A Service-Based Web Portal for Integrated Reverse Engineering and Program Comprehension

A Thesis

Submitted to the Faculty

of

Drexel University

by

William M. Mongan

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Science

August, 2008

Acknowledgements

Many thanks to my advisor, Dr. Spiros Mancoridis, for years of support, advice and guidance in my research and development. Thanks also to Dr. William Regli, who has also supported me and provided me with a domain on which to test and apply my research. My acknowledgments to my thesis committee: Drs. Spiros Mancoridis, Bruce Char, Yuanfang Cai and Adam Fontecchio. Particular gratitude goes to Dr. Eli Fromm and Dr. Adam Fontecchio for supporting me as an NSF GK-12 Fellow throughout my graduate work. Special thanks to Christina Kirby and to the members of the SERG lab, especially Maxim Shevertalov and Boguste Hameyie, for bouncing ideas with me, for a neverending supply of "have you tried's," and for reviewing this document.

Dedications

To my family, who have supported me in all of my interests and endeavors. I have been truly blessed with their guidance; without them, life as I know it would not have been possible.

Table of Contents

List of Tables

List of Figures

Abstract

A Service-Based Web Portal for Integrated Reverse Engineering and Program Comprehension

William M. Mongan

Advisor: Spiros Mancoridis, Ph.D.

REportal is a web-based reverse engineering portal web site that provides developers with access to a suite of reverse engineering and program comprehension tools via a web browser. REportal was designed to simplify system maintenance by supporting the addition and upgrading of tools without involving the end user. However, the software tools and server technologies used became deprecated so quickly that it was not possible to take full advantage of the architectural vision. Using a service-oriented architecture, we abstract the process flow of the system from the underlying tools, enabling a wizard-style method of adding services to the system, and simplifying maintenance through automation.

This new architecture enables easy installation, deployment, and service management from the user's perspective, and easy service addition and portal maintenance from the developer's perspective. We conducted a case study involving the addition of a legacy tool to the portal as a service, and a description of the usability benefits of a web-based portal that integrates several features for software analysis.

1. Introduction

Practitioners are often placed into large software development projects that began before their involvement. These systems may have been programmed by a number of developers over time who are either no longer available, or are too busy to train the new developers. This, coupled with a lack of up-to-date documentation or formal specification, makes it difficult for new developers to assist on the project in a way that leaves it maintainable in the future.

As a result, the software engineering community has created various tools to aid in program comprehension and software architecture. These tools aim to generate reports or visual representations of software systems, thus reconstructing the as-built specifications of that software system in lieu of accurate paper documentation. Further, tools exist that analyze a system's architecture to extract design patterns, UML representations, or subsystem relationships within the system. This enables developers (or even end users) to discern the architecture of the system.

Although there exists a wide variety of tools enabling such software analyses, they typically represent a heterogeneous suite. Some tools have simple command-line pipe and filter architectures for a particular platform, while others have an elaborate API for some language. Moreover, many tools have overlapping features, though each tool has a proprietary output format. As a result, obtaining a comprehensive view of a software system requires considerable tool integration on the part of the user.

REportal[1] [53] is a Reverse Engineering portal web site that facilitates software architecture decomposition, program analysis, and understanding with a minimal amount of user intervention. The vision behind REportal was that the tool configuration would occur behind the application server, and that REportal would act as a thin client that exposed the features of those services. Tools were presented to the user as common tasks with a familiar web browser interface. The result

[1] http://reportal.cs.drexel.edu/

was a cohesive set of features that allowed for entity browsing, querying relationships in the system (for example, inheritance graphs and call graphs), performing transitive reachability queries using the results of relationship queries, and browsing the cross-referenced source code of discovered entities in the system, as seen in Figure 1.1. These features were exposed mostly through the CIAO [31] command-line Unix tool suite from AT&T Labs Research. A Java-based API for Bunch [52] was used to cluster these results in order to automatically discover software modules and subsystems, and GraphViz [38] was invoked to display graphical results in an applet view, as shown in Figure 1.2. Source code browsing and text search are available, and source code metrics such as Cyclomatic Complexity and inheritance tree depth are provided via additional tools.

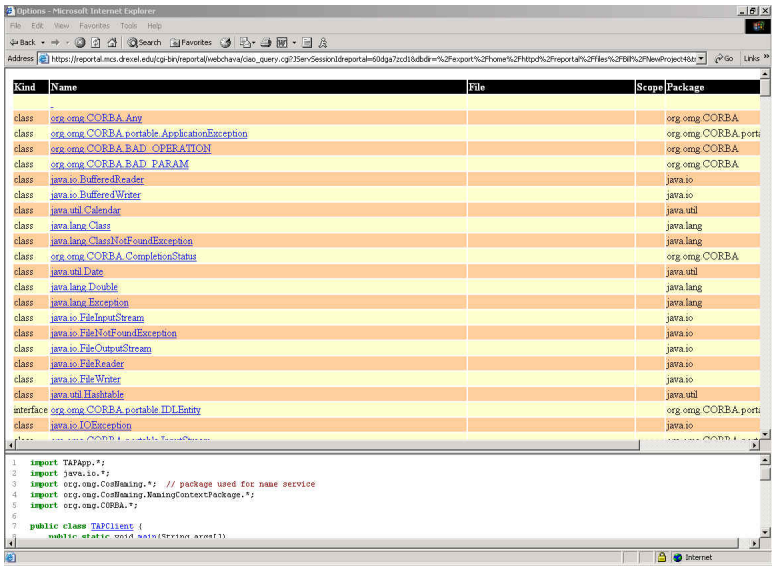

Figure 1.1: Example tabular report view of a software relationship query in the original REportal

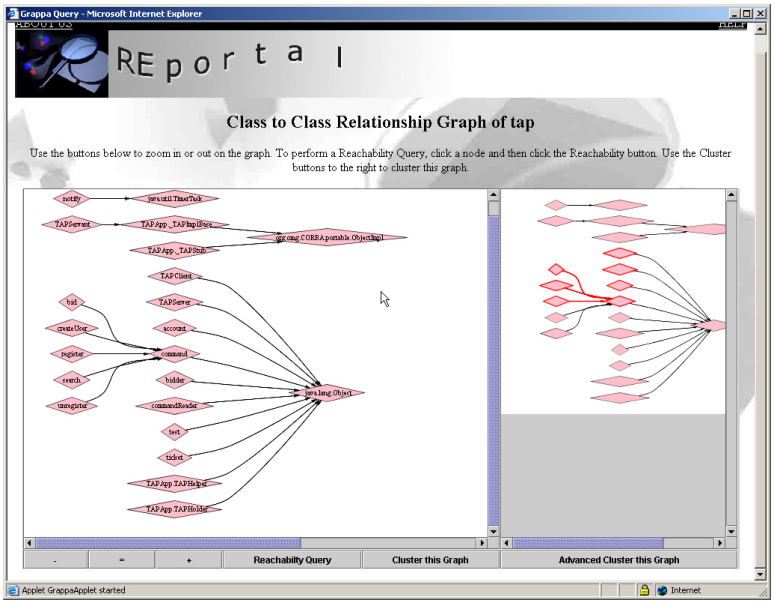

Figure 1.2: Example graphical view of a software relationship query in the original REportal

Despite the utility and ease-of-use benefits of the portal, there were a number of challenges pertaining to the maintenance and extensibility of the system. The original REportal was written as a single Java Servlet, which resulted in a tight coupling between user management, the tools, and the user interface. In addition, some of the primary tools are no longer maintained and, thus, do not effectively parse new language extensions or JDK versions. The system was designed with tool upgrade in mind, but the tight coupling of the portal to the tools and the interface made complete tool overhaul difficult. The portal was also tightly coupled to the host on which it executed. REportal tools had to reside on the same platform, and tool output was parsed without any adaptation, normalization or standardization. For example, the source code database repository tool would output a text-based report that was read in by the portal for web display. Any changes to the tool, or to its output, requires a change to the interface that reads it. In some cases, the tools themselves required specific and non-standard programs to execute. For example, the source code database builder tool required a specific binary build of `ksh` that was not publicly available. This made the

original portal difficult to install on other hosts. Survivability of the portal relied on few changes in the tools being used, and robustness of the host on which it is run. It was also overly cumbersome to install the portal behind a corporate firewall because the tools were so tightly dependent on the host environment. In addition, web interface tools were based on old `JavaScript` technology and, due to the nonstandard nature of browser rendering and display at the time, these technologies would fail or work on different browsers. As a result, it was also necessary to couple the portal to the most popular browsers, and modify its behavior based on the browser on which it was run. As a result of this coupling between the interface and underlying tools, it is difficult to exchange tools with new ones, or to upgrade existing tools to ones that evolve with the languages they analyze.

These challenges are mostly attributed to the underlying architecture of the original REportal and the technologies used. Because it is software engineering best practice to maintain a system that is robust to these issues; in light of the frequent changes and evolution of the underlying tools being offered to the user, it is necessary to re-architect the portal in a way that eases the integration of new or updated legacy tools and that eases the deployment of the portal on an arbitrary web server. To promote this easy extensibility as well as flexibility in the use of the underlying tools, we propose a distributed service Oriented Architecture (SOA) for the new REportal. This not only facilitates much easier maintenance and extensibility on the system, but it also serves as a microcosm that enables further research in automated service orchestration.

The redesigned REportal [57] offers services for each of the tools enabling changes to an individual tool without requiring extensive modifications to the architecture of the system as a whole. Because each service returns XML messages, it is easier to adapt new tools into the system, replace old tools with new tools whose services output similar XML data, and write heterogeneous (*i.e.*, non-web) clients for the portal. Each of these enhancements yields lower maintenance costs and ease of system comprehension. Furthermore, the initial redesign involves composing these services by hand; that is, the data production services such as the Static Analyzer and Dynamic Analyzer call the Bunch Clustering service to return a clustered graph. This approach was chosen because the tools were

not known *a priori*, and it was more efficient to start with a baseline before introducing refactorings to incorporate unknown services without prior knowledge. Moreover, because the tools are services, it is not necessary (though it is common) to host the tools on the same server that houses the user's projects and the presentation layer. Heterogeneous tools may be hosted on web servers on the platforms on which they run. The new portal architecture places fewer dependencies on the underlying tools or host. This automates installation and deployment, makes service relocation a matter of configuration, and makes service replacement or addition a matter of adaptation to an XML schema.

It is now possible to think of the tools as services that together provide a set of business tasks and solutions. These business tasks include commands like:

- What is the call graph through the system for feature X?
- What are the major subsystems of this system?
- What are some of the software design measures like Cyclomatic Complexity Number (CCN), comment rate, *etc.*?
- What is the runtime feature trace involving these classes?
- Who wrote each part of the code?

Because of the service-oriented architecture that was chosen to implement the new portal, the user interface is separated from the tools themselves. The user interface parses the XML results that are sent back from each service. Thus, changing services requires making a different service call and adapting that XML result instead. Once that data is obtained, it is possible to use it as the basis for calling other services automatically, which would not otherwise be immediately possible when running the tools manually. For example, a user's project can be automatically supplied to a forensics tool that determines code authorship. The portal could, for example, predict the most likely author of each file in the user's project from the set of other users' projects uploaded to REportal.

As another example, a runtime analysis feature provides call traces of one or more features of the user's program. However, this search can be focused to eliminate "noise," or insignificant library calls that would appear in the analysis. Since there is a static analysis feature built into the portal, that feature is executed and the result is used to assist the user in choosing which methods should be traced by the dynamic analyzer.

This approach provides not only a service-based re-architecting of REportal, which provides many decoupling benefits as described in this chapter, but it also introduces an architectural pattern by which tool-based portals can easily to be built. New tools and legacy tools can be integrated into a service-based portal, using the techniques described in this document. This integration previously required much more effort in our own experience hosting a tool-based portal, to such a degree that it was easier to rewrite the entire system to meet evolving needs. A successful portal architecture would enable tool addition and integration in a significantly decreased amount of time. As described in Chapter 6, the manual addition of a service to the portal has become an easier task; moreover, the exercise has made automated service integration a possibility (this is described in Section 7.2). This paradigm has positive implications for a number of stakeholders, including:

- **The software engineering community**: The software engineering community gains a portal architecture to promote program understanding through reverse engineering; this provides automated software engineering practices, tools and techniques to the general software community. All too often these practices are overlooked or underutilized due to their perceived complexity or cost.

- **Software developers**: Developers gain a tool suite through which they can analyze and understand software systems. This tool suite can grow and evolve with the tools, techniques and languages they support. A positive end-product of this effort would be a web "home page" like SourceForge or Google for software engineering.

- **Developers and users of heterogeneous tool suites**: Program comprehension through

reverse engineering was chosen as a specific domain for this tool suite because of its relevance to the software engineering community and to software developers. Apart from this domain, however, this effort provides a "best practice" for architecting a portal of decoupled heterogeneous tool suites, enabling simplified maintenance by those hosting the portal, and transparent invocation by the end user.

The rest of this document is organized as follows: we provide background and related work in Chapter 2; the user's and the developer's perspectives are described in Chapters 3 and 4, respectively; testing and validation are covered in Chapter 5; a case study on the ease of adding services to the SOA-based portal is discussed in Chapter 6; we discuss benefits of the portal and conclude in Chapter 7. For more information on Service Oriented Architectures, see Appendix A. Installation and deployment of REportal is described in Appendix B. Finally, the REportal service definitions are included in Appendix C.

2. Background

The current REportal is a re-architecture of an existing system with software maintenance in mind. However, it also relies heavily on the Service Oriented Architecture paradigm, which enables easier software maintenance, loosely coupled components, and a plug-in style framework for adding new tools or enabling web service standards on existing tools.

In this chapter, we provide background on each of these topics. We discuss the original REportal in Section 2.1, Service Oriented Architecture in Section 2.2, and Related Work on portals and research in SOA in Section 2.3.

2.1 The Original REportal System

The first version of REportal [53] was built upon Java Servlets, a Java web server technology that enables web applications to integrate presentation-layer code with business logic. In this way, it is possible to present a web page to the user while maintaining state information about that user. Thus, the REportal web pages are Java source code files that print HTML onto a web browser. Interwoven with this web display code are the REportal subsystems. Unfortunately, this inherent coupling between the presentation-layer display code and the business logic code caused a high degree of dependency between the portal and the tools, which proved too difficult to maintain in light of the rapidly evolving reverse engineering tools. The fundamental goal of REportal is to provide a heterogeneous tool suite to the user so that the user needs not install, learn how to use, or manage the underlying tools. The portal provided the core functionality of each tool in an intuitive web interface, greatly reducing the learning curve for performing reverse engineering; however, it merely shifted, rather than reduced the burden of tool maintenance to the REportal developers.

Re-architecting REportal into a Service Oriented Architecture (introduced in Section 2.2 and described in technical detail in Appendix A) reduced the burden of individual tool maintenance in

the following ways:

- **Tool Upgrade**: As tools evolve, it is necessary to update REportal. Under the original architecture, this required changes to the presentation layer and to the business logic layer, which is akin to a complete rewrite of the feature.

- **Tool Replacement**: Many of the tools provided through REportal are open source from the research community; therefore, they tend to change ownership or have ambiguous ownership. As a result, they may not undergo the maintenance required to keep them current with existing languages. As a result, it becomes necessary to seek or develop new tools to provide the desired functionality. These tools often have new interfaces and data models; because the data model was coupled to the business logic, and thus to the presentation layer, this too resulted in a complete rewrite of that feature in REportal.

- **Packaging and Deployment**: REportal, as a Java Servlet web application, was easily packaged into a single jar file for distribution. If one could simply package the tools along with this jar, the entire system could be deployed instantly. However, the tools themselves contained dependencies. Particularly, many of the tools were shell scripts or Linux binary applications that relied on rare proprietary versions of Korn Shell. As a result, one had to distribute this particular binary of Korn Shell, which restricted the types of platforms that could run REportal. Because REportal's presentation layer was coupled to the tools, it was not possible to deploy the jar on a separate server from the tools; therefore, the entire system was restricted by this dependency. Moreover, Java support for executing shell commands was new and unstable, causing some tools to lock up during execution or leave runaway processes on the host that were unstoppable from within REportal (which wasted memory and CPU resources). Several corporate clients of REportal wanted to analyze proprietary systems behind their corporate firewall to maintain trade secrecy; however, this was not feasible under the original REportal architecture, given the tools and resources available at that time.

- **Tool Addition and Scalability**: Adding tools was not as difficult, as this required simply adding a new set of classes to the portal front end, and integrating them with the presentation layer for display. Then, the tool called or invoked the process that performed the desired analysis, returning its result to the presentation subsystem. However, it was difficult to integrate new tools with existing ones, enabling a more comprehensive analysis report than the user could generate manually. A decoupled data model would enable REportal to adapt tools to accommodate the integration of their features.

The new "REportal 2.0" [57] described here concerns itself less with the reverse engineering tools and services provided by REportal; rather, it aims to provide an architecture for developing a web portal of services, using reverse engineering as a relevant domain example.

2.1.1 Original REportal Architecture

In this section we describe the original features offered by REportal, and a brief overview of its architecture.

Ciao Source Code Static Analyzer

Ciao is a graph-based source code analyzer for Java, C and C++. This type of source code analysis is referred to as **static analysis** because it is performed on the source code only (a static source), rather than by observing the behavior of the program or system during its execution. Ciao [31] produces a SQL data repository using information from the source code. This database is then queried to find system entities and the interrelations among them. It is also possible to create Bunch-compatible module dependency graphs using a helper tool that executes a sequence of Ciao queries. Ciao is queried via command-line tools that enable users to determine class inheritance hierarchies, data flow, call graphs, and reachability (or dependency trees). These views show developers how a change to one part of the system might impact other subsystems.

Bunch Clustering Tool

Bunch [52, 56] is a clustering tool intended to aid the software developer and maintainer in understanding, verifying, and maintaining a source code base. Bunch relies solely on the information contained in a Module Dependency Graph (MDG) file, considering nodes as program units or modules, such as files or classes, and edges between the nodes as calls or relationships between those modules, such as function calls or inheritance relationships. An MDG is a directed graph created based on queries to the source code repository generated by Ciao (see Section 2.1.1) via a shell script. With this graph, Bunch finds what a "good" clustering for the system is (thus helping when documentation of the code is nonexistent or outdated), and it can also use pre-defined clusters to measure or improve the quality of the system's clustering.

Bunch processes the MDG and produces, as output, a clustered graph that includes all of the original source code and dependencies clustered into logical subsystems. The output from Bunch can be visualized by several popular graph visualization tools such as Dotty [48] and Tom Sawyer [35]. REportal forwards the graph output from Bunch to Dotty for visualization.

Metrics Report Tool

REportal provides an internally developed metrics report to show developers parts of code that might need maintenance attention. For example, undercommented code or overly complex code is measured by employing static analysis techniques. A method with many control structures like loops, switch statements, and conditionals contains more paths of execution than a simpler method and, therefore, is more prone to bugs or unanticipated execution paths.

Text Search Tool

Simple text-search primitives based on the *grep* utility are provided on the source code base. REportal shows all instances of a specific string or pattern, along with the file in which it appears and the line number. This is useful to detect simple (and very common) instances of code cloning.

Source Code Decompilation

Because many of these tools use static source code analysis techniques, it is more important to upload a system's source code for analysis rather than the compiled binary. In a situation where binary code is all that is available, REportal automatically decompiles Java bytecode into its equivalent and near-original source code representation (except for comments). This enables source code analysis when only binary Java byte-code is available.

Example: Generating an MDG Graph from Java Code

To understand the MDG format generated and used by most of the tools in the original REportal, consider the following simple Java example: two source files are given: `First.java` in Table 2.1 and `Second.java` in Table 2.2. In this example, we manually identify dependencies between the classes to generate an MDG representing only dependencies between classes (that is, a dependency between a class and a method will be represented a dependency between that class and the class that owns the method).

Table 2.1: Simple Java code example: `First.java`

```
 1: public class First {
 2:    private static int theValue;
 3:
 4:    public static void main(String[] args) {
 5:       theValue = 5;
 6:
 7:       Second theSecond = new Second(theValue);
 8:
 9:       theSecond.helloWorld();
10:    }
11: }
```

In this code, it can be seen that the `First` class invokes a method in the `Second` class. Specifically, line 9 of `First.java` invokes the `helloWorld` method in `Second`. But it is also the case that the

`First` class instantiates a new object of type `Second`, resulting in a call to the constructor for `Second`. This is an additional dependency between `First` and `Second`. In the MDG format, we represent an `Invoke` relationship between the `First` and `Second` classes as `First Second invoke`. Thus the MDG is a very simple file format, that nonetheless represents a labeled directed graph.

Reading on, we find no invocations in the `Second` class to `First`, but rather between `Second` and the Java `System.out` library, via a call to `println` on line 10. However, there really exists two dependencies here: one to the `java.lang.System` class, and the other to the `java.io.PrintStream` class. Although it is not shown explicitly here, the `System.out` class is actually of type `PrintStream`. What has actually happened is that `Second` has obtained a reference to the `System` object, and then invoked a method (`println`) of one of its objects (`out`). These dependencies are represented as a `get` dependency between `Second` and `java.lang.System`, and as a `invoke` dependency between `Second` and `java.io.PrintStream`.

Finally, both the `First` and `Second` classes invoke initialization code common to all objects and stored in the `java.lang.Object` class. This is, therefore, a dependency between each class and `java.lang.Object`, which is expected in the Java language. Adding these dependencies, we are left with the MDG file shown in Table 2.3. The corresponding graphical representation is shown in Figure 2.1.

This was a simple example in a number of aspects: first, it only considered dependencies between classes, and not between classes and variables, classes and methods, methods and methods, and so on. These were all combined and abstracted into inter-class dependencies, placing these dependencies at a very high level. More detailed analysis would clearly take more time. Moreover, there were only two classes in this example, which hardly makes it a candidate for automated reverse engineering for program comprehension. Nevertheless, a few points are illustrated: first, we described how to construct an MDG from source code and what is contained therein, and second, we illustrated that, even for simple code, it is possible to overlook some dependencies that a machine-generated analysis would not miss.

Table 2.2: Simple Java code example: `Second.java`

```
 1: public class Second {
 2:    private int numTimes;
 3:
 4:    public Second(int _numTimes) {
 5:       numTimes = _numTimes;
 6:    }
 7:
 8:    public void helloWorld() {
 9:       for(int i = 0; i < numTimes; ++i) {
10:          System.out.println("Hello World!\n");
11:       }
12:    }
13: }
```

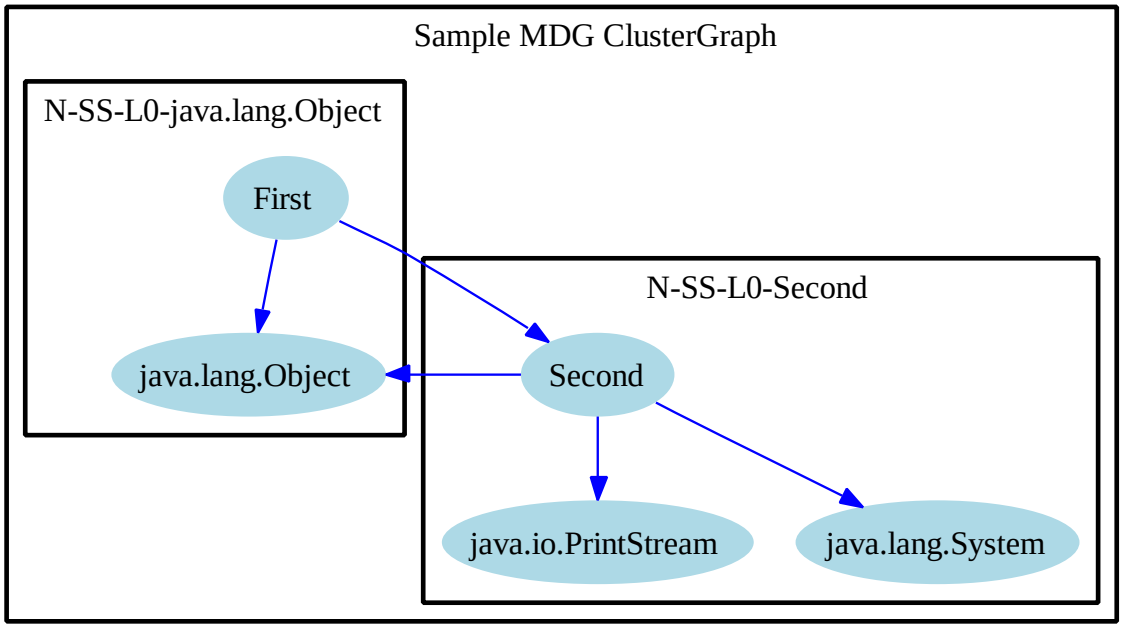

Figure 2.1: The graphical display of the MDG depicted in Table 2.3

Table 2.3: The Module Dependency Graph (MDG) representing dependencies that exist between `First.java`, `Second.java` and the standard Java libraries.

```
First java.lang.Object invoke
First Second invoke
First Second invoke
Second java.lang.Object invoke
Second java.lang.System get
Second java.io.PrintStream invoke
```

2.2 Service Oriented Computing

Service Oriented Computing (SOC)[1] is a movement toward an implementation-independent architecture for distributed computing. "Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains" [59]. Rather than an object-based design and functionality, SOA raises the level of abstraction toward higher level business logic. For example, consider the following classic scenario [59]:

> A small company is currently using separate in-house proprietary systems to process orders, charge credit cards, check inventory and ship products. Data is exchanged from one department to the other via hand-written e-mails, though the inventory and shipping departments were recently integrated and sharing messages using a proprietary format over TCP.

Although SOA allows for a bottom-up design that is amicable to the integration of legacy systems, the real benefit is realized from a top-down, implementation-independent, approach. By using XML schemas to define abstract data types and an XML document to define business-logic operations in terms of these abstract types, a complete interface is derived for a system that naturally hides implementation details and business secrets.

In this example, departments only expose the highest level business logic that each needs to share. For example, the shipping department would only need to know that the credit card transaction was approved and the shipping address provided by the customer. It would not need to know the

[1] For a detailed introduction to SOA and its associated technologies, see Appendix A.

amount of the transaction, the credit card number, or how the sales department actually verified the information. Similarly, the sales department does not need to know any implementation details about the shipping department. Services provide only their public interfaces to one another, and through their design, this information hiding can be achieved automatically. Through the SOA's design, this information is filtered from the passed messages as they travel from service to service, or hidden completely through selective XML encryption. Moreover, if the shipping department was outsourced to a third party company, only the service contract and details limited to implementing the service contract need to be changed. Naturally, that third party shipping company does not wish to expose its private information through its shared business processes, such as its sub-contractors and costs, and this information would also be hidden by the service contract through the design of the SOA. The extent to which a service's functionality is exposed for use is called its **visibility**.

When we think of SOA in its most recent stage, we tend to think first of web services – a rather young platform. While the web services paradigm has gained significant popularity in both the software architecture and the IT infrastructure camps, SOA predates web services and is implemented by a number of existing technologies.

Noteworthy architectures such as CORBA and RPC may cause some to wonder why SOA is considered an emerging paradigm. It is easy to interpret concepts like web services as simply the latest trend. Service descriptions using RPC can be found in CORBA using the Interface Definition Language (IDL) specification [58]. Like XML based services, IDL is language independent and allowed for a heterogeneous implementation. Although the XML messages used by SOA web services are verbose, while IDL specifications more closely resemble Java interface definitions, it is easy to generate compliant service invocations, with or without the aid of code stubs or middleware. Regardless of the underlying architecture, services are easily designed and provide a clear separation between message passing, transport, and functionality. In addition, they separate public business functionality from private business secrets and implementation.

In truth, SOA consists of a number of components and supporting technologies. In its most recent incarnation, SOA technologies include XML Schema (Described in Section A.1), WSDL (Described in Section A.2), SOAP (Described in Section A.2.5), UDDI (Described in Section A.4), standards for QoS and security, and so on. These are described in detail in Appendix A.

The point is that while, in this generation, web services are often synonymous with SOA, SOA is an architectural style that can be realized in a number of ways [68].

To provide a high level overview of services, their relationships and role in software development, the OASIS Committee on SOA released the SOA Reference Model [59]. The SOA-RM aims to provide an architecture and technology independent description of SOA, services, policies and descriptions.

2.2.1 Advantages of SOA

Services are popular because they are discoverable, composable and dynamically bound; they are modular, self-contained and loosely coupled; and they can be easily moved from host to host without disrupting availability [58]. Services "emphasize a distinction between a capability and the ability to bring that capability to bear. While both needs and capabilities exist independently of SOA, in SOA, services are the mechanism by which needs and capabilities are brought together." [59]

2.2.2 Data Flow

Data flow through an SOA system is also described by a structured XML representation and is implemented through some type of message passing. There are many implementations of such message passing in existence, including Message Oriented Middleware (MOM) packages and the Simple Object Access Protocol (SOAP), but this is a transport level implementation detail that is not prescribed by SOA itself.

2.2.3 Service Invocation

Method calls are based on XML documents. An XML document is formed and addressed to the service being hosted. The content of the document is a structure that matches the structure and contains all the data parameters mandated by the service's contract. The service receives that message, executes the method that implements the service, and returns the result in another document as prescribed by the service contract. In either case, the underlying transport is often transparent to the programmer, as the communications implementation is often automatically generated, leaving a code stub that the programmer must implement and call as a normal object method.

2.2.4 Implementing the Client and the Server

Once the WSDL file is formed, a server and/or client implementation may be created. When we consider the usefulness of web services, we see that it is often the case that the client and server are implemented by two different parties [42]. Because WSDL forms a service contract between a client and a server, client code can be generated that conforms to the WSDL and calls the appropriate web service. After all, the WSDL contains all the information about data types, messages, methods to call, parameters to pass and expected return values. Finally, the `<service>` section also shows the physical location with which to interact and communicate. It is possible to obtain the WSDL of a deployed web service from the application server on which it resides; therefore, it becomes straightforward to write a client to interact with service providers such as eBay, Google, Amazon, and so on.

All of this holds true for server implementation as well. As a service contract, the WSDL says nothing about the implementation details of the system. Nonetheless, a great deal of implementation can be generated from the WSDL alone.

It turns out that tools exist to generate this code on both sides automatically. Apache Axis, .NET, and others provide tools to generate code stubs that conform to a given WSDL definition. The developer must simply provide the implementation of that stub, which is essentially the private

algorithms used to execute the advertised functionality.

This idea gives rise to a few design strategies:

Contract-Last Design Because it is possible to generate code from an existing WSDL definition, it stands to reason that one could also generate WSDL from existing code. This turns out to be the case. This is a good strategy for legacy systems, but contains the pitfall that developers will ignore the all-important service design phase in favor of making any and all functionality within an object-oriented system into web services. This is a bottom-up design.

Contract-First Design This strategy is the one described so far in this document and in most literature. The WSDL is defined first, followed by the code stubs. This is a top-down design and is ideal for new services to promote information hiding and a separation of concerns.

Meet-In-The-Middle This approach is meant to combine the best of both strategies, and involves generating the WSDL from existing code stubs (for example, data types), and then hand-writing the remaining WSDL to generate the final code implementation. This avoids the need to write verbose XML Schema definitions of data types by hand.

Independent of the development path chosen, it is easy to separate the private business logic from the public business processes and services being offered. This has been made clear by the advantages of WSDL design previously discussed. It is possible to write the private business logic as a standalone enterprise application, and then link it to the generated web service via an adapter layer. The role of the adapter layer is to take the data from web service requests (this data is packaged in messages typed according to the WSDL), convert it into its individual parameters (or pass it along as a whole), and call the enterprise application functionality. The same is done with the values returned from the enterprise application. The benefit here is that if the web service ever changes (for example, new functionality is desired, new parameters are needed, or data types change), only the adapter layer needs to be changed on the server end.

The client would be re-generated from the new resulting WSDL, and an adapter layer on the client application would provide a convenient buffer from these changes as well. In the case of the client, a simple JSP based frontend could be created that calls the web service directly. The generated WSDL client code is often so transparent that the invocation often appears to be simply a local method call.

2.3 Related Work

For many, re-engineering a software system into an SOA translates into opening an existing system into an IDE, and invoking the "create web service" feature to create a highly customized and proprietary "service" that simply executes the underlying code.

This type of "automated service creation" is not the spirit of this work. In fact, users of this "create web service" IDE feature often create web services for each class in their system. Our project, instead, aims to create web services that wrap around entire legacy systems; exposing the most coarse-grained functionality (for example, the behavior of the `main()` function routine) as a single web service with a relatively small number of service interfaces. To understand this process better, we use the reverse engineering domain (and the existing REportal system) as our case study. In this section, we describe related research in this area.

2.3.1 Legacy System Migration to Service Oriented Architectures

The idea of migrating a legacy system to an SOA has been explored in the literature. For example, one related project provides a mechanism to migrate grid application components to web services automatically [36]. Another introduces a wrapper technique similar to our own for migrating legacy systems to an SOA. Their approach uses a finite state machine and a terminal emulator to allow for the migration of interactive tools such as the Pine mail client and text editor [29]. Our goal, however, is to migrate legacy distributed systems, or to migrate a number of legacy tools into a cohesive distributed suite of services, and then create a business process that finds, selects,

coordinates and executes those services on the fly. The multi-server distributed approach is selected for two reasons: primarily due to the heterogeneous nature of the tools, but also to support parallel execution of the tools in question.

The idea of exposing tool functionality, however, is not a new one. The Purdue University Network Computing Hubs (PUNCH) [43] project exposes computer architecture simulation and design tools for education via a web portal. Like the original REportal, the spirit of PUNCH is to provide access to these tools regardless of the user's physical location or available computing resources. The user simply needs a web browser, and the tools themselves are executed primarily on the server(s), which may be distributed without the user's intervention. PUNCH works by specifying the inputs required by a particular tool, and how to map those inputs to the tool's interface, via HTML and CGI "templates" specified by a high level language. Process flow is enabled by a finite state machine structure. This is very similar to both the old and the new REportal; however, the SOA-based REportal enables scalability to utilize new services by searching UDDI registries for new tools to which to bind. In this sense, it is not necessary to deploy a tool to the system, but instead to point the system to the tool. Further, while PUNCH allows for "cross-enterprise" tool deployment, it is not clear if PUNCH supports a heterogeneous tool suite, including Windows-based applications as well as UNIX tools.

Legacy migration to SOA is also discussed by Sneed [65, 64] and Lewis [51]. Sneed points out that new architectures such as SOA cannot be successful if they cannot "take [legacy] programs along." The problem is compounded by the use of programming languages that have dependencies on the user environment; particularly, those that also provide the runtime environment. Despite this complexity and dependency, it is desirable to create services that are simple and easy to use. He proposes an automated tool that wraps legacy code in WSDL that exposes the high level features, and then the creation of higher level, more abstract and simple services (based on features) that invoke them. Lewis describes the Service-Oriented Migration and Reuse Techique (SMART) [51]. SMART is a methodology for migrating legacy systems to SOA, involving the identification of

stakeholders and functionality, the trade-offs between certain design choices including middleware and architecture, and finally planning the migration to an SOA.

2.3.2 Software Visualization Tools for Program Comprehension

Standalone software visualization tools such as CodeCrawler [50] also exist that explore the structure of the software system as a whole. The Visualization Architecture for Reuse (VARE) [26] system creates an XML repository of a software system, which they query to create Scalable Vector Graphics (SVG) that visualize that system. REportal, however, aims to use an existing tool suite to generate and to standardize the software repository, then query and view the result in an interactive view. The XML Data Storage Environment (XDSE) [25] integrates with VARE and is of particular interest to our work, because it proposes an XML format for storing program traces, and a model for querying that XML repository using XQuery. The XML query engine is exercised using a web service and JSP client. This idea is consistent with the spirit of REportal as a whole, and REportal provides a number of tools for program understanding and exposes them as services. As such, XSDE and VARE are likely good service components integrated and exposed by REportal.

Other research in automatic software modularization includes work that clusters based on data use by subsystems [41]. Data bindings are identified in which a module writes to a variable that is read by another module. This idea serves as the basis for identifying potentially interesting subsystem interactions for clustering purposes. This technique can be applied to legacy systems to simplify program understanding and software maintenance, by identifying subsystems that contain clusters of entities that should be inspected when making modifications to a particular software module [69]. REportal uses Bunch [52] as an implementation of software clustering because it creates a hierarchy of clustered graphs as an XML repository that is easily queried by the visualization tool.

2.3.3 Service Integration

Finally, there is new research in the area of service integration via Service Component Architecture (SCA). SCA has a number of emerging implementations, including Apache Tuscany [4], and allows for the integration of existing services via service metadata that describes their interfaces. The Java Orchestration Language Interpreter Engine (JOLIE) [10] also provides a high level language for service orchestration via the WS-BPEL standard. It is a goal of REportal to contribute to this area of research by providing a baseline and a case study for service integration and orchestration.

3. User Perspective

In this section, we describe the core tools that have been wrapped into REportal services. These are represented by the services depicted in Figure 4.1. The approach taken in designing the REportal use interface is that only the highest-level features of the tools should be exposed. Users of REportal are interested in answering specific questions or gaining a general understanding of a software system. They need not spend time configuring custom attributes of the tools, as their primary objective is to obtain reports with predictable settings. Therefore, REportal consists of features that are report-oriented, or goal-oriented; it is not intended to be highly customizable (although it could easily be made this way, since the underlying tools present numerous customization options). The goal of this approach is that it is straightforward to exercise the common features of REportal.

3.1 User Management

The user management features are largely logistical services provided by REportal; as such, they are mostly transparent to the user. However, there are a few features that are invoked by the user in order to gain access to the portal.

First, the user must log in to REportal. This is outlined in the use case depicted by Figure 3.1. The user's password is encrypted and, along with the username, compared to the database. If the credentials match, the user is permitted to log in. Otherwise the user is notified of the failure and returned to the home page. Every page of REportal begins by checking the user credentials and forces the user to login if the credentials do not exist.

The user may also register at this screen by providing personal information that is stored in the database. Currently, this information is not used except for the username and password. The User Registration use case is shown in Figure 3.2. Users may log in and register on the same screen: the screenshot is given in Figure 3.3.

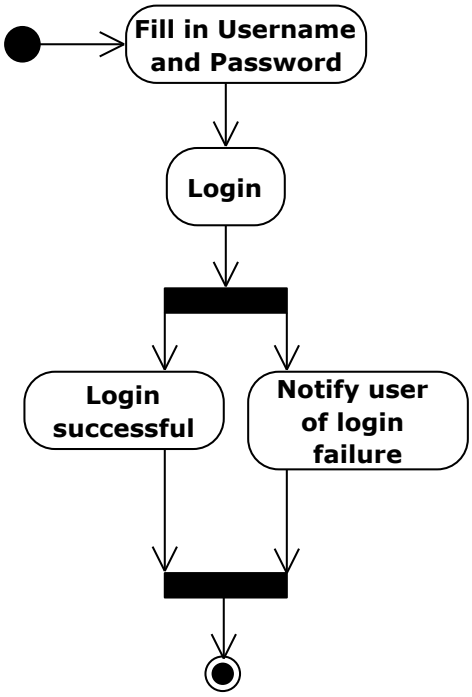

Figure 3.1: The REportal User Login use case

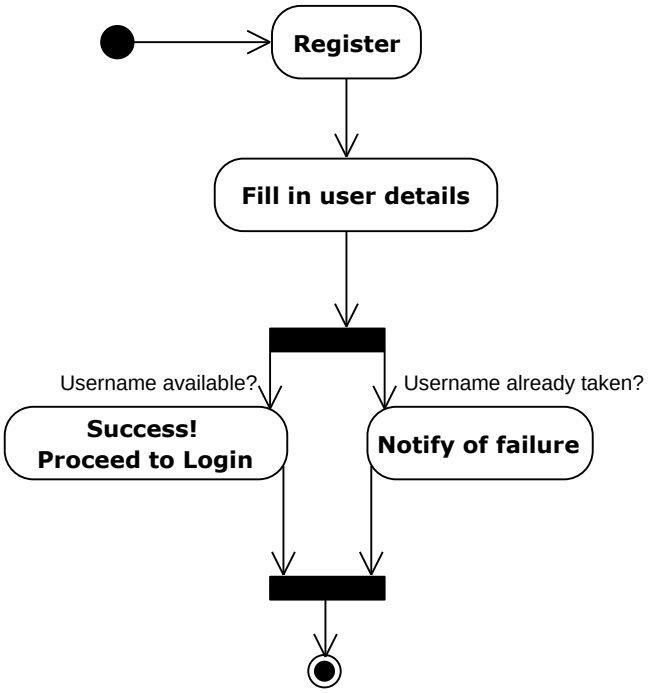

Figure 3.2: The REportal User Registration use case

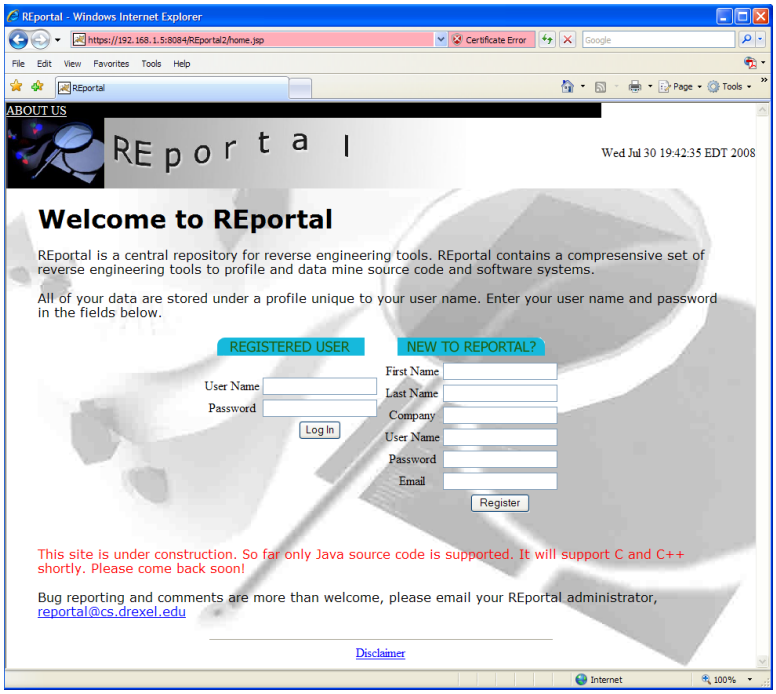

Figure 3.3: REportal Login screen

3.2 Project Management

Once the user has logged in, the user is greeted with a portal of projects owned by that user (see Figure 3.4). The user may add new projects, analyze existing ones, delete them, and so on.

REportal automatically adds a *demo* project for the user when the user first registers for an account. However, it is expected that the user will wish to add a project soon after logging in. There is a link on the projects page that enables the user to add an empty project. The process is shown in Figure 3.5, and a screenshot is given in Figure 3.6.

The user may specify a project name and language (C, C++ or Java). Immediately upon adding the project, the user is presented with a link to upload code to the project for inspection. The user may select a *ZIP* or *JAR* file to upload, and the file is unzipped into the user's directory for manipulation on the project page. The use case is shown in Figure 3.7, and the screenshot is given in Figure 3.8. If the file is unzipped successfully, the user is notified and taken back to the project

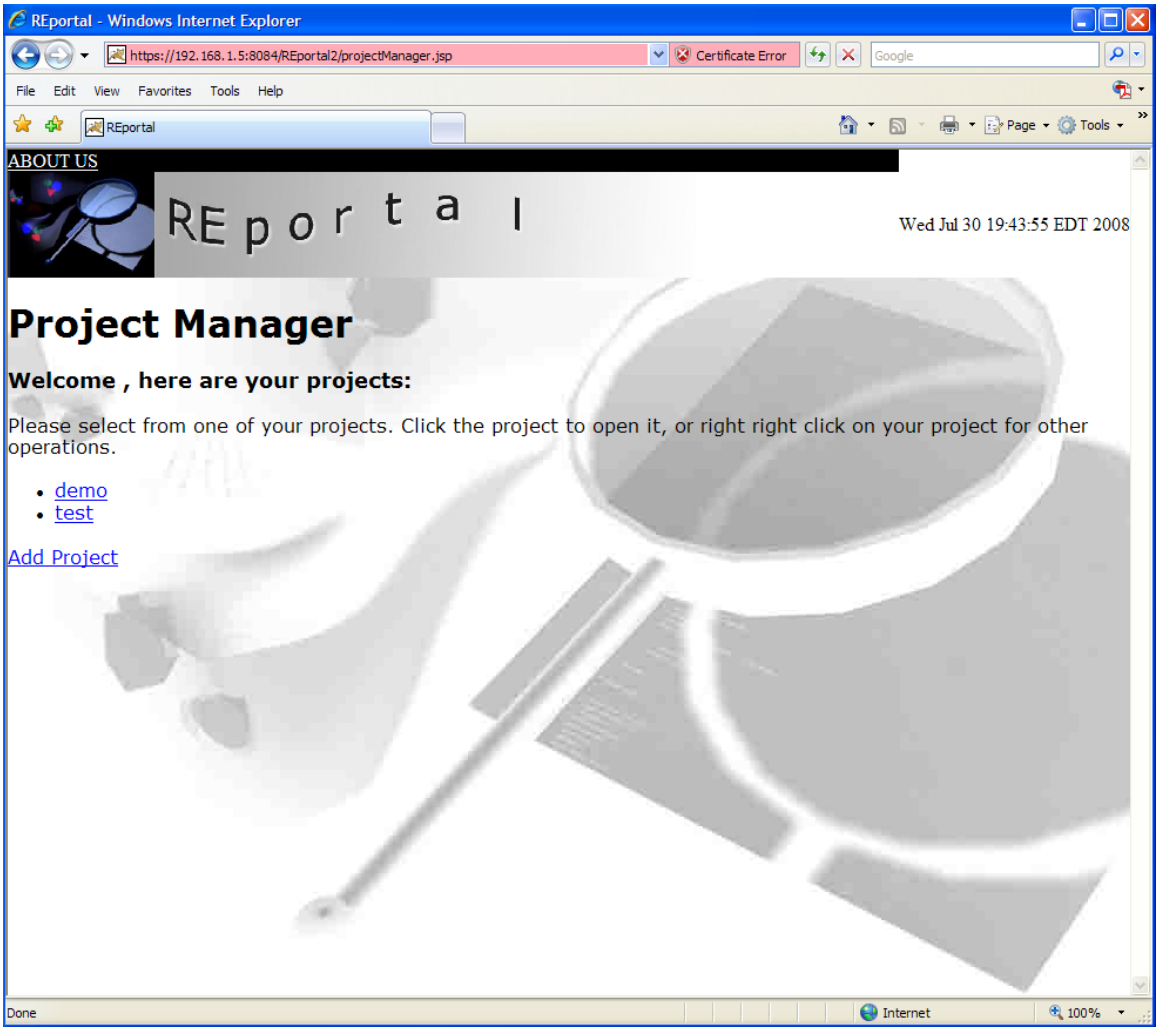

Figure 3.4: REportal Project List screen

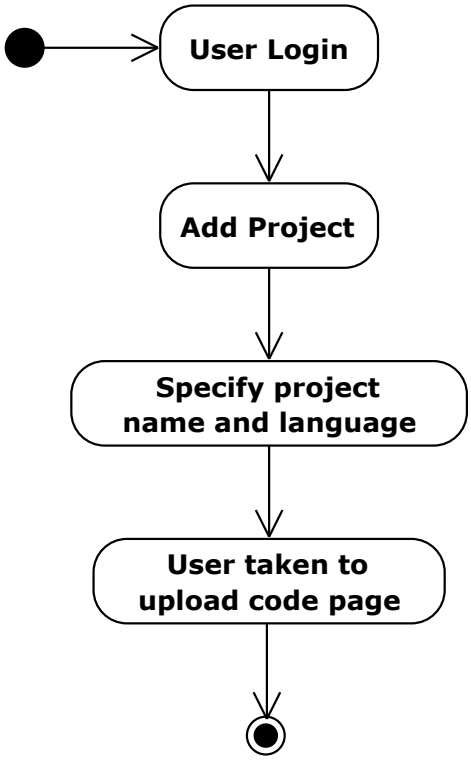

Figure 3.5: The REportal Add Project use case

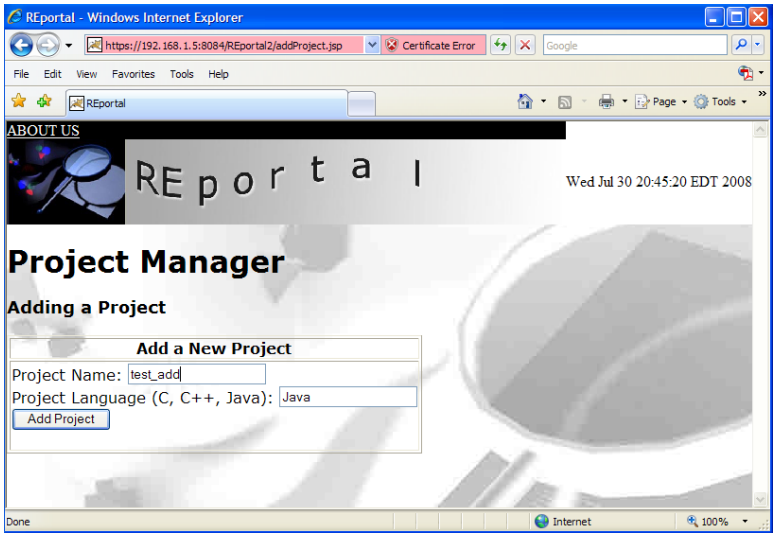

Figure 3.6: REportal Add Project screen

page.

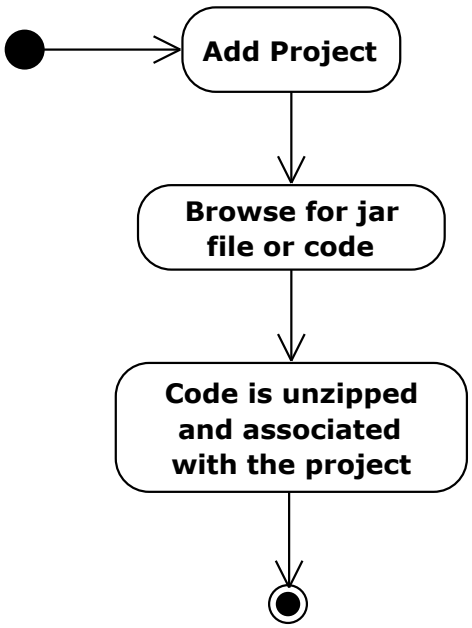

Figure 3.7: The REportal Upload Project Data use case

Now, the user may either remove or open the project. To remove the project (or perform other administrative options), the user may right-click on the project, as detailed in the use case in Figure 3.9. When the user opens the project, the user is presented with the suite of tools made available by the portal, as shown in Figure 3.10.

In the event that a non-source jar is uploaded to REportal, some features like the source code browser and text searching will not return any meaningful values. For this reason, REportal invokes the Jode [9] Java decompiler to decompile the classes in the jar when the project is first opened. The files are placed into the user's project directory so that decompilation need not happen again.

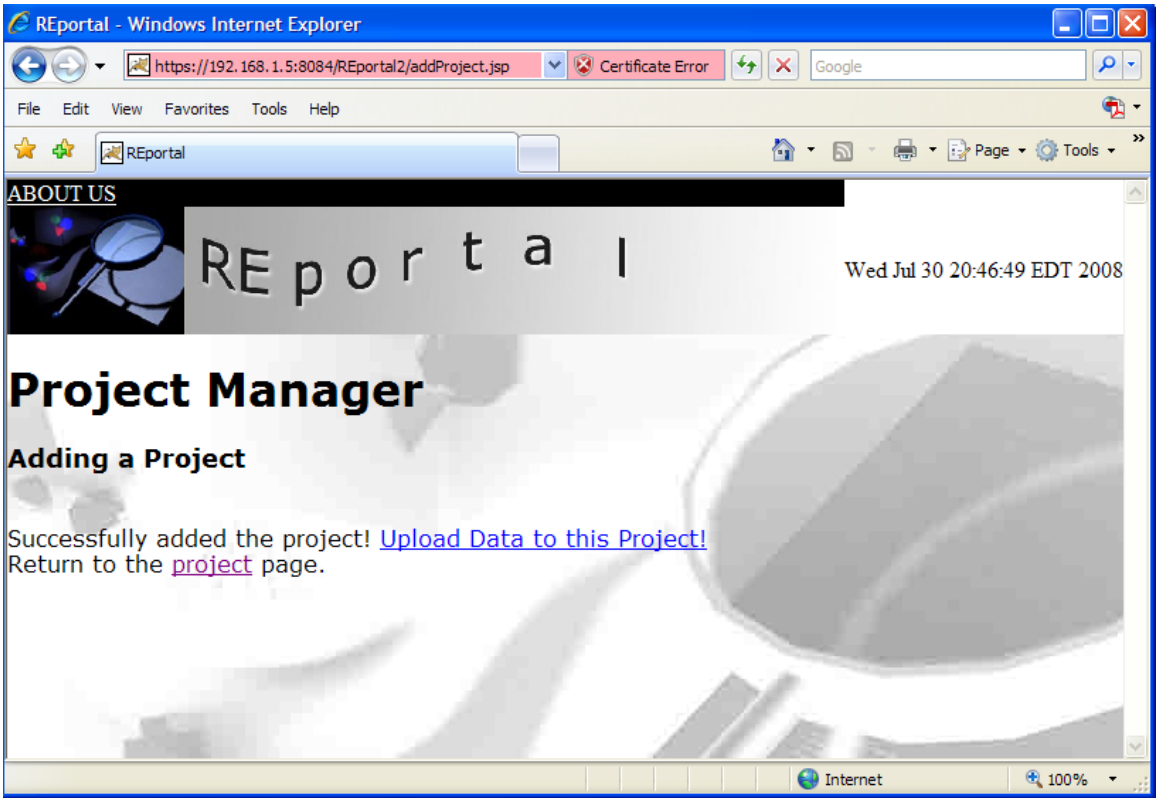

Figure 3.8: REportal Upload Project screen

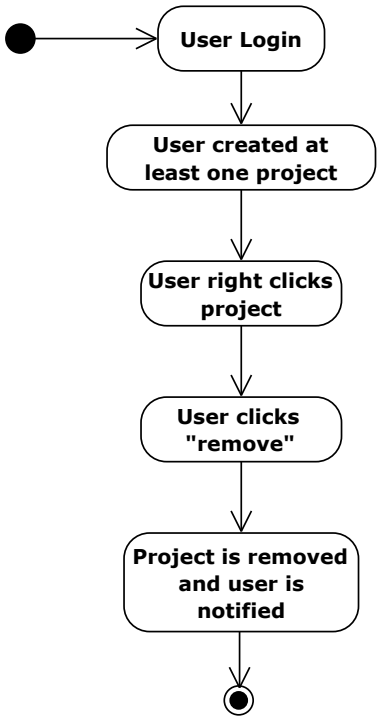

Figure 3.9: The REportal Remove Project use case

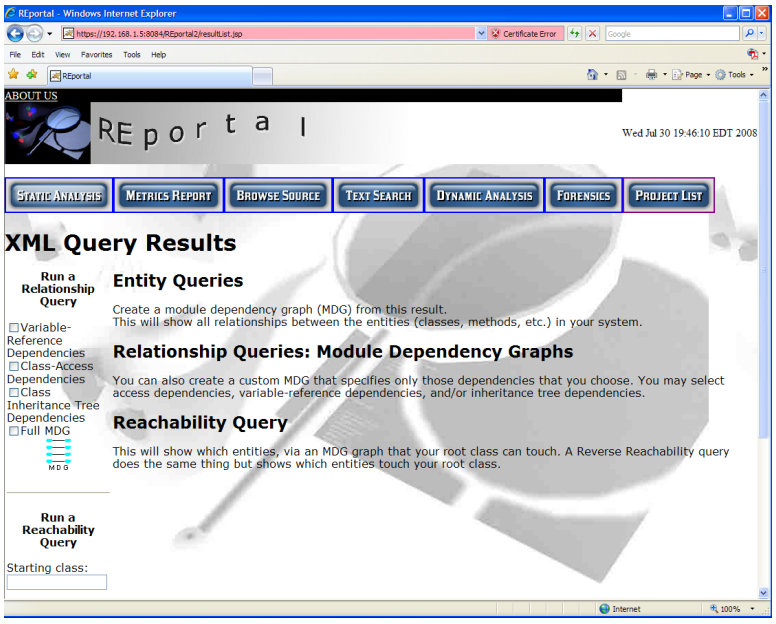

Figure 3.10: REportal Open Project screen

3.3 Source Code Browsing

One of the first things a user may wish to do with a newly opened project is inspect its source code. The Source Code Browser provides a cross-referenced and annotated view of the source code. The Source Code Browser displays source code such that entities within the system are hyperlinks. Variable references or method calls can be clicked on, taking the user to their definition and implementation elsewhere in the project. Clicking on the arrow to the right of the entity brings up a context menu (see Figure 3.12). This context menu allows users to obtain entity usage information at the bottom of the display, including which classes call a method or use the variable, *etc.* Entities are listed in a column on the left side of the display, and are also hyperlinks into the source code.

The Source Code Browsing use case is given in Figure 3.11.

3.4 Static Analysis

Next, it is helpful to obtain a graphical view of the software system. This helps to bring perspective to the relationships found by browsing the source code of the software system. Inheritance trees, call graphs, and transitive reachability queries can all be performed by static analysis. One can list just the entities of the software system as well; these are reported in tabular form as there are no relationships to draw between them. The static analysis features are evident in the use case outlined in Figure 3.13. For the other queries, the results are given in tabular form (see Figure 3.14), and in graphical form in the form of a clustered entity-relationship graph (see Figure 3.15). An example transitive Reachability Query is also shown in Figure 3.16.

3.5 Graphical Display

Graphical display of query results is obtained by visualizing GXL graphs, with clusters represented by special nodes that can be clicked to expand them into their full graph view. This is

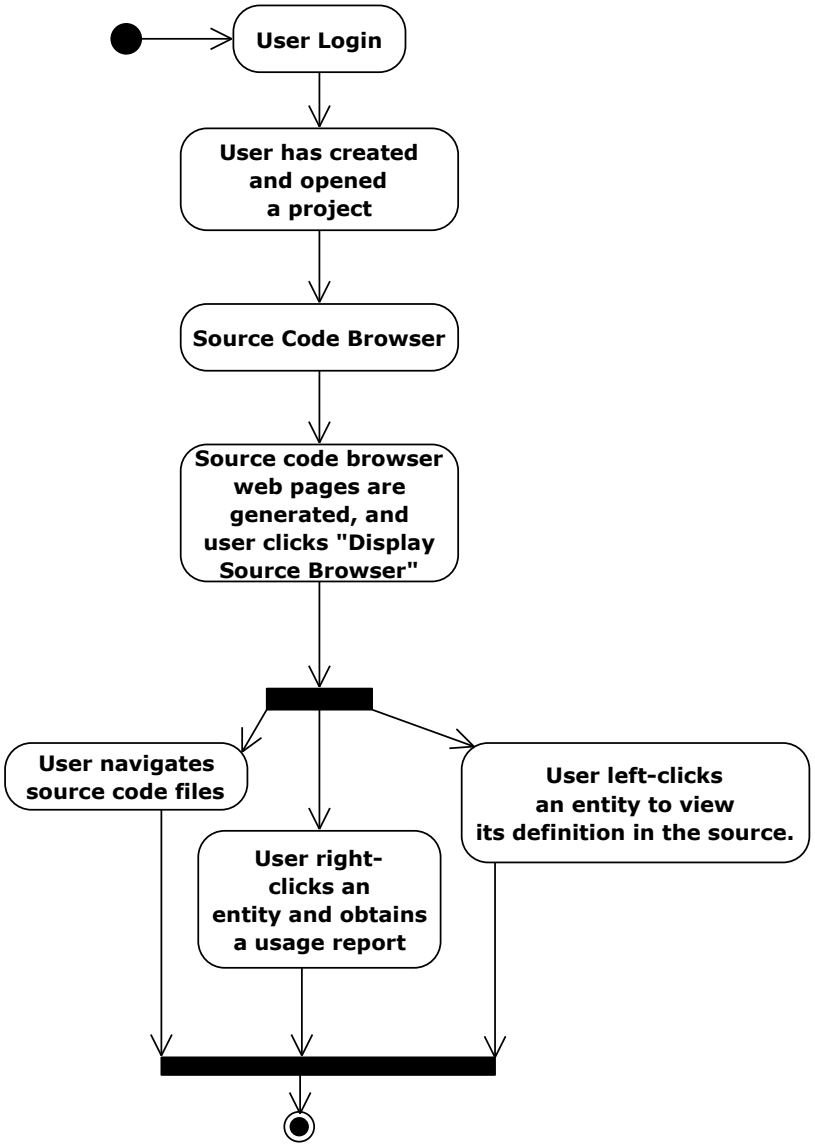

Figure 3.11: The REportal Source Code Browser use case

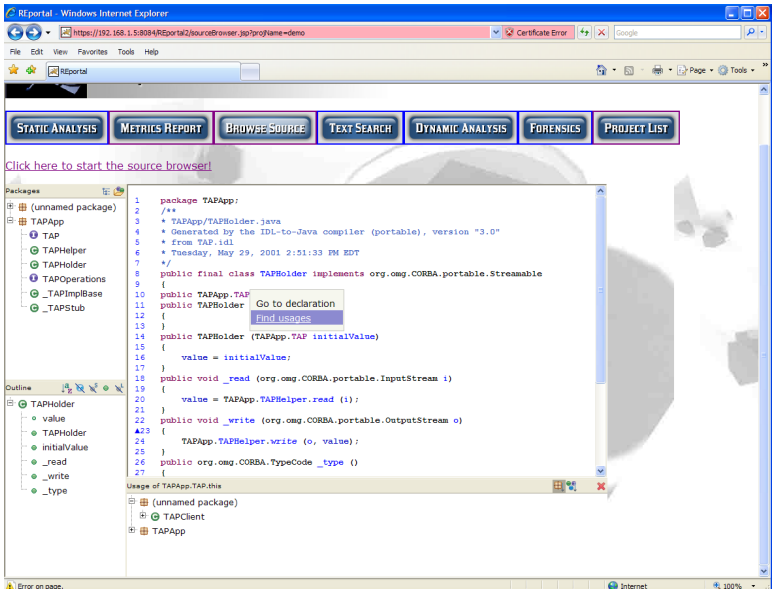

Figure 3.12: REportal Source Code Browser screen featuring cross references in the code

accomplished with a tool called ClusterNav, and allows for visualization of the software system at various levels of abstraction. The tool runs as a Java applet at the JSP presentation layer. Examples of this graphical output are depicted in Figures 3.17 and 3.18.

Clusternav performs XML queries on the GXL [70] graph produced by Bunch. As an XML graph format, GXL represents the graph as a hierarchy of subgraphs, yielding a tree of graphs that represents "clusters of clusters" of nodes. By double-clicking on these clusters the user can expand or collapse clusters into a single node view, causing the graphical display to represent the entire graph at various levels of abstraction; for example, a single cluster can be displayed at the full-detail level, while the remaining clusters are completely collapsed. The cluster nodes are color coded using a spectrum from green to red, comparing the relative number of nodes or sub-clusters within each cluster.

The functionality of the cluster viewer is summarized in the use case given by Figure 3.19.

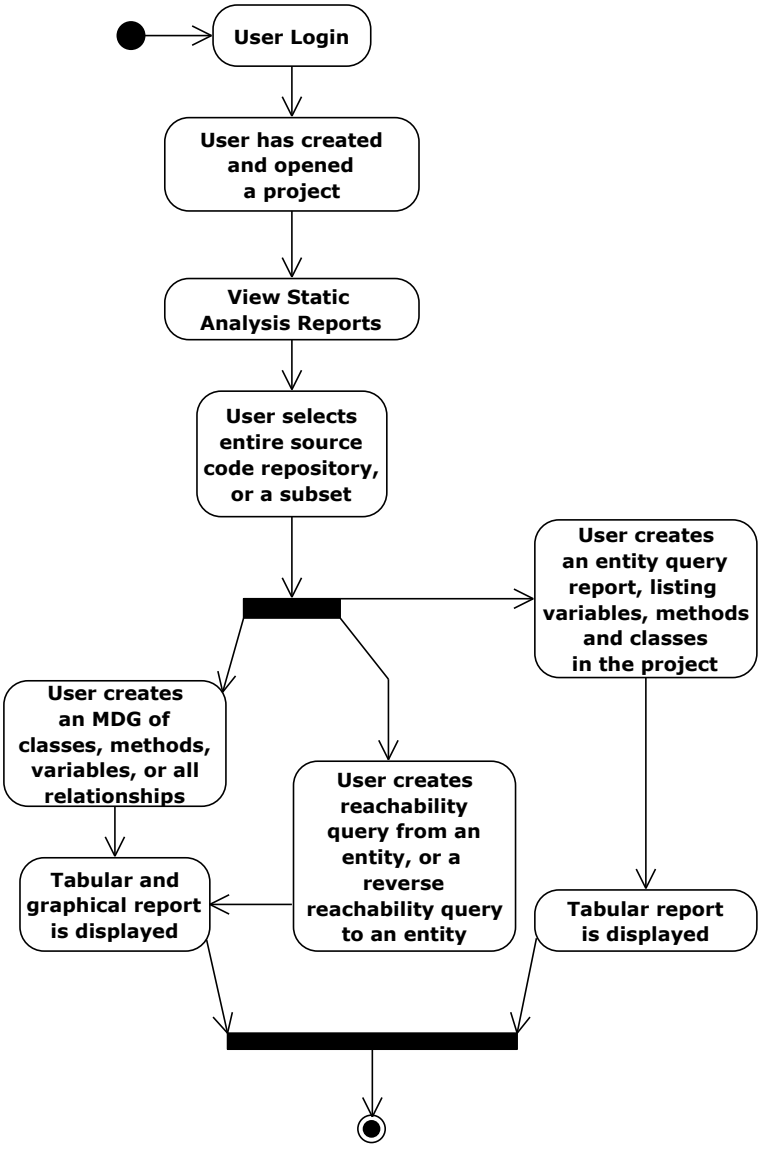

Figure 3.13: The REportal Static Analysis use case

Figure 3.14: REportal Tabular MDG screen

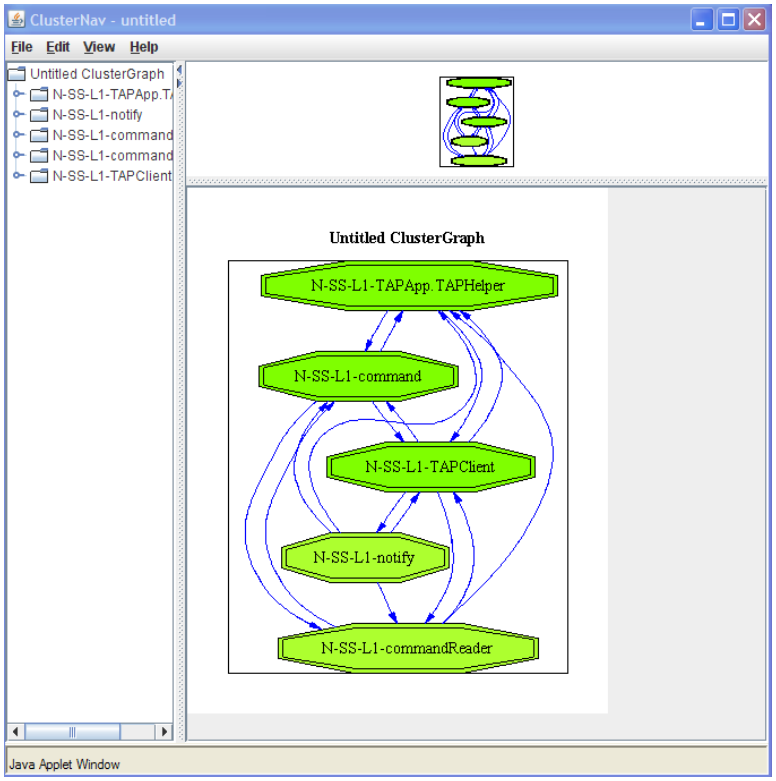

Figure 3.15: REportal Graphical MDG screen

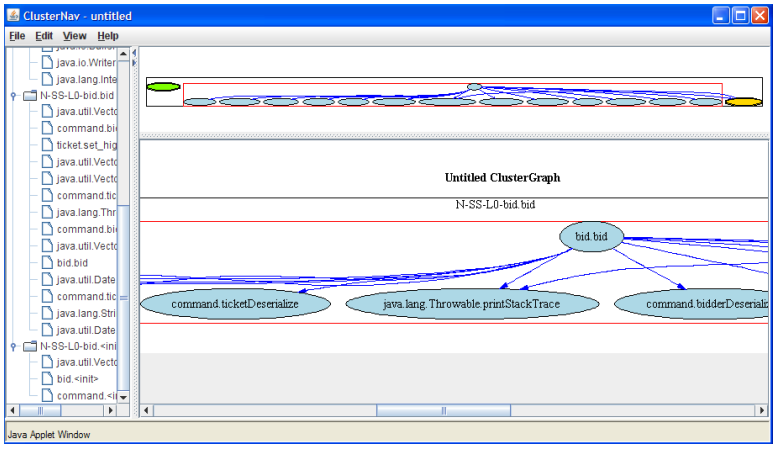

Figure 3.16: REportal Reachability Query screen

3.6 Dynamic Analysis via Aspect Instrumentation

The relationship queries offered by Static Analysis are helpful, but they are often supersets of what the user is looking for. In other words, static analyzers produce a lot of "noise" in their reports. This is because static analyzers do not often detect all of the "dead code," or code that is not reachable, within the system.

Moreover, the static analyzer shows artifacts and relationships that exist within the source code. They do not analyze code on a per-feature basis (for example, just code responsible for saving a file). Static reachability queries help to reduce some of this noise if it is known where the feature is implemented in the code. However, assuming this is the first time the software system is being inspected, it is not likely that the user will know this *a priori*.

For this type of query, Dynamic Analysis provides a more accurate view of the system. Dynamic analyzers inspect a software system at runtime, so an *as-built* view of the system is obtained. A call trace is not obtained simply by reading what the source code would do, but rather by executing one or more features of the software system and observing what the system does. This would give, for example, the most accurate call trace for that feature. For this reason, dynamic analysis is often

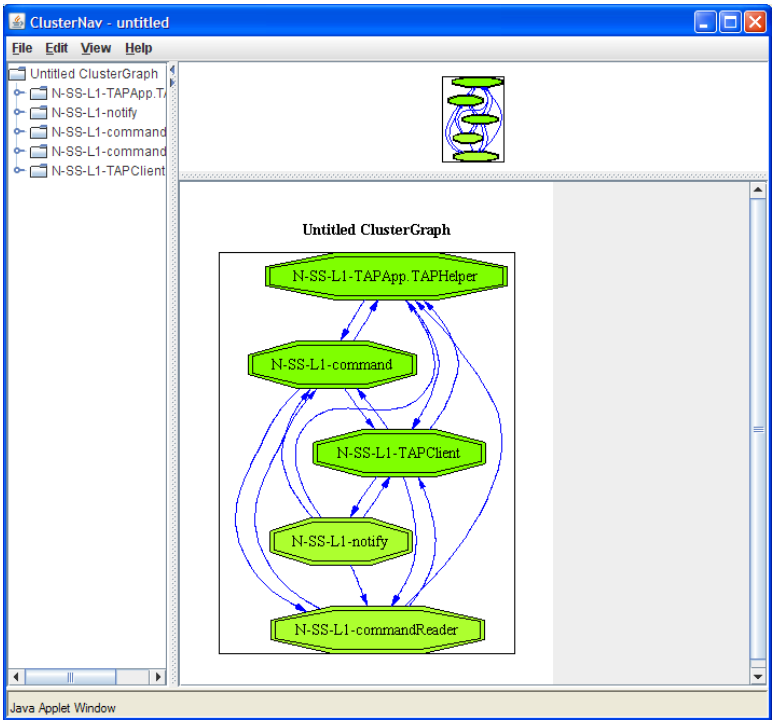

Figure 3.17: Collapsed view of a class relationship graph

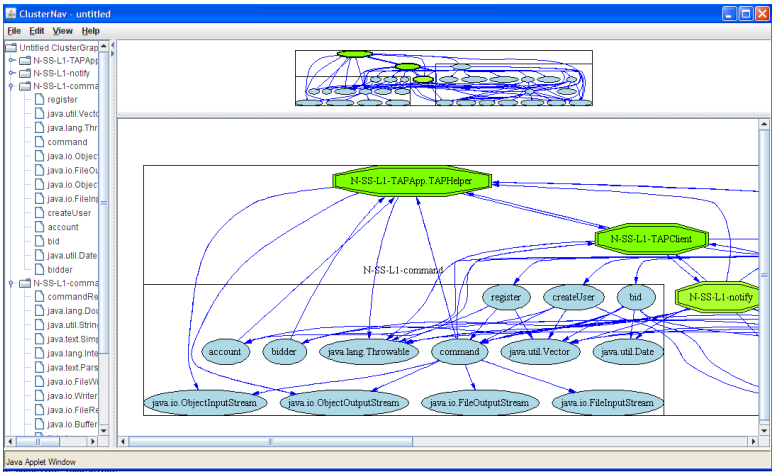

Figure 3.18: Expanded view of a class relationship graph

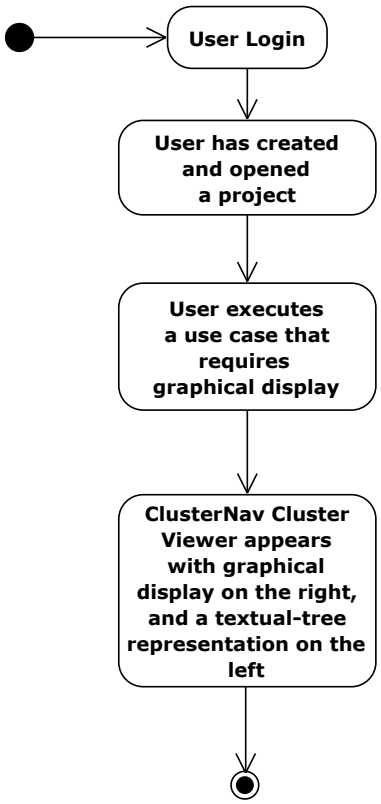

Figure 3.19: The REportal ClusterNav use case

used to augment data collected from static analysis.

However, for security reasons, it is impossible to allow an unknown user to upload arbitrary code to a server and then execute it. A different approach is required, and is outlined in the use case shown in Figure 3.20.

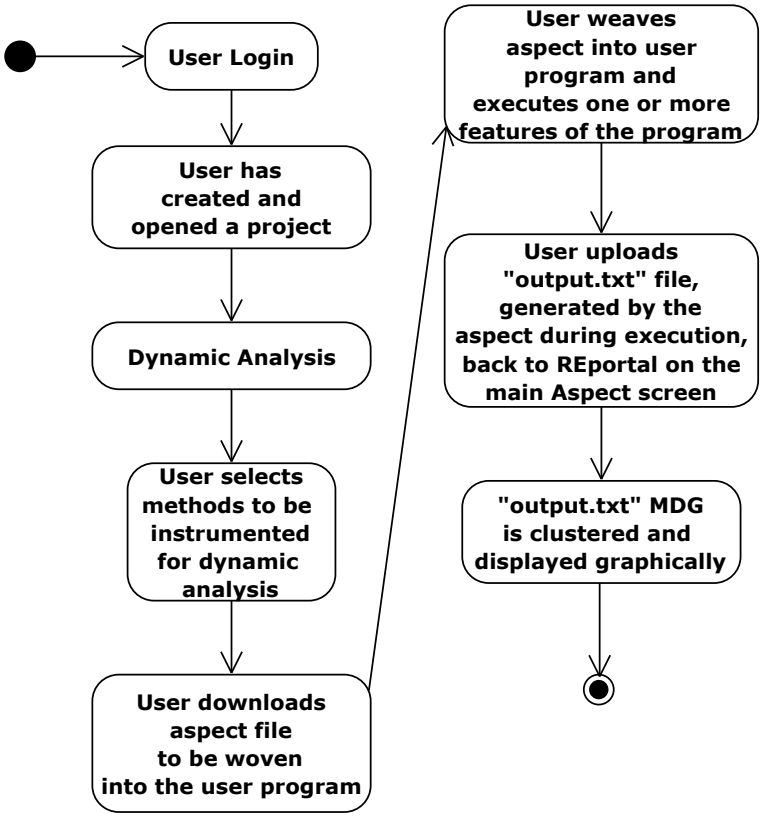

Figure 3.20: The REportal Dynamic Analysis use case

Aspect-Oriented Programming (AOP) [33] is used to instrument the code with logging code. This logging code keeps a trace in a file of every method that is called during the system's execution. The user selects one or more of the methods in the project (see Figure 3.21), and the aspect is generated and downloaded.

Figure 3.21: REportal Dynamic Analysis aspect generation screen

The aspect is woven into the program by downloading AspectJ [33] and using the `ajc` compiler to build the software system along with the downloaded aspect file. The resulting bytecode is executed as normal, and normally the user will execute one or only a few features so that the resulting trace file is small and focused. The trace file *output.txt* is generated automatically, and this file is the call trace graph that resulted from executing the program. This file is uploaded back to REportal as shown in Figure 3.22.

The graph is clustered much like a relationship query in Section 3.4, and displayed to the user (see Figure 3.23).

3.7 Text Search

A more focused search can be obtained by searching for specific strings or regular expression patterns that exist in the source code. The Text Search feature enables this, and its use case is outlined by Figure 3.24. The user enters the search string or regular expression pattern, selects a flag

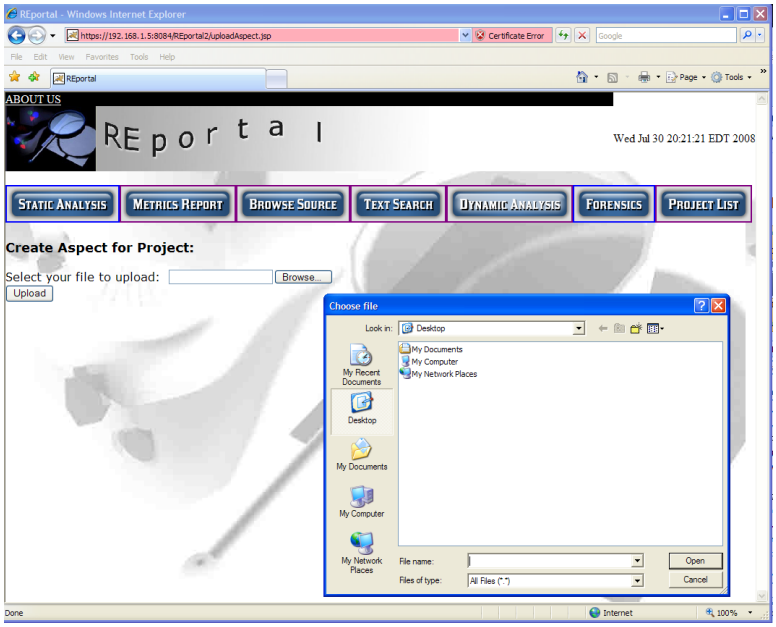

Figure 3.22: REportal Dynamic Analysis trace upload screen

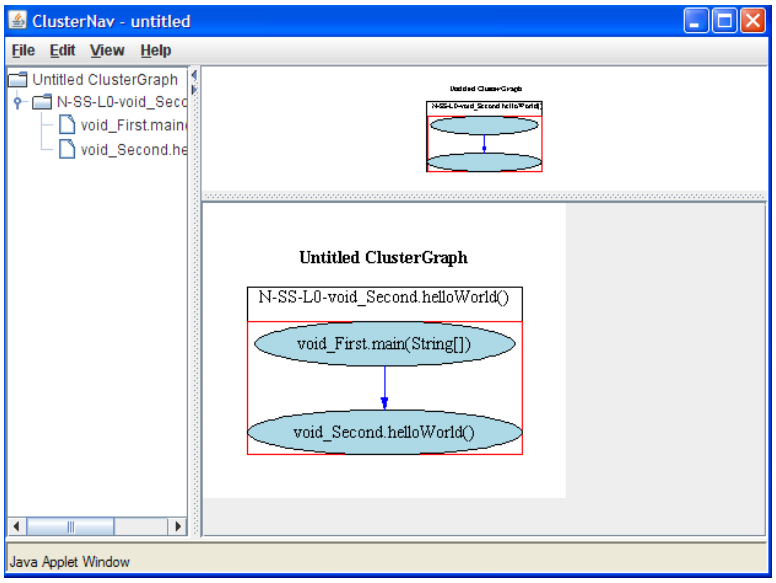

Figure 3.23: REportal Dynamic Analysis graphical result screen

indicating case sensitivity or insensitivity, and the results are displayed in a table (see Figure 3.25).

3.8 Software Metrics

Once the software system is understood by browsing the source code and exercising the static and dynamic analysis features, software maintenance and auditing can be focused by running a metrics report. Metrics data yield information about the size of an inheritance tree, number of methods in a class, complexity of a class (measured by the number of control forks in the code, such as conditionals, loops, *etc.*), comments in a source file, and so on. Classes or files that are outliers, are overly complex, or lack commenting may be candidates for further investigation and/or refactoring. This enables the software developer to focus the investigation, and thus save time in refactoring or re-architecting the system. The metrics report may result in a list of methods or classes that require maintenance; the developer may wish to combine this analysis with a series of reachability queries on these classes or methods to determine what other parts of the software system might be impacted as a result of this maintenance. Thus the developer can also be more careful in performing these more focused maintenance tasks.

The use case for running the metrics report is given in Figure 3.26, and an example screenshot is shown in Figure 3.27.

3.9 Software Forensics to Determine Code Authorship

Finally, the software developer may wish to predict who authored different files within the project. Alternatively, one may need to resolve an intellectual property dispute by determining likely code authorship. REportal provides a feature, developed in-house [47, 49], that produces a tabular report indicating the predicted author of each file.[1] Usage of the Forensics feature is outlined in Figure 3.28.

In order for the Forensics feature to predict which authors wrote which files, the Forensics program must be given a set of possible authors from which to choose. Along with this, the program

[1] This feature is described in more detail in the case study in Chapter 6.

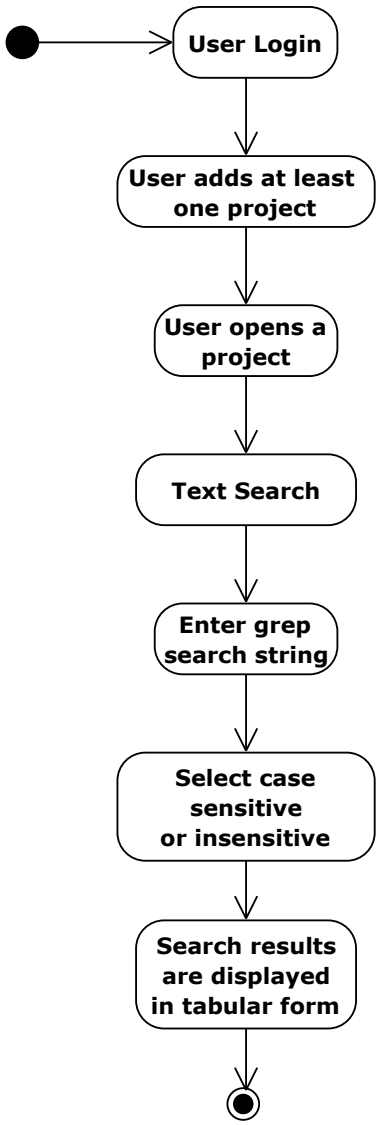

Figure 3.24: The REportal Text Search use case

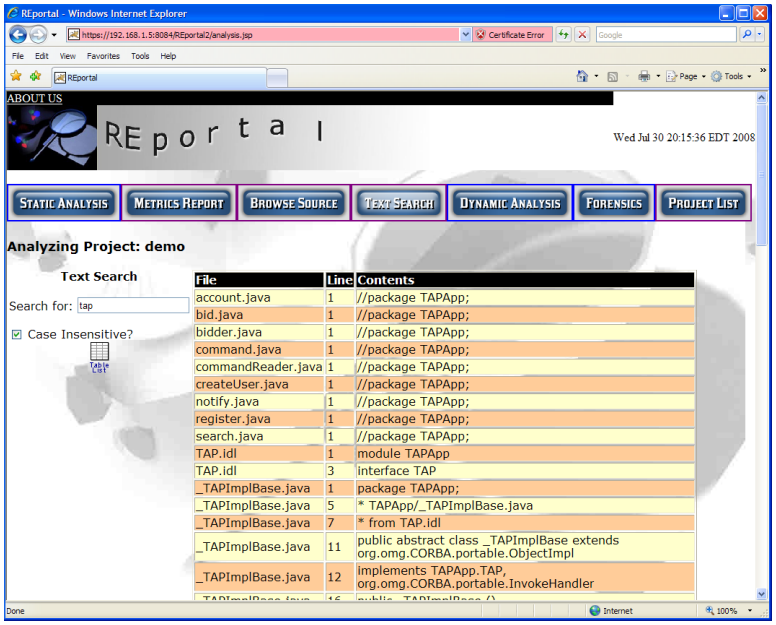

Figure 3.25: REportal Text Search screen

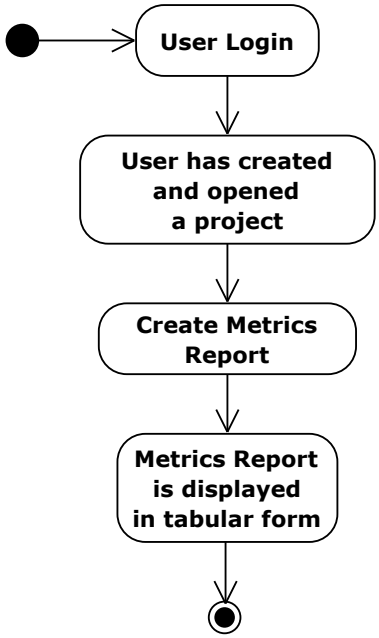

Figure 3.26: The REportal Metrics Report use case

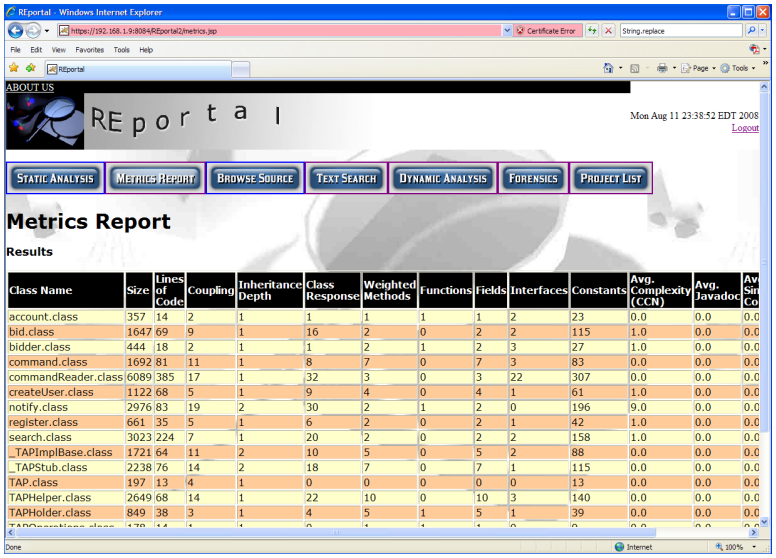

Figure 3.27: REportal Metrics Report screen

must be given a set of code samples from each author. With these samples, the program generates a "profile" for each author that includes each author's coding style. This profile is generated by running a series of metrics reports on each user's sample code.

To provide this information to REportal, the user creates a *ZIP* file containing a series of directories. Each directory corresponds to a possible author, and the directory name is used as the author's name. This file is uploaded to REportal through the Forensics feature as shown in Figure 3.29.

REportal unzips this file, creates the profiles, and then runs the metrics reports on the user's project files. The closest profile that matches the metrics reports run on each file is predicted as the author. Because some matches may be closer than others, the Forensics feature also gives a confidence value, indicating to what degree the user is predicted to be the most likely author of this file. It is important to recall that this prediction is done only on the users provided in the *ZIP* file; it is possible that the actual author is not included in this set, and thus prediction is impossible. Regardless, it is possible to mispredict the author for a number of reasons (for example, the programmer used a different IDE which resulted in a different code style, or multiple authors

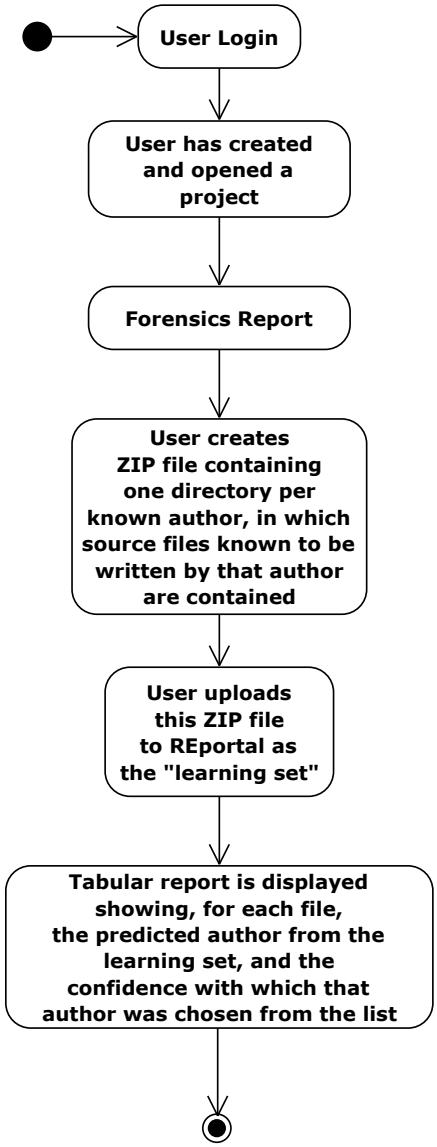

Figure 3.28: The REportal Forensics to Identify Code Authorship feature use case

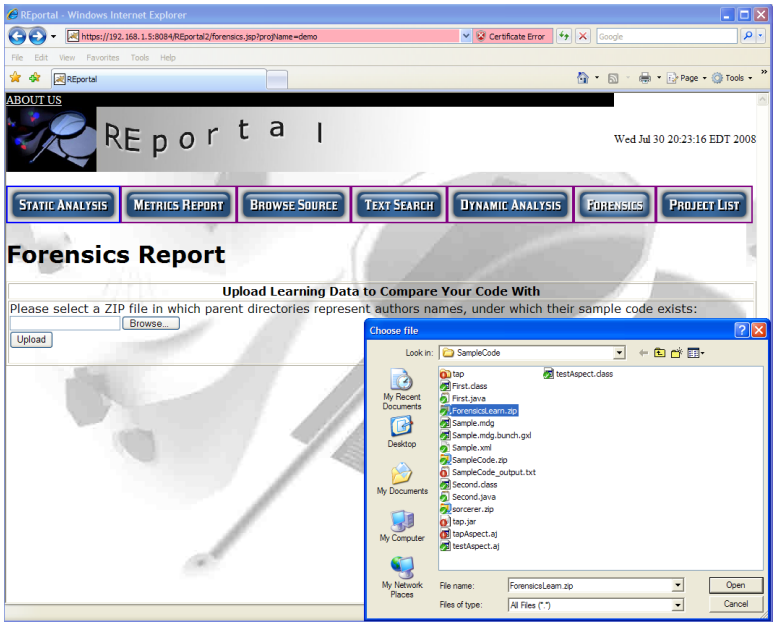

Figure 3.29: REportal Forensics learning set upload screen

contributed to one file), and this is the reason for the confidence factor. The final report is given in tabular form and is shown in Figure 3.30.

So far, we have covered the usage details of REportal and a summary of its architectural benefits. In Chapter 4, we discuss the REportal architecture from a developer's perspective.

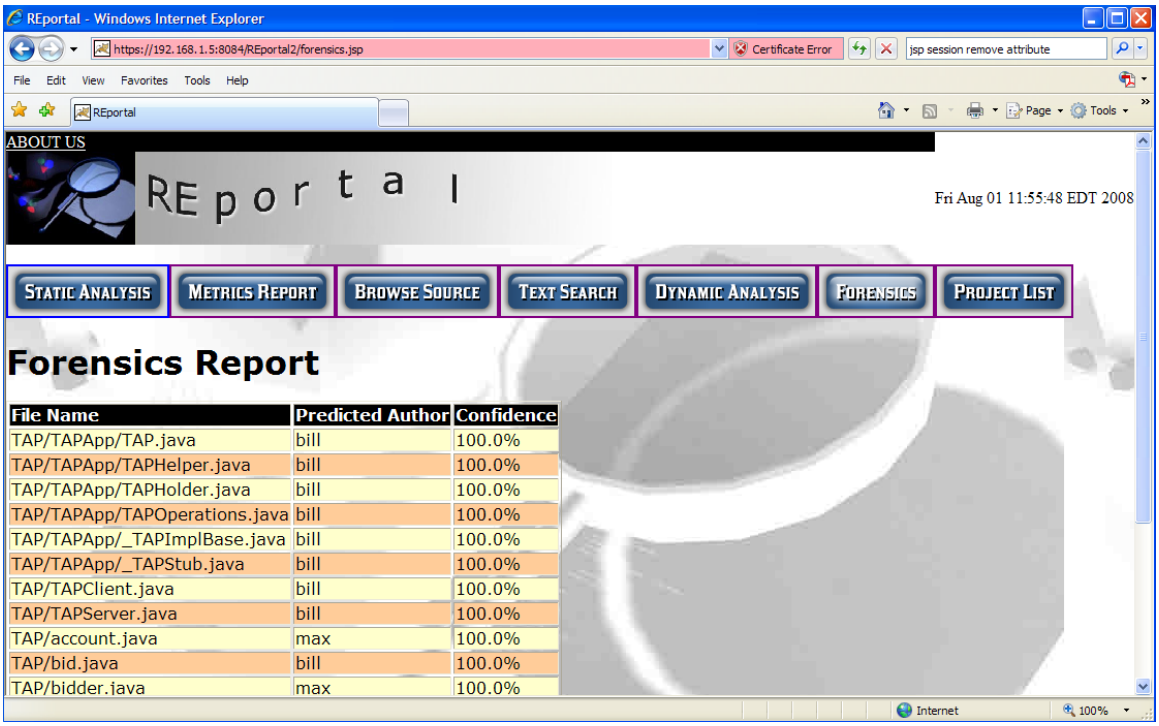

Figure 3.30: REportal Forensics tabular report screen

4. Developer Perspective

The approach taken to compose the new REportal is a tool-centric one. Rather than building a presentation layer that is coupled to the tools, we instead create a generic presentation framework that accepts and queries XML documents, using tools wrapped as web services. These XML documents may be repositories of software systems, Module Dependency Graphs (MDG), call graph representations, and so on. If a new tool is integrated, it is only necessary to represent the tool's output as an XML document and query it at the presentation layer for display. Graphical output is created by the Bunch clustering system, accepting an MDG graph as input and outputting a Graph Exchange Language (GXL) [70] graph, which is an XML graph format. This format is used by the ClusterNav visualization program which uses GraphViz as its underlying drawing tool.

4.1 REportal Web Client

Because each tool is wrapped as a web service, a standalone non-web client can be easily created on any platform by creating an implementation based on the WSDL and the XSD type definitions. We chose instead to implement our client as yet another server, created using JSP documents to represent the presentation layer. SOA client stubs invoke the services and return their results, possibly after some filtering and logistics (including, for instance, unzipping a user's project on the file system). The JSP presentation layer invokes the client stub, queries the resulting XML document, and displays the results on the web page or graphically using the ClusterNav tool. Each source package within the REportal web client corresponds to a different service provided by REportal. The relationship between the presentation and service layers of REportal is noted in Figure 4.1.

The REportal web-client is made up of JSP documents, whose navigation is shown in Figure 4.2.

Users begin at the *index.jsp* or *home.jsp* page (one simply points to the other). Users then have a choice between registering or logging in, at which point they are taken to the *projectManager.jsp*

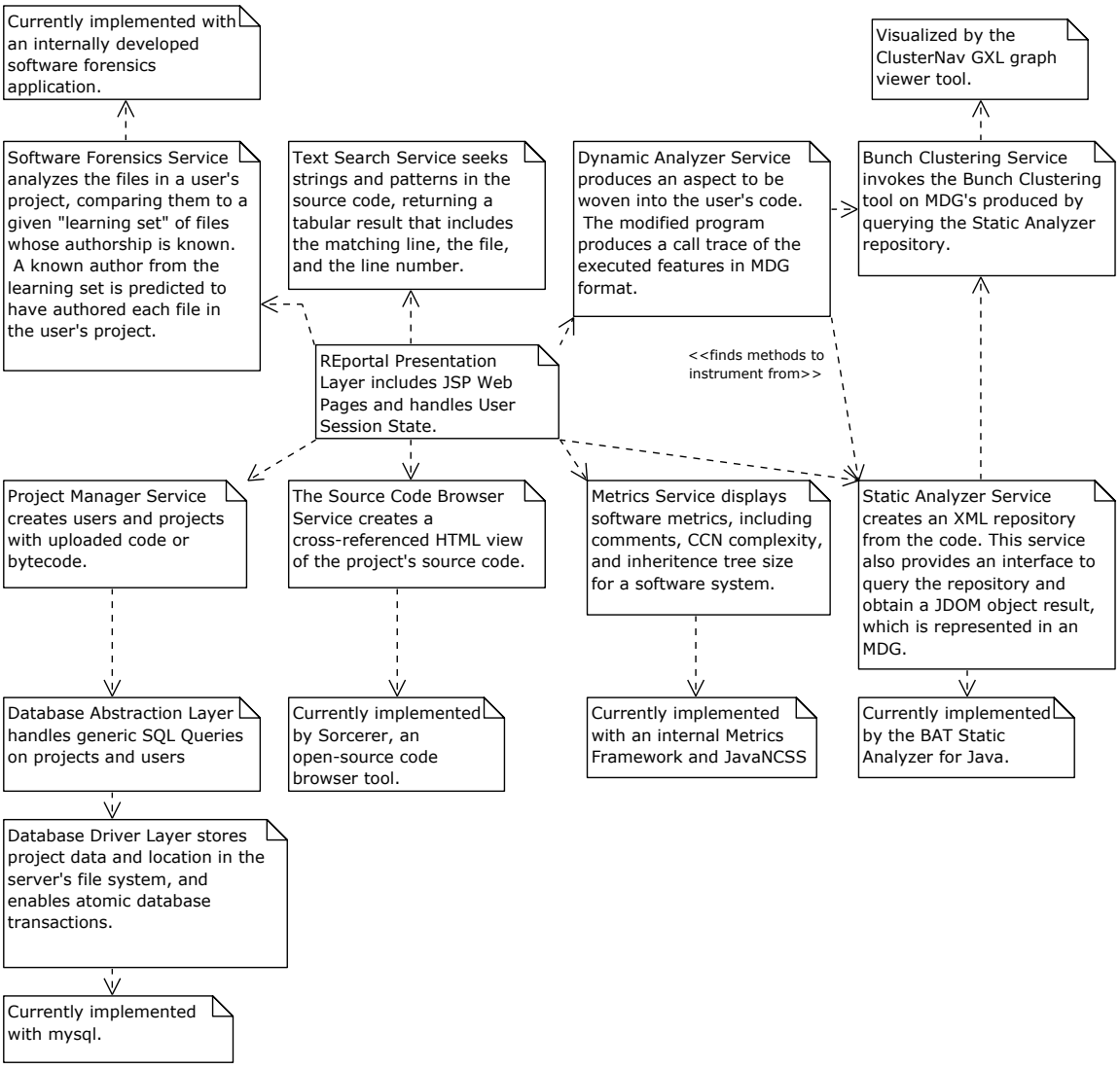

Figure 4.1: Generic layered structure of REportal's underlying services

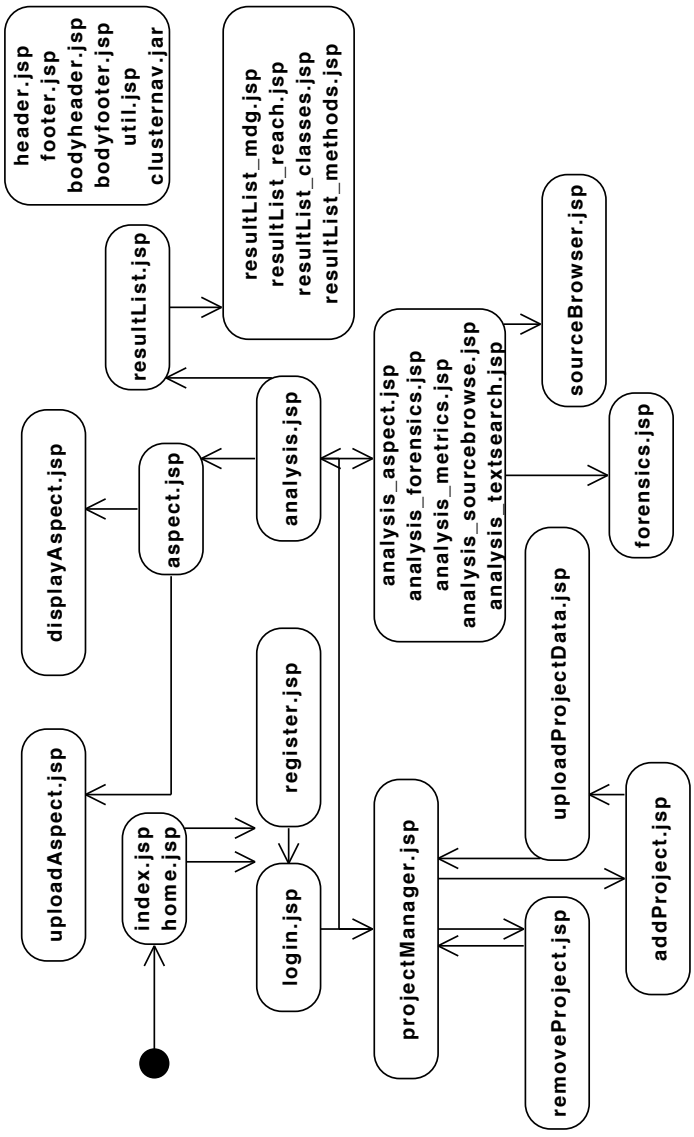

Figure 4.2: JSP navigation diagram of the REportal web client

page. Here, several options are possible; one can add, upload data to, remove, or upload a project. Once the user and project logistics are complete, the user proceeds to the *analysis.jsp* file. This file uses a variable called `analyzeRibbonPage` (managed by the *analyzeRibbonPage.jsp*, which is not pictured for brevity) that represents the feature being invoked. This enables the *analysis.jsp* file to highlight the correct button at the top of the screen, and import the correct *jsp* file into the display. Each of the primary features are accessible from *analysis.jsp*. Alternatively, the user can logout, in which case the user's saved session state is destroyed, and the user is returned to the homepage.

Of primary interest from the *analysis.jsp* page is the *resultList.jsp* page. If static analysis is chosen, the user is given the option to filter their results: one may execute a class entity query, method entity query, reachability query, or an MDG relationship query. In all cases, an XML query is executed against the BAT XML repository, and the user is taken to *resultList.jsp* for display. Some queries require a tabular report display, while others require a graphical display. The files *resultList_mdg.jsp*, *resultList_reach.jsp*, and so on, are responsible for carrying out this display, and the proper file is imported into *resultList.jsp* after the XML query is executed. In this way, there is even some modularity among the *jsp* files, in that JSP code can be imported just-in-time based on the user's session state.

Finally, a number of omnipresent files exist, including a header and footer (for code and HTML header data), as well as a body header and footer (for adding text or graphics). There is also a file called *util.jsp*, which is included by *header.jsp*, and contains a number of overhead methods that might be required by all *jsp* pages – for example, user token verification. Finally, *clusternav.jar* is the Cluster Viewer library used by the presentation layer. Normally, this file would be stored in the *$REPORTAL_ROOT/lib* directory, but this file must be accessible from the web so that it can be downloaded to the client's web browser on demand. Therefore, it is stored with the *jsp* pages for download.

ClusterNav also requires a CGI script to be present in the REportal *web/cgi-bin* directory. This file is called *webdot.pl*, and renders graphs for the cluster viewer to display. CusterNav produces

unrendered *dot* graphs from the clusters provided by Bunch, and uses *webdot.pl* to invoke the *dot* application to render the graph. The rendered graph is also in the *dot* file format, but contains additional rendering information, including where to draw the edges. Configuring Tomcat to use CGI scripts is discussed in Section B.2.2.

Configuring the REportal client is discussed in Section B.2.4.

4.2 REportal Web Services

To integrate the tools on the server, the primary tool functionality is first identified; we find its appropriate customization parameters, inputs, and output format. These inputs and outputs may be file-based via a command line, or string-based via an API. In either case, a web service is created that wraps this primary functionality. An XSD is created to represent the parameters, input and output of the tool, and a WSDL exposes a high-level interface that uses the parameters, inputs and outputs. The WSDL is implemented in any language, on any platform, and on any host, to invoke the tool. The service architecture allows for transparent passing of parameters and results from service host to service client, given the host's service URL. This is a critical element of REportal's new architecture, because some of these tools only run on a particular platform, or on a particular JDK, making it essential to support tools that exist on various hosts. This strategy also eliminates the need to house these services. It is a long-term goal of REportal to support the dynamic location and execution of a service at runtime. Currently, a simple distribution of services exists that allows for compatibility between the application server (Apache Tomcat [27]) and the dependencies of the underlying tool. The current REportal service distribution is illustrated in Figure 4.3. Using this approach, REportal represents a framework for distributing services over the web that expose RE tool functionality.

In this case, the JSP Presentation Node represents the service orchestration entity: based on the user's desired execution task (a call graph relationship, for instance), it chooses the services to invoke and runs them in the proper order, passing the result from each one to the next service until

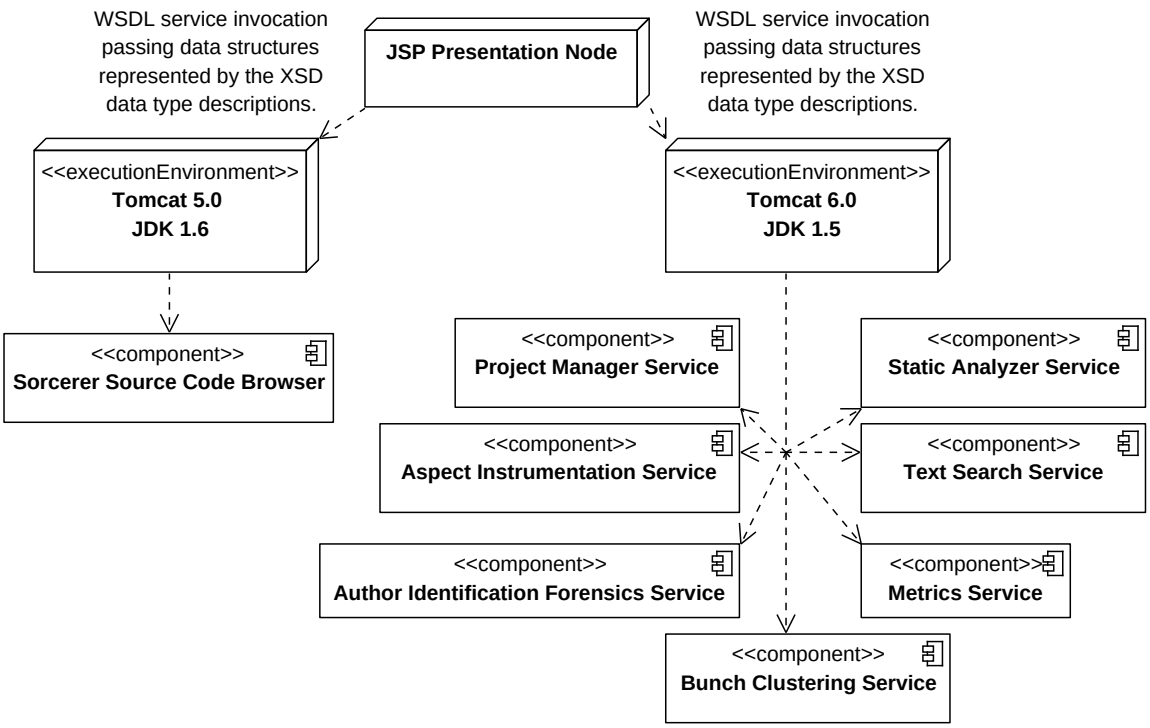

Figure 4.3: Distribution of services among application servers in REportal

the final output is obtained. Ultimately, even this process will be automated via the BPEL. With an automatically generated BPEL, services are selected and orchestrated for a particular execution task. Once an RE business process is established that creates a process thread for these services, it becomes possible to execute these services in parallel. The BPEL process and its potential for SOA is discussed in Section 7.1.

4.2.1 Methodology for Creating a Tool-Centric Service

The service-oriented architecture implemented by REportal is intended so that new tools can be added as web services. However, it is well understood that many of these tools, especially older tools, are implemented as standalone programs, shell scripts, or APIs, rather than as a web service. It is therefore necessary to encapsulate the functionality of the tool as a web service before integrating it with the portal. We have designed a methodology for doing this in a way that preserves the decoupled service and data spirit of REportal while adding minimal burden to the developer. This process was refined via the case study described in Chapter 6, and is described briefly here.

4.2.2 Identification of Core Functionality

First, the core functionality of the tool is identified. This functionality is described at the business-logic or use case level of abstraction. It is not typically necessary to port every feature of a tool to the portal; to do so would add unnecessary complexity to the user interface at the presentation layer by burdening the user with numerous options. Configuration and invocation of these subfeatures is performed automatically from within the service as part of providing the executive-level feature to the user. These core functionalities usually represent one or a small number of methods to be created within the web service description.

4.2.3 Data Type Design

Once the core functionality is identified, the minimal inputs and outputs are determined to invoke the functionality from within the legacy tool. This includes the inputs and outputs that are required to execute intermediate features or programs from within that feature. These form the basis of the data types that will be created within the web service description, and serve as parameters or return values from the methods also described therein.

4.2.4 Design and Implement the Web Service Wrapper

Finally, the web service is designed around the parameters defined in Sections 4.2.2 and 4.2.3. Figure 4.4 describes the behavior performed by the web service to invoke the legacy tool. Notice that the web service operation exposed to the client (the REportal presentation layer) is minimal, consisting of only those nodes which lie outside of the three clusters in Figure 4.4 (the Abstract Tool Encapsulation cluster, the Legacy Tool cluster, and the External Functionality cluster). As a result, the client must simply invoke the single or few web service operations in order to execute the main behaviors of the legacy tool. The service implementation, in turn, invokes the legacy tool, configuring it with the settings needed to obtain the result required by the web service operation. Like the end-user, the presentation layer is not concerned with the details of configuring the tool or the service that invokes it. Instead, it requests a report, analysis, or other information that the legacy tool can provide, and expects the service to handle obtaining that data in the format specified by the web service. This is, indeed, the primary benefit of a design-by-contract approach; it helps to minimize the complexity of the presentation layer as well as to minimize the amount of interaction and configuration required by the end user. The user makes a request via the presentation layer, and the result is displayed to the user via the presentation layer. The underlying service invokes the tool and external functionality, and may have to pipe-and-filter intermediate results through additional service invocations. In this case, the web service becomes a web service client on behalf of the presentation layer, and this is also depicted throughout Figure 4.4.

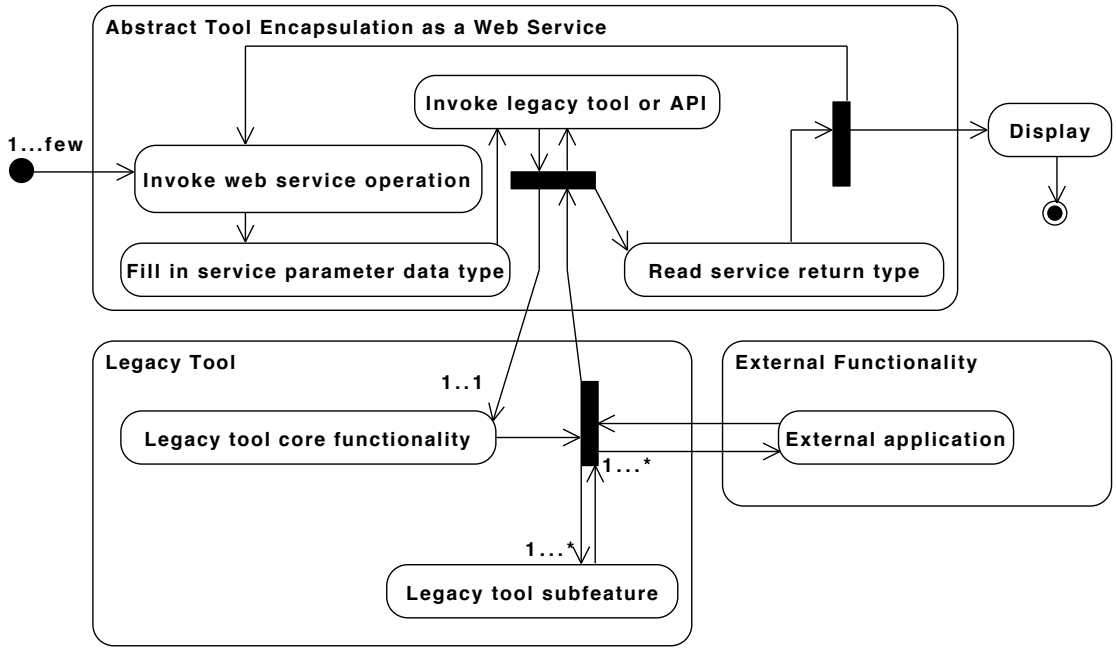

Figure 4.4: The presentation layer invokes a legacy tool using only its identified core functionality, exposed via a web service wrapper.

4.3 REportal Tool-Centric Services

In this section we describe some of the development details for each of REportal's services. For each service, we describe first the interface that defines the functionality provided by the service. We then discuss the data types that are used to implement the service, along with any additional appropriate implementation details. To see the full WSDL service contracts and XML schemas that define the data types, see Appendix C.

4.3.1 Project Management Service

The Project Management service is the one "logistics" service provided by REportal. It is responsible for interacting with the REportal database (see Section 4.4), authenticating users, validating user access to projects, and mapping projects to their locations on the file system. On the file system, each user has a subdirectory under the REportal root directory. Because these services are

distributed across multiple hosts or Tomcat instances, it is possible that the service analyzing a user's project is not located on the file system where the Project Manager resides. In this case, it is necessary to either serialize an object or XML data containing the information required by the service, or to send all or part of the user's project to the requesting service.

As shown in the `ProjectManagerPortType` portion of Figure 4.5, the Project Management service provides several functionality invoked by the presentation layer (described in Section 4.1). These allow the presentation layer to do the following:

- **Add a Project**: The `addProject` behavior takes a project as a parameter of type `ProjectInfo` and adds it to the database of projects owned by that user. It returns the same `ProjectInfo` parameter that it received on invocation to verify that the project was added successfully. Typically, the user's next action is to invoke the `uploadData` behavior, described later in this section, to provide the code and/or binaries associated with the new project.

- **List Projects**: This behavior takes a `UserToken` parameter that specifies a user and a validation token that indicates that the user has successfully authenticated. The Project Manager queries the database for projects associated with the user, and returns a list of `ProjectInfo` objects, which is called a `ProjectList`.

- **Login User**: This behavior takes a partially filled `UserToken` object that contains only the user login name and password. Because it does not contain a valid token from the server, the `UserToken` cannot be used to perform any of the other behaviors in REportal. The server looks up the user id and encrypted password in the database and, if they match, sets the appropriate token value in the `UserToken` which authenticates the user. If the user password does not match, an invalid value that cannot match any user is placed in the token. In either case, the token is returned to the presentation layer, where the user is notified of the authentication status and the `UserToken` is persisted as part of the user's session for use in invoking other behaviors in the system.

- **Register User**: Here, user information such as name, e-mail address, company name, and password are sent to the Project Manager for addition to the database. If the desired username does not yet exist, it is added to the database and assigned a user token. The user's password is encrypted and stored as well along with the rest of the user's provided information. If the user is successfully added to the database, the `addProject` behavior is automatically invoked to add a small demo project for investigation by the user upon login. The user is notified of success or failure of this operation, and, on success, is able to log in.

- **Remove Project**: The project to be removed is specified via a `ProjectInfo` parameter and the database is queried to determine its ownership. If the project is owned by the requesting user, it is "removed." In reality, a flag is set in the database indicating that the project is no longer active, and it is moved to another location on the filesystem. At present, it is not possible to "undelete" a removed project, but it would be an easy matter to undo this behavior if the functionality were provided to the user.

- **Upload Project Data**: File data is persisted as a `ProjectData` object, which contains the binary contents of a file corresponding to a user project's source code and/or binaries. The project manager verifies that the project is owned by the authenticated user, and then stores the file data (stored in a *ZIP* file format within the `ProjectInfo` parameter) on the file system in the appropriate location. That file is unzipped, and the project database entry (that was created when the project was first added) is updated to include the location of this directory.

The data types described here are defined in Figure 4.6. In the `UserToken` type, information about the user is stored, including the user's name, encrypted password, company, e-mail address, and an optional token called `loginId`. In WSDL, the optional attribute is indicated by specifying `minOccurs=0`, declaring that the `loginId` may appear zero times. This is to enable a login attempt by a user. In this case, `loginId` is not populated because the user has not yet authenticated. A `UserToken` without a `loginId` is legal but is considered unauthenticated. By default, `maxOccurs` is

set to 1, but this can be overridden, for example, to *unbounded* in order to create an array. In this case, when the user attempts to log in, all the information except for the token is filled in. The server authenticates the user and, if successful, the `UserToken` is populated with a token value. The other behaviors within REportal check for and validate this token value before executing any behaviors.

Next, the `ProjectList` type, as described in this section, contains a list of `ProjectInfo` objects. This is useful for listing projects that belong to a user, or for searching for a user's project. The `ProjectInfo` type consists of a `language`, which is an enumeration called `FileType`. It enables values such as *Java*, *C*, and *C++*. This type is also defined in the WSDL but is omitted here for brevity. `ProjectInfo` also contains the `filePath` to the project's source and/or binaries, the user ID that owns the project, and the name of the project.

Finally, the `ProjectData` type holds a user's uploaded project files for saving to the file system. It contains a `ProjectInfo` type that links it to a project, and a `data` element that contains a binary sequence of *ZIP* file data to be written to the file system and unzipped by the `uploadProjectData` behavior.

4.3.2 Static Analysis

Static analysis features are currently offered for the Java language, and are provided by the BAT static analyzer [37]. BAT parses the Java bytecode and produces a representation of the code structure and behavior as an XML document in a schema defined by BAT. Using BAT, we created a query engine that performs common software analysis queries on the document, either using XSLT transformations or equivalent XQuery operations [54] (discussed in Section 4.5.1). The result of these queries is represented in a JDOM [12] document, using an XML representation that is currently internal to REportal but will ultimately be developed into a comprehensive XML Schema. BAT currently has a dependency on JDK 5, and for this reason it is run on a separate application server that runs on top of JDK 5.

Because static analysis is one of the most common operations invoked in REportal, the BAT

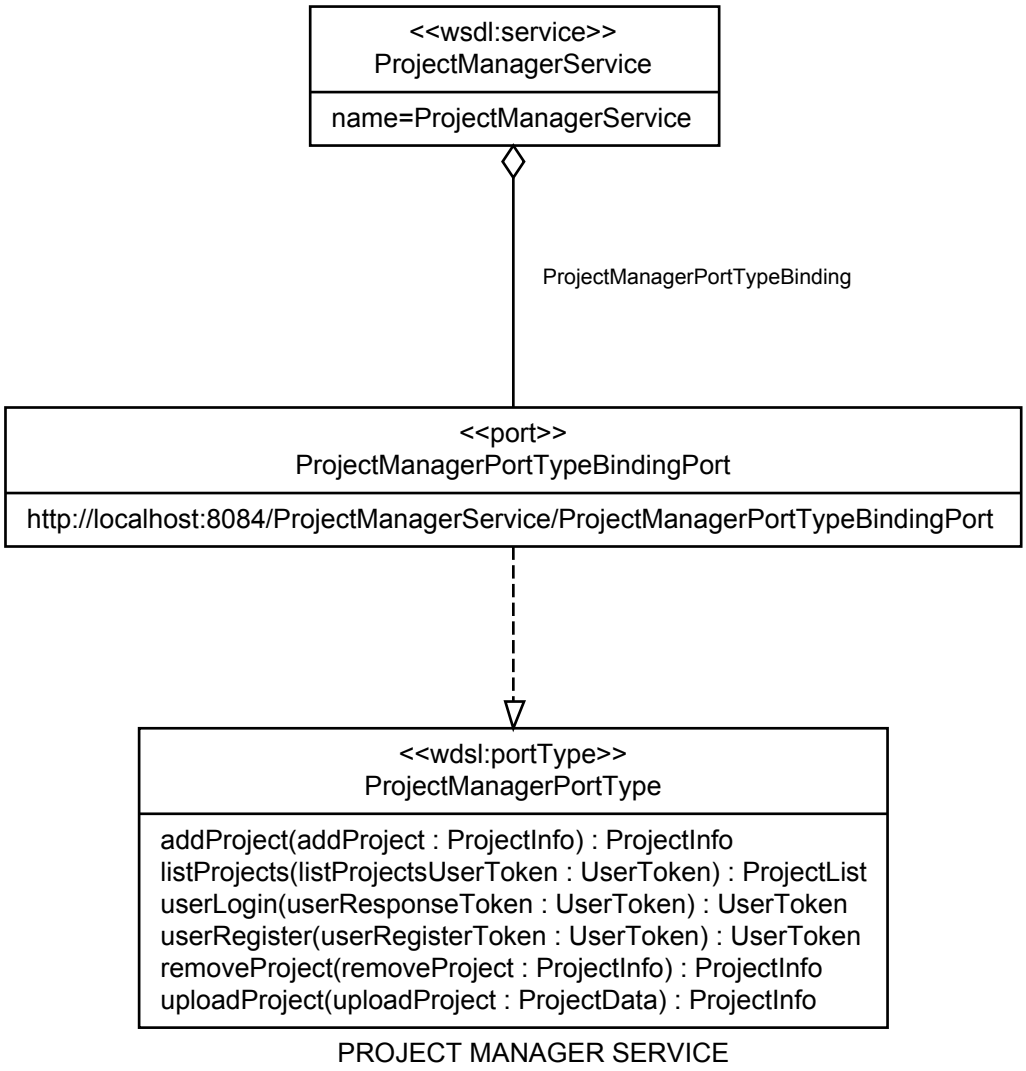

Figure 4.5: The Project Manager service definition

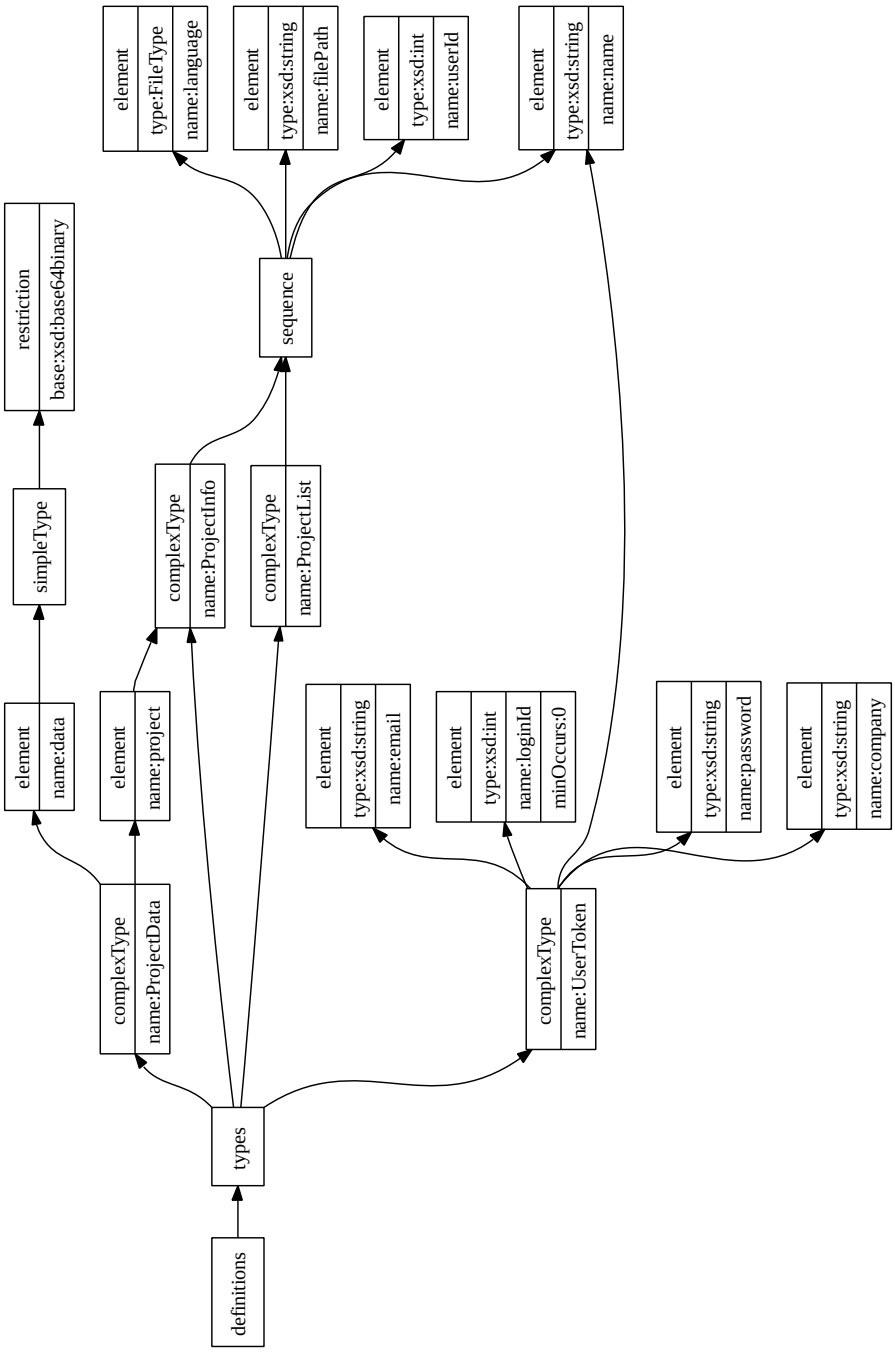

Figure 4.6: The Project Manager data types and structure

Analyzer contains only one operation. Given a Java class or *ZIP* file (uploaded from the Project Manager), the BAT Analyzer service analyzes the entire project and produces a unified XML repository containing all of the project's artifacts. This is returned as a `string`, and saved as part of the user's session by the presentation layer. This effectively caches the user's project information for faster querying. The operation is specified in Figure 4.7.

The data types require to implement this service behavior, as shown in Figure 4.8, are a `FileType`, which is the same `FileType` enumeration used by the Project Manager service and specifies whether the file data is a *ZIP*, *JAR*, or Java *CLASS* file; and a `filePath`, which is the path to the file to be unzipped and/or analyzed.

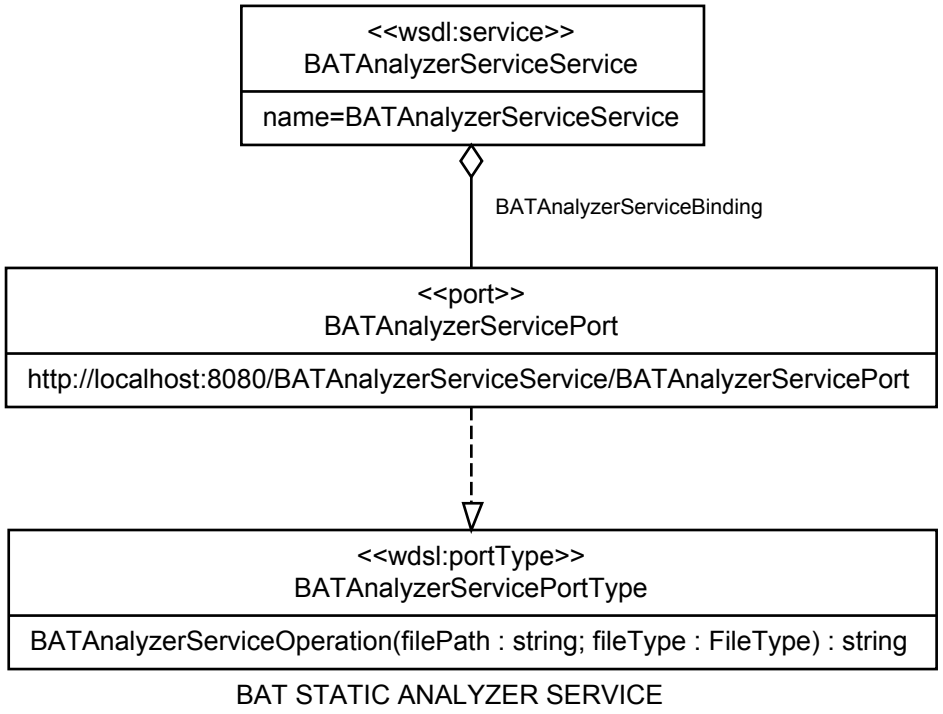

Figure 4.7: The BAT Static Analyzer service definition

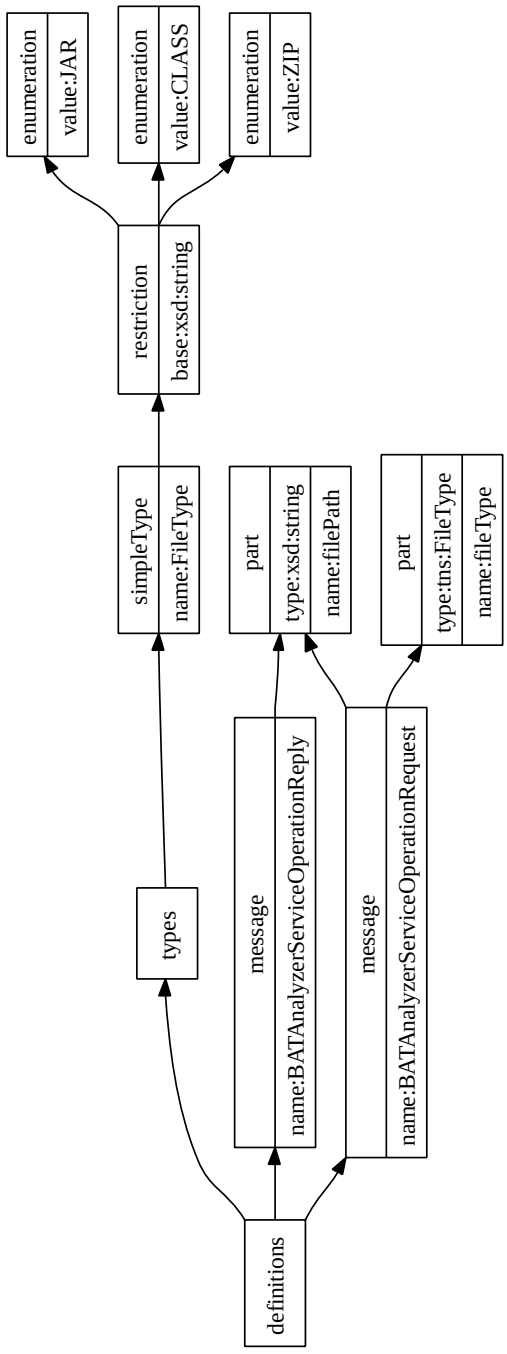

Figure 4.8: The BAT Static Analyzer data types and structure

4.3.3 Bunch Clustering

Querying the resulting XML document from the BAT Analyzer service, a Module Dependency Graph (MDG) is obtained that the Bunch clustering system [52] converts into a clustered graph in GXL [71, 40] format. This clustered graph provides a hierarchical representation of clusters-of-clusters within a graph, that one can navigate with an appropriate viewer (see Section 3.5). For static analysis graphs, this clustered abstraction provides a view of the subject program's subsystem structure, derived from the relationships found between its entities.

Because this service provides a back-end feature to convert XML query results into a graphical display format, the user never directly invokes this feature. Instead, this feature is invoked by another service or by a feature invoked by the presentation layer. The operation is transparent to the user: if the user requests a relationship query, it is performed on the static analysis repository, and automatically passed to the Bunch Clustering service to obtain a clustered graph for display. The resulting graph is passed to the *ClusterNav* graph viewer, developed by the author, for interactive display.

The service, shown in Figure 4.9, provides one operation. This operation takes an MDG as a `string`, and returns a `string` representing the clustered graph in GXL format. As a result, all the data types used by this service are of type `string`, which can be seen in Figure 4.10.

4.3.4 Metrics

The Metrics service is provided by a metrics framework that currently runs JavaNCSS [8] and other metrics computation programs on Java code. Metrics include fan-in and fan-out for classes, inheritance hierarchy size, method Cyclomatic Complexity, and others.

The Metrics service is quite similar in structure and behavior to the BAT Static Analyzer service described in Section 4.3.2. Like the Static Analyzer service, the Metrics service accepts the file path and file type to the user's project, and runs the metrics computation programs. The results are aggregated and represented as an XML document, which is returned as a `string`. This is shown in

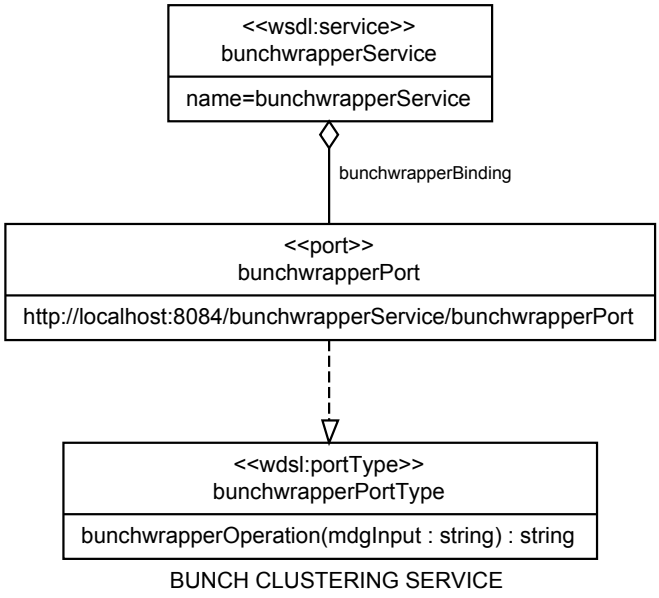

Figure 4.9: The Bunch Clustering service definition

Figures 4.11 and 4.12. The presentation layer parses the XML metrics report for tabular display to the user.

4.3.5 Source Code Browser

Source code browsing features for the Java language are provided by Sorcerer [19]. This service supports Java code only, and requires JDK 6. For this reason, it is run on its own application server which runs on top of JDK 6. The source code browser is different from the other tools in that, instead of returning an XML document to be queried, it returns a *ZIP* file set of HTML pages that are either displayed on a web browser or downloaded for display. To accommodate this, REportal displays the web pages in its presentation layer dynamically via Ajax (see Figure 3.12). Sorcerer's results are displayed in an iFrame in REportal via Ajax. Drop-in replacement services that support other languages can return a set of web pages, or unzip the web pages to a known location, and they can be similarly displayed at the presentation layer. Because the user's project language is stored

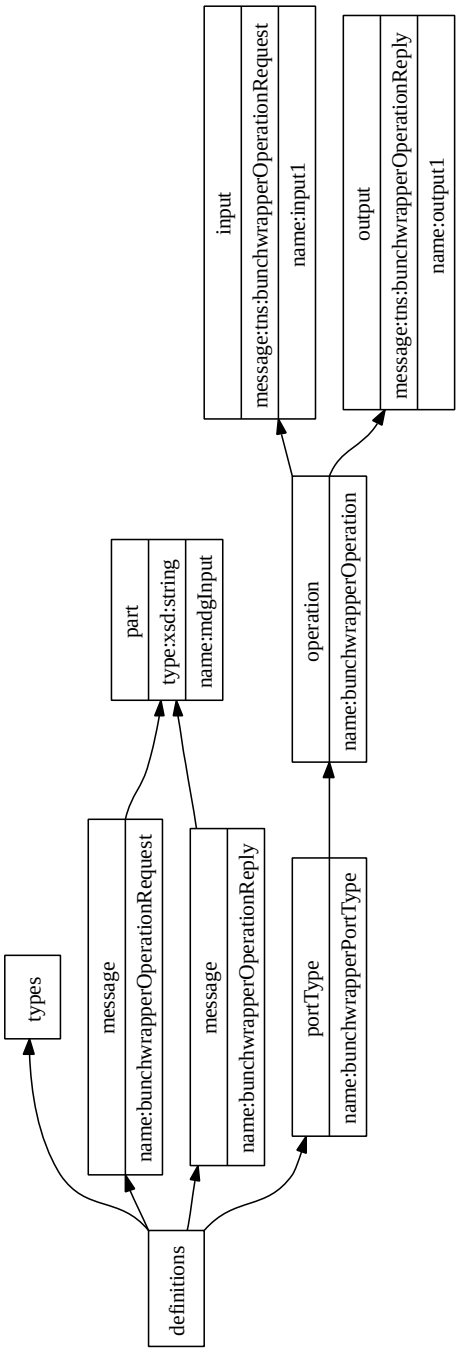

Figure 4.10: The Bunch Clustering service data types and structure

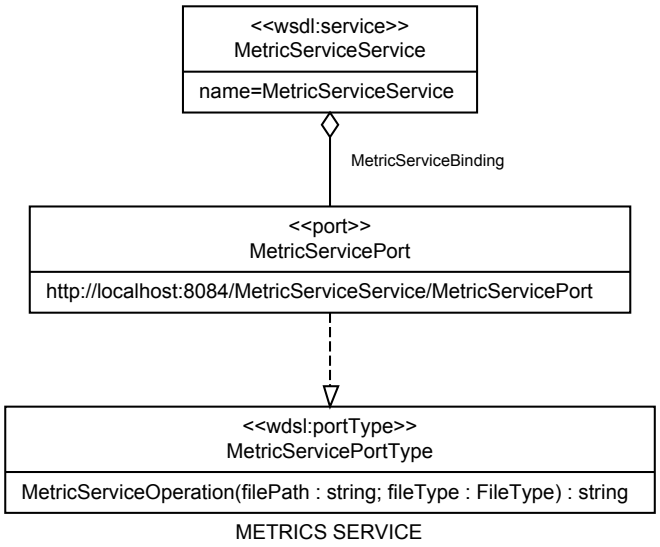

Figure 4.11: The Metrics service definition

in the REportal database, the appropriate service for that language can be chosen at runtime.

The sole service operation (shown in Figure 4.13) accepts the file path to the user's project and runs the source code browser program on the project files, obtaining a series of web pages. Because these web pages are meant to be viewed standalone, and not within another web site like REportal, a number of modifications must be made to the generated web pages and the included JavaScripts, in order to make them compatible with the *iFrame* web object in which they will be displayed. These modifications take place on the server side, and the resulting web pages are stored in *ZIP* format. This file is returned by the service as a binary sequence of data, as shown in Figure 4.14. The presentation layer downloads this *ZIP* file, unzips it, and displays it to the user.

4.3.6 Text Search

The Text Search service performs *grep* operations on the source code repository. It contains one service operation (depicted in Figure 4.15) that takes in and returns a single parameter. These parameters contain all of the usage options and return values, which are created, parsed and displayed

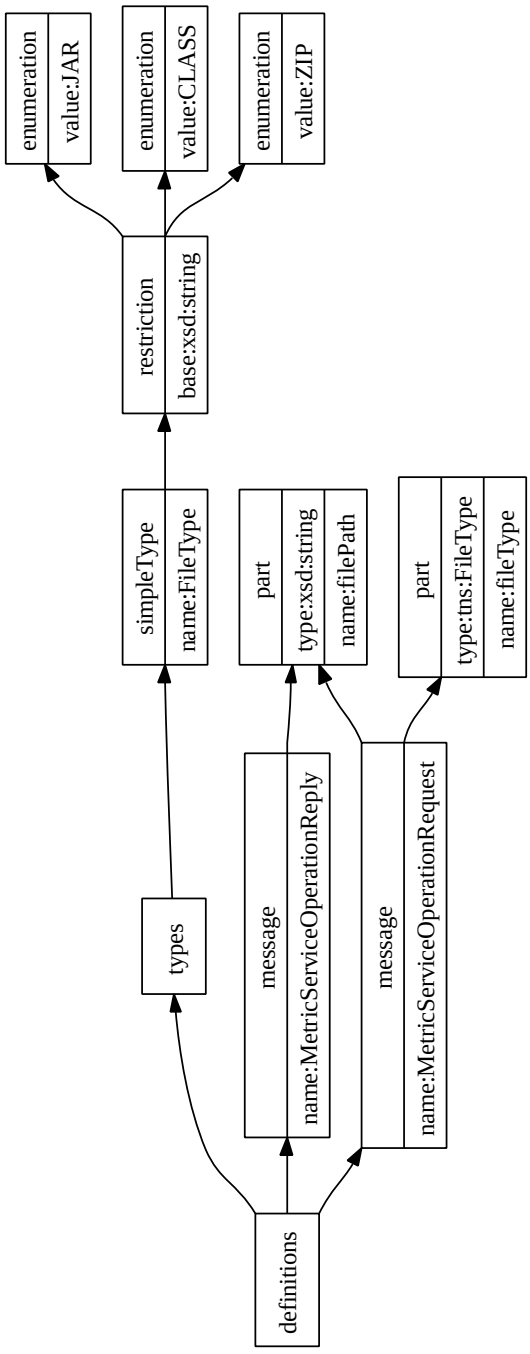

Figure 4.12: The Metric Service data types and structure

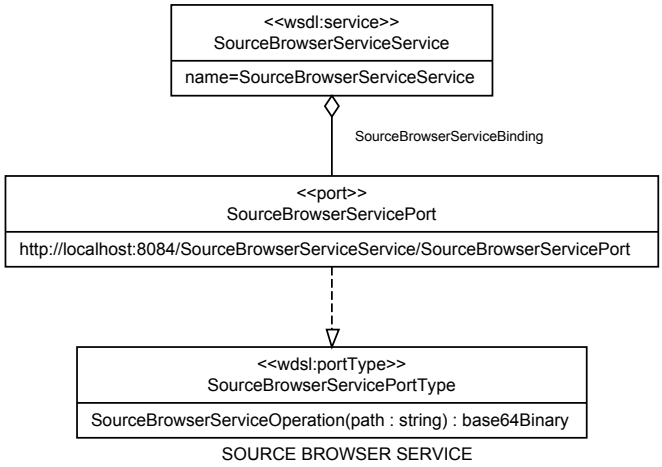

Figure 4.13: The Sorcerer Source Code Browser service definition

by the presentation layer.

The parameters: `TextServiceRequestType` and `TextServiceResponseType`, are shown in Figure 4.16. Each type contains a header, which is also shown, that contains the associated user's project information. `TextSearchRequestType` contains a *grep* search string, a case-insensitive search flag, and the standard header. After performing the search, the results are parsed and stored as a `TextSearchResponseType`, which also contains a header, a status flag called `info`, and a list of search results in the `contents` element. A similar list of file names is stored in the `files`, and the matching line numbers is stored in `lines`. These lists are printed in tabular form by the presentation layer.

4.3.7 Dynamic Analysis via Aspect Instrumentation

Aspect Instrumentation modifies the program's source code such that a logger traces the method invocations of a software system, yielding the dynamic call graph for a particular feature. This is accomplished by creating an aspect using AspectJ [46] for Java code, which runs behind a service and uses the static analysis results of the system to obtain a list of methods that the aspect should

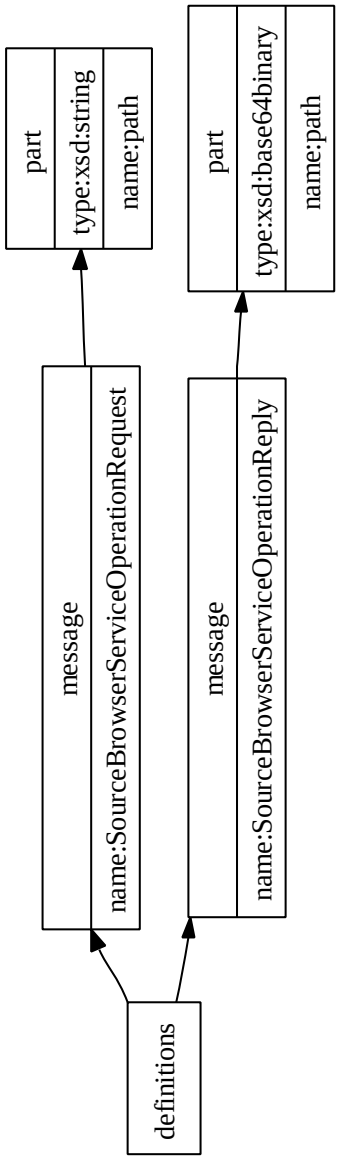

Figure 4.14: The Sorcerer Source Code Browser service types and structure

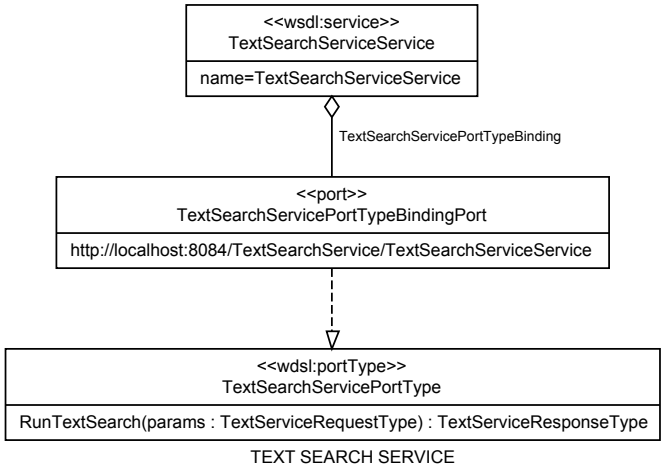

Figure 4.15: The Text Search service definition

instrument.

For security reasons, the user downloads the aspect, instruments the code on the user's local computer, and uploads the result which is generated by the aspect. That result is further analyzed to produce an XML graph that is visualized as described in Section 3.5. This process is detailed in Figure 4.17.

The service exposes two operations, shown in Figure 4.18. The first, `MakeAspect`, is invoked from the presentation layer. The BAT Static Analyzer service, described in Section 4.3.2, is automatically invoked to obtain a list of methods in the user's project. This list is displayed to the user as a list of check boxes, from which the user may select which methods should be traced by the aspect. If the user knows *a priori* which methods are used by the particular feature(s) to be traced, then the size of the result can be reduced by only selecting those methods. It is also acceptable to create an aspect to trace all methods, because only those methods invoked during the feature(s) trace will be included in the final result. This list of selected methods is passed to `MakeAspect` as its `MakeRequest` parameter, and the aspect is created. The generated aspect weaves into the selected methods, and writes the name of the called function to a file every time it is invoked, along with the name of the

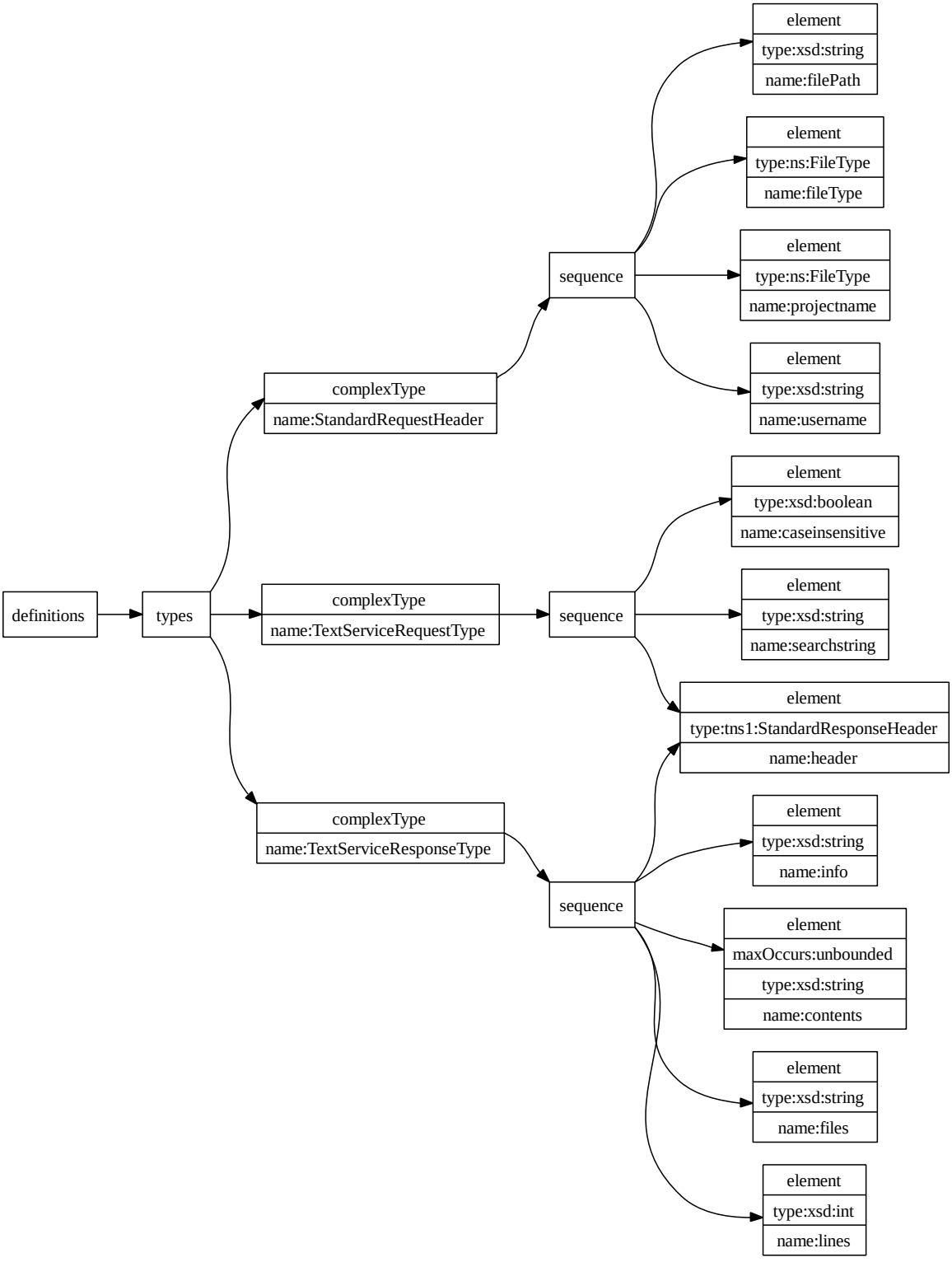

Figure 4.16: The Text Search Service data types and structure

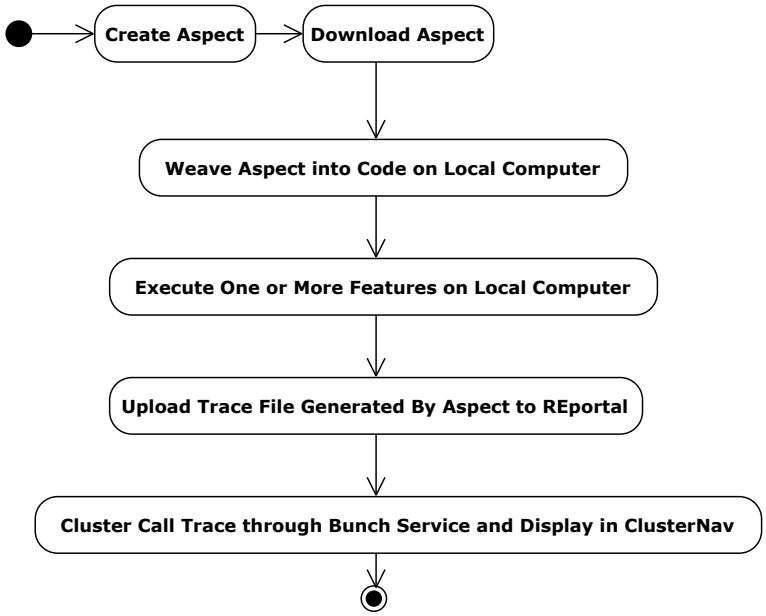

Figure 4.17: Activity diagram detailing the creation and execution of a dynamic analysis aspect, and its subsequent viewing on REportal

function that invoked it. Thus a trace file is generated in MDG format.

The MDG file is written to a file called `output.txt`, which is uploaded via the presentation layer back to the Dynamic Analysis service. This is done through the `AIGraphXML` operation. The MDG is passed via the `GraphRequest` parameter, clustered via an automatic invocation of the Bunch Clustering service, and the resulting GXL graph is returned to the presentation layer for display in ClusterNav via the `GraphResponse` parameter.

These parameters are described in Figure 4.19. The `MakeRequest` contains only a `MakeHeader` object, which contains information about the project, the name of the aspect, and the join points (which is the list of methods to be traced). The `MakeResponse` contains a header and the aspect code, stored in a `string` to be saved to the local computer. The `GraphRequest` contains the project name and the MDG file generated by executing the aspect-woven program. The corresponding `GraphResponse` contains one element: a `string` containing the GXL graph of the clustered MDG, to be displayed by the presentation layer.

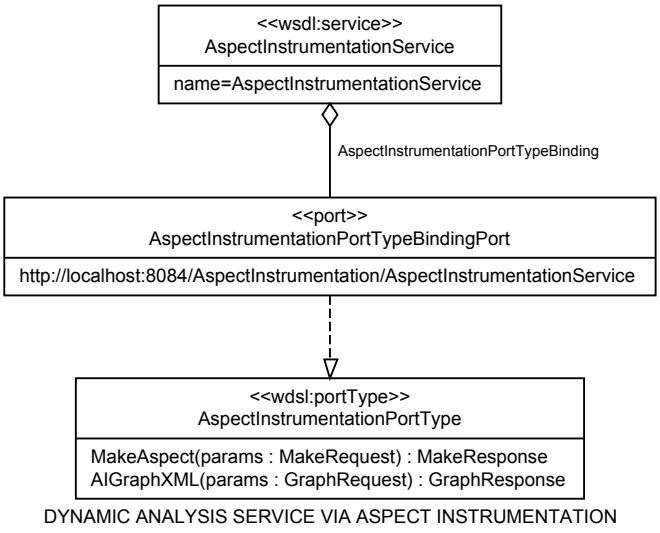

Figure 4.18: The Dynamic Analysis via Aspects service definition

4.3.8 Software Forensics to Determine Code Authorship

We have developed a series of tools to determine source code authorship [49, 47]. It is often helpful for intellectual property disputes, plagiarism detection, or general program comprehension to identify the specific authors of various portions of source code. Kothari, *et al* [47] and Lange, *et al* [49] have developed a technique through which a set of metrics are used to identify authors of source code. First, a **training set** (also called a **learning set**) is analyzed. This training set is a suite of source code whose authors are known. This information is stored about each file, and a series of metrics are computed on each file as well. From this known set of authors, the **testing set** is analyzed against the same set or a subset of metrics to determine which of the known authors most closely matches each testing set file. This tool is the focus of the case study in Chapter 6.

For REportal, a user's project may be considered the testing set. That is, the user may wish to determine which author wrote various portions of the uploaded project. To accomplish this, it is necessary to provide the Forensics service with a learning set of known authors. The training set is a *ZIP* file containing directories. Each directory is named for the author that wrote the files contained

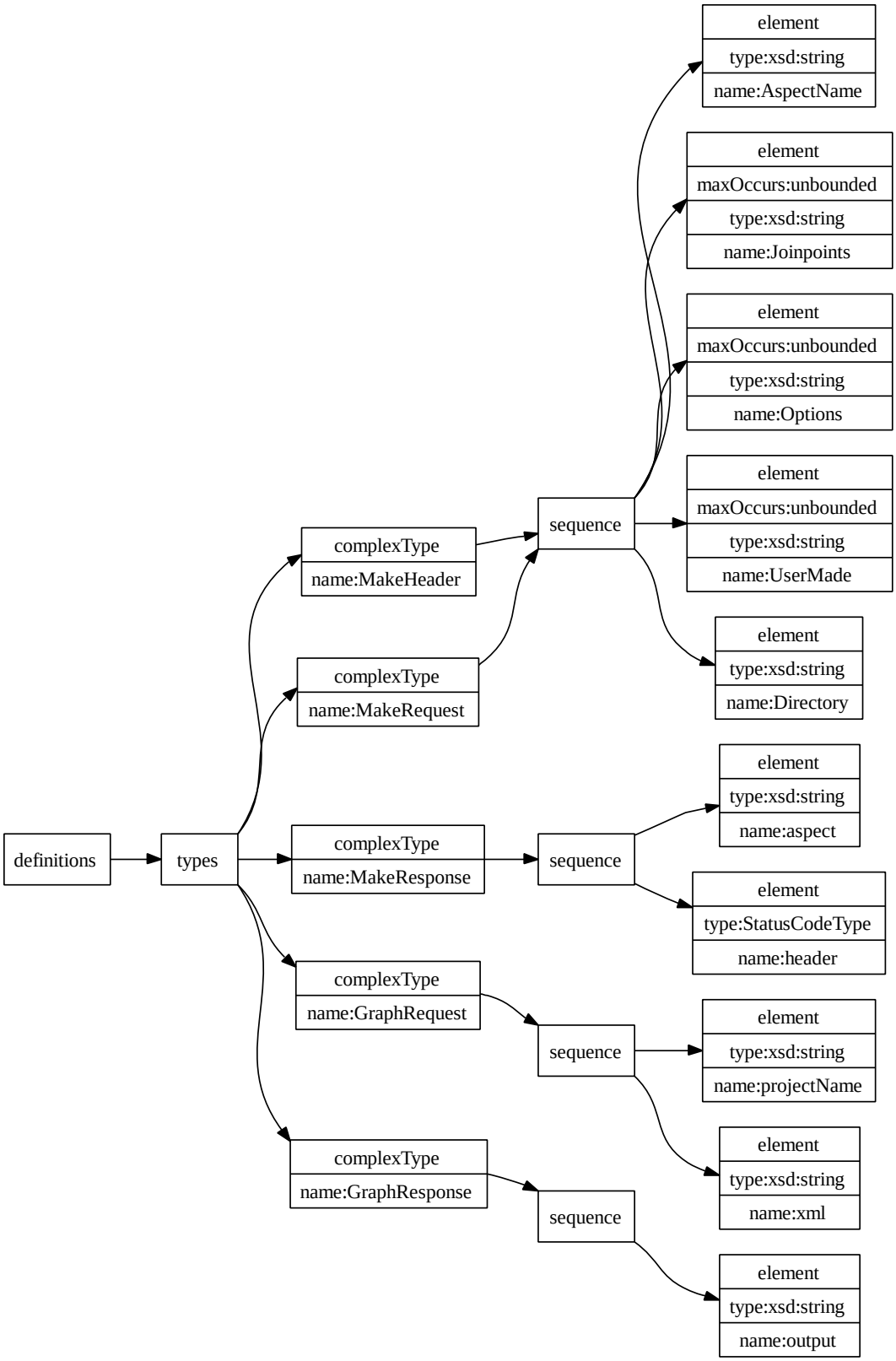

Figure 4.19: The Dynamic Analysis via Aspects service data types and structure

within. The service analyzes both sets and then returns a report in tabular form to be displayed at the presentation layer. An interesting extension to this service might be to use the entire source code repository of code that has been uploaded to REportal. If the authorship of each file there is known in advance, then it can be used as a training set against which new projects can be analyzed.

The Forensics service definition (Figure 4.20) provides a single operation. Given a learning set and a training set, both in *ZIP* format, the service runs and returns a `PredictionTupleList`, which is a list of file names in the testing set along with the name of the predicted author, and the confidence with which that author was selected over any other in the training set.

The Forensics data types definition (Figure 4.21) specifies several structures used by the service. As previously discussed, the primary parameters to the service are the learning and testing sets, which are each a binary sequence of data representing a *ZIP* file. The service reply message type is a `PredictionTupleList`, which is a sequence of one or more `PredictionTuple`s. Each `PredictionTuple` consists of the filename, the predicted author, and the confidence with which that author was predicted.

4.4 Database for User and Project Management

Project and user logistics management is provided by a language-independent service that maintains a database and maintains the repository on the local file system. Users are cross-linked with their projects, in which is stored the project's location on the file system, and the language associated with the project. This database is queried both to locate the user's project, and also as a security check to ensure that the project is owned by the user associated by the login token obtained when the user first authenticated. The database management layer does not depend on a particular platform or database engine, but the underlying database is provided by mysql [14].

The structure of the REportal database is shown in Figure 4.22. It contains four tables, which are:

- **KeyTable**: The `KeyTable` holds only one element. The `KeyString` is a `BLOB` that contains a

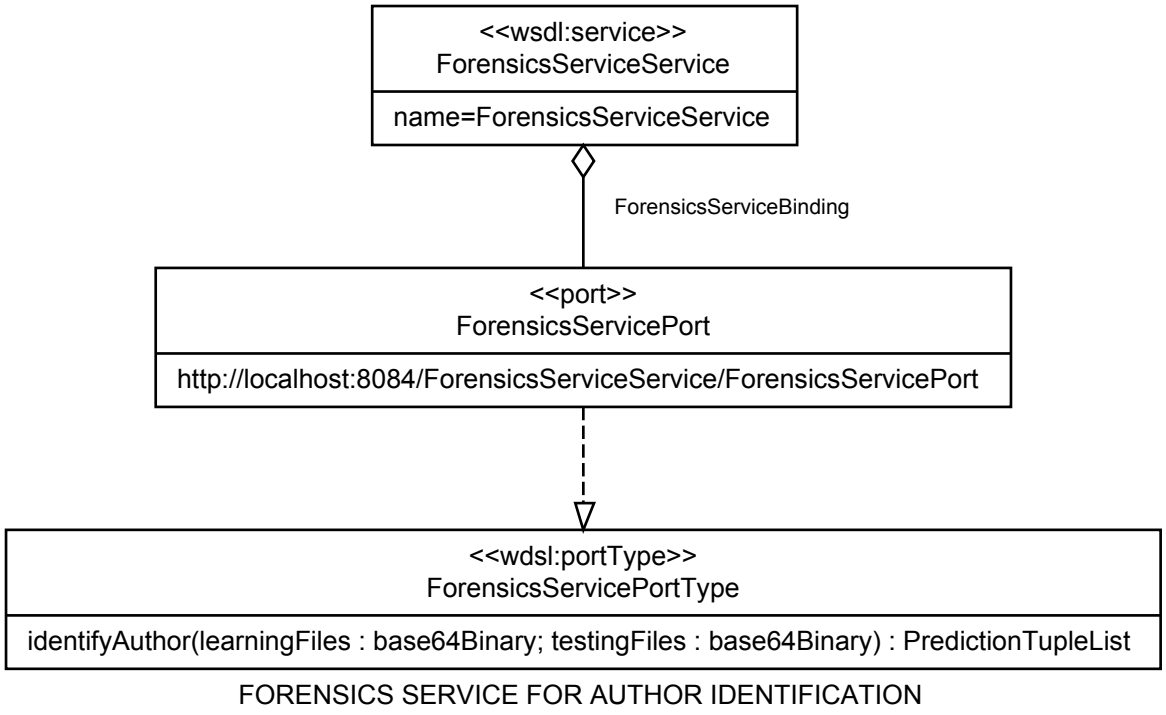

Figure 4.20: The Author Identification via Software Forensics service definition

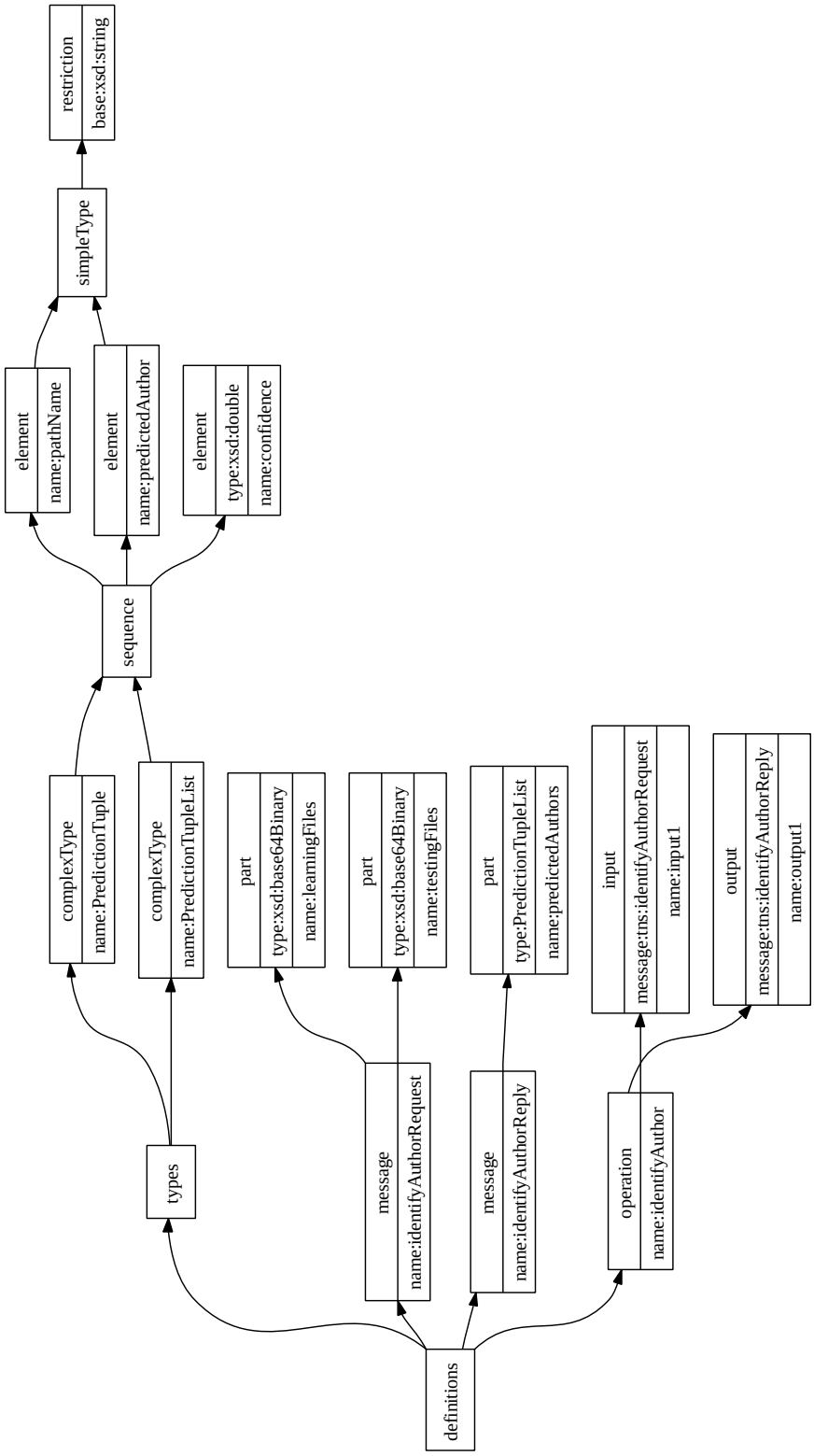

Figure 4.21: The Author Identification via Software Forensics service data types and structure

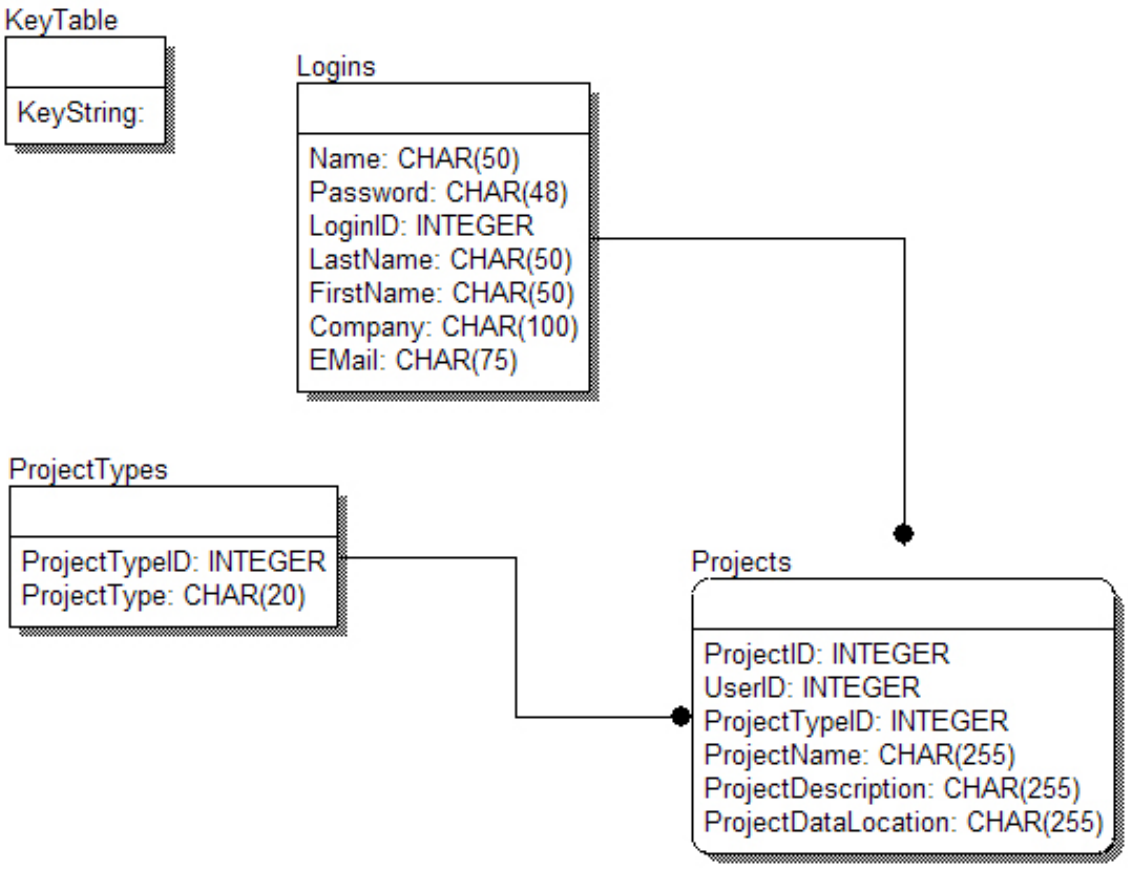

Figure 4.22: REportal database structure

binary encryption key to be used when encrypting and decrypting user passwords. The user password is only in decrypted form when the user first registers for a REportal account. From then on, the password is encrypted when it is entered for login, and when it is compared to the encrypted password stored in the database.

- **Logins**: The `Logins` table holds the user account information that was provided by the Project Manager service when a user reigsters for an account. These fields are described in Section 4.3.1.

- **ProjectTypes**: This table stores the enumerated list of project language choices, such as Java, C, C++, etc. It exists to map the language choices to integers, which are more suitable for use in enumerated types.

- **Projects**: The `Projects` table is the most significant of the REportal tables, and maps user accounts to projects, and project languages to projects, as seen in Figure 4.22. All the information from a `ProjectInfo` type in the Project Manager service (described in Section 4.3.1) is stored here.

4.5 XML Repository for Storing REportal Project Data

The XML repository for the REportal project data is based on the BAT Static Analyzer program discussed in Section 4.3.2. It is described in detail by the BAT authors [37], but we provide an overview here to add clarity to the query mechanisms offered by REportal. The schema describes the Java language elements, and the control flow of the code. Relationships between class entities such as method containment, variable containment, inheritance, control flow, call graphs, *etc.* are contained within the XML Schema.

As such, REportal provides a number of mechanisms to query this XML repository to return customized reports about the user's project. These reports can be displayed either in tabular form or graphically. This is discussed in Section 4.5.1.

For example, consider a basic Hello World program in Java, that prints *"Hello, World!"* to `System.out`, and exits. This provides the following BAT XML repository.

Table 4.1: Example BAT XML document for a Java Hello World program [37]

```
 1: <class name="HelloWorld" sourcefile="HelloWorld.java" visibility="public" >
 2:    <inherits><class name="java.lang.Object" /></inherits>
 3:    <method name="main" visibility="public" static="true" >
 4:        <signature><parameter type="java.lang.String[]" /></signature>
 5:        <code>
 6:           <get declaringClassName="java.lang.System" fieldName="out"
 7:                staticField ="true" type="java.io.PrintStream"/>
 8:           <stringconst><value>HelloWorld</value></stringconst>
 9:           <invoke declaringClassName="java.io.PrintStream"
10:                   methodName="println" >
11:               <signature><parameter type="java.lang.String" /></signature>
12:            </invoke>
13:            <return />
14:         </code>
15:      </method>
16: </class>
```

In this example, one can see that the `HelloWorld` class inherits only from `java.lang.Object` (line 2). In line 3, `main()` is defined as a `public static` method, taking a `String[]` array as its parameter (line 4). The code reads nearly identically to a heavily annotated Java program, beginning with line 5. The document contains so much structure that the queries written against it are highly readable.

One might expect that XML repositories can become quite verbose and, as such, take a long time to create. To mitigate this, REportal caches the XML repositories created by the BAT Static Analyzer and by the Metrics service subsystems, so that they are only created once.

4.5.1 Querying the XML Repositories

The Saxon [15] open-source XSLT engine is used to query the XML repositories obtained by the the tools provided by these REportal services. It was chosen because of its easy integration with JDOM, which was primarily used for integration with the tools. At present, the majority of the XML representations are simple wrappers around MDG graphs, but some XML representations, such as the BAT class file representation, are more complex and enable custom queries to return information such including reachability queries from a particular class.

As discussed in Section 4.5, the code repository XML Schema reads much like a Java program. Its queries reference standard Java constructs like `class`, `method`, *etc.* Consider the example in Table 4.2, which queries the XML repository and returns an MDG as a result. One will notice that the result is not in an MDG native format that can be passed directly into Bunch; rather, it is an MDG wrapped in a single `<MDG>` element. This is done because XQuery requires that a result be an XML document. The presentation layer strips this extra element away and is left with the data required to produce an MDG file. For this example, a line of the MDG graph would consist of the `source` data element, followed by a space, followed by the `target` data element. All of the queries proceed in the same way.

Table 4.2: MDG XML Query on a BAT Java class repository

```
 1: <MDG>
 2: {
 3:    for $c in //bat:class[@name]
 4:       for $x in $c//* where $x/@declaringClassName != $c/@name
 5:          return <relation type=\"{node-name($x)}\">
 6:                    <source>{data($c/@name)}</source>
 7:                    <target>{data($x/@declaringClassName)}</target>
 8:                 </relation>
 9: }
10: </MDG>
```

To find the relationships that exist among classes, the outer loop (line 3) iterates on all the classes defined in the project. It looks for all `class` elements in the XML document, which is an aggregate of all the classes analyzed. The inner loop searches for all XML element definitions inside of the current class (`$c`). Classes of the same name are not analyzed, as relationships between a class and itself are implicitly assumed. The rest of the classes are either method invocations, variable references, *etc.* As an example, see Table 4.1 lines 6 and 9. These correspond to a `get` variable reference and a method `invoke` command, respectively. The name of the element (`get` and `invoke` define the type of relationship that exists between the two classes. Thus, line 5 of Table 4.2 tags the `relation` to be the `type` given by the name of the node that has been found in the inner loop. The outer-loop class name is the source class, and the inner-loop class name is the target of the invocation or variable reference.

As a more complex relationship query example, consider the Reachability Query sample shown in Table 4.3. Although it is dense, this query accomplishes a reachability query in relatively few lines of code because the XQuery language allows for a recursive construction such as this. The class is stored as the `$x` variable (which may be `null` on a recursive call if leaf nodes or library calls are found), the root of the document is stored as the `$r` variable, and a list of currently inspected classes is stored in the `$foundSoFar` variable, to ensure that an infinite recursive cycle is not obtained [44]. This check is made in the first lines of the `local:reachability` function. The function then queries for any classes containing methods that invoke a method in the subject class `$x`, and recursively queries for classes that invoke methods in the invoking methods' classes. At present, the return type is an arbitrary XML document that wraps the essential information in a format that is easily parsed; this is done because there is no relationship information needed for an MDG-style representation. In other words, the return type is really a flat file indicating source and destination entities. Still, we discuss plans for a common XML Schema for capturing software entities and relationships in Section 7.2. The recursive call is handled by the union of an `invoke` element with a new call to `local:reachability` near the end of the `local:reachability` function, to query

the class we just found; this results in a transitive reachability query. Finally, the XML Query call to `local:reachability` is made with an initial parameter (`initialClassName`) specifying the base class for reachability. A similar query exists for `reverse-reachability`, in which the source and target elements of the query are interchanged.

Table 4.3: Reachability XML Query on a BAT Java class repository

```
declare function local:reachability($x as element()?, $r as document-node(),
           $foundSoFar as element()*) as element()* {
   if((count($x) > 0) and not ($x intersect $foundSoFar)) then
      <classReach name="{data($x/@name)}">
      {
         for $y in $x//bat:method
         for $z in $y//bat:invoke[@declaringClassName != $x/@name]
         return <invoke sourceMethod="{data($y/@name)}" targetClass=
                     "{data($z/@declaringClassName)}" targetMethod=
                     "{data($z/@methodName)}" /> |
            local:reachability
                     (($r//bat:class[@name =
                     $z/@declaringClassName])[1],
                     $r, $foundSoFar|$x)
      }
      </classReach>
   else ()
};

<reachability>
{
   local:reachability(//bat:class[@name =
                     '" + initialClassName + "'], root(), ())
}
</reachability>
```

It is also possible to use XSLT transforms to query a data repository such as the XML Java class representation provided by BAT. Using XSLT, one can use regular expressions to query data, as well as more verbose loop constructs that are typically easier to read and modify. So far, we have created XSLT transforms for the BAT repository to support entity and relationship queries among classes, methods and variables.

5. Testing and Validation

Service-oriented architectures provide several unique opportunities for software testing. Although it is not possible to guarantee behavior or performance regardless of testing quality, using traditional testing best-practices, we have found, fixed, and validated each feature of REportal. We have employed three testing techniques, and discuss this experience in Section 5.1.

We have also conducted a user validation study, consiting of typical users of REportal: software developers. Students in a Drexel University graduate course in Dependable Software Systems (CS576) became familiar with REportal, experimented with it, and executed their own tests on the portal. They documented bugs, errors, inconsistencies, areas of confusion, and so forth. These were taken into account while improving the portal and before executing our own internal tests. We describe this study in more detail in Section 5.2.

5.1 REportal Testing

This section discusses the various testing techniques performed on REportal. Because REportal is a large software integration project, consisting of numerous external tools, it is difficult to ensure bug-free integration and invocation of the tools as well as the correct operation of the tools. We can only expect the portal to be as robust and error-free as the tools on which it is built; however, in light of the many technical challenges in integration and deployment discussed throughout this document, it is not difficult to introduce additional bugs into the system. Although a detailed specification of REportal and its services via WSDL contracts does help to enforce that the service operation has been planned in advance, it is also possible that this specification is the source of more bugs. For example, during testing, we discovered that some of our services accepted too many inputs from the REportal presentation layer, and simply discarded those it did not need. It is debatable whether or not this is a bug in traditional terms, as it is unlikely to cause incorrect operation, but it is

an example of the software's architecture not matching its implementation. This, at best, makes the program more difficult to understand; at worst, it can cause a user to expect certain behaviors to occur when they do not. For instance, one of our services accepts a username along with the rest of its input, though it is not used. If this username is omitted or incorrect, the service will still behave normally. This would not be the expected behavior for one who is observing or exercising the software system. It is the goal of testing to create cases that exercise the system in ways that may not have been intended or even provided by the presentation layer, to expose bugs or other issues just like these. In this section, we detail our testing experience, including tests that originally failed when they were executed. Many of these were not "bugs" at all; in fact, nearly none of them were. Instead, they were issues of incorrect specification of the service (as described here), unnecessary parameters, *etc.* To us, they were opportunities to improve and refactor the service specifications and behavior of the portal.

The REportal testing strategy involved the following techniques: service testing (described in Section 5.1.1), which exercises the backend REportal services internally, at a SOAP-XML level; unit testing (discussed in Section 5.1.2), which tests the system at a feature level; and user interface testing (described in Section 5.1.3), which exercises the REportal presentation layer web pages.

5.1.1 Service Testing

Service testing is unique in that it tests services outside the context of the service client. Testing from the service client is more like what is accomplished with unit testing (described in Section 5.1.2). Service testing passes SOAP XML messages that conform to the contract specifications of the service WSDL. In this way, we do not benefit from the "protection" of the client, which may perform data validation, etc., for us.

This technique is particularly helpful for obtaining better code coverage, as we can send at least one SOAP XML service invocation message per WSDL operation, thus ensuring that each service operation is exercised at least once. Unit testing already helps us to verify that each feature operates

correctly under various conditions; therefore, we take advantage of service testing to exercise the service in the presence of "bad data." A typical REportal service test contains at least three invocation messages per operation, as follows:

- **Correct Message**: This message contains "normal" data that is expected to pass in REportal. If this test fails, then we know that there is a REportal configuration error or major bug that must be addressed before testing may continue. If configuration options are accepted by the service, then there may exist more than one **correct message**, in which each **correct message** exercises a different normal invocation of the service.

- **Invalid Data**: A copy of the **correct message** is made for each piece of input data in the message, and one input element is replaced with bad or invalid data (an incorrect path name, corrupt file data, a bad user ID, *etc.*). It is expected that the REportal service will return a message containing an empty result set or the appropriate status message. Typical failure conditions for these tests include the presence of a **SOAP Fault** message or an exception thrown by the service.

- **Empty and/or Null Data**: Finally, a copy of the **correct message** is made for each data input, and one data input per message is set to an empty string and/or removed completely from the service input message. These tests are meant to verify that the service is checking for empty or null inputs before execution. REportal is again expected to return an empty result set or an appropriate status message, and a failure condition would include a **SOAP Fault** response or an exception.

The SoapUI [18] tool was used to perform service testing. Given a WSDL URL, SoapUI generates a sample SOAP XML request message, in which one may fill in the parameter data to be passed to the service. SoapUI then sends that message to the service and displays the response. From this, a test case is generated with assertions to ensure that the response is a valid SOAP document, is not a **SOAP Fault** message, and contains or does not contain certain data. This is shown in Figure 5.1,

in which the request (shown on the left) attempts to login a user given the incorrect password, and the response (shown on the right) correctly shows an assigned `loginId` of -1, the failed login code.

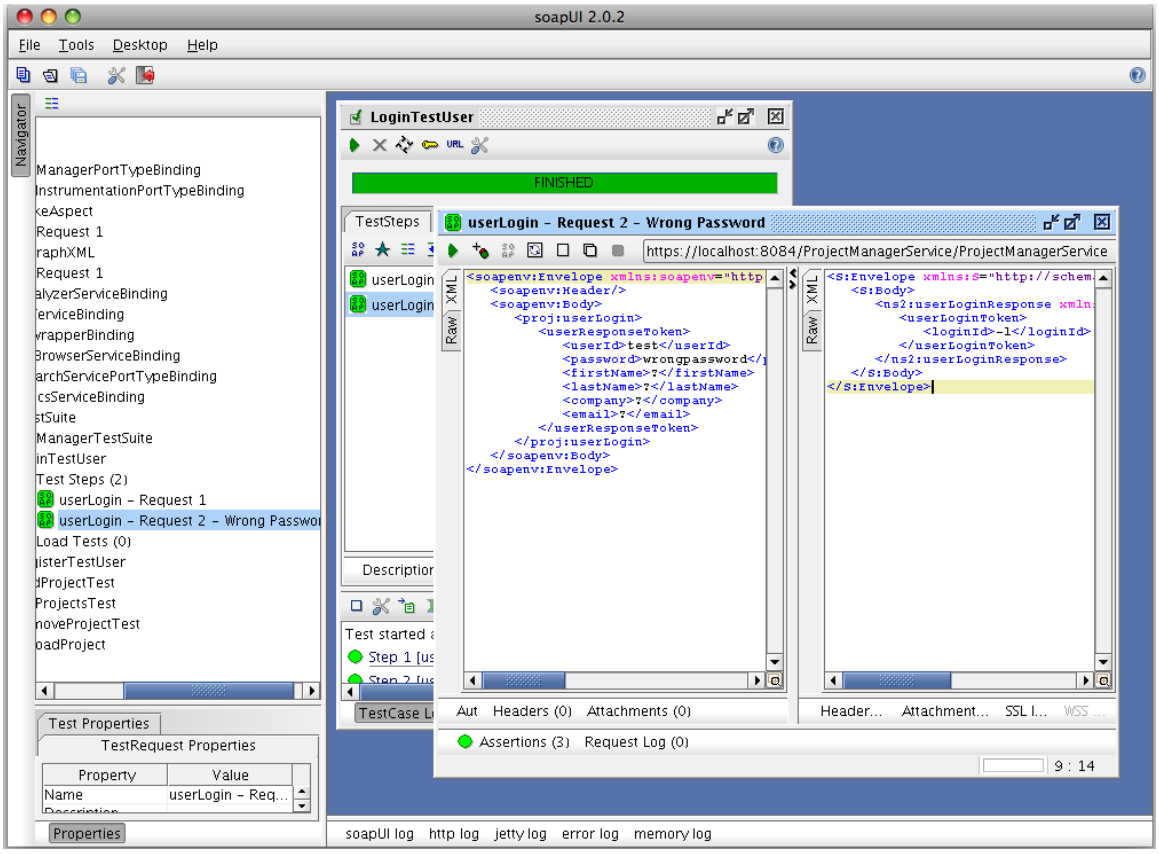

Figure 5.1: SoapUI service testing tool

SoapUI enabled the rapid creation of test cases for all of the REportal services. However, it was, at times, difficult to create test cases from the XML schema alone. For example, the Forensics Service requires base-64 encoded *ZIP* file data to be included as part of the message. This would be difficult to generate manually. It would be helpful in these cases to run *tcpdump*, monitor the service host port, and execute the service using the REportal user interface. This would generate a sample message which would serve as a better template. Because REportal uses SSL encryption

(see Appendix B for deployment and configuration details), this is not feasible. However, REportal is configured to use two ports per Tomcat instance: the default one using SSL encryption (ports 8084 and 8080 for the two Tomcat instances), and another one without encryption (ports 8443 and 9443, respectively). The Axis *tcpmon* tool, shown in Figure 5.2, can monitor a port by setting up a transparent pass-through port. In this case, port 58443 is set to forward to and monitor port 8443, and port 59443 is set to forward to and monitor port 9443. Then, one can reconfigure REportal to use these ports quickly by editing the *reportal.ini* file and changing the service URLs to be *http* instead of *https*, replacing 8084 with 58443 and 8080 with 59443. This causes clear-text SOAP messages to be sent to *tcpmon*, which logs the messages and forwards them to the appropriate service port to continue normal execution. The captured messages were copied into SoapUI for use as message templates.

Running these tests revealed a number of interesting but minor bugs evident in the services. Some of these issues are because the service contract asks for more input than is needed. For example, initial testing revealed that some services asked for a username, password, and project name, when none of these inputs are used. They were not used because this functionality was moved to the presentation layer so that one could manually invoke the services with non-REportal data, if desired. Nevertheless, when passing bad data in these fields, one would expect the service to fail due to an incorrect login, but it will not because the login credentials are not checked here. For service operations which require user validation, only a valid loginId (this is the unique user token obtained from the REportal project manager when the user first authenticates) is needed. This does represent a security vulnerability: if one could obtain another user's loginId token, it would be as valuable as obtaining the user's password in terms of gaining access to services that operate on the user's project. Minor checks of the REportal Project Manager and database would verify that the information provided to the service is correct. Specific results from our first execution of these tests included the following:

- **Project Manager**: 21 out of 29 tests passed. Particularly, adding a project under a user

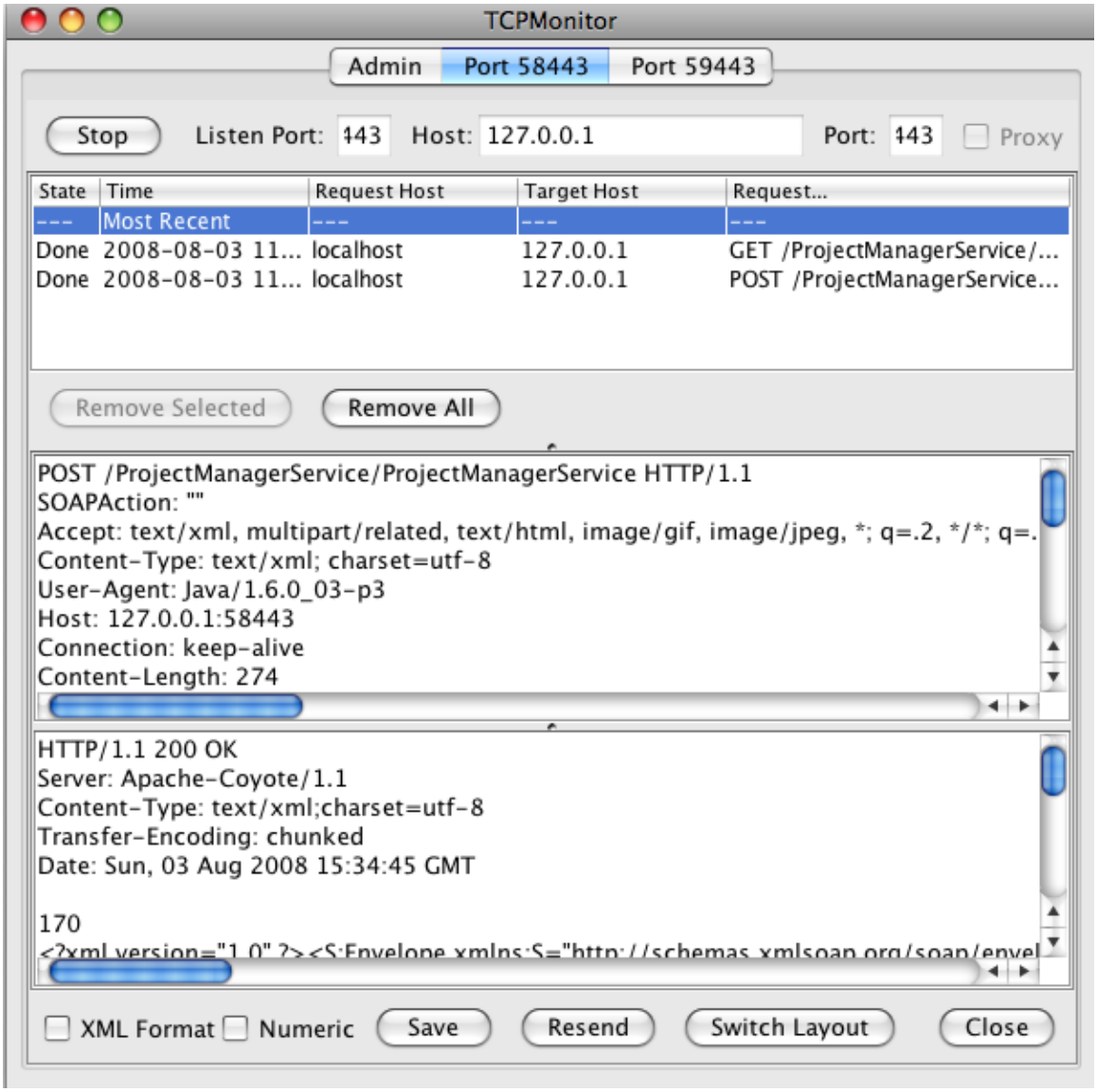

Figure 5.2: The Axis *tcpmon* SOAP port monitoring tool

account when an existing project of the same name already exists should fail, and it does not. A second project of the same name is created. This is problematic because both projects would try to use the same directory in the file system to store files. Additional failures were the result of the service working even when incorrect loginId or password credentials were supplied. For example, an incorrect loginId would cause the service to attempt to create a service in another user's file space. The remaining issues were minor; for example, the Upload feature would return an incorrect loginId as part of its return value, because this value is ignored. The best way to correct issues like this is to return to the service contracts and remove extra unneeded inputs and outputs.

- **BAT Static Analyzer**: 2 out of 3 tests passed. One bug was found in that, if the `FileType` input is left null, the service throws a `NullPointerException`. Simple fault checking will correct this type of issue.

- **Aspect-Instrumentation Dynamic Analyzer**: 3 out of 4 tests passed. The one failure was a result of the service not using the `FilePath` input parameter. This is because the creation of an aspect depends not on the project files, but on the list of methods chosen by the user. This parameter should not be part of the service contract, as it is ignored. If the `FilePath` field represents invalid data, the service continues to work, and this is considered a failure by our testing standards.

- **Forensics Author Identification**: All 4 tests passed.

- **Metrics Report**: 2 out of 3 tests passed. Like the BAT Static Analyzer, passing an incorrect or null `FileType` parameter results in a `NullPointerException` thrown by the service, and a SOAP Fault message returned to the presentation layer.

- **Source Code Browser**: 1 out of 2 tests passed. Providing this service with an incorrect path to the user's project files results in a `NullPointerException` or an exception when attempting

to unzip the code. This results in a SOAP Fault message returned to the presentation layer.

- **Text Search**: 7 out of 12 tests passed. The only failures were, as previously discussed, a result of providing invalid data to the service that is ignored. We assume that invalid data should result in incorrect results, but the service continues to run normally. Again, although this is not a failure from a functionality perspecive, it does alert us to the need to refactor the service contract to remove these unneeded parameters. At best, they detract from service comprehension.

5.1.2 Unit Testing

Unit testing was performed using JUnit [13]. We tested every operation of every service with several representative and boundary-case inputs using service testing, described in Section 5.1.1. This does not fully exercise each service since each operation may require several possible inputs to exercise every possible execution path through each operation. This reinforces the philosophy that software testing is intended to find bugs, not to guarantee bug-free software. Nevertheless, the goal of these unit tests is to exercise typical features of REportal and determine the behavior of each service.

To create each unit test, we first inspect the user interface component that typically invokes it. For example, the Aspect Instrumentation dynamic analyzer serivce is invoked by clicking on the "Dynamic Analysis" button on the REportal UI. This leads to the *analysis_aspect.jsp* web page. This page links to other pages, namely, the *uploadAspect.jsp* and *displayAspect.jsp* pages. The code in each of these pages effectively implements the dynamic analysis feature by invoking one or more web services, including the Aspect Instrumentation service. Therefore, our unit tests concentrate on using this code as a template.

Next, we inspect the use case for this feature in Chapter 3 to determine two things: first, that we are exercising most or all of the feature in our unit test, and second, that we can determine what prerequisites exist to using this feature. We discover that it is necessary to log a user in and open

a project before invoking this service. These features become the basis of the unit test `setUp()` method. There, we invoke the user login service operation with a valid username and password, and open a project.

Now, we have enough data to write our test methods. For this feature, the tests should exercise the generation of a method list for the aspect (this is the list of methods that a user can select to instrument for tracing). We need to be sure that the list of methods returned is a complete list of all the possible methods in the code.[1] The next feature to be tested is aspect generation. Given a subset of the list of methods just calculated, we want to ensure that the service is generating a valid AspectJ file to be instrumented into the code. The service is invoked and the result inspected for correctness.

From here, variations of the test case can be created, including boundary cases (empty method lists, *etc.*), bad data, operation of the service in the absense of the prerequisite `setUp()` conditions, and so on. The portal is expected to be robust under all of these circumstances, and unit testing reveals those in which operation is deficient.

The REportal unit tests are configured to automatically run when REportal is built, as part of its *ant* build script. The JUnit test code is placed into the subject service's *test* directory, where it is automatically executed.

The causes of failures during these tests were discovered during the more detailed service testing described in Section 5.1.1.

5.1.3 User Interface Testing

User interface testing is designed to ensure that the results computed at the backend are properly displayed to the user. Browser configuration issues or different browser rendering techniques may result in unreadable, or even incorrect results. This was a particular problem during the "browser

[1] In reality, this list of methods is obtained by invoking the BAT Static Analyzer service from within the Aspect Instrumentation service; however, we are assuming no such design knowledge so that we can test the portal from a feature perspective.

wars" when the original REportal was being developed, because a number of technologies used by REportal were not compatible with new experimental or very old browsers. The primary goal of user interface testing is to ensure correct operation across different web browsers.

The Selenium [16] web testing tool was used to accomplish user interface testing. Selenium comes with a browser plugin for several popular web browsers. It is capable of recording a user's actions (see Figure 5.3). As a user clicks a link, types in text, *etc.*, the Selenium plugin generates source code that will instrument a browser to repeat the same test. During recording, a user may select text, a figure, a table, *etc.* and instruct Selenium to verify that these elements are present in the proper location on the web page, in which case more code is generated automatically.

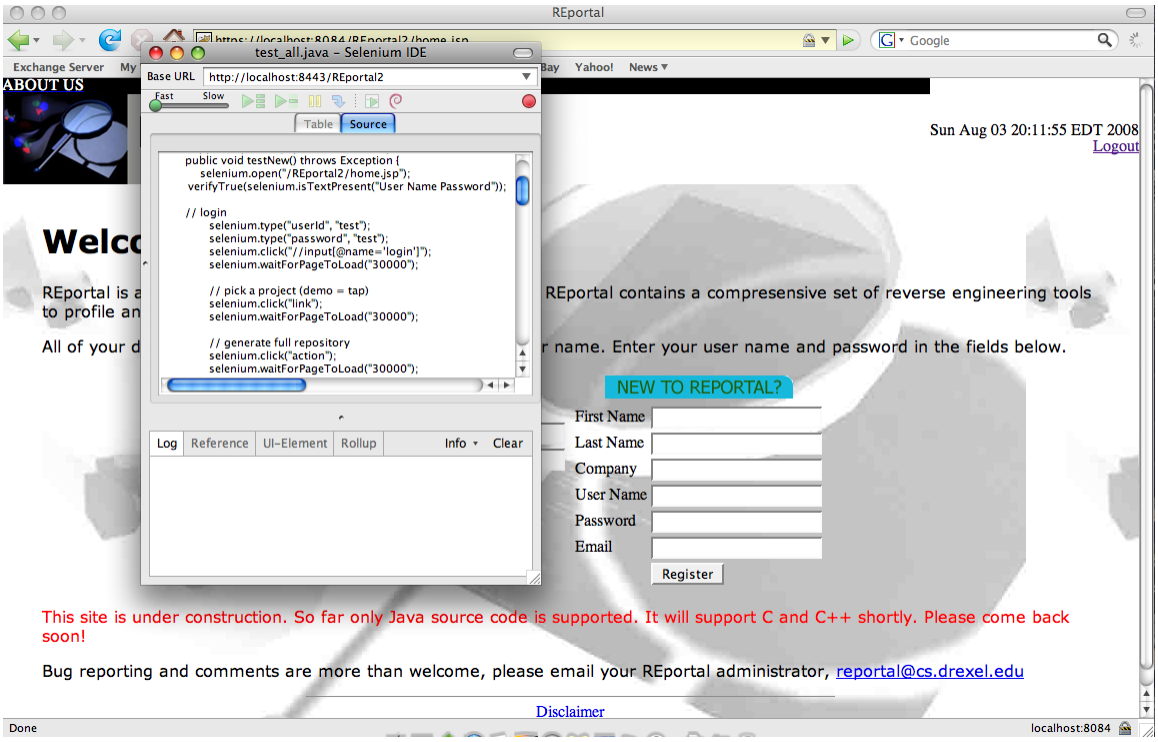

Figure 5.3: Selenium web browser recorder plugin for web testing

Each of the portal's features is executed and recorded by the plugin, and exported to Java test

code. The desired browser is selected as a parameter inside of this code, and executed within the Selenium framework. A Selenium server application acts as a proxy between the browser and the test case, and enables the test code to "drive" the browser, recreating the test. The code also validates the results by ensuring that figures or tables are in the correct locations, the proper text is present on the web page, and so on. These tests all passed on the first attempt.

5.2 REportal User Study

A number of students in the CS576 Dependable Software Systems course worked with REportal both to determine how easily they become familiar with the system, and to have them perform their own tests on the portal. Their work was quite helpful as this provided a perspective, both in usability and in testing, that is difficult to self-assess by the original author of the system. To become familiar with the portal, they first created a document of use cases for REportal, in which they documented the presence, usage, and prerequisites of each feature. Most students were able to identify and document all of the REportal features, even though their implementations were largely hidden behind underlying services and tools. As they were using a very early and largely undocumented version of REportal, some of them raised issues that were the result of lacking documentation showing how a feature was to be invoked. They also reported bugs that were not discovered during unit testing; for example, that registering the same user name more than once sometimes worked. This issue was resolved. Some of the students even reported that services were vulnerable to error when they were exercised independently of the user interface, because the user interface performed checks that the services took for granted. Students also indicated that, when a *jar* file is uploaded without source code, the metrics report and source code browser are largely empty.[2] This is by design, as these services depend on source code. This issue was worked around by introducing a decompiler that operated on no-source *jar* files when the project was opened for the first time. Finally, the early version of the new REportal included features that were meant for

[2]The metrics report will still compute binary or bytecode metrics that do not require the source code, but the remaining elements are not filled in.

debugging and development purposes; for example, it was possible to manipulate the BAT XML data repository through one of the JSP pages in REportal. This was not intended as a user-feature, but rather as a tool to test XSLT transformations that are the basis of the static analysis queries in the system. Users were not aware of this and some expressed confusion over the presence of these mysterious and undocumented "features." They have been removed for production release. As a result of this study, some user interface elements were moved for clarity, and a **logout** feature was provided.

6. Case Study: Adding a Service to the Portal

One of the primary benefits of the re-architected REportal is that adding a new tool should be possible without excessive difficulty. The original and re-architected REportal provided all of the tools described in Chapter 3 except for the Forensics service. The Forensics tool (described in Section 6.1) is an existing application to predict code authorship. As this has impacts for software maintenance and program understanding, it was an excellent candidate for an experiment in adding a new service to REportal. The goal, as usual with adding services to REportal, is to expose as little functionality as possible while reusing as much existing REportal data as possible. This experience is detailed throughout this chapter.

6.1 Forensics Application

As discussed in Sections 3.9 and 4.3.8, the Forensics for author identification application [47, 49] uses source code metrics to determine a author's coding style. When an unknown code sample is analyzed with these metrics, it is possible to predict which author wrote the code sample, given the set of known authors. Using this technique, the authors were able to obtain better than 70% accuracy when predicting the author, and better than 90% accuracy when predicting the top three most likely authors. The metrics used are text-based metrics, including the depth to which an author indents lines, use of underscores, and line lengths. After computing these and other metrics for the known authors, the data is analyzed to determine which metrics best characterize each known author – in other words, each author is assigned "characteristic traits." If these "characteristic" metrics are observed in a new piece of code, then it is possible that the author exhibiting those "traits" has authored all or part of that code. The extent to which a match is found, or to which an author is uniquely identified, is given as a confidence value.

There exists a GUI tool that invokes the application; however, it is rather difficult to use unless

one knows the order in which the various parts of the GUI are to be invoked. If a part of the program is executed out of order, a number of exceptions are thrown and the program crashes. Moreover, learning set files have to be added one author at a time, then followed by the unknown testing files to be analyzed (see Figure 6.1).

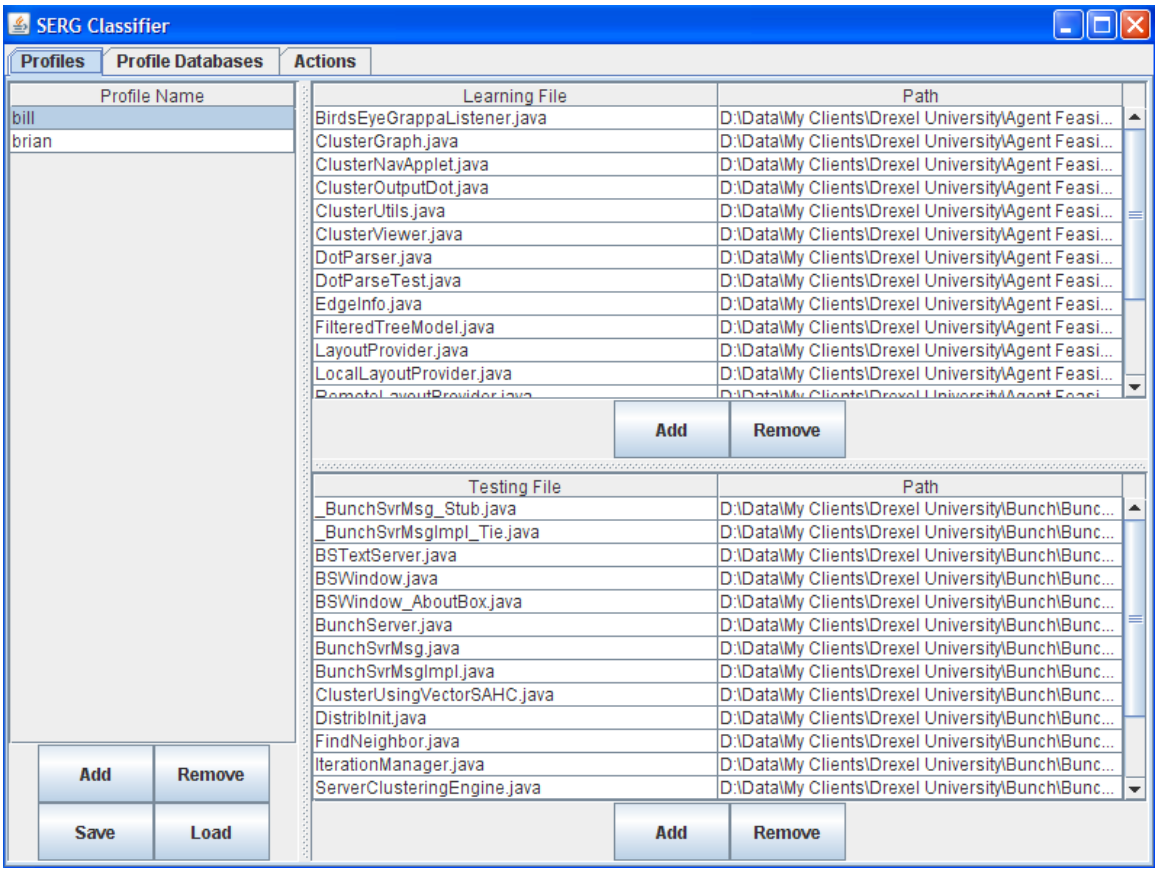

Figure 6.1: Setting up the Forensics GUI tool to use two learning profiles and analyze a testing set

Then, as shown in Figure 6.2, one must select which metrics should be analyzed and for which profiles, before running the report.

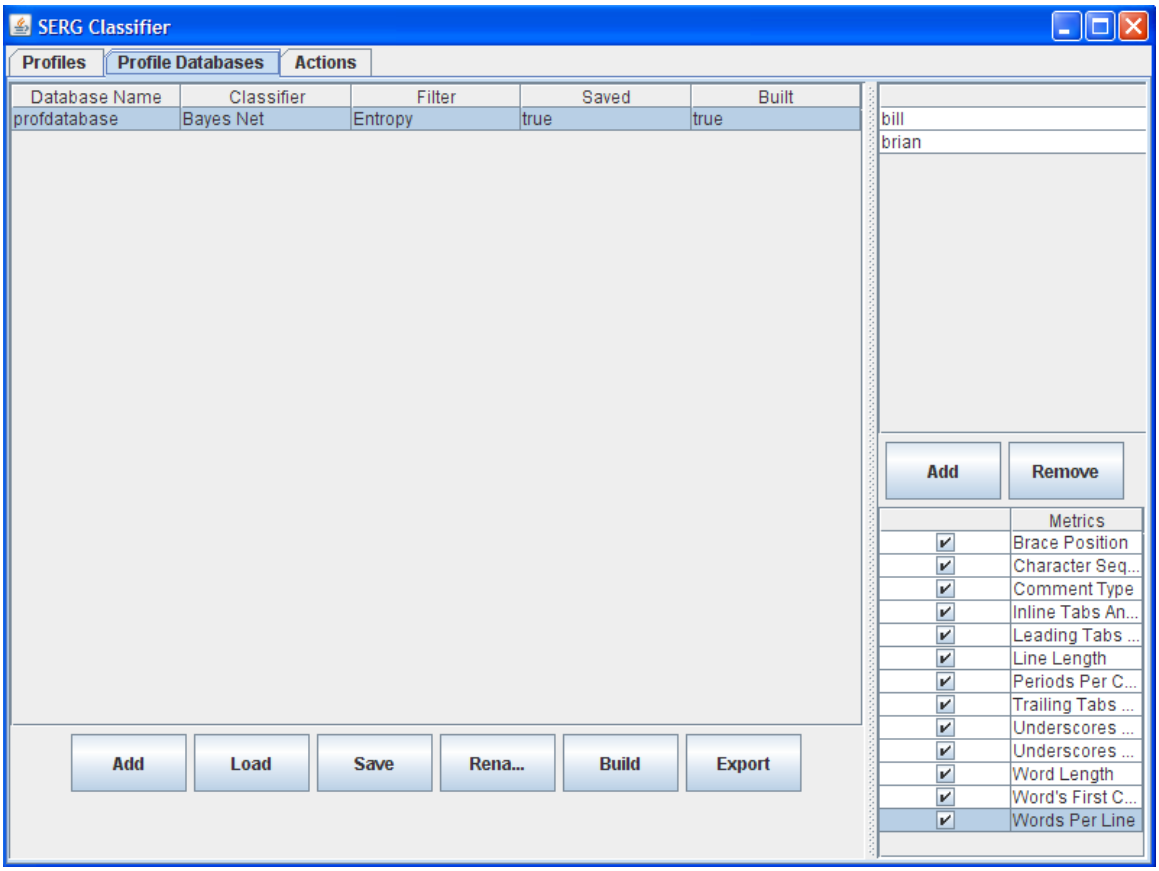

Figure 6.2: Setting up the Forensics GUI tool to use a database of metrics coupled to the learning profiles previously created

As a result it takes a great deal of effort to obtain a report. Because the spirit of REportal is to provide the high level business logic of the tool with the user's existing project, we wish to add the essential functionality of the system to REportal. To this end, there appear to be some opportunities for automatic configuration of the tool. For example, REportal could use the user's current project as the testing set of code to be analyzed. This way, the user only needs to upload

sample code with known authors to be compared. Moreover, the tool could be configured to try to use all of the avaiable metrics to obtain characteristic profiles, eliminating the need to perform any manual configuration on the tool.

We first identify the business logic functionality we wish to expose as a REportal service. In this case, we want a service that is capable of predicting authorship of the source files of the user's existing project. The only thing that should be required of the user is a set of learning code from which to predict. All other configuration options, profiles, databases, metrics, *etc.*, as described in this section, should be provided automatically by REportal.

Studying the actual Forensics application API used by the GUI tool, we find that a few configuration settings are required to set up the profiles and metrics, not unlike the GUI tool. Ideally, our API would expose functionality at the level of this business logic that we just discussed. To accomplish this, we wrote a small library called the *Forensics High-Level API*. The High-Level API provides only two methods to be used by the Forensics service, and is discussed in Section 6.2. It is responsible for invoking the original Forensics API to obtain the functionality we discussed here, and then return a report of predictions to the presentation layer.

6.2 Forensics High-Level API

The Forensics High-Level API is a library that mediates between the REportal Forensics service and the Forensics application. As seen in Figure 6.4, the Forensics Low-Level API provides a number of methods, of which the primary three are as follows:

- `createProfile`: This creates a learning set for each known author. Authors are attached to the files that they are known to have written.
- `execute`: This builds the configuration of metrics and maps it to the profile, learning set and testing set, and runs the metrics on the testing set.
- `predictFiles`: Based on the data collected during the invocation of the `execute` method, this

creates a mapping of unknown testing set files to their predicted authors from the learning set of known authors. Invoking this method requires that the caller has chosen a classifier, and mapped the authors to their files.

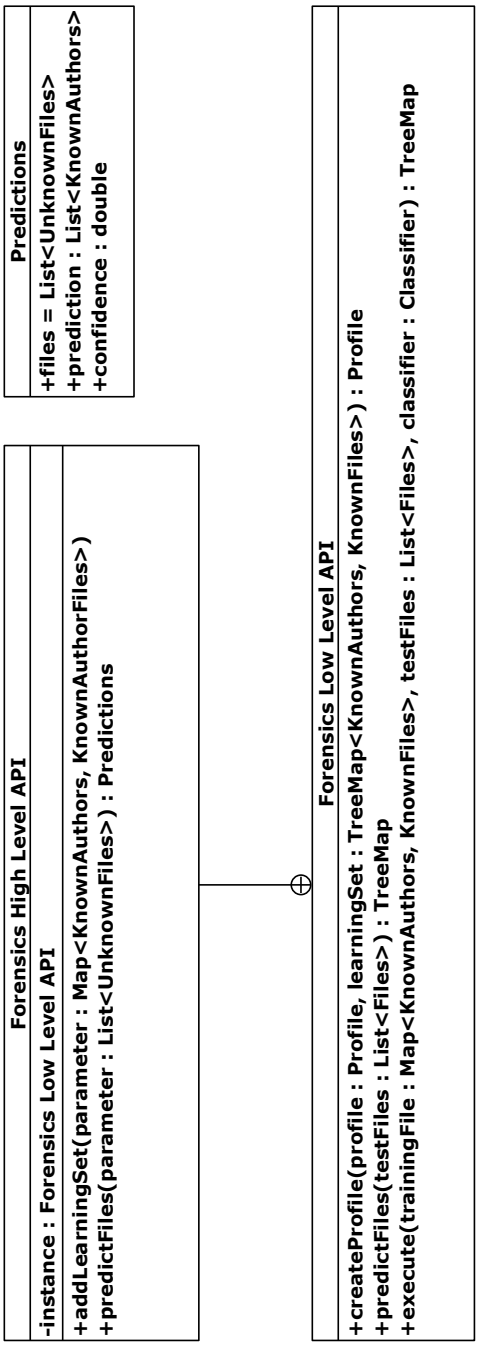

Figure 6.3: Class diagram showing the dependencies between the Forensics application and the REportal Forensics service via a mediating "high-level" API

We do not wish to burden the REportal presentation layer with the responsibility of configuring these profiles and metrics. Instead, the High-Level API exposes two easy-to-use methods, which are also seen in Figure 6.3:

- `addLearningSet`: This adds the map of known authors to known files. However, rather than create the mapping, the presentation layer will provide a *ZIP* file containing a directory for each known author, and each directory contains the files written by that author.

- `predictFiles`: This analyzes the list of testing set files and generates `Predictions`, which are created as part of the high-level API and shown in Figure 6.3, and consist of a simple list of the files, the predicted author, and the confidence with which that author was chosen. Again, rather than burdening the presentation layer with building this map (which would couple it to the service and to the tool), the presentation layer instead passes the list of source files from the user's current project.

Because REportal uses the data (such as the user's project) that it already has to invoke or implement a service, the presentation layer remains decoupled from the underlying services and tools. If a tool were to be replaced, the presentation layer would send this data in order to invoke the new tool, and the mediating API would be modified to translate this data into the data structures required by the tool. This decoupling is shown by Figure 6.4.

6.3 Designing the Service WSDL

Next, a web service contract is specified that uses the functionality of the high-level mediating API. The functionality exposed by the service is shown in Figure 4.20, and the WSDL can be found in Section C.4. Recall that the goal of our Forensics service business logic is to expose only one feature to the user: to identify authorship of the files in the user's project. We must map this single feature to the two methods exposed by the high-level API, and this is described in Section 6.3.1.

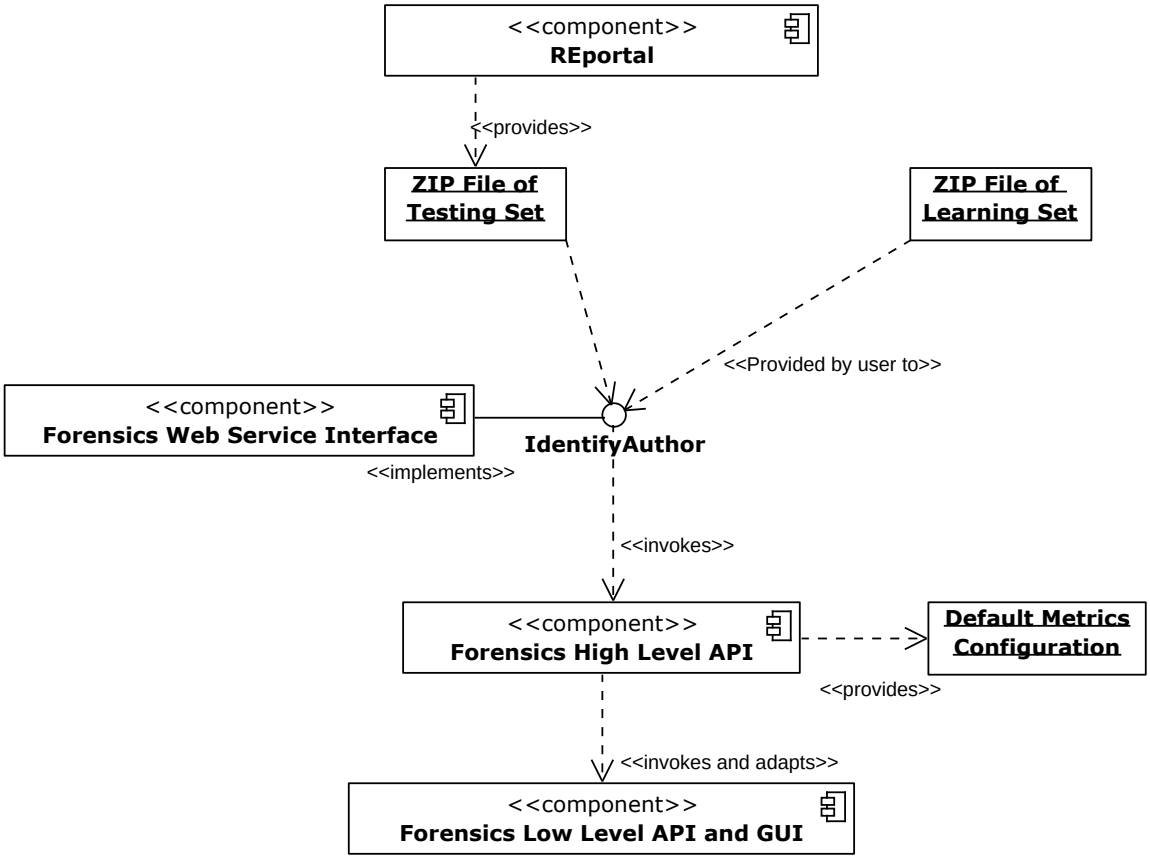

Figure 6.4: Interactions between the Forensics tool API and our Forensics service high-level API

6.3.1 Implementing the Service WSDL on the Server-Side

To implement the service, the WSDL is automatically implemented as Java code. This Java code contains one empty method, as defined by the WSDL. This method will identify authorship, given a learning set and a testing set of files. Recall also that we wish to only burden the presentation layer to provide these sets of files as *ZIP* files, with no additional configuration information. Therefore, the service implementation must unzip these files when it receives them from the presentation layer. The learning set directory structure is then parsed. For each top-level directory, a new author profile is created. The files associated with that author are those contained within the directory. Thus a Forensics application `Profile` can be created; however, to avoid coupling the service to the application, this is passed to the high-level API as a simple list of author names and files. At this point, the `addLearningSet` method of the high-level API is invoked, which creates all the `Profile`s required by the underlying application. This behavior is irrelevant to the service.

Next, the `predictFiles` method of the high-level API is called, and a list of `Predictions` is obtained. The `Predictions` list is parsed into the service return type, which is a report listing file name, predicted author name, and confidence for each file.

6.3.2 Implementing the Service WSDL on the Client-Side

The REportal presentation layer only requires the user to upload a *ZIP* file containing directories of known files, where each directory corresponds to a different known author. This is shown in Figure 3.29. The presentation layer creates a second *ZIP* file containing the user's project source code. These two zip files are the parameters required by the Forensics service, so the service is ready for invocation. The service invokes the high-level API, which in turn invokes the low-level API and the underlying application (described throughout this chapter and shown in Figure 6.4). A *for* loop iterates through the list of `Prediction`s, and writes them to a tabular report on screen, as shown in Figure 3.30.

6.4 Case Study Results

The author, along with a small team of developers not familiar with REportal, wrote the Forensics service independently of REportal. This was done purposely so as to avoid any inadvertent coupling to the portal, and also to better gage the difficulty of adding an arbitrary service, not just one specifically written for integration with REportal.

The author coordinated with the owner of the Forensics GUI application to write a "high-level" API specifically to interact between the service and the existing Forensics API. To do this, the Forensics service WSDL was studied to determine the data types and the single behavior that was decided on in the service contract. The API was written to interface with that contract in a way that was easily invoked by the service implementation. The result is described in Sections 6.2 and 6.3.1.

Next, the service was implemented so that it invoked the high-level API and returned the results as a simple list to the presentation layer. Finally, the service client was implemented automatically from the WSDL, and the service URL was added to the *reportal.ini* file so that it can be located by the presentation layer for invocation (see Section B.2.4). Integrating the client into REportal required a new button on the menu of REportal features, a new page that requested the *ZIP* learning set file and invoked the service client.

In total, this effort required approximately one week after training the developers, some of whom were neither familiar with REportal nor with Service Oriented Architectures. The experience was a success and demonstrates the reduced burden involved with integrating existing or legacy tools into REportal by identifying the business-level functionality and exposing it while reusing as much of REportal's existing data as possible.

6.5 Opportunities for Automation

From this experience, it seems possible to create a "wizard" tool that would analyze a legacy tool to find its public interfaces (this could be done via REportal's static analysis features, for example),

and map these interfaces to the existing REportal data model. The result would be an automatically generated WSDL, along with client and service stubs that invoke it. In doing so, the addition of legacy reverse engineering or program comprehension tools to REportal could be automated; this idea is described in more detail in Section 7.2.1.

7. Conclusion and Future Work

We discussed the original REportal system, and detailed the challenges that we faced in maintaining REportal and its underlying tools. We re-engineered REportal by creating service wrappers around its tools, and then integrating those services using a JSP-based web service client. This new architecture allows us to maintain REportal by using XML Schema to represent the data model, and by using XML queries to manipulate that data model. This facilitates adding new tools to the system by insulating them in a decoupled service layer, and permits heterogeneous REportal client instances that independently invoke the portal's services. Using REportal as a case study, significant potential in the area of SOA research was enabled, including a business process model and an XML representation for reverse engineering.

7.1 Benefits

Re-architecting REportal has resulted in several benefits for the portal's deployment, usability and maintenance, as well as opportunities for ongoing software engineering research. These benefits are described in this section.

7.1.1 Maintenance

The service-oriented architecture improves the usability of the portal and tools, as well as the ability to add or upgrade tools. If, as was experienced in the past, a tool becomes incompatible with current compilers, or a presentation feature becomes incompatible with current web browsers, only a simple drop-in code replacement is required. New tools do not depend on or impact existing features due to the separation of concerns inherent in the service-based approach.

7.1.2 Deployment

It is easy to deploy the entire system; client deployment involves distributing a single web client whose service URLs point to the deployed service locations on the web. Customers who are not comfortable with uploading their code to a remote service location for analysis may deploy the entire system on their local machines; this is also a simple matter of updating the web client's service URLs to point to these internal locations. The only requirements are a system with the required JDK, tools and any library dependencies, and an application server such as Apache Tomcat. In a heterogeneous tool environment such as REportal, multiple versions of JDK and/or multiple instances of Tomcat may be required to run the tools. Deployment details are described in Appendix B.

Because SVN was used as the revision control system for REportal's development, getting updates simply involves running an `svn update` command. The *ant* script builds REportal's presentation layer and its supporting services, and copies the `war` files to the Tomcat webapps deployment directory. This causes the services and REportal to be automatically updated, built and deployed. If the user desires to distribute the services among multiple servers, it is sufficient to edit the *reportal.ini* file to reflect the new service WSDL and mysql database URLs, and, if needed, change the Tomcat ports and REportal startup script to refer to the new Tomcat ports.

7.1.3 Heterogeneous Clients

It is important to note that web services do not necessarily imply a web-based solution as REportal provides. This is the most logical client to offer to users due to its ease of use and access; however, it is possible to write local command line tools or custom GUI applications in any language that run on any platform. As before, it is necessary to direct SOA clients to the deployment locations of the individual tools being used. Currently, REportal services for static analysis are being invoked without the aid of the web client via command line scripts that obtain the BAT XML repository and perform XSL transformations on it to obtain query results. These scripts were intended as drop-in replacements to the legacy Ciao scripts that had formerly been used, complete with the same

command line arguments, etc.; merely the implementation of the scripts must change to invoke a service that returns an XML dataset.

7.1.4 Contributions to SOA Research

One primary contribution of a service-oriented web portal is to serve as a case study for ongoing research into Service Oriented Architectures. For example, in creating REportal, we have described the "business model" for performing software analysis on a subject system. This model includes uploading the project, performing one or more common queries for relationships, metrics, dynamic analysis data, *etc.*, and then visualizing that data. REportal provides an extensible framework for carrying out this business model, much like a travel agent provides generic functionality for booking travel details.

Currently, REportal parses the output of these tools into XML formats for easy adaptation, but even from this we can identify certain pieces of information that are more often essential to the task of performing reverse engineering tasks. Other data (such as the Java bytecode that implements a particular method) might be included by a tool but not essential for our portal. Ideally, we can filter out only what is necessary for a user's anticipated needs, in order to boost performance of the system. It is also possible to create an complete XML Schema describing data that is required (and optional) for certain activities, boosting compatibility between the tools, the portal, and external work in reverse engineering.

This service-based portal also enables semi-automated service addition, binding, and orchestration. Currently, given a new service to be integrated, one must a) construct a service client that invokes the service and wraps its results in a data structure, b) invoke that service client from a menu in the presentation layer, and c) display its results. If the tool does not have a corresponding service, a WSDL providing a few methods to exercise the primary features of the tool is constructed, resulting in an automatically generated service wrapper that is compatible with REportal. To display a graphical representation of the tool's output, an adapter is needed to construct an MDG graph

from the tool's output, and this MDG graph is run through the Bunch service, producing a GXL graph that is displayed in our internally developed ClusterNav viewer. However, even this process can be improved, and the new REportal provides a test bench for such improvements. For example, since service generation for existing tools is a nearly automated process, a configuration file might specify the interfaces to a tool's primary functionality (or they can be obtained by performing static analysis to find system methods invoked by the `main()` function), and the WSDL wrapper would expose those features along with their input and output types. Moreover, it might be possible to construct the presentation layer on the fly, invoking the WSDL methods and displaying the output results directly to the user in a tabular representation, or as an MDG graph that is run through the Bunch clustering service for graphical display. This is the focus of future research, using a configuration file that specifies how the service dispatcher should handle the input and output types of the tool, and is described in Section 7.2.

7.1.5 Reverse Engineering as a Business Process

Another contribution of REportal as a service-oriented architecture is that, as the "business model" for software analysis evolves with new tools, that business process can be expressed as an abstract BPEL process, in which generic process flow is described without specifying the actual services that implement it. That BPEL can be filled in dynamically by REportal, removing much of the small amount of coupling that still exists between REportal and its tools. Instead, REportal can choose between the tools using a database of configurations that specify how to bind to the services and pass relevant data (i.e., via the proposed XML Schema), or ultimately by dynamically locating appropriate services using UDDI service registries and semantic service descriptions.

7.2 Future Work

Section 7.1 described a number of contributions to both the REportal system and to users of reverse engineering tools. It also serves as a "proof of concept" for automating a service oriented

business process, and provides a case study upon which SOA automation techniques can be applied. There is some current research on the automation of SOA BPEL models, and automating service orchestration; however, for our long term research goals in SOA, it is helpful to have a well defined universe such as reverse engineering in order to avoid the details of semantic descriptions. We are more interested in working with the dynamic selection and binding of tools, in which an important detail is to identify the interfaces, input and output types.

To facilitate this, however, an XML Schema is necessary to provide a common adapter between the tools. The services must provide either a common interface to the primary data artifacts, have a configuration that specifies how to get this data, or have a semantic description by which this information can be determined at runtime. Because we have chosen reverse engineering tools as our domain, we can specify an XML schema that would benefit both communities.

Using SOA, it is possible to instantiate these ideas in place of the current presentation layer, without disrupting the existing system. An automated, BPEL-based thin client can co-exist with the current thin JSP client, which co-exists with the existing command line interface to the tools. Such a BPEL client is another area of future research based on this project, and will construct abstract BPEL templates on the fly based on the tasks to be performed. It then uses dynamic service binding to find and fill-in the actual services from a database. Again, using reverse engineering as the proof-of-concept domain enables us to pre-populate this database with tools and services to test the BPEL and WSDL generators as well as the service binding mechanism.

7.2.1 WSDL-Wizard Interface Design for Automated Composition

However, one challenge to a service-based portal is that many tools are not written as services; particularly, none of the tools currently deployed on REportal were originally web services: they were standalone console tools or tools within a GUI interface. It was necessary to wrap WSDL documents and associated service implementations around the tools. These implementations tend to be rather simple, as they specify the inputs and outputs of each tool (which would soon be compatible with a

common data model as described in Section 7.2), and the calls to the tool or the tool's API, if one is available.

This "service wrapping" process may seem straightforward; however, if the service composition process is automated, it would make sense to also automate the process of wrapping the tool into a service to be composed. We propose a tool for this as well called WsdlWizard. WsdlWizard will take a tool and wrap it into a service by reading the `main()` function for its top-level function calls (called "Front-Line Functions" or FLF [34]). These front level calls become a WSDL whose implementation passes the inputs and outputs between the WSDL and the API as modeled by the `main()` function. The result is a slightly bloated WSDL, because not all front-line functions were intended to be top level interface invocation points as defined by the design.

This task is straightforward; however, an additional challenge lies in that REportal integrates a heterogeneous set of tools. It would be an administrative convenience to use the same language for the purpose of hosting the services on a homogeneous set of application servers, although this is certainly not a requirement of REportal or web services. To integrate heterogeneous tools, it is necessary to wrap the tool as an object library and use the Java Native Interface to wrap the tool as a Java class, which is then wrapped as a service for invocation. To date, a proof of concept tool has been written to convert C code into JNI-enabled Java. We have authored a prototype tool to convert C++ classes to equivalent C calls, which are in turn compatible with JNI-enabled Java. The drawbacks to this approach are bloated machine-generated code, hard to understand generated code, and code that is wrapped at several layers. However, even a C wrapper will often be faster than equivalent Java code, making Java the limiting factor. Because this code is executing over the web in a Java application server, we postulate that this added complexity is not more than what already exists to send the data over the network through that application server. Nevertheless, this is an area where serious performance improvements are possible. Additionally, we argue that the machine-generated code would not be read by a human, and can be re-generated whenever the host program evolves; therefore, this too is hardly restrictive.

Appendix A. SOA Overview

Service Oriented Architecture (defined in Section 2.2) is an implementation-independent architecture for distributed computing. SOA enables one to describe a contract, specifying the functionality, inputs and outputs of a software system at the business-logic level. Services consist of components or managed code that runs in a container such as an application server. These components may be, for example, Enterprise JavaBeans (EJB), CORBA components, or COM objects. Similarly, a number of application server containers exist, including Sun Application Server [20], the GlassFish project [7] and Apache Tomcat [3].

Distributed services[1] take advantage of XML data exchange, as opposed to older methods including binary data exchange. Moreover, SOA does not prescribe a particular transport mechanism; although TCP is the most popular choice, there is no transport protocol requirement. One could use a UDP connection to send messages, HTTP over TCP, SMTP, FTP, and so on. This flexibility facilitates a heterogeneous and asynchronous service environment. Of course, one requirement is that the decision must be made, and agreed upon, by the service and caller.

Figure A.1 shows some of the primary components involved with specifying, implementing and deploying a service oriented architecture. Each of these components are described in this chapter.

A.1 Data Type Descriptions: XML Schema

An XML Schema Document (XSD) is created to reflect the data types that will be used throughout the system. Implementation-independent complex data types can be represented via an XSD structured document. In the context of SOA, these data types describe the messages that are passed from a caller to a service. The message structure determines the functionality to be invoked within the service.

[1] Taken in part from http://www.cs.drexel.edu/∼bmitchel/course/cs575/lectures/SOAReferenceArchitecture.pdf

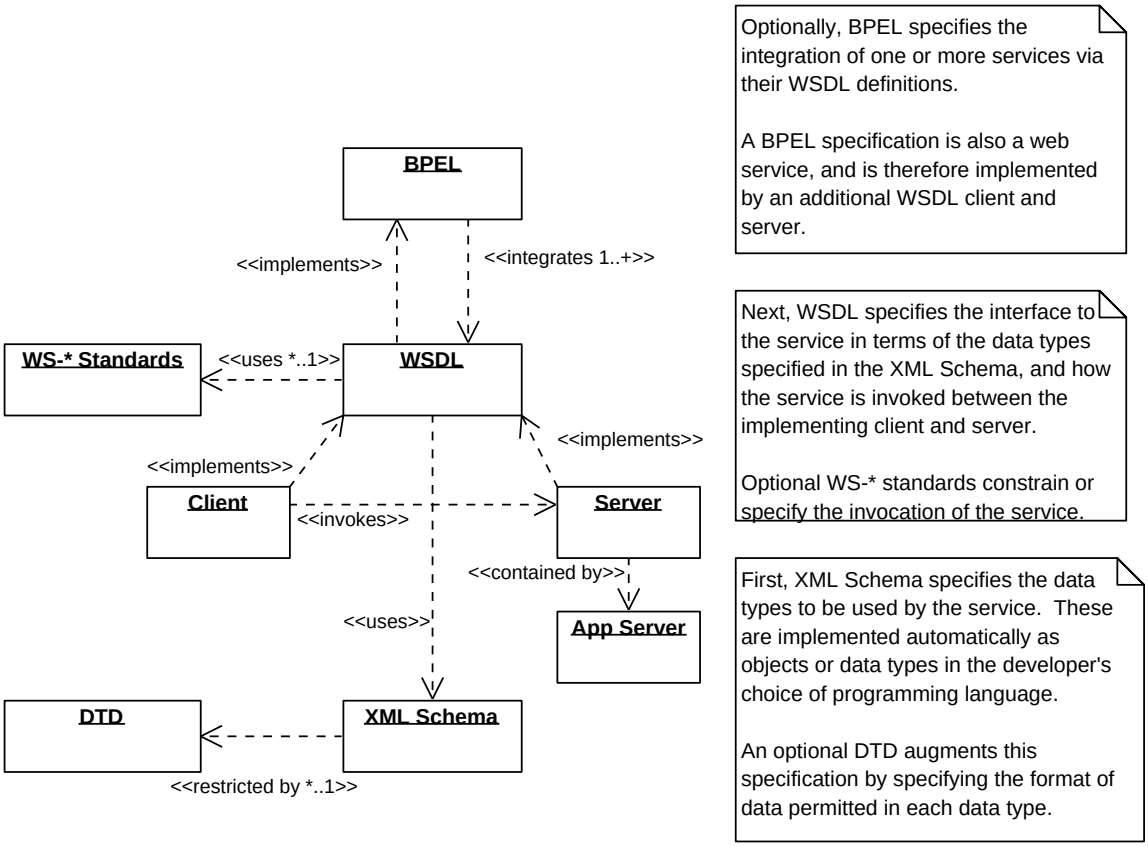

Figure A.1: Hierarchy of primary web service components.

For example, suppose we create an XSD that includes a type representing a sales order, and that type contains the item number, buyer's name, address and credit card number. We could then create a corresponding message type in our WSDL (see Section A.2) that conforms to this schema. In other words, we would produce an XML message identical in structure to the XSD, but includes the appropriate data as well. This XML message would be used as part of the service invocation to make a purchase.

For now, decisions related to XSD construction are limited to the design phase and include only the abstract types to be created. In this way, they would map closely to the JavaBeans concept, in which the types represent the data to be used.

A.1.1 Document Type Declaration (DTD)

XSD can be further augmented to include constraints such as valid data format and structure, among others. This allows for data type validation constraints to be placed at design time. This is accomplished through the XML Document Type Declaration (DTD).

Consider the following DTD example describing valid postal addresses [74]:

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE address [<!ELEMENT address (#PCDATA)>]
```

and the corresponding XML implementation:

```
<address>
Mr Ed U. Cate
12 Soap Street
Service City
B1 1AA
United Kingdom
</address>
```

Although this is a valid XML document per the DTD given above (note that PCDATA means any "Parsed Character Data"), XML Schema allows for sequences of mandatory and/or optional data elements through the DTD. Thus it is possible to describe the data elements of a postal address, and their proper format. Consider a better DTD for the XML Schema, as follows [74]:

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE address [
<!ELEMENT address (name+, street, city, postal-code, country)>
<!ELEMENT name (title?, first-name, last-name)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
<!ELEMENT country (#PCDATA)>
]
```

This DTD more precisely describes a postal address as a sequence of entries of character data, in a hierarchical format. Particularly, an `address` is described as a sequence of one or more `name`s, followed by the `street`, `city`, `postal code`, and `country`. The resulting XML Schema document would create data types that reflect the DTD above, and resulting XML messages would contain data properly constrained by the DTD.

A.1.2 Schema

While it was possible to restrict the structure of XML messages using a DTD, it is not so easy to use the same DTD to place data validation restrictions on the message [74]. To do this, the actual types that were constrained by the DTD are created in XML schema, in which these further validation restrictions can be placed. For example, building upon the DTD in Section A.1.1, we may use a `restriction` to constrain the zip code `simpleType` to be a sequence of five digits, a dash, and a sequence of four digits:

```
<simpleType name="zip">
<restriction base="string">
<pattern value="[0-9]5-[0-9]4"/>
</restriction>
</simpleType>
```

Similarly, `complexType`s can be created that represent entire classes. Enumerations, `sequence`, `choice`, and `all` constructs are supported by XSD. Further, other data restrictions are possible

including minimum and maximum length of the data, number of data elements of that type to be included in each message (for array type processing), and so on.

The purpose of the XSD is to be imported into (or written directly within) a WSDL description document, and serve as the basis for the messages that are passed to and from the service. This is described in Section A.2.

A.2 Service Description Language: WSDL

The Web Services Description Language (WSDL) builds upon the design considerations expressed in the XML Schema in Section A.1. Despite its name, WSDL is applicable to services in general, not just to web services. It separates "what" a service does and expresses it separately from "how" and "where" the service does it [68]. As is described in this section, a WSDL describes the data types (Section A.2.2) that a service uses (usually imported from the XSD schema, although the XSD schema can also be written directly in the WSDL), the messages (Section A.2.3) that are either sent or received by the service (these conform to the data types also described by the XSD schema), the operations (Section A.2.4) the service supports and messages that are passed as parameters and sent as return values, the protocol(s) (Section A.2.5) over which those messages are sent (SOAP over HTTP, *etc.*), and finally the physical location (Section A.2.6) at which the service is accessible. Together these form an XML representation of the service, structured as shown in Figure A.2.

It is important to note that there is a great deal more flexibility and functionality offered by WSDL (and XML Schema, for that matter) that is not described here. This entire section is meant only as a quick overview of the available technology.

A.2.1 `definitions` Section

The WSDL definitions section serves as the root XML element of the document. It gives the namespace and other header-type information about the WSDL document to follow. One or more `type`s, `message`s, `portType`s, `binding`s, and `service`s will appear in this document to define the

entire service.

A.2.2 `types` Section

As described in Section A.1, WSDL types are defined by XML Schema or any other type definition language. Due to the popularity of XML Schema for describing WSDL types, we restrict our conversation to this language only. Other languages for type description include RelaxNG, DTD, the IME type system, OMG IDL, or COBOL copybooks [68]. These types can include primitive data types, complex defined data types via XML Schema, or MIME-type binary encoded data elements.

An example of a WSDL `types` section is provided below:

```
<types>
<xsd:schema targetNamespace="http://foo.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<complexType name="address">
<sequence>
<element name="street" type="string" minOccurs="1"
maxOccurs="2"/>
<element name="city" type="string"/>
<element name="state" type="string"/>
<element name="zipCode" type="zip"/>
</sequence>
</complexType>

<simpleType name="zip">
<restriction base="string">
<pattern value="[0-9]5-[0-9]4"/>
</restriction>
</simpleType>
<xsd:schema>
</types>
```

The `types` section imports or embeds XSD directly. As such, everything inside of the `types` section is of type `xsd:schema`. In this example, we define a `complexType` called an address, which will be used in the definition of other `complexType`s or in `message`s (see Section A.2.3), which are the inputs and outputs to the service interface..

This address is a sequence of several items. Street addresses may consist of one or two lines. An optional second line might specify an office number, apartment number, *etc.* We constrain the

`street` element by allowing it to consist of at lesat a single element (`minOccurs="1"`), or of up to two elements (`maxOccurs="2"`). The `street` is implemented as an array of up to two elements in size.

The `city` and `state` entries are straightforward `string` data elements. Notice, however, that the `zipCode` is not of type `string`, but rather of type `zip`. This is a reference to the `simpleType` `zip` defined in Section A.1.2. This `simpleType` would also be included in the WSDL `types` section, and is included immediately below the `complexType` definition shown here.

`complexType` definitions are typically a series of one or more user-defined `simpleType` definitions, as well as internal data types defined by the schema (including strings, integers, *etc.*).

A.2.3 `message` Section

Messages are the fundamental unit of information exchange for services. Messages can be designed for input or output (this distinction is not made until the `portType` section described in Section A.2.4) and are comprised of a number of message parts.

Although it is generally a good design strategy to restrict all type definitions to the `types` section, it is possible to create entire data types via Messages and Message parts. Most of the flexibility that was present in the design of the XML Schema is present for the design of the WSDL Message. Good practice, however, is to define complex data types in the `types` section, and then to define `message` elements consisting of just one part that is typed according to the `types` section.

Consider the following example, which builds upon the `types` example shown in Section A.2.2:

```
<message name="Letter">
<part name="message" type="xsd:string"/>
<part name="toAddress" type="address"/>
<part name="returnAddress" type="address"/>
</message>
```

In this example, a `Letter` message is defined to contain three elements: the `message` body (a `string`), the `toAddress` and the `fromAddress` (both of type `address`, which was defined in the `types` section).

Each `message` is automatically implemented as an abstract data type containing fields corresponding to each `part`. Some of these fields may be other data types defined by the XSD in the `types` section.

A.2.4 `portType` Section

`portType`s define any number of `operation`s that the service will support. Because services are message driven, these operations will take as inputs and provide as outputs `message`s that were defined in the `message`s section. It becomes clear, then, that the WSDL file results in a contract that both the caller and the service can agree upon in terms of usage and data types. There are four types of interactions that exist between a client and a server. They are protocols by which the service is invoked, and these protocols are defined by the message passing behavior of the client and the server, as follows [68]:

- **One-way**: A message is sent to the service from the client, and the service produces no output in response.
- **Request-response**: A message is sent to the service from the client, and the service produces an output message in response.
- **Solicit-response**: The service sends a message to the client and receives a response or input from the client.
- **Notification**: The service sends a message without receiving any input parameters or response from the client.

The following is a sample `portType` section (unrelated to the other examples throughout this chapter) that illustrates these four protocols [68].

```
<portType name="exampleProtocols">
<operation name="oneWay">
<input message="x:m1"/>
</operation>
<operation name="requestResponse">
<input message="x:m1"/>
<output message="y:m2"/>
</operation>
<operation name="solicitResponse">
<output message="x:m1"/>
<input message="y:m2"/>
</operation>
<operation name="notification">
<output message="x:m1"/>
</operation>
</portType>
```

In this example, the `message`s are referenced from the `message`s defined in the `message` section of the WSDL document.

Within each operation, it is possible to specify a `<fault>`, that is "handled" by sending a message. The syntax of this declaration is similar to those in the `<input>` and `<output>` examples above in that it references a `message` type that is sent in lieu of the `output` message if a fault occurs during the invocation of that `operation`.

As a concrete example, consider an operation for mailing a letter using the message type we defined in Section A.2.3. In this case, only one `message` was defined: the `letter`. Therefore, no reply `message` is specified and this is a one-way operation.

```
<portType name="sendMailPortType">
<operation name="sendLetter">
<input message="letter"/>
</operation>
</portType>
```

A.2.5 `binding` Section

`Binding`s specify the format, protocol and encoding of the messages that are sent. A popular choice for encoding and protocol is to use SOAP over HTTP. There are a number of available protocols to choose from, each with its own advocacy group [24]. As is the general pattern in WSDL

documents, a binding is applied to each portType, and a protocol is assigned to each message of each operation previously defined.

Simple Object Access Protocol (SOAP) The Simple Object Access Protocol (SOAP) [17] allows for an XML-based lightweight transport of messages and binary data. SOAP is often (but not necessarily) sent over the TCP transport layer via HTTP(S). One could send SOAP data over other protocols such as SMTP or FTP. Service messages defined by the WSDL are wrapped in a **SOAP body**, combined with header information known as a **SOAP header**, and finally sent as a single unit called the **SOAP envelope**. At this point, the messages are processed via standard XML parsers on both the client and the server side.

Apache Axis [2, 5] is a newer implementation of the SOAP transport and XML processor. Axis will be discussed in more detail in Section 2.2.4.

There are two options for sending SOAP messages [68]:

- **Document style**: Document style basically means that all the parts of the `<message>` are inserted into the SOAP envelope as children of the `<Body>` element. That is, the data elements in a `message` correspond to an XSD in the `types` section. This style is currently preferred by the .NET platform.

- **RPC style**: RPC style basically means that all the parts of the `<message>` are wrapped in some outer element representing the RPC. Then that resulting single wrapper element is inserted as the single child of SOAP's `<Body>` element. In other words, the schema and type information is embedded into the `message` section. This style is currently preferred by the Java platform.

Moreover, there are two types of uses of each style. These are called **encoded** and **literal** [23]. The `encoded` type specifies the corresponding XSD data type associated with each data element within the `message`, whereas the `literal` style does not. The `document/literal` style is the most common style and usage when using SOAP over HTTP.

An example of a SOAP `<binding>` is provided here, and builds upon the `portType` defined in Section A.2.4:

```
<binding name="sendMailBinding" type="sendMailPortType">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="sendLetter">
<input>
<soap:body use="literal"/>
</input>
</operation>
</binding>
```

This specifies that the `sendMail portType` (and its associated `message`) is implemented via the `document/literal` encoding.

A.2.6 `service` Section

Finally, the service itself can be defined in the `<service>` section. The service takes each binding and assigns to it a physical address. For example, if SOAP over SMTP were used as a particular binding, the service would point to an email address. In the typical case (in this paper, that's SOAP over HTTP), the service location will be a Uniform Resource Identifier (URI) such as a web URL or email address.

Consider an example of a `<service>` section, which maps the `portType` to the web location where the service can be invoked (by sending messages to the service that correspond to the `message` structure defined in the WSDL).

```
<service name="SendLetterService">
<port binding="sendMailBinding"
name="sendMailPort">
<soap:address location="https://localhost:8080/SendLetter"/>
</port>
</service>
```

The service specifies a location for the service binding, which aggregates one or more port types (operations). An operation is specified by the input and output messages, if any, that it receives and sends. Because messages are aggregations of one or more types defined in XSD, each part of the WSDL definition builds upon the one before. This relationship is shown in Figure A.3.

A.3 Quality of Service and Security Constraints

In addition to its simplified design, SOA provides a number of Quality of Service (QoS) requirements that are also defined by additional XML documents. Because messages sent between services are standardized in XML, normal XML technologies apply including fast parsing, security, permissions, encryption and compression. These are known as the WS-* standards: they are XML addons to a WSDL document to provide additional constraints, protocols, search information, *etc.* Particular details of each standard are outside the scope of this document, but they are discussed at length in the literature [68, 74, 39].

A.4 Service Discovery: UDDI

Universal Description, Discovery, and Integration (UDDI)[2] is the first step in integrating and using existing services. UDDI is intended to describe an entity's business-level services, and to discover and integrate with other services that are provided by external entities [66].

UDDI registries store and provide a number of pieces of information about services. These include a **white pages** that feature contact information for the services and the companies that house them, a **yellow pages** that contain more semantic information including provided services and auxiliary query information, and a **green pages** that contain location information and other logistics for actually invoking a web service. As has been the case throughout this paper, UDDI definitions are also provided via structured XML documents.

A number of public UDDI registries exist, though at present most UDDI registries are held within private intranets. Similarly, there are a number of UDDI implementations (and alternatives) for various platforms, including Business to Business XML (ebXML), Jini, and the Java API for XML Registries (JAXR) [55].

Like WSDL **portType**s, a UDDI service description describes the interfaces offered by a service. In UDDI, these descriptions are called **tModel**s. Searches (inquiries) are performed on the UDDI

[2]For more information, see [21].

registry to find services that have been published that match the given search criteria.

A.4.1 Using UDDI

Several APIs exist to query UDDI service registries, including UDDI4J [22]. Using the metadata obtained from the UDDI registry jUDDI+ can perform automated matchmaking services based on UDDI metadata [32].

The UDDI publish and inquiry operations are summarized here [22, 11]. In both operations, a UDDI4J UDDI Proxy is created that contains the URLs to perform search and inquiry operations.

Publish To publish a service to a UDDI registry, the publisher must first authenticate with the UDDI registry. This is done with an authentication token. Then, a `BusinessEntity` is created and registered on the server that represents the publishing company or entity. The registry returns a unique key to the publisher, and the publisher can use that key, a `tModel` that describes the publishing scheme, and a list of categories to further categorize the publisher within the registry. This process is essentially repeated for a `tModel`, which describes the service to be published. This results in a `tModel` key returned from the server, which is used to actually publish an actual service and bind it to the `tModel` interface description.

Inquiry UDDI4J can also search a UDDI registry [11]. The proxy is called with the desired search terms, and the corresponding `tModel` and access information is returned.

A.5 Business Process Execution Language: BPEL

Because services are a primitive unit in SOA, they can be composed just as objects are in the OO paradigm to create new applications according to business processes. This is called **service orchestration**. Unlike objects, services are also registered with a name service that can be searched. In this way, services can be dynamically found, bound, and consumed at runtime. An interesting and open problem (discussed in Section A.6) is to dynamically bind to services based on a query of

that service's auxiliary semantic information, which may have been specified by an external standard like DAML-S (described in Section A.6.1). In other words, based on a search criteria, an application will automatically find, gather and use services that implement some higher level business process.

BPEL4WS is a language that describes composability of web services, much as a WSDL document describes service details of a web service [45, 60, 67]. It includes primitives to `invoke`, `reply` to, and `receive` invocations, and to coordinate between services during execution. These operations are further composed into more complex workflows, creating a system of coordinated services.

Using BPEL, services are connected via their port types, thus exposing their operations to one another according to the contract specified by their corresponding WSDL. BPEL describes web service composition in either a concrete or an abstract manner. Abstract workflow descriptions need not explicitly specify which services are to be bound. Moreover, there are three methods for binding to services through BPEL [45]:

- **Design Time**: Partner services are paired at design time, such that the services in question are hard coded into the BPEL description.
- **Deployment Time**: Partner services are paired when the process is deployed, enabling discovery of suitable services after the BPEL process is designed.
- **Run Time**: Partner services are not paired until they are needed. This is the most generic method of binding partner services, and typically uses a registry service such as UDDI to locate suitable services.

The distinctions between and the implications of these binding methods are discussed in Section A.6.

A.5.1 BPEL Development Environments

ActiveBPEL [1] provides a visual environment for designing a BPEL-based service composition.

> BPEL is geared towards "programming in the large", which supports the logic of business processes. These business processes are self-contained applications that use Web services as activities that implement business functions. BPEL does not try to be a general-purpose programming language. Instead, it is assumed that BPEL will be combined with other languages, which are used to implement business functions ("programming in the small") [28].

To accomplish this, BPELJ [28] is a system that combines BPEL with Java constructs for service integration.

On the other hand, not everyone believes that the WS-* standards adequately support reliable service composition. For example, it has been argued [30] that standards like WS-Reliability are insufficient for reliable service composition, and alternatives like AO4BPEL [30], a middleware container, might better manage BPEL processes. The goal of this aspect-based container is to "address non-functional concerns in BPEL processes" including persistence, message and transaction reliability, and security [30]. These features are provided as aspects, and are framework-level functionality available to the BPEL process.

Another such composition manager is WSCE: A Flexible Web Service Composition Environment [73]. They motivate the problem by indicating the need for a web services simulated environment, because services in production mode would change the state of the world as they execute. This alone, however, could be addressed by separating test and production servers. However, WSCE also provides a visual environment for simulating the deployment and execution of business processes.

SWORD: A Developer Toolkit for Web Service Composition [61] is intended for "semi-automatic" service composition. To do this, SWORD takes an entity-relationship graph of the data inputs and outputs of each action, and then maps the actions to one another. SWORD takes "what it knows" from user input, and what it "wants to know" as its desired result state, and uses the given actions that will take the inputs and intermediates to attain that result. It is, then, a rule-based service composition environment. Fully automated service composition (in which services are found "on the

fly") is described in Section A.6.

A.6 Using Semantics for Automated Service Discovery and Composition

The composition of web services requires compatibility between those web services, from a syntactic and from a semantic point of view. Syntactic compatibility is enforced by WSDL: if two services cannot agree on their message formats and data exchange, they will not be inter-operable. Ideally, if services are sufficiently loosely coupled to one another, they become interchangeable during the execution of a business process. If a service goes down, introduces a bug, or does not meet a desired non-functional requirement such as speed or security, another service can be invoked in its place. Moreover, entire applications can be defined simply by defining the business-level functionality, and services to achieve that goal could be automatically discovered and incorporated into the composed business process.

All of this involves applying semantics to many of the concepts discussed in this document, including automatic service discovery, and automatic service composition. There are four types of incompatibilities when integrating services: *structural*, *value*, *encoding*, and *semantic* [62]. In one approach [62], static and dynamic analysis techniques are used to identify compatibility between services, semi-automatically generate a middleware adapter to address incompatibilities, and introduce a framework in which to design compatibility-friendly services. Semantic interoperability among services remains an open problem. Open research in this area includes the Semantic Web and Semantic Markup Languages, which are described in detail in Section A.6.1.

It is possible, for example, to create a service that can be executed based on certain WSDL contractual specifications, yet does not produce the expected results. This is a *value* incompatibility. As another example, a service that is semantically compatible with another service in terms of its data exchange, but is incompatible in terms of the structural makeup of the messages, has a *structural* incompatibility. An *encoding* incompatibility exists when services have compatible structure and values, but differing XML schema [62], and a *semantic* incompatibility is essentially

a service mismatch in terms of actual functionality provided.

Another approach uses semantic meaning to facilitate semi-automatic composition of web services [63]. This approach uses some user interaction to pick the services to be invoked, but semantics are used to filter this list to a manageable size. DAML+OIL (described in Section A.6.1) and the Web Ontology Language (OWL) are used to provide this filtering mechanism.

A.6.1 DARPA Agent Markup Language (DAML-S): Semantic Markup for Web Services

The DARPA Agent Markup Language (DAML-S) [6] provides a language, taxonomy and categorization (called an **ontology**) for adding semantics to web service descriptions to facilitate automated discovery and, ultimately, automated composition. DAML-S provides an ontology for describing processes at the business logic level, it provides for ontology extensions for describing QoS attributes, locality, and so on. It is used to describe what a service does, how to access it, and how it works. For example, a process might include a sequence of conditionals that repeat while another condition is met. This is captured in the DAML-S Process Ontology.

There are three kinds of process types described by the Process Ontology [72]. They are called **atomic** processes, **composite** processes, and **simple** processes. Atomic processes consist of a single, uninterruptable operation. Simple processes are similar but do not have the atomic property. Composite processes are compositions of other processes.

One possible open problem is to automate the creation of DAML-S representations from available WSDL and UDDI metadata. Current service based systems tend not to provide rich metadata for this type of searching and composition. As a result, most matchmaking services have to simulate a service environment that provides this metadata.

Figure A.2: Syntactic Structure of the WSDL 1.1 Language [68]

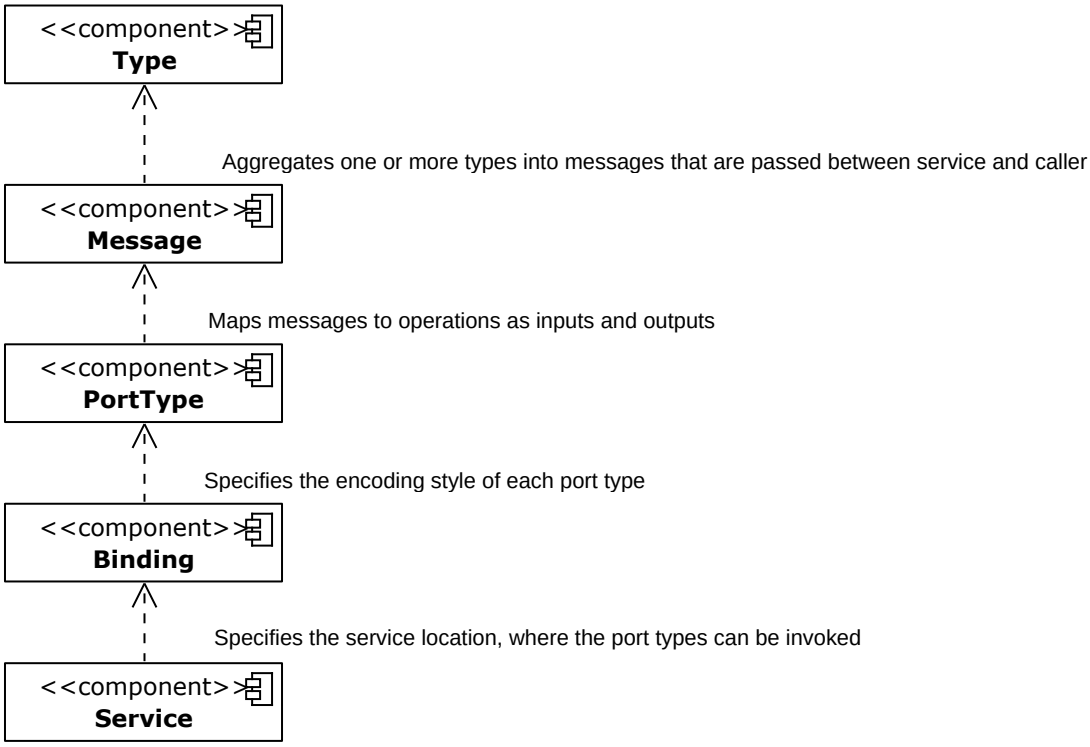

Figure A.3: Relationship among WSDL part definitions

Appendix B. Installation and Deployment

In this appendix, we describe the installation and deployment instructions for REportal. In most cases, a quick configuration is required to point REportal to the appropriate builds of JDK instances and the mysql database (described in Sections B.2.1 and B.2.2). The prepackaged REportal distribution includes an *apps_dist* directory which contains the binary builds for these applications. However, users who wish to build REportal from source, or run REportal on a platform other than Linux or MacOS must manually install JDK instances and mysql before configuring them for use. This process can be automated to a large extent; nevertheless, this appendix describes the entire procedure as a manual process to familiarize the user with the technical details of the REportal environment.

Throughout this appendix, we refer to *$REPORTAL_ROOT*, which is assumed to be the root directory in which REportal is installed.

B.1 Building from Source Code

It is possible to build REportal from source. To build REportal from source code, a few steps are possible. First, one may need to edit the *project.properties* and set the JDK in use to be one appropriate for building the service. In most cases, the default JDK will do, and thus no change is required. However, for instance, in the case of the BAT Analyzer Serivce (located in *$REPORTAL_ROOT/services/BATAnalyzerService*, one must set the JDK to specifically use JDK1.5. This is because the BAT program depends on JDK1.5.

To make this change, one must edit the file:

$REPORTAL_ROOT/services/BATAnalyzerService/nbproject/project.properties

and add a line such as the following:

```
platforms.soylatte.home=$REPORTAL_ROOT/apps_dist/soylatte16-i386-1.0.2
```

This creates a platform called *soylatte* (a MacOS JDK1.6 distribution), and sets its path. Also in the file is an entry called *platform.active*, which must be set to the desired platform. In this case, we set:

```
platform.active=soylatte
```

to use the platform we just added. In the *apps_dist* directory under REportal, we included JDK1.5 and JDK1.6 for Linux, as well as the soylatte JDK1.6 for MacOS. If another platform is used, it may be necessary to install an additional JDK (as is required per Section B.2.1) and have each *project.properties* file point to that JDK installation using this procedure.

To build services in REportal, one must set the classpath for each service. Classpaths should point to any library dependencies of the service. This is done by default; however, if one is authoring a new service based on an existing service's project structure, a few steps are necessary to do this. First, one should add a `file.reference` line to the *project.properties* file found in the `nbproject` directory. This line is of the form:

```
file.reference.xstream-1.1.3.jar=../../lib/xstream-1.1.3.jar
```

In this example, the $file.reference.xstream-1.1.3.jar variable points to the *xstream-1.1.3.jar* file in the REportal *lib* directory. The *../..* exist because the working directory is *$REPORTAL_ROOT/services/NameOfService* directory, and we are referring to a library file in *$REPORTAL_ROOT/lib* directory.

Next, the *nbproject/build-impl.xml* file, which contains the `Class-Path` variable, must be edited to add the jar to the classpath. Find the `Class-Path` file and add the line referring to the variable that was just created in the *project.properties* file.

Finally, in the file copy section of the *build-impl.xml* file, one must copy the library file into the finished web application (*war*) file under the *WEB-INF/lib* directory, by adding the line:

```
<copy file="$file.reference.xstream-1.1.3.jar"
  todir="$build.web.dir.real/WEB-INF/lib"/>
```

again, where `$file.reference.xstream-1.1.3.jar` refers to the variable created in *project.properties*.

Once this is done, one may build REportal by executing *./buildAll.sh*[1] from within the *$REPORTAL_ROOT* directory. This causes the REportal services (found in the *services* directory) to build, followed by a service client stub for each of these services, followed by the REportal presentation layer that requires the client stubs.

Finally, if the build is successful, the web application archives (*war* files) for each service and for the presentation layer are automatically copied to the Tomcat *webapps* deployment directory, where they are automatically deployed by Tomcat.

B.2 Installation

Because REportal is based on services, it is necessary to deploy a few required applications: namely, a JDK, an application server, and an instance of mysql (all built for the target platform), create an empty REportal database with the username and password, and deploy the *war* web service application archive files into the application server. Like the build process described in Section B.1, these steps can be scripted for automation, but are described as a manual process for illustrative purposes.

B.2.1 JDK

Whether building from source or from binaries, it is necessary to have installed a JDK distribution appropriate for the platform being used. Because some services require JDK1.5 and some require JDK1.6, it is necessary to have both versions installed. As described in Section B.1, one must set the JDK active platforms to the appropriate JDK (this is typically JDK1.5 for the BAT Analyzer Service, and JDK1.6 for all the others).

To complete the installation of the JDKs, it is necessary to populate a few *jar* library files required

[1]These scripts assume a Linux or UNIX distribution; if using Windows or a different platform, executing the commands found within the script should work, provided the prerequisite applications described in this appendix are installed for the desired build.

by REportal. This ensures that these *jar* files will be in the classpath and accessible by REportal or its services. In the case of JDK1.5, these files are copied into the *$JAVA_HOME/jdk/jre/lib/ext* directory, and in the case of JDK1.6, they are copied into the *$JAVA_HOME/jre/lib/ext* directory. These required files are:

- ***ForensicsSimple.jar***: this program invokes the underlying Forensics author identification application for the Forensics service.
- ***MetricsFramework.jar***: this program computes metrics for the Metrics service.
- ***bunch.jar***: this program executes the clustering algorithms used throughout REportal for visualization.
- ***dom4j-1.6.1.jar***: this library interfaces with the REportal database for the Project Manager service.
- ***grappa1_2.jar***: this library enables the visualization of dotty graphs, which are output by Bunch and the clusternav ClusterViewer program.
- ***jode-1.1.2-pre1.jar***: this application is a Java decompiler. Using reverse engineering, its interface was obtained, and is invoked to decompile non-source Java projects uploaded to REportal.
- ***xstream-1.1.3.jar***: this library enables serialization of objects into XML for transmission over the network between services.

These files can all be found in the *$REPORTAL_ROOT/lib* directory for copying.

B.2.2 Application Server

Once the JDK is installed, the *startAll.sh* script must be modified so that Tomcat uses these JDK directories. In the *startAll.sh* file, one notices the following line, which starts one of the two instances of Tomcat.

```
./startTomcat.sh 8084 $JAVA_HOME &
```

The *startTomcat.sh* script takes two parameters: the Tomcat working directory location, and the JDK to use. The Tomcat working directory location is located in the *$REPORTAL_ROOT/apps_dist/apache-tomcat-5.5.25/bases* directory, and contains the configuration files (in the *conf* directory), *war* file deployments (in the *webapps* directory), *etc.* For convenience, the name of the directory is the same as the port number that this instance of Tomcat is configured to use.

In addition to starting Tomcat, the *startTomcat.sh* script (which is called automatically by the *startAll.sh* script, and thus not invoked by the user) also specifies the Java **endorsed libraries** directory to be used by Tomcat. JDK1.6 introduces a bug into Tomcat because it ships with an early version of the *Soap with Attachments API for Java (SAAJ).* This library takes precedence over the more current versions of SAAJ, causing Tomcat to fail. By setting the **endorsed** directory before starting Tomcat, one can specify a location where the most recent version of the SAAJ libraries reside. The **endorsed** directory takes precedence over libraries included with the JDK, which works around this error. Here, the **endorsed** directories are *$JAVA_HOME/jre/lib/endorsed* for JDK1.5 and for JDK1.6, although the SAAJ libraries are only needed in the JDK1.6 **endorsed** directory. The files copied to this directory are:

- *jaxb-api.jar*
- *jaxb-impl.jar*
- *jaxws-api.jar*
- *jsr173_api.jar*
- *saaj-api.jar*

Configuration for CGI Scripts and SSL

Tomcat is configured to use SSL on its primary port (in this case, ports 8080 and 8084 are used by the two instances of Tomcat). The Tomcat running on port 8084 arbitrarily hosts the REportal presentation layer; for convenience, a non-SSL port 8443 is configured by this Tomcat instance as well. REportal comes with self-signed certificates for encryption; however, one may desire to create or install a different certificate for use.

To do this, one must install the certificate into the two JDK instances (one JDK for each instance of Tomcat). To install a certificate (self-signed[2] or otherwise), one follows the steps given in Table B.1.

[2]To generate a self-signed certificate, see http://java.sun.com/j2ee/1.4/docs/tutorial-update6/doc/Security6.html.

Table B.1: Commands to install an SSL certificate

```
cd $REPORTAL_ROOT/apps_dist

# These commands import a certificate
# and into the JDK 1.5 and 1.6 keystores.
keytool -import -v -trustcacerts -alias tomcat -file
apache-tomcat-5.5.25/server/server.cer -keystore
jdk1.6.0_03/jre/lib/security/cacerts

keytool -import -v -trustcacerts -alias tomcat -file
apache-tomcat-5.5.25/server/server.cer -keystore
jdk1.5/jdk/jre/lib/security/cacerts


# These commands create a Tomcat keystore
# and import the server certificate into
# the keystore.
# The keystore is the same for both Tomcat
# instances, so it is copied
# into the other Tomcat keystore location
keytool -genkey -alias tomcat -keyalg RSA -keystore
apache-tomcat-5.5.25/bases/8084/keystore

keytool -export -alias tomcat -storepass changeit -file
apache-tomcat-5.5.25/server/server.cer -keystore
apache-tomcat-5.5.25/bases/8084/keystore

cp apache-tomcat-5.5.25/bases/8084/keystore apache-tomcat-5.5.25/bases/8080

# These commands import the certificate into the JDK keystores
keytool -import -v -trustcacerts -alias tomcat -file
apache-tomcat-5.5.25/server/server.cer -keystore
jdk1.5/jdk/jre/lib/security/cacerts

keytool -import -v -trustcacerts -alias tomcat -file
apache-tomcat-5.5.25/server/server.cer -keystore
jdk1.6.0_03/jre/lib/security/cacerts
```

Next, it is necessary to configure Tomcat to use CGI scripts on the instance that is running the REportal JSP front-end. This is because a CGI script, *webdot.pl* is used by the clusternav cluster graph viewing application to render *dot* graphs. CGI is disabled by default for security; to enable it, one must rename the *servlets-cgi.renametojar* file to *servlets-cgi.jar* in the *$TOMCAT_HOME/server/lib* directory. Next, in the *$TOMCAT_HOME/bases/8084/conf/web.xml* file, uncomment the *cgi* section, and the *cgi* servlet mapping definition. This enables the *web/cgi-bin* directory for cgi scripts.

B.2.3 Database Setup

The next step is to install and configure the database. The database server is mysql. One should install mysql for the platform being used.

The first step is to set the root password on the mysql instance. We will use the password *repadmin*, which is the one used by REportal by default. We discuss how to use a different password in Section B.2.4. Table B.2 shows how to set the root password on the database.

Table B.2: Commands to configure the mysql database

```
cd $MYSQL_ROOT
./bin/mysqladmin -u root -h localhost password repadmin
```

The database is started along with Tomcat upon executing the *startAll.sh* script. One must modify the following line in *startAll.sh*:

```
cd $MYSQL_ROOT ./bin/mysqld_safe --port=8306 & cd ..
```

so that `$MYSQL_ROOT` points to the directory in which mysql is installed.

Finally, one must install an empty REportal database into the mysql server. To do this, one must start the mysql client and run the following commands:

```
CREATE DATABASE reportal2;
USE reportal2;

 CREATE TABLE KeyTable (
        KeyString          BLOB NOT NULL
 );

 CREATE TABLE Logins (
        Name               CHAR(50) NOT NULL,
        Password           CHAR(48) NOT NULL,
        LoginID            INTEGER NOT NULL AUTO_INCREMENT,
        FirstName          CHAR(50),
        LastName           CHAR(50),
        Company            CHAR(100),
        EMail              CHAR(75)
        UNIQUE(LoginID)
 );

CREATE UNIQUE INDEX LoginIDIndex ON Logins
(
       LoginID            ASC
);

 CREATE TABLE Projects (
        ProjectID          INTEGER AUTO_INCREMENT,
        UserID             INTEGER NOT NULL,
        ProjectTypeID      INTEGER NOT NULL,
        ProjectName        CHAR(255) NOT NULL,
        ProjectDescription CHAR(255) NULL,
        ProjectDataLocation CHAR(255) NOT NULL,
        UNIQUE(ProjectID)
 );

CREATE UNIQUE INDEX ProjectIDIndex ON Projects
(
       ProjectID           ASC
);

 CREATE TABLE ProjectTypes (
        ProjectTypeID      INTEGER AUTO_INCREMENT,
        ProjectType        CHAR(20) NULL,
        UNIQUE(ProjectTypeID)
 );

CREATE UNIQUE INDEX ProjectTypesIDIndex ON ProjectTypes
(
       ProjectTypeID           ASC
);

INSERT INTO ProjectTypes (ProjectType) VALUES ('Java');
INSERT INTO ProjectTypes (ProjectType) VALUES ('C');
```

```
INSERT INTO ProjectTypes (ProjectType) VALUES ('C++');
```

To start the mysql client, run:

```
mysql --user=root -h localhost --port=8306 --protocol=TCP -p &
```

and provide the password that was chosen earlier in this section.

B.2.4 REportal Configuration

After the application server and database are set up, it is necessary to configure REportal so that it can locate the services that the presentation layer will execute. Moreover, it is necessary to set the location of the mysql database and the login information so that the REportal services may access it.

All of this is accomplished via the *$REPORTAL_ROOT/reportal.ini* file. A sample *reportal.ini* file is shown in Table B.3.

There are two sections in Table B.3: *reportal* and *wsdl_location*.

Table B.3: Sample *reportal.ini* file

```
[reportal]
db_connect=jdbc:mysql://localhost:8306/reportal2
db_username=root
db_pw=repadmin
db_jdbc=com.mysql.jdbc.Driver
projroot=.

[wsdl_location]
AspectInstrumentation=
https://localhost:8084/AspectInstrumentation/AspectInstrumentationService?wsdl

BATAnalyzerService=
https://localhost:8080/BATAnalyzerService/BATAnalyzerServiceService?wsdl

BunchWrapper=
https://localhost:8084/BunchWrapper/bunchwrapperService?wsdl

MetricsService=
https://localhost:8084/MetricsService/MetricServiceService?wsdl

ProjectManagerService=
https://localhost:8084/ProjectManagerService/ProjectManagerService?wsdl

SourceBrowserService=
https://localhost:8084/SourceBrowserService/SourceBrowserServiceService?wsdl

TextSearchService=
https://localhost:8084/TextSearchService/TextSearchServiceService?wsdl

ForensicsService=
https://localhost:8084/ForensicsService/ForensicsServiceService?wsdl
```

The *reportal* section defines REportal commands, like the default location of user projects (the Tomcat working directory, by default), and the database parameters. This includes the URL of the database, the username and password that were chosen to configure the database in Section B.2.3, and the class library that interfaces with the database (JDBC for mysql, in this case).

The *wsdl_location* section defines the URL for all service WSDLs used by REportal. Normally, one compiles WSDL client stubs with hardcoded URL values that are called by an application's presentation layer. To decouple the services from the presentation layer, we instead allow the presentation layer to lookup the WSDL dynamically and instantiate a client object from the WSDL at runtime. During source code compilation, default hardcoded client stubs are built and can also be used. However, this approach enables one to move or replace a service, and update its location in the *reportal.ini* file. One notices that the URL for all the services except for the BAT Analyzer service are running on the Tomcat port 8084 instance. BAT, which depends on JDK 1.5, runs on the Tomcat port 8080 instance, which is running on top of JDK 1.5. In this case, because the Tomcat instances are running on the same computer as the presentation layer (in fact, the presentation layer web application is running on the Tomcat port 8084 instance, though this is arbitrary), the service URLs all refer to *localhost*.

B.2.5 Starting and Stopping REportal

Once REportal is installed and its supporting applications are configured, one can start the REportal server and its services by executing:

```
$REPORTAL_ROOT/apps_dist/startAll.sh
```

and stop the server by executing:

```
$REPORTAL_ROOT/apps_dist/stopAll.sh
```

B.3 Usage Notes

To use REportal, visit `https://localhost:8084/REportal2` from the host on which REportal was installed (or replace *localhost* with the name of the host on which REportal was installed). One might find it necessary to set the memory limits on the JRE on which the REportal web browser is running, so that the ClusterViewer will have enough memory to render its images. To do this, one should set the following JRE flags in the Java control panel or web browser settings, as shown in Figure B.1:

```
-Xmx256m -Xms256m
```

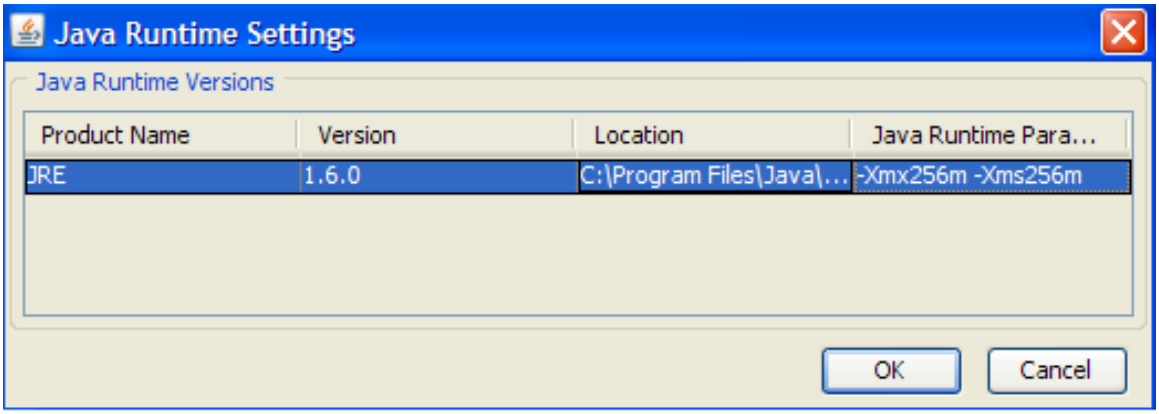

Figure B.1: Sample Java control panel settings

Finally, it is necessary to edit the file *$JAVA_HOME/jre/lib/security/java.policy*,[3] and enable a security permission. This permission is required by the ClusterViewer in order to visualize output. If either this or the memory setting described in this section is incorrect, REportal may not render images in the ClusterViewer. The rest of the portal will work normally, and the tree hierarchy in clusternav will correctly list the entities and clusters in the graph, but no graphical display will be present. The line that must be added to enable the security permission is as follows:

```
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
```

A sample *java.policy* file, with this required permission included at the bottom, is shown in Table B.4.

[3] On MacOS, if using Firefox, the *java.policy* file should be placed in the *user.home* directory.

Table B.4: Sample *java.policy* file used by the JDK

```
// Standard extensions get all permissions by default
grant codeBase "file:${{java.ext.dirs}}/*" {
permission java.security.AllPermission;
};

// default permissions granted to all domains
grant {
// Allows any thread to stop itself using the java.lang.Thread.stop()
// method that takes no argument.
// Note that this permission is granted by default only to remain
// backwards compatible.
// It is strongly recommended that you either remove this permission
// from this policy file or further restrict it to code sources
// that you specify, because Thread.stop() is potentially unsafe.
// See "http://java.sun.com/notes" for more information.
permission java.lang.RuntimePermission "stopThread";
// allows anyone to listen on un-privileged ports
permission java.net.SocketPermission "localhost:1024-", "listen";
// "standard" properies that can be read by anyone
permission java.util.PropertyPermission "java.version", "read";
permission java.util.PropertyPermission "java.vendor", "read";
permission java.util.PropertyPermission "java.vendor.url", "read";
permission java.util.PropertyPermission "java.class.version", "read";
permission java.util.PropertyPermission "os.name", "read";
permission java.util.PropertyPermission "os.version", "read";
permission java.util.PropertyPermission "os.arch", "read";
permission java.util.PropertyPermission "file.separator", "read";
permission java.util.PropertyPermission "path.separator", "read";
permission java.util.PropertyPermission "line.separator", "read";
permission java.util.PropertyPermission "java.specification.version", "read";
permission java.util.PropertyPermission "java.specification.vendor", "read";
permission java.util.PropertyPermission "java.specification.name", "read";
permission java.util.PropertyPermission
             "java.vm.specification.version", "read";
permission java.util.PropertyPermission
             "java.vm.specification.vendor", "read";
permission java.util.PropertyPermission "java.vm.specification.name", "read";
permission java.util.PropertyPermission "java.vm.version", "read";
permission java.util.PropertyPermission "java.vm.vendor", "read";
permission java.util.PropertyPermission "java.vm.name", "read";
permission java.lang.reflect.ReflectPermission
             "suppressAccessChecks";
};
```

Appendix C. Service WSDL and XML Schema Definitions

C.1 Project Manager Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://projectmanager.service"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ProjectManager"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/"
    xmlns:tns="http://projectmanager.service"
    xmlns:ns="http://datatypes.reportal">

    <types>
        <xsd:schema targetNamespace="http://datatypes.reportal"
                    xmlns:tns1="http://datatypes.reportal">
            <xsd:complexType name="ProjectInfo">
                <xsd:sequence>
                    <xsd:element name="name">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:string"/>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="userId">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:int"/>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="language">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="Java">
                                    <xsd:annotation>
                                        <xsd:documentation>Java</xsd:documentation>
                                    </xsd:annotation>
                                </xsd:enumeration>
                                <xsd:enumeration value="C">
                                    <xsd:annotation>
                                        <xsd:documentation>C</xsd:documentation>
                                    </xsd:annotation>
                                </xsd:enumeration>
                                <xsd:enumeration value="C++">
                                    <xsd:annotation>
                                        <xsd:documentation>C++</xsd:documentation>
                                    </xsd:annotation>
                                </xsd:enumeration>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="filePath">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:string"/>
                        </xsd:simpleType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="UserToken">
                <xsd:sequence>
```

```
                <xsd:element name="userId">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="password">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="firstName">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="lastName">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="company">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="email">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="loginId" type="xsd:int" minOccurs="0"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="ProjectList">
            <xsd:sequence>
                <xsd:element name="projects" type="tns1:ProjectInfo"
                             minOccurs="0" maxOccurs="unbounded">
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="ProjectData">
            <xsd:sequence>
                <xsd:element name="project" type="tns1:ProjectInfo"></xsd:element>
                <xsd:element name="data" type="xsd:base64Binary"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:schema>
</types>

<message name="AddProjectRequest">
    <part name="addProject" type="ns:ProjectInfo"/>
</message>
<message name="AddProjectResponse">
    <part name="addProjectResponse" type="ns:ProjectInfo"/>
</message>
<message name="ListProjectsRequest">
    <part name="listProjectsUserToken" type="ns:UserToken"/>
</message>
<message name="ListProjectsResponse">
    <part name="projectList" type="ns:ProjectList"/>
</message>
<message name="UserLoginRequest">
    <part name="userResponseToken" type="ns:UserToken"/>
</message>
```

```
<message name="UserLoginResponse">
    <part name="userLoginToken" type="ns:UserToken"/>
</message>
<message name="UserRegisterRequest">
    <part name="userRegisterToken" type="ns:UserToken"/>
</message>
<message name="UserRegisterResponse">
    <part name="userRegisterResponseToken" type="ns:UserToken"/>
</message>
<message name="RemoveProjectRequest">
    <part name="removeProject" type="ns:ProjectInfo"/>
</message>
<message name="RemoveProjectResponse">
    <part name="removeProjectResponse" type="ns:ProjectInfo"/>
</message>
<message name="UploadProjectResponse">
    <part name="uploadProjectResponse" type="ns:ProjectInfo"/>
</message>
<message name="UploadProjectRequest">
    <part name="uploadProject" type="ns:ProjectData"/>
</message>

<portType name="ProjectManagerPortType">
    <operation name="addProject">
        <input name="addProjectRequest" message="tns:AddProjectRequest"/>
        <output name="addProjectResponse" message="tns:AddProjectResponse"/>
    </operation>
    <operation name="listProjects">
        <input name="listProjectsRequest" message="tns:ListProjectsRequest"/>
        <output name="listProjectsResponse" message="tns:ListProjectsResponse"/>
    </operation>
    <operation name="userLogin">
        <input name="userLoginRequest" message="tns:UserLoginRequest"/>
        <output name="userLoginResponse" message="tns:UserLoginResponse"/>
    </operation>
    <operation name="userRegister">
        <input name="userRegisterRequest" message="tns:UserRegisterRequest"/>
        <output name="userRegisterResponse" message="tns:UserRegisterResponse"/>
    </operation>
    <operation name="removeProject">
        <input name="removeProjectRequest" message="tns:RemoveProjectRequest"/>
        <output name="removeProjectResponse" message="tns:RemoveProjectResponse"/>
    </operation>
    <operation name="uploadProject">
        <input name="uploadProjectRequest" message="tns:UploadProjectRequest"/>
        <output name="uploadProjectResponse" message="tns:UploadProjectResponse"/>
    </operation>
</portType>

<binding name="ProjectManagerPortTypeBinding" type="tns:ProjectManagerPortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="addProject">
        <soap:operation/>
        <input name="addProjectRequest">
            <soap:body use="literal" parts="addProject"
                       namespace="http://projectmanager.service"/>
        </input>
        <output name="addProjectResponse">
            <soap:body use="literal" parts="addProjectResponse"
                       namespace="http://projectmanager.service"/>
        </output>
    </operation>
    <operation name="listProjects">
        <soap:operation/>
```

```
        <input name="listProjectsRequest">
            <soap:body use="literal" parts="listProjectsUserToken"
                       namespace="http://projectmanager.service"/>
        </input>
        <output name="listProjectsResponse">
            <soap:body use="literal" parts="projectList"
                       namespace="http://projectmanager.service"/>
        </output>
    </operation>
    <operation name="userLogin">
        <soap:operation/>
        <input name="userLoginRequest">
            <soap:body use="literal" parts="userResponseToken"
                       namespace="http://projectmanager.service"/>
        </input>
        <output name="userLoginResponse">
            <soap:body use="literal" parts="userLoginToken"
                       namespace="http://projectmanager.service"/>
        </output>
    </operation>
    <operation name="userRegister">
        <soap:operation/>
        <input name="userRegisterRequest">
            <soap:body use="literal" parts="userRegisterToken"
                       namespace="http://projectmanager.service"/>
        </input>
        <output name="userRegisterResponse">
            <soap:body use="literal" parts="userRegisterResponseToken"
                       namespace="http://projectmanager.service"/>
        </output>
    </operation>
    <operation name="removeProject">
        <soap:operation/>
        <input name="removeProjectRequest">
            <soap:body use="literal" parts="removeProject"
                       namespace="http://projectmanager.service"/>
        </input>
        <output name="removeProjectResponse">
            <soap:body use="literal" parts="removeProjectResponse"
                       namespace="http://projectmanager.service"/>
        </output>
    </operation>
    <operation name="uploadProject">
        <soap:operation/>
        <input name="uploadProjectRequest">
            <soap:body use="literal" parts="uploadProject"
                       namespace="http://projectmanager.service"/>
        </input>
        <output name="uploadProjectResponse">
            <soap:body use="literal" parts="uploadProjectResponse"
                       namespace="http://projectmanager.service"/>
        </output>
    </operation>
</binding>

<service name="ProjectManagerService">
    <port name="ProjectManagerPortTypeBindingPort"
          binding="tns:ProjectManagerPortTypeBinding">
        <soap:address
          location=
          "http://localhost:8084/ProjectManagerService/ProjectManagerPortTypeBindingPort"
        />
    </port>
</service>
```

```
</definitions>
```

C.2 BAT Static Analyzer Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://batanalyzer/service"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://batanalyzer/service"
    name="BATAnalyzerService"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">

    <types>
        <xsd:schema targetNamespace="http://batanalyzer/service"
                    xmlns:tns1="http://batanalyzer/service">
            <xsd:simpleType name="FileType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="JAR"/>
                    <xsd:enumeration value="CLASS"/>
                    <xsd:enumeration value="ZIP"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:schema>
    </types>

    <message name="BATAnalyzerServiceOperationRequest">
        <part name="filePath" type="xsd:string"/>
        <part name="fileType" type="tns:FileType"/>
    </message>
    <message name="BATAnalyzerServiceOperationReply">
        <part name="part1" type="xsd:string"/>
    </message>

    <portType name="BATAnalyzerServicePortType">
        <operation name="BATAnalyzerServiceOperation">
            <input name="input1" message="tns:BATAnalyzerServiceOperationRequest"/>
            <output name="output1" message="tns:BATAnalyzerServiceOperationReply"/>
        </operation>
    </portType>

    <binding name="BATAnalyzerServiceBinding" type="tns:BATAnalyzerServicePortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="BATAnalyzerServiceOperation">
            <soap:operation/>
            <input name="input1">
                <soap:body use="literal" namespace="http://batanalyzer/service"/>
            </input>
            <output name="output1">
                <soap:body use="literal" namespace="http://batanalyzer/service"/>
            </output>
        </operation>
    </binding>

    <service name="BATAnalyzerServiceService">
        <port name="BATAnalyzerServicePort" binding="tns:BATAnalyzerServiceBinding">
            <soap:address location=
                  "http://localhost:8080/BATAnalyzerService/BATAnalyzerServiceService"/>
        </port>
    </service>

    <plink:partnerLinkType name="BATAnalyzerServicePartner">
        <plink:role name="BATAnalyzerServicePortTypeRole"
                    portType="tns:BATAnalyzerServicePortType"/>
    </plink:partnerLinkType>
```

```
</definitions>
```

C.3 Bunch Clustering Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://serg.cs.drexel.edu/bunch/bunchwrapper"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://serg.cs.drexel.edu/bunch/bunchwrapper"
    name="bunchwrapper"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">

    <types/>

    <message name="bunchwrapperOperationRequest">
        <part name="mdgInput" type="xsd:string"/>
    </message>
    <message name="bunchwrapperOperationReply">
        <part name="gxlOutput" type="xsd:string"/>
    </message>

    <portType name="bunchwrapperPortType">
        <operation name="bunchwrapperOperation">
            <input name="input1" message="tns:bunchwrapperOperationRequest"/>
            <output name="output1" message="tns:bunchwrapperOperationReply"/>
        </operation>
    </portType>

    <binding name="bunchwrapperBinding" type="tns:bunchwrapperPortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="bunchwrapperOperation">
            <soap:operation/>
            <input name="input1">
                <soap:body use="literal"
                           namespace="http://serg.cs.drexel.edu/bunch/bunchwrapper"/>
            </input>
            <output name="output1">
                <soap:body use="literal"
                           namespace="http://serg.cs.drexel.edu/bunch/bunchwrapper"/>
            </output>
        </operation>
    </binding>

    <service name="bunchwrapperService">
        <port name="bunchwrapperPort" binding="tns:bunchwrapperBinding">
            <soap:address location=
                          "http://localhost:8084/bunchwrapperService/bunchwrapperPort"/>
        </port>
    </service>

    <plink:partnerLinkType name="bunchwrapperPartner">
        <plink:role name="bunchwrapperPortTypeRole" portType="tns:bunchwrapperPortType"/>
    </plink:partnerLinkType>
</definitions>
```

C.4 Forensics Service to Determine Code Authorship

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ForensicsServiceRIT62936"
    targetNamespace="http://j2ee.netbeans.org/wsdl/ForensicsService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://j2ee.netbeans.org/wsdl/ForensicsService"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype">

    <types>
        <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/ForensicsService">
            <xsd:complexType name="PredictionTuple">
                <xsd:sequence>
                    <xsd:element name="pathName">
                        <xsd:simpleType>
                            <xsd:restriction xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                             base="xsd:string"/>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="predictedAuthor">
                        <xsd:simpleType>
                            <xsd:restriction xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                             base="xsd:string"/>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="confidence" type="xsd:double"></xsd:element>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="PredictionTupleList">
                <xsd:sequence>
                    <xsd:element name="predictions" type="tns:PredictionTuple"
                                 minOccurs="0" maxOccurs="unbounded"></xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:schema>
    </types>

    <message name="identifyAuthorRequest">
        <part name="learningFiles" type="xsd:base64Binary"/>
        <wsdl:part name="testingFiles" type="xsd:base64Binary"/>
    </message>
    <message name="identifyAuthorReply">
        <part name="predictions" type="tns:PredictionTupleList"/>
    </message>

    <portType name="ForensicsServicePortType">
        <wsdl:operation name="identifyAuthor">
            <wsdl:input name="input1" message="tns:identifyAuthorRequest"/>
            <wsdl:output name="output1" message="tns:identifyAuthorReply"/>
        </wsdl:operation>
    </portType>

    <binding name="ForensicsServiceBinding" type="tns:ForensicsServicePortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="identifyAuthor">
            <soap:operation/>
            <wsdl:input name="input1">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/ForensicsService"/>
            </wsdl:input>
            <wsdl:output name="output1">
```

```
            <soap:body use="literal"
                       namespace="http://j2ee.netbeans.org/wsdl/ForensicsService"/>
        </wsdl:output>
    </wsdl:operation>
</binding>

<service name="ForensicsServiceService">
    <wsdl:port name="ForensicsServicePort" binding="tns:ForensicsServiceBinding">
        <soap:address location=
              "http://localhost:8084/ForensicsServiceService/ForensicsServicePort"/>
    </wsdl:port>
</service>

<plnk:partnerLinkType name="ForensicsServiceRIT629361">
    <plnk:role name="ForensicsServicePortTypeRole"
               portType="tns:ForensicsServicePortType"/>
</plnk:partnerLinkType>
</definitions>
```

C.5 Metrics Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://metricservice"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://metricservice"
    name="MetricService"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">

        <types>
            <xsd:schema targetNamespace="http://metricservice">
                <xsd:simpleType name="FileType">
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="JAR"/>
                        <xsd:enumeration value="CLASS"/>
                        <xsd:enumeration value="ZIP"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:schema>
        </types>

    <message name="MetricServiceOperationRequest">
        <part name="filePath" type="xsd:string"/>
        <part name="fileType" type="tns:FileType"/>
    </message>
    <message name="MetricServiceOperationReply">
        <part name="metricResponse" type="xsd:string"/>
    </message>

    <portType name="MetricServicePortType">
        <operation name="MetricServiceOperation">
            <input name="input1" message="tns:MetricServiceOperationRequest"/>
            <output name="output1" message="tns:MetricServiceOperationReply"/>
        </operation>
    </portType>

    <binding name="MetricServiceBinding" type="tns:MetricServicePortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="MetricServiceOperation">
            <soap:operation/>
            <input name="input1">
                <soap:body use="literal" namespace="http://metricservice"/>
            </input>
            <output name="output1">
                <soap:body use="literal" namespace="http://metricservice"/>
            </output>
        </operation>
    </binding>

    <service name="MetricServiceService">
        <port name="MetricServicePort" binding="tns:MetricServiceBinding">
            <soap:address location=
                        "http://localhost:8084/MetricServiceService/MetricServicePort"/>
        </port>
    </service>

    <plink:partnerLinkType name="MetricServicePartner">
        <plink:role name="MetricServicePortTypeRole" portType="tns:MetricServicePortType"/>
    </plink:partnerLinkType>
</definitions>
```

C.6 Source Code Browser Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://service.sorcerer"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://service.sorcerer"
    name="SourceBrowserService"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">

    <types/>

    <message name="SourceBrowserServiceOperationRequest">
        <part name="path" type="xsd:string"/>
    </message>
    <message name="SourceBrowserServiceOperationReply">
        <part name="zipOutput" type="xsd:base64Binary"/>
    </message>

    <portType name="SourceBrowserServicePortType">
        <operation name="SourceBrowserServiceOperation">
            <input name="input1" message="tns:SourceBrowserServiceOperationRequest"/>
            <output name="output1" message="tns:SourceBrowserServiceOperationReply"/>
        </operation>
    </portType>

    <binding name="SourceBrowserServiceBinding" type="tns:SourceBrowserServicePortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="SourceBrowserServiceOperation">
            <soap:operation/>
            <input name="input1">
                <soap:body use="literal" namespace="http://service.sorcerer"/>
            </input>
            <output name="output1">
                <soap:body use="literal" namespace="http://service.sorcerer"/>
            </output>
        </operation>
    </binding>

    <service name="SourceBrowserServiceService">
        <port name="SourceBrowserServicePort" binding="tns:SourceBrowserServiceBinding">
            <soap:address location=
               "http://localhost:8084/SourceBrowserServiceService/SourceBrowserServicePort"
            />
        </port>
    </service>

    <plink:partnerLinkType name="SourceBrowserServicePartner">
        <plink:role name="SourceBrowserServicePortTypeRole"
                    portType="tns:SourceBrowserServicePortType"/>
    </plink:partnerLinkType>
</definitions>
```

C.7 Text Search Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="services.TextSearch"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="services.TextSearch"
    name="TextSearch"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/"
    xmlns:ns="TSMessages">

    <types>
        <xsd:schema targetNamespace="TSMessages" xmlns:tns1="TSMessages">
            <xsd:complexType name="SimpleResponseMessage">
                <xsd:sequence>
                    <xsd:element name="header" type="tns1:StandardResponseHeader"/>
                    <xsd:element name="info" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="StandardRequestHeader">
                <xsd:sequence>
                    <xsd:element name="username" type="xsd:string"/>
                    <xsd:element name="projectname" type="xsd:string"/>
                    <xsd:element name="filePath" type="xsd:string"/>
                    <xsd:element name="fileType" type="ns:FileType"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="StandardResponseHeader">
                <xsd:sequence>
                    <xsd:element name="status" type="tns1:StatusCodeType"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:simpleType name="FileType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="JAR"/>
                    <xsd:enumeration value="CLASS"/>
                    <xsd:enumeration value="ZIP"/>
                </xsd:restriction>
            </xsd:simpleType>
            <xsd:simpleType name="StatusCodeType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="OK"/>
                    <xsd:enumeration value="FAILURE"/>
                </xsd:restriction>
            </xsd:simpleType>
            <xsd:complexType name="TextServiceRequestType">
                <xsd:sequence>
                    <xsd:element name="header" type="tns1:StandardRequestHeader"/>
                    <xsd:element name="searchstring" type="xsd:string"/>
                    <xsd:element name="caseinsensitive" type="xsd:boolean"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="TextServiceResponseType">
                <xsd:sequence>
                    <xsd:element name="header" type="tns1:StandardResponseHeader"/>
                    <xsd:element name="info" type="xsd:string"/>
                    <xsd:element name="files" type="xsd:string" maxOccurs="unbounded"/>
                    <xsd:element name="lines" type="xsd:int" maxOccurs="unbounded"/>
                    <xsd:element name="contents" type="xsd:string" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:schema>
```

```
    </types>

    <message name="TextSearchRequestMessage">
        <part name="params" type="ns:TextServiceRequestType"/>
    </message>
    <message name="TextSearchResponseMessage">
        <part name="response" type="ns:TextServiceResponseType"/>
    </message>

    <portType name="TextSearchServicePortType">
        <operation name="RunTextSearch">
            <input name="input2" message="tns:TextSearchRequestMessage"/>
            <output name="output2" message="tns:TextSearchResponseMessage"/>
        </operation>
    </portType>

    <binding name="TextSearchServicePortTypeBinding" type="tns:TextSearchServicePortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="RunTextSearch">
            <soap:operation/>
            <input name="input2">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/TextSearchService"/>
            </input>
            <output name="output2">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/TextSearchService"/>
            </output>
        </operation>
    </binding>

    <service name="TextSearchServiceService">
        <port name="TextSearchServicePortTypeBindingPort"
              binding="tns:TextSearchServicePortTypeBinding">
            <soap:address location=
                  "http://localhost:8084/TextSearchService/TextSearchServiceService"/>
        </port>
    </service>

    <plink:partnerLinkType name="TextSearchServicePartner">
        <plink:role name="TextSearchServicePortTypeRole"
                    portType="tns:TextSearchServicePortType"/>
    </plink:partnerLinkType>
</definitions>
```

C.8 Dynamic Analysis via Aspect Instrumentation Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="services.AspectInstrumentation"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="services.AspectInstrumentation"
    name="AspectInstrumentation"
    xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/"
    xmlns:ns="AIMessages">

    <types>
        <xsd:schema targetNamespace="AIMessages" xmlns:tns1="AIMessages">
            <xsd:simpleType name="StatusCodeType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="FAILURE"/>
                    <xsd:enumeration value="OK"/>
                </xsd:restriction>
            </xsd:simpleType>
            <xsd:complexType name="MakeHeader">
                <xsd:sequence>
                    <xsd:element name="AspectName" type="xsd:string"/>
                    <xsd:element name="Joinpoints" type="xsd:string" maxOccurs="unbounded"/>
                    <xsd:element name="Options" type="xsd:string" maxOccurs="unbounded"/>
                    <xsd:element name="UserMade" type="xsd:string" maxOccurs="unbounded"/>
                    <xsd:element name="Directory" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="MakeRequest">
                <xsd:sequence>
                    <xsd:element name="header" type="tns1:MakeHeader"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="MakeResponse">
                <xsd:sequence>
                    <xsd:element name="header" type="tns1:StatusCodeType"/>
                    <xsd:element name="aspect" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="GraphRequest">
                <xsd:sequence>
                    <xsd:element name="xml" type="xsd:string"/>
                    <xsd:element name="projectName" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="GraphResponse">
                <xsd:sequence>
                    <xsd:element name="output" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:schema>
    </types>

    <message name="GetRequestMessage">
        <part name="params" type="ns:MakeRequest"/>
    </message>
    <message name="GetResponseMessage">
        <part name="response" type="ns:MakeResponse"/>
    </message>
    <message name="GraphRequestMessage">
        <part name="params" type="ns:GraphRequest"/>
    </message>
```

```
    <message name="GraphResponseMessage">
        <part name="response" type="ns:GraphResponse"/>
    </message>

    <portType name="AspectInstrumentationPortType">
        <operation name="MakeAspect">
            <input name="input2" message="tns:GetRequestMessage"/>
            <output name="output2" message="tns:GetResponseMessage"/>
        </operation>
        <operation name="AIGraphXML">
            <input name="input" message="tns:GraphRequestMessage"/>
            <output name="output" message="tns:GraphResponseMessage"/>
        </operation>
    </portType>

    <binding name="AspectInstrumentationPortTypeBinding"
             type="tns:AspectInstrumentationPortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="MakeAspect">
            <soap:operation/>
            <input name="input1">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/AspectInstrumentation"/>
            </input>
            <output name="output1">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/AspectInstrumentation"/>
            </output>
        </operation>
        <operation name="AIGraphXML">
            <soap:operation/>
            <input name="input2">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/AspectInstrumentation"/>
            </input>
            <output name="output2">
                <soap:body use="literal"
                           namespace="http://j2ee.netbeans.org/wsdl/AspectInstrumentation"/>
            </output>
        </operation>
    </binding>

    <service name="AspectInstrumentationService">
        <port name="AspectInstrumentationPortTypeBindingPort"
              binding="tns:AspectInstrumentationPortTypeBinding">
            <soap:address location=
                 "http://localhost:8084/AspectInstrumentation/AspectInstrumentationService"
            />
        </port>
    </service>

    <plink:partnerLinkType name="AspectInstrumentationPartner">
        <plink:role name="AspectInstrumentationPortTypeRole"
                    portType="tns:AspectInstrumentationPortType"/>
    </plink:partnerLinkType>
    <plink:partnerLinkType name="AspectInstrumentationPartner1">
        <plink:role name="AspectInstrumentationPortTypeRoleGraph"
                    portType="tns:AspectInstrumentationPortTypeGraph"/>
    </plink:partnerLinkType>
</definitions>
```

Bibliography

[1] Activebpel. http://www.activebpel.org/.

[2] Apache axis. http://ws.apache.org/axis.

[3] Apache tomcat. http://tomcat.apache.org.

[4] Apache tuscany. http://incubator.apache.org/tuscany/.

[5] Axis: The next generation of apache soap.
http://www.javaworld.com/javaworld/jw-01-2002/jw-0125-axis.html.

[6] Daml-s: Semantic markup for web services.
http://www.daml.org/services/daml-s/0.7/daml-s.html.

[7] Glassfish project. https://glassfish.dev.java.net.

[8] Java ncss metrics computation program.
http://www.kclee.com/clemens/java/javancss/.

[9] Java optimze and decompile environment (jode).
http://jode.sourceforge.net/.

[10] The java orchestration language interpreter engine (jolie).
http://sourceforge.net/projects/jolie/.

[11] Jboss.com uddi inquiry example.
http://wiki.jboss.org/wiki/Wiki.jsp?page=UDDIExample.

[12] Jdom. http://www.jdom.org/.

[13] Junit. http://www.junit.org/.

[14] mysql sql server. http://www.mysql.com/.

[15] Saxon xml parser. http://saxon.sourceforge.net/.

[16] Selenium web tester. http://selenium.openqa.org/.

[17] Simple object access protocol (soap).
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[18] Soapui. http://www.soapui.org/.

[19] Sorcerer source code browser. https://sorcerer.dev.java.net/.

[20] Sun application server.
http://www.sun.com/software/products/appsrvr/index.xml.

[21] Uddi learning guide.
http://searchwebservices.techtarget.com/originalContent/0,289142,sid26_gci916789,00.html.

[22] Uddi publishing with java.
http://www.phptr.com/articles/printerfriendly.asp?p=101595&rl=1.

[23] Using soap with j2ee.
http://www.awprofessional.com/articles/article.asp?p=169106&seqNum=5&rl=1.

[24] Xml protocol technology reference.
http://webservices.xml.com/pub/a/ws/2000/11/01/protocols/quickref.html.

[25] Craig Anslow, Stuart Marshall, Robert Biddle, James Noble, and Kirk Jackson. Xml database support for program trace visualisation. In *APVis '04: Proceedings of the 2004 Australasian symposium on Information Visualisation*, pages 25–34, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[26] Craig Anslow, Stuart Marshall, James Noble, and Robert Biddle. Software visualization tools for component reuse. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[27] Apache. Apache tomcat, May 2007.

[28] BEA and IBM. Bpelj: Bpel for java.

[29] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. Migrating interactive legacy systems to web services. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 24–36, Washington, DC, USA, 2006. IEEE Computer Society.

[30] Anis Charfi and Mira Mezini. An aspect-based process container for bpel. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM Press.

[31] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995.

[32] Francesco Colasuonno, Azzurra Coppi, Stefano andRagone, Luca Scorcia, Tommaso Di Noia, and Eugenio Di Sciascio. juddi+: A semantic web services registry enablingsemantic discovery and composition. In *The 8th IEEE Conference on E-Commerce Technologyand the 3rd IEEE Conference on EnterpriseComputing*, 2006.

[33] A. Colyer and A. Clement. Aspect-oriented programming with aspectj. *IBM Syst. J.*, 44(2):301–308, 2005.

[34] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 266, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Ugur Dogrusöz, Brendan Madden, and Patrick Madden. Circular layout in the graph layout toolkit. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 92–100, London, UK, 1997. Springer-Verlag.

[36] Jan Dünnweber, Sergei Gorlatch, Françoise Baude, Virginie Legrand, and Nikos Parlavantzas. Towards automatic creation of web services for grid component composition. In Vladimir Getov, editor, *Proceedings of the Grids@Work Plugtest, Sophia-Antipolis, France*, October 2005.

[37] Michael Eichberg. Bat2xml: Xml-based java bytecode representation. *Electronic Notes in Theoretical Computer Science*, 141(1):93–107, December 2005. Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005).

[38] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[39] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[40] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. Technical Report 1–2000, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2000.

[41] David H. Hutchens and Victor R. Basili. System structure analysis: clustering with data bindings. *IEEE Trans. Softw. Eng.*, 11(8):749–757, 1985.

[42] Will Iverson. *Real World Web Services*. O'Reilly Media, Inc., October 2004.

[43] Nirav H. Kapadia, Renato J. Figueiredo, and José A. B. Fortes. Punch — Web portal for running tools. *IEEE Micro*, 20(3):38–47, /2000.

[44] Michael Kay. Defining your own functions in xquery. http://www.stylusstudio.com/xquery/xquery_functions.html.

[45] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-oriented composition in bpel4ws. In *WWW (Alternate Paper Tracks)*, 2003.

[46] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001*, pages 327–354, Heidelberg, June 2001. Springer Verlag.

[47] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis. A Probabilistic Approach to Source Code Authorship Identification. *Proceedings of the 5th International Conference on Information Technolog: New Generations (ITNG 2007, Las Vegas, April 7-9),. IEEE Computer Society*, 2007.

[48] Eleftherios Koutsofios and Stephen C. North. Editing graphs with *dotty*, 1994. *dotty* User Manual.

[49] Robert Charles Lange and Spiros Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In Hod Lipson, editor, *GECCO*, pages 2082–2089. ACM, 2007.

[50] M. Lanza. Codecrawler — lessons learned in building a software visualization tool, 2003.

[51] Grace Lewis, Edwin Morris, Dennis Smith, and Liam O'Brien. Service-oriented migration and reuse technique (smart). In *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 222–229, Washington, DC, USA, 2005. IEEE Computer Society.

[52] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, Washington, DC, USA, 1999. IEEE Computer Society.

[53] Spiros Mancoridis, Timothy S. Souder, Yih-Farn Chen, Emden R. Gansner, and Jeffrey L. Korn. Reportal: A web-based portal site for reverse engineering. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[54] Jim Melton and Stephen Buxton. *Querying XML: XQuery, XPath, and SQL/XML in context (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[55] Sun Microsystems. Jax-r tutorial. http://java.sun.com/developer/technicalArticles/WebServices/jaxrws.

[56] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Software Eng.*, 32(3):193–208, 2006.

[57] William M. Mongan, Maxim Shevertalov, and Spiros Mancoridis. Re-engineering a reverse engineering portal to a distributed soa. In *16th International Conference on Program Comprehension (ICPC 2008)*. IEEE Computer Society, 2008.

[58] Aart van Halteren Nikolai Dokovski, Ing Widya. Paradigm: Service oriented computing, 2004.

[59] OASIS Committee on Service Oriented Architecture. Reference model for service oriented architecture, 2006.

[60] Hewlett Packard. Web services orchestration.

[61] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition, 2002.

[62] Shankar R. Ponnekanti and Armando Fox. Interoperability among independently evolving web services. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 331–351, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[63] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions, 2002.

[64] Harry M. Sneed. Integrating legacy software into a service oriented architecture. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society.

[65] Harry M. Sneed and Stephan H. Sneed. Creating web services from legacy host programs. *wse*, 00:59, 2003.

[66] UDDI.org. http://www.UDDI.org/.

[67] W3C. Web service choreography interface (wsci).

[68] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[69] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA, 1997. IEEE Computer Society.

[70] Andreas Winter. Exchanging Graphs with GXL. Technical Report 9–2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[71] Andreas Winter. Exchanging Graphs with GXL. Technical Report 9–2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[72] Dan Wu, Bijan Parsia, Evren Sirin, James Hendler, and Dana Nau. Automating daml-s web services composition using shop2.

[73] Xiulan Yu, Long Zhang, Ying Li, and Ying Chen. Wsce: A flexible web service composition environment. In *ICWS*, pages 428–435. IEEE Computer Society, 2004.

[74] Olaf Zimmermann, Mark R. Tomlinson, and Stefan Peuser. *Perspectives on Web Services : Applying SOAP, WSDL and UDDI to Real-World Projects (Springer Professional Computing)*. Springer, September 2003.