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 27, 2012

Contents

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 of its members is the Controls Group which is responsible for the design and implementation of both software and hardware used in the controls systems. This involves providing software frameworks, alarm systems, 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 in order to provide desired quality and to maximize the availability of the systems.

1.2 Smooth upgrades

One of the main goals is to achieve *smooth upgrades*. Smooth upgrade means when a software is being updated than all the other softwares which depend on it will remain

operational. A straightforward 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 to achieve smooth upgrades. For instance 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 the solution presented in this paper implements – is the analysis of *incoming dependencies*. As an example let's say 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 wants to check who is using the API, which parts are used and should remain unattached and which are the unused parts. The users (as in code parts) are considered as incoming dependencies. The following chapters describe an implementation for querying these 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. In normal case seems possibe. But if we look at he CERN Control Systems, where we have thousands of jars (and tens of thousands if we count the possible versions) this approach is – evidently – impossible.

Fine-graininess of dependencies The existing tools usually cannot determine the desired dependency resolution level. For example JDepend [7] and JBoss Tattletale [6] does dependency analysis on the class level, but we want to see method-level and fields level dependencies as well.

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 also eliminates a number of possible tools.

After considering these arguments we at CERN decided to implement our own solution internally called Dependency Analysis Tool. In the following chapters I will present the design and the implementation of this tool, how is it currently applied az CERN Controls Group and how it was extended at Budapest University of Technology.

Chapter 2

Related Technologies

2.1 Java Runtime

2.1.1 Java byte code specification

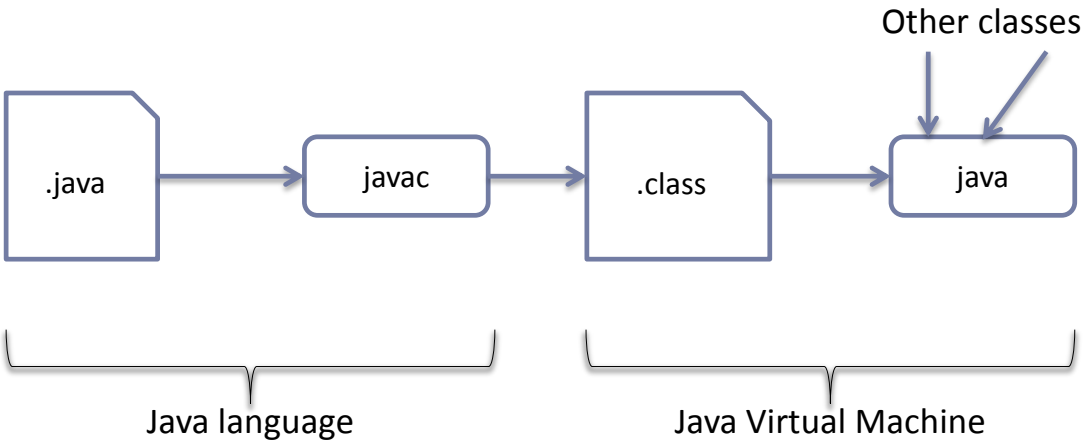
The Java [11] programming language is a general-purpose, concurrent, object-oriented language. Its syntax is similar to C and C++, but it omits many of the features that make C and C++ complex, confusing, and unsafe. The Java platform was initially developed to address the problems of building software 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 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.

Figure 2.1: *Compilation and execution of Java classes*



Internals of a class file

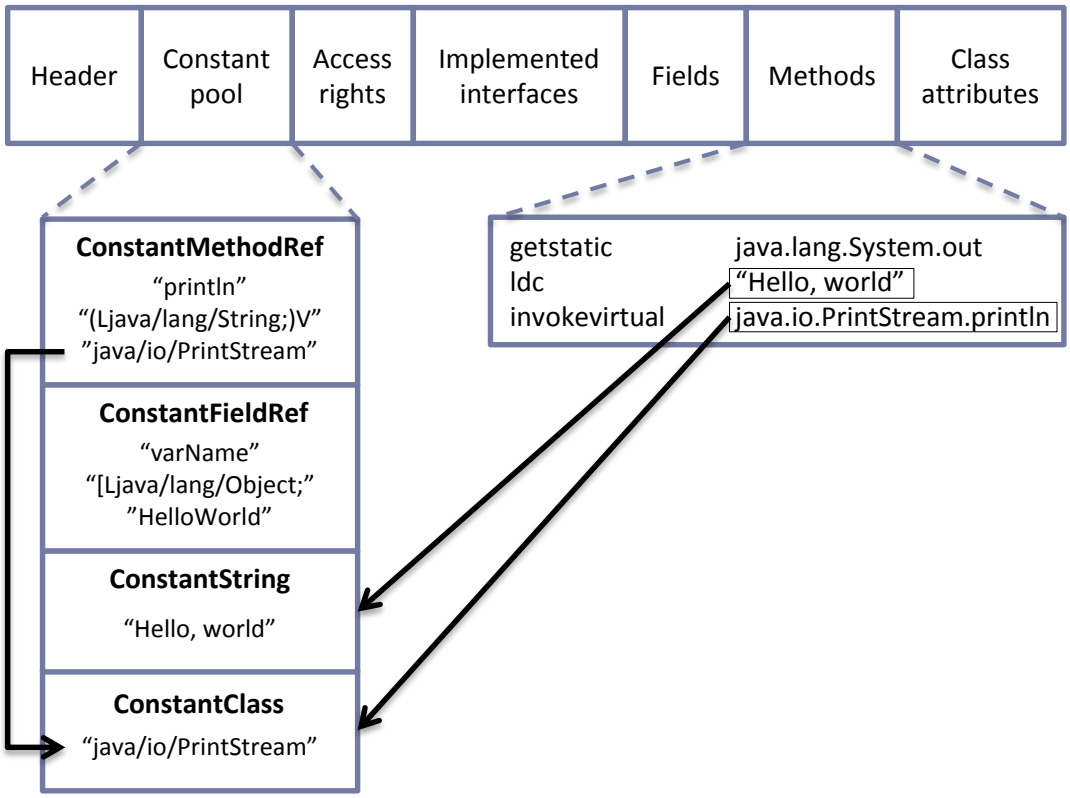
Figure 2.2 shows a simplified example of the contents of a Java class file: It starts with a header containing a "magic number" (0xCAFEBABE) and the version number, followed by the constant pool, which can be roughly thought of as the text segment of an executable, the access rights of the class encoded 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.

Because all of the information needed to dynamically resolve the symbolic references to classes, fields and methods at run-time is coded with string constants, the constant pool contains in fact the largest portion of an average class file, approximately 60%. In fact, this makes the constant pool an easy target for code manipulation issues. The byte code instructions themselves just make up 12%.

The right upper box shows a "zoomed" excerpt of the constant pool, while the rounded box below depicts some instructions that are contained within a method of the example class. These instructions represent the straightforward translation of the well-known `System.out.println("Hello, world");` statement.

The first instruction loads the contents of the field out of class `java.lang.System` onto the operand stack. This is an instance of the class `java.io.PrintStream`. The `ldc` ("Load constant") pushes a reference to the string "Hello world" on the stack. The next instruction invokes the instance method `println` which takes both values as parameters (Instance methods always implicitly take an instance reference as their first argument).

Figure 2.2: *Simplified structure of a class file*



Instructions, other data structures within the class file and constants themselves may refer to constants in the constant pool. Such references are implemented via fixed indexes encoded directly into the instructions. This is illustrated for some items of the figure emphasized with a surrounding box.

For example, the `invokevirtual` instruction refers to a MethodRef constant that contains information about the name of the called method, the signature (i.e., the encoded argument and return types), and to which class the method belongs. In fact, as emphasized by the boxed value, the `MethodRef` constant itself just refers to other entries holding the real data, e.g., it refers to a `ConstantClass` entry containing a symbolic reference to the class `java.io.PrintStream`. To keep the class file compact, such constants are typically shared by different instructions and other constant pool entries. Similarly, a field is represented by a `Fieldref` constant that includes information about the name, the type and the containing class of the field.

The constant pool basically holds the following types of constants: References to methods, fields and classes, strings, integers, floats, longs, and doubles.

2.1.2 Apache Commons Byte Code Engineering Library

The Byte Code Engineering Library (BCEL) [1] is intended to give users a convenient way to analyze, create, and manipulate (binary) Java files. Classes are represented by

objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular.

The BCEL API abstracts from the concrete circumstances of the Java Virtual Machine and how to read and write binary Java class files. The package responsible for reading the class files is called the ”static” part. It contains a set of classes reflecting the class file format not intended for byte code modifications. The classes may be used to read and write class files from or to a file. This is useful especially for analyzing Java classes without having the source files at hand. The main data structure is called `JavaClass` which contains methods, fields, etc..

By using the BCEL library we can extract all static information from the Java binaries without ending up doing parsing binary data by hand. With this library it fairly easy to i) gather any structural information such as class name, superclass, signature of the defined methods, etc. and ii) process all external references defined in the bytecode.

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 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 Java application development on the Eclipse platform. It contains an incremental Java builder, support for code assist, a searching facility and a *Java Model*, which 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 task to query the model for checking it 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 [10] 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 changes. 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. on top of the models.

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. Using its features it is a solid platform for creating clear, well-tested and maintainable applications.

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, which will be download 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 settings get applied 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

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 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 Repository management

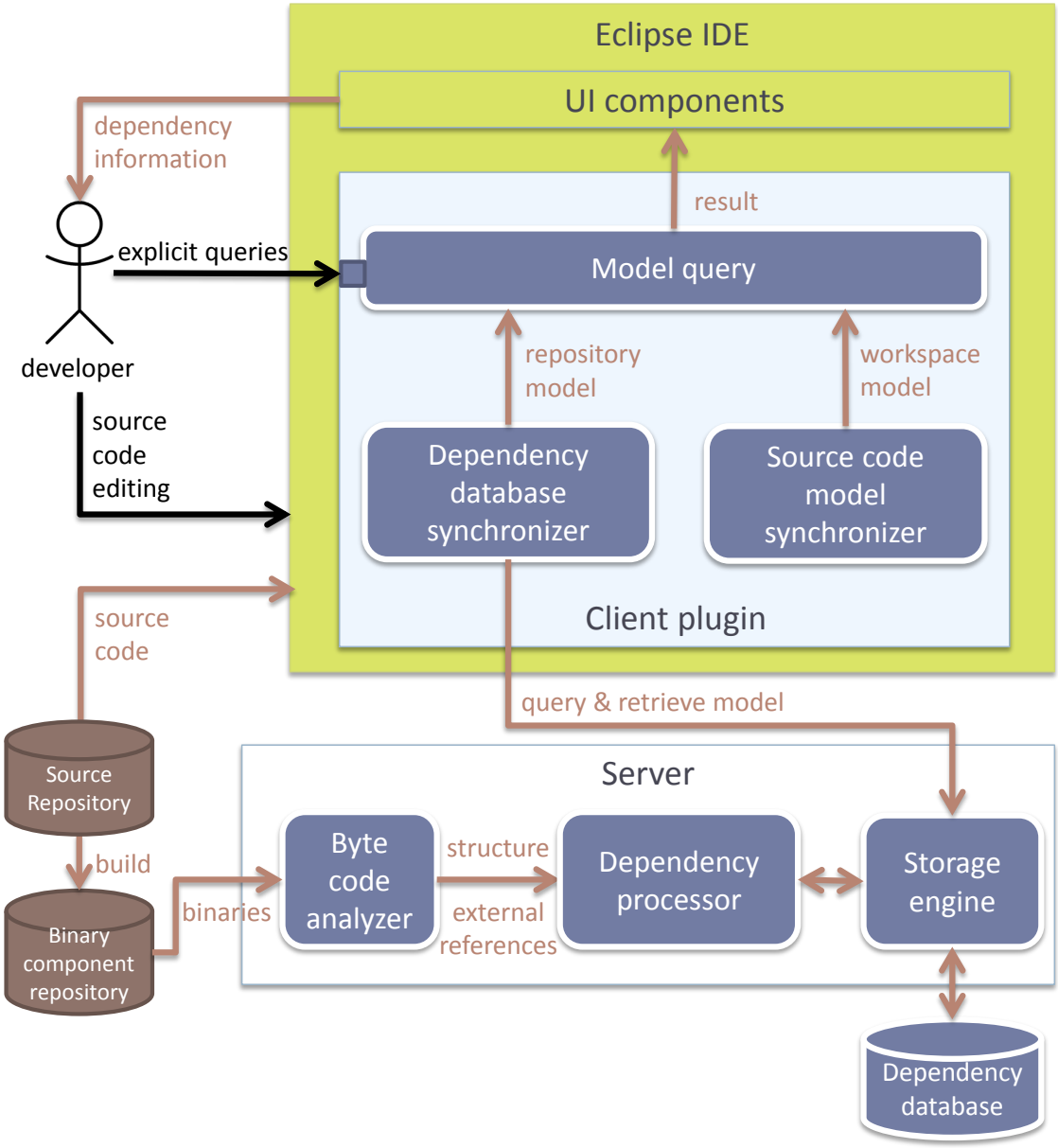
The first elements on the figure to discuss are the "Source repository" and the 'Binary component repository' . They contain both the source code and the build binary version of the internally developed softwares[1]. These elements are centrally managed and have a well-defined structure. The developers work on the source code and if they finish one milestone they publish their improvements by putting the compiled version into the binary repository through an automated process. The binary repository is the input for the dependency analysis.

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.

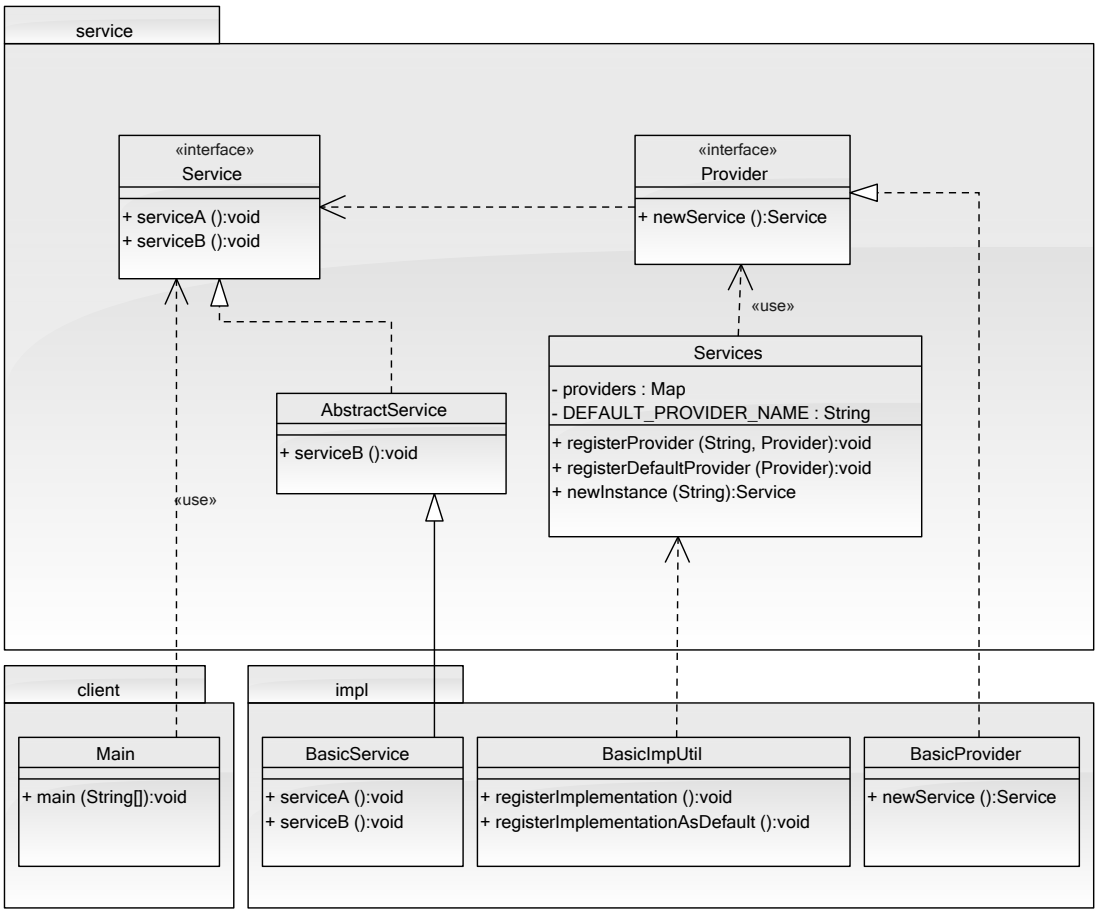
Using direct queries brings no limitations. The queries are simple remote method calls and the result set is a relatively small data set which easy to store and present on the Eclipse UI. On the other hand, the pattern matching solution is far from being that easy, because in order to load make EMF-IncQuery 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

3.2.1 Description of the pattern

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" [8]. This pattern is the simplified version how the Java Database Connectivity (JDBC) works.

Figure 3.2: *Structure of Service Provider Framework pattern*



You can see the structure of the pattern on 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.

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 ser-

vice 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 A.

3.2.2 Description of the problem

As it was described in the introduction we would like give the developers the ability to check who is using the certain parts of the API. To present this this pattern will be our use-case.

Imagine that the three packages are distributed into three different software which have their individual developers. If somebody wants to change some parts of the service without precisely knowing who is using which part of the code he won't know how many dependant projects will be broken afterwards.

Let's see two examples. The responsible for the service package wants to modify two parts of the classes: he wants to add a string parameter to the `Service.serviceA()` method, and he wants to change the name of the `Services.DEFAULT_PROVIDER_NAME` field.

By using Dependency Analysis Tool the developers are able to execute queries on any part of the API. The result will be the primary starting point for considering what to do. Three different outcome can happen: the selected part of the API doesn't have dependencies, it has a few dependencies or it has got a lot and the developer's decision depends on it.

If there is no incoming dependency than any modifications can be done without causing

any compatibility issues. The developer can do whatever he wants. If there is a lot of incoming dependency, than it means that the queried code element is used by a lot of clients and any modifications would cause a build error therefore backward compatibility has to be maintained. The third option is that there are several dependencies. In this case the developer has to negotiate with the clients how to proceed: he has to provide backward compatibility or the clients has to align their code to the modifications.

Chapter 4

Details

4.1 Infrastructure at CERN controls systems

At CERN Controls group has a definite set of tools used for development. Because C/C++ and Java applications are developed, the used Integrated Development Environment is an internally maintained Eclipse distribution with correct plugins installed.

To maintain concurrent versions of the source code an SVN server is used. Also there are a variety of tools used for typical developments tasks: issue etc.

The central element for the development is a tool called `Common Build` [9]. It is a build and release 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. This tool was developed before the first version of complex build systems such as Apache Maven was published so Common Build remained as an in-house build tool.

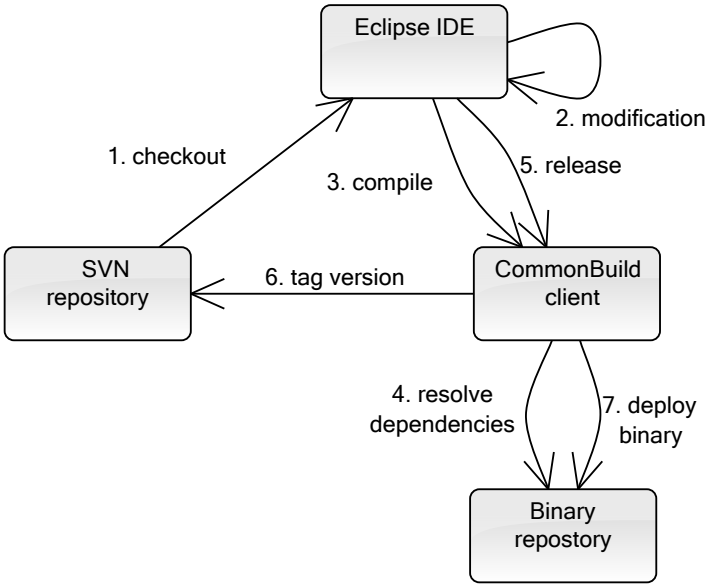
Common build is tightly coupled both to the SVN server and to the binary release repository as it automates the not just the building but the release of the Java products too.

Figure 4.1 highlights this relationship by showing a typical workflow of the development process. The developer first checks out the source code from the SVN repository and starts working on it. Along source code modification there is an XML descriptor containing the information required by Common Build (such as name and version of the product and the required dependencies). Also Common Build is capable to setup build path and the related options for Eclipse development.

When the source code is ready, the developer executes the Common Build client to build it. The client itself is a customized Ant distribution and acts in the same way. It resolves the dependencies from the dedicated binary repository which contains all previously released products and all used third party libraries. The content is also defined in a XML file.

If all source code is ready the developer can hit release. This fist initiates the same

Figure 4.1: *Build workflow using Common Build*



build process, but it is extended by two things. First the source code is tagged in the SVN repository which the version number as tag name. Second the compiled jar file is put into the binary repository.

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

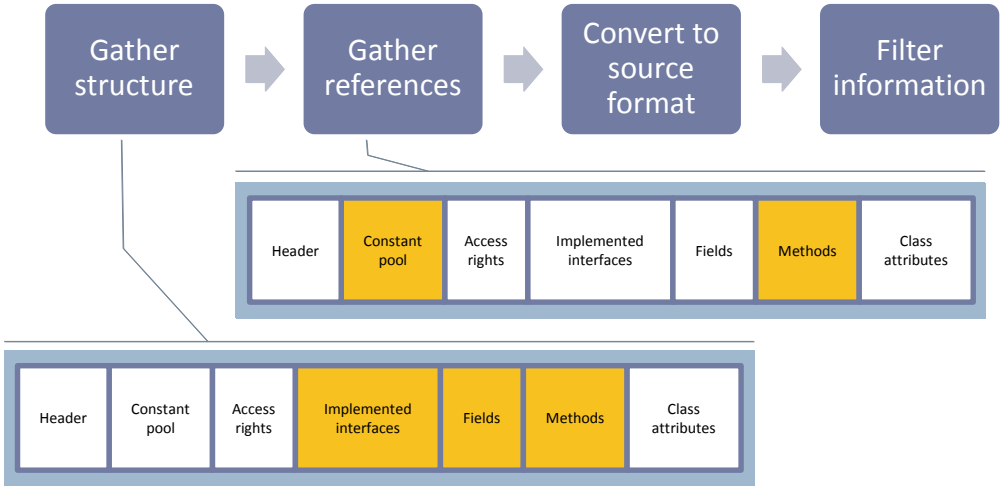
Afterwards comes the definition of the methods. It contains all the necessary information for the virtual machine to execute the methods: the resources to allocate for the execution, the exception handlers and the bytecode itself.

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

Figure 4.2: *Steps of the bytecode analysis process*



files for each step.

The analysis starts with gathering the basic structure. This involves acquiring the class' 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 clean up the gathered information. At this step we 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.

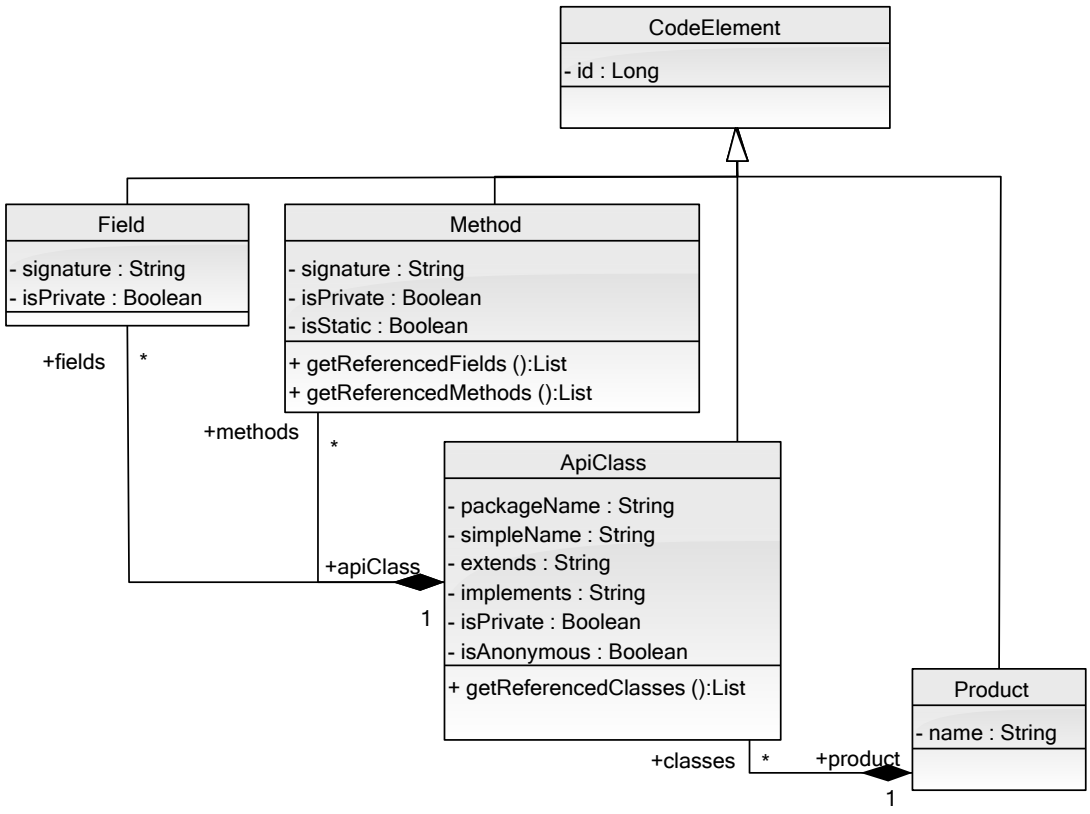
All element inherits from abstract `CodeElement` class which is a top-level interface for handing the items of the model. The processed jar files are named as `Products`, because that's what it is: a software product. A product contains several classes named `ApiClass` which store the class-level properties. The required classes are stored in the `referencedClasses` list. The classes contain some `Fields` and some `Methods` 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

The bytecode analyzer gathers all structural elements and the textual references of the dependencies. For resolving the references the dependency processor component is re-

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



sponsible.

To achieve this the processor takes the models of the jar files, compares the contained references with the structure of the external jars. When a match is found it means that there is a dependency between a two elements and as a result a dependency is stored.

4.3.1 Discovered dependencies

During the work at CERN we decided to narrow down the search for basic dependencies which could be extracted from the binary code without interpreting what does a class really do. By this we could exactly define what information will be accessible for the users. The following list contains the discovered dependencies:

Class import When a class uses *any* part of an external class. Practically this is a relationship between two classes when one class requires the other to get loaded in the Java virtual machine. On a binary level it is expressed as a `ConstantClass` entry in the constant pool.

Inheritance When a class inherits from another. This dependency type covers both the case when an interface is implemented and when a class is extended.

Method call When a method calls an another method. Covers both static and non-static method calls.

Method override When a class extends from an another and the subclass has a method with a same signature which was already defined in the superclass.

Field access When a method accesses or gives a value of a field defined in an external class.

4.3.2 Discovery process

After defining the searched dependencies let's see how exactly the dependency discovery process work. The main objective here is to do the analysis on a large set of Java binaries without having memory problems. Obviously if one loads all jars at the same to the memory and searches for all references it will need an enormous size of memory which will go up if the input number of the binaries increasing.

Figure 4.4: *Sequence of the dependency discovery process*

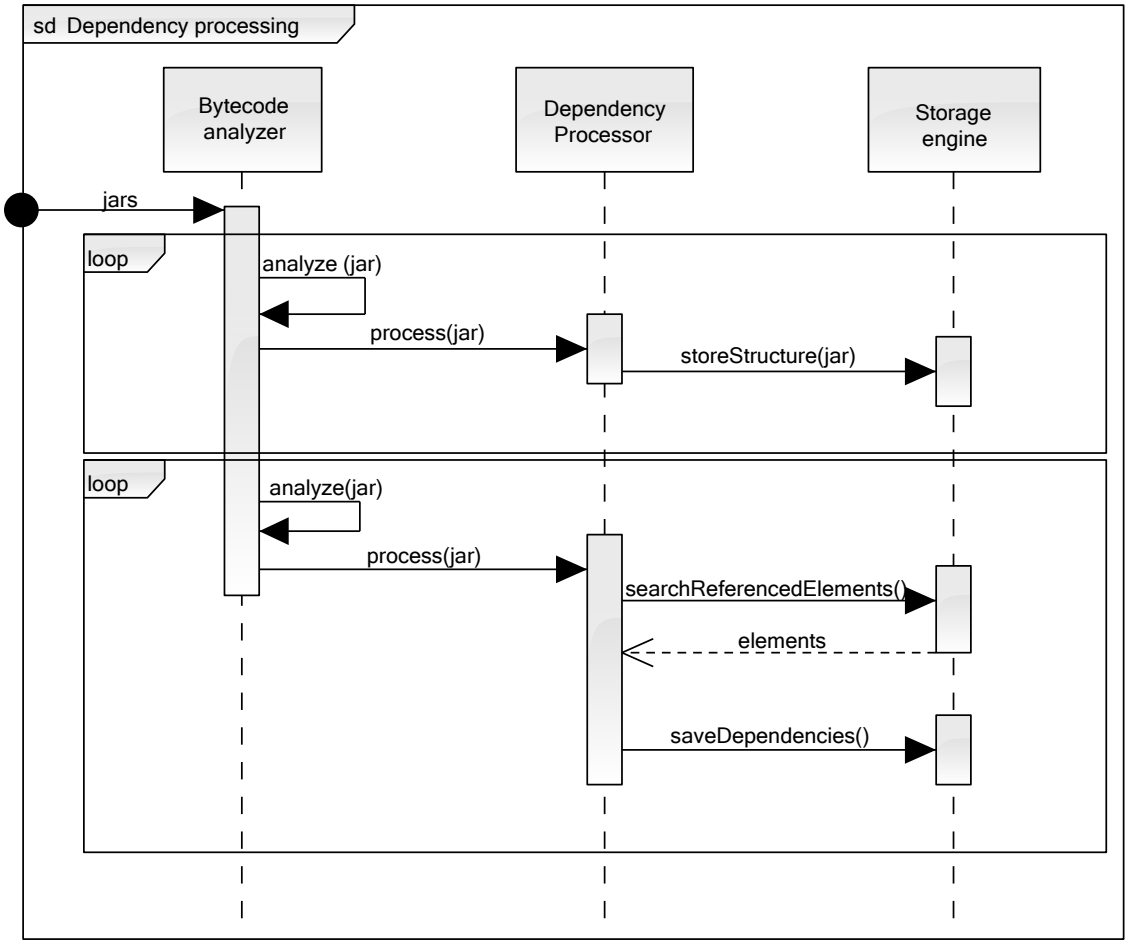


Figure 4.4 shows how the tool solves this issue. The process starts with an update in the binary repository. The new, not analyzed binaries are passed to the bytecode ana-

lyzer components which produces a model for each jar files. These models passed for the dependency processor which stores the jar structure in the database. By doing this the implementation holds only one model in the memory which implies that the memory requirement is independent from the number of input binaries. Also only the structure is pushed into the database, the references are considered as transient data. This is necessary because the references need huge space to store: on average they took 60% of the size of a class file and leaving them out is a big saving in the storage.

After all jar's structure is saved in the database, the process re-initiates the bytecode analysis on every jars one-by one and passes the models again to the dependency processor. The processor now tries to execute certain queries on the database for searching dependencies. For external references it tries to find the elements with the same fully qualified, for inheritance searches for superclass in the database and for method override it looks for methods with the same signature in the superclass. Every found element from the database result in a dependency entry at the end of the analysis of the current jar.

The result of the analysis is a database containing all the structure and the dependency information. The clients execute queries on it to analyze the relationships between certain elements in order to decide whether or not a specific code could be changed.

4.4 Storage engine

The dependency processing requires some database functionality to properly work. This is defined in the storage engine component.

Although this component is defined by a single Java interface it comes with important and complex responsibilities. First it is an abstraction layer over the concrete database implementation. Second it provides transactional behavior automatically to the database.

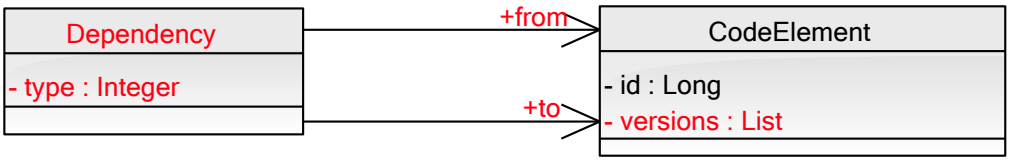
The interface defines operations for two purposes: i) Store and retrieve the structure of a jar and the dependencies ii) search for items based on their names and iii) query incoming dependencies of an element.

The usage of the first is easy the understand: if the analyze process starts it has to be checked if the a jar is in the database and if not, it has to be stored. The second in the list is part of the dependency processing. It is the responsibility of the database component how to represent the data and thus finding references also belongs here. The last part covers the queries which are initiated by the clients and the result is also provided by this module.

To make the domain model usable to store dependencies and make it work with the same jar products with multiple versions some additions were added (see Figure 4.5). Every element has a list of version numbers where they are present. Also a `Dependency` class is added to effectively handle dependencies between them.

This database engine is just a specification how it should work. It has two practical

Figure 4.5: *Additions to the domain model*



implementation: one which uses Oracle database and one which stores the data in an EMF model. The Oracle database maps the entities into tables and execute complex SQL queries in order to find the desired elements.

The EMF version is an in-memory implementation with an optional serialization feature implemented. The EMF metamodel used for creating the model has the same structure as the domain model (see Figure 4.3 and Figure 4.5).

4.5 Dependency database synchronizer

The Dependency database component is responsible for querying the server for dependency information related to a selected element or elements. It loads the sub-part of the dependency data stored by the storage engine module through a simple RMI interface and pushes it to the model queries components where the result is evaluated.

The component has to inputs: either it can load a compacted representation of the entire dependency model or it can query the incoming dependencies for just one single code element.

The single element query is the use-case where the user of the Dependency Analysis Toolexecutes direct queries by asking the incoming dependencies of one element in the workspace. In this case a simple RMI interface method is called where the argument is the queried object (as an instance of a ApiClass, a Method or a Field types from Figure 4.3. This argument is passed to the server, which turns it into an SQL query or a search on the EMF model (depending which back-end is loaded). The result is passed back as a collection of `Dependency` instances. The result model is unattached by the model queries component and gets instantly visualized in the UI component.

For letting the model queries component run real queries, this model has to load a representation of the data stored by the database engine component. If it was loaded one-by-one into an EMF model and passed to the client, then it would be simply too big to load. The binary repository holds more than a thousand jars just as latest production versions. The serialized EMF model equivalent is more than 400 MiB in size. To load an EMF model this size, more than one GiB of RAM is needed.

Important details about the environment at CERN is that the developers usually do the development on dedicated virtual machines which have more privileges to access internal resources. The drawback is that usually a virtual machine has 1-2 GiB RAM total, so we

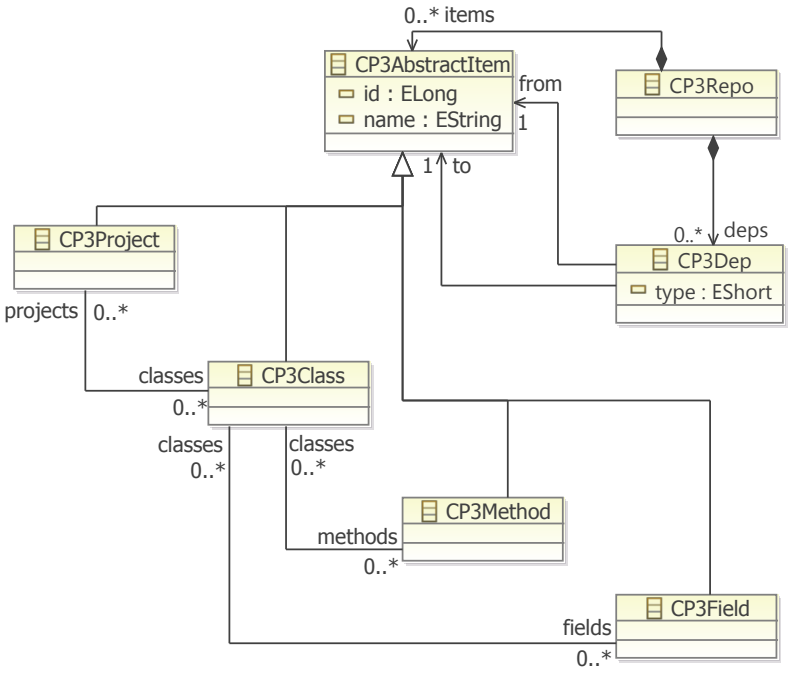
cannot make an Eclipse plugin which consumes all remaining resources at once. Also EMF-IncQuery has a practical limitation as it can easily handle EMF instance models up to 100 MiB.

These facts implied that the repository model has to be compacted. We can drop the unnecessary information or merge certain data and structure to spare some space without introducing false negative results. If we leave information the clients may end up false conclusions as they for example see an empty incoming dependency set where should be some.

Sure this compacting process introduces some false-positive results, but in return lets the model queries component to do super-fast model queries which are automatically updated every time the source code has changed.

The structure of the compacted EMF model is shown on Figure 4.6. The first thing

Figure 4.6: *Compacted repository model*



which was left out are most of the fields of the classes; only a `name` field stores the textual representation of the code element. This is reasonable because the dependency discovery is done on the server and we navigate only on the dependency edges. The second thing is not explicitly visible on the figure. The fully qualified name was merged down into simple names. It means that there is only one `Service` class for all products in the repository and one `newService()` function. If there are more than one exist, they are merged into one and also all dependencies becomes common. This also comes with the change that the original one-to-many containment relationships became many-to-many. The third compacting change is to drop the signature of the methods, only the name has left. The merging process is the same as described.

This compacted repository model is loaded at start-up time and used till Eclipse is closed. This is reasonable because the repository is not a rapidly changing entity. Only a few elements are modified each day. Apart from that it wouldn't be hard to implement a model eviction mechanism, where a new repository model is loaded every time when somebody does a change in the repository. Luckily this wouldn't change anything in the results of this paper.

The described compact model sadly contains some simplification but small enough to load it into the memory: the serialized representation of the entire workspace is around 70 MiB which is considered small enough to work with.

4.6 Source code model synchronizer

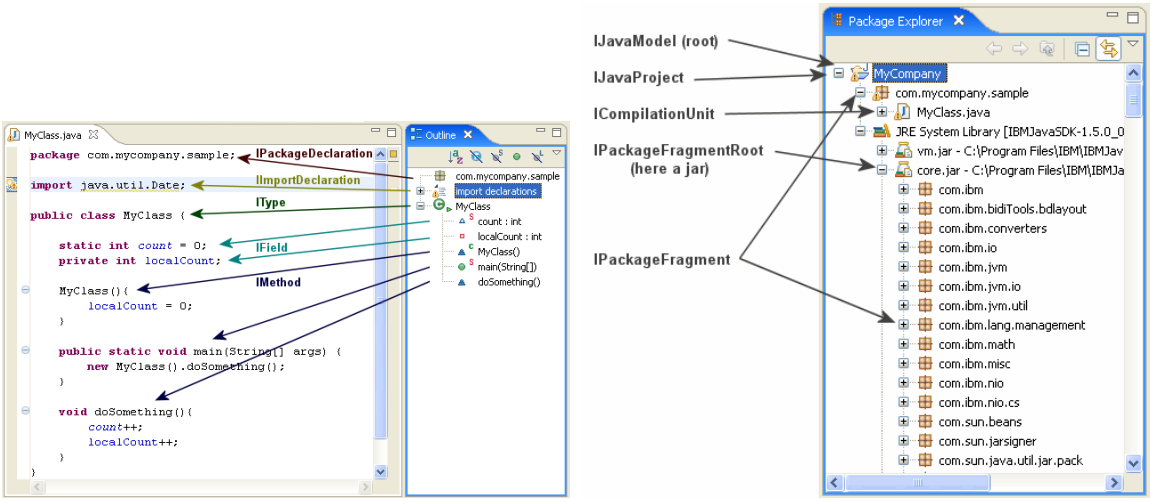
To make the repository model comparable with the state of the workspace the source code model synchronizer generates an EMF model describing the elements in the workspace and keeps it synchronized along every workspace modification. Maintaining an EMF model lets the model queries component dynamically examine the current state of the workspace without accessing the JDT API.

4.6.1 Eclipse Java Model

Eclipse Java Development Tools main objective is to give feature-rich, integrated and extendable environment for editing Java source code. It comes with incremental builder and rich and extensive tooling for Java editing. Contributors can get all the necessary information about the state of the workspace.

The state of the Java projects are exposed through an API called *Java Model*. It is a complex yet intuitive set of Java interfaces which can be traversed easily. Instead explaining what classes and interfaces are present in this API, let's examine which API elements represents which Eclipse- and Java-specific elements. Figure 4.7 (which comes from JDT's documentation) shows these relationships as the main elements of the Java model. Visible: literally every aspect is accessible from the defined projects down to the method level. As the Java Model is hierarchical the implementation simply traverses it to build the java model.

The elements explained above is enough to create a model about the structure of the Java projects. To add dependency information to this model, the JDT's search feature comes into picture. With this the component is able to query the dependencies for every structural element.



(a) *Java Model elements from the source code* **(b)** *Java Model elements from the navigator*

Figure 4.7: *Java Model elements*

4.6.2 The generated EMF model

Figure 4.8: *EMF model storing the structure of the Java projects*

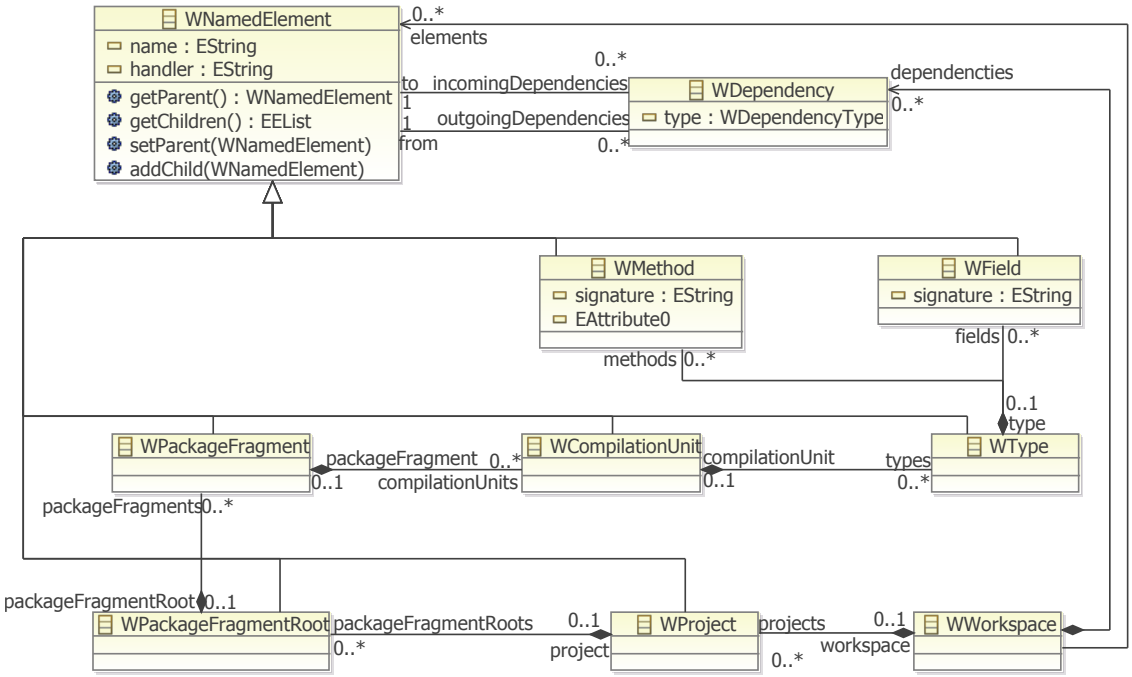


Figure 4.8 shows the EMF model which is extracted and maintained from from the Java Model. It mostly mirrors the structure of the Java model's interface structure. The key element is the `WNamedElement` which is common superclass for all items. The `handler` property holds an identifier string which uniquely identifies the represented Java Model object. To gain access to the correspondent Java Model object, only the `JavaCore.create(String)` is required.

The containment hierarchy starts form the lower right corner of the figure with the

WProject class which represents a Java project loaded into the workspace. It contains WPackageFragmentRoot elements which represent the source folders and the jar files on the classpath and has some Java packages as children(as WPackageFragment instances). Under the Java packages there are the WCompilationUnits which are represent the java files and which define some java types (WType). On the lowest level in the containment hierarchy lie the methods and fields (WMethod and WField instances).

Just like in the repository model here also the dependencies are defined upon the common supertype, but here a WDependency has a two-way relationships for both the source and the target part of the dependency relation. For one WNamedElement the incoming and outgoing dependencies are directly accessible. Again, the same dependency types are defined as before.

The model has one common root, a WWorkspace instance which contains both the structural elements and the dependencies.

4.6.3 The model synchronization process

Figure 4.9: *The maintenance process of the source code model*

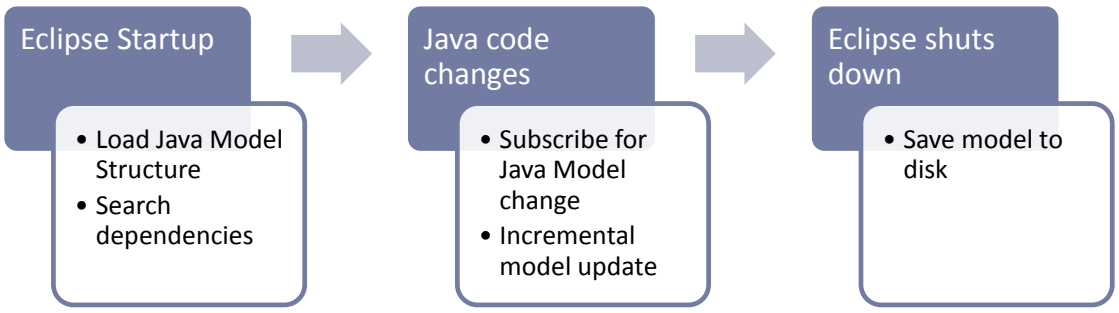


Figure 4.9 shows how the process of model building works. It starts with mapping the entire workspace structure (projects, classes, etc.) into the model. During the process the entire workspace is traversed. If it is done the components executes a dependency search for each relevant element utilizing JDT's integrated search engine. The found dependencies are associated with the proper model element.

When the initial model build is finished the next task is to subscribe for the workspace changes. This can be done by registering a simple listener object with the JavaCore.addElementChangedLi static method. With this every event related to the java editing is available. The component only handles the events where the source files are changed; dealing with working copy events would be too frequent and would contain some incoherent data about the workspace.

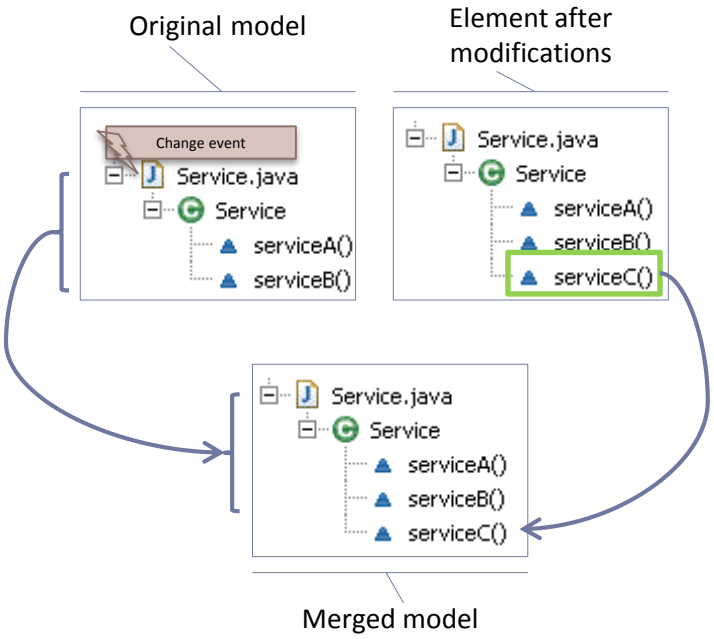
After the component is set up it waits for the modifications. If it happens, an incremental model update fires, maps the related Java Model element to the EMF model and modifies it properly.

The change notifications comes in a form of a hierarchical delta holder object. It refers to an object, states if it was added, deleted, or changed and provides the list of the affected children. For example if a Java class definition has been deleted, the generated event contains a changed project down to the container changed compilation unit which has a child delta pointing to the deleted class.

The use-case when an element was added or deleted the model update is straightforward. The event contains the reference of the modified item. It is identified via the `handler` property. The result of the event is a creation of a new element or removing an existing one along with all his children and their dependencies.

If a modification happens in the source file it is also incrementally modified but treated slightly differently. This is necessary because the delta information doesn't provide usable information about what has changed below the compilation unit. This process is shown

Figure 4.10: *Updating EMF workspace model when source code is modified*



in Figure 4.10. The updater checks which source file is modified and re-generates its EMF representation. The generated small model is compared with the source of the model. If a new method is added than it is merged into the old model. If an element is missing, it gets deleted from the original model too. Dependencies are the same: any differences are immediately pushed back to the original model. With this approach the EMF representation of the workspace stays consistent and only local modifications happen on the elements.

When Eclipse closes, the model is stored on the disk in order not to regenerate the entire model when on the next start-up. If no external modifications happen than the model is loaded one-by-one which is faster than gathering all structural and dependency information.

4.7 Model queries

The model queries components executes complex queries on the loaded models and gives and exposes the result for visualization.

Both the result of the explicit queries and the EMF-IncQuery-based use-case is handled inside this component. As it was discussed previously not much has happening here with the result of the explicit queries: it is passed to the UI components as is.

The more interesting part of the module is the queries based on EMF-IncQuery. The main reason why it was implemented is that normally, if the analysis is done only on the server side, than all the dependency information will be based on the released source code. So if a developer changes something and after tries to figure out what are the dependencies of the changes elements, he won't get any results. This is why we want to compare the repository with the current state of the workspace.

Applying this approach will not only give the developers precise information about the impact a possible release would cause but it gives immediate feedback about the change every time the source code is saved. Also important that all results of all queries is available upon usage, no manual query is required.

To achieve this, the component first loads the model of the repository and the model of the current workspace, loads them into the EMF-IncQuery engine, and initiates the queries and exposes the results on an API which also updates the dependency information dynamically.

The queries are developed with EMF-IncQuery's own query language. This language is used to express certain patterns over EMF models. You can think about it as a regular expression over an object-model. To highlight this, let's see an example of a pattern expressed with this language which matches on all Java files and all defined types in it from the workspace model (see Figure 4.8).

```
package example

import "http://inf.mit.bme.hu/donat/incquery-deps/wsmodel"

pattern typesInJavaFiles(cu : WCompilationUnit, t : WType) = {
        WCompilationUnit(cu);
        WType(t);
        WCompilationUnit.types(cu, t);
}
```

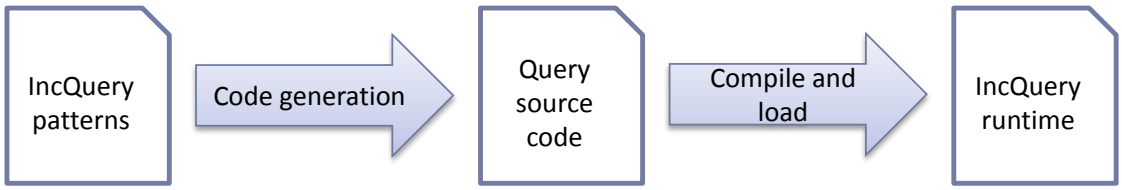
The import declaration loads the schema of the EMF moel. The pattern signature contains a name and a parameter list. Here the parameters define the result as the found/-matched items from the input model.

The body of the pattern starts with two type constraints. This can be translated as "match only on the elements which have the `WCompilationUnit` type for the `cu` variable and match the `WType` object for `t` variable". Alone this would list all compilation units

and all types as a result in all possible combinations, just like in SQL when one select data from two different tables at once without joining them. The third statements is the connects the result items together. It can be translated as ”match only the on the objects where the `cu` variable references the `t` as a contained type”. Expression power of the pattern language is much bigger; for complete reference check the documentation site [4].

The usage of the queries takes multiple steps shown on Figure 4.11. First the queries

Figure 4.11: *Development of EMF-IncQuery queries*



are defined with the query language. The tooling takes the queries and automatically generates source code from the queries. With this there is no need for manually using the pattern definition part of the EMF-IncQuery engine. The generated source code along with the engine exposes a simple interface for registering the queries and getting back the results when needed. The only manual coding takes place at the last part on load-time, when it has to be specified manually which queries should be evaluated.

The generated code contains a factory service which registers the query in the engine. The query is represented as a `Matcher` object in the generated code which gives back the results through the `Matcher.getAllMatches()` method. The result is represented by `Match` instances which hold the result and all the necessary information to make it usable.

The following list enumerates all the queries developed for the dependency analysis.

Join queries In order to compare the workspace with the repository it is necessary to find the correspondent element in the two model. The join queries are basic patterns which joins the projects, classes, fields and methods in the models. The result of the queries are all the found elements which are the input for the queries below.

Workspace changes These queries discover the differences between the workspace and the repository. The results are the projects, classes, methods and fields added or removed from the source code.

Incoming dependencies For every source code loaded in the Eclipse these queries shows all elements depending on them. All incoming dependencies for all classes methods and fields are the result of these queries. Also the type of the dependency is available.

Outgoing dependencies The queries in this category can be highlighted with the following: If a dependency exists in the workspace it should exist in the repository too. If

the workspace changes and the dependencies change it can be source of the problem. By checking the changed outgoing dependencies which start form the selected items the developer can analyze it.

Commit impact This list contains queries which are the combination of the workspace changes and incoming dependencies category. They show the incoming dependencies of items which have changed in the workspace. The result is the impact which a possible release would cause. One use-case is to show the incoming dependencies of the deleted elements. Another can be inheritance on the classes where a method is added.

On start-up these queries are registered and accessible outside via a simple API. The UI components use this API to access the dependency information and show the current state in different views.

4.8 UI Components

The UI components extends the Eclipse platform's user interface to provide access to the dependency information and to show the results in an integrated way.

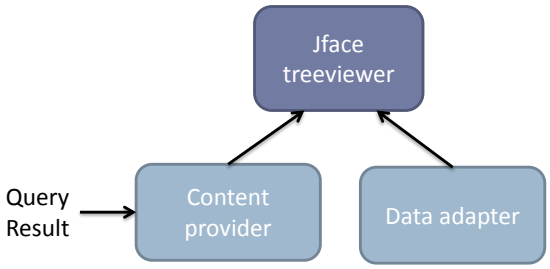
The first part is the UI extension for explicit queries. The Java source code editor gets an new element in the context menu with the label "Show Incoming Dependencies". The developer selects any code element in the source code (he can point to a class, a field or a method) right clicks on it and chooses this menu element. The initiated action first resolves the selected element and tries to get a fully qualified name of it. If it is successful the query is sent to the server side for evaluation. After the result of the query is obtained, the result is pushed into an Eclipse view.

EMF-IncQuery also shows the result data in an Eclipse view but it does not need an explicit call to update the visuals. A change listener registered in order to to update the the result. Every time the source code changes or when a different project or class is in the focus the result is updated accordingly.

The Eclipse views utilize JFace, more precisely JFace tree viewers to present the information. Using JFace has two advantages: displaying the data is not bound to the original structure of the input and EMF has tight integration to JFace out of the box.

Figure 4.12 shows a simplified view how the data is presented in JFace viewers. Originally the results are a simple net of objects. In the first step it is restructured into a into an internal tree structure. This tree is the input content for the display. To make JFace understand the structure of this data, a data adapter is loaded (through Eclipse platform service registration). With both the content and the description how the data should be displayed JFace automatically shows the results and provides convenience functions such as tree folding, selection service, etc.

Figure 4.12: *Displaying data with JFace*



Chapter 5

Evaluation

5.1 Example revisited

After showing the details of the implementation, let's see it in action. We will use the tool to solve the problem described in section 3.2. We will go though on every component and check what is happening there with the example.

Let's start with the repositories. In our example the source and the binary repository contains three projects, each of them holds one package from the example. the name of the projects are service, client and impl, the name of the jars files respectively service.jar, client.jar and impl.jar. For the sake of simplicity we have only one version from each project.

5.1.1 On the server side

Now let's move forward to the server process and the dependency analysis. As the binary projects appear in the repository, they get discovered by the server process.

The first step in the process is the bytecode analysis. In this phase the structure and the reference of the dependencies are discovered from the binaries. As of the structure the analyzer finds 3 projects, 8 classes 2 fields and 19 methods which get stored in the database. The methods are composed of the 12 defined methods, 6 constructor (from all non-interface classes) and 1 static initializer (which comes from the Services class as it gives a default value to a static field).

The discovered dependencies are formatted and filtered in order not to have internal or platform defined elements (dependencies pointing inside the `java.*` package). For example let's see the `BasicImplUtil` class' `registerImplementation()` method. The first one is a `ConstantMethodRef` type reference. During the analysis they are first converted The first one is a class reference the second one is a method reference. The references are formatted to source format as it is shown in the second column. Now the first references is eliminated from the list, because it points inside the `impl` project. However the second

Table 5.1: *External references of the BasicImplUtil.registerImplementation() method*

Binary format	Source format	Type
impl/BasicProvider	impl.BasicProvider	ConstClass
service/Services/registerProvider (Ljava/lang/String; Lservice/Provider;)V	service.Services.registerProvider (java.lang.String, service.Provider):void	Constant MethodRef

one does not, so it remains at its place.

The dependency processor takes the remaining dependencies from the elements and searches the database if there is one. In case of our method reference, the processor finds that there is a `registerProvider()` method defined with the same exact signature signature in the database. Because the reference was stored on the called method list, the processor creates a new method call dependency starting from `impl.BasicImplUtil` targeting `Services.registerProvider()`.

At this point the storage engine stores all the structure and the dependencies of the three products. The schema of the Oracle database implementation and the EMF database was described in details previously. Now let's check how this data can be used on the client side.

5.1.2 On the client side

To continue our example, we will examine the view of the developer who knows only about his own `service` project. If the Dependency Analysis Tool is installed in his Eclipse, the EMF model – shown on Figure 5.1 – is built and synchronized automatically. On the

Figure 5.1: *Workspace model containing Service package from the example*

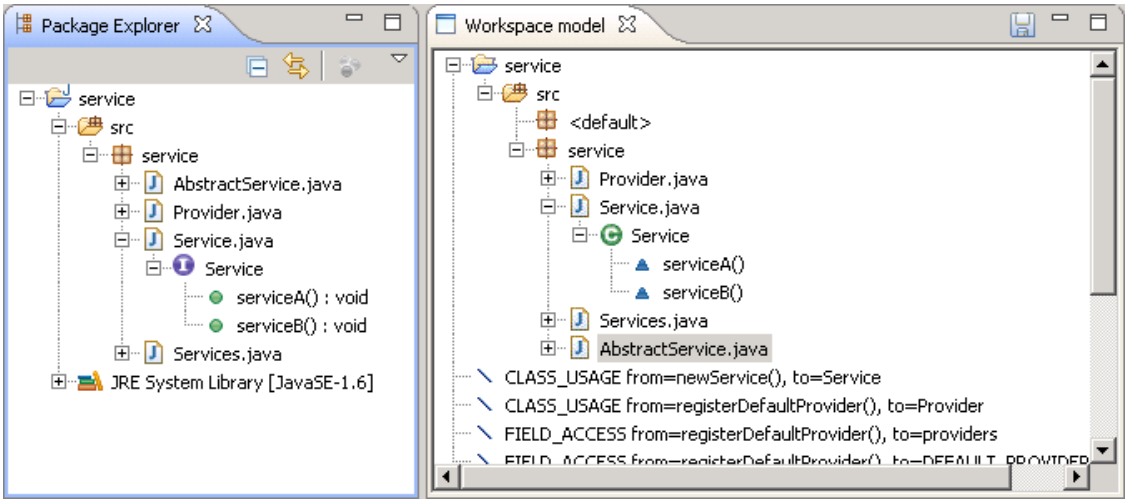
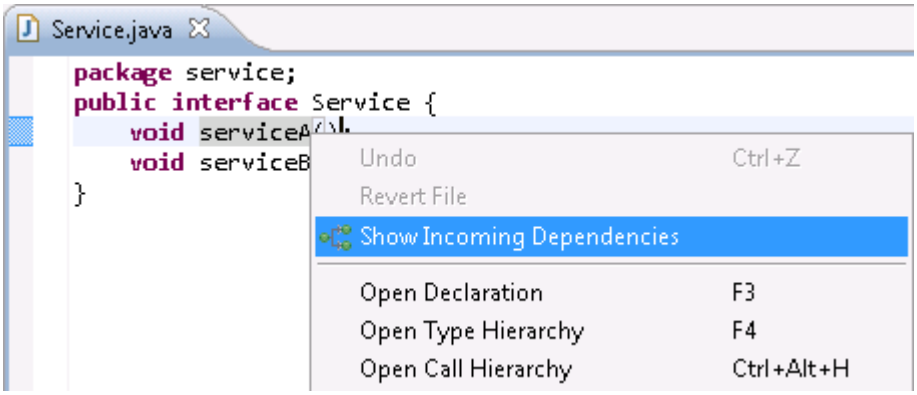


figure we can see, that the structure of the project (left) corresponds to the content of the EMF model (right). In the model view, you can also see that the in-project dependency relationships are also registered.

Because the `client` and the `impl` projects have references to the `service` project, both of them are loaded to the repository model. It has all the element but with the compacted version, so no fully qualified names. But because there is no overlapping names between the classes and method than the compacted model does not introduce false-positive results.

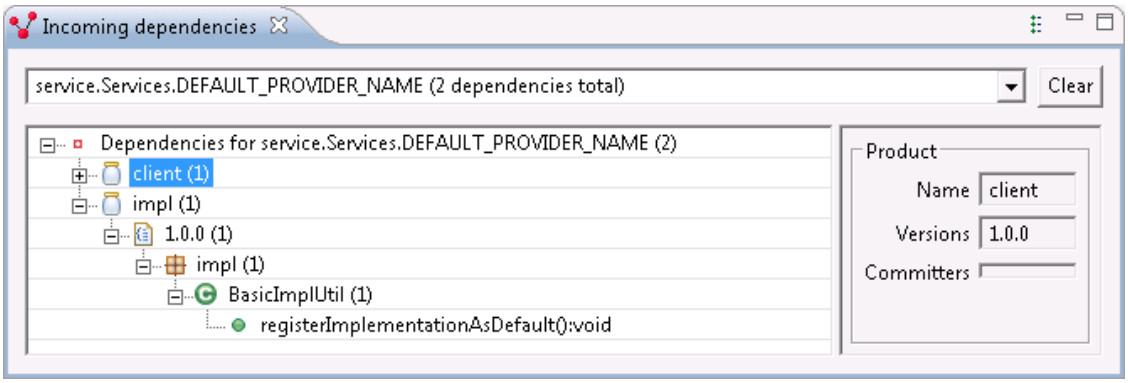
Let's see how the developer checks the incoming dependencies through the explicit queries. Figure 5.2 shows the context menu contribution for initiating an explicit query.

Figure 5.2: *Execute explicit queries from the source code editor*



The query takes a few seconds to respond. The results are shown in the viewer like on Figure 5.3.

Figure 5.3: *Result of explicit queries*



The same result can be retrieved using model queries, but without any interaction: all result for the selected project is automatically showed. Plus, if something is changed, for example the `serviceB()` method is deleted, it i) is compared against the repository model and marked as deleted method and all the incoming dependencies is listed as a possible impact if the modified code gets committed.

put image of the result of the model queries

Back to example. The developer checks the incoming dependencies of the `Services.DEFAULT_PROVIDER_NAME` field and the `Service.serviceA()` method. In the first case there are some incoming dependencies which can be examined. In our example there are two

external users of the field, so the developer can either contact them and coordinate the smooth upgrades by simultaneously releasing a new versions of all projects. On the other side, the `serviceA()` method has no incoming dependencies, as nobody uses it. In this case the developer has the right to change the signature however he wants.

5.2 Performance

After the functionality was presented, let's check the overall performance of the system. We have to check the individual components that they can complete their requirement as the input or usage scales up to a level of the production applications. First we have to verify whether the dependency fast enough and can process large number of Java binaries. The next one is to see if the explicit queries are reasonably fast. The last yet the most important is to test to performance of the EMF-IncQuery engine how big instance models can be loaded into the EMF-IncQuery engine.

The jar files used as the sample data come from real-life applications; they are operational softwares from the CERN controls systems. By using them we can eliminate the problems of having badly generated homogeneous sample data.

5.2.1 Dependency processing

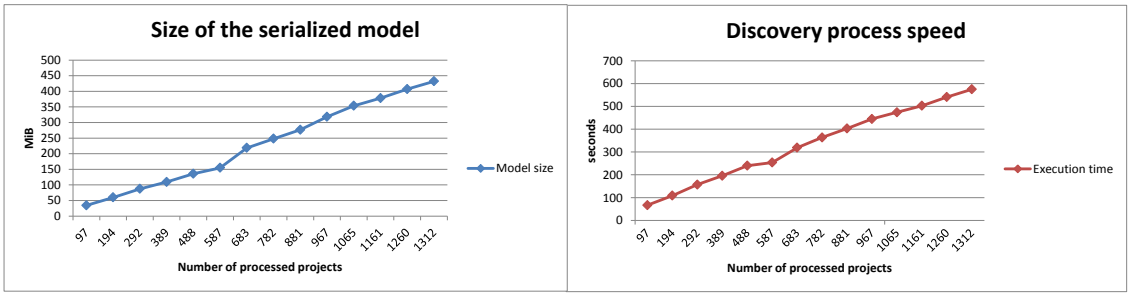
The first element to measure is how the server process perform. A measurement has to check how efficiency the bytecode analysis and the dependency processing together works. Minimal preliminary requirement is that is should be much faster then the source code of the software in the repository changes, because otherwise the dependency database will contain constantly outdated information.

As a test I took all of the import binaries, started the server process as a standalone Java application, initiated the dependency discovery on the subset of the binaries and measured the statistics of the database and the speed of the discovery. After the test has finished I erased the database and repeated the test with a bigger and bigger subset of the jar files until the entire set was analyzed. For the test I used a desktop computer with the following specifications:

- Processor: Intel Core i3-2120, 3,3 GHz,
- RAM: 8GB DDR3,
- OS: Windows 7 Enterprise, 64 bit, Service Pack 1,
- JVM: Oracle JDK 1.6.0_27-b07.

The tests showed that on average one project contains 80 classes, 513 methods, 304 fields and 778 dependencies. The result of the test are shown on Figure 5.4 The left chart shows how big generated EMF instance model depending on the input jar size. It is clearly visible that the size linearly increases as the input grows. Also the execution time (right chart)

Figure 5.4: *Result of dependency processing measurement*



shows steady growth as the input size increases. Looking at the analysis of all (1300) jars, the average processing takes roughly $0,43$ seconds. Knowing that even in large software repositories usually a few, maybe a few dozen releases happen, then having a system which analyses the differences within a second is more than enough.

5.2.2 Explicit queries

Next question is how fast are the explicit queries. We want to see that the results of a dependency query returns the data and visualizes the information within reasonable time even when the result set bigger than usual.

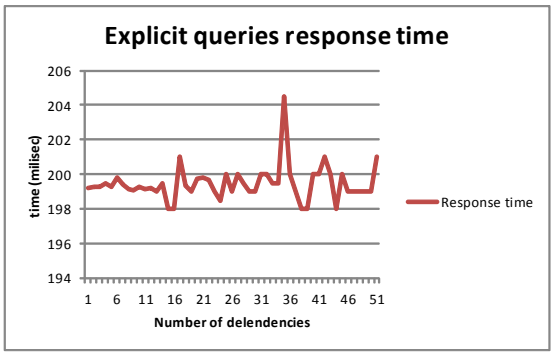
I set up the server process on the same machine, what I used for testing the dependency processing. I filled the database with with all the dependency information. For measuring the speed of the queries used an average virtual machine which from the CERN ecosystem which has similar specification what is usually used for development:

- Processor: Intel Xenon E5645, 2,4 GHz,
- RAM: 4GB DDR3,
- OS: Windows 7 Enterprise, 32 bit, Service Pack 1,
- JVM: Oracle JDK 1.6.0_35-b10.

Because this machine runs on a virtualized infrastructure it can't be considered as a steady platform; performance could change as the usage changes, but it is strong enough for comfortable Java development.

For the tests I chose a specific project which happens to be a widely-used library by the other jars. During the measurement a query is initiated for the incoming dependencies for every single element of this project. I dropped the result where there was no dependencies and made a statistics, how the size of the result set affects on the response time. The results are shown on Figure 5.5. As it turned out, that the response time does not depend on the size of the result, it was returned roughly 200ms on every sample. This result means that utilizing dependency queries return the result in reasonable time.

Figure 5.5: *Explicit queries measurement result*



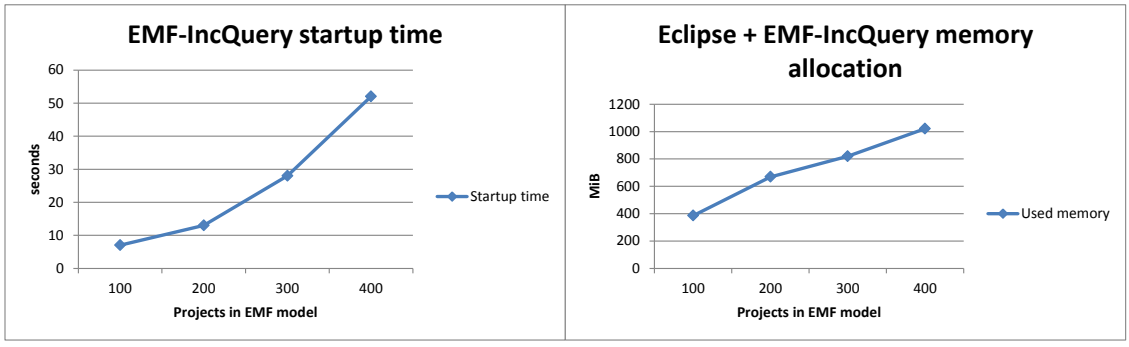
5.2.3 Model queries

Using EMF-IncQuery the question is always about the memory. Does my model fit into the memory? If it does, and the engine is able to build the data structure on top of it then we are in good position because afterward the update mechanism requires minimal resources.

So, to test model queries I reused the sub-models from the previous tests (see subsection 5.2.1), and generated the correspondent compacted models from them. The description of the compacted model is at section 4.5. Normally this a sub-instance model would load automatically from the server side, but now we want to see how big model could be loaded at once to decide the applicability of the component.

The used machine is the same virtual pc introduced in subsection 5.2.2. Because this is a typical development machine, it is also considered as a reference for the usability. In the measurement I opened an Eclipse instance, loaded the patterns and made some modifications in the workspace. I checked how fast the EMF-IncQuery initializes (including loading the model), how much memory the entire Eclipse distribution consumes and afterwards how fast can it react to the workspace modifications.

Figure 5.6: *Model queries measurement result*



The result of the measurement is shown on Figure 5.6. The query engine initialized relatively fast, even the model which holds 400 projects could load within a minute. Considering that this has to be done once per session it is acceptable. Also with the memory

allocation was linear with the model size, it didn't explode, grew linearly with the input size.

No models bigger than 400 elements could fit into the memory (with 500 Eclipse exited with `OutOfMemoryError`, even with the maximum allowed 1.5GiB heap size) because of the 32 bit platform limitation. Applying 64 bit systems could extend the range, but we have calibrate the solution to the target machines which happens in this case.

Nevertheless, the target machine could load and hold an instance mode holding structural and dependency information for 400 projects. It means that the number of projects depending on the ones loaded into Eclipse lower than this number, than the tool can work effectively. In the sample that is the case, the most frequently used project has less than a hundred project-level users. For other software repositories it is presumably true as there should be no ”super-projects” which are referenced from all other projects in the repository.

In conclusion we saw that all components have the desired effectiveness factor. The system itself is usable, scalable with the input and a feasible solution for achieving smooth upgrades.

Chapter 6

Conclusion and Future work

The Dependency Analysis Tool is now a complete solution both with the explicit queries and its EMF-IncQuery-based extension. Without the extension the tool is used in production at CERN and it is fairly possible that extension will become the part of that tool.

The tool can be useful for a Java library developer who works at a large, software-oriented organization where lots of inter-depending softwares are developed and maintained and where binary compatibility and smooth upgrades are mission-critical requirements.

Both parts proved useful, fast and scalable enough for adaptation. EMF-IncQuery showed that it can be applied to a specific domain and can be a useful core for fast model queries.

As of the future plans, there are plenty of directions to go. First it is interesting what kind of dependencies can be extracted. It is possible to extract more detailed dependencies from the binaries by constructing symbolic execution traces based on the bytecode. With this, a variety of new dependencies can be discovered. The second plan can be an extension for analysing not just Java, but C++ softwares too. The third plan is to reuse the model gathered from the project and from the workspace and then to define certain metrics over the models. With this a source code could be checked against certain requirements about the code quality.

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] Jboss tattletale website. http://www.jboss.org/tattletale, October 2012.

[7] Jdepend website. http://clarkware.com/software/JDepend.html, October 2012.

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

[9] Petter Enes. Build and release management, supporting development of accelerator control software at cern. Master's thesis, Norwegian University of Science and Technology, 2007.

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

[11] 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 void serviceB() { System.out.println("Overriden 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();
        }
}
```