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

Incremental dependency analysis of a large software infrastructure

SCIENTIFIC STUDENTS' ASSOCIATIONS REPORT

Author
Donát Csikós

Supervisors
Dr. István Ráth, Ákos Horváth

October 26, 2012

Contents

Chapter 1

Introduction

1.1 Motivation

In today's software development practice, large projects that involve many interrelated software components are a commonality. To tackle the complexity of such development processes, a primary aim for both project managers and developers is to reduce inconsistency issues that may arise due to the combination of (i) development by large and distributed teams and (ii) the complex dependency relationships within the system under development.

In a Java context, a typical example of such problems is when a newer version of some component is binary incompatible with others that depend on (link to) it. Even though linking errors can be detected at compile time in theory, in practice this may not be always feasible as the entire source tree may not be available to every developer.

The main challenge addressed in this work is to achieve *smooth upgrades* [**?**]. A smooth upgrade means that after a new version of a software component has been released, all the other software components which depend on it will remain operational.

There are a variety of approaches and tools to support smooth upgrades. For instance, one can enforce development process policies that involve manual synchronization steps that all developers must follow. An alternative approach – perhaps better aligned with agile principles – is to provide automated tools that relieve the developers of the difficulties associated with software infrastructure management.

In this work, we aim to develop such an automated tool based on static *dependency analysis*. In a few words, the goal of the tool is to assist developers in situations such as when e.g. a bug needs to be fixed which requires a change in a library API. In this case, the tool should help the developer in checking which other components are using that API, more specifically which parts of the API are being used, which functions should remain untouched and which can be changed freely. Using information provided by the tool, the developer should be able to resolve such issues on her own, or in the worst case know

exactly which other team members are to be notified and called in for assistance.

1.1.1 Controls Systems at CERN

The context of this work is the author's internship at the Controls Group of CERN, the European Organization for Nuclear Research. CERN runs the the world's biggest particle physics laboratory with the main goal to operate particle accelerators (such as the Large Hadron Collider) and all the necessary infrastructure. At CERN, numerous scientist and engineers are working on the design and the maintenance of the software and hardware equipment 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.

The controls system 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. The primary objective for the Controls Group is to maintain uninterrupted operation without any downtime. During the internship, the author was tasked with designing and implementing a developer tool that should aid the software developers at CERN in minimizing software component upgrade problems.

1.1.2 Requirements specific to the deployment environment

There are certain requirements specific to the software infrastructure at CERN that needed to be taken into consideration while designing the system.

Technological environment The software components to be managed by the new tool are all plain Java programs that do not make use of any framework (such as e.g. OSGi) for dependency and lifecycle management, but rely on basic features of the Java Compiler and the Java Virtual Machine to link to each other.

Complexity of a large software infrastructure In a simple case, dependency analysis could be supported by a development environment such as Eclipse based on integrated source/binary code analysis. However, CERN Controls Systems deals with an infrastructure that consists of several thousands components (JARs) and even more if versioning information is taken into consideration. The resources available for development environments at developer PCs are not enough for supporting standard IDE features over a system of this size.

Granularity of dependencies Existing dedicated dependency analysis tools offer a rather limited feature set for defining dependency relationships. For example JDepend [**?**] and JBoss Tattletale [**?**] does dependency analysis on the class level, but CERN's requirement is to see method-level and fields level dependencies as well.

Source code analysis alone not feasible Due to legacy reasons at CERN, the (up-to-date) source code is not available for some software components. In other cases, the source code is very large in size because it is automatically generated. Therefore, tools that rely solely on source code analysis cannot be used.

1.2 Overview of the proposed solution

After analysing available off-the-shelf solutions, it became clear that the unique requirements called for a custom solution. The result of the work, called Dependency Analysis Tool, implements hybrid dependency analysis based on both source and binary software components, and relies on a client-server architecture where end-user features are specifically aligned with the computing resources of the host environment. The tool uses a state-of-the-art incremental model query evaluation engine called EMF-IncQuery (developed at the Department of Measurement and Information Systems of the Budapest University of Technology and Economics), to provide on-the-fly dependency analysis results integrated into the Eclipse development tool.

1.2.1 Server features

On the server side, the Dependency Analysis Tool provides the following features:

- *Binary dependency discovery based on byte code analysis* ensures that all (including legacy) software components can be taken into consideration for smooth upgrades, at the desired level of granularity.
- A *scalable relational dependency database* backend ensures that the entire, large software infrastructure can be supported.
- A *data access layer based on Java RMI* allows accessing dependency data by Java-based clients in an efficient way.

1.2.2 Client features

On the client side, the Dependency Analysis Tool provides the following features:

- Efficient *source dependency discovery based on incremental AST processing*, supported by the Eclipse Java Development Tools.
- A *model-based dependency representation for both local (source) and remote (binary) dependencies* based on the Eclipse Modeling Framework. The industry stan-

dard modeling format allows the processing of dependency data by third party modeling tools.

- *On-the-fly dependency queries based on the joined local and remote dependency model* supported by EMF-IncQuery. As the dependency queries are evaluated very efficiently, the system is able to provide instantaneous dependency analysis feedback (relevant to both local and server-side information) as the source code is being changed by the developer.
- *User interface integration components for both client-side and server-side queries* that allow ease of use for developers accustomed to the Eclipse environment.

1.3 Structure of the report

The rest of this report is structured as follows. Chapter 2 gives a brief overview of technologies that have been used in the design and implementation. Chapter 3 discusses the role of the tool in development workflow (Section 3.1), provides a high-level overview of the software system architecture (Section 3.2) and introduces a running example that is used in explanations later on (Section 3.3). Chapter 4 elaborates the design and implementation details, by discussing components both on the server side (Section 4.1) and client side (Section 4.2). Chapter 5 presents experimental results on the performance evaluation of the system, and Chapter 6 concludes the report with discussing on future work.

Chapter 2

Related technologies

The current chapter introduces the different technologies used to build the Dependency Analysis Tool. In section 2.1 we give a short overview about the the Java platform and the internals of the Java binaries. In section 2.2 we go through the frameworks which were used to build and implement the server side of the tool. Last, but not least section 2.3 summarizes the technologies used on the client side. At some places, the followings rely on official tool documentation as marked by citations in section titles.

2.1 Java basics

2.1.1 Java byte code specification [?]

The Java 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: *Compilation and execution of Java classes*

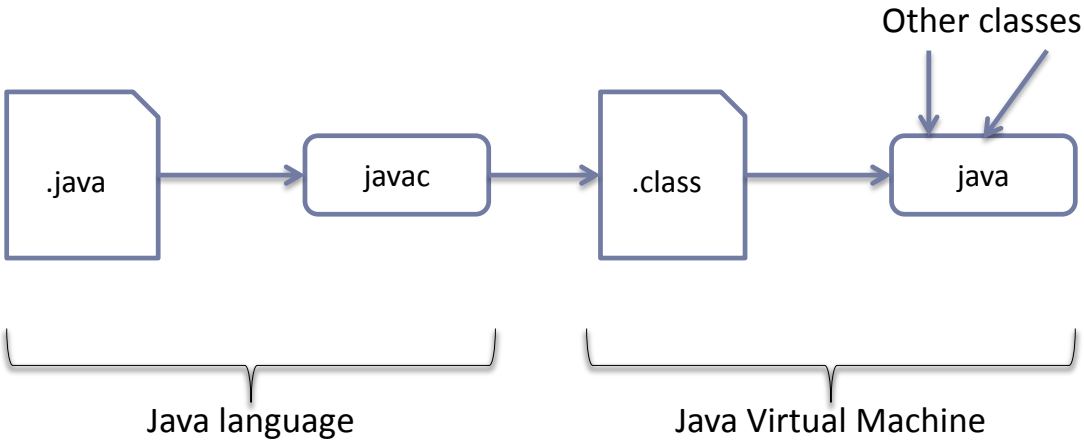


Figure 2.1 illustrates the procedure of compiling and executing a Java class: The source file is compiled into a Java class file, loaded by the byte code interpreter and executed. In order to implement additional features, researchers may want to transform class files (drawn with bold lines) before they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

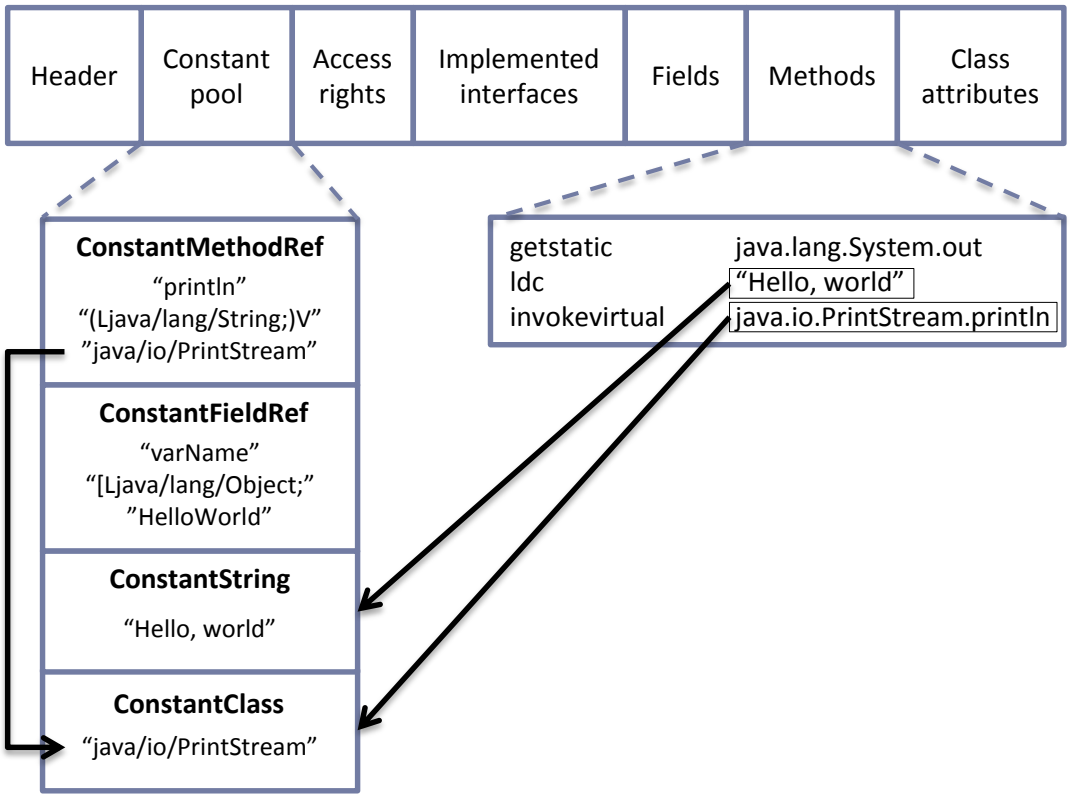
2.1.2 Internals of a Java class file [?]

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 the box on the lower right depicts some instructions that are contained within a method of the

Figure 2.2: *Simplified structure of a class file*



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).

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.

2.2 Server side

2.2.1 Application platform

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.

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.

2.2.2 Build system

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 to achieve various non-standard goals such as integration testing or building non-standalone Java applications.

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.

Apache Commons Byte Code Engineering Library [?]

The Byte Code Engineering Library (BCEL) 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.

2.3 Client side

2.3.1 Eclipse Integrated Development Environment [?]

The Eclipse Project 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.

2.3.2 Eclipse Java Development Tools [?]

The Eclipse Java Development Tools (JDT) project 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.

2.3.3 Eclipse Modeling Framework [?]

The Eclipse Modeling Framework (EMF) 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 provides automated code generation and tooling (e.g. notification, persistence, editor) for Java representation of models. EMF models consist of an (acyclic) containment hierarchy of model elements (*EObjects*) with crossreferences – some of which may only be traversed by programs in one direction (unidirectional references). Additionally, each object has a number of attributes (primitive data values). Models are stored in *EResources* (e.g. files), and interrelated resources are grouped into *EResourceSets*.

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.

Ecore Metamodeling [?]

EMF uses *Ecore* metamodels to describe the abstract syntax of a modeling language. The main elements of Ecore are the following: *EClass* (represented graphically by a rectangle on Figure 4.5), *EAttribute* (entries in the rectangle) and *EReference* (depicted as edges). EClasses define the types of EObjects, enumerating EAttributes to specify attribute types of class instances and EReferences to define association types to other EObjects. Some EReferences additionally imply containment (graphically represented by a diamond). Unidirectional references are represented by arrows. Both ends of an association may have a multiplicity constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. The most typical multiplicity constraints are i) the at-most-one (0..1), and (ii) the arbitrary (denoted by *). Inheritance may be defined between classes (depicted by a hollow arrow), which means that the in-

herited class has all the properties its parent has, and its instances are also instances of the ancestor class, but it may further define some extra features.

EMF-IncQuery [?]

When working with models, it is quite common task to query the model for validating certain properties or searching for interesting parts. EMF-IncQuery [**?**] provides a 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. The current stable version (0.6.8) 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 [**?**] 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 the form of graph 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 contains 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 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.

Chapter 3

Overview of the approach

One of the most effective way of doing smooth upgrades is to discover incoming dependencies. To achieve this, we designed and implemented the Dependency Analysis Tool. The the first part of the current chapter introduces the environment where the tool is deployed and used. The second part gives an overview on the Dependency Analysis Tool approach by describing the objectives of the contained components. In the last part a running example is introduced to make the solution more understandable.

3.1 CERN software infrastructure and processes

3.1.1 Software infrastructure

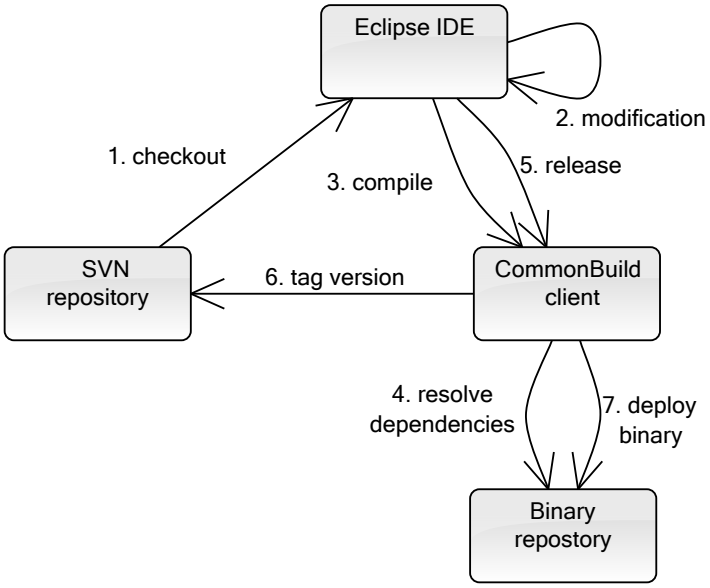
The CERN Controls group has a definite set of tools used for development. Because C/C++ and Java applications are developed, the used Integrated Development Environment is an internally maintained Eclipse distribution with correct plugins installed.

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` [**?**]. 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, as a result 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 only the building but the release of the Java products too.

Figure 3.1: *Build workflow using Common Build*



3.1.2 Development workflow

Figure 3.1 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). Traditionally Common Build is capable to setup build paths and the related options for Eclipse development.

Build process

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 works 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.

Release process

If all source code is ready the developer can hit release. This first initiates the same build process, but it is extended by two things. First the source code is tagged in the SVN repository using the version numbers as tag names, then, the compiled jar file is put into the binary repository.

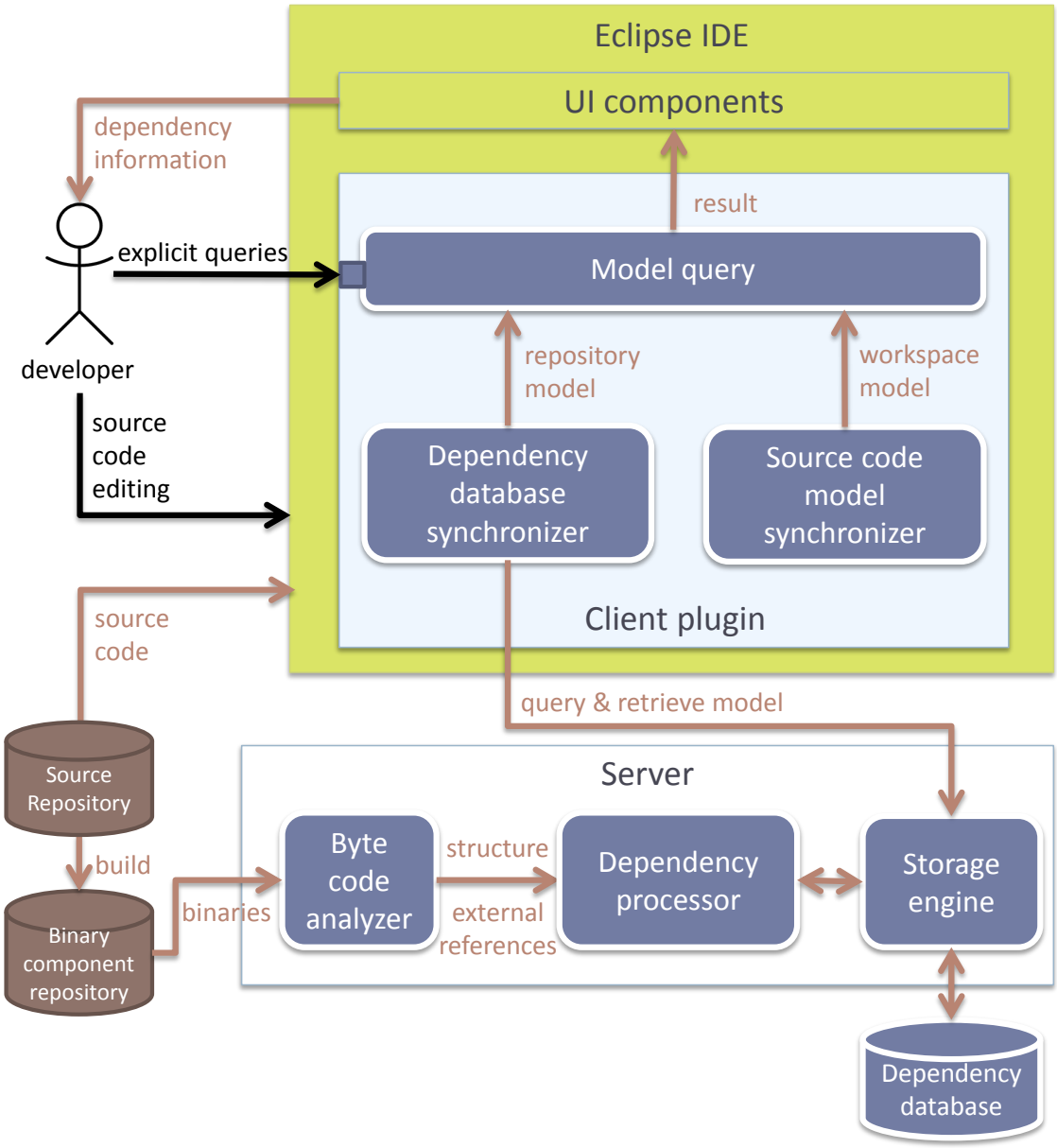
Smooth upgrades and the development workflow

In Figure 3.1, smooth upgrades should be supported during working with the Eclipse IDE ("2. modification"), in the pre-release phase when the developer is making local changes with the aim to ensure that these changes will not break existing software components already released.

3.2 System architecture

Figure 3.2 shows the architecture of the Dependency Analysis Tool. The system ap-

Figure 3.2: *Overview of the implemented system*



plies the standard server-client architecture. The server side discovers the Java binaries,

processes and stores their structure and stores them in a database. The client side is an Eclipse plugin, which gives the ability to the users to obtain and analyse the dependency information about the softwares loaded into the workspace.

3.2.1 System boundaries

The system allows the users to examine the inter-project dependencies from the development environment. It exposes two access points to access this information. First the developer can initiate queries explicitly by selection the target element from he source code editor. The other option to check result of the model queries (described later). This information is updated automatically, every change in the development environment is reflected to the result set. Both versions give visual representation of the result which the developer can analyse and he can use it to make decisions what to and what not to change in the source code.

3.2.2 Repository management

This component consists of the ”Source repository” and the ’Binary component repository’. They contain the source code and the built binary version of the internally developed softwares. 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 approach.

3.2.3 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. It has three elements, the bytecode analyser, the dependency processor, and the storage engine. The components of the server side are explained in detail in section 4.1.

Bytecode analyser

When a new binary jar file is discovered the bytecode analyzer discovers the internal structure and the dependency references. The result of the bytecode analyser is an instance model representing the Java binary structure.

Dependency processing

The dependency processor takes the output of the bytecode analyzer, stores the structure of the Java binaries and tries to resolve the external references. For both storing the structure and resolving the dependencies, this component executes several queries on the dependency database. As a result, all structural information are stored in this database, which serves the client queries too.

3.2.4 The client side

The client side of the solution is a set of Eclipse plugins, which give the developer a convenient way to access to the dependency information.

Model load and direct queries

The base component of the client side 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).

Workspace model creator

The workspace model creator generate an EMF instance model describing the state of the workspace. The generated model contains the loaded projects, the declared packages, classes, methods, etc. This instance 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.

On-the-fly queries

The pattern matcher module executes complex model queries on top of the acquired instance models to provide dependency information. First, it loads an EMF instance model from the server describing the structure of all projects in the central repository. Afterwards the workspace instance model is loaded from the module (see section 3.2.4). 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. The result of the queries are maintained live and automatically as the workspace instance model changes. Because the state of the workspace is followed, more information can be queried than just comparing the source code with the state of the repositories (section 3.2.4): The source code changes and their effects are also queried.

User Interface

Both direct- and on-the-fly queries return their result in Eclipse views. After the result is displayed, it is the user's responsibility to evaluate their results and act accordingly. The user interface consist of the following.

- Menu contributions: The Java source code editor gets a new elements for initiating dependency discovery.
- Result viewers: The result is displayed in Eclipse views.

The details of the user interface are in subsection 4.2.3.

Implementation-specific requirements

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 graph-pattern matching solution is somewhat limited, because in order to make EMF-IncQuery work, the entire repository has to be loaded. By default the repository instance 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.

3.3 Running example: Service Provider Framework

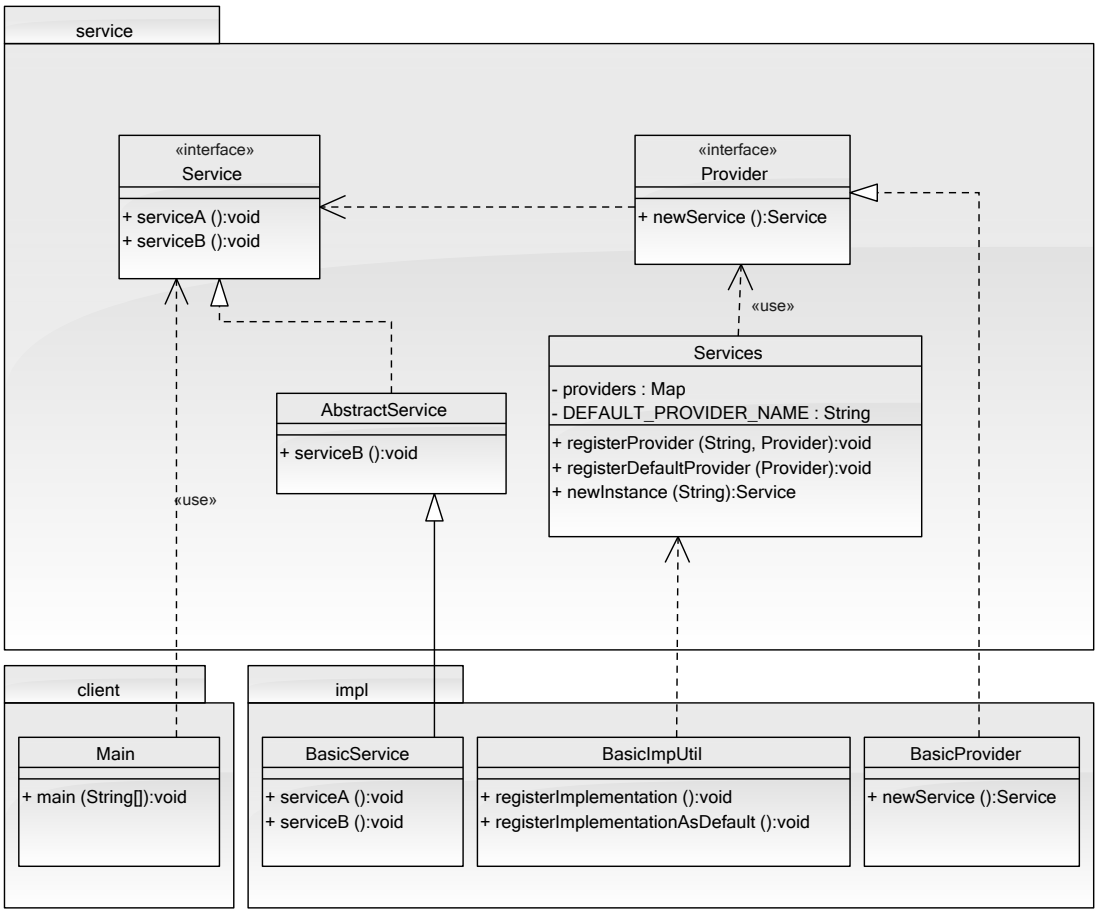
To make the following chapters easier to understand, we are going to introduce a simple use-case example. It is a design pattern called Service Provider Framework. It is a practical application of the original Adapter pattern and it was described in the book *Effective Java* [**?**]. This pattern is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

3.3.1 Description

Figure 3.3 shows the structure of the design pattern. As of the packages it contains 3 major parts. The `service` package contains the core pattern classes. The `client` and the `impl` packages are external users of the pattern and therefore they are considered depending client libraries.

The main goal for the design pattern is to provide a registry of implementations for a desired service. This service is described in the `Service` interface. The `Provider` interface 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 are an identifier string for the registered service and a `Provider` object

Figure 3.3: *Structure of Service Provider Framework design pattern*



which will instantiate the `Service` instances. The clients will instantiate the `Service` instance with the `newInstance()` function. Depending on the passed identifier string, the method will look up whether 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 while 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 simple, the related source code can be found in Appendix A.

3.3.2 Smooth upgrade challenges

As our main goal we would like give the developers the ability to check who is using certain parts of the API. To highlight this, imagine that the three packages described in subsection 3.3.1 are distributed into three different software which have their individual developers. If the developer responsible for the `service` package wants to change some parts of the code without precisely knowing who is using which part of the code, he won't know how many dependant projects will be broken afterwards.

By using Dependency Analysis Tool the developer is able to execute queries on any part of the API. as a result three outcome can happen.

No incoming dependencies If there is no incoming dependency than any modifications can be done without causing compatibility issues. The developer can do whatever he wants.

A few incoming dependencies In this case the developer has to negotiate with the clients how to proceed: either he has to provide backward compatibility or the clients has to adapt their code to the modifications.

A lot of incoming dependencies In this case the queried code element is used by a lot of external clients and any modifications would cause a build error, therefore backward compatibility has to be maintained.

The elaboration of this example is in subsection 4.1.3 and in subsection 4.2.5.

Chapter 4

Elaboration of the design and implementation

4.1 Server side

4.1.1 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.

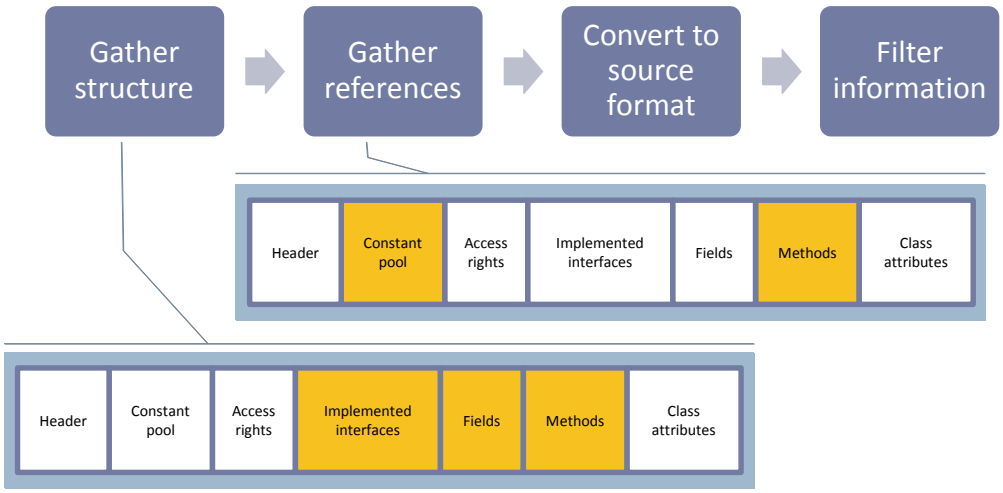
The analysis process

The module extracts all necessary information from the class files by processing them. The analysis is done via Apache BCEL to effectively parse the class file and to obtain the necessary information via simple API calls. Figure 4.1 shows the workflow of the analysis process and the used information from the class files for each step.

Initialization 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.

Gathering external references 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

Figure 4.1: *Steps of the bytecode analysis process*



bytecode and saves all methods which uses them.

Conversion 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`.

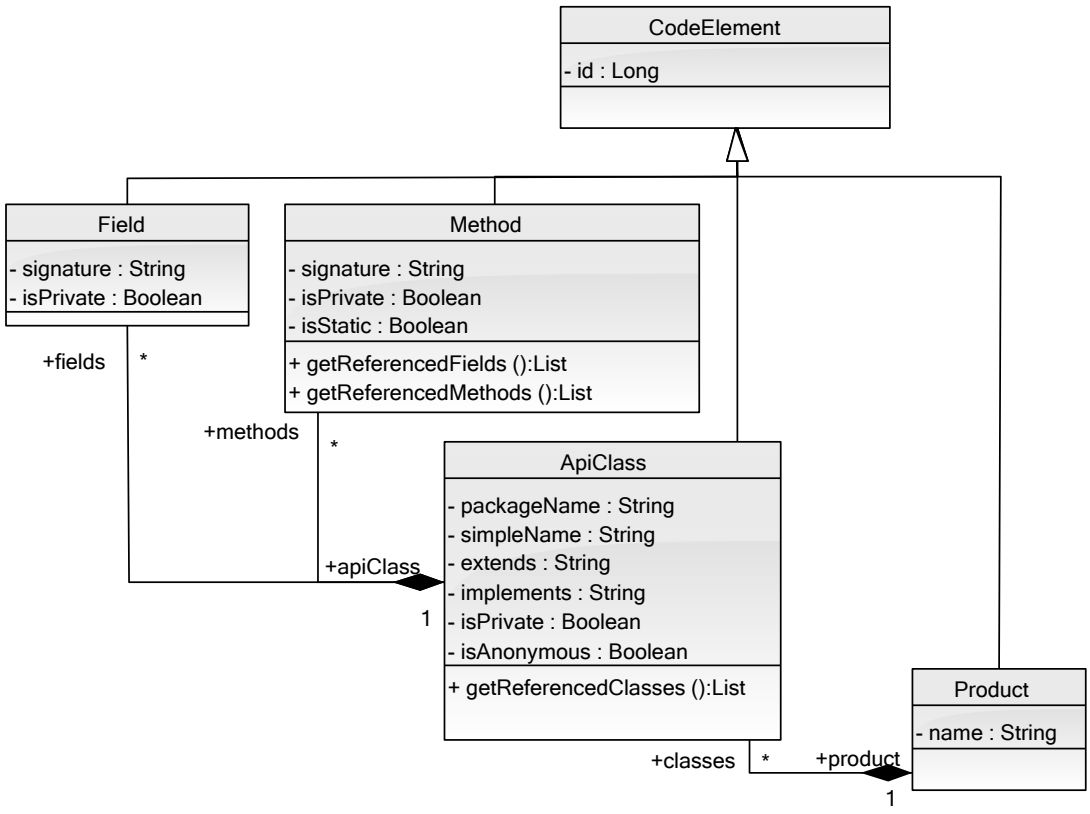
Filtering The final step is to filter the gathered information. At this step we have to consider that if we mapped all information then the gathered data would have a comparable size with the binary repositories which is unacceptable when we have to deal with thousands of jars. To resolve this, the implementation 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.

Extracted domain model

The output of the analyzer is a Java domain metamodel shown on Figure 4.2 . Because this model is used for the dependency processing too the model has additional elements holding the dependency references.

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

Figure 4.2: *Classes of the domain model used by the bytecode analyzer*



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.

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 responsible.

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 two elements and as a result a dependency link is stored.

Discovered dependencies

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.

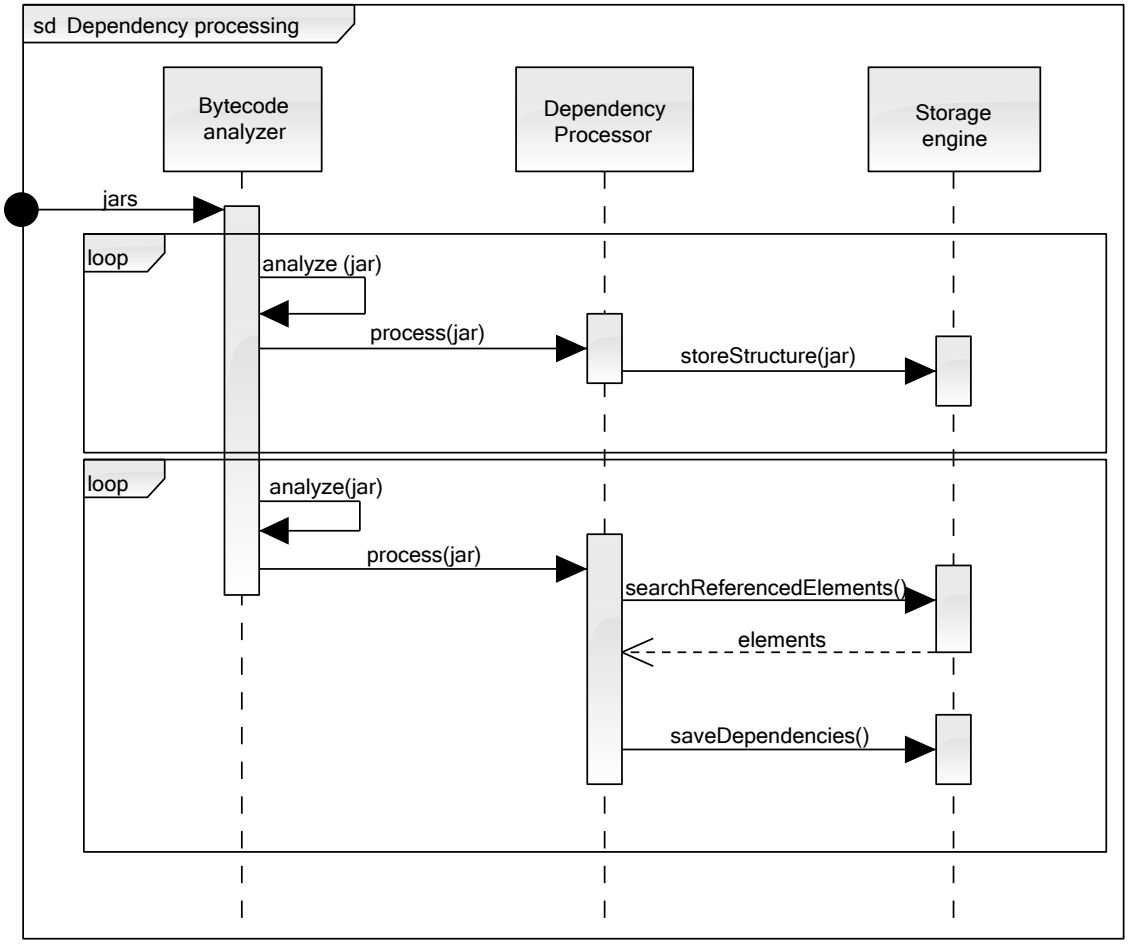
Field access When a method accesses or gives a value of a field defined in an external class.

Two-pass discovery process

The main objective here is to do the analysis on a large set of Java binaries without having memory problems. If one loads all jars to the memory at the same time, it will need an enormous size of memory which will go up if the input number of the binaries increasing.

Binary update processing Figure 4.3 shows how the tool solves this issue. The process starts with an update in the binary repository. The new, not yet analyzed binaries are passed to the bytecode analyzer components which produces a model for each jar files. These models are 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. Additionally only the structure is pushed into the database, the references are considered as transient data. This is necessary because the references may require a large amount of memory: on average they took 60% of the size of a class file and leaving them out significantly reduces the required storage space.

Figure 4.3: *Sequence of the dependency discovery process*



Dependency reprocessing 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 name, for inheritance it searches for superclass in the database and for method override it looks for methods with the same signature in the superclass. For every found element a separate dependency entry is saved at the end of the analysis of the active jar.

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

4.1.2 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.

Interface

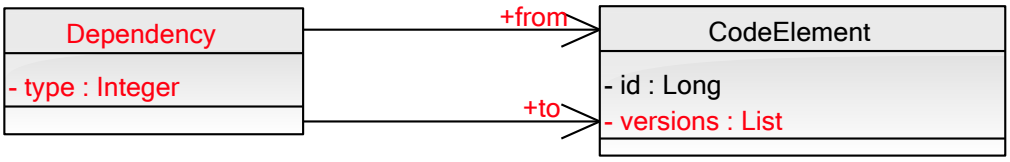
The interface defines operations for two purposes:

1. store and retrieve the structure of a jar and the dependencies,
2. search for items based on their names, and
3. 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 the domain model was extended (see Figure 4.4). Every element has a list of version numbers where they are present. Also a `Dependency` class is added to effectively handle dependencies between them.

Figure 4.4: *Additions to the domain model*



Implementations

This database engine is just a specification how it should work. It has two practical 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 4.2 and Figure 4.4).

4.1.3 Elaboration of the example

After showing the details of the implementation, the current section highlights certain parts through our running example described in section 3.3. We will check how the individual components evaluate the example as an input and how the users can utilize the provided information.

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. This will be the input for the bytecode processing.

Bytecode processing 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).

Dependency extraction 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

Table 4.1: *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

are first converted into source format. 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 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 proces-

sor creates a new method call dependency starting from `impl.BasicImplUtil` targeting `Services.registerProvider()`.

Persistence 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.

4.2 Client side

4.2.1 Dependency database synchronizer

Overview

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

The database synchronizer has two modes of operation: 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 Tool executes 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 an `ApiClass`, a `Method` or a `Field` types from Figure 4.2. 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.

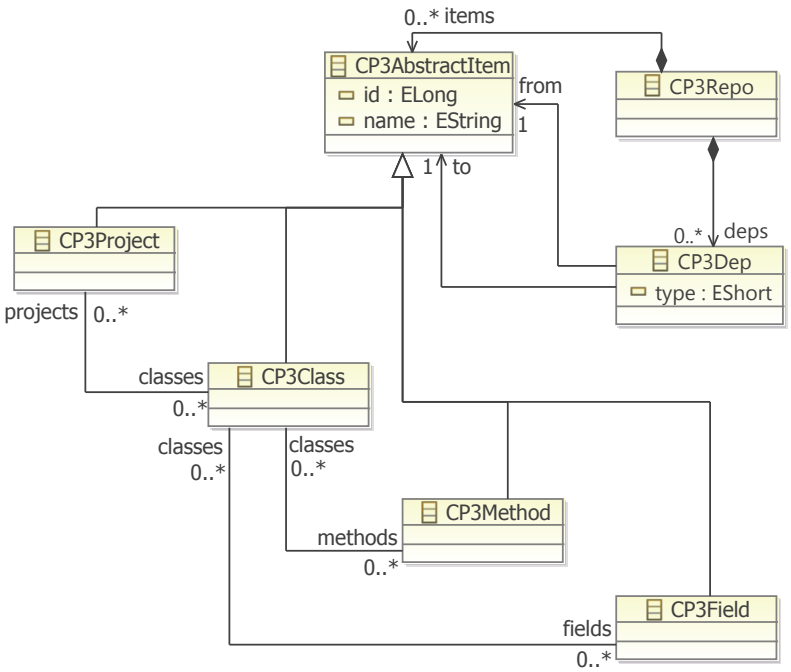
Compacting models to balance resource usage with analysis precision

The resource constraints of virtualized developer PCs at CERN implied that the repository model has to be compacted. 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.

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

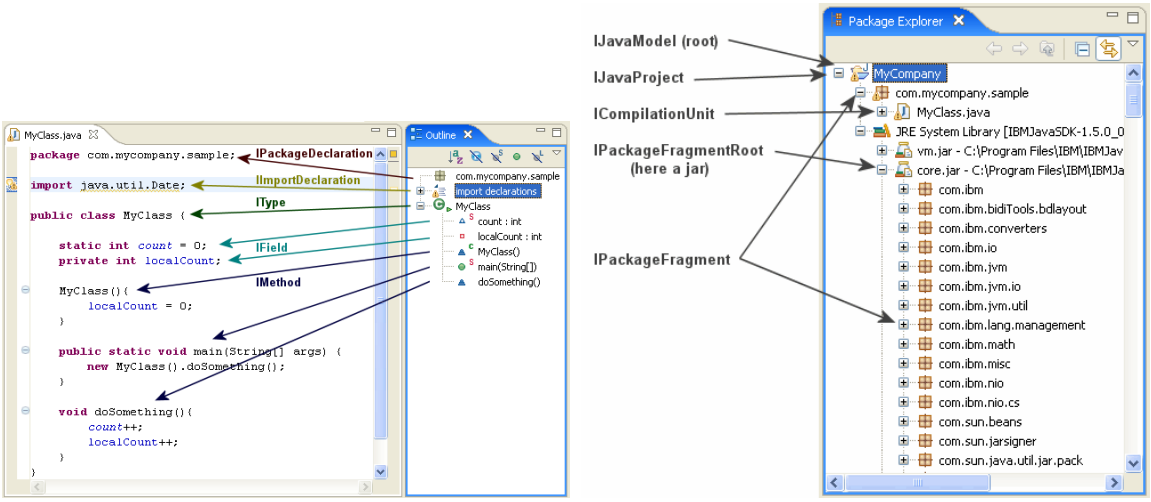
where should be some. While this compacting process results in result over-approximation (potentially false-positive results), but in return lets the model queries component to do fast model queries which are automatically updated every time the source code has changed.

Figure 4.5: *Compacted repository model*



Metamodel for compacted repository models The structure of the compacted EMF metamodel is shown on Figure 4.5. The first thing 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 trimmed 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 exists, then 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. Here the same merging process happens.

Managing repository models on the client side This compacted repository model is loaded at start-up time and used till Eclipse is closed. This is reasonable because the repository changes are considered as rare events: In the environment described in section 3.1 only a few releases happen a day.



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

Figure 4.6: *Java Model elements*

The compact model is small enough to load it into the memory: the serialized representation of the entire CERN software infrastructure is around 70 MiB which is considered small enough to work with.

4.2.2 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.

Eclipse Java Model

The objective of the Eclipse Java Development Tools [**?**] is to provide a 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 the *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 4.6 (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.

Generating EMF models from the Java model

Figure 4.7: *EMF model storing the structure of the Java projects*

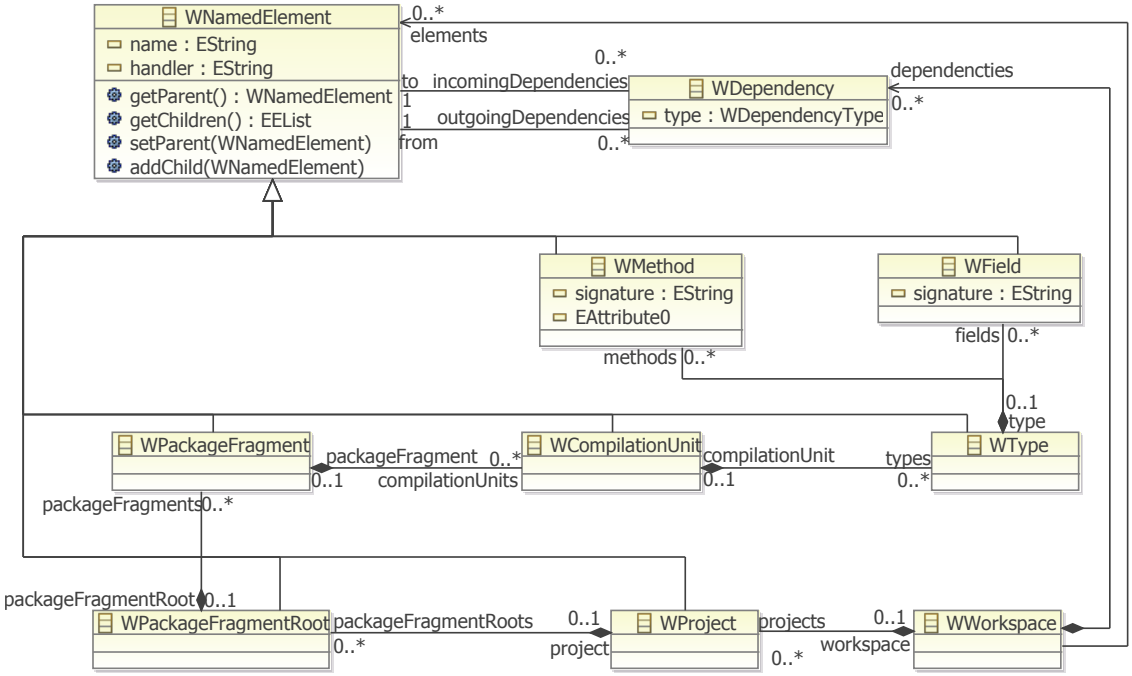


Figure 4.7 shows the EMF model which is extracted and maintained from from the Java Model. It mostly mirrors the structure of the Java model's interface structure. The key element is the `WNamedElement` which is common superclass for all items. The `handler` property holds an identifier string which uniquely identifies the represented Java Model object. To gain access to the correspondent Java Model object, only the `JavaCore.create(String)` is required.

Structural hierarchy 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).

Dependency hierarchy 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.

The model synchronization process

Figure 4.8: *The maintenance process of the source code model*

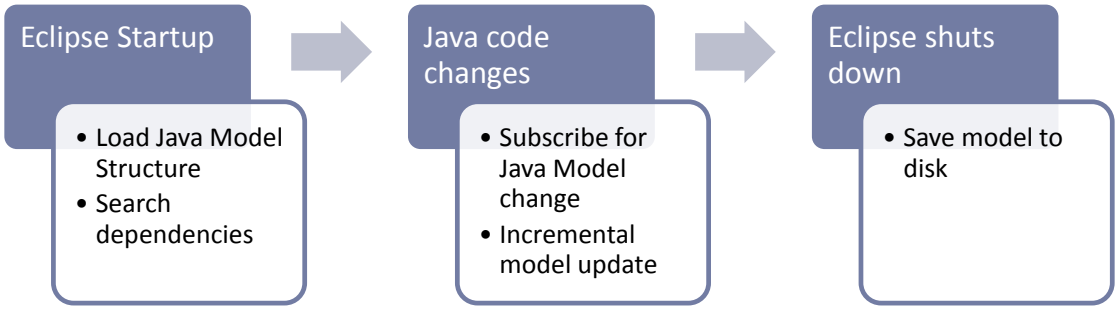


Figure 4.8 shows how the process of model building works. It starts with mapping the entire workspace structure (projects, classes, etc.) into the model. During the process the entire workspace is traversed. If it is done the components executes a dependency search for each relevant element utilizing JDT's integrated search engine. The found dependencies are associated with the proper model element.

Capturing local workspace changes 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.addElementChangedListener(IElementChangedListener)` 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 accordingly.

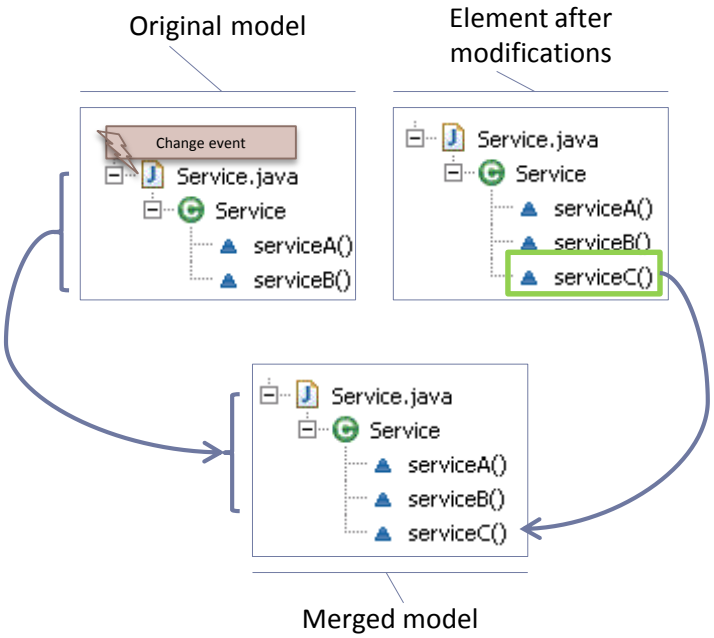
Processing local workspace changes The change notifications comes in a form of a hierarchical delta holder object. It refers to an object, states if it was added, deleted, or

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.

Workspace content modifications 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.

Source file modifications 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.

Figure 4.9: *Updating EMF workspace model when source code is modified*



This process is shown in Figure 4.9. 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.

Generated model lifecycle 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.

4.2.3 Explicit queries

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

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 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 4.10: *Displaying data with JFace*

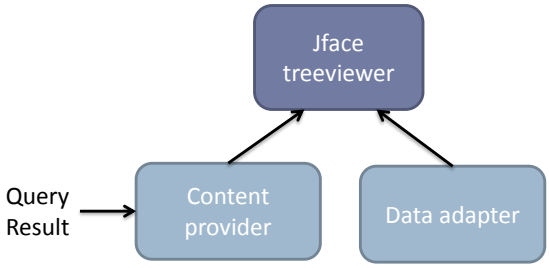
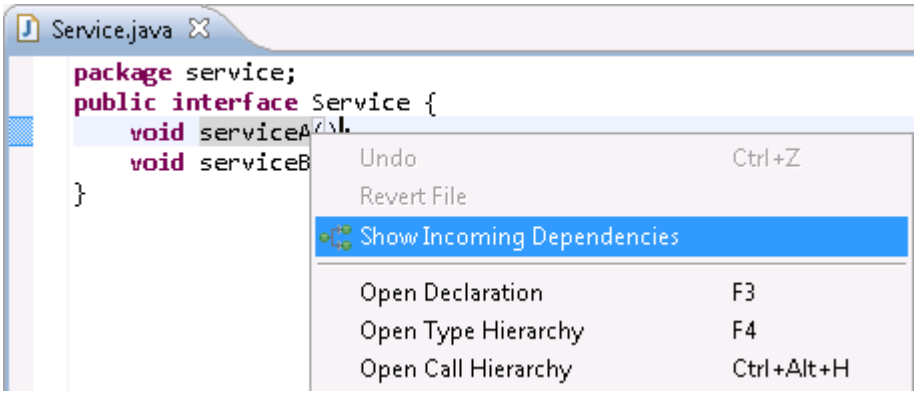


Figure 4.10 shows a simplified view how the data is presented in JFace viewers. Originally the results are a simple net of objects. In the first step it is restructured into a into an internal tree structure. This tree is the input content for the display. To make JFace understand the structure of this 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.

UI integration for explicit query results 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.

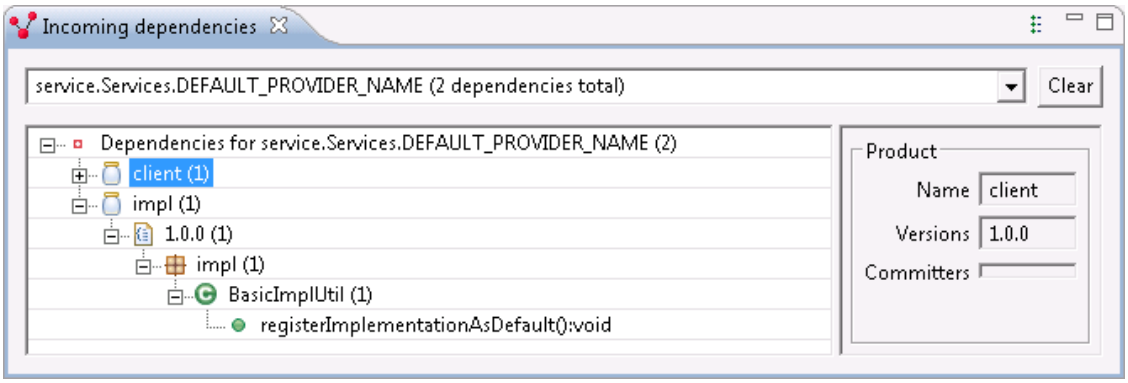
Let's see how the developer checks the incoming dependencies through the explicit queries. Figure 4.11 shows the context menu contribution for initiating an explicit query.

Figure 4.11: *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 4.12.

Figure 4.12: *Result of explicit queries*



4.2.4 On-the-fly model queries

The main functional motivation for implementing client-side queries is to support pre-release comparison between the local source projects in the workspace and the last released state contained in the server-side database. Additionally, the goal is to provide on-the-fly query evaluation so that the feedback from dependency analysis can instantly be displayed to developers as they are making changes to the source code, to speed up the development work.

To achieve this, the system 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.

Query specifications based on graph patterns

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 4.7).

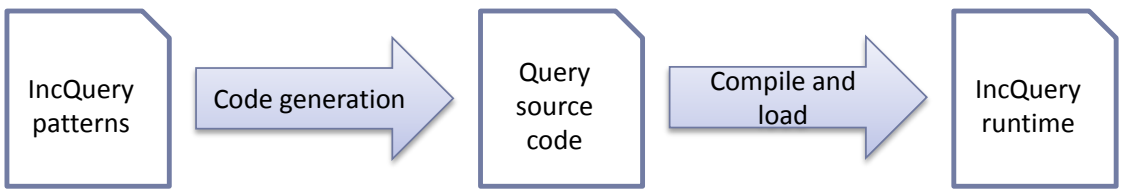
```
1 package example
2
3 import "http://inf.mit.bme.hu/donat/incquery-deps/wsmodel"
4
5 pattern typesInJavaFiles(cu : WCompilationUnit, t : WType) = {
6   WCompilationUnit(cu);
7   WType(t);
8   WCompilationUnit.types(cu, t);
9 }
```

The import declaration loads the EMF metamodel. 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 the `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 refers the `t` as a contained type". Expression power of the pattern language is much bigger; for complete reference check the documentation site [**?**].

Developing model queries The usage of the queries takes multiple steps as shown in Figure 4.13. First, the queries are defined with the query language. The tooling takes

Figure 4.13: *Development of EMF-IncQuery queries*



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.

Integrating queries at runtime 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.

Query definitions

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.

Detailed elaboration One of the implemented queries is the one which retrieves the incoming method call dependencies from the model. The entire definition of this query is the following.

```
1  private pattern joinProjects(repoProject : CP3Project, wsProject : WProject) = {
2    CP3Project.name(repoProject, commonName);
3    WProject.name(wsProject, commonName);
4  }
5
6  private pattern joinMethods(repoMethod : CP3Method, wsMethod : WMethod) = {
7    CP3Project.classes.methods(repoProject, repoMethod);
8    WProject.packageFragmentRoots.packageFragments.compilationUnits.types.
9      methods(wsProject, wsMethod);
10   find joinProjects(repoProject, wsProject);
11   CP3Method.name(repoMethod, commonName);
12   WMethod.name(wsMethod, commonName);
13 }
14
15 pattern incomingMethodCalls(wsTarget : WMethod, repoSource : CP3Method) = {
16   find joinMethods(repoTarget, wsTarget);
17   CP3Dep.from(dependency, repoSource);
18   CP3Dep.to(dependency, repoTarget);
19   CP3Dep.type(dependency, 1);
20 }
```

Listing 4.1: *Elements of the incoming method call dependency query*

The first `joinProjects` query takes the projects from the workspace instance model and connects them to the equivalent element in the compacted repository instance model This join operation is based on the common `name` property.

The `joinMethods` query does a similar thing, but in this case the methods are joined together. The first two instructions gathers the container projects' reference. The third instruction calls the `joinProjects` query, so the results are filtered to only to the elements which exist both in the workspace and in the repository instance model. The methods itself also joined by checking the common `name` property.

The last `incomingMethodCalls` query returns the required results. It lists the methods present in the compacted workspace instance model and its incoming dependency elements in pairs. To find them, first the `joinMethods` query is executed. Then it filters to all the elements which are pointed by a dependency instance. Note that `repoSource` argument is bound to the repository dependency object.

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.

The complete source code for all the queries is available in Appendix B.

4.2.5 Elaboration of the example - continued

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 4.14 – is built and synchronized automatically. On the

Figure 4.14: *Workspace model containing Service package from the example*

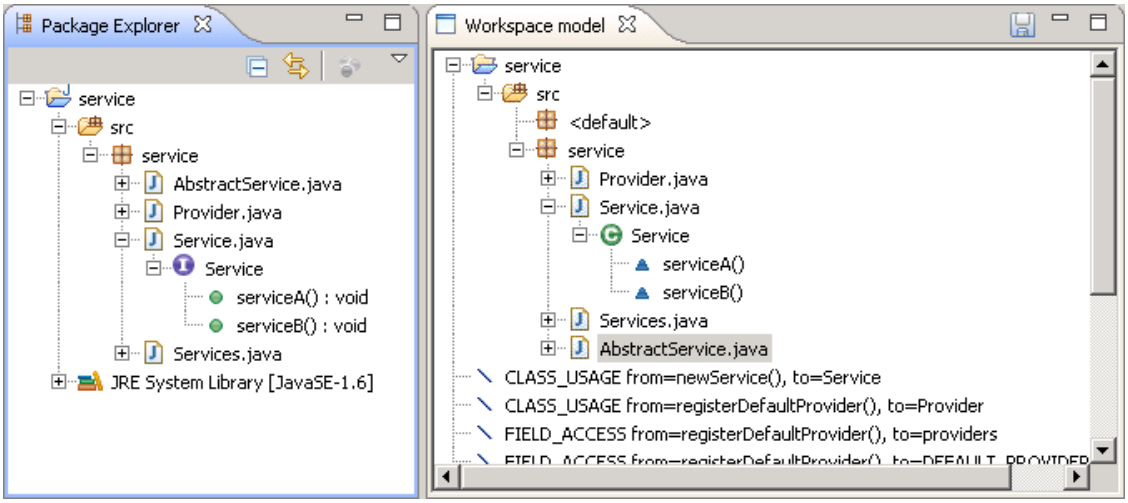


figure we can see, that the structure of the project (left) corresponds to the content of the EMF model (right). In the model view, you can also see that the in-project dependency relationships are also registered.

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.

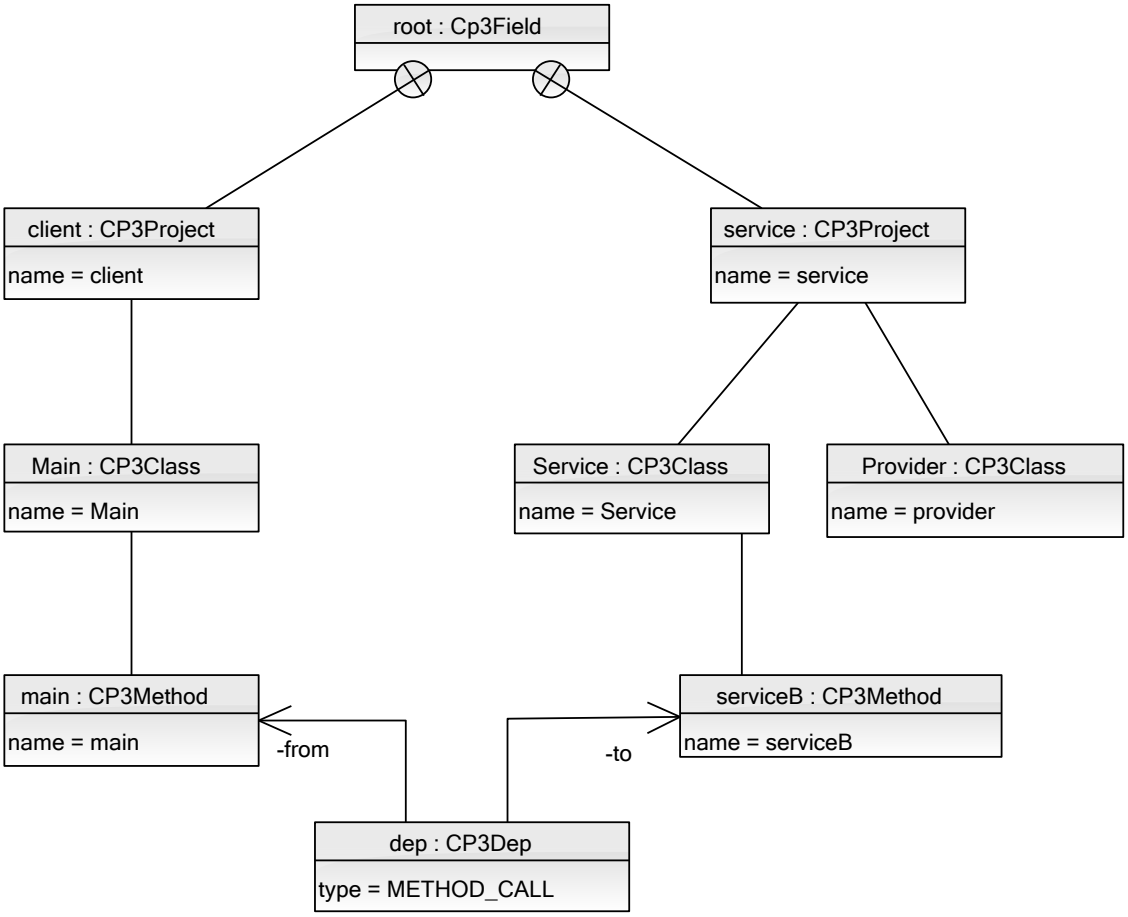
The compacted model

The compacted version of the repository instance model (produced by the dependency database synchronizer) contains all the three projects from the repository. Part of the model is shown on Figure 4.15.

The structure contains the same projects and the same classes as the original model, however the following information is eliminated.

- The class objects has only their simple name left, the fully qualified name is left out. There is no overlapping classes but they would have merged together if there was any. Here the `client.Service` instance has only the name `Service`.
- The method signature was trimmed down int their names. All methods have only their name section of the signature, the parameter list, and the return value are dumped. The object representing the `service.Service.serviceB()` method contains only the `serviceB` formation and has the reference on its container.

Figure 4.15: *Elements of the compacted instance model*



- The only visible dependency on the figure points to the same elements as before. It defines a method call dependency which starts from the object representing the `client.Main.main()` method.

On-the-fly model query results

After the compacted model and the workspace model is loaded the model and the EMF-IncQuery is initialized, the results of the pattern queries can retrieved any time. The pattern matching strategy for the query described in Listing 4.1 is shown on Figure 4.16.

The pattern finds, that the `serviceB` method object from the workspace has a counterpart in the repository: They have the same name and their container project have also the same name (the project object are not shown). Then the query finds, that there is a dependency which point on the `serviceB` object at the repository as the target of a method call dependency. Because these requirements are satisfied, The source of the dependency (the `main` method) from the repository model and the `serviceB` method object from the workspace are returned (elements with the red border).

The query was evaluated with the sample data in EMF-IncQuery's UI tooling. The results are on Figure 4.17. This figure shows that the evaluation gave back the same exact

Figure 4.16: *Pattern matching on the example*

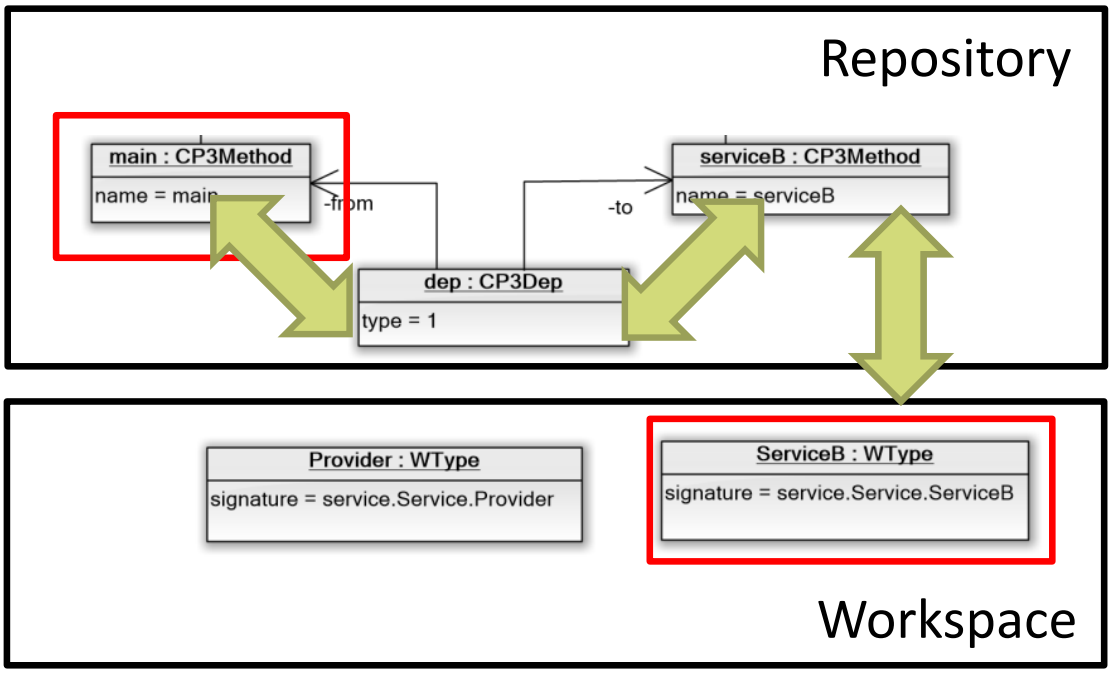
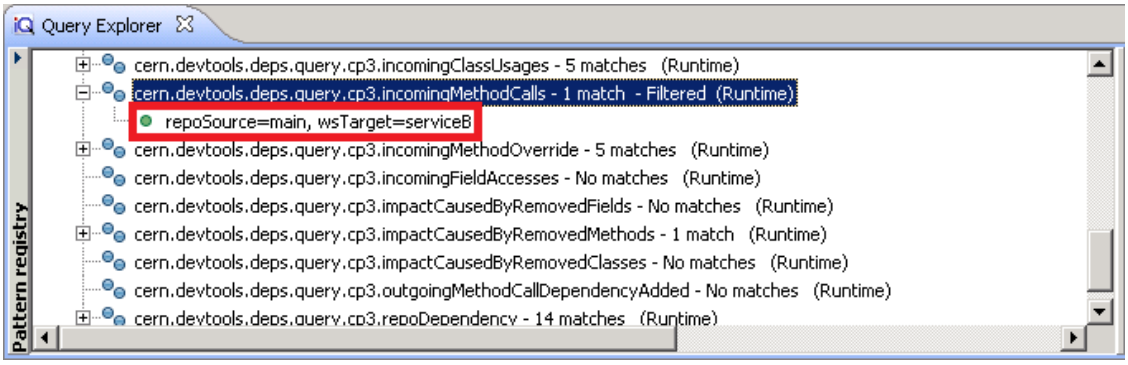


Figure 4.17: *Result of the incoming method call dependency query*



results as it was described on Figure 4.16.

In this example a dependency was found. In this case the developer has to decide whether he will maintain backward compatibility or he notifies the maintainer of the `client` project that the API will break soon. If he tried to modify the `serviceA()` method, we would have seen that there are no incoming dependencies. It means that the developer can break the signature of the method as he wishes.

Chapter 5

Performance evaluation

After the functionality was presented, let's check the overall performance of the system. We have to check the individual components that they fulfill their requirement as the input or usage scales up to a level of the production applications. First we have to verify whether the dependency analysis is fast enough and can process large number of Java binaries. The next task 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 by checking how big instance models can be loaded.

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

5.1 Dependency processing

The first task to measure is how the server process performs. A measurement has to check how efficiency the bytecode analysis and the dependency processing together works. A minimal preliminary requirement is that it should be much faster than the pace at which the source code in the repository changes, because otherwise the dependency database will contain constantly outdated information.

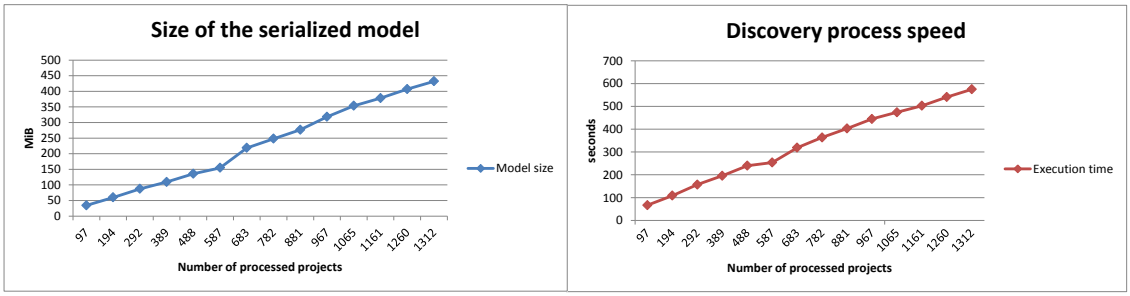
As a test we took all of the import binaries, started the server process as a standalone Java application, initiated the dependency discovery on the subset of the binaries and measured the statistics of the database and the speed of the discovery. After the test has finished, we erased the database and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

- CPU: 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 5.1 The left chart shows

Figure 5.1: *Result of dependency processing measurement*



how big the generated EMF instance model becomes, 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) 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 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 is bigger than usual.

We set up the server process on the same computer that we used for testing the dependency processing. We filled the database with with all the dependency information. For measuring the speed of the queries, we used an typical developer virtual machine from the CERN ecosystem:

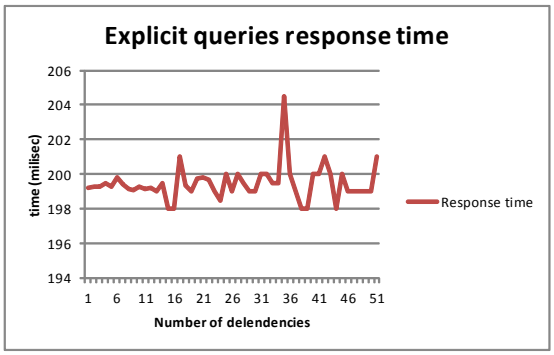
- CPU: Intel Xeon E5645, 2.4 GHz,
- RAM: 4GB,
- OS: Windows 7 Enterprise, 32 bit, Service Pack 1,
- JVM: Oracle JDK 1.6.0_35-b10.

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 resourceful enough for comfortable Java development.

For the tests we 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. We dropped the result where there were no

dependencies and accumulated the results to show how the size of the result set affects on the response time. The results are shown on Figure 5.2. As it turned out, the response

Figure 5.2: *Explicit queries measurement result*



time does not depend on the size of the result, as the result was returned in about 200ms on every sample.

It is important to note that the measurement does not include the network delay. Also the returned returned results contain dependency information *for one single element*. Overall, however, this is considered an acceptable response time for retrieving (individual) dependency information.

5.3 Model queries

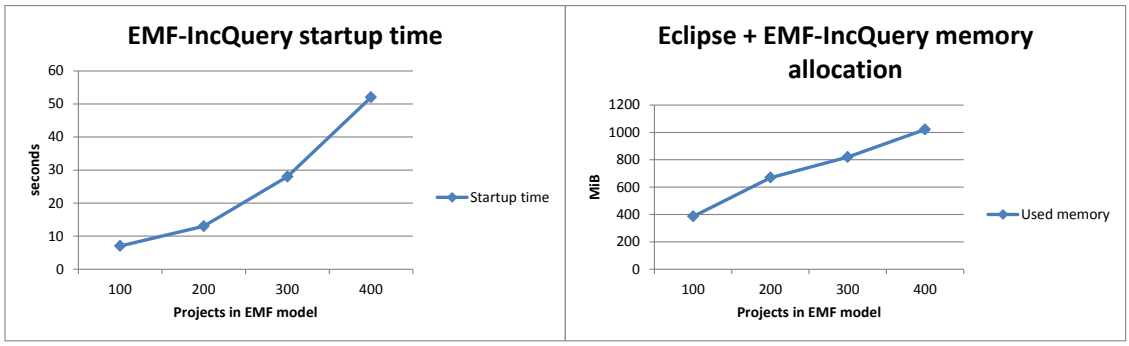
Using EMF-IncQuery the question is always about the memory usage. If the model and the query caches fit into RAM, then the update mechanism requires minimal resources to keep query results consistent with the (changing) model.

So, to test model queries we reused the models from the previous tests (see section 5.1), and generated the corresponding compacted models from them. The description of the compacted model is at subsection 4.2.1. Normally this model would load automatically from the server side, but now we want to see how large models could be loaded at once to judge the practical feasibility of our approach.

The test environment is the same virtual PC introduced in section 5.2. Because this is a typical developer machine, it is also considered as a reference for the usability. In the measurement we opened an Eclipse instance, loaded the patterns and made some modifications in the workspace. We checked how fast the EMF-IncQuery initializes (including loading the model), how much memory the entire Eclipse instance consumes and afterwards how fast can it react to the workspace modifications.

The result of the measurement is shown on Figure 5.3. The query engine initialized relatively fast, even the model which holds 400 projects could load within a minute. Considering that it has to be done once per working session it is acceptable. Also with the

Figure 5.3: *Model queries measurement result*



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 because of the 32 bit platform limitation (64 bit systems could, naturally, extend this range). The response times for query updates after model modifications were always instantaneous (regardless of query or model size), which is consistent with literature results reported in [**?**].

5.4 Performance evaluation conclusions

In summary, the performance evaluation of the system has concluded with the following findings:

On the server side the system provides fast analysis, persistence and queries for the global (binary) network of dependencies with full granularity.

On the client side the system is able to support on-the-fly dependency analysis, with full granularity on the local (source) scope and limited granularity on the union of the local and global scopes (which is an acceptable tradeoff for performance and resource utilization).

It is also important to note that the 400 project workload is an over-approximation of what is actually needed in practice. The reason for this is that in each session, a developer actually only needs a small subset of the 400 projects to work with as the high-level dependency network between CERN projects is relatively sparse (this assumption presumably holds for other typical organizations as well). Overall, all requirements could be fulfilled deeming the approach suitable for deployment.

Chapter 6

Conclusions and future work

6.1 Results of the report

In this paper we proposed an effective dependency analysis approach for supporting smooth upgrades in large Java software infrastructure consisting of tens of thousands of classes. It is based on a two-tiered approach, where the server side is responsible for accumulating, storing and processing the incoming dependency relations between the different API elements of the complete software infrastructure, while the client side supports the developer by providing instantaneous feedback on the dependency relations between the software modules currently under development and the overall software infrastructure.

As a summary, my results in this work are the following:

- I designed a *client-server based architecture for incoming dependency analysis* of large Java software infrastructures.
- I developed a *binary dependency discovery module* for finding dependencies between API elements within JARs.
- I implemented a *storage system* for the dependency relations and an access layer for querying the dependency information.
- I proposed a *model-based dependency representation* based on EMF for capturing local and remote dependencies.
- I defined an *incremental AST processing module* for discovering source dependencies in the local workspace.
- I implemented an *on-the-fly dependency query engine* using EMF-IncQuery for instantaneous dependency analysis feedback.
- Finally, I evaluated the approach on the *complete Java software infrastructure at CERN* consisting of more than 1300 separate JARs.

Application of the results As a major achievement, the server side modules of the porposed approach are already used in production at CERN and its client side extension is

also being considered to be included in the upcoming release of the Dependency Analysis Tool.

6.2 Future work

As of the future plans, there are plenty of directions to extend the capabilities of the approach:

- In order to provide a better user notification on the client side we plan to create a dedicated UI for highlighting the result of the local dependency analysis. As Eclipse already provides a facility for searching for cross compilation unit references, it is a straightforward task to integrate the model queries there. Additionally, due to the instantaneous evaluation performance, dependency analysis results could be displayed to the user as warnings or even errors (supported by inline markers in Eclipse's source code editors) that appear on-the-fly as the user is making a change that potentially violates smooth upgrade policies.
- The second plan can be an extension for source-analysing not just Java, but C++ programs too. We could make use of the capabilities of the Eclipse CDT to provide AST support.
- As a future improvement, we plan to support the definition of query-based software metrics and use our on-the-fly query evaluation engine to enforce these policies directly on the AST models.
- Finally, an additional step forward would be the use of dynamic dependency analysis techniques such as symbolic execution traces that would provide more detailed dependency information between the methods of the various modules.

Appendix A

Service Provider Framework source

```
1  public interface Service {
2    void serviceA();
3    void serviceB();
4  }
5
6  public abstract class AbstractService implements Service {
7    public void serviceB() {System.out.println("Default serviceB().");}
8  }
9
10 public interface Provider {
11   Service newService();
12 }
13
14 public class Services {
15   private static final Map<String, Provider> providers =
16     new ConcurrentHashMap<String, Provider>();
17   public static final String DEFAULT_PROVIDER_NAME = "<default_provider>";
18
19   private Services() {
20   }
21   public static void registerDefaultProvider(Provider p) {
22     providers.put(DEFAULT_PROVIDER_NAME, p);
23   }
24   public static void registerProvider(String name, Provider p) {
25     providers.put(name, p);
26   }
27   public static Service newInstance(String name) {
28     Provider p = providers.get(name);
29     if (p == null)
30       throw new IllegalArgumentException("Provider does not exist.");
31     return p.newService();
32   }
33 }
```

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

```
1  public class BasicService extends AbstractService {
2    public void serviceA() { System.out.println("Executed basic service"); }
3    public void serviceB() { System.out.println("Overriden basic service"); }
4  }
5
```

```
6  public class BasicProvider implements Provider {
7    public Service newService() { return new BasicService(); }
8  }
9
10 public class BasicImplUtil {
11   public static void registerImplementation() {
12     Services.registerProvider("basic", new BasicProvider());
13   }
14   public static void registerImplementationAsDefault() {
15     Services.registerProvider(Services.DEFAULT_PROVIDER_NAME,
16       new BasicProvider());
17   }
18 }
```

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

```
1  public class Main {
2    public static void main(String[] args) {
3      BasicImplUtil.registerImplementationAsDefault();
4      Service service = Services.newInstance(Services.DEFAULT_PROVIDER_NAME);
5      service.serviceB();
6    }
7  }
```

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

Appendix B

Model queries source code

```
package cern.devtools.deps.query.cp3

import "http://inf.mit.bme.hu/donat/incquery-deps/wsmodel"
import "http://inf.mit.bme.hu/donat/incquery-deps/cp3model"
import "http://www.eclipse.org/emf/2002/Ecore"

//------------------------------------------------------------------------
// Helper patterns for composition
//------------------------------------------------------------------------

private pattern projectsWithSameName(repoProject : CP3Project, wsProject : WProject) = {
  CP3Project.name(repoProject, commonName);
  WProject.name(wsProject, commonName);
}

private pattern methodsWithSameNameAndSameProject(repoMethod : CP3Method,
 wsMethod : WMethod) = {
  // Find the items contained by the same named projects.
  CP3Project.classes.methods(repoProject, repoMethod);
  WProject.packageFragmentRoots.packageFragments
     .compilationUnits.types.methods(wsProject, wsMethod);
  find projectsWithSameName(repoProject, wsProject);

  // Join the methods by signature.
  CP3Method.name(repoMethod, name);
  WMethod.name(wsMethod, name);
}

private pattern typesInWsProject(wsType : WType, wsProject: WProject) = {
  WProject.packageFragmentRoots.packageFragments
     .compilationUnits.types(wsProject, wsType);
}

private pattern methodsInWsType(wsMethod : WMethod, wsType : WType) = {
  WType.methods(wsType, wsMethod);
}

private pattern fieldsInWsType(wsField: WField, wsType : WType) = {
  WType.fields(wsType, wsField);
}

```

```
42 private pattern classNamesInRepoProjects(repoProject, repoClassName) = {
43   CP3Project(repoProject);
44   CP3Project.classes.name(repoProject, repoClassName);
45 }
46
47 private pattern classNamesInWorkspaceProjects(wsProject, simpleName) = {
48   WProject.packageFragmentRoots.packageFragments
49       .compilationUnits.types.name(wsProject, simpleName);
50 }
51
52 private pattern methodNamesInRepoProjects(repoProject, name) = {
53   CP3Project(repoProject);
54   CP3Project.classes.methods.name(repoProject, name);
55 }
56
57 private pattern methodNamesInWorkspaceProjects(wsProject, name) = {
58   WProject(wsProject);
59   WProject.packageFragmentRoots.packageFragments
60      .compilationUnits.types.methods.name(wsProject, name);
61 }
62
63 private pattern fieldNamesInWorkspaceProjects(wsProject, name) = {
64   WProject(wsProject);
65   WProject.packageFragmentRoots.packageFragments
66      .compilationUnits.types.fields.name(wsProject, name);
67 }
68
69 private pattern fieldNamesInRepoProjects(repoProject, name) = {
70   CP3Project(repoProject);
71   CP3Project.classes.fields.name(repoProject, name);
72 }
73
74 private pattern classesWithSameNameAndSameProject(repoClass : CP3Class,
75   wsClass : WType) = {
76   CP3Project.classes(repoProject, repoClass);
77   WProject.packageFragmentRoots.packageFragments
78       .compilationUnits.types(wsProject, wsClass);
79   find projectsWithSameName(repoProject, wsProject);
80   CP3Class.name(repoClass, name);
81   WType.name(wsClass, name);
82 }
83
84 private pattern fieldsWithSameNameAndSameProject(repoField : CP3Field,
85   wsField : WField) = {
86   // Find the items contained by the same named projects.
87   CP3Project.classes.fields(repoProject, repoField);
88   WProject.packageFragmentRoots.packageFragments
89       .compilationUnits.types.fields(wsProject, wsField);
90   find projectsWithSameName(repoProject, wsProject);
91
92   // Join the methods by signature.
93   CP3Field.name(repoField, name);
94   WField.name(wsField, name);
95 }
96
97
98 private pattern classesWithSameFQNAndSameProject(repoClass : CP3Class,
99   wsClass : WType) = {
```

```
100   // Pattern applies only classes within a project with the same name.
101   CP3Project.classes(repoProject, repoClass);
102   WProject.packageFragmentRoots.packageFragments
103       .compilationUnits.types(wsProject, wsClass);
104   find projectsWithSameName(repoProject, wsProject);
105
106   // Classes must have the same name...
107   CP3Class.name(repoClass, commonSimpleName);
108   WType.name(wsClass, commonSimpleName);
109 }
110
111 // -------------------------------------------------------------------------
112
113 pattern addedClasses(wsClass: WType) = {
114   // Find the following attributes for the selected type:
115   //  package name, simple name, project.
116   WType.name(wsClass, simpleName);
117   WProject.packageFragmentRoots.packageFragments
118       .compilationUnits.types(wsProject, wsClass);
119
120   // Find the project in the repository.
121   find projectsWithSameName(repoProject, wsProject);
122
123   // Check that the repository project does not contain a class with the same name.
124   neg find classNamesInRepoProjects(repoProject, simpleName);
125 }
126
127 pattern removedClasses(repoClass: CP3Class) = {
128   CP3Project.classes(repoProject, repoClass);
129   CP3Class.name(repoClass, name);
130
131   // Find the project in the workspace.
132   find projectsWithSameName(repoProject, wsProject);
133
134   // Check that the workspace does not contain a class with the same name.
135   neg find classNamesInWorkspaceProjects(wsProject, name);
136 }
137
138 pattern addedMethods(wsMethod: WMethod, wsClass: WType) = {
139   // Find the method's project.
140   WMethod.name(wsMethod, name);
141   WProject.packageFragmentRoots.packageFragments
142       .compilationUnits.types.methods(wsProject, wsMethod);
143
144   //Assign the type to the class.
145   WType.methods(wsClass, wsMethod);
146
147   // Find the project in the repository.
148   find projectsWithSameName(repoProject, wsProject);
149
150   // Ensure that the method does not exist in the selected repository project.
151   neg find methodNamesInRepoProjects(repoProject, name);
152 }
153
154 pattern removedMethod(repoMethod : CP3Method) = {
155   // Find the method's project
156   CP3Method.name(repoMethod, name);
157   CP3Project.classes.methods(repoProject, repoMethod);
```

```
158
159   // Find the project in the workspace.
160   find projectsWithSameName(repoProject, wsProject);
161
162   // The workspace project is missing.
163   neg find methodNamesInWorkspaceProjects(wsProject, name);
164
165 }
166
167 pattern addedFields(wsField: WField, wsClass: WType) = {
168   // Find the method's project.
169   WField.name(wsField, name);
170   WProject.packageFragmentRoots.packageFragments
171      .compilationUnits.types.fields(wsProject, wsField);
172
173   //Assign the type to the class.
174   WType.fields(wsClass, wsField);
175
176   // Find the project in the repository.
177   find projectsWithSameName(repoProject, wsProject);
178
179   // Ensure that the method does not exist in the selected repository project.
180   neg find fieldNamesInRepoProjects(repoProject, name);
181 }
182
183 pattern removedFields(repoField: CP3Field) = {
184   // Find the method's project
185   CP3Field.name(repoField, name);
186   CP3Project.classes.fields(repoProject, repoField);
187
188   // Find the project in the workspace.
189   find projectsWithSameName(repoProject, wsProject);
190
191   // The workspace project is missing.
192   neg find fieldNamesInWorkspaceProjects(wsProject, name);
193
194   // Bugfix: remove static initialiser methods.
195   //check(! name.contains("<clinit>"));
196 }
197
198 pattern incomingInheritances(repoSource : CP3Class, wsTarget : WType) = {
199   // Find equal classes in the two models.
200   find classesWithSameFQNAndSameProject(repoTarget, wsTarget);
201
202   // Join on dependencies.
203   CP3Dep.from(dependency, repoSource);
204   CP3Dep.to(dependency, repoTarget);
205
206   // Select only the inheritance dependencies.
207   //CP3Dep.type(dependency, 4);
208   CP3Dep.type(dependency, type);
209
210   // Dependency type: inheritance.
211   check (type == 5);
212 }
213
214 pattern incomingClassUsages(repoSource : CP3Class, wsTarget : WType) = {
215   // Find equal classes in the two models.
```

```
  find classesWithSameFQNAndSameProject(repoTarget, wsTarget);

  // Join on dependencies.
  CP3Dep.from(dependency, repoSource);
  CP3Dep.to(dependency, repoTarget);

  // Select only the classes usage dependencies.
  CP3Dep.type(dependency, type);

  // Dependency type: method usage.
  check (type == 4);
}

pattern incomingMethodCalls(repoSource : CP3Method, wsTarget : WMethod) = {
  // Join the methods.
  find methodsWithSameNameAndSameProject(repoTarget, wsTarget);

  // Join on dependencies.
  CP3Dep.from(dependency, repoSource);
  CP3Dep.to(dependency, repoTarget);

  // Select only the classes usage dependencies.
  CP3Dep.type(dependency, type);

  // 1. method call
  check (type == 1);
}

pattern incomingMethodOverride(repoSource : CP3Method, wsTarget : WMethod) = {
  // Join the methods.
  find methodsWithSameNameAndSameProject(repoTarget, wsTarget);

  // Join on dependencies.
  CP3Dep.from(dependency, repoSource);
  CP3Dep.to(dependency, repoTarget);

  // Select only the classes usage dependencies.
  CP3Dep.type(dependency, type);

  check (type == 1);
}

pattern incomingFieldAccesses(repoSource : CP3Method, wsTarget : WField) = {
  // Join the fields.
  find fieldsWithSameNameAndSameProject(repoTarget, wsTarget);

  // Join on dependencies.
  CP3Dep.from(dependency, repoSource);
  CP3Dep.to(dependency, repoTarget);

  // Select only the classes usage dependencies.
  CP3Dep.type(dependency, type);
  check (type == 3);
}

pattern impactCausedByRemovedFields(repoFrom : CP3Method,
  repoTo : CP3Field, type : EShort) = {
  find removedFields(repoTo);
```

```
  CP3Dep.from(d, repoFrom);
  CP3Dep.to(d, repoTo);
  CP3Dep.type(d, type);
  check(type == 4);
}

pattern impactCausedByRemovedMethods(repoFrom : CP3Method,
  repoTo : CP3Method, type : EShort) = {
  // method call
  find removedMethod(repoTo);
  CP3Dep.from(d, repoFrom);
  CP3Dep.to(d, repoTo);
  CP3Dep.type(d, type);
  check (type == 1);
} or {
  // or method override
  find removedMethod(repoTo);
  CP3Dep.from(d, repoFrom);
  CP3Dep.to(d, repoTo);
  CP3Dep.type(d, type);
  check (type == 3);
}

pattern impactCausedByRemovedClasses(repoFrom : CP3Class,
  repoTo : CP3Class, type : EShort) = {
  // class usage.
  find removedClasses(repoTo);
  CP3Dep.from(d, repoFrom);
  CP3Dep.to(d, repoTo);
  CP3Dep.type(d, type);
  check (type == 3);
} or {
  // or inheritance
  find removedClasses(repoTo);
  CP3Dep.from(d, repoFrom);
  CP3Dep.to(d, repoTo);
  CP3Dep.type(d, type);
  check (type == 3);
}

pattern outgoingMethodCallDependencyAdded(wsMethodFrom: WMethod,
  wsMethodTo: WMethod) = {
  // Finds method call dependencies between projects.
  WDependency.from(d,wsMethodFrom);
  WDependency.to(d, wsMethodTo);
  WDependency.type(d,::METHOD_CALL);
  WProject.packageFragmentRoots.packageFragments
    .compilationUnits.types.methods(wsP1, wsMethodFrom);
  WProject.packageFragmentRoots.packageFragments
    .compilationUnits.types.methods(wsP2, wsMethodFrom);
  check (!wsP1.equals(wsP2));

  // The elements exist in the repository
  find methodsWithSameNameAndSameProject(repoMethodFrom, wsMethodFrom);
  find methodsWithSameNameAndSameProject(repoMethodTo, wsMethodTo);

  // But the dependency does not exist in the workspace
  CP3Dep.type(repoD, type);
```

```
332   check(type == 3);
333   neg find repoDependency(repoMethodFrom, repoMethodTo, repoD);
334 
335 }
336 
337 
338 private pattern repoDependency(ceFrom: CP3AbstractItem,
339   ceTo: CP3AbstractItem, d : CP3Dep) = {
340   CP3Dep.from(d, ceFrom);
341   CP3Dep.to(d, ceTo);
342 }
```