Fachbereich 4: Informatik

Design Patterns for API Analysis & Migration

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Joachim Pehl

Erstgutachter: Prof. Dr. Ralf Lämmel
Institut für Informatik

Zweitgutachter: Dr. Markus Kaiser
Institut für Informatik

Koblenz, im Januar 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Acknowledgments

I would like to thank my adviser Prof. Dr. Ralf Lämmel for offering me this interesting topic and for his assistance during the developement of this thesis. He provided me with assistance during the implementation of the tools mentioned in this thesis and with many interesting ideas.

I would also like to thank Dr. Markus Kaiser for his advises regarding the structure of my thesis.

Furthermore, I would like to thank the members of the Software Languages Team, especially Ekaterina Pek, David Klauer and Jan Baltzer whose implementations for the API 2.0 project became a foundation of this thesis.

I am also grateful to Cecile Zedtwitz and Marc Zobel for proofreading my thesis.

Finally, I would like to thank my family, especially my parents, for their support and patience during my studies.

Contents

1 Introduction **2**
1.1 Motivation 2
1.2 Related work 3
1.3 Outline 6

2 Pattern and Technology Overview **8**
2.1 Analysis Patterns 8
2.2 Migration Patterns 9
2.3 Main Technologies 10

3 Analysis Patterns **11**
3.1 Configure - Analyze - Export (CAE) 11
3.1.1 Intent 11
3.1.2 Motivation 11
3.1.3 Applicability 11
3.1.4 Structure 12
3.1.5 Participants 13
3.1.6 Implementation 13
3.1.7 Related Patterns 14
3.2 Variable Analyzer 14
3.2.1 Intent 14
3.2.2 Applicability 14
3.2.3 Motivation 14
3.2.4 Structure 14
3.2.5 Participants 16
3.2.6 Implementation 16
3.2.7 Related Patterns 18
3.3 Independent Data Model 18
3.3.1 Intent 18
3.3.2 Applicability 19
3.3.3 Motivation 19
3.3.4 Structure 19
3.3.5 Participants 19
3.3.6 Implementation 20
3.3.7 Related Patterns 21
3.4 Multiple Outputs 21
3.4.1 Intent 21
3.4.2 Motivation 21

3.4.3 Applicability . . . 22
3.4.4 Structure . . . 22
3.4.5 Participants . . . 22
3.4.6 Implementation . . . 23
3.4.7 Related Patterns . . . 25

4 Migration Patterns **26**
4.1 Simple Mapping Language . . . 26
4.1.1 Intent . . . 26
4.1.2 Motivation . . . 26
4.1.3 Applicability . . . 27
4.1.4 Structure . . . 27
4.1.5 Participants . . . 28
4.1.6 Implementation . . . 29
4.1.7 Related Patterns . . . 31
4.2 Mapping Language Extension . . . 31
4.2.1 Intent . . . 31
4.2.2 Motivation . . . 32
4.2.3 Applicability . . . 32
4.2.4 Structure . . . 32
4.2.5 Participants . . . 33
4.2.6 Implementation . . . 34
4.2.7 Related Patterns . . . 39
4.3 Mapping Error Tracking . . . 39
4.3.1 Intent . . . 39
4.3.2 Motivation . . . 39
4.3.3 Applicability . . . 40
4.3.4 Structure . . . 40
4.3.5 Participants . . . 41
4.3.6 Implementation . . . 42
4.3.7 Related Patterns . . . 45
4.4 Generic Wrapper Recycler . . . 46
4.4.1 Intent . . . 46
4.4.2 Motivation . . . 46
4.4.3 Applicability . . . 47
4.4.4 Structure . . . 47
4.4.5 Participants . . . 47
4.4.6 Implementation . . . 48
4.4.7 Related Patterns . . . 53
4.5 Wrapper Creator . . . 53
4.5.1 Intent . . . 53
4.5.2 Motivation . . . 53
4.5.3 Applicability . . . 53
4.5.4 Structure . . . 53
4.5.5 Participants . . . 53
4.5.6 Implementation . . . 55
4.5.7 Related Patterns . . . 58
4.6 AST Migrator . . . 58
4.6.1 Intent . . . 58

4.6.2 Motivation 58
4.6.3 Structure 59
4.6.4 Applicability 59
4.6.5 Participants 60
4.6.6 Implementation 60
4.6.7 Related Patterns 62

5 Observations and Conclusions **64**
5.1 API analysis patterns 64
5.2 API migration patterns 65
5.3 Future work 66

6 Appendix **67**
6.1 Tool Introduction 67
6.1.1 Fundamental differences between analyzers 67
6.1.2 ASM 68
6.1.3 Recoder 70
6.1.4 JDT 72
6.1.5 Other tools 75
6.2 JDT AST Nodes 76
6.3 Complete DTD 76
6.4 Example grammar for the *Simple Mapping Language* 78
6.5 Mapping complexity 78

Abstract

Software projects typically rely on several, external libraries. The interface provided by such a library is called API (application programming interface). APIs often evolve over time, thereby implying the need to adapt applications that use them. There are also reasons which may call for the replacement of one library by another one, what also results in a need to adapt the applications where the library is replaced.

The process of adapting applications to use a different API is called API migration. Doing API migration manually is a cumbersome task. Automated API migration is an active research field.

A related field of research is API analysis which can also provide data for developing API migration tools. The following thesis investigates techniques and technologies for API analysis and API migration frameworks. To this end, design patterns are leveraged. These patterns are based on experience with API analysis and migration within the Software Languages Team.

Deutsche Zusammenfassung

Software Projekte nutzen typischerweise mehrere externe Programmblibliotheken. Die Schnittstelle, die solch eine Programmblibliothek zur Verfügung stellt, wird als API (application programming interface) bezeichnet. APIs werden üblicherweise laufend weiterentwickelt, was es notwendig macht, dass die Anwendungen, welche sie verwenden, entsprechend modifiziert werden. Zudem kann es kann vorkommen, dass eine Programmblibliothek durch eine andere ersetzt werden soll, was ebenfalls zur Folge hat, dass die Anwendungen, wo die API verwendet wurde, modifiziert werden müssen.

Den Vorgang eine Anwendung so zu modifizieren, dass eine andere API verwendet wird, bezeichnet man als API Migration. Manuelle API Migration ist eine mühselige und zeitintensive Aufgabe, deshalb ist automatische API Migration ein aktives Forschungsfeld.

Ein verwandtes Forschungsgebiet ist API Analyse, welches Daten zur Verfügung stellt, die helfen können Werkzeuge für API Migration zu entwickeln. Die hier vorliegende Arbeit behandelt Techniken und Technologien für die Entwicklung von Werkzeugen für API Analyse und API Migration. Die Ergebnisse werden als Design Patterns präsentiert, welche auf unseren Erfahrungen mit API Analyse und API Migration innerhalb des Software Language Teams basieren.

Chapter 1

Introduction

1.1 Motivation

Modern developers often use third-party software libraries for certain tasks. While the programmer has to learn the interface (API or application programmer interface) of a library, he does not have to reimplement the features of the libraries and once learned he can easily use it in other software projects. Since most commonly used APIs are highly sophisticated, the usage of APIs often improves the quality of the code and save developement time. However, software libraries and their respective APIs evolve over time. It may also happen that a developer decides to switch to a different API (for example, for performance reasons). In both cases he has to adapt the software to the new API. The process of switching APIs is called API migration. API migration is usually done manually, which can be a time-consuming task and it has to be applied to every software projects which use the API in question. Therefore, there is a high interest in (semi)automatic API migration.

A field of research related to API migration is API analysis which can provide facts usefully for API migration (for example, by providing information about API usage). Furthermore, API analysis may help to improve code quality by granting a better understanding of the properties of software projects, for example, by comparing the API usage between software projects.

During our own research, regarding API analysis and API migration, we designed several tools for API analysis and migration. A big amount of time was spent for redesigning them. Dong et. al specified the problem with such developement cycles precisely:

”Many software developers routinely apply design patterns in their software systems to reuse expert design experience and record design decisions. However, such high-level design information is typically lost in system source code when the systems are deployed. The architectural design document is normally not deployed with source code. Even the design document is available, it may not be consistent with the source code after the system has evolved and been changed due to new requirements.” [DZ07]

After seeing that other analysis/migration tools had similar problems (and used similar solutions), we decided to extract domain-specific design patterns out of our tools, respectively out of the experienced gained during the developement of the tools, for later reuse.

Design patterns provide guidelines for developing software which has certain properties. Furthermore, usage of design patterns should improve the readability of the code, since the programmer may recognize the pattern, and the resulting software should be easy to expand.

Design patterns may vary strongly in scope, the most well-known catalogue of general design patterns "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV95] already contains patterns that go from single classes (*Singleton* for example) to patterns like the *Flyweight* pattern which are designed for a large structure. Other works describe even smaller patterns (for example, "Micro patterns in Java code" [GM05]) or patterns which are restricted to a specific domain (for example, "An analysis pattern for invoice processing" by Fernandez et al. [FY10]). While domain-specific patterns are usually not useful for other domains, they are very helpful for developing code in that specific domain. They usually already consider domain-specific problems. Many domain-specific patterns can be seen as special cases of other patterns or as composed versions of multiple patterns.

This thesis presents the patterns which we recognized during our work.

1.2 Related work

We have multiple key aspects throughout this thesis. Therefore this section is split into three parts:

Design Patterns Here we mention work related to design patterns in general and specific patterns which are used or are helpful for the patterns described in this thesis.

Migration Here we will mention some important work about API migration.

Language Technologies The last part presents some technologies and tools for code analysis and transformation.

In addition to the following works, there are two projects which should be mentioned. This thesis was not planned from the start, but resulted from two different projects. The first project had the goal to analyze API usage. We developed several tools for the two iterations of that project which are referenced as API 1.0 and API 2.0 throughout this thesis [LPS11].

The second project was going to be a thesis about API migration. While, at the time of writing, the project did not produce enough scientific data for a thesis, we gained a lot of experience for developing API migration tools. This experience is presented in this thesis in the form of the API migration patterns.

Design Patterns

As mentioned above, the patterns presented in this thesis were created by extracting them from the source code of our developed tools and/or by creating them based on the problems and requirements we observed during their implementation and maintenance. This made sense since they are domain-specific unlike the patterns specified in [GHJV95]. Furthermore, we incorporated features and limitation of the Java language [GJS05].

While this approach worked for us, developing procedures for mining or creating of design patterns is an ongoing research field. Winters et al. ("Dealing with abstraction: Case study generalization as a method for eliciting design patterns" [WM09]) have a catalogue of possible technologies and are proposing a methodology for pattern mining/creation by generalisation. Another possible approach was identified by Claudia Iacob ("A Design Pattern Mining Method for Interaction Design" [Iac11]). She proposes a structured method of detecting patterns based on the results of workshops on a specific design issue.

Of course most of the patterns presented in this thesis are related or composed of other patterns. Some of these relations are mentioned in the ”related patterns” section at the end of each presented pattern. The patterns out of the pattern catalog [GHJV95] are well-known so we do not need to explain them here in more detail, but we also mention some less-known patterns. Those are explained in more detail below.

Even if one (or multiple) pattern(s) are present for a problem and a programmer is aware of them, it will not be certain that he will apply them due to time constraints or a misinterpretation of the problem space. Jensen et al. (”On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns” [JC10]) propose an approach to introduce design patterns automatically during the refactoring process based on software engineering metrics and genetic programming. A related problem is improper usage of a pattern. It might be the case that a pattern is not applied as it should be due to later changes or misinterpretation from the programmer. A pattern which is applied improperly may render the pattern's properties invalid. Blewitt et al. (”Automatic Verification of Design Patterns in Java” [BBS05]) present a pattern specification language called SPINE and a proof engine for that language.

Interesting patterns Multiple of our patterns require a *Factory* of some kind. Welicki et al. described a pattern which provides big amounts of flexibility since it is driven by Metadata (”The *Dynamic Factory* Pattern” [WYWB08]), Welicki et al. also described a related pattern which allows to manipulate existing objects by reloading their Metadata definitions (*Adaptive Object Model Builder* [WYWB10]). If an AST based technology is used, it will often be necessary to traverse ASTs with a big amount of different ASTNodes. Bringert et al. presented a pattern which is focused on traversing the AST and effective reuse of that traversal component (A Pattern for Almost Compositional Functions [BR06]).

The *Model View Controller* pattern [KP88] is a rather universal structural pattern. The structural patterns for API analysis, presented in this thesis, focuses on other parts of the analysis software. But, if these patterns were used together, they could be seen as an implementation of the *MVC* pattern.

Some of the patterns presented in this thesis require configuration information, for example, all frameworks where a third-party tool is used for analyzing code, will need some kind of configuration. The *Variable Analyzer* pattern (see 3.2) focuses on the configuration file itself. However, it is also necessary to consider how to handle the configuration, read it every time, save it in memory etc. The *Configuration Data Caching* Pattern [Wel06] by Leon Welicki provides a structure that enables the code to handle effectively the configuration. Summarized, it stores the configuration in memory in a repository that only parses the configuration files if necessary.

Guerra et al. present ”A Pattern Language for Metadata-based Frameworks” [GSF10]. While most of the patterns presented in this thesis used an AST, representing the facts from the analyzer as Metadata is also a sensible approach. Therefore, multiple of the pattern presented by Guerra et al. are useful for our approaches. The following patterns are the most important ones (regarding this thesis):

Metadata Container This pattern separates the reading of the metadata from the framework which uses them.

Metadata Repository This pattern provides a repository for accessing the data and eliminating unnecessary readings of the metadata.

Some more useful patterns can be found in Nock Clifton's Data Access pattern catalogue (”Data Access Patterns: Database Interactions in Object-Oriented Applications”

[Noc03]). Especially interesting are:

Data Accessor which decouples data reading and data access in the framework. The previously mentioned *MetaData Container* pattern is based on the same principle.

Demand Cache which populates the data lazily on framework demand.

Active Domain Object which abstracts the semantics of a data model (and its access details) for the framework and provides the client with an easy solution to retrieve and modify the requested data.

Cache Collector which, similar to a garbage collector, deletes data sets which are no longer needed.

Domain Object Assembler which is a *Factory* pattern for populating a data repository.

Primed Cache which tries to populate a cache first with the most frequently required objects.

Object-relational map which decouples domain objects from their underlying data models and data access operations.

Migration

Basically we can group the ongoing research regarding API migration into three groups. The first group is migration between different versions of the same API, this is often applied by improved refactoring capabilities. An example would be CatchUp! by Henkel et. al [HD05]. In that approach the refactorings applied to the API by its designer are recorded via CatchUp!. A user who wants to migrate to another version of that API can use CatchUp! and the saved refactorings on his code. A related approach is ComeBack! by Savga et. al [SRG08], it also stores the refactoring steps, but it tries to generate an *Adapter* for the upgraded API. Another approach in that area is proposed by Tammo Freese ("Refactoring-aware version control" [Fre06]). Freese tries to capture the refactorings by an extended version control system, which has the advantage that it has no problem handling updates done by multiple developers. A suitable way for expressing refactoring steps is presented by Tip et al. ("Refactoring Using Type Constraints" [TFK+11]), which is used by Balaban et al. for class library migration ("Refactoring support for class library migration" [BTF05])

These solutions have the drawback that they wont work if the refactoring step is applied without tool assistance. Dig et. al propose an approach to locate these refactorings later. The found refactorings can than be applied in a similar way. For example, by migration or by an adapter layer ("Automated Upgrading of Component-Based Applications" [DJ06]).

The next group is migration between different APIs, this includes our own research interest.

The last group is migration between APIs in different languages. Zhong et al. developed an algorithm (and an implementation of it) that mines mapping relations between API classes and methods by comparing two versions of the same project written in different languages ("Mining API Mapping for Language Migration" [ZTX+10]).

Language technologies

Our main interest for API analysis is to gain knowledge about API usage [LPS11].

One problem big that occurs in real world examples is the inclusion of API code in the source code of a program. This means that either the programmer copied the code or that he reimplemented it. Kawrykow et al. proposed a technique to find such cases ("Improving API Usage through Automatic Detection of Redundant Code" [KR09]).

We mentioned multiple works to focus on migration between different versions of the same API, these techniques have in common that there is a need for analyzing the differences between these versions. Neamtio et. al. developed an algorithm and a tool for analyzing the differences by matching the ASTs of the parsed programs (in that case C programs), what could also be a basis for a migration framework ("Understanding Source Code Evolution Using Abstract Syntax Tree Matching" [NFH05]).

Kniesel et al. provided a comparison between various analyzer frameworks, by using them for design pattern detection ("Comparison of LogicBased Infrastructures for Concern Detection and Extraction" [KHR07]). Amongst others, JTransformer [JTr] and JQuery [JDV03] which are mentioned below.

Many of the used frameworks are relying on an AST for data storage. Since we are usually interested in analyzing the whole structure, this makes sense. However, if we want to know specific traits of a class, it will be annoying to traverse the whole AST. Therefore, tools that are made for assisting direct code developement, usually use other approaches, for example the JTransformer framework provides a Prolog interface for easily querying the source code (internally it still works on the JDT AST). Another example would be the Java Tool Language (JTL) by Cohen et al.[CGM06] which allows queries "formulated in First Order Predicate Logic augmented with transitive closure (FOPL*)" on Java code.

Other Frameworks One interesting framework, which was already mentioned, is the JTransformer [JTr] framework which is a query and transformation engine for Java source code. It uses the JDT libraries to analyze Java source code and represents the data as Prolog facts. The facts can be manipulated by using conditional transformations.

A tool designed for browsing the source code is JQuery [JDV03]. It is implemented on top of an expressive logic query language. While it lacks code manipulation features, it tries to combine the advantages of hierarchical code browsers and query tools.

Another interesting work is JastADD ("The JastAdd extensible Java compiler" [EH07]) presented by Ekman, Torbjörn and Hedin, Görel. It is a Java compiler which allows the user to extend Java with new language constructs. It is also usable for building a static analysis tool. The paper "Building Semantic Editors using JastAdd" [SH11], by the same authors, presents an example application for JastADD.

1.3 Outline

Used terminology The following list explains some terms which are used in this thesis.

Target Language The target language is the programming language in which the APIs are written. In this thesis this always means Java, but the presented patterns are not restricted to Java.

Source API The source API is the API which is in use before the migration process.

Target API The target API is the API which we want to use after the migration process.

Source/Target Method Similar to source and target API, we use the term source method to specify the method which is used before the migration process and the target method indicates the method which should be used after the migration.

Mapping A mapping is a description written in a mapping language.

Data Consumer A data consumer is a piece of software that reads data from an analysis process and uses it in some way. For example, by formatting and storing them to a file.

Mapping Complexity The mapping complexity is a classification of the requirements for migrating a specific method. It is described more precisely in Section 6.5 of the appendix and in the pattern description where it is used.

Pattern presentation The patterns described in this thesis will be structured in the following manner:[1]

Intent A short explanation of the intention behind this pattern.

Motivation An explanation of the motivation behind the described pattern.

Applicability A description of the most common scenarios where this pattern can or should be applied.

Structure This section will usually contain an UML diagram of the code structure for the described pattern. For some pattens we will also discuss the structure of important files.

Participants At that place we will explain the role of the elements shown in the structure section.

Implementation The implementation section explains the components of the UML diagram in more detail and contains example code. Furthermore, possible implementation issues and variants are mentioned here. This section differs a lot for the different pattern since the scope of the patterns presented in this thesis varies highly.

Related Patterns If the presented pattern has a close relation to other patterns, these patterns will be mentioned there. Furthermore, we will mention patterns or technologies that can assist the presented pattern.

Chapter outline This chapter explained our motivation and related work for this thesis.

The second chapter will give an overview over the patterns presented in this thesis, the relationship between the patterns and the used technologies.

Chapter 3 contains the patterns for API analysis while the fourth chapter contains the patterns for API migration. Both chapters are the main contribution of this work.

Chapter 5 will summarize our conclusions and observations and discuss possible future work regarding this thesis.

The last chapter contains the appendix. It consists of an introduction to the main analysis and transformation frameworks which are used by us. It also contains some additional information which we consider interesting or helpful, but these information would bloat their respective sections (for example, complete versions of code snippets in the implementation sections).

[1] We did chose a presentation related to the one used in [GHJV95].

Chapter 2

Pattern and Technology Overview

This chapter will present a short overview over the patterns which we identified. The relationship between these patterns will also be explained. For a more detailed explanation for each pattern read their respective ”intent” or ”motivation” section.

We will also mention the used technologies. A more detailed introduction to the different tools can be found in the appendix (see Section 6.1).

2.1 Analysis Patterns

This section introduces the patterns we identified for creating an analysis framework.

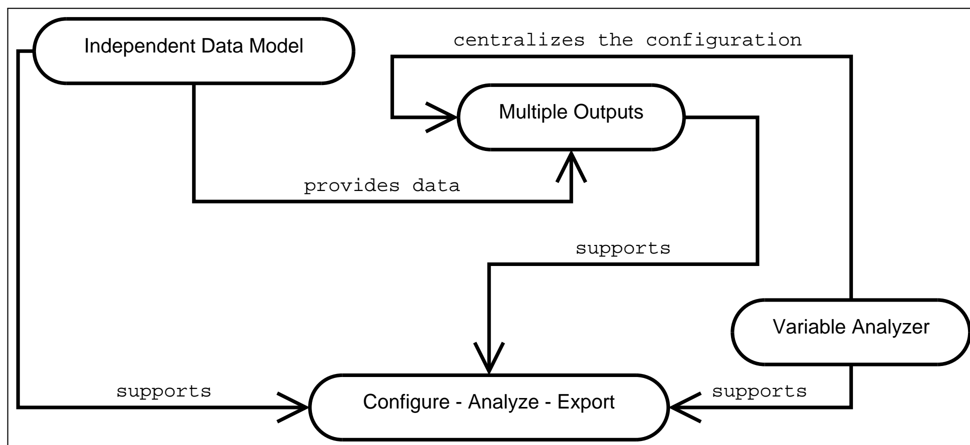

Figure 2.1: Relations between the analysis patterns

Figure 2.1 shows you the relations between the analysis patterns described in this thesis (each entity in the diagram represents one of the analysis patterns).

The four patterns are:

Configure - Analyze - Export It provides the basic structure for an analyzer.

Independent Data Model It supports the structure provided by *CAE*, stores the generated data in memory and provides an interface for accessing the data.

Multiple Outputs It handles the consumer accessing the data generated by the analyzer.

Variable Analyzer It configures the analyzer.

If all four patterns are applied we will have an analyzer that meets our requirements for a flexible analysis framework.

These patterns emphasize heavily that it is important to be able to switch easily the analyzer framework. Besides possible technical aspects, the analyzers are either source code or bytecode based, which results in strong differences between the analyzers' results. In summary, the compiler is able to change the syntax of the source code. This is not limited to some special cases but this effect is noticeable in many cases. We strongly advise to read Section 6.1.1 where this is explained in more detail.

2.2 Migration Patterns

The patterns introduced in this section are the patterns we identified for an API migration framework. These migration patterns can be divided into three groups.

Mapping Specification The *Simple Mapping Language* and *Mapping Language Extension* provide a guideline how to setup the grammar for the mapping and a software structure for parsing them.

Migration Assist The *Wrapper Creator* and *AST Migrator* patterns are meant to assist the API migration process.

Runtime Assist The *Mapping Error Tracking* and *Generic Wrapper Recycler* patterns are supposed to assist a migrated software during runtime.

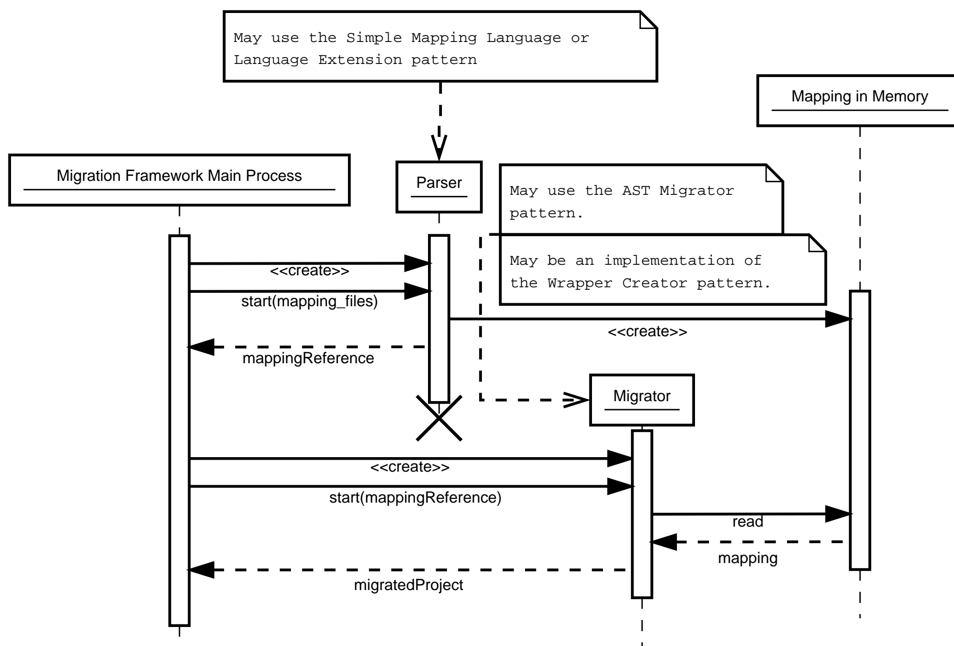

Figure 2.2: An overview of the relation between the migration patterns

Figure 2.2 shows an overview about the migration patterns that are used for the migration part of the software. *Simple Mapping Language* and *Mapping Language Extension* are providing a specification for the mapping and a software structure for the parser. The *Wrapper Creator* and *AST Migrator* patterns are useful for the migration part of the framework.

The two remaining patterns are shown in Figure 2.3. They assist the migrated software.

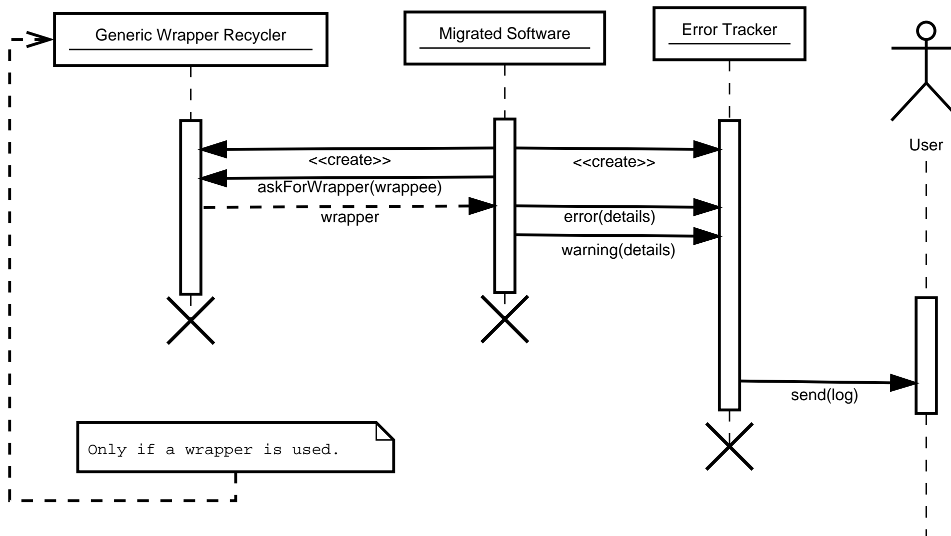

Figure 2.3: An overview of the relations between the Migration patterns

These four patterns can be used together but they are more independent from each other than the analysis patterns. They are more problem-oriented and a possible user has to consider carefully what else he needs for a migration framework. But the *Simple Mapping Language* and *Mapping Language Extension* patterns provide a basis for such a framework.

2.3 Main Technologies

Programming Language The patterns presented in this thesis were implemented in Java. The target projects for our migration and analysis tests were also implemented in Java. Because of that, we incorporated features and problems of the Java language into the description of the patterns, but that does not mean that these patterns are limited to the Java language. The only additional language which was used was Prolog, mainly for maintaining and querying the facts generated from the analysis. We will mention that in the implementation section of the pattens where Prolog was used.

Important Frameworks Three major APIs were used for the implementation of the tools which we used for our API analysis and API migration tests, these are ASM [Eri], Recoder [rec] and the JDT libraries [JDT] from the Eclipse system [ecl]. A short introduction to these tools and a short comparison can be found in Section 6.1.

Chapter 3

Analysis Patterns

This chapter contains the patterns we identified for API analysis. An overview over these patterns is in section 2.1.

3.1 Configure - Analyze - Export (CAE)

3.1.1 Intent

Provide a software structure which simplifies common modifications to the analyzer and switching of the components of an analyzer framework.

3.1.2 Motivation

As already explained in section 1.2, we developed several tools for analyzing APIs. During that work the structure of the tools was heavily changed in order to improve our generated datasets or to broaden the scope of our tools.

We noticed that the necessary code changes can be assigned to the following three sets:

1. Refinement of the analyzer's configuration process.
2. Refinement of the analysis code.
3. Refinement of the output.

By separating these three parts of the code we were able to reduce heavily the amount of recoding if one of these three parts needed to be changed or extended.

So we suggest in this pattern that the structure of an API analyzer framework is based upon these three sets.

3.1.3 Applicability

This pattern provides a basic structure for an API analyzer. It is intended to be used for statical API analysis. Its applicability for dynamical analysis depends on the used analyzer framework and the demands to the performance of your program.

3.1.4 Structure

We described three sets of code changes in the motivation section of this pattern.

This means that we want to split the functionality of the framework in 3 corresponding parts:

- Configure
- Analyze
- Export

Figure 3.1 depicts a sequence diagram, showing an example for our suggested data flow in an analyzer framework.

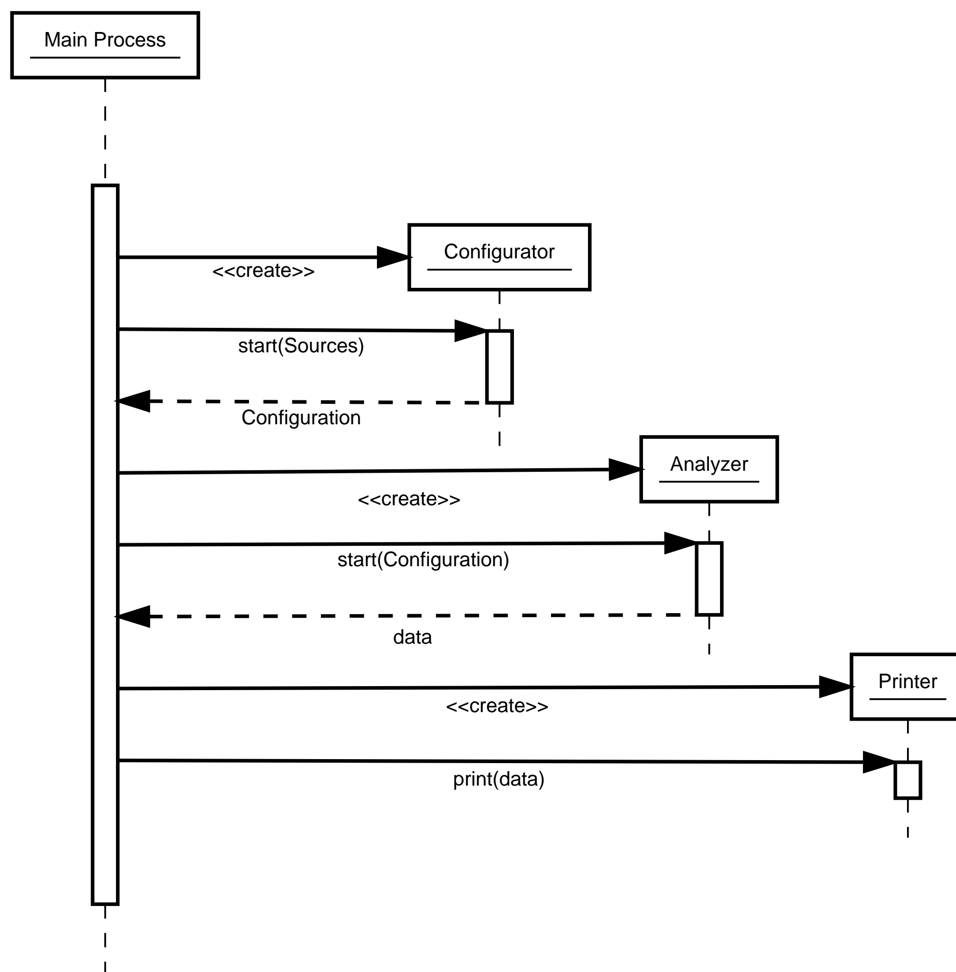

Figure 3.1: *Configure - Analyze - Export* sequence diagram

3.1.5 Participants

The participants of the sequence diagram in figure 3.1 are a client process and three distinct objects which contain the functionality of the framework:

`Configurator` It is responsible for configuring the analyzer. In this simple example it creates a configuration object which is returned to the client and passed to the `Analyzer`.

`Analyzer` It takes a configuration object, runs the analysis and returns the results to the client.

`Printer` This object is responsible for exporting the data. Later patterns will call this part a data consumer, because it is not restricted to printing the data.

3.1.6 Implementation

Communication The three distinct parts of the analyzer framework have to communicate. The sequence diagram in figure 3.1 shows that an argument is passed back from the `Analyzer` to the main client which passes it to the `Printer`. However, this is not feasible for large data sets. So you most likely want to provide an interface for these classes. The *Independent Data Model* pattern provides a possible interface. For the communication between the `Configurator` and the `Analyzer` take a look at the *Variable Analyzer* pattern. It will give you a guideline how to setup the configurator and how communication between the `Configurator` and the `Analyzer` could be handled.

Configurator The `Configurator` is responsible for providing the necessary settings for the `Analyzer`. The `Configurator` should have the following properties:

- Easily extensible.
- High readability.
- Independent from the `Analyzer`.

The first two properties are self-explanatory, the third one means that there should be no settings specific to the used analyzer library.

For an example implementation, take a look at our *Variable Analyzer* pattern (see section 3.2).

Analyzer The implementation of the `Analyzer` depends heavily on the used analyzer framework (see section 6.1 for some suggestions).

Printer The `Printer` part is responsible for exporting the generated data to the client. While this can be easily implemented in a trivial way (for example, by simply writing the data straight to some text files), you should consider the case that you want to change the way how to export the data later. In that case take a look at the *Multiple Outputs* pattern (see section 3.4).

3.1.7 Related Patterns

CAE can be seen as the foundation for the other analysis patterns, described in this thesis. Therefore, it is connected with them and we already mentioned several other patterns in the implementation section of this pattern.

The *Model View Controller* paradigm *MVC* [KP88]) splits the different responsibilities of a given software into three parts, a project that uses the *CAE* pattern and the *Independent Data Model* pattern can be considered as an implementation of a *Model View Controller* system. The `Model` is represented by the data model used for the *Independent Data Model* pattern. The `View` is represented by the output capabilities of the software and the `Controller` is represented by the analyzer (the configuration capabilities also belong to the `Controller`). If *Independent Data Model* is not used, the `Model` will be represented by the data model of the analyzer. But in that case `Controller` (analyzer's data model) and `View` (analyzer) are a single entity, thereby not meeting the requirements of *MVC*.

3.2 Variable Analyzer

3.2.1 Intent

Provide an independent and extensible configuration process for the analyzer.

3.2.2 Applicability

This pattern will be applicable if the *Configure - Analyze - Export* pattern is used or if the configuration options of the framework are going to be extended later.

3.2.3 Motivation

It is easy to imagine a simple analyzer which analyzes the public signatures of APIs used by different projects. For such an analyzer it would be sufficient to place all APIs in a single location. In such a scenario you only need two distinct modes for the analyzer: analyze a given *.jar file or all *.jar files in a specific directory.

Our first version of the ASM analyzer worked in that way. Later on we broadened the scope of the tool and used more features of the analyzer framework. This resulted in a rather unwieldy configuration. In order to remove that problem we redesigned the starting process in two ways.

1. We started to use XML files for configuration, since XML code will be easily extensible and will have a high readability if used correctly.
2. We removed all settings specific to the analyzer.

This pattern will show you a possible software structure for the configuration process, a list of options which we found to be useful and it will discuss sensible possible configuration options for the framework.

3.2.4 Structure

The structure section of this pattern consists of two parts.

- An UML diagram presenting a possible code structure.

- An informal description of a possible structure for configuration files.

Code Structure Figure 3.2 shows an example structure for a possible configurator.

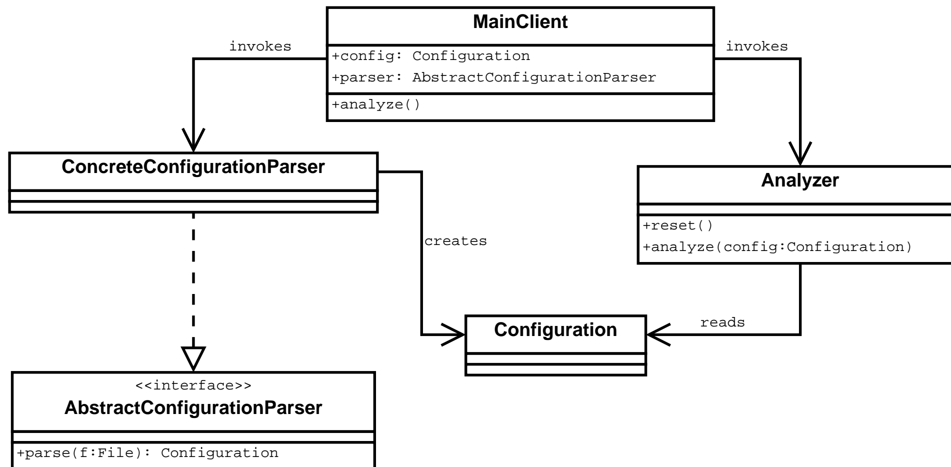

Figure 3.2: Structure of the code

File Structure This section discusses the basic structure of possible configuration files for the analyzer.

Task configuration This part should be at the top level of the configuration file and the mode of the analysis should be configured there.

You will most likely need at least two modes for the framework. An unguided mode where the framework analyzes a restricted number of files, and a guided one where additional information are passed to the analyzer. A simple example for a guided mode is to analyze all elements in a file that lists all *.jar files and all classes that where created during the compilation of a given project.

Source configuration It is necessary to specify the sources to analyze. That would be the files (or projects) that should be analyzed or the files for a guided analysis.

Furthermore, sometimes you will need to specify additional sources or information, for example, if your analyzer needs third-party libraries or if you are building the project at the start of the analysis.

Analysis configuration It is possible for most analysis frameworks to specify how much should be analyzed. For example, if no information about local variables were needed, you could disable type-binding in the JDT (see section 6.1.4) or configure ASM (see section 6.1.2) so that it skips debug information provided by the Java compiler. In most cases you want to do that because it heavily impacts performance. Literature about the used frameworks usually mentions which options increase the calculation time.

For our tools we usually wanted either a complete analysis or an analysis on the visible elements (visible in this context means accessible for the programmer). So we consider it useful to implement both as possible default modes:

Blackbox mode Analyze only the visible signature.

Whitebox mode Complete analysis.

You should also provide a way for the user to specify the analysis in detail.

Output configuration This part should contain the information for the data consumer. A detailed description of the output elements is not feasible, since it is highly depending on the output engines you are using. For example, it may be the case that you store them in a database or that you store them on a disk and so on.

3.2.5 Participants

`AbstractConfigurationParser` The interface for the parser.

`ConcreteParser` A concrete implementation of a parser.

`Configuration` This object is created by a concrete parser, it contains the information for the analyzer.

`MainClient` The `MainClient` represents the framework starting the analysis. It invokes the `ConfigurationParser` and passes the configuration to the analyzer.

`Analyzer` The framework doing the actual analysis. It reads the created configuration.

3.2.6 Implementation

This section will show an example for a configuration file. As it is important that the configuration is easily extensible, we will use the XML format in this example.

The Header

The following code snippet shows the header for our example DTD.

Listing 3.1: DTD for the header

```
1 <!ELEMENT RUN (TASK)+>
2  <!ELEMENT TASK (SOURCES,ANALYZE, OUTPUT)>
3   <!ATTLIST TASK mode (list|project) #REQUIRED>
```

`RUN` is the XMl main attribute. It contains at least one `TASK`. Each `TASK` consists of three elements which resemble the `SOURCES`, `ANALYSIS` and `OUTPUT` configuration options, we mentioned in the previous section.

As already explained, you should at least have two possible modes for the analyzer, which is the case in this example. Most likely you are going to add additional modes later.

The two modes in the DTD above are:

`list` An unguided mode, it just analyzes a list of given files.

`project` A guided mode, it analyzes certain files in a project.

Sources

Listing 3.2: DTD for the SOURCES element

```
1 <!ELEMENT SOURCES(DIRECTORY)>
2   <!ATTLIST SOURCES additional_configuration CDATA #IMPLIED>
3   <!ELEMENT DIRECTORY>
4     <!ATTLIST DIRECTORY location CDATA #REQUIRED>
5     <!ATTLIST DIRECTORY subdirectories  (true|false) 'false'>
```

The listing above shows a possibility for specifying the sources. It consists of a simple listing of the possible source directories and an attribute containing the path to additional configuration files. The `subdirectories` attribute will describe if subdirectories should also be included for the analysis. In that section you will most likely have attributes that only matter for a certain mode. For example, the `subdirectories` attribute will only be useful if there are directories which are going to be searched for class files.

Analyze

Listing 3.3: DTD for the ANALYZE element

```
1 <!ELEMENT ANALYZE(CONFIG)?>
2   <!ATTLIST ANALYZE mode
3     (whitebox|blackbox|specialized) #REQUIRED>
4   <!ATTLIST ANALYZE analyze_method_calls (true|false) false>
5   <!ELEMENT CONFIG>
6     <!ATTLIST CONFIG
7       classes (public|private|protected|default) "public"
8       methods (public|private|protected|default|false) "public"
9       fields (public|private|protected|default|false) "public"
10      local_fields (true|false) "false">
```

A simple configuration for a Java analyzer which is similar to the one we use. You can either use the `whitebox` or `blackbox` default configuration, or specify the depth of the analysis in detail.

Output

Listing 3.4: DTD for the OUTPUT element

```
1 <!ELEMENT OUTPUT(CLAUSES)>
2   <!ELEMENT CLAUSES>
3   <!ATTLIST CLAUSES directory CDATA #REQUIRED>
```

The `OUTPUT` element is rather simple. We assume that the analysis data are stored as Prolog clauses. The only necessary information for the `OUTPUT` part of the analyzer is the target directory for storing the data. It may easily happen that you need to extend the options for an `OUTPUT` element later, or that you need to add additional ones. If the *Multiple Outputs* pattern is used, it makes sense to have an `OUTPUT` element for each used data consumer.

Independency of the configuration file

When setting the outline for the configuration (for example, by designing a DTD for XML) you should try to keep all settings as general as possible. It may be the case that you want to switch the analysis framework later, because you need a more efficient or more powerful framework.

For example, the used framework is ASM and we want to specify if it needs to read the debug information of a class. Such a setting would be meaningless for another framework, hence it is more useful to add a setting for the desired results. That would mean a setting had to be added which specifies if local variables should be analyzed. This information can be used by every analysis framework.

Extensibility

We mentioned that the configuration should be extensible. During the development of our tools the following parts of the configuration were extended:

- More modes for the analyzer.
- The settings for the analyzer got more detailed.
- More detailed options for the output part.

These extensions require very different measures. It is important that it should be possible to extend the configuration and it should still retain its readability. This can be done by using XML or a similar technology.

When adding more details to the analyzer, it is important that all added settings have a sensible default setting, so that old configuration files can still be used. Due to our experience, when you discover that you need more detailed settings, these settings will only be needed in some cases. It will be inefficient if you make older files invalid just because they do not use those settings. Furthermore, in most cases your analysis framework will already contain default settings, but it makes more sense if you add default settings manually. If you would just use the default settings of a framework, it would lead to confusion if the new framework uses different settings. By providing these defaults yourself (and clearly documenting them) you can avoid confusion.

3.2.7 Related Patterns

This pattern showed the possible variance of an analyzer. By separating the configuration from the actual analyzer it supports the *CAE* pattern.

For handling effectively the parsed information, you may consider to use the *Configuration Data Caching Pattern* by Welicke et al. (see Section 1.2 or [Wel06]).

3.3 Independent Data Model

3.3.1 Intent

Provide a (non-persistent) data model, which is independent from the analyzer, for the facts generated by an analysis tool.

3.3.2 Applicability

This pattern will be applicable if a strong disjunction between the analyzer and the data consumption is needed, for example, if you want to be able to switch easily the analyzer framework.

3.3.3 Motivation

There are three possible ways to handle the analyzer's output:

1. Keep the data model which is created by the analyzer.
2. Use the data immediately.
3. Keep the data in a data model independent from the analyzer.

The first option will be useful if you are sure that you will keep the analyzer framework and if the framework provides a good data structure. An example for that is the JDT (see Section 6.1.4). The JDT provides an AST closely related to the Java model and it can be used directly to modify Java code.

Our first analyzer used the second option, it stored its output directly in a SQL database, we switched later to a system that stored the facts as Prolog clauses. This showed us two shortcomings of our tool.

1. We had no simple way to add a new format (or switch to a new one) (this problem is also related to the *Multiple Outputs* pattern (see 3.5)).
2. The SQL commands were scattered in the analyzer's source code.

Because of that, we were forced to rewrite more parts of the code than necessary.

In order to avoid such problems it is advisable to use the third option. Doing so helps to disjoint the analysis framework and the data consumption and leads to two major advantages:

- The analyzer and the data export classes are independent and can easily be changed without modifying the other part.
- If the analyzer framework changes, the necessary code changes will be minimized.

This pattern will describe how to design a suitable structure for a non-persistent data model. This pattern is heavily inter-weaved with other patterns presented in this thesis. Since we gain the most benefit from this pattern. If analyzer and data consumer will be separated and if analyzer and data consumer will be prone to be changed or extended, we assume that the *CAE* pattern and the *Multiple Outputs* pattern are used.

3.3.4 Structure

Figure 3.3 shows an UML diagram of an independent data model as it is envisioned for this pattern.

3.3.5 Participants

Independent Data Model This entity represents the data structure.

`Factory` The `Factory` is responsible for populating the independent data model.

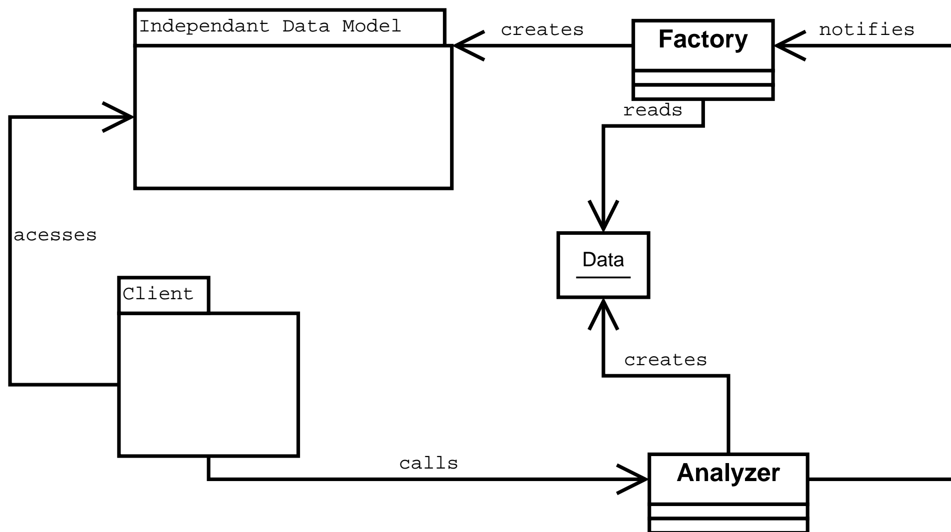

Figure 3.3: Software Structure

`Data` The data in the form created by the analyzer, which is used by the `Factory`.

`Analyzer` This class represents the analyzer.

`Client` The `Client` represents the class accessing the data, for example an implementation of the *Multiple Outputs* pattern.

3.3.6 Implementation

Structure of the data

Usually it is advisable to keep the structure close to the target language. This has several advantages:

1. It will be easy to understand the data model if the users knows the target language.
2. The transformation of the analyzer data should be straight forward as the data structure often resembles the data structure of the language.

One possible drawback is, if you are aiming for a framework that accepts several languages, it will most likely be hard for a user, not knowing the language being the inspiration for the data model, to understand the data model. However, implementing a data model for each language that can be analyzed by the framework, is not feasible. Therefore, this drawback can usually be neglected. Furthermore, you are aiming most likely for languages that share similarities in the first place. For example, object-oriented languages like Java and C++.

Another important design decision will be if the data model should be modifiable. An example where a modifiable database is used is JTransformer [JTr].

The UML diagram in Figure 3.4 shows a simple example model for Java.

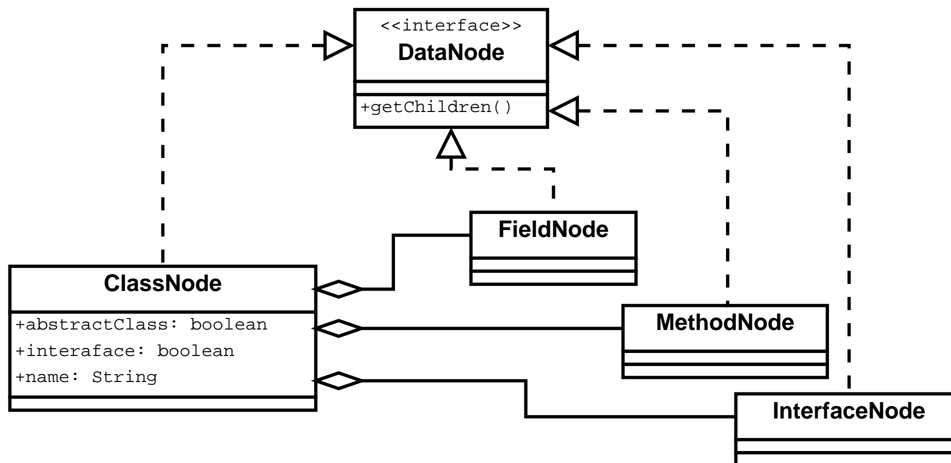

Figure 3.4: An example for a simple Java data model

3.3.7 Related Patterns

We had rather low requirements for the data model regarding performance. If you have to care about that, chances will be high that design patterns for caches will be able to provide help. A solid source for them is Nock Clifton's Data Access pattern catalogue [Noc03]. For example, the *Primed Cache* is a useful candidate since it provides a cache that is populated depending on the usage of the data or the *Demand Cache* which populates the cache lazily depending on the demand of the framework. In general *Independent Data Model* has similarities to a cache, but the main purpose of *Independent Data Model* is not easy of efficient access of the data, but the clean separation between the analyzer, the data consumer and the data model.

3.4 Multiple Outputs

3.4.1 Intent

Provides an analyzer with multiple data consumer that can easily be modified, removed or exchanged.

3.4.2 Motivation

As already mentioned, we first stored the data from the analysis in a SQL database and later switched to storing the data as Prolog facts. This switch required a lot of recoding, since the code responsible to store the data was nested in the analysis code. In order to avoid such a problem in the future we separated the data exporting part of the software from the analyzer. The classes that export the data will be referenced as data consumer in this pattern.

The separation between the analyzer and the data consumers ensured that we minimized recoding if we want to modify a data consumer. It also helps if the analyzer is modified. This separation is part of the *CAE* pattern. We also want to allow multiple data

consumers. For example, if we are storing the data in a SQL database and want to store them locally as Prolog facts.

Another feature we want is a built-in logging function. We needed that for our tools, since we measured the analysis time for the different projects in order to improve the performance of our tools. Of course, it would have been possible to dump that information into the usual logging files of the tool, but in our opinion it is a better choice to offer the user a separate logging mechanism for two reasons:

- It is easier to turn them off without affecting the overall analysis.
- Since the data sets are isolated it is easier to analyze them.

In conclusion we want a framework where:

1. Multiple data consumers are possible.
2. It is easy to add, delete or modify data consumer without modifying the analyzer.
3. It is easy to add additional logging information, independent from the general logging features of the framework.

This pattern describes a structure that meets these requirements.

3.4.3 Applicability

This pattern is applicable if it is important to easily add, modify the data consumer and/or multiple data consumers are needed.

3.4.4 Structure

Figure 3.5 shows an UML diagram for the *Multiple Outputs* pattern.

3.4.5 Participants

`AbstractDataConsumer` This abstract class is the base for all data consumers. It has an abstract `saveInfo(Database db)` method which starts the data consumption.

`ConcreteDataConsumer` It is a basic subclass of `AbstractDataConsumer`.

`ConsumerControl` The `ConsumerControl` class is able to manage the different consumers. It has methods for adding and removing consumers and one method which starts the consumers.

`Config` An object of a class that implements the `Config` interface is used for configuring a consumer. For example, most data consumers will need a target where they should store their results.

`Database` The `Database` class holds or has access to the data model of the analysis[1]. The `ConsumerControl` will forward the database objects to its registered consumers.

`MainClient` This package represents the main class of the framework using that pattern.

[1]If the *Independent Data Model* pattern is used, the `Database` will access that data model.

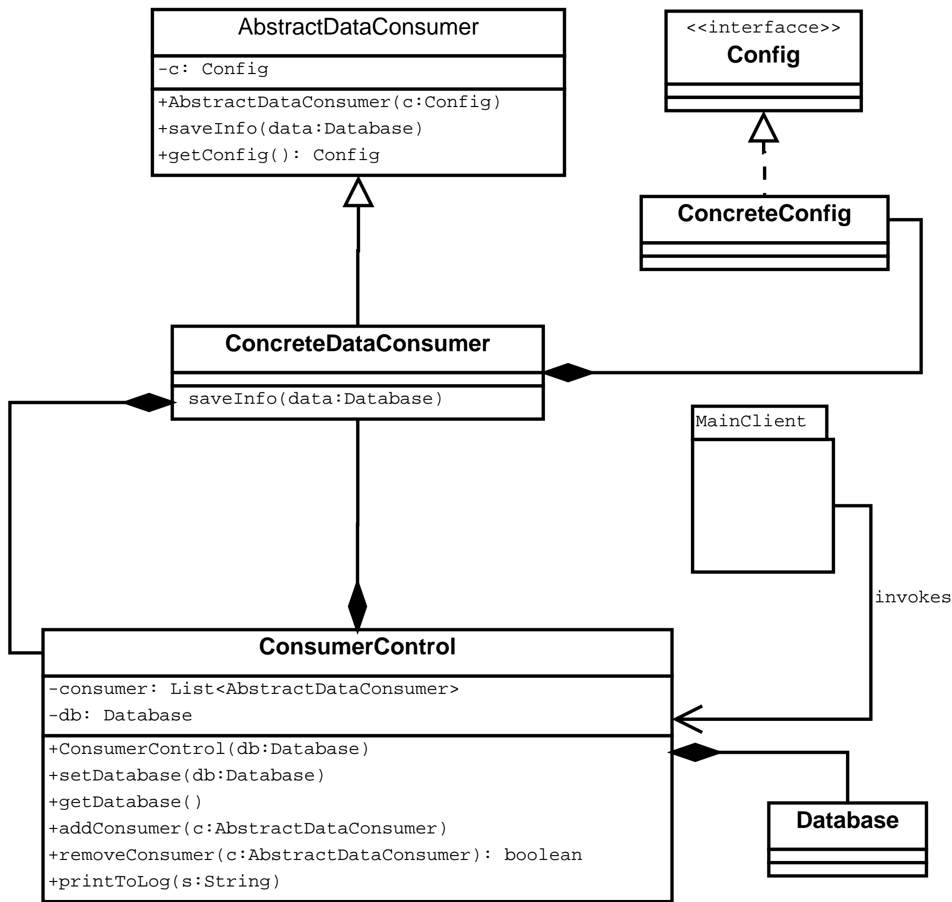

Figure 3.5: UML diagram for the *Multiple Outputs* pattern

3.4.6 Implementation

AbstractDataConsumer The code in Listing 3.5 shows a rather simple possible `DataConsumer` and an appropriate `Config`.

Listing 3.5: A consumer and its config class

```
1  public class MyPrinterConfig implements Config{
2  private File target= null;
3  public boolean append = null;
4   public MyPrinterConfig(File t, boolean a){
5    target = t;
6    append = a;
7   }
8   public File getTarget(){
9    return target;
10  }
11 }
12 public class MyPrinter implements AbstractDataConsumer{
```

```
13   public saveInfo(Database db){
14     String[] data = db.getDataSets();
15     File target = this.getConfig().getTarget();
16     if(!this.getConfig().append){
17       //clear File target
18     }
19     for(String d:data){
20       //add String d to target
21     }
22   }
23 }
```

The `MyPrinterConfig` class has two fields, a `File`, representing a local file, and a `boolean` that will indicate if the data sets are going to be appended to the file. When the `saveInfo()` method is called by the `ConsumerControl`, it forwards the current data model to the printer which reads all data and saves it in the file that is specified in the `MyPrinterConfig` object. In general, each class that realizes the `AbstractDataConsumer` interface will need a specific realization of the `Config` interface. If the realizations of `AbstractDataConsumer` and `Config` were small, like in the example above, it could make sense that both are realized by a single class.

ConsumerControl The following code shows a basic `ConsumerControl`.

Listing 3.6: The ConsumerControl

```
1  public class ConsumerControl{
2    private DataBase db = null;
3    private ConsumerControl(){
4    }
5    private Vector<AbstractDataConsumer> printers =
6      new Vector<AbstractDataConsumer>();
7
8    private static ConsumerControl instance =
9      null;
10   public Database getDatabase(){
11     return db;
12   }
13   public Database setDatabase(Database db){
14     this.db=db;
15   }
16   public static ConsumerControl getInstance(Database db){
17     if(instance==null){
18       instance = new ConsumerControl();
19       instance.setDatabase(db);
20       return instance;
21     }
22   }
23
24   public void addConsumer(AbstractDataConsumer dp){
25     printers.add(dp);
```

```
26   }
27
28   public boolean removeConsumer(AbstractDataConsumer dp){
29    return printers.remove(dp);
30   }
31
32   public void saveInfo(){
33    for(AbstractDataCnsumer dp:printers){
34     try{
35      dp.saveInfo(db);
36     }catch(Exception e){
37      System.err.println(e);
38      }
39    }
40   }
41   public void printToLog(String s){
42    //Write the String to some log file
43   }
44  }
```

Listing 3.6 shows a possible `ConsumerControl`. The code should be self-explanatory. Of course, the exception handling should be more sophisticated in a real program.

You may notice that the *Singleton* pattern was used. It is not absolutely necessary to take advantage of it. However, in our opinion it is advisable for the `ConsumerControl`, because we think that multiple `ConsumerControl` objects would only lead to confusion.

Database The `Database` represents or accesses the data model of the framework. This could be the data model of the used analyzer or an implementation of the *Independent Data Model* pattern. In both cases the `Database` acts as an interface for reading the data. If transformation methods are necessary, a suitable location for them will be the `Database` class. Of course, it would also make sense to add them to the `AbstractDataConsumer` which needs them. But that may cause a problem. If another `AbstractDataConsumer` also needs the same method you will either have a duplicate implementation of the method or a confusing cross-reference between two independent classes. So we propose to implement these formatting methods either in the `Database` class or in a designated class for such methods.

3.4.7 Related Patterns

We used the *Singleton* for the `ConsumerControl` class.

The core of this pattern, the various data consumers, can be seen as an implementation of the *Strategy* pattern.

Chapter 4

Migration Patterns

This chapter will present the patterns we identified for API migration. An overview over these patterns and their relationship can be found in Section 2.2.

4.1 Simple Mapping Language

4.1.1 Intent

Provide a mapping language for specifying a simple migration job.

4.1.2 Motivation

The purpose behind a mapping language is to provide the user with an effective way to describe the relations between the classes and methods of two related APIs.

We started the development of our migration framework with a very simple mapping language. It worked for our first test case, a mapping between `java.util.Vector` and `java.util.ArrayList`. However, as we tried to apply it to a more complex case[1], our language was lacking multiple necessary features. The *Mapping Language Extension* pattern tries to solve that problem. If we have a simple case, we benefit from a small mapping language. A simple mapping language has the following advantages (compared to a complex solution):

- The more compact mappings are more comprehensible to understand for a reader.
- It is easier to apply the language for other programming languages.
- The footprint of the mapping in memory is small.
- The structure of the mapping language is easier to enforce.

This pattern will give you a guideline how to setup such a language and a proposal for the structure of the parsing part of the framework.

[1] A mapping between two XML libraries.

4.1.3 Applicability

Mapping via a simple mapping language is reasonable for API migration if the APIs in question have a similar signature[2].

4.1.4 Structure

Mapping Structure

This section will discuss how to design such a language.

Resemblance to target language You have to consider that your mapping language should resemble an existing language. The main advantage is that the finished language is easier to read for someone who knows the existing language. A possible drawback is that it encourages the developer of the mapping to integrate more features of the language than necessary.

Keywords In addition to keywords originating from the target language you should also add keywords for the following entities:

The current target class instance Basically the equivalent to the `this` operator in Java. While it would be possible to use `this`, a new keyword simplifies the parsing of the mapping. For example, if a wrapper is created, the parser will have to decide whether the `this` operator is a reference to the wrappee or to the wrapper.

An indication that a mapping follows In order to differentiate between an usual class and a mapping.

Helper methods As long as there is exactly one method in the target API for each method in the original API and there is no need for non-trivial transformation of the method arguments, you can keep the language relatively simple. However, this scenario will not happen in most cases. If the target API is a newer version of the original API we will most likely have such a case which is realistic in a real-world example. But even then you may encounter some cases where it is not possible to map directly between all methods. So you will need a way to handle these methods which can not be simply replaced with another method in the target API. Usually you will encounter three cases (or combinations of these three).

- You have to perform non-trivial transformations on the method arguments.
- You have to invoke multiple methods.
- The target API misses that functionality.

In these cases you have to define a new method which simulates that functionality of the target method. We call these methods helper methods.

A possible solution which does not complicate the mapping language is to design a utility class where you add the missing methods and refer to them in the mapping.

[2]For example, two different version of the same library.

Sensible language constructs This section will discuss some language constructs which might be suitable candidates for including them in the mapping language.

Casts You might encounter mappings where method arguments do not belong to the same class, but are compatible. For example, if the target and the source method return a `LinkedList` and the signature of the target method only specifies that a `List` is returned a simple cast will be sufficient.

Simple Arithmetic It might be usefule to allow simple arithmetic operations like incrementing or decrementing a value.

Constructor Calls The ”new” keyword will be useful if there is a relation between a method in the original API and a constructor in the target API.

Default values It might be the case that the source API features a method and a simplified version of it while the target API only has the complex version. It is often possible to simulate the simplified version by using default values for the argument which are not needed by the simplified version.

Code Structure

Figure 4.1 shows an UML diagram for a possible framework that uses a mapping language.

4.1.5 Participants

`ClassMapping` Represents the mapping in memory. It consists of further `ClassMapping` objects, representing inner classes, and various instances of the `MethodMapping` class.

`MethodMapping` A `MethodMapping` represents a relation between a method in the source API and one in the target API.

`ArgumentMatching` An `ArgumentMatching` specifies the location of an argument from the source method in the argument list of the target method. The `additionalOptions` Field can be used for storing additional informations, for example, storing a cast.

`Call` A `Call` object represents a method call.

`Parser` The `Parser` package in the diagram is a placeholder for the code that parses the mapping files. Its purpose is to generate the `ClassMapping` objects.

Migrator The migrator package in the diagram is a placeholder for the API migration famework that uses the mapping, for example, an implementation of the *Wrapper Creator* pattern.

`MappingProvider` The `MappingProvider` calls the parser used by the framework and is the bridge between the `Migrator` and the `ClassMapping`. If *Factory* were used methods for creating the mapping classes, it could make sense to add these methods to the `MappingProvider` class.

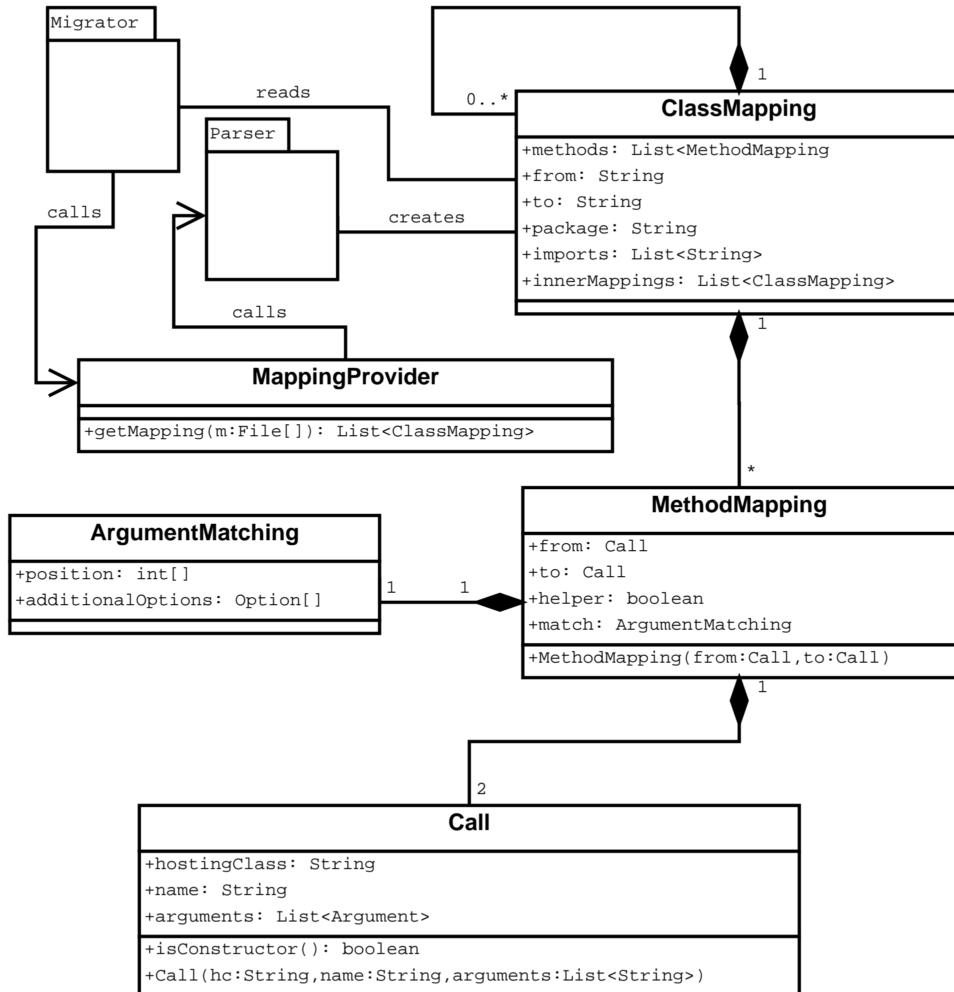

Figure 4.1: Parser for a mapping language

4.1.6 Implementation

Here we will discuss a possible mapping language for Java.

The following keywords are used.

`that` This keyword is a reference to the wrappee.

`util` We envisage that a utility class is used for helper methods. The `util` keyword is a placeholder for the qualified name of that class.

`map` This keyword precedes a mapping.

`to` This keyword is placed between the source and target class (or method).

The grammar for the header is shown in Figure 4.2. An example for a valid header is shown in listing 4.1. The header is similar to that of a Java class. You could consider to remove the import arguments because they could be pulled from the original file. However,

⟨MAPPING_FILE⟩→⟨JAVA_CLASS⟩|⟨MAPPING⟩
⟨MAPPING⟩→⟨HEADER⟩ '{' ⟨BODY⟩ '}'
⟨HEADER⟩→[⟨PACKAGE⟩] [⟨IMPORT⟩] ⟨UTILCLASS⟩ ⟨MAPSTART⟩
⟨PACKAGE⟩→'package' ⟨JAVA_QUALIFIEDNAME⟩ ';'
⟨JAVA_IMPORT⟩→⟨JAVA_QUALIFIEDNAME⟩ ['.*'] ';'
⟨IMPORT⟩→ 'import' ⟨JAVA_IMPORT⟩ [⟨IMPORT⟩]
⟨UTILCLASS⟩→'util' '=' ('null' | ⟨JAVA_QUALIFIEDNAME⟩)
⟨MAPSTART⟩→ 'map' ⟨JAVA_NAME⟩ 'to' ⟨JAVA_QUALIFIEDNAME⟩ '{'⟨MAPBODY⟩'}'

Figure 4.2: The header

Listing 4.1: Example of a header

```
1 package java.util.migration;
2 import java.util.*;
3 map Vector to ArrayList {
4 }
```

by fetching them from the APIs, we would most likely fetch unnecessary imports. Furthermore, we might need to import additional classes. The disadvantage is, that it makes the mapping longer. As it only affects the header, this effect is negligible.

Next is the type declaration. We dropped the visibility modifier as they can be pulled from the the source API, if they are needed. The `map` and `to` keywords specify that this is a mapping. It is not absolutely necessary, because a mapping could also be identified by the additional classname in the declaration. But we think that the improved readability is more important.

You will notice that we omitted generics. This was done to keep the example as simple as possible. In fact, the latest version of our mapping language allows generics. But allowing generics adds a lot of complexity, so you should carefully consider to support them or not.

Figure 4.3 shows the remaining part of the grammar (specifying the relation between the methods). The `MAPBODY` is similar to the body of a Java class declaration and may contain fields, methods and inner mappings. A method (or constructor) delegates between a single method in the source API and one method in the target API. We dropped again the visibility modifier and the return type of the method. Both can be extracted out of the API.

`UTIL_METHOD` is a call to a static method in an external class. Listing 4.2 is an example for a valid `MAPBODY` according to our grammar. The grammar was split in this section in order to improve the readability of this section. A complete version is listed in the appendix of this thesis (see Section 6.4).

⟨MAPBODY⟩→(⟨FIELD⟩|⟨METHOD⟩|⟨INNER_CLASS⟩|
⟨INNER_MAP⟩)[⟨MAPBODY⟩]
⟨FIELD⟩→⟨FIELD_DECLARATION⟩ ['='
(⟨DEFAULT_VALUE⟩|⟨UTIL_METHOD⟩)] ';'
⟨METHOD⟩→'map' ⟨JAVA_QUALIFIEDNAME⟩ '('
⟨PARAMETER_DECLARATION⟩')' 'to'
(⟨CONSTRUCTOR⟩|⟨METHOD_CALL⟩|⟨UTIL_METHOD⟩) ';'
⟨CONSTRUCTOR⟩→ 'new' ⟨JAVA_QUALIFIEDNAME⟩'
('⟨PARAMETER⟩')'
⟨UTIL_METHOD⟩→'util.' ⟨JAVA_NAME⟩'('⟨PARAMTER⟩')'

Figure 4.3: Classbody and methods

Listing 4.2: Example of a MAPBODY

```
1 Vector() to new ArrayList();
2 Vector(Collection c)
3  to new ArrayList(c);
4
5 add(Object o) to that.add(o);
6 add(int i, Object o) to that.add(i,o);
7 copyInto(Object[] target)
8  to Util.copyInto(that, target);
```

4.1.7 Related Patterns

It might be useful to use a *Factory* for populating the `ClassMapping` objects. Since the mapping language is prone to be extended and changed during its developement and usage, you might consider to use a flexible *Factory* pattern from the start. For example, *The Dynamic Factory Pattern* by Welicki et al. [WYWB08] [3]

.

Depending on the features of your framework you might consider a special data model for the mapping. For example, you could store them as Prolog facts in order to be able to use Prolog for querying the data. Two interesting patterns for storing the data are the *MetaData Container* pattern and the *MetaData Repository* pattern [GSF10].

4.2 Mapping Language Extension

4.2.1 Intent

Provide a mapping language which allows to specify a mapping for a migration scenario that is to complex for the *Simple Mapping Language* pattern.

[3] see 1.2 for more information.

4.2.2 Motivation

The mapping language we showed in the previous pattern is suitable for cases where source and target API are highly related. However, it will not be feasible if there is a certain amount of differences between source and target API.

We tried to use the mapping language presented in the previous pattern, for writing a mapping that resembles a XOM-JDOM wrapper [BCLS09]. Unfortunately, the mappings mostly consisted of calls to helper methods, the main reason for that was that most methods from the source API could not be replaced by a single method from the target API. Because of that we decided to allow to map to multiple Java statements, but enforced a specific structure for the statements.

Unfortunately, if it is allowed to map to multiple statements, it will be necessary to implement much more concepts of the target language. Due to time constraints we tried to find a solution which would allow us to reuse existing techniques. So, instead of writing a mapping language from scratch, we decided to expand the Java language. This had the following advantages:

- It is possible to reuse existing parsing technologies.
- It is easier to read a mapping for someone who is experienced with the target language.
- If the target language and the used parsing technology are updated, it wont be always necessary to update your extension. However, if you are not using an existing language and its tools, it will always be necessary.

Of course, there are also good reasons for not using an existing technology for parsing the code of the target language, for example, a lack of tools that fulfill your requirements. But for the Java language, where our focus on analysis and migration lies, are multiple good tools available. This pattern will focuses on such a scenario.

4.2.3 Applicability

This pattern is applicable if the migration scenario is too complex for a *Simple Mapping Language*.

4.2.4 Structure

Workflow for a call of a migrated method

Figure 4.4 shows an activity diagram representing the workflow of a call to a migrated method. The suggestions for building a language extension in the implementation section of this pattern are based on it. In that figure we mention mapping complexity. During the implementation of our tools we categorized the possible scenarios which may be encountered during a migration process. For this pattern we only care whether it is possible to simulate the source method with one call of the target method or not. If it is possible, we will have a mapping complexity less or equal to 2. A complete list of the possible scenarios and more detailed description of the mapping complexity is in the appendix of this thesis (see Section 6.5).

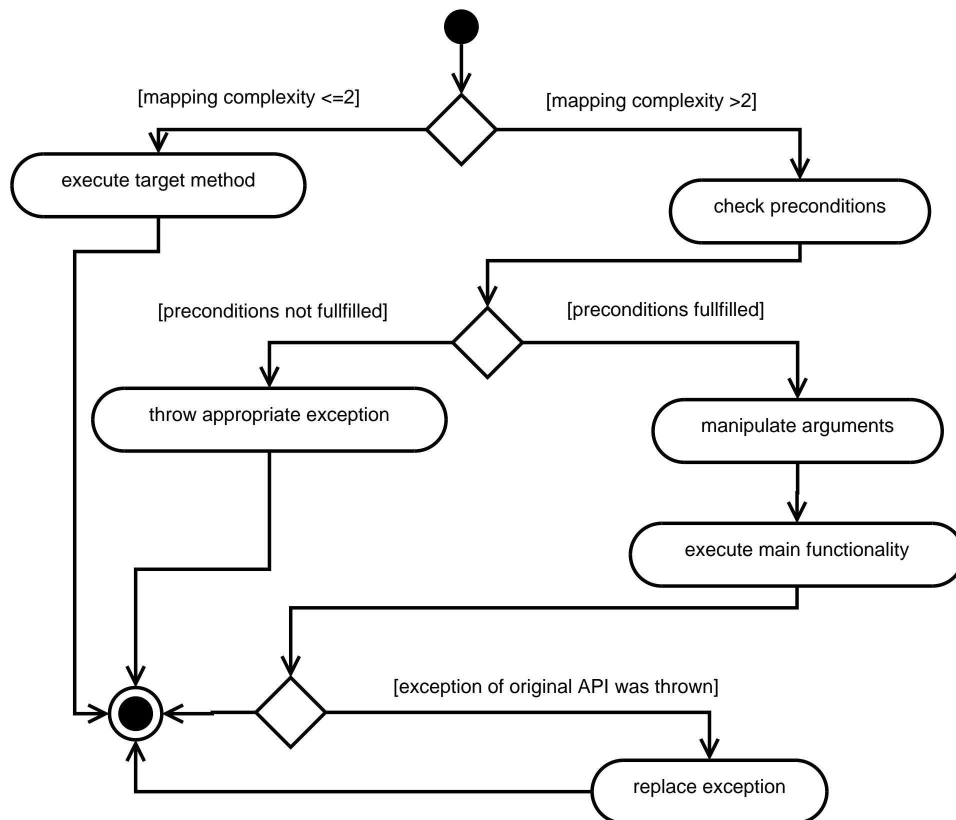

Figure 4.4: An activity diagram for a language extension

Structure of the framework

The UML diagram in Figure 4.5 represents a possible framework for using a mapping, based on a language extension that uses a preprocessor and does not modify the existing parsing tools directly.

4.2.5 Participants

`MainParser` This class steers the parsing process.

`Preprocessor` This class preprocesses the mapping and controls the `CoreParser`.

`CoreParser` The `CoreParser` is an external parser for the target language.

`Mapping` Represents the mapping in memory.

`DataConsumer` The `DataConsumer` represents the client that will use the mapping, an example would be a Wrapper Creator[4].

[4]Wrapper Creator Pattern (4.5)

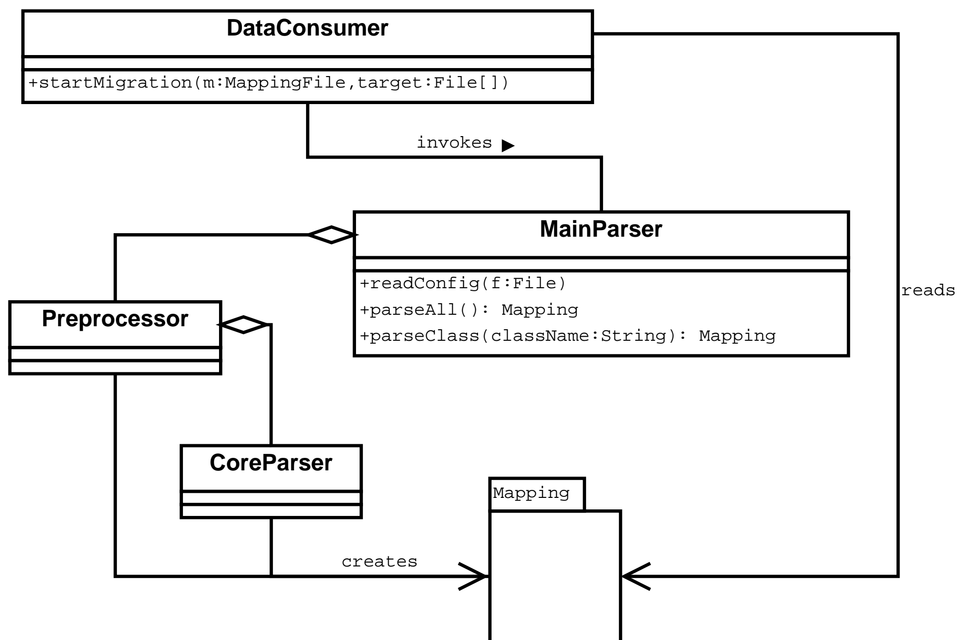

Figure 4.5: Structure of the framework

4.2.6 Implementation

The Parser

We already mentioned that we want to use an existing parser for most parts of the code, since this has the following advantages:

- You do not have to implement all features of the target language yourself.
- You are more inclined to add as few language constructs as possible.
- If the language gets an update, the chances are higher not to have to modify your preprocessor.
- You may be able to reuse the data model of the parser.

Designing the language extension

Figure 4.4 shows what should happen if a migrated method was called. The language extension should allow the user to configure the method so that the framework can create an appropriate method declaration. Furthermore, it should allow the user to set all necessary information.

The header of the class The only necessary change to the header would be the ability to express that it is a mapping and the target of the mapping (for example, if a wrapper is created, you will need the target class to determine the class of the wrappee object).

Listing 4.3: Example for a header

```
1 package nu.xom;
2 import java.net.URI;
3 import java.net.URISyntaxException;
4 import java.util.List;
5 public class Attribute extends Node to org.jdom.Attribute{
6  // Body of the mapping
7 }
```

This example was taken out of our mapping from XOM to JDOM. The only difference is the keyword `to` and the target class (`org.jdom.Attribute` in this case).

There are 2 cases which you must allow for a mapping language which can handle complex migration scenarios:

1. The target API does not implement all features of the source API. In that case you have to reimplement the features. Most of the time it will be limited to missing methods, but in the worst case you will have to implement multiple classes. Therefore, it should be allowed to add normal classes to the mapping. In order to clarify the purpose of the class to a reader. It might still be useful to classify it as a mapping, for example, by stating that the target is `null`.

2. The case that there are multiple target classes for a single source class. If you allow multiple target classes, you will also need a way to distinct between these classes. You can distinct them by their qualified name. But it could become cumbersome and you might consider to allow access them by a keyword and an identifier. For example, in the *Simple Mapping Language* pattern we used the keyword `that` as a reference to the instance of the target class. If we had multiple targets we could address them as `that(0),.., that(n)` where `n` is the amount of targets and the order is set by their declaration in the mapping.

Fields We did not change the fields, since we can not easily redirect from one field to another in Java.

Inner Classes Inner classes could either be a mapping or a normal class.

Methods This is where the bulk of the additions are applied. Take a look at the activity diagram in figure 4.4. First, there are two cases, depending on the mapping complexity. If the semantics of the original method can be expressed by a single method we will be able to map them directly.

Listing 4.4: Simple method mapping

```
1 public boolean addAll(Collection<? extends E> c)
2  to that.addAll(c);
```

The difference to the example in the *Simple Mapping Language* pattern is that we allow every method call as a mapping target, that means more complex statements are possible. Of course, the programmer could use this to avoid the constraints of being a simple method call. But, we are of the opinion that it is not feasible to prohibit it on a technical side, since some relaxations could prove quite useful.

Listing 4.5 shows an example for a complete method mapping for a non-trivial case.

Listing 4.5: Complex method mapping

```
1  public Attribute(String name, String value)  to {
2  prepare{
3   int index = name.indexOf(":");
4   String prefix = null;
5   String localName = name;
6   boolean prefixPresent = false;
7   if (index != -1) {
8    prefix = name.substring(0, index);
9    localName = name.substring(index + 1, name.length());
10  }
11 }
12 requires{
13  if (prefixPresent && prefix.equals("xml")) {
14   throw new NamespaceConflictException(
15   "creating xml: prefix attribute"+
16   " without proper namespace");
17  }
18  if (prefixPresent && prefix.equals("xmlns")) {
19   throw new IllegalNameException(
20    "creating xmlns attribute", name);
21   }
22  }
23  try {
24   that = new org.jdom.Attribute(localName, value);
25   } catch (org.jdom.IllegalDataException e) to {
26     throw new IllegalDataException(e, value);
27   } catch (org.jdom.IllegalNameException e)
28     to(e, localName:String) {
29    throw new IllegalNameException(e, localName);
30   }
31 }
```

Again we use the keyword `to` to indicate a mapping. Subsequently the method is split into three blocks.

Prepare Before the `requires` block is code which should be executed before preconditions are checked. The main reason for this block is to keep the preconditions as simple and readable as possible. The code inside this block should have no side effects, so it makes sense to copy the variables which are going to be modified. There is no need for a keyword since this part is defined by the location of the `requires` keyword.

Preconditions After the `prepare` block follows the preconditions, what is indicated by the keyword `requires`. If possible, these preconditions should be of the form **if** (**boolean** check) **throw new** exception. If there is a necessity for additional code, preceding the preconditions, it should be in the prepare block.

Run The remaining code is responsible for providing the actual features of the method.

Exceptions There is another necessary step during the call of a migrated method. If an exception is thrown from the target API and a wrapper is used for migration, the exception object will have to be wrapped. We realize this by introducing an exception mapping. If a catch clause is followed by the `to` keyword, an exception mapping will follow. In the example above, the first `to` is followed by the replacement code (the meaning of the second catch clause is explained later). If the migration approach does not require to change the exception, the preprocessor will simply discard the exception mappings.

As it may easily happen that the same exception mapping is used multiple times, it could make sense to specify these mappings globally (for example, on class or package level).

Listing 4.6: Exception mapping

```
1 OldException(int i) to{
2  //do
3  throw new MyException{}
4  }
5  OldException(String s) to MyException(s);
```

Listing 4.6 shows an example for a global exception mapping. It looks like a method call mapping, but it has no return value. Therefore, the parser can differentiate it from normal method calls or mappings (and from constructors via its name). The code after `to` has to be either another exception or a code block (which should result in throwing the correct exception). The example in Listing 4.7 shows how the mapping should be applied.

Listing 4.7: Applying the exception mapping

```
1  try{foo();}
2  catch(OldException e) to(e,int:i)}
3  try{foo2();}
4  catch(OldException e) to(e, String:s)}
5
6  //should become
7
8  try{foo();}
9  catch(OldException e) {throw new MyException{}}
10 try{foo2();}
11 catch(OldException e) {throw new MyException(s)}
```

Both `foo` methods may throw an `OldException`. The first argument inside the parentheses is always the variable containing the thrown exception. It is followed by an arbitrary amount of variables followed by a colon and the type of the variable. While this example does not use the variables you will often need runtime variables for creating the exceptions. You could consider dropping the type declaration for the variables besides the original exception. Unfortunately, this requires that your parser is able to identify the type of the variable, in order to replace it with the correct exception mapping. Also, you will encounter problems if you have to specify mappings which are using different but compatible variable types. Therefore, we think it is a better solution to specify the types directly. As we require that the first variable is the reference to the thrown exception, we do not need to specify its type again.

Keywords The following lines are a summary of the used keywords in this extension.

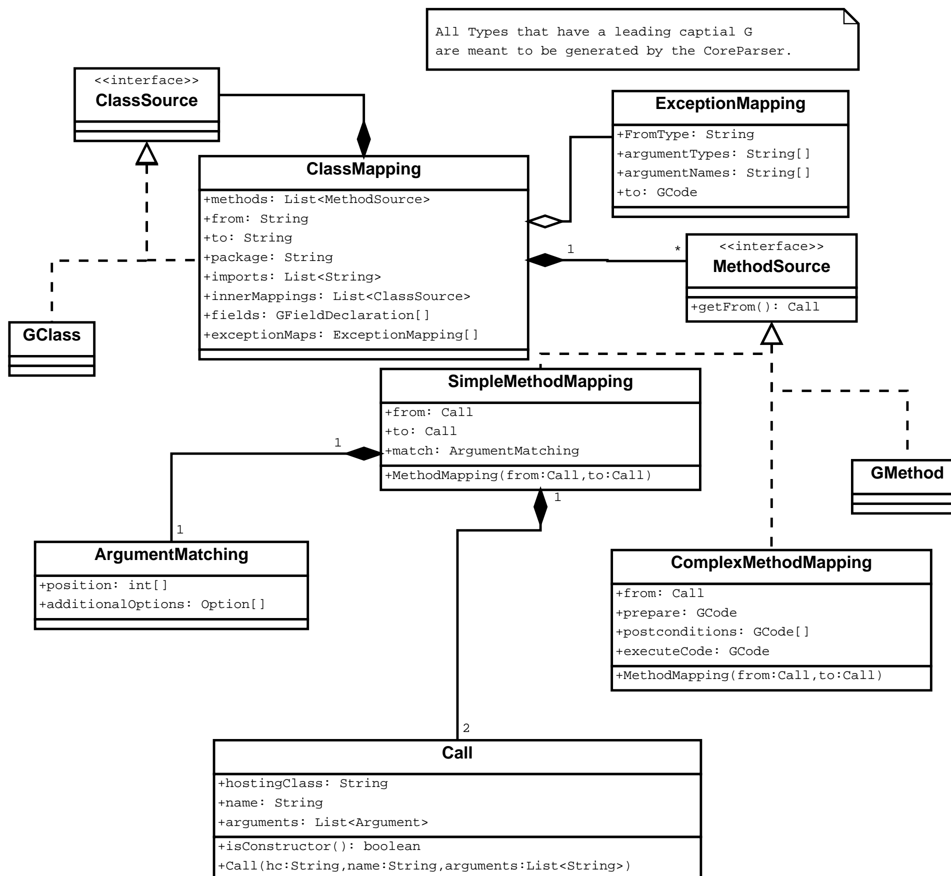

Figure 4.6: Example UML diagram for a mapping

that This keyword represents the wrappee. If multiple wrappees are allowed for a single class, you will have to be able to differentiate between the wrappees, for example, by putting an index behind `that`.

to This keyword indicates that a mapping is going to follow.

requires The `requires` keyword specifies the location of the three blocks (prepare, preconditions, run) of a migrated method.

In order to avoid name clashes it might be suitable to put a special sign before the keywords which are allowed in the language. For example, a $ sign would be reasonable for Java since it works fine for the compiler, but should not be used besides machine generated code according to Java naming conventions.

Mapping in memory Figure 4.6 shows a UML diagram which represents a possible structure for a mapping in the memory. The part which resembles simple mappings is based on the one we used in the *Simple Mapping Language* pattern (see Figure 4.1).

Additions are the `ExceptionMapping` class with represents an exception mapping (see Section 4.2.6), the `GClass` and `GMethod` classes which represents normal classes

and methods parsed with the `Coreparser` and the `ComplexMethodMapping` which resembles a method mapping according to this pattern.

4.2.7 Related Patterns

If the client, using the mapping has to keep the data in memory, you should prepare your implementation to have an efficent way to access the data and to store them. The paper "A pattern language for metadata-based frameworks" [GSF10] by Guerra et al. states some patterns capable of handling efficiently metadata. For example *Metadata Repository* would be a reasonable approach to store the mapping. These patterns are based on some of the patterns in the Data Access Patterns catalogue[5], interesting are the *Data Accessor*, *Demand Cache* and *Primed Cache* pattern (see the related works section 1.2) for more details on these patterns.

Another possibility for extending the Java language is the JastADD compiler ("The JastAdd extensible Java compiler" [EH07]) presented by Ekman, Torbjörn and Hedin, Görel.

4.3 Mapping Error Tracking

4.3.1 Intent

There is a high potential for errors or situations resulting from the migration process that should be forwarded to the user. The intent of this pattern is to add logging features, independent from existing logging in the software, which are generated out of a specification in the mapping language.

4.3.2 Motivation

The reasons we think that additional error/warning tracking capabilities are useful or even needed are:

1. While most software projects already have logging capabilities for warning and errors, we want to separate warnings and errors introduced by the migration from those that are already used in the software project.
2. We want to ensure that migrations specific warnings can be generated. For example, imagine that we have a method which converts a given `String`. The source API transforms the `String` even if it contains illegal characters, while the target API throws an error. The migrated version of that method would also have to accept illegal `String` arguments. In such a case it might be interesting to know how often an illegal `String` was passed to the migrated method.
3. We want to be able to track exactly the origin of an error introduced by the migration, since there is no guarantee to differentiate between errors correctly, thrown by the target method and errors thrown by precondition checks.
4. It may be the case that you need more information from runtime variables than the information visible through the caught exception.

[5]Data Access Patterns: Database Interactions in Object-Oriented Applications – [Noc03]

5. We want to ensure that all preconditions of a method will be checked even if one fails.

Listing 4.8 shows a simple method which illustrates some of our reasons above.

Listing 4.8: Simple Error Check

```
1  public String convertString(String arg1)
2   throws IllegalArgumentException{
3
4   if(arg1==null)
5    throw new IllegalArgumentException();
6
7   if(!syntaxCheck(arg1))
8    throw new IllegalArgumentException();
9
10  /*
11  Method code
12 */
13 }
```

The method takes a `String` as an argument. It performs some syntax checking on the `String` and tests if `String` is `null`. If it is not the case, an `IllegalArgumentException` will be thrown.

There are some cases to notice. First, if both errors happen only the first one will be seen since the following tests wont be performed.

Second, depending on the way the client forwards the error to the user, he might not be able to check whether the error was thrown due to a syntax check or `arg` was a `null` reference.

Third, we might be interested how often a specific value was given to that method, for example, if `arg` was the empty `String`. Another example, where it is useful to get additional information, would be if we had two `String` converters with slightly different specifications. Both are expecting a number. The old version accepts comma and dot as a separator for a decimal fraction, while the new one only accepts dots. It is relatively easy to modify the code on the fly, but you might be interested in how often this situation occurs. Therefore, we want to be able to specify warnings.

4.3.3 Applicability

This pattern will be applicable if more detailed error tracing capabilities are required for the migrated software.

4.3.4 Structure

Structure of an error Independent from how error checks are specified in the mapping, you must ensure the following:

- Each error has an unique ID.
- There is a way to specify additional information (usually the current state of some variables during runtime).

Structure of the Tracker

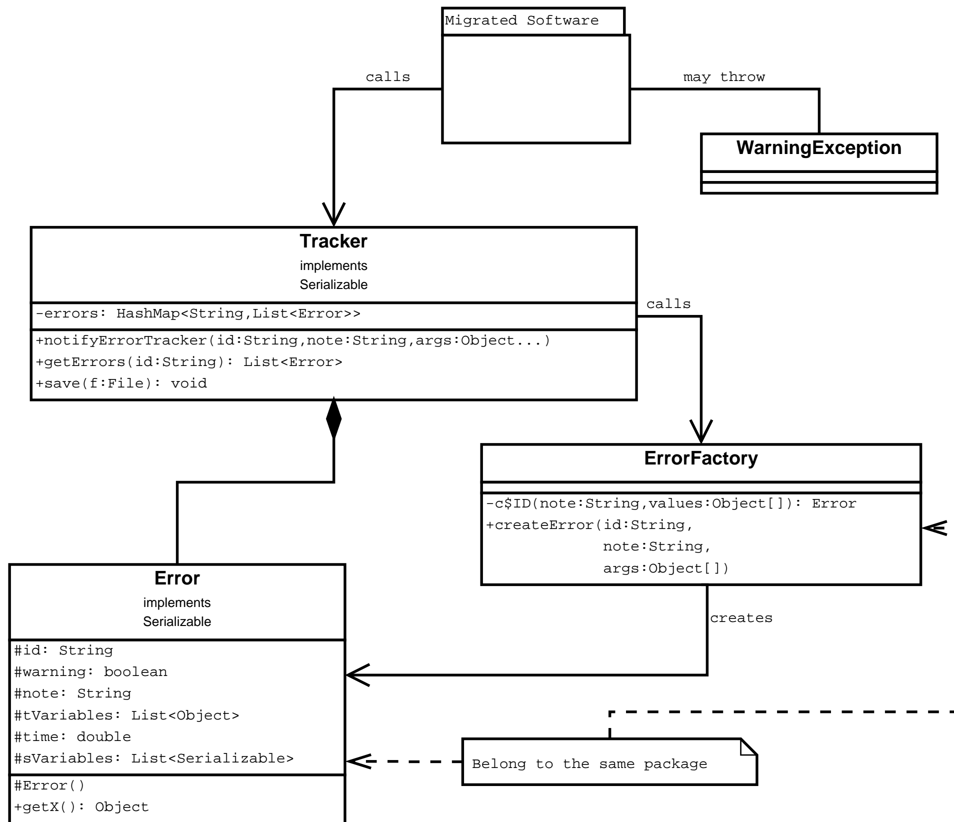

Figure 4.7: UML diagram for a tracker

Figure 4.7 displays a UML diagram for an error tracker.

4.3.5 Participants

Migrated Software This package represents a migrated software project which uses this pattern.

`Tracker` The `Tracker` is responsible for tracking the errors and warnings during runtime. It is called by the migrated software and calls the `ErrorFactory`. The warnings can be accessed by its getters or stored for later analysis.

`Error` An object of this class represents a warning/error that has occurred. It has the following fields (and appropriate getters):

- **`id`** The unique ID of the error (specified in the mapping file).
- **`warning`** A boolean indicating if this object is an error or only a warning.
- **`note`** Some additional description about the error.
- **`sVariables`** This list stores the important objects, regarding the error/warning, which can be serialized.

tVariables This list stores the important objects, regarding the error/warning, which can not be serialized. This field is transient.

time A timestamp when the error occurred.

ErrorFactory The `ErrorFactory` is responsible for creating the `Warning` objects.

WarningException A `WarningException` indicates a warning during the run of the program. This is explained in more detail in the implementation section. If warnings are not supported or if the `Tracker` is implemented differently, the `WarningException` will become obsolete.

4.3.6 Implementation

When using this pattern, you have to design/implement 3 parts:

- Extend a migration language so that it is possible to specify errors or warnings.
- Design how the migration software should implement calls to the tracker.
- Design the tracker library.

The first two items are explained in this section. The tracker library should be implemented according to the UML diagram shown in the structure section of this pattern.

Specification of errors and warnings

Listing 4.9 shows an example for a possible error specification.

Listing 4.9: Extended requires

```
1  public String convertString(String arg)
2   throws IllegalArgumentException{
3
4   requires(id1,"null",arg){
5    if(arg1==null)
6     throw new IllegalArgumentException();
7    }
8   }
9   requires(id2,"syntax check failed",arg){
10   this.syntaxCheck(arg);
11  }
12  notify(id3,"empty string"){
13   if(arg="") throw new WarningException();
14  }
15  /*
16  Method code
17 */
18 }
```

The example is related to the examples in the *Mapping Language Extension* pattern's implementation section. It features an extended `requires` clause.
Instead of one `requires` block, containing all preconditions, we have multiple `requires`

statements. Each `requires` statement represents a precondition and is followed by brackets with 2 or more arguments. The first argument is the ID representing the error, the second one is a `String` that may hold a note about the error and the last arguments are a variable number of Java statements which usually should be runtime variables. For our implementation to work it is necessary that an exception has to be thrown by the code inside the `requires` block if an error has to be caught.

The third statement, in the mapping above, is not a `requires` clause but a `notify` clause, which we use to indicate a warning. Instead of using a second keyword, you could also assign a level to each `requires` clause which would be useful if you need further distinction between the errors. You might need this, for example, if you have preconditions where you must not check the remaining preconditions, thereby using effectively three kind of errors.

The `WarningException` is not an exception from the migrated API, but it is needed for indicating that a warning should be saved by the `Tracker`, similar to the requirement for the `requires` block that an exception has to be thrown.

Notifying the tracker

Listing 4.10: Simple Error check extended

```
1  public String convertString(String arg)
2   throws IllegalArgumentException{
3
4   boolean id1$boolean = false;
5   Exception id1$exception = null;
6
7   boolean id2$boolean = false;
8   Exception id2$exception = null;
9
10 try{
11  if(arg==null)
12   throw new IllegalArgumentException();
13  }catch(Exception e){
14   id1$exception = e;
15   id1$boolean = true;
16   notifyErrorTracker(”id1”,arg1);
17  }
18 try{
19  this.syntaxCheck(arg);
20  }catch(Exception e){
21   id2$exception = e;
22   id2$boolean = true;
23   notifyErrorTracker(”id2”,arg);
24  }
25 try{
26  if(arg=””) throw new WarningException();
27 }catch(Exception e){
28  notifyErrorTracker(”id3”,arg);
29 }
```

```
30  if(id1$boolean) throw id1$exception;
31  if(id2$boolean) throw id2$exception;
32  /*
33  Method code
34  */
35  }
```

The code in listing 4.10 can be generated out of the mapping shown in Listing 4.9. First, a `boolean` and an `Exception` variable are generated for each `requires` clause in the mapping. After that follows a `try catch` block for each `requires` and `notify` clause. The code inside each `try` block is similar to the code inside the corresponding `requires` or `notify` block. Inside the `catch` block the exception is stored, the boolean variable is set to `true` and the `notifyErrorTracker` method is called (its arguments are the ID, the note and the statements specified in the mapping). In the case of the block originating from the `notify` block, we only notify the `Tracker`.

After the try catch blocks, we throw the caught exceptions again (of course, not the `WarningException`). This approach ensures that all preconditions were checked and recorded by the `Tracker`.

By ensuring that all checks are applied, you may change the semantics of the code. While these checks should be side-effect free, in a real world example it is quite possible that it is not the case.

Tracker and ErrorFactory

The `Tracker` and the `ErrorFactory` should be created automatically out of the mapping.

```
1  public class Tracker implements Serializable{
2   private HashMap errors <String,List<Error>> =
3    new HashMap<String,List<Error>>();
4   public void notifyErrorTracker(
5    String id, String note, Object... args){
6    List<Error> l = null;
7    if((l=errors.get(id))==null){
8      l=new LinkedList<Error>();
9      errors.put(id,l);
10   }
11   l.add(ErrorFactory.createError(id,note,args));
12  }
13
14  public boolean safe(File f){
15   //Store the tracker to f
16  }
17  public List<Error> getErrors(String id){
18   return errors.get(id);
19  }
20 }
```

Listing 4.3.6 shows a possible `ErrorTracker`. It has a `HashMap` containing the errors, the keys of the `HashMap` are the various errors' IDs. The `ErrorTracker` implements the `Serializable` interface for easy storage.

```
1  public class ErrorFactory{
2   public Error createError(
3    String id, String note, Object[] args){
4    if(id.equals("id1")){
5     return c$id1(note,args);
6    }
7    if(id.equals("id2")){
8     return c$id2(note,args);
9    }
10   if(id.equals("id3")){
11    return c$id3(note,args);
12   }
13   ...
14  }
15  private c$id1(String note, Object[] args){
16   Error ret = new Error();
17   ret.id = "id1";
18   ret.warning = false;
19   ret.note = note;
20   ret.tVariables = new LinkedList<Object>();
21   ret.sVariables = new LinkedList<Serializable>();
22   ret.time = System.currentTimeMillis();
23   for(Object o:args){
24    if(o instanceof Serializable)
25     ret.sVariables.add(o);else
26     ret.tVariables.add(o);
27   }
28  }
29  private c$id2(String note, Object[] args){
30  //Similar to c$id1, only with the correct id
31  }
32  private c$id3(String note, Object[] args){
33  //Similar to c$id1, only with the correct id
34  //and warning set to true
35  }
36 }
```

Listing 4.3.6 shows an `ErrorFactory` created out of the mapping in Listing 4.9. In our example we assumed that the `Tracker` can be stored by serializing it. The code in the `ErrorFactory` example simply separates the objects which implement the `Serializable` interface from those which do not implement it. If a user wants to change the error creation for a specific error, he can simply change the generated `c$ID` method. If you assume that this might often be the case, you might want to extend the mapping language further so that the user can specify the error creation method in the mapping.

4.3.7 Related Patterns

This pattern is designed as an extension to a mapping language. It is possible to add this capabilities to a simple mapping language, like that in the *Simple Mapping Language*

pattern. Yet, we do not suggest it since it would not fit with the premise that it is used for the simple cases. This pattern is more suitable for the cases presented in the *Mapping Language Extension* pattern.

The `WarningFactory` is an instance of the *Factory* pattern. The `Tracker` itself should be unique to the client, so the *Singleton* pattern is useful.

4.4 Generic Wrapper Recycler

4.4.1 Intent

Provide a class that handles the wrapper-wrappee relations.

4.4.2 Motivation

In a trivial environment you will generate exactly one wrapper for each wrappee and no other object has a reference to the wrappee. However, in a more realistic scenario, it might be the case that there are wrappees where it is unknown if a wrapper was already generated. Even if we know that a wrapper was generated at some point, we will need to get the corresponding wrapper.

Take a look at the code in Listing 4.11, which illustrates the problem which will appear if no *Generic Wrapper Recycler* is used.

Listing 4.11: An example illustrating the problem

```
1  Collection<Wrapper> c;
2  for(Wrapper w:c){
3   if(w.needsModification())
4    Factory.consumeWrappee(w.getWrappee());
5  }
6  //
7  //At a later point during the program executetion
8  //
9  TreeSet<Wrapper> c;
10 Collection<Wrappee> m =  Factory.getModifiedWrappees();
11 for(Wrappee we:m){
12  Wrapper wr = new Wrapper(we);
13  if(!c.contains(wr)) c.add(wr);
14 }
```

In Listing 4.11, we have a list of wrappers given to an external class which modifies the wrappees.

Afterward, the modified wrappees should be added to a list of wrappers. Some of the wrappees and their respective wrappers are already in that list. All wrappees that are not already in the list should be added. This will be done by creating a new wrapper for the wrappee and adding it to the list if `c.contains()` returns `false`.

The semantics of the code depends on the way how the equality of the wrappers is checked. If it was checked by the content of the wrappee (in other words the appropriate methods were overridden), it should work. Therefore, a wrapper, which is already inside the collection and the newly created wrapper, will delegate the method calls by the `TreeSet` to the appropriate method in the same wrappee. However, if equality is checked

by reference, it will not work as the reference of the old wrapper and the reference of the new wrapper are not identical, which will lead to the case where we have the same wrappee twice in the collection. If we were able to trace the existing wrapper from the wrappee, we would be able to use the existing wrapper. By doing so we could keep the semantics of the code.

The solution presented in this pattern is that we use a class which stores wrapper-wrappee relations, thereby allowing us to find an existing wrapper just by access to the wrappee.

With such a class available, our example could now looks like this:

Listing 4.12: An example

```
1 TreeSet<Wrapper> c;
2 Collection<Wrappee> m =  Factory.getModifiedWrappees();
3 for(Wrappee we:m){
4  Wrapper wr =
5   WrapperRecycler.getWrapperForWrappee(we);
6  if(!c.contains(wr)) c.add(wr);
7 }
```

Instead of generating a new wrapper, the `WrapperRecycler` is asked for the appropriate wrapper.

4.4.3 Applicability

This pattern is applicable if a wrapper is used and it might happen that the wrapper and the wrappee will become disconnected.

4.4.4 Structure

Figure 4.8 depicts the structure for a *Generic Wrapper Recycler*.

4.4.5 Participants

`WrapperRecycler` This class handles the wrapper-wrappee relations. It consists of the following:

- A `map` pointing from a class object to a `map` that contains the relation between wrappees and their respective wrappers.
- A method to get the correct wrapper for a wrappee.
- A method for binding a wrapper to a wrappee.
- A private method which creates a wrapper for a wrapperless wrappee.

`WrapperReference` The `WrapperReference` is either a weak or a strong reference to a `Wrapper`. The necessity is explained in the implementation section (see section 4.4.6). It consists of a constructor and several methods for changing the kind of reference.

`Wrapper` This interface is the contract for each wrapper.

`ConcreteWrapper` An example for an implementation of a `Wrapper`. It contains a static field. This field must be implemented by each wrapper.

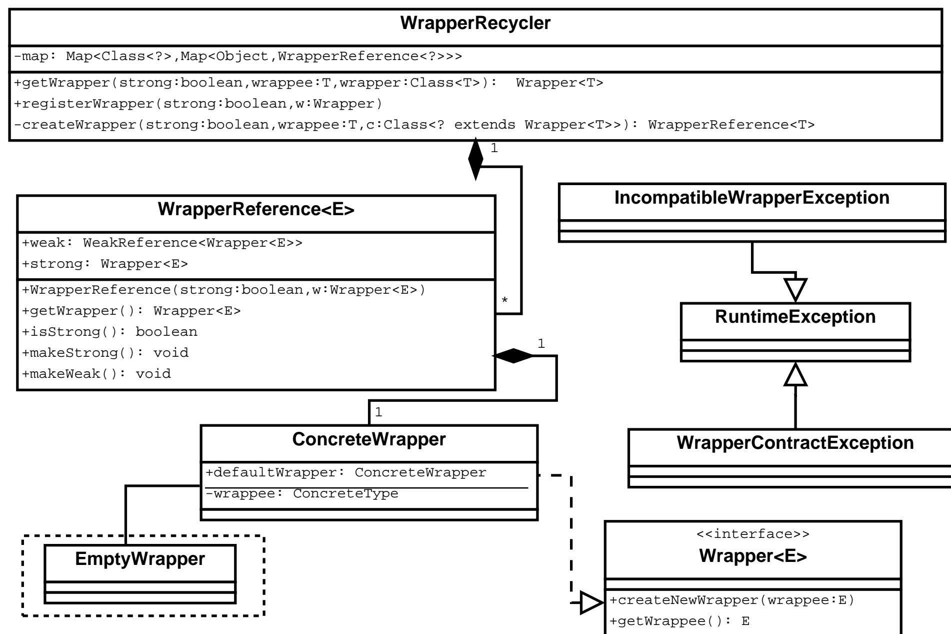

Figure 4.8: UML diagram of the *Generic Wrapper Recycler*

`WrapperContractException` A runtime exception which should be thrown if a wrapper broke the wrapper contract.

`IncompatibleWrapperException` A runtime exception which should be thrown if you tried to wrap a wrappee with an incompatible `Wrapper`.

`EmptyWrapper` The `EmptyWrapper` class is optional. We used it in our example as an easy way to ensure that a `DefaultWrapper` is created.

`Wrapper` The Wrapper interface serves as a contract for possible wrappers.

4.4.6 Implementation

Here we will show one concrete implementation of a `WrapperRecycler` and explain the design decisions in more detail.

First, we will talk about the structure of the `map`, storing the wrapper-wrappee relations. Here we split the list of all different wrappees by their respective classes.

Listing 4.13: The map

```
1  Map<Class<?>,Map<Object,WrapperReference<?>>> map = null;
2
3  private Map<Object,WrapperReference<?>>
4    createHashMapforClassIfNecessary(Class<?>c){
5      Map<Object, WrapperReference<?>> m = map.get(c);
6      if (m==null){
7        m =new WeakHashMap<Object, WrapperReference<?>>();
8        System.out.println("Map created for "+c);
```

```
9        map.put(c, m);
10     }
11     return m;
12   }
```

The private method should be called if the `WrapperRecycler` tried to obtain a wrapper. It returns the appropriate `map` for the class of the wrappee. If no map is stored for the class, a new one will be created and returned. It is important that a `WeakHashMap` is used. If we use a normal `Hashmap`, the Garbage Collector would not be allowed to delete the wrappee, because there is a reference to the wrappee in the wrapper and a reference to the wrapper in the `HashMap`. By using a `WeakHashMap` the Garbage Collector may delete the wrapper in question if there was no strong reference to it besides the weak reference in the `WrapperRecycler`.

Listing 4.14: The wrapper getter

```
1  public  <T> Wrapper<T> getWrapper (
2    boolean strong ,T wrappee , Class<? extends Wrapper<T>> c){
3   Map<Object , WrapperReference<?>> m =
4    createHashMapforClassIfNecessary(wrappee.getClass());
5   WrapperReference<T> ret = null;
6   ret= (WrapperReference<T>) m.get(wrappee);
7   if(ret==null){
8    ret = createWrapper(strong ,wrappee ,c);
9    m.put(wrappee , ret);
10  }
11  return  ret.getWrapper();
12 }
```

First, the `WrapperRecycler` accesses the appropriate `map` for the given class. Then it checks if there was already a wrapper for this wrappee. If a wrapper was found, it will be returned. Otherwise, it tries to create a new Wrapper, which will be returned and stored in the `map` for later access.

Listing 4.15: Creating a new Wrapper

```
1  private <T>WrapperReference<T> createWrapper (
2    boolean strong ,T o,Class<? extends Wrapper<T>> c){
3   WrapperReference<T> ret = null;
4   Field f = null;
5   try {
6    f =c.getDeclaredField("defaultWrapper");
7   } catch (SecurityException e) {
8    throw new WrapperContractException();
9   } catch (NoSuchFieldException e) {
10  throw new WrapperContractException();
11  }
12  Wrapper<T> defaultWrapper=null;
13  try {
14   defaultWrapper = (Wrapper<T>)f.get(null);
15  } catch (IllegalArgumentException e) {
16   throw new WrapperContractException();
```

```
17    } catch (IllegalAccessException e) {
18      throw new WrapperContractException();
19    }catch(ClassCastException e){
20      throw new IncompatibleWrapperException();
21    }
22    ret=new WrapperReference<T>(
23      strong, defaultWrapper.createNewWrapper(o));
24    return ret;
25    }
26  }
```

The code above is responsible for creating a new wrapper. It tries to get the static field `defaultWrapper` from the given `Wrapper` class. If it does not get the field, the wrapper will break the contract. If that happened, a `WrapperContractException` should be thrown. Unfortunately, with Java it is not possible to enforce the inclusion of that field or of a static method with the same purpose. We also can not use an abstract class for the wrapper contract because that would prohibit that a wrapper has another super class (at least if the programming language is Java). So we settled for the static field.

If the field is present, `createNewWrapper` will be invoked. `createNewWrapper` must be implemented due to the `Wrapper` interface, and the new wrapper will be returned.

A `ClassCastException` may occur. This will be the case if the given class is not compatible with `Wrapper<wrappee.class>`. In that case an exception will be thrown.

Listing 4.16: Register a Wrapper

```
1   public<T> boolean registerWrapper(
2       boolean strong, Wrapper<T> w){
3     T wrappee =w.getWrappee();
4     Map<Object, WrapperReference<?>>
5      m=createHashMapforClassIfNecessary(
6        wrappee.getClass());
7     WrapperReference<T> ret =
8       (WrapperReference<T>) m.get(wrappee);
9     if(ret==null){
10     m.put(wrappee, new WrapperReference<T>(strong,w));
11    }else{
12      if(ret.getWrapper()!=w)return false;
13    }
14    return true;
15  }
```

The code in Listing 4.16 stores a wrapper. If the wrapper is already stored or if none is registered for the wrappee, it will return `true`. If there already was a wrapper for the wrappee and it is different from the given wrapper, the method will return `false`.

The following example shows an implementation of the `WrapperReference`.

Listing 4.17: The Wrapper reference.

```
1   class WrapperReference<E> {
2    private WeakReference<Wrapper<E>> weak = null;
3    private Wrapper<E> strong=null;
```

```
4
5   public WrapperReference(Wrapper<E> e){
6    this(false ,e);
7   }
8   public WrapperReference(boolean strong ,Wrapper<E> e){
9    if(strong){
10     this.strong=e;
11   }else{
12     this.weak=new WeakReference<Wrapper<E>>(e);
13   }
14  }
15  public boolean isStrong(){
16   return strong!=null;
17  }
18  public void makeWeak(){
19   if(isStrong()){
20     weak = new WeakReference<Wrapper<E>>(strong);
21     strong=null;
22   }
23  }
24  public void makeStrong(){
25   if(!isStrong()){
26     strong = weak.get();
27     weak=null;
28   }
29  }
30  public Wrapper<E> getWrapper(){
31   Wrapper<E> ret = strong;
32   if(ret==null){
33     ret=weak.get();
34   }
35   return ret;
36  }
37 }
```

The `WeakReference` class is part of the JavaCore API. It serves as a wrapper for a reference. The difference between this and a usual reference (also called strong reference) is that the Java Garbage Collector ignores the `WeakReference` instances. That means if an object is only reachable through a `WeakReferences`, the Garbage Collector will be able to delete that object.

As mentioned before, we would basically eliminate the features of the Garbage Collector for all wrappers and for all wrapped objects if we would use only strong references. As soon as the references would be registered in the `WrapperRecycler` we will always have a strong reference to the wrapper and to the wrapper reference. We could of course create a method which deletes an object from the `map`, but consequently we would have to check or know when an `Object` is no longer needed. Furthermore, we would run into scaling issues as soon as the program uses a high enough amount of objects.

Usually it will be sufficient if a `WeakReference` is used, because the main program should always have a strong reference to the wrapper as long as it is needed. However, it may be the case that only a reference to the original wrappee exists, a wrapper was already

created and it is necessary to keep that exact wrapper. In that case a strong reference should be used. For example, the wrapper stores additional information about the wrappee. The class also features methods to change the kind of the used reference.

Listing 4.18: The Wrapper interface

```
1 public interface Wrapper<E> {
2  public abstract E getWrappee();
3  public abstract Wrapper<E> createNewWrapper(E e);
4 }
```

Listing 4.18 shows the contract for each `Wrapper` class. One method for creating a new wrapper and one which returns the wrappee.

Listing 4.19: A concrete Wrapper

```
1 public class ConcreteWrapper implements Wrapper<String> {
2
3   public static ConcreteWrapper defaultWrapper = null;
4   static {
5     defaultWrapper =
6      new ConcreteWrapper((EmptyWrapper)null);
7   }
8   private String wrappee = null;
9   ConcreteWrapper(String e){
10    wrappee = e;
11  }
12   private ConcreteWrapper(EmptyWrapper nill){
13   }
14
15   @Override
16   public Wrapper<String> createNewWrapper(String e) {
17     return new ConcreteWrapper(e);
18   }
19
20   @Override
21   public String getWrappee() {
22     return wrappee;
23   }
24 }
```

Figure 4.19 is an example for a concrete wrapper. In that case a wrapper for a `String`. Note, we only implemented the methods, essential for our `WrapperRecycler`. In order to be a complete wrapper we would have to implement the whole signature of the `String` class.

This wrapper is straight forward, it has a constructor which creates a new wrapper for a given `String`.

The purpose of the `defaultWrapper` is to gain access to the non-static `createNewWrapper()` method. In order to avoid type clashes we added a constructor which accepts an `EmptyWrapper` as an argument.

4.4.7 Related Patterns

It is advisable that the `WrapperRecycler` is unique to its client. Therefore the *Singleton* pattern is useful.

4.5 Wrapper Creator

4.5.1 Intent

Provide a basic framework for creating a wrapper automatically out of a mapping.

4.5.2 Motivation

The *Simple Mapping Language* and *Mapping Language Extension* pattern (see Section 4.1 & 4.2) should help to specify the relations between two APIs. However, we also want to put these to practical use. This pattern will describe a possible basic structure of a framework that creates a wrapper out of a provided mapping.

Advantages of a wrapper for migration are (in comparison to migration via code inlining):

- You do not have to inline the method code. That is beneficial because you can easily run into non-trivial problems.
- If you have a working wrapper, you will be able to use it by simply switching the libraries of the program which is going to use it. Of course, there are cases where you can not simply switch the libraries. For example, the project copied the source code of the API instead of using the provided libraries.

The disadvantages are:

- You may run into problems concerning object identity (a subset of this problem can be solved with a wrapper recycler [6]).
- The code gets longer as it features the calls to the original API and the wrapper code.

4.5.3 Applicability

This pattern will be applicable for automatically creating a wrapper if a mapping is present.

4.5.4 Structure

Figure 4.9 depicts a UML diagram of a possible *Wrapper Creator* according to this pattern.

4.5.5 Participants

`MappingReader` The purpose of the `MappingReader` is to parse the mappings and to store the data in memory.

The `MappingReader` contains one public method which starts the parsing and returns the mapping as an object.

[6] See Section 4.4 for the *Generic Wrapper Recycler* pattern.

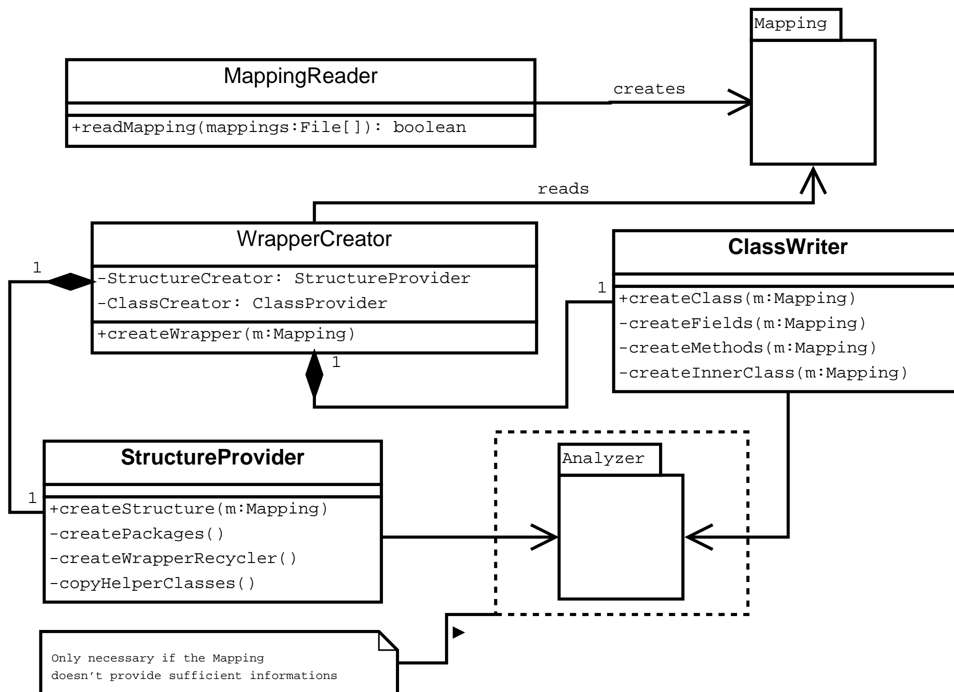

Figure 4.9: UML diagram of a WrapperCreator

`Mapping` The `Mapping` represents the mapping in memory.

`WrapperCreator` The `WrapperCreator` is responsible for creating the wrapper. However, it makes sense to split its features into the following classes

- **`StructureProvider`** The `StructureProvider` creates the file structure for your wrapper by creating the necessary directories and the creation of assisting classes (for example, creating an implementation of a *Generic Wrapper Recycler*).

- **`ClassProvider`** The `ClassProvider` creates the actual wrappers. It has a private method for creating the class header, the fields, the methods, and the inner classes.

`Analyzer` The `Analyzer` package represents a class (or a set of classes), capable to read the code structure of the original API. If the mapping has sufficient information this wont be necessary. For example, if a small mapping language is used or you are planning to remove the dependencies on the mapping later, you will need to use an analyzer to gain the signatures of the non-private classes and classmembers of the source API.

4.5.6 Implementation

Parsing the mapping

The *Simple Mapping Language* and *Mapping Language Extension* patterns feature some UML diagrams outlining a possible structure for the parser and some recommendations for parsing technologies.

The StructureProvider

The `StructureProvider` is responsible for creating the following entities:

The File Directories The `StructureProvider` should create the directories needed by the wrapper. That means the package hierarchy and directories for assisting classes. In the case of the Vector-Arraylist wrapper that would be a directory for the highest hierarchy, for example the `java.util` directory. Depending on the framework that creates the java files, it might also be useful to create empty files for the source-files at that point following.

Generic Wrapper Recycler If the *Generic Wrapper Recycler* pattern was applied, its files should be created by the `StructureProvider`, unless this is done via an imported project.

Mapping Error Tracking If the *Mapping Error Tracking* pattern was used, its necessary classes should also be created by the `StructureProvider`, with the exception of the `WarningFactory` which should be populated by the `ClassWriter`.

The ClassWriter

The `ClassWriter` is responsible for populating the packages (or classes) which were created by the `StructureProvider`.

Listing 4.20 shows an example for the header of a created wrapper.

Listing 4.20: An example wrapper (header)

```
1  public class Source implements Wrappee<Target>{
2
3   public static Source defaultWrapper = null;
4   static {
5      defaultWrapper = new Source((EmptyWrapper)null);
6    }
7
8   private Source(EmptyWrapper nill){
9   // only for default Wrapper}
10
11  private Target wrappee = null;
12
13  Source(Target t){
14    wrappee = t;
15  }
16
17  @Override
18  public Wrapper<Target> createNewWrapper(Target t) {
```

```
19     return new Source(t);
20   }
21
22   @Override
23   public Target getWrappee() {
24     return wrappee;
25   }
```

This code is independent from the mapping. Therefore, it could also be created by the `StructureProvider`. This possibility depends on the technology used by the `ClassWriter`. If the `ClassWriter` can not populate easily such files, it will make more sense that the `ClassWriter` creates the whole file.

Listing 4.21 shows an example for the body of a wrapper.

Listing 4.21: An example wrapper (body)

```
1  public static final int PROVIDED_CONSTANT = 0;
2
3  public Source(int i){
4   Target t = new Target(i);
5   wrappee = t;
6   registerWrapper(false, this);
7  }
8
9  public Source(String s) throws SourceException{
10  try{
11   Target.syntaxCheck(s);
12  }catch (TargetException te){
13   throw new SourceException(s);
14  }
15  Target t = new Target(s);
16  registerWrapper(false, this);
17  wrappee = t;
18 }
19 public String doStuff(String name, int value)
20  throws SourceException, IllegalArgumentException{
21  String prefix="";
22  if((name.length())>1)
23   prefix = name.substring(0,1);
24  if(name.length<3){
25   throw new SourceException();
26  }
27  if(value<0 || value >200){
28   throw new SourceException();
29  }
30  if(wrappee.check(name)==false)
31   throw new IllegalArgumentException();
32  if(prefix.equals{"$"}){
33   name = name.substring(1,name.length());
34  }
35  try{
```

```
36     return  wrappee.convert(name,(short)value);
37   }catch(TargetException t){
38     throw new SourceException(t.getValue());
39   }
40 }
```

It consists of two constructors and one method. Listing 4.22 is the appropriate mapping for this class. The main work of the `WrapperCreator` is to remove the new keywords, to replace calls to `that` with the appropriate call, to add calls to the `WrapperRecycler`, to remove name clashes and to add calls to the error tracker (if the *Mapping Error Tracking* pattern was used).

Listing 4.22: An example wrapper (mapping)

```
 1 public class Source to Target{
 2  public static final int PROVIDED_CONSTANT = 0;
 3  public Source(int i) to new Target(i);
 4  public Source(String s) throws SourceException to{
 5
 6  requires{
 7   try{
 8     Target.syntaxCheck(s);
 9   }catch (TargetException te){
10     throw new SourceException(s);
11   }
12  }
13   that = new Target(s);
14  }
15  public String doStuff(String name, int value)
16   throws SourceException, IllegalArgumentException to{
17  prepare{
18   String prefix="";
19   if((name.length())>1)
20     prefix = name.substring(0,1);
21  }
22 requires{
23  if(name.length<3){
24   throw new SourceException();
25  }
26  if(value<0 || value >200){
27   throw new SourceException();
28  }
29  if(wrappee.check(name)==false)
30   throw new IllegalArgumentException();
31
32  if(prefix.equals{"$"}){
33   name = name.substring(1,name.length());
34  }
35 }
36  try{
37   return  wrappee.convert(name,(short)value);
```

```
38     }catch(TargetException t){
39       throw new SoruceException(t.getValue());
40     }
41   }
42 }
```

4.5.7 Related Patterns

There is no need for multiple instances of the classes used in this pattern. So you should consider using the *Singleton* pattern. If you were using an AST for the mapping, the *Factory* pattern could make sense.

If you are using a wrapper for migration, you will have to consider the necessity of wrapper recycling. Therefore, it is highly advisable that a generic wrapper creator has the capability to create a basic implementation of the *Generic Wrapper Recycler* pattern.

If you are using the *Mapping Error Tracking* pattern it also will be the responsibility of the `WrapperCreator` to add the necessary classes and calls.

4.6 AST Migrator

4.6.1 Intent

Provide a technique for migrating an AST based structure independent to the transformation capabilities of the AST.

4.6.2 Motivation

This pattern originates from a problem that we encountered using the JDT. There are two ways to modify a JDT AST. Either manipulate the AST directly and the changes will be recorded by the JDT API in an `ASTRewrite` object, or create an empty `ASTRewrite` object and specify the modifications directly. In both cases you will encounter problems if you manipulate a method call and its arguments in one step. That problem forced us to handle the AST in a different way. It took more time than we were comfortable with to implement the original migration, notice the error and locate the reason of the error. So we tried to implement a solution that is as independent to the features of the AST as possible. Usually it is not recommendable to not use existing features of a used framework. However, in this case there are some interesting advantages:

- It is trivial to change the migrator during program execution. A possible scenario is where you want to apply a different migrator or no none at all for the childs of a node.
- This technique works for an immutable AST.
- If the AST is switched with the AST of another framework, it wont be necessary to learn the technique for modifying the AST provided by the framework.
- The resulting code is strongly structured and therefore should be quite comprehensible.

Our solution copies the tree and adjusts it while traversing it (very similar to a typical *Visitor*). Of course, it is necessary that the framework provides a way for the user to create AST nodes. We think that this requirement is fulfilled by most used ASTs.

However, there are some disadvantages that result from the fact that the whole tree is copied:

- The code is usually longer because it is necessary implement a copy method for each possible AST node.
- If the the AST received an update, it could be that you need to update code for several AST nodes.
- You may encounter performance problems.

4.6.3 Structure

Figure 4.10 shows an UML diagram for an `ASTMigrator`.

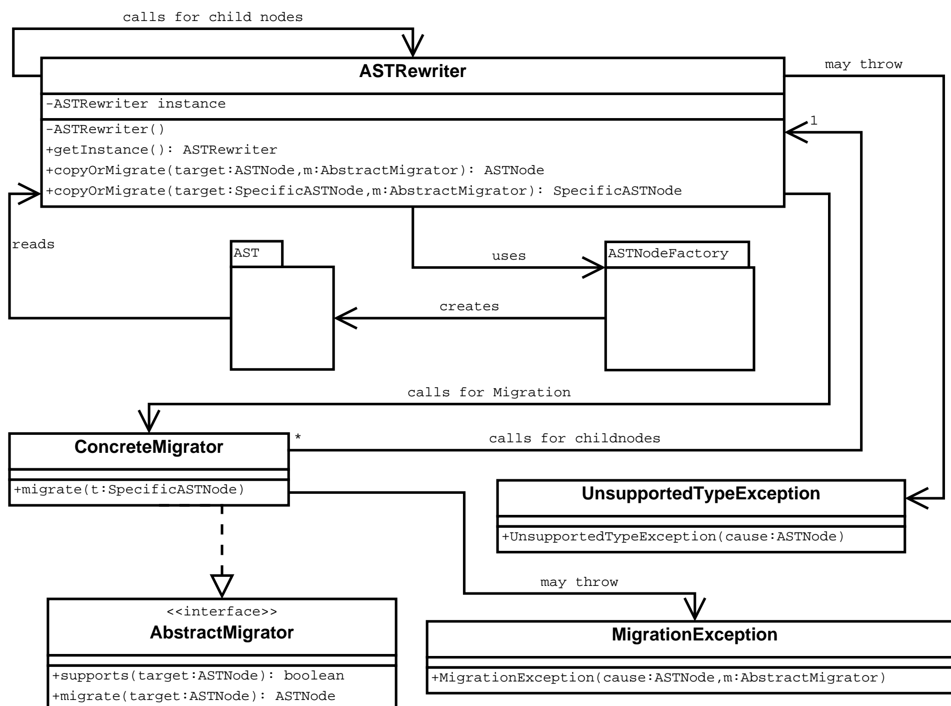

Figure 4.10: UML diagram for an ASTMigrator

4.6.4 Applicability

This pattern will be applicable if you have an AST that offers constructors or *Factory* methods and you do not want to rely on the transformation features provided by the framework of the AST (or the AST does not provide such feature).

It wont be applicable (or at least not recommendable) if performance is an important factor and you have large trees. Furthermore, if you are sure that the AST is not changed

and the frameworks already offers highly sophisticated transformation methods, you will have to consider carefully the application of this pattern.

4.6.5 Participants

AST This package represents the AST.

ASTRewriter This class is responsible for creating the new AST. The method `copyOrMigrate(target ASTNode, m AbstractMigrator)` delegates the `ASTRewriter` to the method which handles the specific `ASTNode`. Then it will check if the object is supposed to be migrated or copied. For the first case, the `ASTRewriter` will call the `ConcreteMigrator`. For the second case, the node is copied and `copyOrMigrate` is called for the child nodes.

ASTNodeFactory It resembles a *Factory* which creates the individual `ASTNode` objects.

AbstractMigrator This is the interface which must be implemented by a migrator. It has two methods:

supports(ASTNode n) This method checks if the migrator supports `n`.

migrate(ASTNode target) This method will delegate to the correct migrate method for the specific AST node.

ConcreteMigrator This class realizes the `AbstractMigrator` interface. Besides the methods required due to the the interface, it also contains the methods which will migrate the specific AST nodes.

UnsupportedTypeException Such an exception should be thrown if a `copyOrMigrate` method was called with an unknown `ASTNode`.

MigrationException A `MigrationException` will be thrown if something went wrong during the migration step.

4.6.6 Implementation

The following lines will show an example implementation for this pattern and explain it in more detail.

Copy capabilities

Listing 4.23: Skeleton of main copyOrMigrate

```
1 public ASTNode copyOrMigrate(
2   ASTNode target, AbstractMigrator m){
3  if (target instanceOf SpecificASTNode1){
4   return copyOrMigrate((SpecificASTNode1)target, m);
5  ...
6  if (target instanceOf SpecificASTNodeN){
7   return copyOrMigrate((SpecificASTNodeN)target, m);
8  }
9 }
```

The code above shows the Java code for the main `copyOrMigrate` method. It checks the type of the target argument and calls the `copyOrMigrate` method for that specific type.

Listing 4.24: Skeleton of specific copyOrMigrate

```
1  public ASTNode copyOrMigrate(
2   SpecificASTNode target, AbstractMigrator m){
3   if(m.supports(target)) return m.migrate(target) else{
4    /*
5    for(childNodes c){
6    copyOrMigrate(c,m);
7   }
8    */
9    return ASTNodeFactory.createSpecificASTNode(arguments);
10  }
11 }
```

An example for a specific `copyOrMigrate` method. First, it is checked whether the migrator class supports that specific `ASTNode` (this may also depend on some fields of the node). If it is supported, the migrator will migrate it. Otherwise, `copyOrMigrate` is called for all child nodes of `target` and the node is copied (in this example, this is invoked by the call to the `ASTNodeFactory`). If you are going to apply a different migrator for the child-nodes (or none at all), this can be achieved easily by using the new migrator for the child-nodes. If you are just overwriting the `ASTRewriter`, you will have to use a flag variable for the `ASTRewriter` or a similar approach.

The code snippet in Listing 4.25 shows an optional but useful migrator, its `support` method always returns `false`. Therefore, its sole purpose is to ensure that all nodes are copied. If you do not use such a `NullMigrator` and the migrator variable is set to `null`, it will be necessary to add a `null` check to every specific `copyOrMigrate` method of the `ASTRewriter`.

Listing 4.25: Null Migrator

```
1  public static class NullMigrator
2    implements AbstractMigrator{
3   public NullMigrator(){}
4   public boolean supports(ASTNode target){
5    return false;
6   }
7   public ASTNode migrate(ASTNode target){
8    throw new UnsupportedOperationException();
9   }
10 }
```

Migration capability A realisation of the `AbstractMigrator` interface has a structure similar to the `ASTRewriter` structure. One method (which is enforced due to the interface) that delegates to the migrate method for the specific ASTNode type, the methods for the specific ASTNode types and one method which provides a fast check if the node should be migrated. The following code shows an example for such class.

Listing 4.26: Pseudo code example for an AbstractMigrator

```
1  public class MyMigrator implements AbstractMigrator{
2   public boolean supports(ASTNode node){
3    if (/*node is a methodcall and is name is oldDo*/)
4     return true;
5   return false;
6   }
7   public ASTNode migrate(ASTNode target){
8    if (target instanceof methodcall)
9     return migrate((methodcall)target);
10   throw new SupportTypeClashException(target, this);
11  }
12  public methodcall migrate(methodcall target){
13
14 //It is already checked that the name is oldDo
15   String methodname = "newDo";
16   Arguments[] args =
17    new Arguments[target.getArguments.size()];
18   for(int i=0;i<args.length;i++){
19    args[i]=copyOrMigrate(
20     ASTRewriter.getInstance().
21      copyOrMigrate(target.getArguments()[i],this));
22
23    return ASTNodeFactory.createMethodCall(methodname,args);
24   }else throw new MigrationException(target, this);
25  }
26 }
```

The code above shows a simple migrator which purpose is to rename all method calls `oldDo` to `newDo`. The `supports` method only returns `true` if the `ASTNode` is an instance of the `methodcall` class and its name field is equal to `oldDo`. The main migrate method delegates to the migrate method for `methodcall` objects. If `migrate` is called with an `ASTNode` type different from `methodcall`, it will throw a `Migration-Exception`.

The specific migrate method migrates the name and delegates the nodes in the arguments back to the `ASTRewriter`. Afterward, the `ASTNodeFactory` is called for creating a new `methodcall` which is returned.

4.6.7 Related Patterns

It makes sense to implement the `ASTRewriter` as a *Singleton*.

For creating the ASTNodes a *Factory* is proposed, it may be useful to use a more flexible variant for the `ASTNodeFactory` since it must be updated whenever the underlying AST structure changes. For example, the *Dynamic Factory* pattern by Welicki et al. [7] (see Section 1.2 for more information).

This pattern is very close to an implementation of a *Visitor*. But we do not implement the migration functionality by specialized visitors but by replacing the appropriate methods. You can still use it like a regular *Visitor* by removing the calls to the migrators

[7] The Dynamic Factory Pattern – [WYWB08]

and subclassing the `ASTRewriter`. But we think that our implementation has a higher readability. Furthermore, it is easy to switch the migrator on the fly.

Bringert et al. specified a pattern with similar capabilities ("A pattern for almost compositional functions" [BR06]), but for a general purpose.

The AST's you encounter might become quite big. It might be useful to supply tool support for visualizing the AST. One such tool is VAST by Martinez et al. ("VAST – Visualization of Abstract Syntax Trees within Language Processors" [AMUFVI08]).

Chapter 5

Observations and Conclusions

This chapter summarizes our observations and conclusions regarding this thesis. It will summarize our thoughts and impressions for the patterns presented in Chapter 3 and 4. Possible future work is discussed in Section 5.3.

5.1 API analysis patterns

Configure - Analyze - Export provides a base for the structure of a flexible static analyzer which should be easy to extend and maintain as it groups the components of the framework in a sensible way. While this pattern is simple, it is easily the most important one of the analysis patterns. The programmer does not have to follow the pattern extremely close, but it is very important that he considers carefully how he should separate the components of his framework.

Variable Analyzer ensures that the framework is easy to configure, that the configuration is independent from the other components and that it is easy to extend. This pattern provides assistance to the *Configure - Analyze - Export* pattern. The *Variable Analyzer* seems not to be so important, but in our opinion this is not correct. If a user wants to utilize a new framework, first he will have to understand how it is configured. In our experience it is very likely that the configuration of the framework gets constantly refined during its implementation and usage. So, providing a solid configuration process is very important.

Independent Data Model separates the analyzer from the data model and thereby increases the advantages gained by the *Configure - Analyze - Export* pattern and simplifies the application of the *Multiple Outputs* pattern. Its necessity is strongly influenced by the possibility that the analyzer framework is changed and by the quality of the analyzer's data model. An actual need for it should arise in fewer cases than with the other patterns, but its benefits should be enormous if it happened. In addition, it would be a cumbersome task to apply this pattern later since the data model is used by most parts of the software.

The *Multiple Outputs* pattern ensures that it easy to add new ways to consume the data of the analyzer and adds the ability to use multiple data consumers. It is, in our opinion, not as essential as the other three patterns, since it may easily be the case that the user is sure that only one data consumer is used. But, even if only one consumer was used, it could be advisable to use a structure similar to the one proposed by the *Multiple Outputs* pattern, since it also simplifies the modification or a complete exchange of the consumer.

These four patterns represent the knowledge we got during the implementation and usage of our API analysis tools. The analysis patterns provide a base for implementing a

static API analyzer and they may also provide help for implementing tools for dynamical API analysis. If these four patterns are used for an analyzer, or at least *CAE*, *Independent Data Model* and *Variable Analyzer*, the resulting framework should be suitable for most analysis tasks and easy to extend and maintain.

5.2 API migration patterns

The *Simple Mapping Language* and *Mapping Language Extension* patterns provide the user with a basis for defining a suitable mapping language and software structure for his migration framework. *Simple Mapping Language* is more suitable for simple scenarios, while the *Mapping Language Extension* pattern will be recommendable if the differences between source and target API are too big. If the target API is just a different version of the source API, *Simple Mapping Language* will be sufficient in most cases. Both patterns are rather open. We expect that these patterns have to be refined during further work in that field (what already happened since the the *Mapping Language Extension* pattern can be seen as a specialization of the *Simple Mapping Language* pattern). But we think that they are sufficient so far as a first version or as a foundation for a mapping language.

The *Wrapper Creator* pattern shows one way how an existing mapping could be used to migrate an API. We decided for the ”migration by wrapper” technique as we have more experience and better examples with it. But the structure of a framework that uses a different technique should have many similarities to the structure of an implementation of the *Wrapper Creator* pattern.

The *Generic Wrapper Recycler* pattern helps to counter the identity problem that will appear if a wrapper is used. The structure of this pattern is a little bloated which results mostly from properties of the Java language. It may have been a better solution to present a more generalized version in this thesis. However, a user who wants to implement a *Generic Wrapper Recycler* for another programming language should not be distracted since we already explained the design decisions in detail and the resulting pattern will be more useful if it is used in a Java scenario.

The *Mapping Error Tracking* pattern can be applied to get a better understanding of errors and problems that occur during the execution of a migrated software. If API migration becomes more common and well-understood, it will be a question whether this feature will be necessary or not. At the moment we consider *Mapping Error Tracking* to be quite useful as it prevents that the programmer later on recognizes that he could use such a feature and implements it in a way that decreases the quality of the code. Since the error tracking capabilities are encoded in the mapping, it will be easy to ignore them for the migration framework if they are no longer needed.

The *AST Migrator* pattern shows a way how to migrate an AST with as few as possible dependencies on the transformation capabilities of the AST. It may be that there is a more efficient process for a specific AST migration scenario or that the transformation capabilities of the framework work are especially good. But it was also our intention for that pattern to animate the user to consider carefully if he wants to use the transformation capabilities provided by the AST.

These API migration patterns should provide assistance for implementing an API migration framework suitable for the user's requirements. In contrary to the API analysis patterns the API migration patterns are not intended to be used as a whole. The API migration patterns try to provide solutions for specific problems in the API migration field. Furthermore, they are more open than the analysis patterns. That means that the patterns most likely need to be adapted while further experience with API migration is gained.

5.3 Future work

This thesis is the result of two active projects. We are confident that our analysis tools are not going to be heavily modified in the near future, except some polishing and maintenance. The next interesting question from a technical point of view would be how to handle effectively the data generated by the analysis. The *Multiple Outputs* pattern allows multiple ways to use the data. But we still have to recognize the advantages and disadvantages of the different ways to save the data sets. Now, as we have the necessary tools we have to learn how to configure them best. Of course, this depends on the actual research question that we wish to answer with an analysis. While we have some ideas, it would be necessary to apply case studies to support these ideas. Another interesting aspect would be to adapt the patterns for dynamical analysis. We think that the solutions presented in this patterns are also helpful for dynamical analysis, but currently we lack experience in that field. Another point would be the user interface. All patterns in this thesis are about internal processes and configuration and besides some advices in the *Variable Analyzer* pattern, we do not consider interaction with the user. This will be even more important if this work is adapted for dynamical analysis.

Another aspect that could be researched, would be patterns for using the transformation capabilities of the analyzer. All three of our used analyzers have code transformation features (which are invoked in different ways). It would be helpful to have a code structure which also allows easy access for the user to the transformation features. It would also be helpful if this access was disconnected from the analyzer, so that the analyzer can be exchanged easily (similar to the disconnection between the analyzer and the data model, provided by the *Independent Data Model* pattern). Furthermore, such a structure could be useful for an API migration framework, too.

The patterns for API migration are more open, respectively more problem-oriented. Since API migration is still an active field of research it is likely that there are more common problems which could be handled with patterns which are similar to the *Generic Wrapper Recycler* pattern. By pushing the API migration tools and techniques further or by analyzing data on API usage, it is likely that more such problems can be classified. So, improving our understanding of the API usage, specifying problems that appear during API migration and finding approaches to handle these problems would be a possible future goal. Furthermore, it is likely that more experience with API migration could be used to improve the *Simple Mapping Language* and *Mapping Language Extension* patterns.

At the moment we intensively push the ”migration by wrapper approach”, due to the difficulties from code-inlining. However, with better understanding the migration scenarios it might be possible to develop better inlining techniques. In addition to getting a better understanding on API usage, it would be interesting to make case studies on scenarios where API migration was applied.

Chapter 6

Appendix

6.1 Tool Introduction

The following sections will present the three major frameworks which were used for our tools.

Each section will give you a brief introduction to the tool, mention some interesting properties and a code example for a simple analyzer which analyzes a single class and exports the classes classname, declared fields and the signature of its declared methods.

The following class is used as an analysis example:

Listing 6.1: A simple class for demonstrating the capabilities of an analyzer frameworks

```
1  public class AnalyzeMe {
2   public final String CONST = "const";
3   private static int temp = 0;
4   public static void main(String[] args) {
5   }
6   public AnalyzeMe(){
7   }
8   public void foo(){
9   }
10  public String foo(String s){
11   return s;
12  }
13  public void foo(int i){
14  }
15 }
```

6.1.1 Fundamental differences between analyzers

The presented tools either analyze the source code or the bytecode of the target file(s). This leads to some interesting differences between the analysis' results.

This is partially caused by the fact that most compilers do modify the code during compilation. While the compiler keeps the semantics, it often changes the syntax. Usually,

either for increasing the performance[1] or to transform it[2]. So, a source code based analyzer would see a method call which was removed during compilation while a bytecode based compiler would not see it (for example, unreachable code is often removed during compilation). Another possible cause for different results is explained below. Take a look at the following example:

Listing 6.2: Compiler preprocessing

```
1 public class MyClass{
2  public String d=doStuff();
3  public MyClass(String s){
4   Systen.out.println(s+doStuff());
5  }
6  public MyClass(){
7  }
8 }
```

If we would analyze this code snippet in order to know how often the programmer used `doStuff()`, we would expect a result of 2. However, if we use ASM to analyze this code snippet, we will get the result of 3. The reason for that is, whenever a `MyClass` object is created, all initializers of `MyClass` are called. Therefore `doStuff()` is called two times in the first constructor and one time during the second constructor. In other words, a bytecode analyzer sees how often a method is called in the code, while a source code analyzer sees how often a method call is declared. This change is similar to changes like removing generics, but it is not as obvious.

6.1.2 ASM

Introduction to ASM ASM [Eri] is a bytecode manipulation and analysis framework. One advantage of ASM is that it only needs the bytecode of the classes which you want to analyze. The other presented frameworks do require all dependencies of the project which is going to be analyzed.

Tool Requirements As already mentioned above, ASM only needs the bytecode of its targets. However, sometimes you need bytecode created with debug informations (for example, you need debug information for analyzing local variables).

ASM does not need further third-party APIs (respectively, they are provided in the ASM library).

Code example There are two ways to access the data generated by ASM, either by overwriting the various ASM listener which are called during the analysis or by reading the fields of the generated objects. The following code shows a simple analyzer:

Listing 6.3: A simple ASM analyzer

```
1 public static void printMethodSignatures(InputStream fis)
2   throws IOException{
```

[1] For example, by removing redundant reads or unreachable code.

[2] For example, Java generics are transformed during compilation in order to preserve backward compatibility.

```
3      ClassReader cr = new ClassReader(fis);
4      ClassNode cn = new ClassNode();
5      cr.accept(cn,
6       ClassReader.SKIP_DEBUG|ClassReader.SKIP_CODE);
7      String className = cn.name; // java/lang/Object
8      System.out.println(className);
9      List<?> methods = cn.methods;
10     List<?> fields = cn.fields;
11     for(Object o:fields){
12       FieldNode node =(FieldNode)o;
13       String nname = node.name;
14       Type type =
15        org.objectweb.asm.Type.getType(node.desc);
16       int acc = node.access;
17       System.out.println(acc + " " +
18        type.getClassName() + " "+nname+";");
19     }
20     for(Object o:methods){
21       MethodNode node = (MethodNode) o;
22       int acc = node.access;
23       String mname = node.name;
24       Type[] type = org.objectweb.asm.Type.
25        getArgumentTypes(node.desc);
26       String r = null;
27       r = Type.getReturnType(node.desc).
28        getClassName();
29       System.out.print(acc + " " +r + " "+mname+"(");
30       for(int i=0;i<type.length;i++){
31         if(i<type.length-1)
32           System.out.print(type[i]+",");
33         else System.out.print(type[i].getClassName());
34       }
35       System.out.println(")");
36     }
37   }
```

First an `InputStream` is needed, usually this will be a `FileInputStream`. Then a `ClassReader` object and a `ClassNode` object are created. The `ClassReader` is a *Visitor* and the `ClassNode` will contain the data from the analysis.

The method call `cr.accept(arg1, arg2)` starts the analysis and after that call is processed, it is possible to access the data from the analysis, the `arg1` argument is the *Visitor* and the `arg2` argument holds flags which configure ASM (for example, if local variables are not needed, the `SKIP_DEBUG` flag may be used).

The following code is a simple way to access the data from the analysis. In order to access the necessary data, the content of the `ClassNode` fields are read. One minor annoyance is that most values have to be converted, the modifiers are coded by an integer (`access`) and the type names are in the `desc` field. For the type, there are static convenience methods in the `org.objectweb.asm.Type` class. A second annoyance is that the current version of ASM does not use generics, which leads to some unnecessary casts like those above to `FieldNode` or `MethodNode`.

Below are the results of the analysis for the example file shown in listing 6.1 by using the ASM analyzer in Listing 6.3.

Listing 6.4: Output for our simple analyzer

```
1 AnalyzeMe
2 17 java.lang.String CONST;
3 10 int temp;
4 8 void <clinit>()
5 9 void main(java.lang.String[])
6 1 void <init>()
7 1 void foo()
8 1 java.lang.String foo(java.lang.String)
9 1 void foo(int)
```

Each line is started with an integer value representing the access modifiers[3]. If the line represents a field, the access code will be followed by the field's name. If the line represents a method, the access code will be followed by the method's return type, name and argument types. The qualified names of the types can be transformed to the usual format by the `.getClassName()` method.

There are two important method names which act as a keyword, `<init>` is the default method name given to a constructor by ASM. `<clinit>` is the default method name given to static initializers by ASM. In our example, the static was included by the Java compiler in order to initialize the fields.

6.1.3 Recoder

Introduction to Recoder Recoder [rec] is a source code based analyzer. See Section 6.1.1 for an explanation of the difference to bytecode based analyzers.

Tool requirements Recoder needs compilable code and all dependencies of the code must be fulfilled. While it should be possible to partially analyze code where we do not have all dependencies, our results were not satisfying for that. The old version of Recoder featured a hand-written bytecode parser, since version 0.94 this one has been replaced by an ASM driven parser and Recoder needs the ASM API.

Code example

Listing 6.5: A basic analyzer for Recoder

```
1 public static void
2  printMethodSignatures(File f){
3  CrossReferenceServiceConfiguration crsc
4   = new CrossReferenceServiceConfiguration();
5  crsc.getProjectSettings().setProperty(
6   PropertyNames.INPUT_PATH,
7   f.getAbsolutePath());
8
```

[3] you can find the corresponding opcodes in `org.objectweb.asm.Opcodes`

```
 9  crsc.getProjectSettings().
10   ensureSystemClassesAreInPath();
11  crsc.getProjectSettings().
12   ensureExtensionClassesAreInPath();
13  SourceFileRepository sfr =
14   crsc.getSourceFileRepository();
15  List<CompilationUnit> cul = null;
16  try {
17   cul = sfr.getAllCompilationUnitsFromPath();
18  } catch (ParserException e) {
19   e.printStackTrace();
20  }
21
22  crsc.getChangeHistory().updateModel();
23
24  for (CompilationUnit cu : cul) {
25   TreeWalker tw = new TreeWalker(cu);
26   TypeDeclaration td =
27    cu.getPrimaryTypeDeclaration();
28   System.out.println(td.getFullName());
29   List<FieldSpecification> fields =td.getFields();
30   for(FieldSpecification fsp:fields){
31    System.out.println(fsp.getType().getName()+
32     " "+fsp.getName()+";");
33   }
34   List<Method> methods = td.getMethods();
35   methods.addAll(td.getConstructors());
36   for(Method m:methods){
37    String sig = "";
38    if(m.isPublic()) sig+="public";
39    if(m.isPrivate()) sig+="private";
40    if(m.isProtected()) sig+="protected";
41    if(m.isStatic()) sig+=" static";
42    if(m.isStrictFp()) sig+=" strict_fp";
43    if(m.isAbstract()) sig+=" abstract";
44    if(m.isFinal()) sig+=" final";
45    if(m.isNative()) sig+=" natic";
46    if(m.isSynchronized()) sig+=" synchronized";
47    String ret=" ";
48    if(!(m instanceof Constructor))
49     if(m.getReturnType()==null) ret=" void "; else
50     ret+=m.getReturnType().getName()+" ";
51    System.out.print(sig+ret+m.getName()+"(");
52    for(Type type:m.getSignature()){
53     System.out.print(" "+type.getName());
54    }
55   System.out.println(")");
56
57  }
```

```
58 }
```

The first two statements initialize Recoder, the following two statements ensure that the necessary classes are in the classpath (in this example the Java API). The method call `sfr.getAllCompilationUnitsFromPath();` causes Recoder to parse the source files. However, the data is only accessible for the user after the `updateModel()` method call which updates the data model.

In order to read the data, the user can either use a `Visitor` to traverse the AST or read the respective fields. The code for reading the data in this example is rather easy to understand and similar to the code in the ASM example. However, there are two interesting differences to the ASM code:

1. We already have convenience methods for reading the modifiers for the fields and methods.
2. Recoder uses generics.

The code in Listing 6.6 is the output of a run of that program with our example class in Listing 6.1.

Listing 6.6: Output for the basic Recoder analyzer

```
1 AnalyzeMe
2 String CONST;
3 int temp;
4 public static void main( String[])
5 public void foo()
6 public String foo( String)
7 public void foo( int)
8 public AnalyzeMe()
```

The output is as expected. Differences to the ASM output are:

- The modifiers are not encoded, since we did use the available transformation methods.
- Recoder distinguish between constructors and methods on an AST level and uses the classname as the name of the constructor.
- There are no static initializer methods. Recoder can only analyze static intializers that are specified by the programmer.

6.1.4 JDT

Introduction to JDT The JDT or Java Developement Tools[4] are an API present in the Eclipse system [ecl]. It is used for code analysis and transformation by the Eclipse system.

Like Recoder, it is a source code based analyzer (see Section 6.1.1).

A significant advantage of the JDT is its name resolution. The JDT is capable to provide the user with a full name and (static) type resolution for all variables in the source code.

[4] See `http://www.eclipse.org/jdt/`

Tool requirements In order to be able to use all features of the JDT, the source code must be able to compile and all of its dependencies are needed. However, the JDT is pretty error prone, it is able to analyze incorrect Java files up to a certain degree, which makes the JDT especially useful for code development assistance.

The JDT is designed to be used as an Eclipse plugin. It is possible to use the JDT API outside of Eclipse. However, it uses many other Eclipse libraries and it is a cumbersome process to provide the JDT parser with all necessary files and libraries. If the JDT API is used within the Eclipse system, it is easier to find the dependencies since JDT can access the dependencies configured in the project settings.

Code example The following code example is similar to those described in the ASM and Recoder section. However, it is not implemented as a standalone analyzer but as a minimal Eclipse plugin.

The following code shows a simple JDT analyzer.

Listing 6.7: A basic analyzer using the JDT

```
1  public static void printMethodSignatures
2    (ICompilationUnit icu){
3   ASTParser parser = ASTParser.newParser(AST.JLS3);
4   parser.setKind(ASTParser.K_COMPILATION_UNIT);
5   parser.setSource(icu);
6   parser.setResolveBindings(true);
7   CompilationUnit cu = (CompilationUnit)
8    parser.createAST(null);
9   ASTVisitor av = new ASTVisitor(){
10   public void printModifier(int modifier){
11    if(Modifier.isPublic(modifier))
12     System.out.print("public ");
13    if(Modifier.isPrivate(modifier))
14     System.out.print("private ");
15    if(Modifier.isProtected(modifier))
16     System.out.print("protected ");
17    if(Modifier.isFinal(modifier))
18     System.out.print("final ");
19    if(Modifier.isAbstract(modifier))
20     System.out.print("abstract ");
21    if(Modifier.isStatic(modifier))
22     System.out.print("static ");
23    if(Modifier.isSynchronized(modifier))
24     System.out.print("synchronized ");
25    if(Modifier.isVolatile(modifier))
26     System.out.print("volatile ");
27    if(Modifier.isStrictfp(modifier))
28     System.out.print("strict_fp ");
29    if(Modifier.isNative(modifier))
30     System.out.print("native ");
31    if(Modifier.isTransient(modifier))
32     System.out.print("transient ");
33   }
```

```
34
35     @Override
36     public boolean visit(TypeDeclaration node) {
37       System.out.println(node.getName());
38       return super.visit(node);
39     }
40
41     @Override
42     public boolean visit(FieldDeclaration node) {
43       printModifier(node.getModifiers());
44       int i=0;
45       System.out.print(node.getType().toString()+" ");
46       for(Object o:node.fragments()){
47         VariableDeclarationFragment vdf =
48           (VariableDeclarationFragment)o;
49         System.out.print(vdf.getName());
50         i++;
51         if(i<node.fragments().size())
52           System.out.print(", ");
53       }
54       System.out.println(";");
55       return super.visit(node);
56     }
57
58     @Override
59     public boolean visit(MethodDeclaration node) {
60       printModifier(node.getModifiers());
61       Type t = node.getReturnType2();
62       if(t!=null)System.out.print(t.toString()+" ");
63       System.out.print(node.getName()+"(");
64       int i=0;
65       for (Object o:node.parameters()){
66         SingleVariableDeclaration svd =
67           (SingleVariableDeclaration)o;
68         System.out.print(svd.getType().toString());
69         i++;
70         if(i<node.parameters().size())
71           System.out.print(", ");
72       }
73       System.out.println(")");
74       return super.visit(node);
75   }};
76   cu.accept(av);
77 }
```

First, the parser is configured by setting the Java Language Specification that should be used and the source which should be parsed. In this case the source code is given as an `ICompilationUnit` which represents a source code file in an Eclipse project, that makes it possible for Eclipse to access its dependencies. In addition to setting the source, the kind of the source (in our case `ASTParser.K_COMPILATION_UNIT` must

be specified. The line `parser.setResolveBindings(true);` is optional, if it is set, the parsing step will need more memory and takes longer. The advantage is that the static type information for the variables are analyzed.

The anonymous class `ASTVisitor av = new ASTVisitor()...` is the *Visitor* which is used for fetching the data from the analysis. Contrary to ASM and Recoder, there are no fields that can be accessed directly for the information.

The first method in the *Visitor* formats and prints the modifier of an `ASTNode`. The modifiers are stored encoded as an integer and the `Modifier` class has static methods for decoding them. The next three methods will be called if a `TypeDeclaration`, a `FieldDeclaration` or a `MethodDeclaration` is visited. An overview over all JDT ASTNode is given in the Appendix (see section 6.2).

The code should be easy to understand, but there is some inconsistency within the JDT API. For example, the return type of a `MethodDeclaration` is accessible by a getter while the method parameters are stored in a public field of the `MethodDeclaration` and there is no getter for them.

The following code is the output of a run of that program with our example class in Listing 6.1.

Listing 6.8: Output for the basic JDT analyzer

```
1 AnalyzeMe
2 public final String CONST;
3 private static int temp;
4 public static void main(String[])
5 public AnalyzeMe()
6 public void foo()
7 public String foo(String)
8 public void foo(int)
```

The output is similar to the Recoder output. The output would differ if we had a nested class. The fields of Recoder (or ASM) only store data belonging to the current class, while the example JDT analyzer above visits all `MethodDeclarations` and `FieldDeclarations` in a class. Normally, it would be necessary to overwrite the visit method for nested classes so that the *Visitor* stops there.

6.1.5 Other tools

This section introduced the the three tools which we primarily used for code analysis (and in the case of JDT for API migration).

Another interesting tool is JTransformer [5] which uses the JDT API for analysis and code transformation and allows the user to use Prolog queries to gather data.

Another possible approach is to use the Java compiler (which es used for our first analyzer). There is a tool called APT (APT - Annotation Processing Tool [APT]) which allows the user to access the annotations of a program during compilation time, it is also possible (but not clearly documented) to access the AST of the java compiler, which allows the user to analyze the currently compiled program. We do not recommend this, since we encountered some strange problems using that, but it was mostly sufficient for the API 1.0 analysis project (see Section 1.2).

[5] See the related works section in Section 1.2 for more information.

6.2 JDT AST Nodes

Figure 6.2 shows all ASTNodes of the current JDT API.

6.3 Complete DTD

Listing 6.9: Complete DTD of the variable analyzer

```
1  <!ELEMENT RUN (TASK)+>
2   <!ELEMENT TASK (SOURCES,ANALYZE, OUTPUT)>
3    <!ATTLIST TASK mode (list|project|subset) #REQUIRED>
4  <!ELEMENT SOURCES(DIRECTORY)>
5   <!ATTLIST SOURCES additional_configuration CDATA #IMPLIED>
6   <!ELEMENT DIRECTORY>
7    <!ATTLIST DIRECTORY location CDATA #REQUIRED>
8    <!ATTLIST DIRECTORY subdirectories  (true|false) 'false'>
9  <!ELEMENT ANALYZE(CONFIG)?>
10  <!ATTLIST ANALYZE mode
11    (whitebox|blackbox|specialized) #REQUIRED>
12  <!ATTLIST ANALYZE analyze_method_calls (true|false) false>
13  <!ELEMENT CONFIG>
14   <!ATTLIST CONFIG
15     classes (public|private|protected|default) "public"
16     methods (public|private|protected|default|false) "public"
17     fields (public|private|protected|default|false) "public"
18     local_fields (true|false) "false"
19   >
20 <!ELEMENT OUTPUT(CLAUSES)>
21  <!ELEMENT CLAUSES>
22  <!ATTLIST CLAUSES directory CDATA #REQUIRED>
```

- AnnotationTypeDeclaration
- AnnotationTypeMemberDeclaration
- AnonymousClassDeclaration
- ArrayAccess
- ArrayCreation
- ArrayInitializer
- ArrayType
- AssertStatement
- Assignment
- Block
- BlockComment
- BooleanLiteral
- BreakStatement
- CastExpression
- CatchClause
- CharacterLiteral
- ClassInstanceCreation
- CompilationUnit
- ConditionalExpression
- ConstructorInvocation
- ContinueStatement
- DoStatement
- EmptyStatement
- EnhancedForStatement
- EnumConstantDeclaration
- EnumDeclaration
- ExpressionStatement
- FieldAccess
- FieldDeclaration
- ForStatement
- IfStatement
- ImportDeclaration
- InfixExpression
- InstanceofExpression
- Initializer
- Javadoc
- LabeledStatement
- LineComment
- MarkerAnnotation
- MemberRef
- MemberValuePair
- MethodRef
- MethodRefParameter
- MethodDeclaration
- MethodInvocation
- Modifier
- NormalAnnotation
- NullLiteral
- NumberLiteral
- PackageDeclaration
- ParameterizedType
- ParenthesizedExpression
- PostfixExpression
- PrefixExpression
- PrimitiveType
- QualifiedName
- QualifiedType
- ReturnStatement
- SimpleName
- SimpleType
- SingleMemberAnnotation
- SingleVariableDeclaration
- StringLiteral
- SuperConstructorInvocation
- SuperFieldAccess
- SuperMethodInvocation
- SwitchCase
- SwitchStatement
- SynchronizedStatement
- TagElement
- TextElement
- ThisExpression
- ThrowStatement
- TryStatement
- TypeDeclaration
- TypeDeclarationStatement
- TypeLiteral
- TypeParameter
- VariableDeclarationExpression
- VariableDeclarationStatement
- VariableDeclarationFragment
- WhileStatement
- WildcardTypetrue

Figure 6.1: All JDT AST nodes

6.4 Example grammar for the *Simple Mapping Language*

⟨MAPPING_FILE⟩→⟨JAVA_CLASS⟩|⟨MAPPING⟩
⟨MAPPING⟩→⟨HEADER⟩ '{' ⟨BODY⟩ '}'
⟨HEADER⟩→[⟨PACKAGE⟩] [⟨IMPORT⟩] ⟨UTILCLASS⟩
⟨MAPSTART⟩
⟨PACKAGE⟩→'package' ⟨JAVA_QUALIFIEDNAME⟩ ';'
⟨JAVA_IMPORT⟩→⟨JAVA_QUALIFIEDNAME⟩ ['.*'] ';'
⟨IMPORT⟩→ 'import' ⟨JAVA_IMPORT⟩ [⟨IMPORT⟩]
⟨UTILCLASS⟩→'util' '=' ('null' | ⟨JAVA_QUALIFIEDNAME⟩)
⟨MAPSTART⟩→ 'map' ⟨JAVA_NAME⟩ 'to'
⟨JAVA_QUALIFIEDNAME⟩ '{'⟨MAPBODY⟩'}'
⟨MAPBODY⟩→(⟨FIELD⟩|⟨METHOD⟩|⟨INNER_CLASS⟩|
⟨INNER_MAP⟩)[⟨MAPBODY⟩]
⟨FIELD⟩→⟨FIELD_DECLARATION⟩ ['='
(⟨DEFAULT_VALUE⟩|⟨UTIL_METHOD⟩)] ';'
⟨METHOD⟩→'map' ⟨JAVA_QUALIFIEDNAME⟩ '('
⟨PARAMETER_DECLARATION⟩')' 'to'
(⟨CONSTRUCTOR⟩|⟨METHOD_CALL⟩|⟨UTIL_METHOD⟩) ';'
⟨CONSTRUCTOR⟩→ 'new' ⟨JAVA_QUALIFIEDNAME⟩'
('⟨PARAMETER⟩')'
⟨UTIL_METHOD⟩→'util.' ⟨JAVA_NAME⟩'('⟨PARAMTER⟩')'

Figure 6.2: The example grammar for the Simple Mapping Language pattern

6.5 Mapping complexity

In the easiest case it is possible to map directly between a single method of the source API and a single method of the target API. In the worst case a new method must be written. Here is a classification of the complexity of a mapping, which was used during the developement of our tools and mentioned in the *Mapping Language Extension* pattern. [6]

Level 1 If we have a level 1 relation we can delegate from a method `m1` to a method `m2` directly, by only changing the name of the method and rearranging the arguments.

Level 2 In addition to the possible steps for a level 1 relation the arguments may be slightly modified. It is allowed to use constants, casts and to use less or more arguments. Simple arithmetic operations (adding or subtracting constants) are also allowed.

Level 3 Everything is allowed that is possible in the target language.

[6] These classifications are loosely based on the levels of adaption specified in [BCLS09], but take into account implementation details from a Java perspective

Bibliography

[AMUFVI08] ALMEIDA-MARTÍNEZ, Francisco J. ; URQUIZA-FUENTES, Jaime ; VELÁZQUEZ-ITURBIDE, J. Á: VAST: visualization of abstract syntax trees within language processors courses. In: *Proceedings of the 4th ACM symposium on Software visualization*. New York, NY, USA : ACM, 2008 (SoftVis '08). – ISBN 978–1–60558–112–5, 209–210

[APT] *APT - Annotation Processing Tool*. http://download.oracle.com/javase/1,5.0/docs/guide/apt/GettingStarted.html

[BBS05] BLEWITT, Alex ; BUNDY, Alan ; STARK, Ian: Automatic verification of design patterns in Java. In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA : ACM, 2005 (ASE '05). – ISBN 1–58113–993–4, 224–232

[BCLS09] BARTOLOMEI, Thiago T. ; CZARNECKI, Krzysztof ; LÄMMEL, Ralf ; STORM, Tijs van d.: Study of an API Migration for Two XML APIs. In: *SLE*, 2009, S. 42–61

[BDF+11] BARTOLOMEI, Thiago T. ; DERAKHSHANMANESH, Mahdi ; FUHR, Andreas ; KOCH, Peter ; KONRATH, Mathias ; LÄMMEL, Ralf ; WINNEBECK, Heiko: Combining multiple dimensions of knowledge in API migration. In: *First International Workshop on Model-Driven Software Migration (MDSM 2011)*, CEUR, 2011. – 4 pages. To appear.

[BR06] BRINGERT, Björn ; RANTA, Aarne: A pattern for almost compositional functions. In: *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*. New York, NY, USA : ACM, 2006 (ICFP '06). – ISBN 1–59593–309–3, 216–226

[BTF05] BALABAN, Ittai ; TIP, Frank ; FUHRER, Robert: Refactoring support for class library migration. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM, 2005 (OOPSLA '05). – ISBN 1–59593–031–0, 265–279

[CGM06] COHEN, Tal ; GIL, Joseph (. ; MAMAN, Itay: JTL: the Java tools language. In: *SIGPLAN Not.* 41 (2006), October, 89–108. http://dx.doi.org/http://doi.acm.org/10.1145/1167515.1167481. – DOI http://doi.acm.org/10.1145/1167515.1167481. – ISSN 0362–1340

[DJ06] DIG, Danny ; JOHNSON, Ralph: Automated upgrading of component-based applications. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06). – ISBN 1–59593–491–X, 675–676

[DZ07] DONG, Jing ; ZHAO, Yajing: Experiments on Design Pattern Discovery. In: *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007 (PROMISE '07). – ISBN 0–7695–2954–2, 12–

[ecl] *The Eclipse project*. `http://www.eclipse.org/`

[EH07] EKMAN, Torbjörn ; HEDIN, Görel: The JastAdd extensible Java compiler. In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. New York, NY, USA : ACM, 2007 (OOPSLA '07). – ISBN 978–1–59593–865–7, 884–885

[Eri] ERIC BRUNETON AND REMI FORAX AND EUGENE KULESHOV AND ANDREI LOSKUTOV: *ASM Framework Homepage*. `http://asm.ow2.org/`

[Fre06] FREESE, Tammo: Refactoring-aware version control. In: *Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006 (ICSE '06). – ISBN 1–59593–375–1, 953–956

[FY10] FERNANDEZ, Eduardo B. ; YUAN, Xiaohong: An analysis pattern for invoice processing. In: *Proceedings of the 16th Conference on Pattern Languages of Programs*. New York, NY, USA : ACM, 2010 (PLoP '09). – ISBN 978–1–60558–873–5, 10:1–10:10

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E. ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA : Addison-Wesley, 1995. – ISBN 978–0–201–63361–0

[GJS05] GOSLING, James ; JOY, Bill ; STEELE, Guy L.: *The Java Language Specification - Third Edition*. Addison-Wesley Longman, 2005

[GM05] GIL, Joseph (. ; MAMAN, Itay: Micro patterns in Java code. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM, 2005 (OOPSLA '05). – ISBN 1–59593–031–0, 97–116

[GSF10] GUERRA, Eduardo M. ; SOUZA, Jerffeson T. ; FERNANDES, Clovis T.: A pattern language for metadata-based frameworks. In: *Proceedings of the 16th Conference on Pattern Languages of Programs*. New York, NY, USA : ACM, 2010 (PLoP '09). – ISBN 978–1–60558–873–5, 3:1–3:29

[HD05] HENKEL, Johannes ; DIWAN, Amer: CatchUp!: capturing and replaying refactorings to support API evolution. In: *Proceedings of the 27th international conference on Software engineering*. New York, NY, USA : ACM, 2005 (ICSE '05). – ISBN 1–58113–963–2, 274–283

[Iac11] IACOB, Claudia: A design pattern mining method for interaction design. In: *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. New York, NY, USA : ACM, 2011 (EICS '11). – ISBN 978–1–4503–0670–6, 217–222

[JC10] JENSEN, Adam C. ; CHENG, Betty H.: On the use of genetic programming for automated refactoring and the introduction of design patterns. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. New York, NY, USA : ACM, 2010 (GECCO '10). – ISBN 978–1–4503–0072–8, 1341–1348

[JDT] *Java Develeopement Tools*. `http://www.eclipse.org/jdt/`

[JDV03] JANZEN, Doug ; DE VOLDER, Kris: Navigating and querying code without getting lost. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2003 (AOSD '03). – ISBN 1–58113–660–9, 178–187

[JTr] *The JTransformer project*. `https://sewiki.iai.uni-bonn.de/research/jtransformer/start`

[KHR07] KNIESEL, Günter ; HANNEMANN, Jan ; RHO, Tobias: A comparison of logic-based infrastructures for concern detection and extraction. In: *Proceedings of the 3rd workshop on Linking aspect technology and evolution*. New York, NY, USA : ACM, 2007 (LATE '07). – ISBN 978–1–59593–655–4

[KP88] KRASNER, Glenn E. ; POPE, Stephen T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. In: *J. Object Oriented Program.* 1 (1988), August, 26–49. `http://portal.acm.org/citation.cfm?id=50757.50759`. – ISSN 0896–8438

[KR09] KAWRYKOW, David ; ROBILLARD, Martin P.: Improving API Usage through Automatic Detection of Redundant Code. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2009 (ASE '09). – ISBN 978–0–7695–3891–4, 111–122

[LPS11] LÄMMEL, Ralf ; PEK, Ekaterina ; STAREK, Jürgen: Large-scale, AST-based API-usage analysis of open-source Java projects. In: *SAC'11 - ACM 2011 SYMPOSIUM ON APPLIED COMPUTING, Technical Track on "Programming Languages"*, 2011

[NFH05] NEAMTIU, Iulian ; FOSTER, Jeffrey S. ; HICKS, Michael: Understanding source code evolution using abstract syntax tree matching. In: *Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA : ACM, 2005 (MSR '05). – ISBN 1–59593–123–6, 1–5

[Noc03] NOCK, Clifton: *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Prentice Hall Professional Technical Reference, 2003. – ISBN 0131401572

[rec] *ReCoder*. http://sourceforge.net/apps/mediawiki/recoder/index.php?title=Main_Page

[SH11] SÖDERBERG, Emma ; HEDIN, Görel: Building semantic editors using JastAdd: tool demonstration. In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. New York, NY, USA : ACM, 2011 (LDTA '11). – ISBN 978–1–4503–0665–2, 11:1–11:6

[SRG08] SAVGA, Ilie ; RUDOLF, Michael ; GOETZ, Sebastian: ComeBack!: a refactoring-based tool for binary-compatible framework upgrade. In: *Companion of the 30th international conference on Software engineering*. New York, NY, USA : ACM, 2008 (ICSE Companion '08). – ISBN 978–1–60558–079–1, 941–942

[TFK+11] TIP, Frank ; FUHRER, Robert M. ; KIEŻUN, Adam ; ERNST, Michael D. ; BALABAN, Ittai ; SUTTER, Bjorn D.: Refactoring using type constraints. In: *ACM Trans. Program. Lang. Syst.* 33 (2011), May, 9:1–9:47. http://dx.doi.org/http://doi.acm.org/10.1145/1961204.1961205. – DOI http://doi.acm.org/10.1145/1961204.1961205. – ISSN 0164–0925

[Wel06] WELICKI, Leon: The configuration data caching pattern. In: *Proceedings of the 2006 conference on Pattern languages of programs*. New York, NY, USA : ACM, 2006 (PLoP '06). – ISBN 978–1–60558–372–3, 28:1–28:6

[WM09] WINTERS, Niall ; MOR, Yishay: Dealing with abstraction: Case study generalisation as a method for eliciting design patterns. In: *Comput. Hum. Behav.* 25 (2009), September, 1079–1088. http://dx.doi.org/10.1016/j.chb.2009.01.007. – DOI 10.1016/j.chb.2009.01.007. – ISSN 0747–5632

[WYWB08] WELICKI, León ; YODER, Joseph W. ; WIRFS-BROCK, Rebecca: The dynamic factory pattern. In: *Proceedings of the 15th Conference on Pattern Languages of Programs*. New York, NY, USA : ACM, 2008 (PLoP '08). – ISBN 978–1–60558–151–4, 9:1–9:7

[WYWB10] WELICKI, León ; YODER, Joseph W. ; WIRFS-BROCK, Rebecca: Adaptive object-model builder. In: *Proceedings of the 16th Conference on Pattern Languages of Programs*. New York, NY, USA : ACM, 2010 (PLoP '09). – ISBN 978–1–60558–873–5, 4:1–4:8

[ZTX+10] ZHONG, Hao ; THUMMALAPENTA, Suresh ; XIE, Tao ; ZHANG, Lu ; WANG, Qing: Mining API mapping for language migration. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA : ACM, 2010 (ICSE '10). – ISBN 978–1–60558–719–6, 195–204