Extraction and Use of Class Dependency Information for Java

Larry A. Barowski and James H. Cross II
Computer Science and Software Engineering
Auburn University, AL 36849
{larrybar, cross}@eng.auburn.edu

Abstract

In this paper, a method for extracting class dependency information from Java class files is described. Advantages and disadvantages of using this method are discussed. The problems of virtual dependence and synthetic methods are explained, and solutions offered. A convenient user interface is presented for making use of the dependency information in the form of an interactive UML class diagram, which is automatically generated from Java class files. This interface is a component of the jGRASP integrated development environment.

1. Introduction

Basic dependency and architectural information is essential for understanding an object-oriented software system. This information is perhaps most useful when presented as a directed graph where nodes represent classes and edges represent dependencies. This type of diagram can be used, for example, in finding groups of related classes or in determining which classes must be examined when the specification of a class is changed. The UML class diagram is an example of such a nodes-and-edges representation [2]. UML class diagrams are well known, standardized, and include the necessary elements for class dependency display.

In addition to determining dependencies among classes, it is also useful to know the reasons for those dependencies (the fields and methods referenced), and to be able to quickly jump to those references in the source code. This can be used, for example, to find and eliminate unwanted dependencies among classes, or to find and understand unexplained dependencies. This idea can be extended to formal code reviews where the process includes examining each dependency and verifying that it should exist.

Above, we use *class* to refer to either a Java class or interface, and *method* to refer to either a Java method or constructor. We will continue to do so throughout this paper, and the general meanings should be assumed unless otherwise specified. We also use the general meaning of *dependency*, which includes inheritance and interface implementation (generalization and realization in UML terminology). We refer to inheritance and interface implementation dependencies together as *architectural dependencies* and others as *references*.

2. Extracting Dependency Information from Class Files

Dependency information is typically gathered from source code. In the case of Java, it is possible to get this information from object code (class files). Existing dependency-analysis tools use either source code or class files. There are three important advantages to using class files. First, the compiler has done the complicated and error prone work of parsing the source code and building symbol tables. Java compilers are heavily tested and reliable. Second, the dependencies in source code are not always clear. Dependencies that are not obvious may exist, and apparent dependencies may not be real. A "reasonable" interpretation of source code dependencies is not as reliable as the extraction of actual dependencies from class files. Third, source code may not be available. This is especially likely in the case of third party libraries. Tools exist for reconstructing source code from class files, but doing so is inconvenient, and may be impossible in the case of obfuscated class files.

The primary disadvantage of using class files to gather dependency information is that Java compilers do in some cases alter dependencies through the creation of synthetic methods. It is generally possible to work around this problem, and a basic strategy for doing so is described here in section 2.4. Another disadvantage of using class files is that the source code locations of dependencies (field, method, and class reference locations) can be identified only to the nearest line number. If source files are used, the exact source code locations (line and column numbers) can be identified. Finally, generating dependency information from class files requires that the source files be compiled before dependency analysis is performed. Since parsing source code to gather the data typically requires compilable code, this is not a problem in most cases.

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 **IEEE**

2.1 Java Class Files

A class file is a component of a Java executable corresponding to a single Java class. It contains tables describing the structure of the class and virtual machine byte code for the class methods. Since class files are "live" (if dependencies are satisfied, they can be moved from one project to another without recompiling), they must contain all dependency information. In other words, a Java compiler is not allowed to remove any dependencies through optimization processes (in Java, optimization is done during runtime). The Java class file format is very referential, but straightforward. The DataInputStream class in the Java libraries provides helpful methods for lexical scanning of class files, so parsing class files using Java is convenient. In byte code, classes, fields, and methods are referenced symbolically using information contained in a table, so it is not necessary in general to parse byte code to find dependencies (though byte code must be parsed in order to find source code locations of reference dependencies). The class file format is detailed in the "Java Virtual Machine Specification" [1]. Since all Java compilers must generate class files conforming to this specification, extraction of dependency data is not dependent on the particular compiler used, except with respect to synthetic methods as described in section 2.4.

2.2 Extracting Raw Dependency Data

Each class file contains a reference to the parent class, if any, as well as a list of the interfaces the class implements. This comprises all of the architectural dependencies. General dependencies (references) can be found in the class file's constant pool. This structure contains representations of, among other things, the external classes, methods, and fields that may be used by the class. As a practical matter, it can be assumed that any class, method, or field in the constant pool is used by the class and represents an actual dependency. A table of field representations can be used to distinguish associations from other references. A table of methods contains a representation of each method including the byte code for the method.

2.3 Dependencies on Virtual Members

In an object-oriented language, a virtual class member is any inheritable field or method of a class that a subclass is allowed to override. In Java this would be any class member that is not *private* and not *final*. A reference to a virtual class member through a class which does not override that member raises a question as to whether the dependency is on the referenced class, or the nearest ancestor of the referenced class which actually contains the referenced member.

In the case of virtual class members, we consider a reference to a particular class to be a dependency on that class, even when a referenced field or method is not present in that class. This is sensible, since the member may be defined in a super class or sub class, and this may change each time the reference is used at runtime. In the case of Java, the referenced class might even be switched at runtime for one in which the referenced member is defined in the class and is not virtual. For example, consider the code in Figure 1. Class C references field x of class B. Although x is not defined in class B, this is still a dependence of class C on class B.

One exception is a virtual self-reference. We know whether or not the class itself redefines a virtual member. It is meaningful to consider a self-reference to a member that is not defined in the class to be a dependence on the parent class. For example, in the code of Figure 1, class B makes a reference to field x. Class B is allowed to override x, but we know it does not. This is considered a dependence of class B on class A.

```
public class A {
   int x = 6;
}

public class B extends A {
   int getX() {
      return x; }
}

public class C {
   int getX(B b) {
      return b.x; }
}
```

Figure 1. Virtual Dependencies

2.4 Synthetic Methods in Java

In order to minimize changes to the Java Virtual Machine when inner classes were added to the Java language, Sun elected to work around its limitations by having the compiler add synthetic methods. An inner class is a class defined within the scope of another class. The enclosing class is referred to as the "outer" class. Inner and outer classes have access to each other's private members, but other pairs of classes do not. No knowledge of inner and outer classes, which would enable access to private members, is contained in the class file. Instead, the compiler generates an accessible method for each type of access or modification to a private member of an inner or outer class by its corresponding outer or inner class, and changes those accesses and modifications into synthetic method calls. For example, in Figure 2, class In accesses the private field x in class Out. The

compiler will generate a method allowing In to access x as shown in the figure.

```
public class Out {
   private int x;

   class In {
      public int getX() {
         return x; }
      }
   }

   // Compiler will generate a
   // method equivalent to this.
   static int access$000(Out o) {
      return o.x; }
}
```

Figure 2. Synthetic Methods

In the case of synthetic methods, the important information is the "real" references they represent. This information can be extracted from the byte code for those methods. All synthetic methods are flagged in the class file and thus are easily identified. In the case of Java compilers from both Sun and IBM, synthetic methods are simple and take a limited number of forms, so finding the real references does not require a complete byte code scan. To guarantee a correct result for any possible Java compiler, a full byte code scan and analysis would be necessary, as other compilers could work around the inner/outer class problem in slightly different ways. In class files generated by Sun or IBM compilers, each synthetic method: calls a method and possibly returns a value, returns a field value, or sets a field value. A rough examination of the byte code is sufficient to determine the method that is being invoked, or the field that is being accessed or set. Figure 3 shows the byte code for the synthetic method of Figure 2, with raw byte code in the left column. The argument to *getfield* is a two-byte index into the constant pool for class Out. The field at this index (Out.x) is the real dependency for any class that calls Out.access$000. The Java byte code format is detailed in the "Java Virtual Machine Specification" [1].

```
static int access$000(Out o)

42        aload_0 o
180 0 1   getfield Out.x
172       ireturn
```

Figure 3. Byte Code for Synthetic Method

2.5 Reference Dependency Locations

In most cases, Java class files contain line number information in tables that relate byte code locations to source code lines (obfuscated class files may not contain this information or it may be scrambled). Therefore, a complete scan of the byte code for a class can be used to find reference dependency locations (field, method, and class reference locations), to the accuracy of a line. Positions of the references within a line however, cannot be determined in this way.

3. Dependency Display as an Interactive UML Class Diagram

Since UML class diagrams are well known and standardized, we chose this format for display of class dependencies. A nodes-and-edges structure is an obvious representation for classes and the dependencies among them. UML specifies a specific appearance for these nodes and edges [2].

3.1 Interactivity

When a UML class diagram is presented in software, interactive functionality can be added. One useful function is to have selecting (or clicking on) a dependency display detailed information, such as the fields and references that cause the dependency. Similarly, clicking on a class can be used to show the fields of the class and the methods implemented by it.

Also, interactivity allows a user to reformat the diagram. Classes can be moved and edge bends added or moved to produce a more pleasing layout than can be done automatically, or to highlight a particular segment of the diagram.

3.2 Display Choices

In displaying dependency information as a UML class diagram, we elected not to distinguish use in a field (UML association) from other references. Association and its more specific forms (UML aggregation and composition) are difficult to distinguish automatically, and in some cases the difference is a matter of interpretation.

Also, it was decided not to show a reference edge between classes when an inheritance or interface implementation edge also exists. Thus, only one edge in each direction is shown between any two classes. This makes the diagram less cluttered. Information for the reference is shown when the inheritance or interface implementation edge is selected by the user.

4. UML Diagrams in jGRASP

The jGRASP interactive development environment provides for the automatic generation of UML class diagrams from the class files generated by the Java compiler as discussed in the previous sections [3]. Figure 4 shows a typical jGRASP desktop, composed of a Control Panel containing a menu and three panes: (1) left pane with tabs for **Browse**, **Project**, **Find**, and **Debug**, (2) right pane for CSD (control structure diagram) Windows, and (3) lower pane with tabs for jGRASP messages, Compile messages, and input/output for Run. The Project pane (left) indicates that PersonalLibraryProject consists of five source files. The source code for PersonalLibrary.java, which contains the *main* method, is displayed in the CSD window (right pane), and the control structure diagram (a control-flow and data structure diagram) has been generated. The message window (lower pane) indicates the program has been successfully compiled.

4.1 Generating the UML

The simplest method of generating a UML class diagram in jGRASP is to open the UML window and drag-and-drop Java source, class, or jar files into it. These files will automatically be added to the current project and the diagram will be immediately generated.

More commonly, the diagram will be generated for an existing project in jGRASP. After the program is

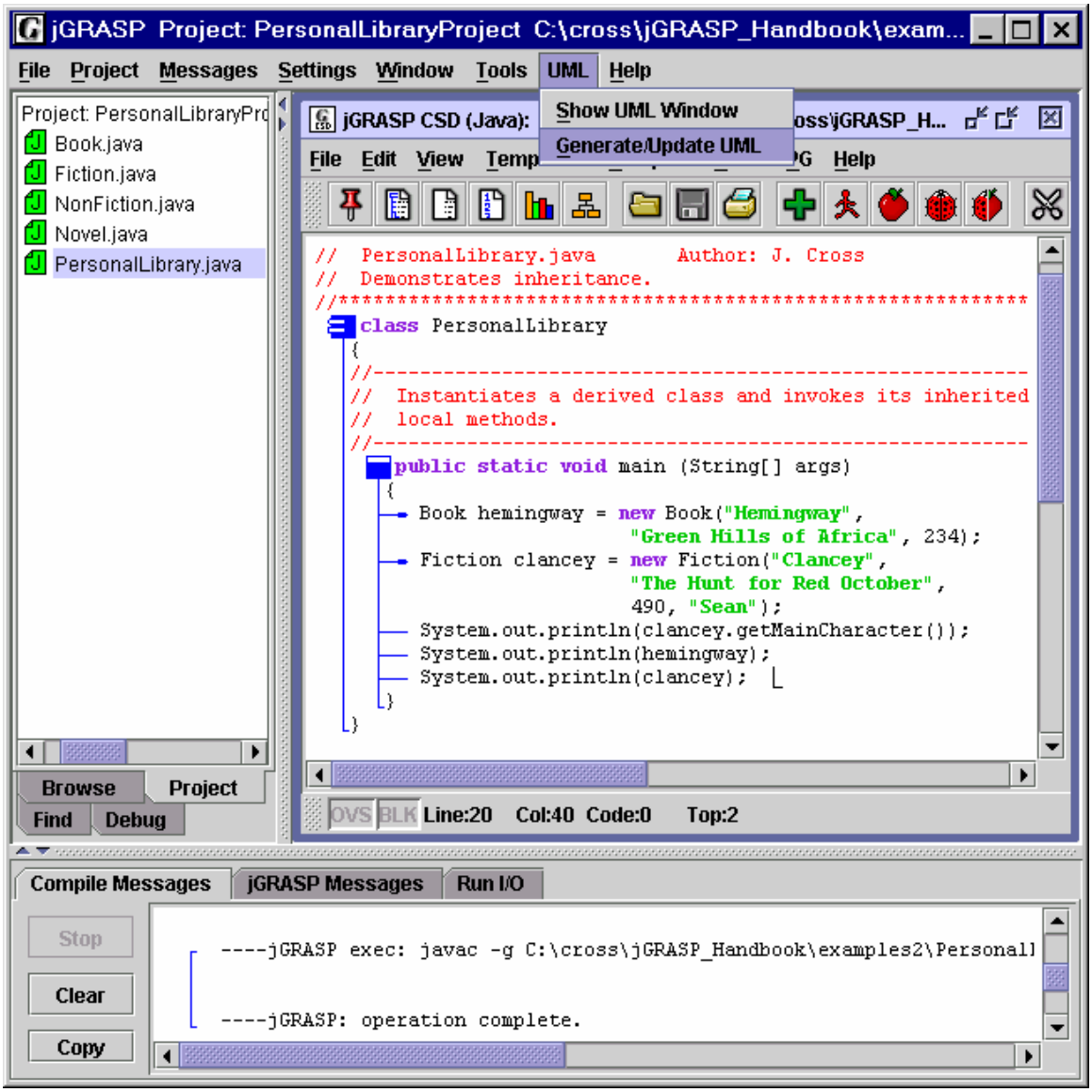

Figure 4. Generating the UML

compiled, the user simply clicks Generate/Update UML on the jGRASP desktop menu (Figure 4), and a separate jGRASP UML window containing the class dependency diagram for the project opens, as shown in Figure 5. Classes are depicted as rectangles. When viewed in color, the user's classes are shown in green and Java library classes in gray. Solid black lines with closed arrowheads indicate inheritance. Dashed red lines with open arrowheads indicate reference dependencies. A customizable legend with icons for each UML element type generated is drawn near the bottom of the diagram.

Additional UML elements that are not in this example include the following: (1) user's interfaces, colored cyan, (2) classes other than the user's and JDK classes (we refer to these as *external* classes), which are transparent, (3) dashed blue lines with closed arrowheads indicating interface implementation, and (4) solid green lines with open arrowheads linking inner and outer classes.

Although it may be useful to show the dependencies for Java and other library classes in some cases, often the users want to include only their classes in the diagram. UML generation settings available in jGRASP allow the user to separately exclude the Java Object class, Java library classes, external super classes, external interfaces, and other external references from their diagrams. In addition, under View, the user can selectively hide any

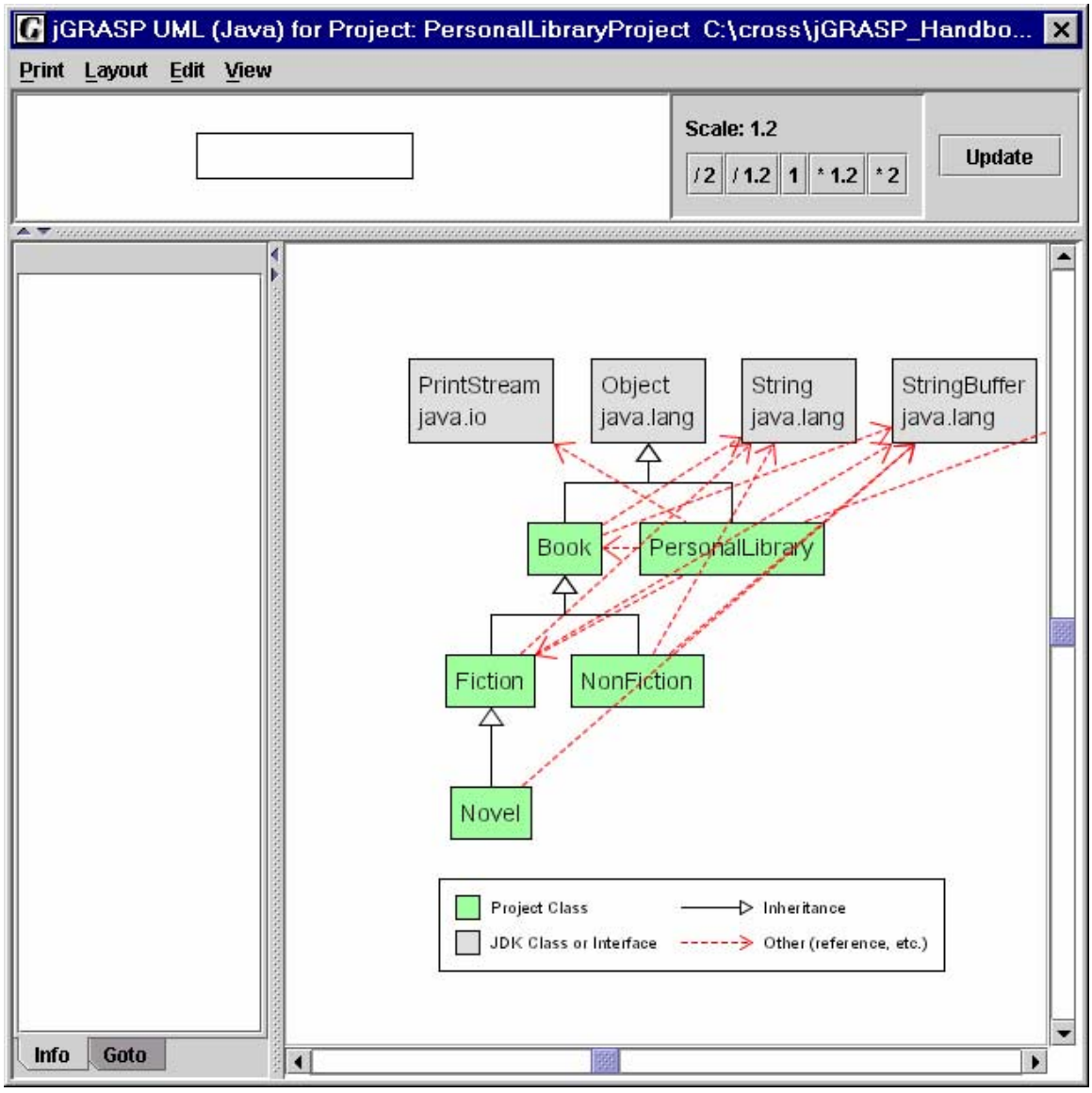

Figure 5. UML Window after initial Generate

object or dependency type. For example, the user could choose to view only inheritance dependencies by hiding all others.

The jGRASP UML window has controls that allow the user to scale the diagram to any size. A separate panning control makes it easy to navigate large diagrams. A diagram, or any portion of it, can be printed at any size, and large diagrams can be printed on multiple pages.

The UML diagram can be used to navigate the source code in jGRASP. For example, clicking on a class icon will open the associated source file for viewing or editing in a CSD window.

4.2 Refining the UML

Although the class hierarchy in Figure 5 for Book and its subclasses is automatically displayed using a tree layout, normally the user will want to fine-tune the layout by manually selecting and dragging some of the class symbols in the diagram. Once this is done, jGRASP remembers the layout from one generate/update to the next. After the user makes changes to the source code, including the addition of new classes, and recompiles the program, clicking Update redraws the dependencies, adding new classes as appropriate, but leaving the existing classes as the user placed them. Figure 6 shows a user-refined layout for the PersonalLibraryProject.

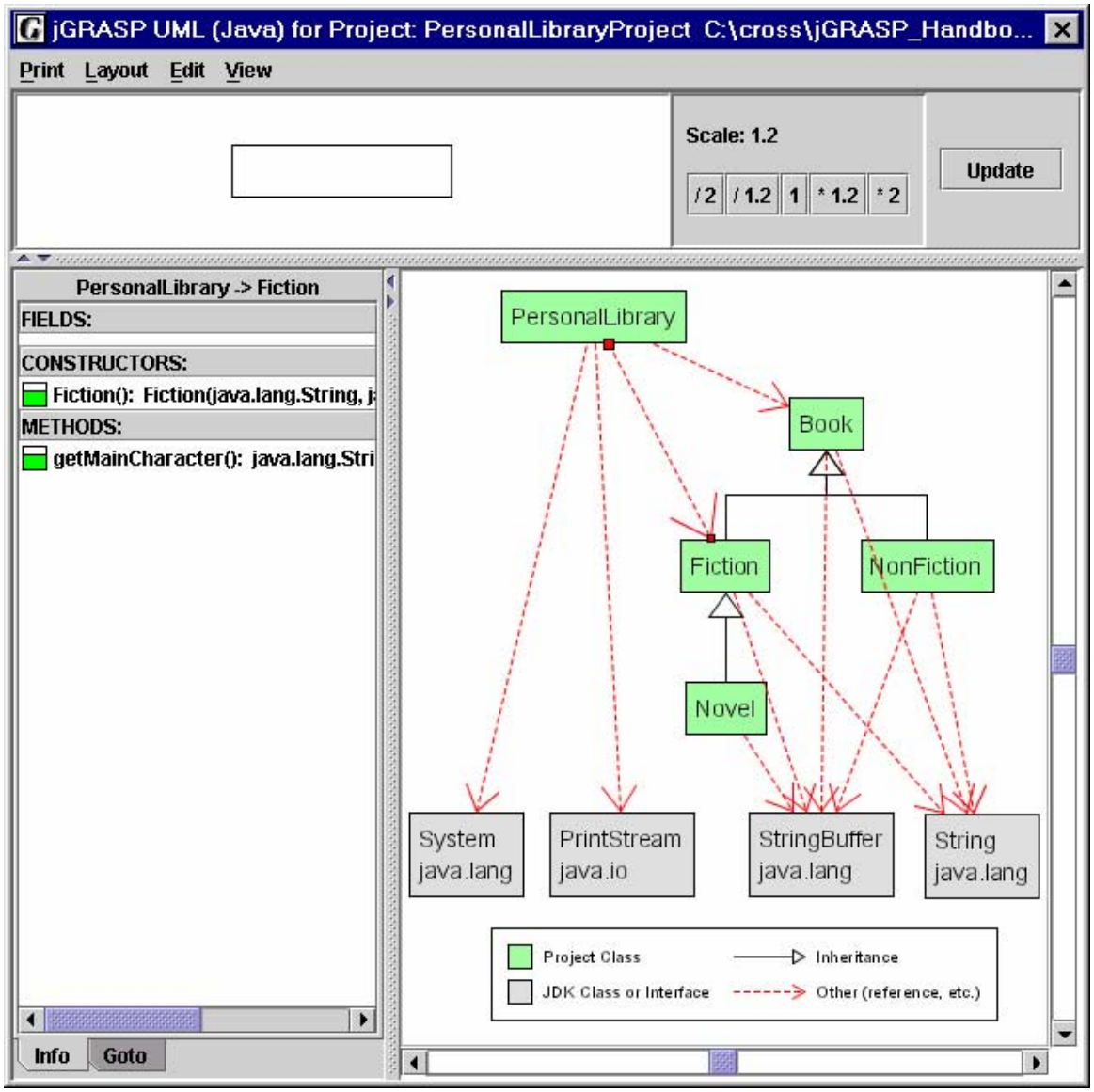

Figure 6. Dependency information

4.3 Displaying Dependency Information

The specific dependencies between any two classes can be quickly identified by selecting a dependency edge. In Figure 7, the edge from class PersonalLibrary to class Fiction is selected, and the specific dependencies between the two classes are displayed in the left pane.

5. Summary

Parsing Java class files is a reliable and straightforward way to find dependencies among Java classes. The Java compiler alters some dependencies through synthetic methods, but it is possible to determine the original dependencies by examining the byte code. Java class files also contain line number information that can be used to locate dependencies in source code.

UML class diagrams provide a view of the overall architecture of a program by showing all of its components and the dependencies among them. These diagrams can be used during development, maintenance, and reverse engineering. Generating and updating the diagrams automatically allows them to be integrated efficiently and effectively into the software process. An interactive user interface allows the diagram to provide more detailed dependency information. The jGRASP integrated development environment is freely available [3].

Acknowledgments

This work was supported, in part, by a grant from the National Science Foundation (EIA-9806777).

References

[1] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specication. The Java Series. Addison-Wesley, second edition, 1999.
Available online at URL:
http://java.sun.com/docs/books/vmspec/index.html

[2] Object Management Group. The Unified Modeling Language (UML) Specification, Version 1.4, September 2001.
Available online at URL:
http://www.omg.org/technology/documents/formal/uml.htm

[3] Graphical Representations for Algorithms, Structures, and Processes (Software and Documentaion)
Available online at URL:
http://www.eng.auburn.edu/grasp/

[4] Source Navigator (Software and Documentaion)
Available online at URL:
http://sources.redhat.com/sourcenav

[5] SNIFF++ (Software and Documentaion)
Available online at URL:
http://www.windriver.com/products/sniff

[6] BlueJ (Software and Documentaion)
Available online at URL:
http://www.bluej.org/