AutoCode: Using Memex-like Trails to Improve Program Comprehension

Richard Wheeldon, Steve Counsell and Kevin Keenoy
Department of Computer Science
Birkbeck College, University of London
London WC1E 7HX, U.K.
{richard,steve,kevin}@dcs.bbk.ac.uk

Abstract

This paper presents AutoCode - a system for identifying "trails" of classes in Java programs. These trails are computed with regard to five coupling relationships (Aggregation, Inheritance, Interface, Parameter and Return Type) and are presented in a Web-based interface.

1 Introduction

In his seminal paper "As We May Think" [1], Vannevar Bush suggested a future machine called a "memex". In doing so, he introduced the world to the concept of linked documents and of the *trail* - a sequence of linked pages. The concept of trails is well established in the hypertext community and many systems have been built which support their construction [3].

Previous work has described a *navigation engine* for automatically constructing trails as a means of assisting users browsing Web sites [5]. This navigation engine was further used to provide search and navigation facilities for Javadoc program documentation. If JavaDoc-style program documention, derived from source code, can be indexed, it seems logical that the source code itself should be indexed also.

We have developed a new tool called AutoCode based upon the navigation engine design. AutoCode provides full-text indexing of Java source code and uses a probabilistic best-first algorithm to identify trails in graphs of coupling-type relationships.

2 Trails on Java Code

Classes and objects in OO systems do not work in isolation. They are connected to each other by various dependencies. The Java language connects classes together via five coupling relationships - Aggregation, Inheritance, Interface, Parameter and Return Type [4]. Each of these coupling relations can be used to construct a graph of dependencies. An illustration of how these graphs can be derived from Java source code can be seen in Figure 1. AutoCode constructs trails on each of these five graphs and presents them in a Web-based interface.

The *NavSearch* user interface used to present the trails (Figure 2) has three main elements. At the top is a *navigation tool bar* comprising a trail of classes considered most relevant (the "best trail"). On the left is a *navigation tree window* showing all the trails. Whenever the mouse pointer moves over these trails, a small pop-up appears which shows metadata and an extract. The rest of the display is dedicated to showing the source code of the selected class. The original Web-site search interface on which it was based was proven to be highly effective at allowing users to complete information seeking tasks [2]. It is hoped the same will apply to the AutoCode interface, a demonstration of which is available at http://nzone.dcs.bbk.ac.uk/.

Each trail is colour-coded according to the type of coupling involved. This coupling type is also shown in the pop-up for each class. Green trails denote parameter type references, cyan trails denote return-type references, gold trails show interface extensions, purple trails shows chains of aggregation links and orange trails show inheritance relationships from subclass to superclass.

Figure 2 shows how the trails are presented for the results to the query "zip" on the JDK 1.4 source code. Figure 3 shows the trails more clearly. It can be easily seen from the first trail that there is a member variable

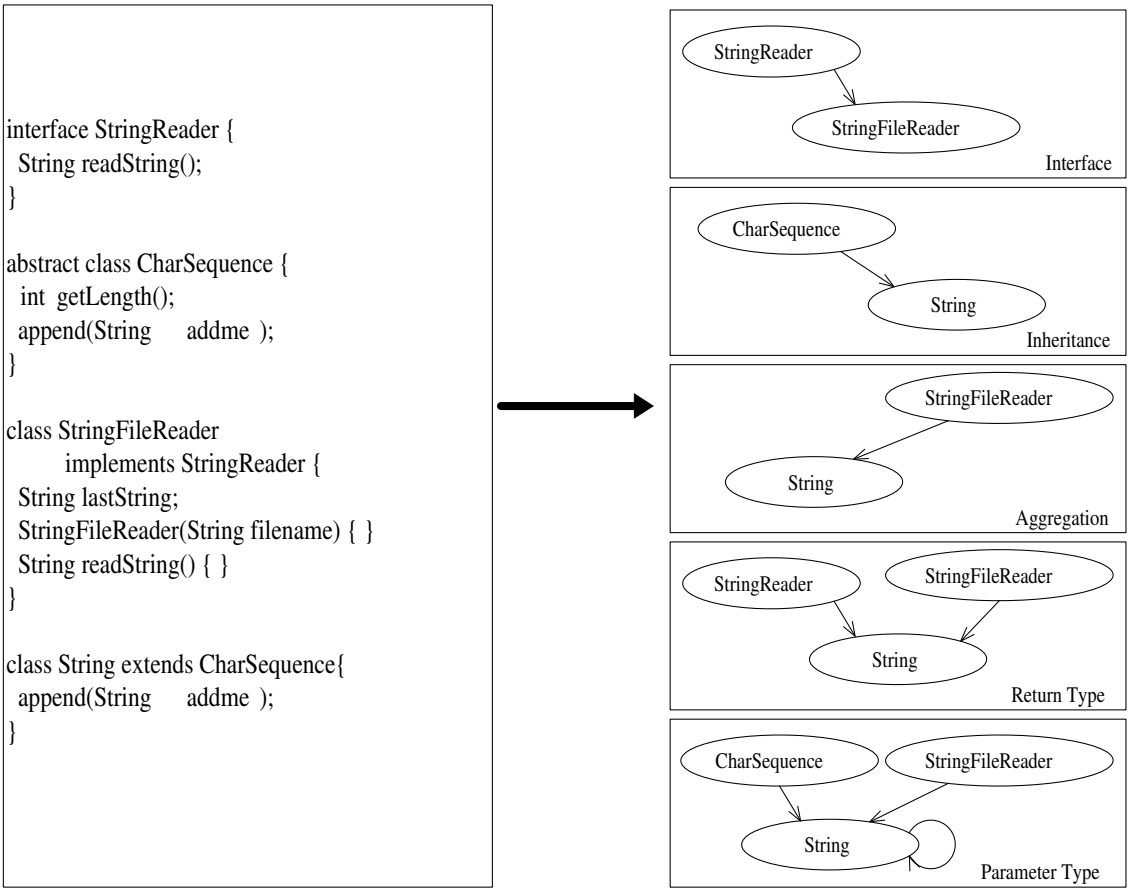

Figure 1. Illustration of coupling types and their graph representations.

of type `ZipFile` in the class `ZipFileInputStream`. The second and third trails start with the common root, `ZipFile`. These show that one or more methods in the `ZipFile` class must take `ZipEntry` as a parameter and that `ZipFile` has a subclass called `JarFile`. The fourth trail shows that `ZipFile` implements the interface `ZipConstants`. The fifth shows that `ZipOutputStream` has a member variable of type `ZipEntry`. The sixth and seventh trails show that both `ZipInputStream` and `JarFile` have methods which take `ZipEntrys` as parameters. The eighth trail shows that `JarInputStream` has at least one method which returns a `ZipEntry` and the ninth shows that `ZipEntry` is the superclass of `JarEntry` which is, in turn, the superclass for `JarFile.JarFileEntry`.

Figure 3. Trails returned for the query "zip" on the JDK 1.4 source code.

3 Automating Trail Discovery

Given the graphs of related classes, the navigation engine can be used to construct trails. This works in 4 stages. The first stage is to calculate scores (using $tf.idf$) for each of the classes matching one or more of the keywords in the query, and to isolate a small number of these for future expansion, by combining these scores with a metric called *potential gain* [5, 4]. The second stage is to construct the trails using the *Best Trail* algorithm [5]. The algorithm builds trails using a probabilistic best-first traversal. The third stage involves filtering the trails to remove redundant information. In the fourth and final stage, the navigation engine computes small summaries of each class and formats the results for display in a web browser. Jason Shattu's Java2HTML[1] is used to present the source code, since it provides effective syntax highlighting, has a public API and makes links to both Javadocs and between classes in source code.

3.1 Architecture

AutoCode indexes the Java code using a custom doclet. Figure 4 shows how this works with the other elements of the navigation engine. The doclet uses the class structure to construct the five coupling graphs. It also communicates with an external parser, which manipulates the HTML representation of the source code to create an inverted file. This inverted file is used by the *query engine* to compute relevance scores for each class or page. The *trail engine* uses these scores and the coupling graphs to compute the trails. The NavSearch interface presents the trails as shown in Figure 2.

4 Future Work

Object Oriented languages gain particular benefit from the mapping between classes and Web pages. It is intended that AutoCode be extended to support both C++ and C#. It is also hoped that the system can be extended to allow personalized results so that programmers working on a particular field have query results tailored to their needs.

Certain compromises have been made in the development of AutoCode, which should also be addressed in any future development. AutoCode neither shows the relationships between inner and outer classes nor discriminates between static and object references.

Other graphs can be constructed through static and runtime analysis. These include in-memory object references graphs and call-graphs. Any such graphs could be adapted for use with AutoCode.

AutoCode has been developed as a standalone tool operating within JavaDoc. As such it can be updated by any tool which can control JavaDoc, notably build tools such

[1] http://java2html.com/

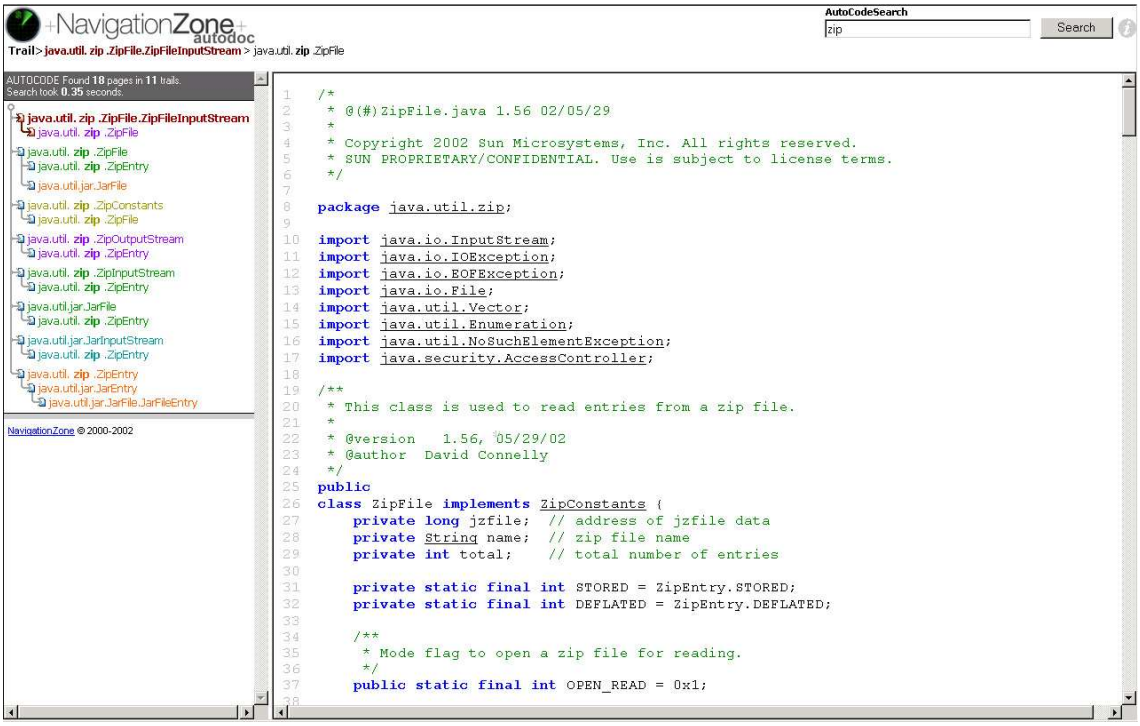

Figure 2. Results for the query "zip" on the JDK 1.4 source code.

as Apache Ant. However, it would be beneficial to embed the interface within a Java IDE so that identified classes can be immediately edited.

Combining these elements would provide developers with a much more flexible tool for identifying relevant classes and the relationships between them.

5 Conclusions

This paper has presented AutoCode - a Web-based tool for computing and presenting memex-like trails across coupling graphs. It benefits from a simple, web-based interface with a strong, well-explored metaphor for displaying class relationships. It works well with very large programs and libraries. For example, the JDK libraries which contain over 6 000 classes and over 1 400 000 lines of code. AutoCode also benefits from platform and IDE independence and uses indexes which can easily be updated during the build process.

However, AutoCode is not without problems. It is restricted to a single language - Java, and ignores certain important relationships between classes. However, it is restricted by a lack of editing features, meaning that identified classes cannot be manipulated without a separate editor.

References

[1] Vannevar Bush. As we may think. *Atlantic Monthly*, 76:101–108, 1945.

[2] Mazlita Mat-Hassan and Mark Levene. Can navigational assistance improve search experience: A user study. *First Monday*, 6(9), 2001.

[3] Siegfried Reich, Leslie Carr, David De Roure, and Wendy Hall. Where have you been from here? : Trails in hypertext systems. *ACM Computing Surveys*, 31(4), December 1999.

[4] Richard Wheeldon and Steve Counsell. Making refactoring decisions in large-scale java systems: an empirical stance. *Computing Research Repository*, cs.SE/0306098, June 2003.

[5] Richard Wheeldon and Mark Levene. The best trail algorithm for adaptive navigation in the world-wide-web. In *Proceedings of 1st Latin American Web Congress*, Santiago, Chile, November 2003.

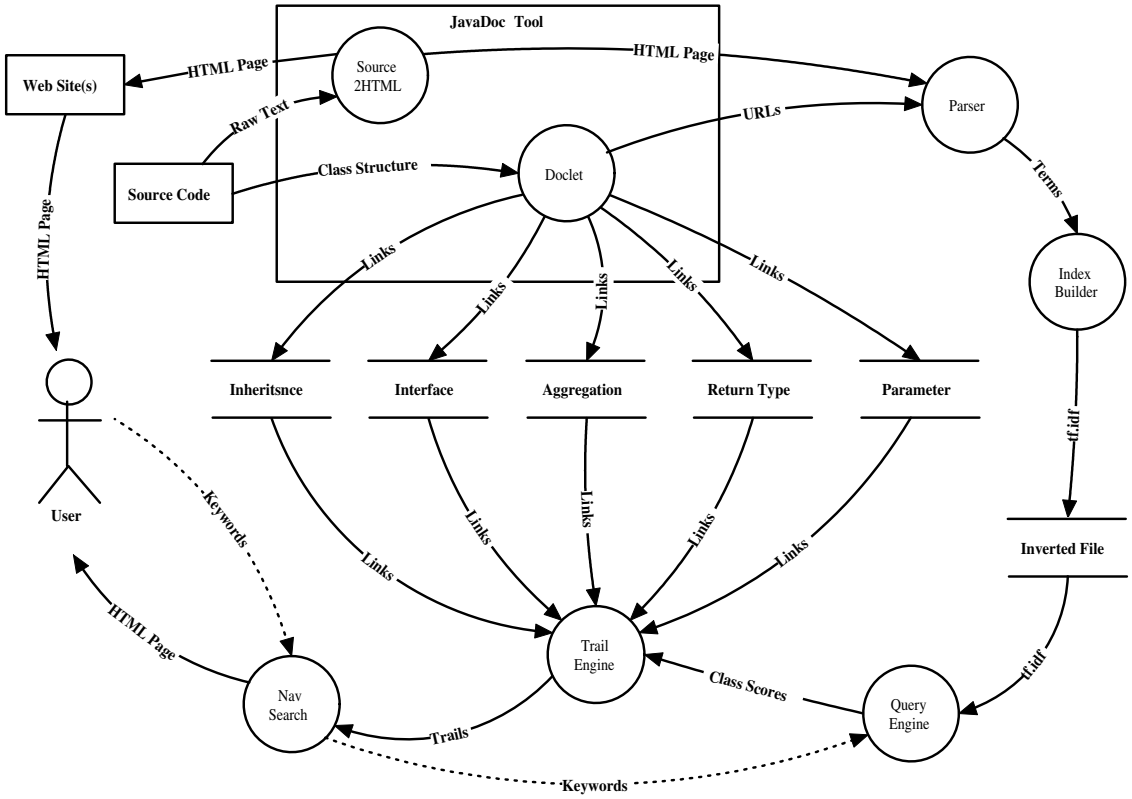

Figure 4. Architecture of AutoCode. Boxes represent external data sources, open-ended boxes represent internal data stores, circles represent processes, solid arrows represent data flow and dotted arrows represent flows of important information (URLs and Queries). Simple keyed "get" instructions (for example in HTTP requests) are omitted for clarity.