Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Incremental dependency analysis over large software infrastructure

SCIENTIFIC STUDENTS' ASSOCIATIONS REPORT

Author
Donát Csikós

Supervisors
dr. István Ráth, Ákos Horváth

October 18, 2012

Contents

1 Introduction **3**
- 1.1 Control systems at CERN . . . 3
- 1.2 Main goal: Smooth upgrades . . . 3

2 Related Technologies **5**
- 2.1 Java Runtime . . . 5
  - 2.1.1 Java byte code specification TODO . . . 5
  - 2.1.2 Apache Commons Byte Code Engineering Library - TODO . . . 6
- 2.2 Eclipse . . . 6
  - 2.2.1 Eclipse Integrated Development Environment . . . 6
  - 2.2.2 Eclipse Java Development Tools . . . 7
  - 2.2.3 Eclipse Modeling Framework . . . 8
  - 2.2.4 EMF-IncQuery . . . 8
- 2.3 Other related technologies . . . 9
  - 2.3.1 Spring Framework . . . 9
  - 2.3.2 Maven . . . 9
  - 2.3.3 Tycho . . . 10
  - 2.3.4 Oracle database . . . 10

3 Overview **11**
- 3.1 Architecture . . . 11
  - 3.1.1 Central repository management . . . 11
  - 3.1.2 The server side . . . 11
  - 3.1.3 The client side . . . 12
- 3.2 Service Provider Framework . . . 13

4 Details **16**
- 4.1 Infrastructure at CERN controls systems . . . 16
  - 4.1.1 Used tools . . . 16
  - 4.1.2 Development workflow . . . 16
- 4.2 Bytecode analyzer . . . 16
  - 4.2.1 Anatomy of a Java binary . . . 16
  - 4.2.2 The analysis process . . . 17
  - 4.2.3 Extracted domain model . . . 18

4.2.4 Anatomy of class files . . . 19
4.2.5 Execution of dependency discovery . . . 19
4.3 Dependency processor . . . 20
4.4 Persisting dependencies . . . 20
4.5 Direct queries . . . 20
4.6 Repository EMF model . . . 21
4.7 Creation of workspace EMF model . . . 21
4.8 Pattern matching . . . 21

5 Evaluation **22**
5.1 Example revisited . . . 22
5.2 Performance evaluation . . . 22

6 Conclusion and Future work **23**

Bibliography **24**

A Service Provider Framework source **25**

Chapter 1

Introduction

1.1 Control systems at CERN

The European Organization for Nuclear Research known as CERN is an international organization which runs the the world's biggest particle physics laboratory. Established at 1954 and situated on the Swiss-French border next to Geneva. The organization's main goal to operate particle accelerators (such as the Large Hadron Collider) and all the necessary infrastructure.

Obviously not only physicists work az CERN: numerous scientist and engineers are working on the design and the maintenance of the software and hardware equipments in a well-defined structure. One member of this community is the Controls Group which is responsible for – amongst the others – the design and implementation of both software and hardware used in the controls systems. This involves providing software frameworks, providing alarm systems and communication middleware and numerous related tasks.

After the LHC became operational the primary objective for the Controls Group is to maintain uninterrupted operation without any downtime. This implies that a upgrading certain parts of the systems should not interfere with arbitrary components.

The controls systems itself is a complex, distributed and highly modular system which has three tiers. On the top there are the GUI applications which are written in Java. The middle or business layer is also consists of Java applications. On the lowest or hardware level there are C/C++ applications running on real-time systems. Maintaining a complex system like this requires a set of special approaches provide desired quality and maximize the availability of the systems in one.

1.2 Main goal: Smooth upgrades

Smooth upgrade means when a software is being updated than all the other softwares which depend in the upgraded one will remain operational. A basic example of braking this principle, when the newer version is binary incompatible with dependant softwares and cause load-time error.

There are a variety of tools and methods how to achieve smooth upgrades. For in-

stance one can introduce manual procedures and protocols to the development lifecycle. Another possibility to enforce some code metrics to achieve this goal.

The last approach – which my solution tries to implement – is the concept of *incoming dependencies*. As an example let's see a developer who is working a Java library which happens to be used by some clients. Now let's say there is an error in the bug which requires a change in the library's API for the resolution. Our developer obviously wants to know who is using that part of the API. The users (as in code parts) are considered as incoming dependencies.

My solution which I present in this paper is a possible implementation for querying incoming dependencies. The first valid question is: why didn't we just reuse an existing implementation? The answer is: because there is no one. There are specialities about the situation my tool has to solve.

Size limitation If we are able to load every single jars into an IDE the problem would be solved: we could use the internal navigation features to navigate through the dependency graph. But because we have thousands of jars (and tens of thousands if we count the possible versions) it is evidently impossible.

Fine-graininess of dependencies The existing tools usually cannot determine the desired dependency resolution level.

No source code analysis We don't want to do a source code analysis because it would imply a dependency resolution for concerned projects every time a new version of a product comes out. This requirement eliminates a number of possible tools.

Because of these reasons we decided to implement our own solution internally called Dependency Analysis Tool. In the following chapters I will present the design and the implementation for this tool and how is it currently applied az CERN Controls Group.

Chapter 2

Related Technologies

Before proceeding to the main part of this paper, I am going to give an overview about the related technologies used by my solution. I Why do we have to present the related technologies.

2.1 Java Runtime

2.1.1 Java byte code specification TODO

The Java [8] programming language is a general-purpose, concurrent, object-oriented language. Its syntax is similar to C and C++, but it omits many of the features that make C and C++ complex, confusing, and unsafe. The Java platform was initially developed to address the problems of building software for networked consumer devices. It was designed to support multiple host architectures and to allow secure delivery of software components. To meet these requirements, compiled code had to survive transport across networks, operate on any client, and assure the client that it was safe to run.

The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at runtime. It is reasonably common to implement a programming language using a virtual machine. The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the `class` file format. A `class` file contains Java virtual machine instructions (or *bytecodes*) and a symbol table, as well as other ancillary information.

The class file format is a hardware- and operating system-independent binary format which is executed by the virtual machine. Typically (but not necessarily) stored in a file. The class file format precisely defines the representation of a class or interface, including details such as byte ordering that might be taken for granted in a platform-specific object file format.

Figure Figure 2.1 illustrates the procedure of compiling and executing a Java class: The source file is compiled into a Java class file, loaded by the byte code interpreter and executed. In order to implement additional features, researchers may want to transform class files (drawn with bold lines) before they get actually executed. This application area is one of the main issues of this article.

Figure 2.1: *Compilation and execution of Java classes*

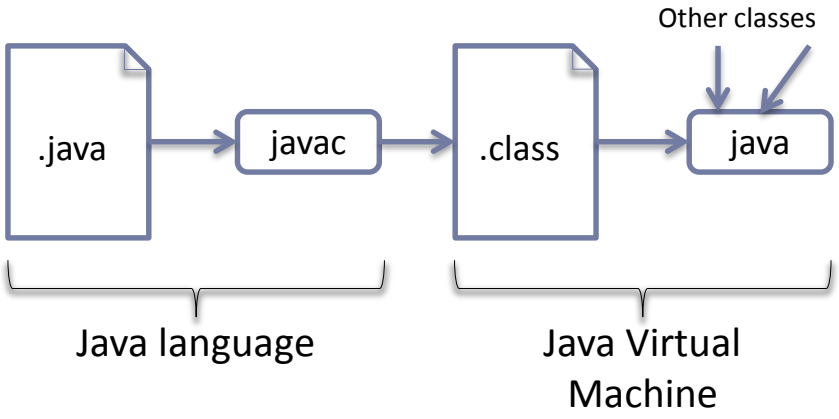


Figure Figure 4.1 shows a simplified example of the contents of a Java class file: It starts with a header containing a "magic number" (0xCAFEBABE) and the version number, followed by the constant pool, which can be roughly thought of as the text segment of an executable, the access rights of the class encoded by a bit mask, a list of interfaces implemented by the class, lists containing the fields and methods of the class, and finally the class attributes, e.g., the SourceFile attribute telling the name of the source file. Attributes are a way of putting additional, user-defined information into class file data structures. For example, a custom class loader may evaluate such attribute data in order to perform its transformations. The JVM specification declares that unknown, i.e., user-defined attributes must be ignored by any Virtual Machine implementation.

2.1.2 Apache Commons Byte Code Engineering Library - TODO

[1] Abstraction over Java byte code specification. How it works. Create diagram! Possible use-cases.

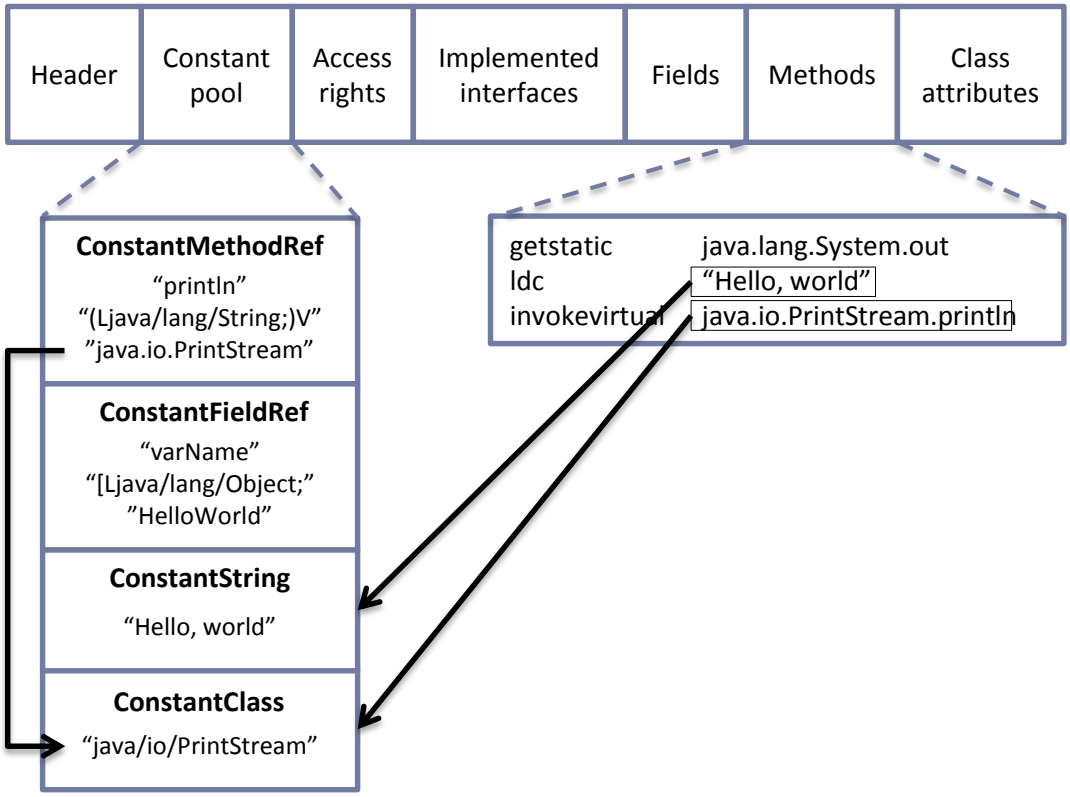
2.2 Eclipse

2.2.1 Eclipse Integrated Development Environment

The Eclipse Project [3] is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It was developed by IBM from 1999, and a few months after the first version was shipped, IBM donated the source code to the Eclipse Foundation.

The Eclipse project consists of many subprojects, the most important being the Eclipse Platform, that defines the set of frameworks and common services that collectively make up „integration-ware" required to support a comprehensive tool integration platform. These services and frameworks represent the common facilities required by most tool builders, including a standard workbench user interface and project model for managing resources, portable native widget and user interface libraries, automatic resource delta

Figure 2.2: *Simplified structure of a class file*



management for incremental compilers and builders, language-independent debug infrastructure, and infrastructure for distributed multi-user versioned resource management.

The Eclipse Platform has an easily-extendable modular architecture, where all functionality is achieved by plugins, running over a low-level core called Platform Runtime. This runtime core is only responsible for loading and connecting the available plugins, every other functionality, such as the editors, views, project management, is handled by plugins. The plugins bundled with Eclipse Platform include general user interface components, a common help system for all Eclipse components, project management and team work support.

2.2.2 Eclipse Java Development Tools

The Eclipse Java Development Tools (JDT) project [5] contributes a set of plugins that add the capabilities of a full-featured Java IDE to the Eclipse platform. The JDT plugins provide APIs so that they can themselves be further extended by other tool builders. The JDT plugins are categorized into five main projects: Annotation Processing (APT), Core, Debug, Text, and User Interface (UI).

The most important part – in the context of this paper – is the Core project which implements all the necessary non-UI infrastructure for developing Java applications on the Eclipse platform. It contains an incremental Java builder, support for code assist, a searching facility and a *Java Model*, that provides API for navigating the Java element tree.

This lets the contributors access the structure and the changes of the java applications loaded into the workspace.

2.2.3 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [2] is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF is also maintained by the Eclipse Foundation within the scope of Eclipse Tools Project. EMF started out as an implementation of the OMG Meta Object Facility (MOF) specification, and evolved into a highly efficient product for model-based software design.

EMF requires a metamodel as an input; it can import metamodels from different sources, such as Rational Rose, XMI or XML Schema documents, or a special syntax can be used to annotate existing Java source files with EMF model properties. Once an EMF model is specified, the built-in code generator can create a corresponding set of Java implementation classes. These generated classes can be edited to add methods and instance variables; additions will be preserved during code-regeneration. In addition to simply increasing productivity, building an application using EMF provides several other benefits like model change notification, persistence support including default XMI and schema-based XML serialization, a framework for model validation, and a very efficient reflective API for manipulating EMF objects generically. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

EMF has a built-in serialization facility, which enables the developer to save (and load) in-stances of the model into industry-standard XMI format. EMF also provides a notational and persistence mechanism, on top of which model-manipulating tools can easily be constructed.

2.2.4 EMF-IncQuery

Working with models it is quite common to query the model for checking and validating certain properties. The EMF-IncQuery [4] tool provides one solution for this problem: it is a framework to execute fast model queries over EMF models. It is actively developed at Budapest University of Technology and Economics. Although it is still in development status (the current version is 0.6.0) it already proved its usability through industrial use-cases and university researches.

The core of the framework is a query evaluator engine built on top of graph pattern matching engine using RETE [7] algorithm adapted from expert systems to facilitate the efficient storage and retrieval of partial views of graph-like models. In a nutshell RETE maintains a hierarchical query data structure on top of the model which stores the result of sub-queries. On model change, the event propagates through this data structure leaving the unmodified part of the model untouched. This results in fast, near zero response time and size-independence on small model change. In return the model and the query structure has to be loaded into the memory which can be a significant resource expense.

To access the capabilities of the core, an easy-to-use, type safe API is defined. Using

the API, EMF resources and object hierarchies can be loaded and queried incrementally. In addition certain extensions – such as the validation framework – can be attached.

Along the API, a complete query language is defined. It provides a declarative way to express the queries over the EMF model in a form of patterns. With the language the user can express combined queries, negative patterns, checking property conditions, simple calculations, calculate disjunctions and transitive closures, etc.

The framework comes with a rich UI tooling which helps the users to effectively develop test and integrate queries into their solution. The first element of the tooling is the rich XText based editor for the query language which aids writing well formed queries providing content assist, error markings and such. The next part of the tooling is the ability to load EMF models and execute the queries on them as the user writes them giving rich visual feedback about the result. The last important part is the code generation. The tooling dynamically generates the source code which contains the programmatic equivalent of the model queries. The users can integrate this code out of the box in Eclipse plugins as well as headless applications to execute queries and get back the results from the source code level.

2.3 Other related technologies

2.3.1 Spring Framework

The Spring Framework is an open source application framework and Inversion of Control container for the Java platform. The core features of the Spring Framework can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Although the Spring Framework does not impose any specific programming model, it has become popular in the Java community as an alternative to, replacement for, or even addition to the Enterprise JavaBean (EJB) model.

The main advantage of using Spring framework is a configurability through dependency injection.

2.3.2 Maven

In three words Apache Maven is a project management framework. The main goal is similar to Ant but Maven offers an integrated approach to make Java development manageable.

Maven standardizes the entire development process by defining explicit lifecycle steps for dependency management, building, packaging, deploying and so on. Although the lifecycle elements are set, they can be configurable through the key element of Maven: the Project Object Model (POM). POM is an XML document in the project's folder describing every necessary information about it to make maven be able to do its job.

Through the POM file almost everything related to the project is configurable, yet the most important feature is the dependency management. One can define certain dependencies in this file, and during the build Maven will download them automatically from

the Maven repositories.

Also POM is the place to define almost every relevant information about the project: source and binary folders, package format, deployment location, etc. These information are used during the build process.

Maven is also extendable through plugin mechanism. One can develop extensions for it for achieve various non-standard goals such as integration testing or building non-standalone Java applications.

2.3.3 Tycho

Tycho is a set of Maven plugins and extensions for building Eclipse plugins and OSGi bundles with Maven. It exists because Eclipse plugins have their own way of describing metadata such as version numbers and dependencies which normally would be placed in the Maven POM. Tycho reuses this native metadata and uses the POM to configure and drive the build. Tycho also knows how to run JUnit test plugins using OSGi runtime and there is also support for sharing build results using Maven artifact repositories. Tycho introduces new packaging types and the corresponding lifecycle bindings that allow Maven to use OSGi and Eclipse metadata during a Maven build.

Although Tycho is useful for building Eclipse plugins in a headless way but it still has not reached the production quality (the current version number is 0.16) and it is still under heavy development. The current modifications and enchantments can be found on the project's website.

2.3.4 Oracle database

The Oracle Database (also known as Oracle DB, Oracle RDBMS or just Oracle) is a relational database management system (RDBMS) from the Oracle Corporation. Originally developed in 1977 by Lawrence Ellison and other developers, Oracle DB is one of the most trusted and widely-used relational database engines.

The system is built around a relational database framework in which data objects may be directly accessed by users (or an application front end) through structured query language (SQL). Oracle is a fully scalable relational database architecture and is often used by global enterprises, which manage and process data across wide and local area networks. The Oracle database has its own network component to allow communications across networks.

Chapter 3

Overview

In this chapter...

3.1 Architecture

As it was discussed previously one of the most effective way of achieving smooth upgrades is to discover incoming dependencies. For this I designed and implemented a solution called Dependency Analysis Tool. Figure Figure 3.1 shows the overview of it.

This systems was implemented in two separate steps. My work at CERN covered an implementation of a system which gives the ability to the users to query certain parts of the source code for incoming dependencies. After my job was done I created an extension which utilizes EMF-IncQuery to run faster, information-rich and more extensive queries. Let's examine the elements of the figure one-by-one.

3.1.1 Central repository management

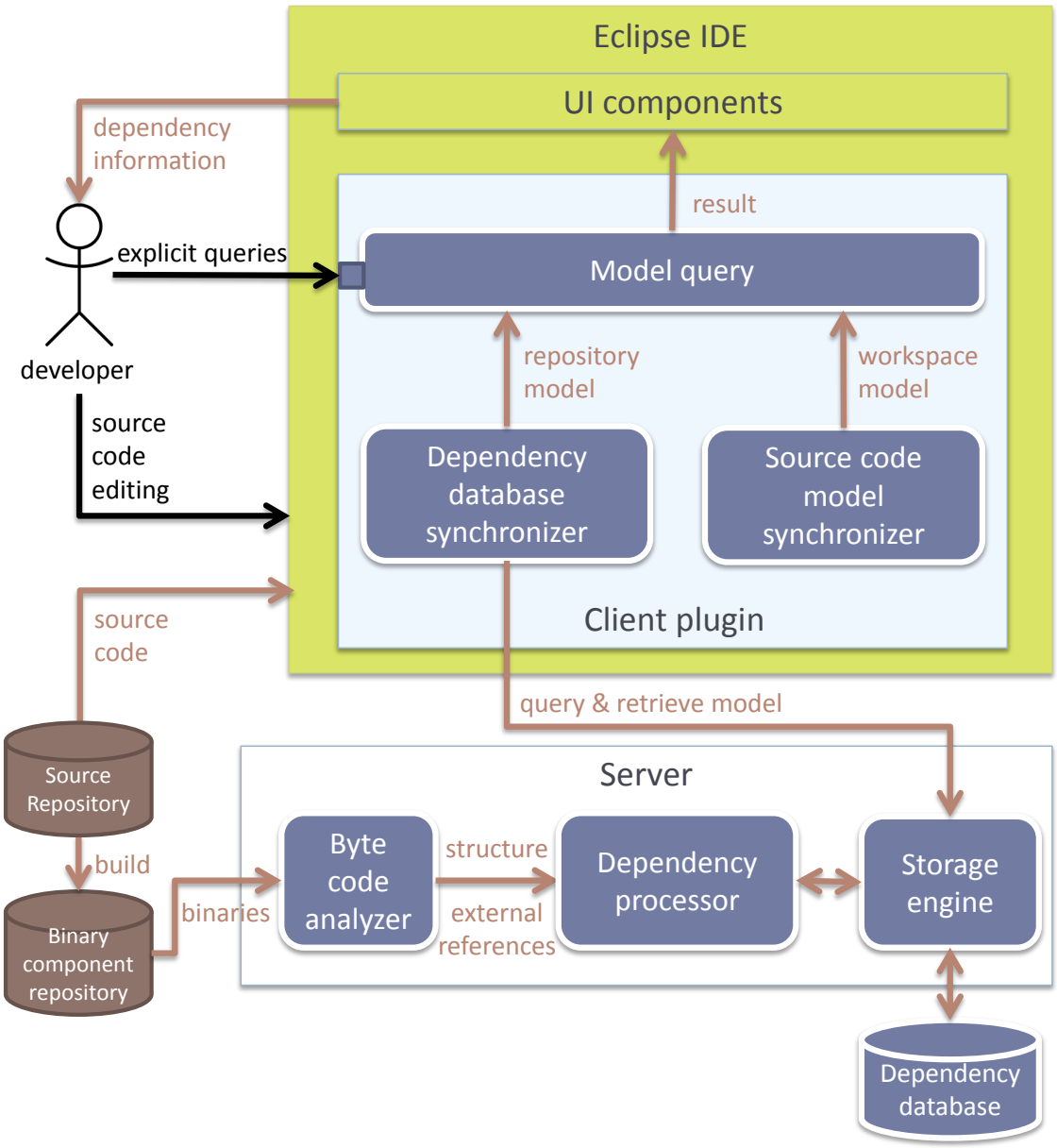
The first element on the figure is the „Central repository management". It refers all the internal resources CERN Control systems contains. Its central element is a tool called `Common Build` which is a build tool for Java softwares. Common Build is an Apache Ant based software similar to Maven. It provides functionality to describe and resolve dependencies, build, generate documentation and release the softwares[1]. The released softwares are put into a binary repository which has a well-defined layout. On top of it, source repository is tightly attached to the release repository; there is a 1-1 relationship between the source code and the binaries.

3.1.2 The server side

The server side is a standalone Java application which runs on a Linux server. It has two functionality: 1) it discovers and stores the structure of the products and 2) provides an interface for serving dependency queries.

[1] Or *products* as they are referred at CERN

Figure 3.1: *Overview of the implemented system*



For discovering, the process listens if a new release happens in the repository. If it happens the freshly added binaries (jar files) are passed to the byte code analysis module which parses the file and discovers the contained structure and the dependencies utilizing the Apache BCEL library. The structure and the dependencies are passed to the storage engine to store it. The storage engine itself defines a set of operations to find, store and retrieve certain subset of the model. The remote query interface also use this module to get the necessary dependency information for the clients.

3.1.3 The client side

The client-side of the solution is an Eclipse plugin (or more precisely a set of plugins) which gives the developer a convenient way to access to the dependency information.

The base of the plugin is the repository model loader. It provides a simple API for accessing and querying the dependency information from the server. The simple use-case for this, when the user directly asks the dependency information from the Java source editor through UI contribution (marked as direct queries on the figure).

The workspace model creator is module generating an EMF model describing the state of the workspace. The generated model contains the loaded projects, the contained packages, classes, methods, etc. The model also contains the dependency structure between the elements (e.g. method calls, field accesses and such). The information is gathered through the Eclipse Java Development Tools' API. After the model is created it is incrementally maintained even through restarts.

The pattern matcher module executes complex model queries on top of the acquired models to provide dependency information. First it loads an EMF model from the server describing the structure of all projects in the central repository. Afterward the workspace model is loaded from the module described above. Both of the models are loaded to the EMF-IncQuery engine. Through complex queries the two models are joined and queried for the dependency information. With this more accurate and extensive dependency analysis can be achieved because this approach takes into account the changes which were done in the workspace. The workspace model is updated upon every change of the workspace giving up-to-date feedback for the users while they are working on the development.

Both direct queries and the pattern matching module return the result in Eclipse views. After the result is evaluated, the user's responsibility to evaluate their results and act accordingly.

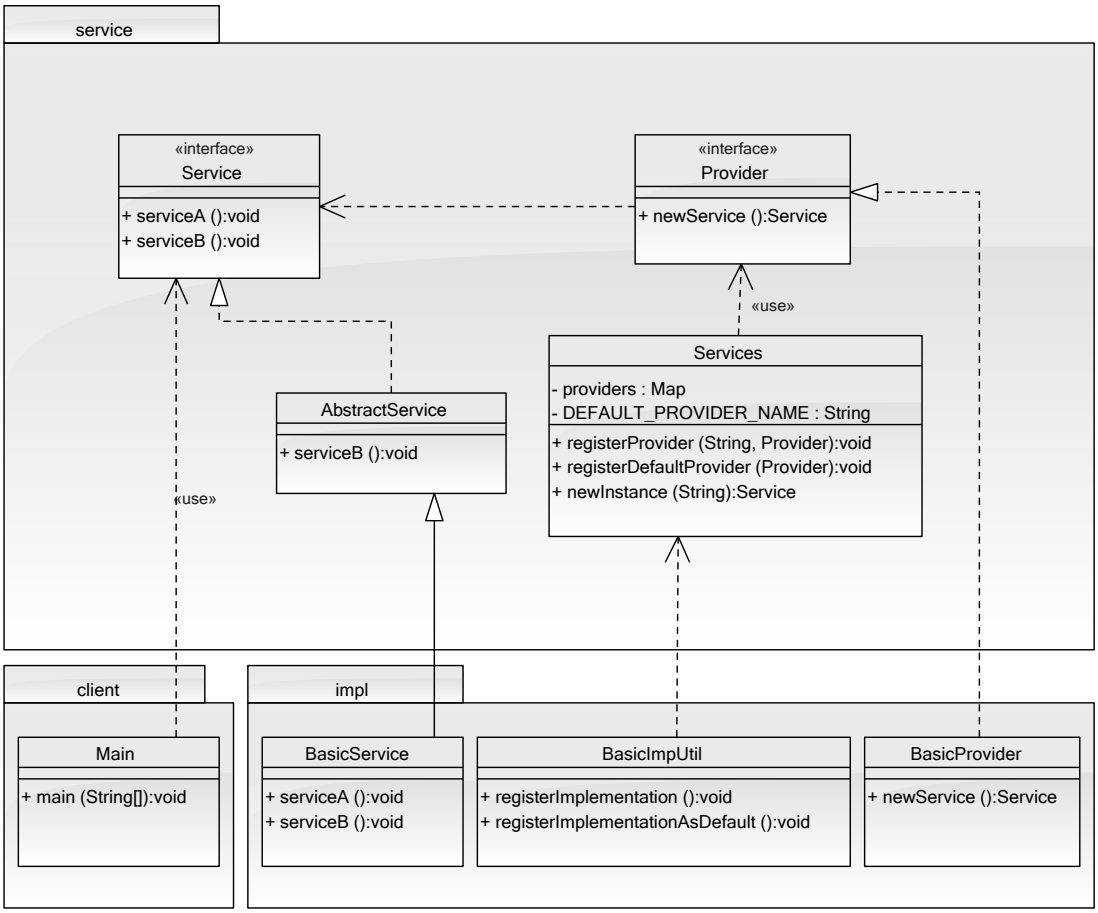
Using direct queries brings no limitations. The queries are simple remote method calls and the result set is a relatively small data set which easy to store and present on the Eclipse UI. On the other hand, the pattern matching solution is far from being that easy, because in order to load make EMF-IncQuery work, the entire repository has to be loaded. By default this model is a few hundred megabytes sized in a serialized form. This implies that the implementation has to optimize this model without dropping useful information to make it loadable to the memory.

3.2 Service Provider Framework

To make the following chapters easier to understand, I am going to introduce a simple use-case example. It is a design pattern called Service Provider Framework. It is a practical application of the original Adapter pattern and it was described in the famous book „Effective Java" [6]. This pattern is the simplified version how the Java Database Connectivity (JDBC) works.

You can see the structure of the pattern on figure Figure 3.2. As of the packages it contains 3 major parts. The `service` package contains the core pattern classes. The `impl` and the `impl` packages are external users of the pattern and therefore they are considered depending client libraries.

Figure 3.2: *Structure of Service Provider Framework pattern*



First let's discuss the pattern itself. The main goal for the patter is to provide a registry of implementations for a desired service. This service is described in the `Service` interface. The `Provider` interface is serves as a factory instance; it has a single function to instantiate a new Service object. The `Services` class has the role of the registry. The service implementers register their implementation using the static `registerProvider()` and `registerDefaultProvider()` methods. The parameters the identifier string for the registered service and Provider instance which will instantiate the Service instances. The clients will instantiate the Service instance with the `newInstance()` function. Depending the passed identifier string, the method will look up if a Provider was registered with the same name, and if the answer is yes then it calls the `Provider.newInstance()` and return its result.

The `DEFAULT_PROVIDER_NAME` is a static public field which can be used to obtain the default Service implementation. The `AbstractService` is a utility class which implements one of the function of the Service interface.

The `impl` package contains one possible implementation to use the described pattern. The `BasicService` contains the implementation and the `BasicProvider` is responsible for properly initializing and returning a new instance of this version. The `BasicImplUtil` – as its name implies – holds utility classes which register the implementation in the Services

class.

The `client` package holds one single `Main` class, which contains a simple evaluation. It invokes the `Services.newInstance()` function passing the default provider's name as an argument and invokes the `serviceA()` and `serviceB()` function.

Although the example is fairly simple, the related source code can be found in appendix Appendix A.

So why are we looking at this example? Imagine that the three packages are distributed into three different software which have their individual responsible developer. Now let's say, somebody wants to change some parts of the service without precisely knowing who is using which part of the code. Not this will be our example which I will use to present how my software works and how can this problem be solved by it.

Chapter 4

Details

4.1 Infrastructure at CERN controls systems

4.1.1 Used tools

- Describe, how a typical developer works at CERN BE-CO - Tools: SVN, Eclipse, Common Build, JIRA, etc.

4.1.2 Development workflow

- Original idea comes from the typical development workflow lifecycle at CERN control systems.

4.2 Bytecode analyzer

Before the system performs dependency analysis, it needs to extract the necessary information from the source code. The Bytecode analyzer module takes the input java binaries (in the form of jar files) parses the contained class files with the help of Apache BCEL and maps it into an object graph which can be effectively used during the dependency processing.

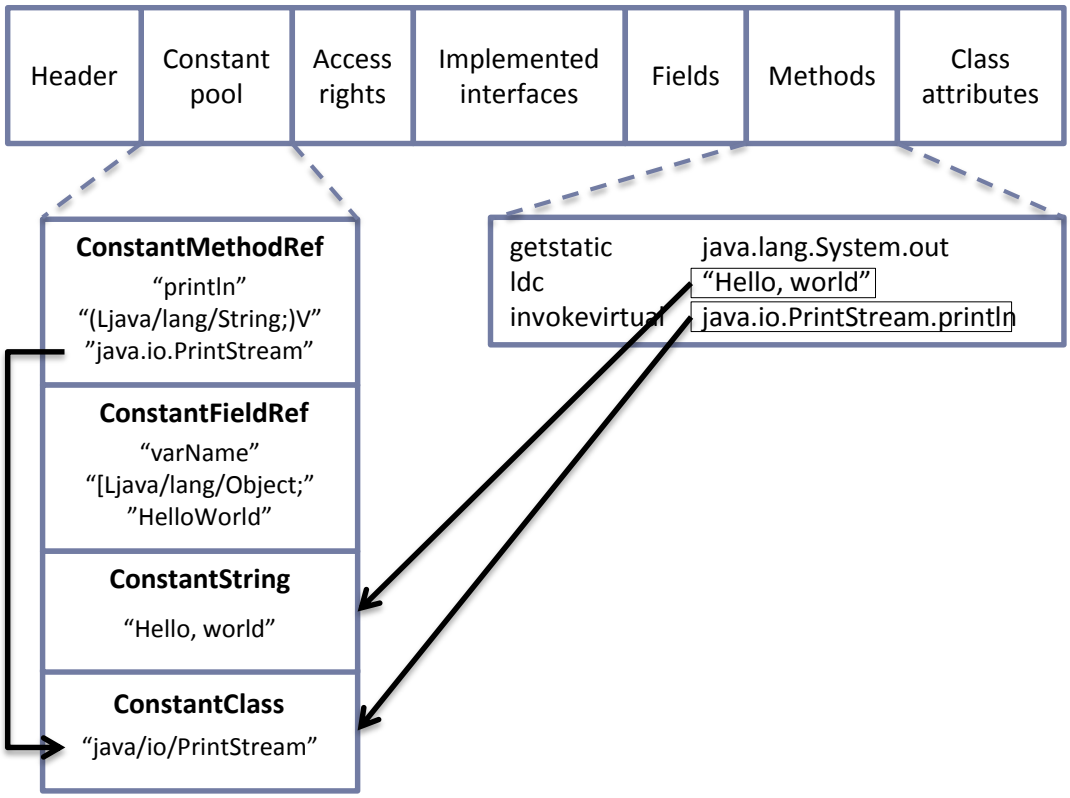
4.2.1 Anatomy of a Java binary

The Bytecode analyzer module takes a set of jars as an input. A jar file is essentially a zip file containing Java-related resources: resource files, binary class files and meta-information. Because the system has to discover dependencies between pure Java applications then no meta-information is used; the bytecode analyzer gets the information from the class files contained in the jars.

Figure Figure 4.1 shows a simplified view how a single class file is built. This structure is precisely defined in Java Runtime Specification. The class begins with a marker header followed by the part called *constant pool* which contains the textual part of the code. It contains the i) constant strings ii) referenced methods and fields and iii) the names of the external classes. The Java virtual machine is only able to load a class file if all the

referenced external classes are loaded.

Figure 4.1: *Internal structure of a class file*



After the constant pool the access flags, the implemented interfaces and the field list are located in separate places.

Afterwards comes the definition of the methods. It contains all the necessary information for the virtual machine to execute the methods: the resources to allocate for the execution, the exception handlers and the bytecode itself.

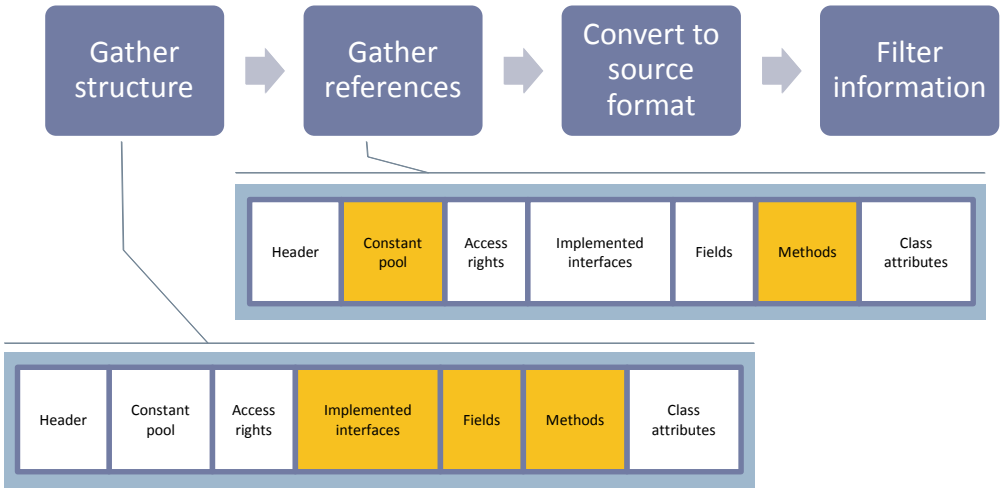
The big question is what information can be extracted from the jar/class files for the dependency analysis? The short answer is everything. The structure of a class file can be obtained one-by-one. The external references what we are looking for are also fairly easily to extract, because they are defined in the constant pool. The only challenging part to solve is to match, where exactly are the external resources are used in the bytecode itself.

4.2.2 The analysis process

The module extract all necessary information from the class files for processing them. This could be done by brute-force parsing binary files, but it would be an tedious and error-prone task. Instead of this the implementation reuses Apache BCEL to effectively parse the class files and and to obtain the necessary information via simple API calls. Figure 4.2 shows the steps of the analysis process and the related information in the class files for each step.

The analysis starts with gathering the basic structure. This involves acquiring the class’

Figure 4.2: *Steps of the bytecode analysis process*



name, the name of the extended class and the implemented interfaces, the defined fields and methods. This information is accessible out of the box through the BCEL API.

The second step is to acquire all external reference pointing outside of the class files. For imported classes it is easy because this is what the `ConstantClass` entries cover in the constant pool. For field and method references, the implementations searches `ConstantFieldRef` and `ConstantMethodRef` occurrences in the bytecode and saves all methods which uses them.

The next step is to convert the information into source format. This is necessary because both the structure and the external references are presented in a format which not readable nor could be easily queried by the users of the data. For example the commonly used `println(String)` function has the following binary format: `println(Ljava/lang/String;)V`. The implementation transforms it into `println(java.lang.String):void`.

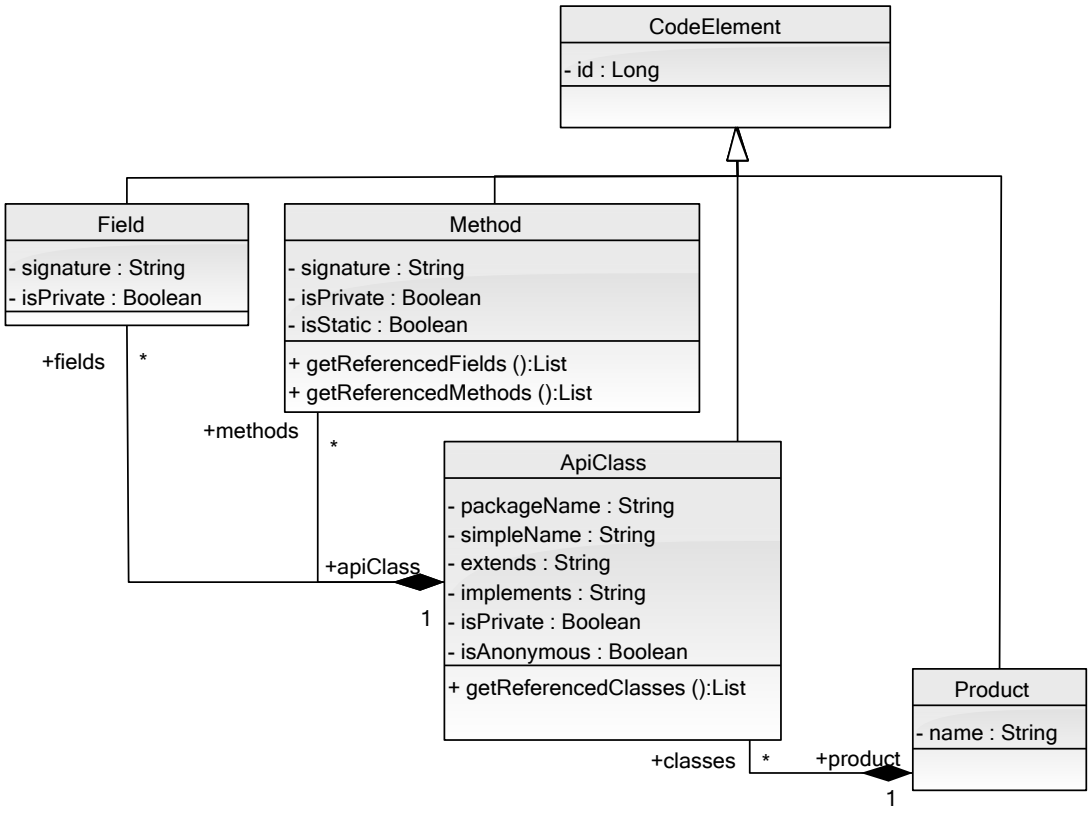
The final step is to cleanup the gathered information. At this step have to consider that if we mapped all information then the gathered data would have a comparable size with the binary repositories which is unacceptable when we have to deal with thousands of jars. To resolve this, the implementation does multiple cleanup steps. First it drops the private methods and fields, because it is not explicitly accessible and by this no dependency would point them. The other trick is to drop a subset of the external references. The platform-provided elements are dropped (references pointing inside the `java.*` package) and the ones which point inside the jar files. This is reasonable because any IDE gives access this information through code traversal capabilities. Of course this cleanup needs do be done after all class files are parsed in a jar file.

4.2.3 Extracted domain model

The output of the analyzer is a java domain model. Figure 4.3 shows the elements of this model. Because this model is used for the dependency processing too the model has additional elements which is now not important.

All element inherits from abstract `CodeElement` class which is a top-level interface for

Figure 4.3: *Classes of the domain model used by the bytecode analyzer*



handing the items of the model. The processed jar files are named as `Products`, because that's what it is: a software product. A product contains several classes named `ApiClass` which store the class-level properties. The required classes are stored in the `referencedClasses` list. The classes contain some `Field`s and some `Method`s which both have a – source formatted – signature and some access properties. The `referencedFields` and the `referencedMethods` hold all the external field and method references which are accessed or invoked in the bytecode of the represented method.

With an instance of this domain model the dependency processing module is capable of discovering the dependency relationships between certain parts of the dependency.

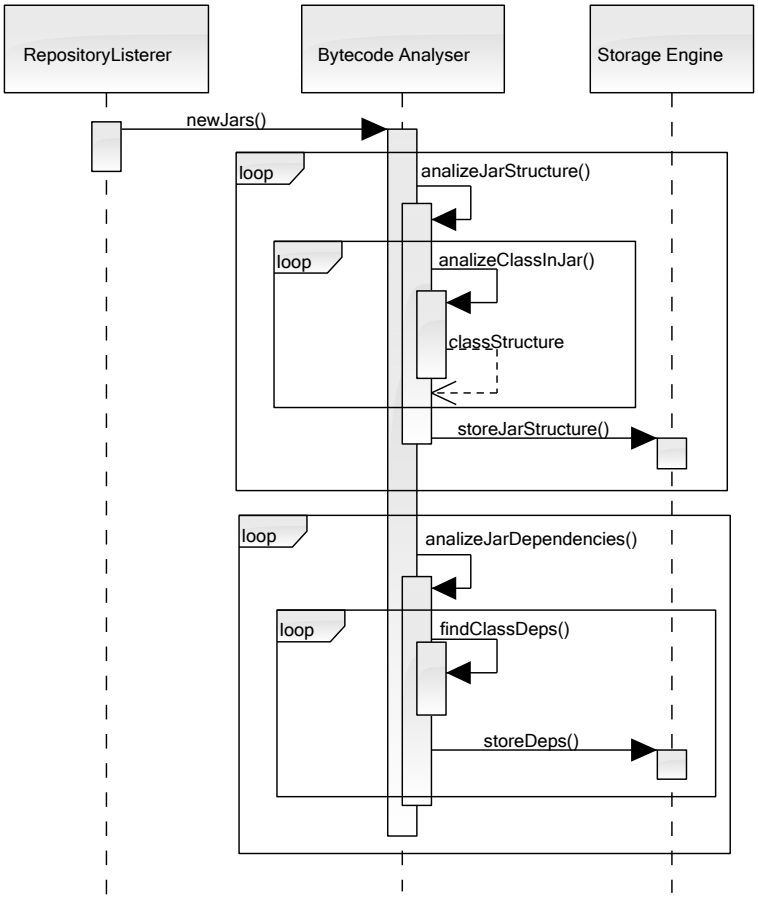
4.3 Dependency processor

- Sequence, how the dependency analysis works. - Summarize what should be found if execute the analysis in the example

4.4 Persisting dependencies

- describe the saved domain model. - describe the how the operations work: * store structure * find and insert dependencies - introduce different implementation - the result

Figure 4.4: *Sequence of bytecode analysis*



4.5 Direct queries

- Explain eclipse plugin; ui contribution. - Simple workflow: right click on source code->jdt resolves it as a class-method-field-> fully qualify it -> sends the query for the server process -> the process returns the incoming dependencies depending the passed type. -> the result is visualized in a view. - Show it on the example: what if the developer wants to change the the signature of a service function OR what if he wants to change the name of the name of the default provider name.

4.6 Repository EMF model

4.7 Creation of workspace EMF model

- Java Model representation in Eclipse. - Dependency search in Eclipse. - EMF model generation: * How to obtain the structure * How to obtain internal dependency environment * and how to listen to changes.

4.8 Pattern matching

- Mention again that the source code and the repository contains the same information. (+svn tags) - How to obtain more precise information. - The repository model. - Describing the queries. * joining the two implementation. * incoming dependencies * impact analysis * added and removed methods * removed methods * changed methods: the outgoing dependency set has new elements, which does not exist in the repo model. - Show it on the example.

Chapter 5

Evaluation

5.1 Example revisited

Show how the tool solves the problem. Go through all components, show all results one by one, what is the input and the output.

5.2 Performance evaluation

Pure measurement. What are the

Chapter 6

Conclusion and Future work

I created a tool. It can be useful for a library developer wqorking at large, software-oriented organization where lots of inter-depending software are developed and maintained. Becomes extremely useful when the concept of smooth upgrades is essential.

Implemented a precise, source-based query solution along with a not entirely precise but fast and reactive display of dependency data.

As for the future there are plenty of directions to go: Slicing the repository model to load just the required part of it. Notification about the repository change. Define and extract more detailed dependencies from the binaries. Extend the dependency resolution to c++ software.

Bibliography

[1] Apache bcel documentation. http://commons.apache.org/bcel/manual.html, October 2012.

[2] Eclipse modeling framework website. http://www.eclipse.org/emf, October 2012.

[3] Eclipse project website. http://www.eclipse.org, October 2012.

[4] Emf-incquery website. https://viatra.inf.mit.bme.hu/incquery/documentation, October 2012.

[5] Java development tools website. http://www.eclipse.org/jdt, October 2012.

[6] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[7] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[8] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

Appendix A

Service Provider Framework source

Listing A.1: *Classes of the* `service` *package*

```
public interface Service {
        void serviceA();
        void serviceB();
}

public abstract class AbstractService implements Service {
        public void serviceB() {System.out.println("Default serviceB().");}
}

public interface Provider {
        Service newService();
}

public class Services {
        private static final Map<String, Provider> providers =
                new ConcurrentHashMap<String, Provider>();
        public static final String DEFAULT_PROVIDER_NAME = "<default_provider>";

        private Services() {
        }
        public static void registerDefaultProvider(Provider p) {
                providers.put(DEFAULT_PROVIDER_NAME, p);
        }
        public static void registerProvider(String name, Provider p) {
                providers.put(name, p);
        }
        public static Service newInstance(String name) {
                Provider p = providers.get(name);
                if (p == null)
                        throw new IllegalArgumentException("Provider does not exist.");
                return p.newService();
        }
}
```

Listing A.2: *Classes of the* `impl` *package*

```
public class BasicService extends AbstractService {
        public void serviceA() { System.out.println("Executed basic service"); }
}

public class BasicProvider implements Provider {
        public Service newService() { return new BasicService(); }
```

```
}

public class BasicImplUtil {
        public static void registerImplementation() {
                Services.registerProvider("basic", new BasicProvider());
        }
        public static void registerImplementationAsDefault() {
                Services.registerProvider(Services.DEFAULT_PROVIDER_NAME,
                        new BasicProvider());
        }
}
```

Listing A.3: *Classes of the* `client` *package*

```
public class Main {
        public static void main(String[] args) {
                BasicImplUtil.registerImplementationAsDefault();
                Service service = Services.newInstance(Services.DEFAULT_PROVIDER_NAME);
                service.serviceB();
        }
}
```