugin provides GUI access to the CodeInspect tool. The plugin also provides
graphical feedback on the status of the codebase analysed. Two new Eclipse views were
developed; Code Charts [C.3] and GWT Log Reader [C.4]. These views are applicable to
Java projects only.

The Code Charts view provides a pie chart view of the analysed source code. The view
creates a chart based on the size of the methods and classes. The initial view shows the
package level chart with the package containing the most code taking up the largest section
of the pie chart. The developer can drill down to the method level of a particular class. This
view is useful because it provides the developer with an efficient way to see where the large
sections of code are, hence code splitting can be focused at this point.

The GWT Log Reader view is a view that polls the local instance of the user usage data file
and provides runtime analysis of the file data. This is useful for the developer when running
the user usage log in local development mode. The view provides a raw view of the file and
some graphical charts. The charts help highlight common usage patterns visually.

The Eclipse plugin, after the analysis is completed, provides a list of GWT splits in an
application. This list is available in the standard bookmark view [C.1] in Eclipse. Double
clicking a split will load the corresponding class file and highlight the split point.

Chapter 5: **Testing and Evaluation**

Testing and evaluation of the research carried out in this project is difficult to quantify with regard to definitive answers such as correct and incorrect. The overall goal of the research is to aid a GWT developer in splitting a given codebase. A split point in an application can never be 100% correct or incorrect.

A decision was made, initially, that the testing and evaluation of the proposed automated code splitting algorithms would be determined by their success in splitting the codebase of the GWT Showcase application [12]. GWT since version 1 has shipped with a sample application called Showcase. This sample application is a showcase of the common features and GUI elements contained within the GWT API. The application is not only used to highlight the features of GWT but also as a "best practice" application for developers to review and learn from. It is built and maintained by the committers and experts from the GWT community. With every new release of GWT the Showcase application is updated with the newly released features. When GWT version 2.0 was released the Showcase application was updated to use code splitting. The code splitting was manually applied by the GWT team.

The Showcase versions compared are a) the code split version released in the GWT 2.0 official download (GWT Manual Split) and b) the version of the Showcase before the GWT team applied splitting (GWT Auto Split). This version is split using the algorithms developed in this research.

5.1   Scope of Testing

A number of components have been built during the course of this research that must be tested. There are three code components, CodeInspect, GWTTrack and an Eclipse plugin. Also, the static analysis algorithm is required to be tested to ascertain its effectiveness.

The code components of the project have been developed and tested in an agile manner. These components have been built to a prototype level of completeness, more work would be required to have them production ready.

5.2   Static Analysis Algorithm

Testing of code splitting is ad-hoc as it is the responsiveness of the application that is tested. Some basic units of testing can be determined. To automate testing of the automatically split application the Selenium [13] test suite is used. This allows for automation of web application interaction and flows. Using Selenium not only speeds up the test process but also provides a base of testing that will be consistent across both applications.

	GWT Manual Split	GWT Auto Split
Number of Splits	37	26
Total Code size	379554	352897
Initial Download Size	193192	66333
Leftovers Size	41808	47096

Table 5.1: Showcase Application Comparisons

5.2.1 Selenium Testing

To test the correctness of the splits and ensure that the application functions post-splitting a Selenium test script is setup on the pre-split application. The test script simulates a user clicking on every link in the Showcase application. The automatically split application successfully passed the Selenium test script. This confirms that the functionality of the application has remained the same between the pre and post-split versions.

5.2.2 Manually Vs Automatically Split Application

The most obvious result, when the two applications are compared, is that the manually split application has 37 splits where the automatically split application has only 26 [Table 5.1]. The splits found by the algorithm do not match with those of the manually split application. This however does not mean that the algorithm splits are incorrect, it merely means they are different. When loading both applications and browsing them there is no obvious speed difference for the user. More extensive testing would be required of the applications, ideally using a large base of users and over a variety of different network speeds.

The leftover fragment of both applications is of a similar size. This fragment contains the shared code required by two or more splits and is downloaded when the first spilt point is reached.

One very interesting result is the that the size of the initial download fragment in the automatically split application is 65% less than that of the manually split application. A key goal in speeding up AJAX applications is reducing the initial download code size of the application. While this is a vast improvement in the initial download fragment size it may come at a cost later in application depending on the flow the user takes through the application. Like all application split testing this only become apparent when widespread user testing of the application is carried out.

Table 5.2 and 5.3 detail the exact splits implemented in the applications. Split 9 in the automatically split application accounts for 60% of the application size. As only the static analysis algorithm has been applied, this is acceptable. With the other algorithms applied this split point would be broken down into smaller fragments.

No.	Class Containing Split	Split Size
1	@pkg.sample.showcase.client.content.i18n.CwConstantsExample.asyncOnInitialize	3952 Bytes (1.04%)
2	@pkg.sample.showcase.client.content.i18n.CwConstantsWithLookupExample.asyncOnInitialize	2509 Bytes (0.66%)
3	@pkg.sample.showcase.client.content.i18n.CwDateTimeFormat.asyncOnInitialize	4276 Bytes (1.13%)
4	@pkg.sample.showcase.client.content.i18n.CwDictionaryExample.asyncOnInitialize	2574 Bytes (0.68%)
5	@pkg.sample.showcase.client.content.i18n.CwMessagesExample.asyncOnInitialize	2352 Bytes (0.62%)
6	@pkg.sample.showcase.client.content.i18n.CwNumberFormat.asyncOnInitialize	7439 Bytes (1.96%)
7	@pkg.sample.showcase.client.content.i18n.CwPluralFormsExample.asyncOnInitialize	1785 Bytes (0.47%)
8	@pkg.sample.showcase.client.content.lists.CwListBox.asyncOnInitialize	2439 Bytes (0.64%)
9	@pkg.sample.showcase.client.content.lists.CwMenuBar.asyncOnInitialize	3781 Bytes (1%)
10	@pkg.sample.showcase.client.content.lists.CwStackPanel.asyncOnInitialize	7426 Bytes (1.96%)
11	@pkg.sample.showcase.client.content.lists.CwSuggestBox.asyncOnInitialize	13835 Bytes (3.65%)
12	@pkg.sample.showcase.client.content.lists.CwTree.asyncOnInitialize	5230 Bytes (1.38%)
13	@pkg.sample.showcase.client.content.other.CwAnimation.asyncOnInitialize	2603 Bytes (0.69%)
14	@pkg.sample.showcase.client.content.other.CwCookies.asyncOnInitialize	3646 Bytes (0.96%)
15	@pkg.sample.showcase.client.content.other.CwFrame.asyncOnInitialize	493 Bytes (0.13%)
16	@pkg.sample.showcase.client.content.panels.CwAbsolutePanel.asyncOnInitialize	5116 Bytes (1.35%)
17	@pkg.sample.showcase.client.content.panels.CwDecoratorPanel.asyncOnInitialize	822 Bytes (0.22%)
18	@pkg.sample.showcase.client.content.panels.CwDisclosurePanel.asyncOnInitialize	6026 Bytes (1.59%)
19	@pkg.sample.showcase.client.content.panels.CwDockPanel.asyncOnInitialize	4353 Bytes (1.15%)
20	@pkg.sample.showcase.client.content.panels.CwFlowPanel.asyncOnInitialize	560 Bytes (0.15%)
21	@pkg.sample.showcase.client.content.panels.CwHorizontalPanel.asyncOnInitialize	515 Bytes (0.14%)
22	@pkg.sample.showcase.client.content.panels.CwHorizontalSplitPanel.asyncOnInitialize	3830 Bytes (1.01%)
23	@pkg.sample.showcase.client.content.panels.CwTabPanel.asyncOnInitialize	3474 Bytes (0.92%)
24	@pkg.sample.showcase.client.content.panels.CwVerticalPanel.asyncOnInitialize	514 Bytes (0.14%)
25	@pkg.sample.showcase.client.content.panels.CwVerticalSplitPanel.asyncOnInitialize	3901 Bytes (1.03%)
26	@pkg.sample.showcase.client.content.popups.CwBasicPopup.asyncOnInitialize	1858 Bytes (0.49%)
27	@pkg.sample.showcase.client.content.popups.CwDialogBox.asyncOnInitialize	5721 Bytes (1.51%)
28	@pkg.sample.showcase.client.content.tables.CwFlexTable.asyncOnInitialize	1773 Bytes (0.47%)
29	@pkg.sample.showcase.client.content.tables.CwGrid.asyncOnInitialize	542 Bytes (0.14%)
30	@pkg.sample.showcase.client.content.text.CwBasicText.asyncOnInitialize	3601 Bytes (0.95%)
31	@pkg.sample.showcase.client.content.text.CwRichText.asyncOnInitialize	10146 Bytes (2.67%)
32	@pkg.sample.showcase.client.content.widgets.CwBasicButton.asyncOnInitialize	934 Bytes (0.25%)
33	@pkg.sample.showcase.client.content.widgets.CwCustomButton.asyncOnInitialize	1463 Bytes (0.39%)
34	@pkg.sample.showcase.client.content.widgets.CwDatePicker.asyncOnInitialize	20911 Bytes (5.51%)
35	@pkg.sample.showcase.client.content.widgets.CwFileUpload.asyncOnInitialize	1250 Bytes (0.33%)
36	@pkg.sample.showcase.client.content.widgets.CwHyperlink.asyncOnInitialize	1669 Bytes (0.44%)
37	@pkg.sample.showcase.client.content.widgets.CwRadioButton.asyncOnInitialize	1235 Bytes (0.33%)

Table 5.2: GWT Showcase Manually Split, Split Breakdown and Size

No.	Class Containing Split	Split Size
1	@pkg.sample.showcase.client.Application.Application	861 Bytes (0.24%)
2	@pkg.sample.showcase.client.Application.Application;#2	686 Bytes (0.19%)
3	@pkg.sample.showcase.client.Application.onWindowResized	527 Bytes (0.15%)
4	@pkg.sample.showcase.client.ContentWidget.createWidget	317 Bytes (0.09%)
5	@pkg.sample.showcase.client.ContentWidget.createWidget;#2	267 Bytes (0.08%)
6	@pkg.sample.showcase.client.Showcase.onModuleLoad	447 Bytes (0.13%)
7	@pkg.sample.showcase.client.Showcase.onModuleLoad;#2	566 Bytes (0.16%)
8	@pkg.sample.showcase.client.Showcase.onModuleLoad;#3	2372 Bytes (0.67%)
9	@pkg.sample.showcase.client.Showcase.onModuleLoad;#4	213433 Bytes (60.48%)
10	@pkg.sample.showcase.client.Showcase$6.onSelection	480 Bytes (0.14%)
11	@pkg.sample.showcase.client.Showcase$7.onChange	598 Bytes (0.17%)
12	@pkg.sample.showcase.client.Showcase$8.onClick	674 Bytes (0.19%)
13	@pkg.sample.showcase.client.StyleSheetLoader.loadStyleSheet	1040 Bytes (0.29%)
14	@pkg.sample.showcase.client.content.i18n.CwConstantsExample.onInitializeComplete	511 Bytes (0.14%)
15	@pkg.sample.showcase.client.content.i18n.CwConstantsWithLookupExample.onInitializeComplete	507 Bytes (0.14%)
16	@pkg.sample.showcase.client.content.i18n.CwMessagesExample.onInitializeComplete	507 Bytes (0.14%)
17	@pkg.sample.showcase.client.content.lists.CwStackPanel.onInitialize	921 Bytes (0.26%)
18	@pkg.sample.showcase.client.content.lists.CwStackPanel.onInitialize;#3	776 Bytes (0.22%)
19	@pkg.sample.showcase.client.content.lists.CwStackPanel.onInitialize;#2	1974 Bytes (0.56%)
20	@pkg.sample.showcase.client.content.other.CwAnimation.onInitialize	1159 Bytes (0.33%)
21	@pkg.sample.showcase.client.content.panels.CwAbsolutePanel.onInitialize	2241 Bytes (0.64%)
22	@pkg.sample.showcase.client.content.panels.CwAbsolutePanel$4.onKeyUp	620 Bytes (0.18%)
23	@pkg.sample.showcase.client.content.panels.CwDisclosurePanel.onInitialize	6021 Bytes (1.71%)
24	@pkg.sample.showcase.client.content.tables.CwFlexTable$2.onClick	457 Bytes (0.13%)
25	@pkg.sample.showcase.client.content.text.RichTextToolbar.RichTextToolbar	760 Bytes (0.22%)
26	@pkg.sample.showcase.client.content.text.RichTextToolbar.RichTextToolbar	746 Bytes (0.21%)

Table 5.3: GWT Showcase Automatically Split, Split Breakdown and Size

Chapter 6: **Conclusions and Future Work**

_____

This research examines ways to aid a developer in applying code splitting to an application. At the most advanced level this is achieved by applying the splits automatically, based on the proposed algorithms. At a simpler level, feedback is given to the developer in a graphical view so they may make decisions on how to split the codebase. The research has examined several techniques that can be used to aid a developer when splitting code.

Considerable design and implementation work has been carried in this project. The research involved a large body of foundation work and as such, a very flexible parser and code analyser has been created. This will enable future research in the area of automated code splitting to focus solely on algorithm development, and not low-level implementation details. The work carried out in this project has resulted in the following code components:

- Extensible Java code parser and analyser based on the Java Compiler API

- GWT module that will add user logging to a GWT application

- Eclipse plugin to that displays graphical code splitting information

This study resulted in the completion of a splitting algorithm, static analysis, from beginning to end. Considerable solution design and development work has been carried out on the feature annotation and user usage algorithms. All that remains is to develop the business logic of these algorithms. The process for adding feature annotations and user usage logging as been completed, as has the parsing of this data into useful data structures.

From the algorithm completed and applied, it is clear that automated code splitting is a viable solution for the provision of code splitting to an application. The research illustrates that it is not only possible to automatically apply code splitting, but that the concepts proposed produce positive results, as can be seen in a reduction in the initial JavaScript code load of the Showcase application by 65%.

6.1   Future Work

The research has highlighted the many possibilities of tool-assisted code splitting for GWT. There is a wide scope for future development on the concepts explored here. Additional efforts would be required to make the prototype tools developed in this research production ready.

Future work that could be carried out in this research area would be to complete the business logic for the feature annotation and user usage algorithms. Improvements could also be made to the developed Eclipse plugin to provide automatic background running of the static analysis algorithm. This would supply instant feedback to the application developer and reduce the current trial and error development cycle of manual code splitting. It would be hoped that the result display of the static analysis could be used as a 'live' replacement for the GWT Compile Report, which is only available after compilation. This feature would reduce the

development cycle time for GWT developers as they would be getting instant feedback on the codebase.

This research has shown some of the possibilities and techniques available for applying automated code splitting to GWT applications. As web applications become more dynamic and complex, the need for code splitting will be greater. It is hoped that the techniques proposed in this project will enable more efficient code splitting.

Bibliography

[1] Google Web Toolkit Overview. Website, 2010.
   http://code.google.com/webtoolkit/overview.html.

[2] Soren Lassen and Sam Thorogood. Google Wave Federation Architecture. Website,
   2009.
   http://www.waveprotocol.org/whitepapers/google-wave-architecture.

[3] Fred Sauer. Riding the Google Wave. Website, 2009.
   http://googlewebtoolkit.blogspot.com/2009/09/riding-google-wave.html.

[4] YUI Project Homepage. Website, 2010.
   http://developer.yahoo.com.

[5] Prototype Project Homepage. Website, 2010.
   http://www.prototypejs.org/.

[6] jQuery Project Homepage. Website, 2010.
   http://www.jquery.com/.

[7] Google Web Toolkit Code Splitting Reference. Website, 2010.
   http://code.google.com/webtoolkit/doc/latest/DevGuideCodeSplitting.
   html.

[8] Eclipse Project Page. Website, 2010.
   http://www.eclipse.org/.

[9] Peter von der Ahe. JSR 199: JavaTM Compiler API. *JSR 199: Java8482*, 2006.

[10] Java Platform Standard Ed. 6 Documentation. Website, 2009.
   http://java.sun.com/javase/6/docs/api/javax/tools/JavaCompiler.html.

[11] B. Livshits and E. Kiciman. Doloto: Code Splitting for Network-bound Web 2.0 Appli-
   cations. *In M. J. Harrold and G. C. Murphy, editors, Proceedings of the International
   Symposium on Foundations of Software Engineering*, pages 350–360, 2008.

[12] Google Web Toolkit Showcase Sample Application. Website, 2010.
   http://gwt.google.com/samples/Showcase/Showcase.html.

[13] Selenium hq project homepage. Website, 2010.
   http://www.seleniumhq.org/.

Appendix A: **GWT Showcase Application**
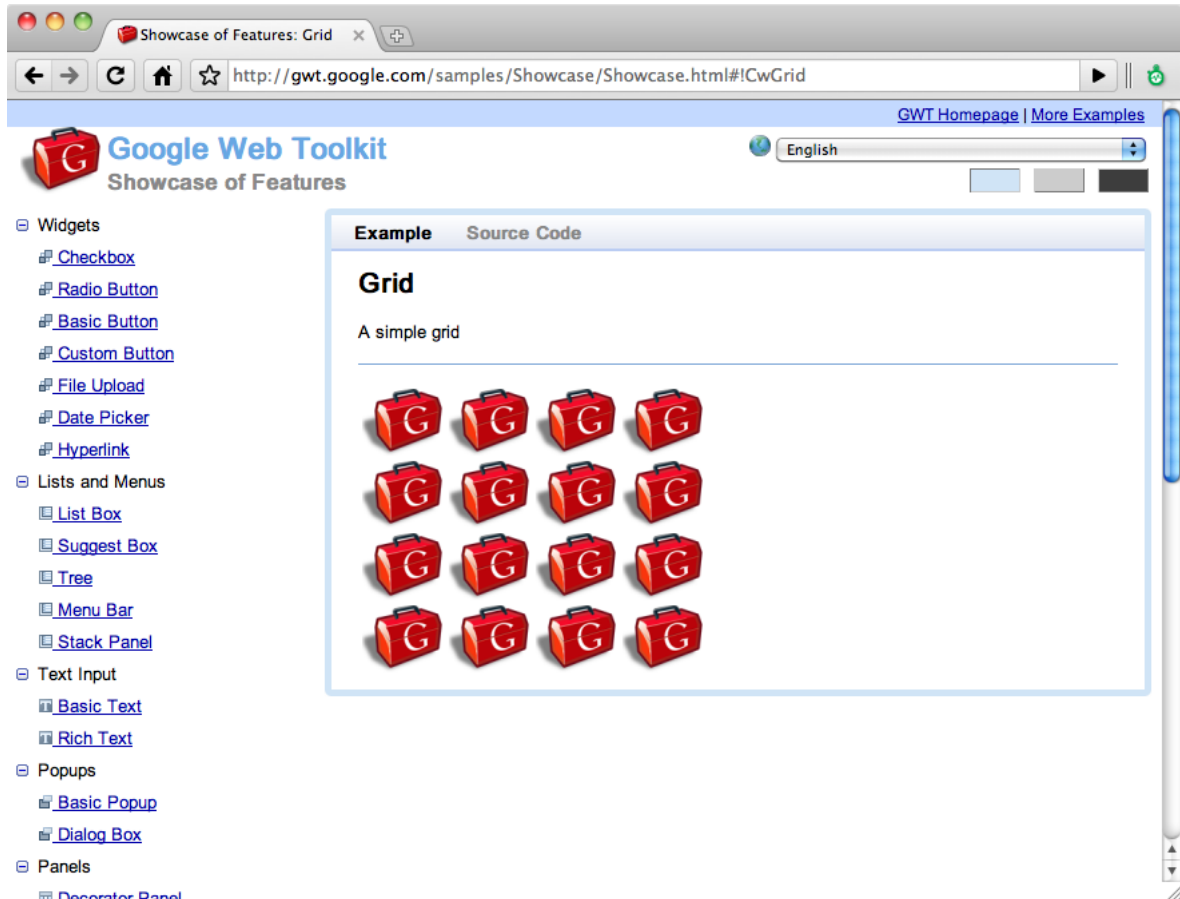
Figure A.1: GWT v2.0 Showcase Application

Appendix B: **GWT Compile Reports**

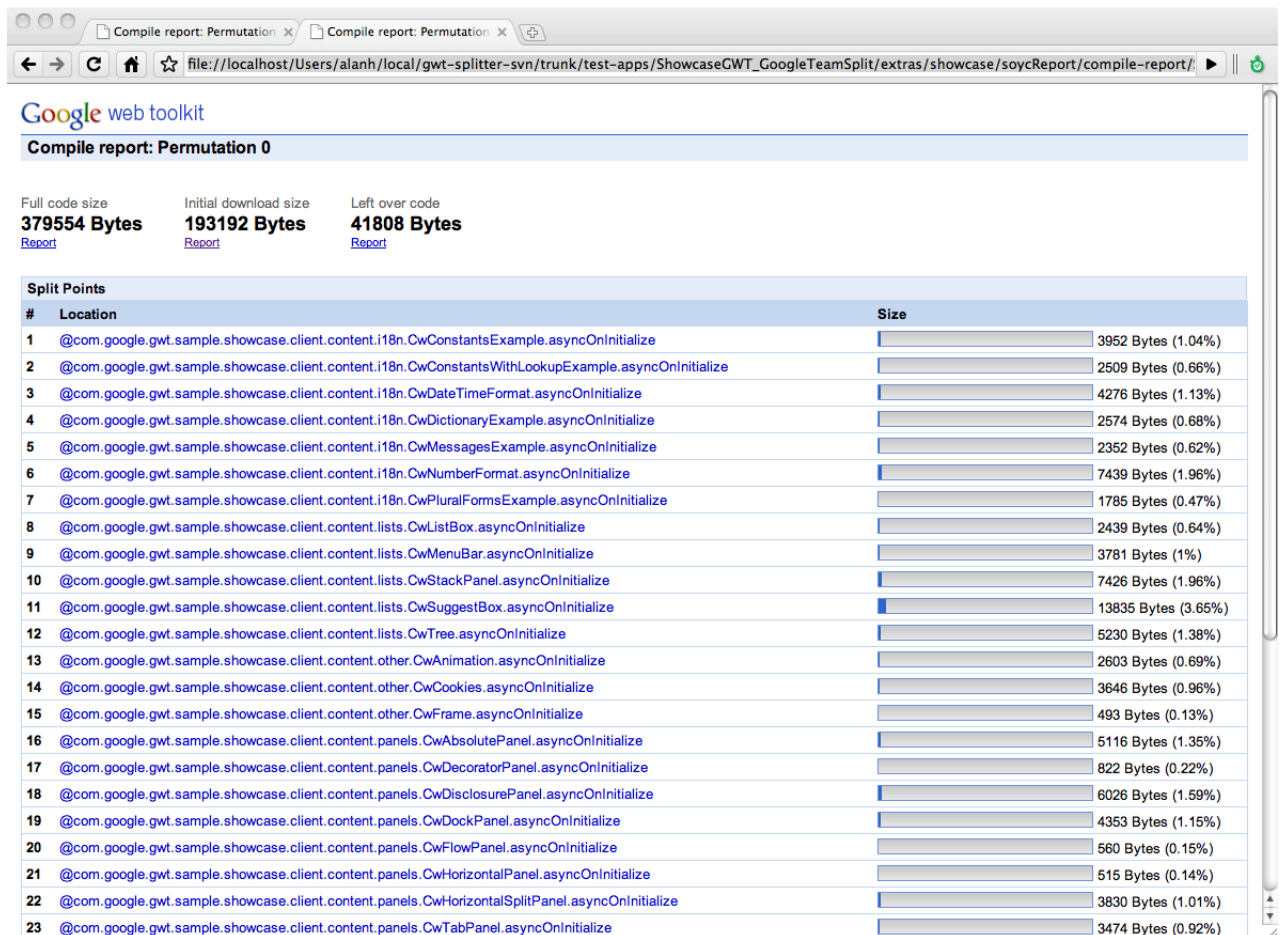## B.0.1 Showcase Application Manually Split

Figure B.1: GWT Compile Report

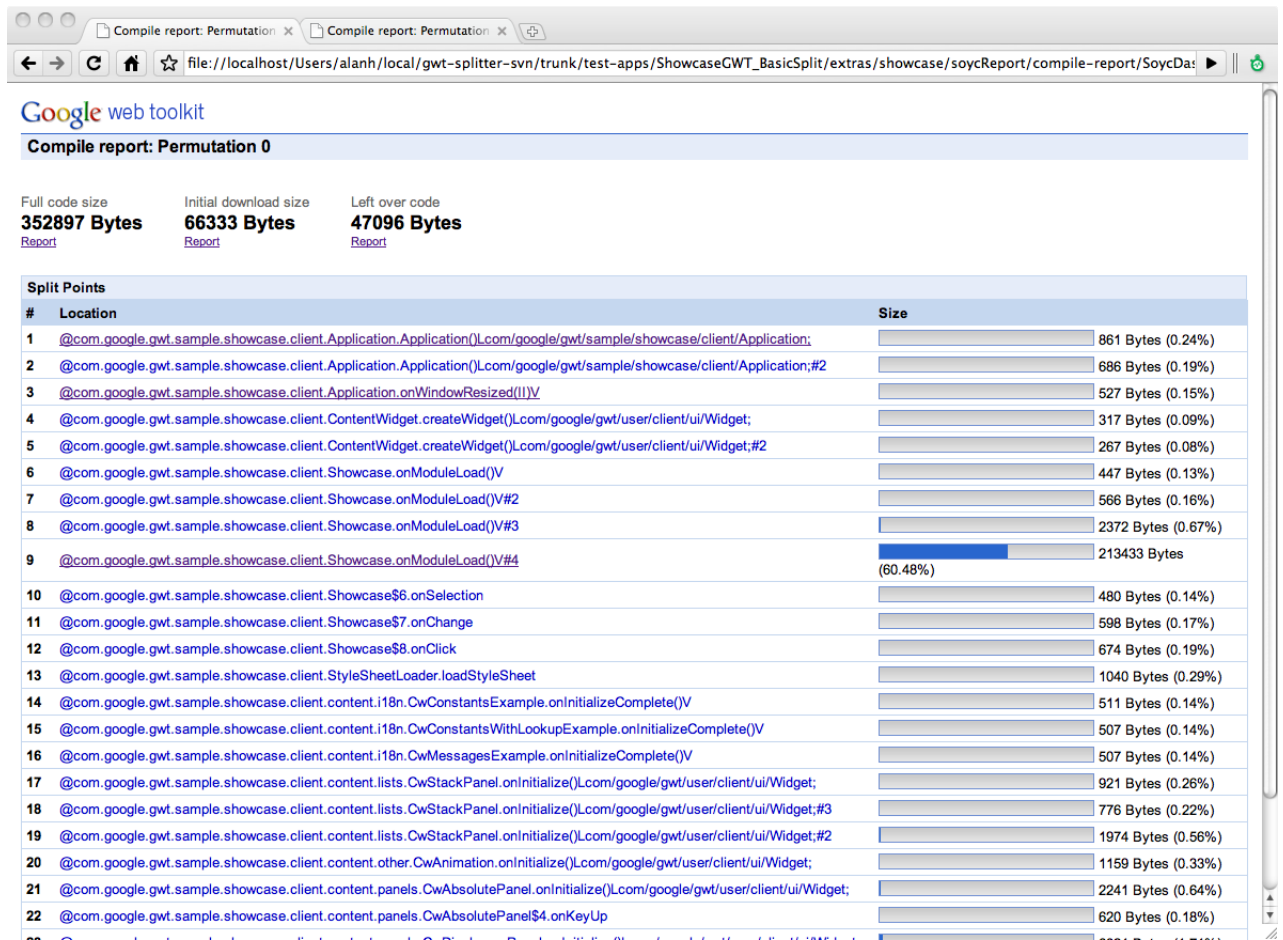## B.0.2 Showcase Application Automatically Split

Figure B.2: GWT Compile Report
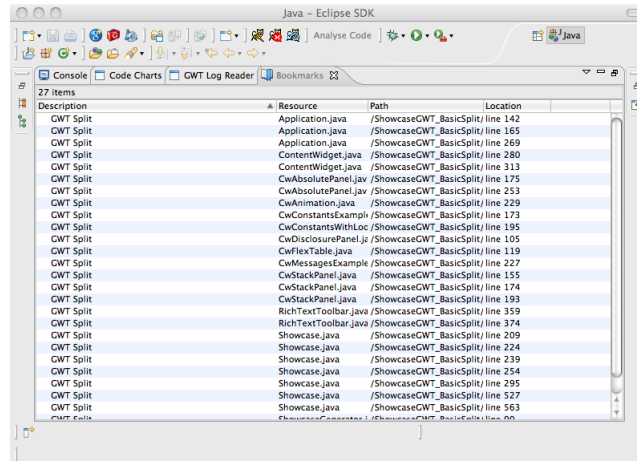
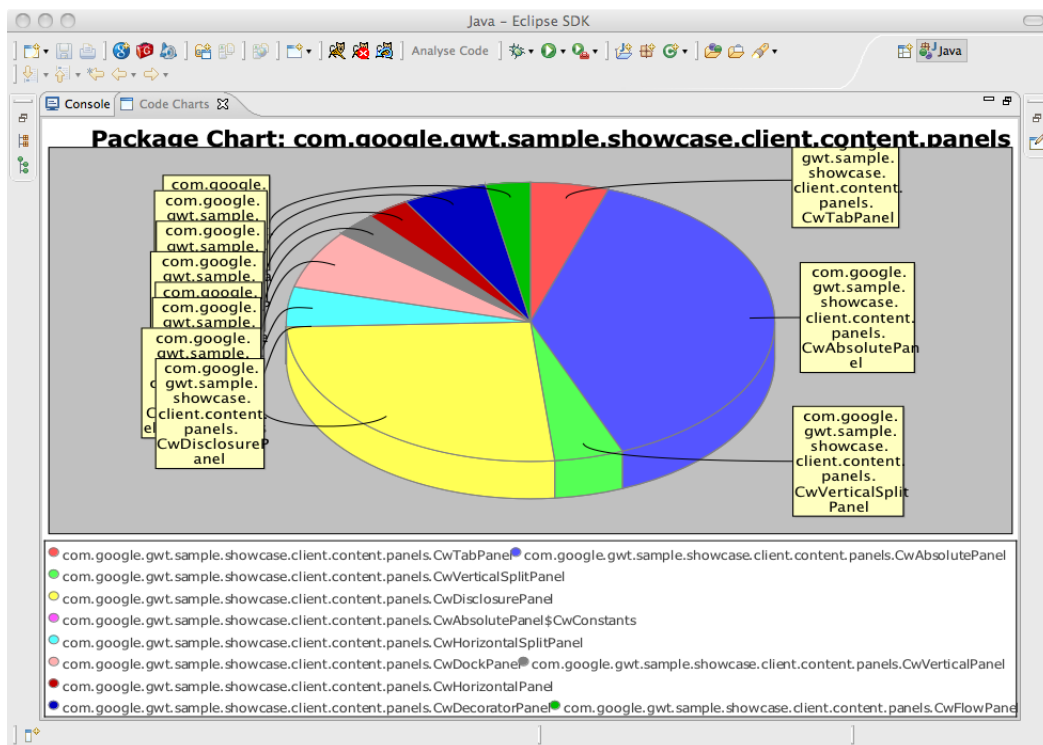# Appendix C: **Eclipse Plugin Screens**

Figure C.1: Eclipse Plugin Bookmark View
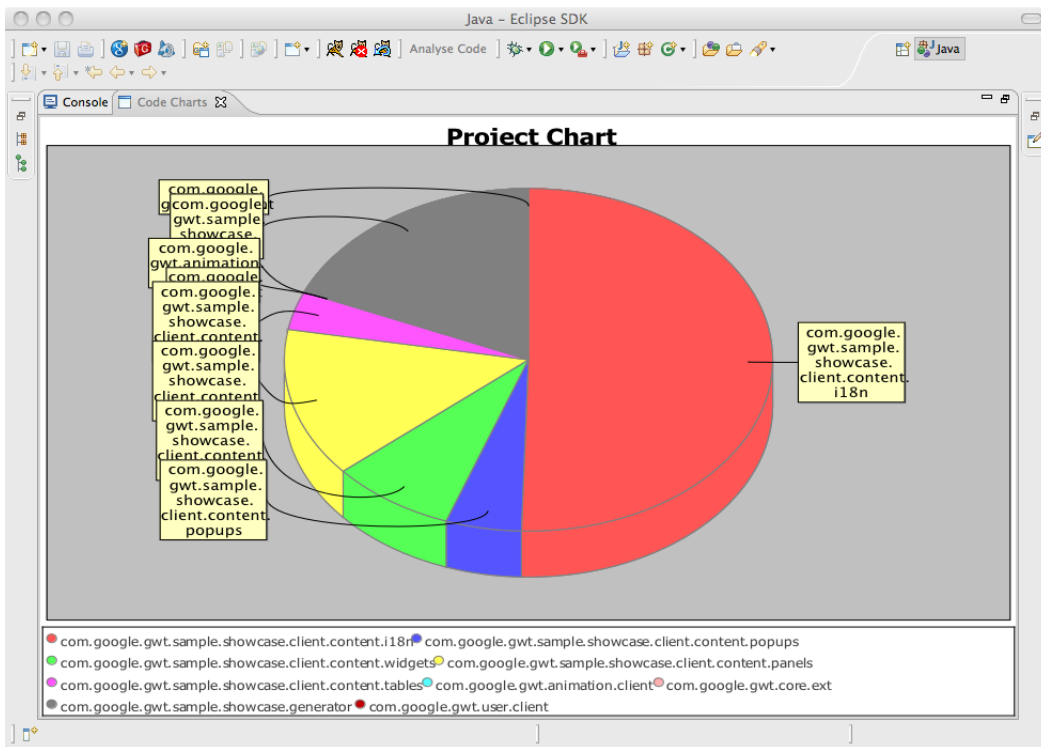

Figure C.2: Eclipse Plugin Class Size Chart View

Figure C.3: Eclipse Plugin Package Size Chart View
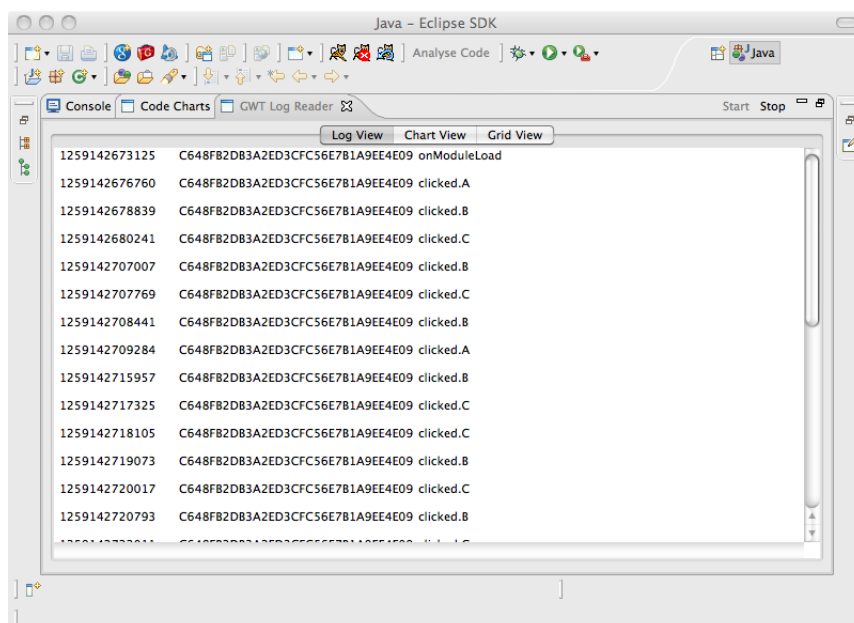
Figure C.4: GWT LogReader, Console View
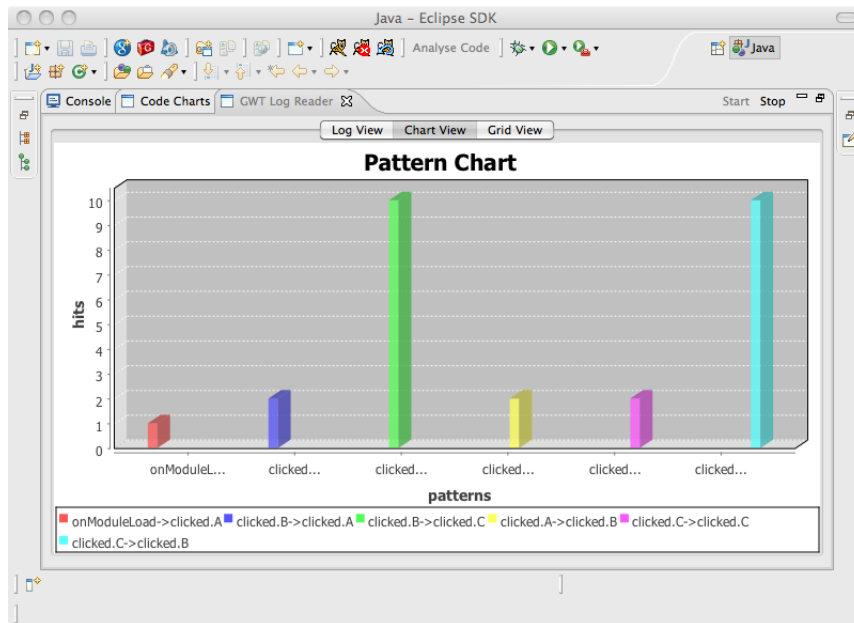
Figure C.5: GWT LogReader, Graphical View

Figure C.6: GWT LogReader, Pattern View

Appendix D: **Source Code**

<hr>

Listing D.1: GSClass.java: CodeInspect Representation of a Class

```java
package ie.gwtsplitter.codeinspect.domain;

import java.util.LinkedList;
import java.util.List;

/**
 *
 * @author Alan Hicks (al.hicks@gmail.com)
 */
public class GSClass {

  public final String name;
  public final String prettyName;
  private List<GSMethod> methods;
  private List<GSSplit> splits;
  public GSClass gsExtends = null;

  public GSClass(final String prettyName, final String name) {
    this.prettyName = prettyName;
    this.name = name;
    methods = new LinkedList<GSMethod>();
    splits = new LinkedList<GSSplit>();
  }


  public final String toString() {
    ...
  }

  public final int getSize() {
    int size = 0;
    for (GSMethod method : methods) {
      size += method.totalSize();
    }
    return size;
  }

  public final boolean findUsages(final GSMethod currentMethod) {
    for (GSMethod method : methods) {
      if (method.name.equals(currentMethod.name)) {
        return true;
      }
    }
    return false;
  }

  public final void addMethod(final GSMethod method) {
    methods.add(method);
  }

  public final List<GSMethod> getMethods() {
    List<GSMethod> tMethods = methods;
    if (gsExtends != null) {
      for (GSMethod m : gsExtends.getMethods()) {
        boolean add = true;
        for (GSMethod t : methods) {
          if (t.name.equals(m.name)) {
            add = false;
            break;
          }
        }
        if (add) {
          tMethods.add(m);
        }
      }
    }
    return tMethods;
  }

  public final GSMethod getMethod(final String methodName) {
    for (GSMethod method : methods) {
      if (method.name.equals(methodName)) {
        return method;
      }
```

```
    }
    if (gsExtends != null) {
      return gsExtends.getMethod(methodName);
    }
    return null;
  }

  public final void addSplit(final GSSplit split) {
    if (!splits.contains(split)) {
      splits.add(split);
    }
  }

  public final List<GSSplit> getSplits() {
    return splits;
  }

}
```

Listing D.2: GSMethod.java: CodeInspect Representation of a Method

```java
package ie.gwtsplitter.codeinspect.domain;

import ie.gwtsplitter.codeinspect.Marker;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 *
 * @author Alan Hicks (al.hicks@gmail.com)
 */
public class GSMethod {

  public final String name;
  public final String prettyName;
  public final boolean codeGenerated;
  public String accessor;
  private int size;
  public final GSClass gsClass;
  public boolean isInnnerSplitMethod = false;
  private String trackTag;
  private Marker marker;

  /**
   * Used to record Marker locations of where this method is called.
   */
  public Map<String, Marker> callMarkers;

  /**
   * Used to record places in where this method is called.
   */
  public List<GSMethod> calls;

  /**
   * Used to record methods that are called within this method.
   */
  public List<GSMethod> innerMethodCalls;


  public GSMethod(final GSClass gsClass, final String name,
      final String prettyName) {
    this.gsClass = gsClass;
    this.name = name;
    this.prettyName = prettyName;
    this.codeGenerated = prettyName.startsWith("<");
    this.calls = new ArrayList<GSMethod>();
    this.innerMethodCalls = new ArrayList<GSMethod>();
    this.callMarkers = new HashMap<String, Marker>();
  }

  public final String toString() {
    ...
  }

  public final void setTrackTag(final String trackTag) {
    this.trackTag = trackTag;
  }

  public final String getTrackTag() {
    return this.trackTag;
  }
```

```java
  public final void setMarker(final Marker marker) {
    this.marker = marker;
  }

  public final Marker getMarker() {
    return this.marker;
  }

  /**
   * Dead code is code that is not called and hence gets
   * removed by the GWT compiler. If method is onModuleLoad return false.
   * This method is the GWT entry point.
   * @return boolean
   */
  public final boolean isDeadCode() {
    return false;
  }


  /**
   * Set the raw size of the method block.
   * @param size method body size
   */
  public final void setSize(final int size) {
    this.size = size;
  }


  /**
   * Calculates the total size of the method including the inner method
   * call sizes.
   * @return total size of method
   */
  public final int totalSize() {
    int size = this.size;
    for (GSMethod method : innerMethodCalls) {
      size += method.totalSize();
    }
    return size;
  }


  /**
   * Construct the full method name.
   * @param uniqueName flatname of method class
   * @param prettyName human readable method name
   * @return full unique method name, example: uniqueName_prettyName
   */
  public static String methodName(final String uniqueName,
      final String prettyName) {
    return uniqueName + "_" + prettyName;
  }

}
```