Testing Source Code with the Logic Programming Language Prolog

Master Thesis

Thomas Handwerker

Supervisors: Prof. Dr. Dietmar Seipel
M.Sc. Falco Nogatz

Chair: Chair for Computer Science I
(Algorithms, Complexity and Knowledge-Based Systems)

Institute: Institute of Computer Science

Submission Date: August 1, 2016

Abstract

Static code analysis allows the calculation of different software metrics of a provided code base. Instead of calculating a metric we dedicate to the examination of design patterns and guidelines in object–oriented implementations. Within the current thesis the testability of source code and what factors influence it in a negative aspect shall be examined.

We want to establish in this approach a complete toolchain that realises static code analysis based on the logical programming language Prolog. Defining analysis tasks are defined within paraphrasing conditional sentences which describe and specify the coding characteristics to examine. A graphical user interface is responsible for the visualisation of the analysis results.

The toolchain provides developers with a flexible analysis tool. Because of its rule–based analysis tasks the examination is not only restricted to testability. Due to the modular architecture of the tool its usage shall not be limited to Java and the related Eclipse IDE. In conclusion, a complete toolchain is realised including an own language to paraphrase analysis rules, the core analysis module, and an integration to a graphical user interface to visualise the results.

Zusammenfassung

Mit der Unterstützung von statischer Code Analyse lassen sich verschiedenste Metriken zu dem Quelltext einer Softwarekomponente berechnen. Im Gegensatz zur Berechnung von Metriken wollen wir uns bei diesem Ansatz vielmehr dem Erkennen von Mustern und Design–Richtlinien in Quelltexten widmen. Der Fokus hierbei liegt auf der Untersuchung der Testbarkeit von Quellcode und welche Faktoren diese beeinflussen.

Im Rahmen dieser Arbeit ist eine komplette Werkzeugpalette zur statischen Code Analyse umzusetzen. Die Implementierung der Komponenten soll hierbei in der logischen Programmiersprache Prolog realisiert werden. Die Definition von Analyse–Tasks sind hierbei als Bedingungssätze zu formulieren und spezifieren die relevanten Design–Pattern. Eine grafische Oberfläche dient zur Visualisierung von den Ergebnissen der Analyse.

Die Werkzeugpalette gibt Entwicklern ein flexibles Analyse–Tool an die Hand, welches aufgrund der regelbasierten Analyse nicht nur auf die Thematik “Testbarkeit” beschränkt ist. Durch die Wahl der modularen Architektur sollen die Einsatzmöglichkeiten von YaCI zukünftig nicht nur auf Java in Verbindung mit der Eclipse IDE eingeschränkt sein. Schlussendlich ist ein einsatzbereites Analysewerkzeug entstanden, dass eine eigenen Sprache für Analyse–Regeln, das Modul zur Code–Analyse, sowie die Integration einer grafischen Oberfläche zur Anzeige der Ergebnisse umfasst.

Contents

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
BDD	Behaviour Driven Development
CI	Continuous Integration
DCG	Definite Clause Grammar
DSL	Domain–specific Language
GUI	Graphical User Interface
IDE	Integrated Development Environment
OOP	Object–oriented Programming
PDT	Prolog Development Tools
TDD	Test Driven Development
UI	User Interface
YaCI	Yet another Code Inspector

List of Figures

List of Tables

Contents of Listings

1. Introduction

Maintenance and analysing legacy code will be the focus of the current work. We want to establish a toolchain which give developers the availability to analyse existing software components and the related source code on the basis of a set of rules. Defining rules shall be as simple as possible in order to give every developer the possibility to phrase own rules. The key area of the analyses in this thesis is to examine and assess the testability of the object–oriented programming language Java. The basic collection of testability rules is extracted from various guidelines and design pattern described in literature. Furthermore, the specification of rules shall be flexible to support developers on writing their own analysis rules and extend the process on analysing the source code. The outcome of intended analysis tool is very important for the developer to improve the legacy code and observe rule violations from the analysis process. For this purpose the integration in a graphical user interface of a Integrated Developement Environment is planned. Because of the fact that a flexible rule language shall be specified it can be possible to write analysis rules which are in another context than testability. In the following sections we want to have a closer look on to the motivation of this topic, the research field shall be defined in a general way, and to search for related work on analysing software components which are written in an object–oriented manner. Finally, the objectives are set for the current thesis we are planned to met. In the last section a short roadmap gives an overview to the procedure while implementing the *Yet another Code Inspector*.

1.1. Motivation

In the past years, software developer didn't realize that writing tests and verify correct functionality of their own implemented source code would be part of their responsibility. Powerful Integrated Development Environments (in short: IDEs) like Eclipse or NetBeans for Java developers have been unknown and not available at this time. Debugging the implemented code consists of simple adding output to command line interface or log files hoping to find helpful information [Osh09, Mar08]. At this time the responsibility of testing was assigned to a particular test team and developers only validate correct functionality while running the code next to their

development process. Lasse Koskela describes this mindset in [Kos13] as following: *[. . .] „testing“ for most programmers meant [. . .] the stuff that somebody does after I'm done coding, or the way you run and poke your code before you say you're done coding.*

But this point of view changed dramatically over the last years. On the topic *Unit Tests* Robert C. Martin look back at the time when developers did not realise how important a well established test suite is helpful to develop good software. He remembers when he wrote a small test for a C++ timer method: *"[. . .] Once I saw it work and demonstrated it to my colleagues, I threw the test code away."* [Mar08]. Nowadays, agile software development is the preferable set of principles in modern software engineering [WMV03]. Under the collective term *Extreme Programming* (also named XP) there are different techniques established, like Test or Behaviour Driven Development. These principles are already encourage developers to write testable code because especially in TDD the unit test have to be implemented before writing the first line of associated code in production.

Those modern approaches like TDD and BDD forces today's developers to write associated test code to improve the quality of the planned software unit. Indeed the reality often shows the contrary. Dirk W. Hoffman check the evolution of software quality in [Hof08] over the past years. As the main problem he list the increasing complexity of today's software projects and their long-term maintainability that often results in project schedules wich are not deliverable to the pitched time. The complexity of the project in later stage of development is an important negative influence on the ease of writing unit tests. Another point is the maintenance of legacy code from existing projects over the last few years. The threshold to change code or implement new features in untested code is much higher then with tested code. With a good test coverage, the evaluation of correct code functionality is easy to check. After changes in code base running these tests give direct outcome, if everything is all right and works smoothly.

The motivation of this thesis is to build up a small toolchain to analyse existing source code in fact of its testability. In other words: we want to check a given code base against predefined and entrenched design patterns which will help to write a code that is testable with less effort. Later on, the term "testability" is discussed in detail and what kind of good and bad patterns exists onto this topic. At the end, the implemented tool should find hard to test placements in code automatically to advise developer refactor the highlighted source section in order to improve testability.

1.2. Research Field

The main topic of this work is to run a static code analysis on an existing code base. First of all, the focus is to make a point about the testability in order to write unit tests with less effort as possible. Based on the analysis results it would be helpful to evaluate a simple metric, which describe the level of reached testability.

For this purpose there are different steps necessary to fulfil the goal. Firstly, one component is a language specification that gives software engineering teams the possibility to phrase own rules which describe testability patterns introduced by unit testing literature research or maybe on top of own experience. The rules are formulated in natural language and not only readable by developers themselves. Another important point is the variety. Teams should be able to describe their own testability smells in form of *"when* $\langle Premise \rangle$ *then* $\langle Conclusion \rangle$*."* rules. In general, static code analysis by our tool will be performed on the abstract syntax tree which is created from given code base.

At least some kind of small output to the command line or the integration in a existing graphical user interface like the Eclipse IDE for instance. Another feasible way to give feedback to the user is to calculate the average rule violation and check the score against predefined threshold value. This approach is useful for the integration of the analysis tool into established continuous life cycle in projects, for instance.

1.3. Related Work

The main focus of this thesis is the analyse of the testability of source code, especially to identify indicators or impacts which influence this in a negative way. Different research publications are already available, that address on that topic. Robert V. Binder talks in [Bin94] about different major factors that describe the testability in the development process. The Googler Miško Hevery published in his blog a collection of testability flaws [Hev08], that he experienced while sensitise other developers at Google in fact of write code that is easier to test. Later on, in chapter *Testability* the focus is on this part of related work and point out different approaches in research.

We decide to realise our approach using the logical programming language *Prolog*. Meanwhile, the implementation *SWI–Prolog* of the ISO–Prolog standard [SS06] reach a respectable acceptance in the Prolog community. There are also other implementations available like *Jekejeke Prolog*[1] for JVM and Android purpose or *SICStus*

[1] `http://www.jekejeke.ch`, accessed on 28th June 2016

Prolog[2] in commercial environment. Various benefits have helped establishing this programming language in a broader community. SWI-Prolog is not longer only a possible approach for usage in different research fields at university. Over the years the development and usage of SWI-Prolog reached a noticeable capacity and its further development is still in progress [Wie16]. In order to support rapid and incremental development, the system must be able to load and run large projects as fast as possible. Synchronisation between changes in source code and the state of running program is another key fact provided by SWI-Prolog. The modular system design is also a huge advantage. Different packages are available to adapt additional components and functionality in one's own Prolog code. Well-known libraries are for example the HTTP library to support access to HTTP servers and provide HTTP server capabilities from SWI-Prolog. There is also a unit testing framework in form of a package available which provide developers to implement simple test suites in Prolog. Jan Wielemaker describes the main features and core concepts of this logical programming language more detailed in [WSTL12].

Roger F. Crew demonstrates in [Cre97] that examining an abstract syntax tree of source code will probably work by using the logical programming language Prolog. His approach introduces the specification of a domain-specific language called ASTLOG to analyse the generated syntax tree and find syntactic artifacts in C++ software components. Another related work on analysing Java code to detect specified design patterns is written by Dirk Heuzeroth and Stefan Mandel in [HML03]. They introduce a high-level language SanD[3] that allows describing design patterns in a more abstract and intuitive way. The pattern detection framework of their approach implements the analysis workflow and finds classified pattern instances evaluated by scanning the abstract syntax tree which is represented as Prolog facts.

Parsing sentences spelled in almost natural language will be a core component of our code analysing tool. The predefined testability and self phrased custom-rules must be parsed into an internal representation for further purpose. To allow developers writing their own rules there have to be a specification who restricts the correct wording of a valid rule sentence in form of a sequentially list. Specification of this type can be realised by the definition of a grammar. William F. Clocksin and Christopher S. Mellish introduced in [CM84] how to solve the parsing problem using SWI-Prolog. A well-known approach to parse sentences is the specification of a Definite Clause Grammar (in short DCG) and is also examined by Alan Bundy and Lincoln Wallen in [BW84]. Another approach to analyse sentences phrased in natural language is described in [PW80] by Fernando Pereira and David Warren. The

[2] `https://www.sics.se/projects/sicstus-prolog-leading-prolog-technology`, accessed on 28th June 2016

[3] Static and Dynamic Specification Language

official documentation[4] to DCGs in SWI-Prolog provides more details and implementation guidelines. In [WH13] Wielemaker and Hendricks describe more detailed what challenges Definite Clause Grammar brings with it and how *quasi quotations*[5] help to solve these problems in SWI-Prolog.

1.4. Implementation Goals

The final version of the tool gives software developers the chance to analyse quickly their source code against self-phrased conditional sentences. First of all the analysis process focuses on the topic Testability. Therefore, a set of rules is predefined that describes common pattern how to write code that is good and easy as possible to test by using unit tests.

The best practices on writing testable code are obtained from different guidelines and books oriented on this topic. For example *Effective Unit Testing* and *Clean Code*. Our tool Yet another Code Inspector (YaCI) will parse the rules into an internal representation and run the analysis process while examining the abstract syntax tree of a given Java project. At least it collects all matches generated from AST analysis and append these to the user interface, based on the JTransformer plug–in for Eclipse. The developers now are able to observe the results and can jump directly into the specified location in source code.

Evaluating our analysis tool against a real world project is essential to get in touch, if YaCI will work as expected. In cooperation with the MULTA MEDIO Informationssysteme AG we have the opportunity to run testability analysis against a productive software, written in Java, with a huge code base. The University of Würzburg provides us with some example projects we can explore, too. These projects have been developed by students during their practical bachelor course and contain simple algorithm implementations in Java for instance. This is a big advantage to improve the created analysis tool to guarantee correct functionality and results received from AST analysis. For the evaluation of YaCI in Chapter 7 tool is running against the open source Java project *Joda-Time*[6]. This utility library provide a quality replacement for default date and time classes available in Java. Thereby, it is a reliable, public accessible, and repeatable test case that could be comprehend from everyone.

[4] `http://www.swi-prolog.org/pldoc/man?section=DCG`, accessed on 28th June 2016

[5] Using quasi quotations in SWI–Prolog allows to embed long strings which contain external language (e.g. XML, JSON) in Prolog. It also provides an alternative representation of long stirngs and atoms in Prolog.

[6] `http://www.joda.org/joda-time`, accessed on 7th July 2016

Finally, the rules shall be customisable to control the result action when violation is matched in the AST. In graphical user interface different reporting levels maybe useful (like info, warning or error). Reporting such severity is meaningful in an Integrated Development Environment, but for instance in a continuous integration lifecycle this kind of result visualisation is not very helpful. Calculating a average scoring value from failed rules while examination and compare against a defined threshold shall be a more useful scenario for CI. Another simple use case on defined rules is to match specific design patterns in code like composition or singleton pattern.

1.5. Roadmap

This thesis is divided into several parts and will show the way of proceeding. In the first part it will be clarified what the term Testability means, what best practices and programming guidelines exists and how to measure the Testability of source code. Chapter 3 focuses on JTransformer, a powerful Eclipse plug–in to analyse and transform Java code using SWI Prolog. That part will describe in detail what functionality the tool provides and how to integrate it into the current research topic.

According to the definition of Testability and the introduction to JTransformer the description of the *YaCI Rule Language* is part of Chapter 4. Phrasing rules will give developers the freedom to define own constructs of analysis patterns. The core part of this research is the implementation of an analyse tool. It will use the rule sentences in order to examine the provided source code according to their description of the pattern characteristics. Instead of the need to implement analysis tasks in a higher level programming language (e.g. Java or Prolog) this approach wants to add an additional abstraction layer specified by analysis rule sentences. The technical details and implementation of YaCI is discussed in detail later on in Chapter 6. It reconciles the different aspects of the toolchain, like parsing testability rules, analyses the abstract syntax tree, and pre-process possible matches to visualise results for developers in the JTransformer UI.

The the evaluation shows that the developed YaCI tool works as expected and provides the correct functionality to end users. An initial set of testability rules in combination with the available example projects show some results, matched by our analysis tool. This will be part of Chapter 7. Finally we will discuss the chosen approach and will give a small summary of the work done. In a short outlook we want to show possible future works and the ability to extend the YaCI toolchain.

2. Testability

This section describes the mean of testability in context of object–oriented programming languages in detail and gives readers a definition of it. Another point is, to examine good respectively bad pattern that exists to improve testability. With these patterns the developers have useful guidelines on how to implement new features and structure code in order to make it easy to test. Afterwards it is important to know how the testability of a source code can be measured. Is there already a established metric available which allows to determines, if the current code base is testable without much effort? Otherwise, what can be a useful metric?

2.1. Testability in General

Exploring testability of software is not an entirely new approach in the domain of code analysis. In the literature are different definitions what does "easy to test" in detail means. In the following, a better understanding shall be provided by the introduction of the important definitions on what does testability mean and how it is defined.

A very common way to define testability is the ease of performing test cases. The IEEE Standard Glossary of Software Engineering Terminology definition reads as follows: "(1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met." [IEE90]. The definition in the ISO Standard is more general. They define testability as *"the capability of the software product to enable modified software to be validated"* [ISO01].

Bruntink and van Deursen [BVD04] give an short overview of various testability definitions. Including the measurement of Visibility Components (in short VC). This can be considered as an adapted version of the DRR[7] measure and was introduced by McGregor and Srinivas [MS96]. This measurement approach can already be used

[7] Domain-Range Ratio is a ratio between the cardinality of the domain to the cardinality of the range.

in the early stage of the development process. The objective of VC is to consider object–oriented aspects like inheritance, encapsulation, collaboration and exceptions. In order to calculate the correct VC value accurate and complete specification documents are required.

Another approach is "domain testability" described by Freedman [Fre91]. It is influenced by hardware testing strategies and based on the concept of observability and controllability. Objects and components have to support this characteristic in order to validate, if predefined test criteria are met with the received output. In some cases it is necessary to extend objects and components to fulfil the imposed requirements. The degree of testability is measured by the effort on implementing this additional requirements. An entire approach from Baudry et. al [BLS02] tend to analyse testability of UML class diagrams. Therefore, the various coupling and interaction between classes is used to measure and characterise the testability.

2.2. Testability in Object–oriented Software

The definitions of testability introduced before are not directly adaptable to our planned approach. This thesis focuses on analysing the compliance of structural design patterns in object–oriented systems. It is not the intention of the author to establish a new testability metric based on mathematical computation. Instead, the aim is to collect established guidelines on how to design object–oriented software in order to improve testability and reduce the effort on writing test cases. Within this fact it is important to define testability for the current approach as following.

2.2.1. Design for Testability

As mentioned earlier, this thesis focuses on investigation of the structural design of object–oriented systems in order to improve testability. Object–oriented programming allows developers to handle complex software components and break them down into small independent units. Each unit is responsible for a particular functionality of the software and are called classes in general. In coherence with the announcement of the programming language *Smalltalk* in 1993 [Kay93], Alan Key characterised object–orientation with three core concepts, which are (1) the support of inheritance, (2) encapsulation and (3) polymorphism [LR09]. Robert V. Binder describes in [Bin94] six major factors which influence the testability during the software development process: characteristics of the design documentation, characteristics of the implementation, built–in test capabilities, presence of a test suite, presence of test tools and the software development process capability respectively

maturity [Mul07]. Especially the factors regarding to the implementation characteristics and the presence of a test suite is important for the chosen research approach of the current work. If source code is testable with less effort is mainly influenced by the architecture and design of the source code.

2.2.2. Testability Design Patterns

There are different good and bad patterns of testable design in object–oriented programming (also called OOP) established in the literature over the past years. These guidelines determine the basis for the different analysis tasks, later on. Lasse Koskela defines different testability patterns in [Kos13] developer should be geared to. The Googler[8] Miško Hevery has been responsible for coaching developers to write easier to test production code in fact of the automated testing culture at Google. In the past, the guidelines published that reflect his experience on testability. At least, Roy Osherove explains in his book *The Art of Unit Testing: With Examples in .Net* [Osh09] the basics on writing unit test for the object–oriented programming language C#. Likewise Koskela, who provides the reader with a list of notable patterns on writing code that is testable with less effort.

Summarized all the mentioned literature above results in the collection of good and bad pattern follows below. With these design pattern we are able to phrase our own testability rules.

#1: Prevent complex, private methods [Kos13]. It is not possible to test private methods within an independent unit test case. The method which is declared as private is indirectly tested when they got called from a public method under test. Because of that, the convention says that you have to prevent writing complex methods that are declared as private. Possible violations of that pattern may be a huge amount of control flows (or even nested), loops or simply a large number of lines. The rule of thump is, to restrict the complexity of private methods to a point, that the output or side effect are understandable and no explicit test is required. Complex private method could be a hint to transfer the logic in a separate class.

#2: Prevent final or static methods [Kos13]. One of the core concepts of object–oriented languages is the inheritance. To declare methods as final prevent developers from creating subclasses of the related class in order to overwrite the included methods. Final methods in most cases are needless and only useful when you do not trust your colleges or yourself. In regards of testability, final methods are hard to test and

[8] A Googler is a person who is employed by the Google Inc.

only testable with much effort. There is no convincing reason to declare methods as final in the source code of a project. Declaration of static methods is another critical type for method declarations in fact of testability. Those methods are difficult to stub out[9] in test cases. Object-oriented programming and the usage of interface-based design has been a huge advantage for developers. Fake objects in test suites can override instance methods and return predefined result value necessary to create a simple test case. In contrast, static methods prevent the advantage of stubbing. It is not impossible but it is hard and test effort increases. As seen from perspective of testability it is not necessary to declare static methods.

#3: Do not use keyword new to often [Kos13]. This design pattern recommends the usage of *new* keyword in methods with care. To call new means the most general form of hard-coding in object–oriented languages. This statement initiated the constructor of an object and return the created instance. In this scenario a dependency on another class is generated which is not useful to write test case for testing the affected method. Another problem is the created instance in method under test can not easily replaced by a prepared fake object. A better solution would be dependency injection where the object is passed to the method under test through parameters.

#4: Prevent constructors with complex logic [Hev08, Kos13, Osh09]. The constructor in object–oriented languages is responsible to create a instance of the class. Every test case in a unit test suite will start with calling the constructor, in order to receive instantiated object of it. If the constructor already consisting of complex logic (like control flows or instantiating of other objects), it is hard to create the class instance under test. The guidelines encourage developers to leave constructors as simple as possible. Moving complex logic in a separate `init()` method will do the trick to simplify constructors and get the possibility to override it in test suite.

#5: Prevent the Singleton pattern [Hev08, Kos13, Osh09]. The Singleton pattern is often used to ensure that only one instance of the class exists at runtime. Another advantage of this pattern is the global access to methods of this instance at every place in the software components. At runtime this could be an advantage but in the test scenario you do not want the behaviour as described. In a test scenario this could lead to unpredictable results which are not really understandable. For example, the properties of the Singleton instance changed in first test case running in the test suite. In the `tearDown()` method of the JUnit[10] test the instance will

[9] Stub out means, to replace the method implementation in test case with a simple fake implementation.

[10] The JUnit test framework is the de facto standard unit testing library for Java. It contain several utilities in order to write unit test cases for the Java code [**?**].

be reset. The `setUp()` – runs for each test case – will receive a new instance of the Singleton. In the special case of Singletons the expected behaviour will not occur after the `setUp()` method. The returned instance is exactly the ones that have been changed by the test case before. Remind: the instance exists during the complete runtime.

#6: Prefer composition pattern opposite inheritance [Kos13]. Inheritance is a big advantage of a object–oriented programming language and helps to support code reuse. The drawback of inheritance is, that at instantiation in every class constructor the `super()` method is called. In view of testability these additional dependencies, produced by the class hierarchy, are awkward to improve testability of software components. Therefore, it is a good criterion to prevent a depth of inheritance greater than three times. The testability guideline suggest to use the composition pattern instead of inheritance. Thereby, the developer has the opportunity to use different implementations of the code at runtime.

#7: Wrap external libraries [Kos13]. Software projects often use external libraries to provide specific functionality or allows the integration of various utility components are helpful to implement software functions. A good example of a third party library is *Apache HttpCompoments*[11] for Java that simplifies the implementation of a HTTP Server. Those frameworks are often do not bring a design of good testability along. Therefore, it is a good idea to mask external libraries behind a wrapper class. In source code the communication with the library goes through your wrapper class which is replaceable easy in test cases.

#8: Prevent service lookups or static method calls [Kos13, Osh09]. It is a common practice to acquire the instance of a service by use of the Singleton pattern through a static method call. To stub out this hard–coded dependency in test cases is technically not impossible, but means much more effort. Passing the service object through parameters into the method is a better work around instead of a setter method for the service lookup class in order to pass through fake objects in test case.

#9: Prefer interface-based design [Osh09]. Substitute objects is a core topic on writing test cases without much effort. Is it easy to stub out objects for test purpose in your code base? Then the code is much easier to test if not. For that reason it is recommended to use an interface-based design instead of concrete class instantiation

[11] https://hc.apache.org, accessed on 24th June 2016

wherever possible. Passing objects via parameter into methods or constructors - so called dependency injection - makes it easier to fake those objects in tests cases. The next step is to change the type of passed parameters to interfaces. Create a fake the implementation of instances using interface-based implementation is much more easier than faking a concrete class.

#10: Do not declare class as final [Osh09]. When a class in Java is declared as final, this means that the class can not be extended. This behaviour could be useful when creating an immutable class. But in the purpose of writing testable code thit is not a good choice. As described earlier, the substitution and creation of fake objects to write simplified test cases is an important technique to increase testability. Since final class can not be subclassed, there is no way to overwrite the methods within.

2.3. Results of Testability Analyses

There are different possibilities to report results of the analysis tasks to the developers. The simplest approach is to print the various result terms to the command line interface. A more advanced result reporting will be the attachment of the extracted information from analysis tasks to a graphical user interface (GUI). In Chapter 3 the JTransformer tool will be introduced that already provides a GUI to attach analysis results. In the current approach it is planned to use these functionality by the developed analysis tool. Possible feedback from the examination can be:

1. the name of the matched rule,
2. the resource (e.g. class in Java) where the rule violation happened and
3. the line number in the corresponding class file

If the integration succeed, then developers have the capability to jump quickly into the flawed source code and can fix the highlighted location in the software component. The aim of the current work is not to calculate specific metrics which describe the testability, like Brutnik and van Deursen present in their publication *"Predicting Class Testability Using Object-Oriented Metrics"* [BVD04]. More generally, we want to show developers on which piece of code the testability is negative influenced as described before.

However, the rule language we want to implement give therefore flexibility for a future proposal. The form *"when* $\langle Premise \rangle$ *then* $\langle Conclusion \rangle$*."* allows to advices the then part to conclude with some scoring value, for example:

when something is wrong then score 10.

The idea behind conclusions of this kind is, to define a threshold which shall not be exceeded by the examination of the software components against the defined testability rules. This can be useful when the analysis tool is integrated in the continuous lifecycle environment of a project development process. If the threshold exceed during the daily build, the current process should be stopped and give feedback in the related log file or other possible outcome of the build process. At the moment, this use case is planned for feature work and will not be considered in the present work.

3. JTransformer

Günter Kniesel, senior lecturer at the Computer Science Department III of the University of Bonn, initiated the development of *JTransformer* in 2002. The original decision to realise this kind of tool was to provide the availability of code analysis on a logic-based infrastructure. In order to detect pattern in source code, the commitment of a general and formal correct definition of the pattern is necessary to guarantee a successful correct match. Against that, JTransformer want to go a step further. They want to abstract the formal definition so far that developer can implement their analysis pattern directly in form of executable code without thinking about mathematical formulations [SRK07]. As a first step, the named tool transforms Java projects into a logic-based fact database which builds up the basis for further analyses.

Since 2002, JTransformer is continuously refined over the years by different student assistants of the University of Bonn. The next big step after the project start was the first version of the plug–in for Eclipse, a well known Integrated Development Environment for Java. A short time after that, the extension got renamed to *Prolog Development Tools* (PDT) and contain all necessary features and functionalities to be a full-valued Prolog integration in Eclipse. This includes a project explorer, an editor with syntax highlighting and code completion, an integrated Prolog console as well as the integration of the SWI-Prolog Debugger. In addition to implement own analyses, JTransformer give the developer also the availability to execute transformations on the current Java program code. Those transformations are realised on manipulate created factbase or change existing facts in order to fulfil the transformation. But this feature will not really be important for the planned research topic, so it is not necessary to go more into detail.

In this section we want to get more familiar with JTransformer and demonstrates, how the existing tool chain will help and provide us with the research topic. The already suggested factbase, generated from a Java project, represents the foundation on which we want to examine our phrased testability rules later.

3.1. Abstract Syntax Tree as a Factbase

As defined in Section 2.2, the testability of software components, written in an object–oriented language, could be improved by the compliance of several design guidelines. Those guidelines are nothing else than a description of patterns in a more general way which could be discovered by analysing the source base. Before examining the code base of a software project is possible a more abstract representation is necessary. It does not make sense to run different kind of analysis directly on the given Java code. On the other hand the AST derived by parsing the source code is a suitable representation to examine, if it obeys the predefined testability rules. Today's compiler are mainly separated in two parts: (1) the so called "Front end" that contain the lexical analysis, parsing the source code and at least a semantic analysis and (2) the "Back end" that consists of a generic analysis, the optimisation phase and finally the code generation [ASU86]. In the syntax analysis step the compiler is parsing the token sequentially in order to identify the correct syntactic structure of the program. Because in this phase there is a more general representation necessary, the compiler parses the code and create the concrete syntax tree (CST) where the AST is derived from and used as the internal representation for further analysis steps. Figure 3.1 shows the abstract syntax tree of a very simple Pseudo code snippet.

Martin Fowler describes in his book *"Domain Specific Language"* [Fow11] the advantages and useful use cases for this representation type of source code in detail. The AST will be derived from parsing the source code according to the programming language grammar. Parsing in general is a strong hierarchical operation, where source code is simply read as text, token by token. The result of this step will be a tree structure including hierarchical information. This type of tree representation is called syntax tree (or parse tree). Iterating through the tree - composed of nodes and edges - allows the examination of hierarchical structure and draw conclusions about the design patterns introduced in Section 2.2.2. With use of the extracted abstract syntax tree we are able to validate the satisfaction of design patterns and conventions established in the chapter before. For example, if the constructor of a class contains the call of another constructor – identified by the keyword `new` – this can be discovered from the AST. Equivalent to the previous characteristic, all other information are obtainable, too.

JTransformer provides the functionality of creating a factbase from existing Java projects. Thereby the structure of a previously described abstract syntax tree breaks down into small unique identifiable and logic elements that are defined as Prolog predicates in the factbase. These AST elements are called *Program Element Facts* (or in short: PEFs). An better overview can be provided by giving a short example of

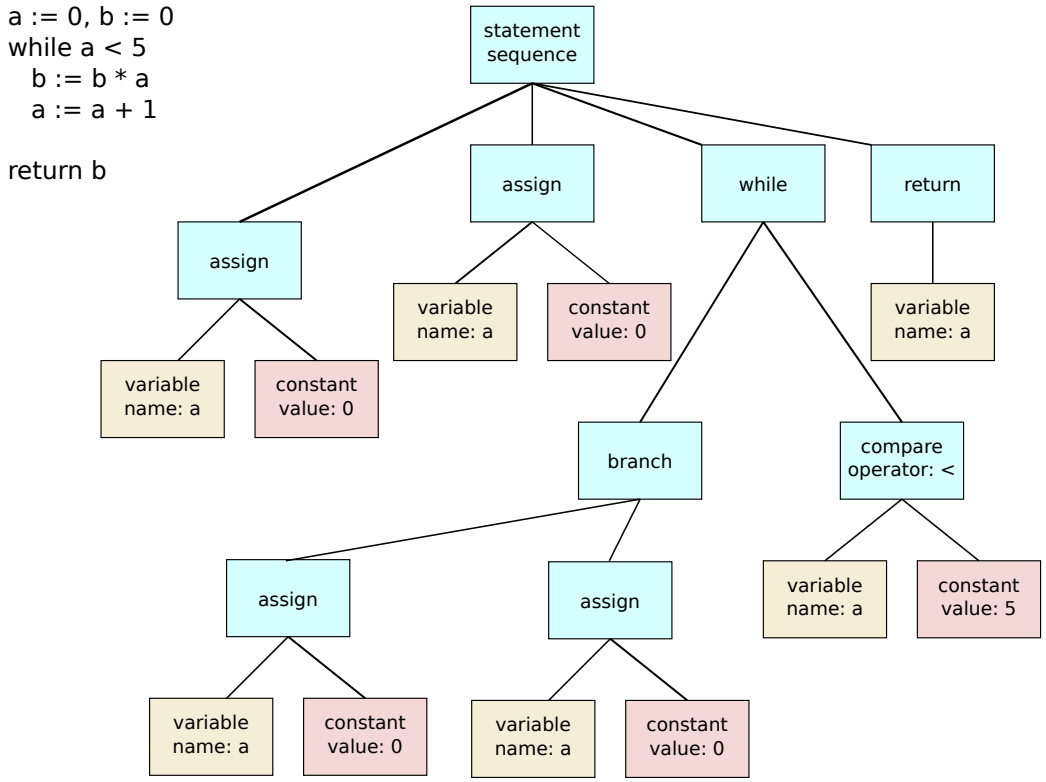

Figure 3.1.: Abstract syntax tree derived from simple Pseudo code

JTransformer's creation process that involves internal representation of the abstract syntax tree and as result the created list of PEFs. Assuming that we have a simple class named *AstExample* which contain private attributes and different declaration of methods. This simple class is shown in Listing D.4.

The Eclipse PDT plug–in gives developers the availability to transform the current Java project into the internal factbase representation of JTransformer. Listing 3.2 contains a shorten output of the created PEFs generated from JTransformer using the example code from Listing D.4. All syntax tree nodes are available as a single fact in the Prolog database and is from now on accessible within a query, that is well-known from SWI–Prolog programs. To each type in Java – like method call, constructor declaration, method definition, field access and so on – there is an equivalent in the generated factbase.

Listing 3.1: Simple implementation of a Java class

```java
package de.uniwue.thesis;
import java.util.ArrayList;
import java.util.List;

public class AstExample {
  private List<Object> mNodes;
```

```
7
8   AstExample() {
9     this.mNodes = new ArrayList<Object>();
10  }
11
12  public Object getNode(int index) {
13    return this.mNodes.get(index);
14  }
15 }
```

Line 3 of the listing shows declaration of the class *AstExample* represented by the fact `classT(ID, Parent, 'ClassName', ParamRefs, Definitions)`. First parameter of every PEF is an identifier, which is a unique integer value in the factbase. Another code snippet from our example class could be reconstructed by the facts `methodT(ID, Class, 'MethodName', ...)`, `paramT(ID, Parent, Type, 'ParamName')` and `modifierT(ID, Parent, 'modifier')`, see line 4, 7 and 11. These Program Element Facts describe the *getNode()* method declaration of Listing D.4 on line 12. All available PEFs generated by JTransformer are documented on the official project website[12].

Listing 3.2: Example factbase of a Java project

```
1  compilationUnitT(28591, 28594, 28593, [28595, 28596], [28597]).                PROLOG
2  packageT(28594, 'de.uniwue.thesis').
3  classT(28597, 28591, 'AstExample', [], [28598, 28599, 28600]).
4  methodT(28600, 28597, getNode, [28613], 10001, [], [], 28614).
5  constructorT(28599, 28597, [], [], [], 28605).
6  fieldT(28598, 28597, 28603, mNodes, null).
7  paramT(28613, 28600, 10080, index).
8  newT(28610, 28608, 28599, null, [], 21199, [], 28612, null).
9  returnT(28616, 28614, 28600, 28617).
10 modifierT(28604, 28598, private).
11 modifierT(28615, 28600, public).
```

For now, we got a short overview through the JTransformer project and their operating principles. Next, the features of the Eclipse plug–in, which are necessary for the code analysis approach, shall be introduced in a nutshell. After that we are going to present an example implementation on how to write an analysis using JTransformer as envisaged from Günther Kniesel and his team. That allows us to get more in touch with analysing Java code using the different PEFs the abstract syntax tree

[12] `https://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/4.1/java_pef_overview`, accessed on 7th July 2016

exists of and the SWI-Prolog programming language. The usage of the factbase by the code analysis tool will be focused of Chapter 6.

3.2. PDT - The Eclipse Integration

In 2004 the first version of the Eclipse plug–in for JTransformer were introduced by the team. The integration into an existing Java IDE has improved the development of analyses and the evaluation with the tool significantly. Creating the JTransformer factbase for each available Java project, the definition of analysis and running those against selected factbase are only the core features of the plug–in. Another point is the good visualisation which is available in Eclipse to view the analysis results or main factbase information that come together with the integration. In the following, the key features that are useful for the YaCI tool will be describe in detail as follows.

In order to run an analysis, the creation of a factbase which can be queried by Prolog is required. The JTransformer Developer perspective in Eclipse offers the option to create those factbases when assign the existing Java project to JTransformer. Use *Right click on project* → *Configure* → *Assign JTransformer Factbase* to initialise the process. After assertion the construction of the abstract syntax tree – in form of PEFs – are automatically started and are available via an initiated SWI–Prolog process. Eclipse Java projects are assigned to JTransformer and contain AST representation results in a drop-down list of available factbases in the Control Center view.

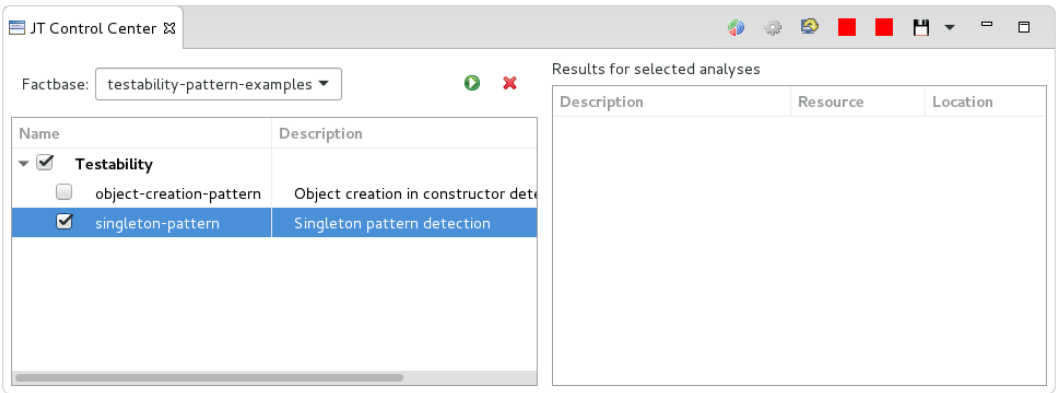

Figure 3.2.: Overview to the JTransformer Control Center

The Control Center view is not only responsible for switching between factbases. Other content which is displayed there, is the list of available analysis definitions created by the user and the result of already executed analyses. Both are shown in Figure 3.2. Each entry in the analysis table is checkable in order to enable or disable

the analysis for the next run. An entry in the table is available, if the analysis definition is loaded in the factbase. How to add definitions is described in Section 3.3: *Implementation of an Analysis Pattern*. On the right hand side of the Control Center the results are in another table perspective. For each analysis result the table contents the following information: a short *description* of the result (defined by developer at analysis implementation step), the *resource* (e.g. the class name) where the match was found and the *location* (line) of the matched analysis result in its resource.

Another considerable feature of the Eclipse plug–in is the ability to export the created factbase to local Prolog file in order to consult it in own SWI-Prolog modules. A statistics view give developers a short overview to the PEF factbase and options to filter these by fact–type, e.g. `methodT/8`, `constructorT/6` and so on. We plan to integrate our YaCI tool to the JTransformer plug–in, especially to use the user interface to give developers direct feedback from analysis examination.

3.3. Implementation of an Analysis Task

In general, the definition of an JTransformer analysis consists of two parts: (1) add analysis and their result to graphical user interface and (2) implement the logical part using PEFs to match analysis on the AST. A work through of the Singleton pattern analysis – available in the JTransformer examples – helps to get in touch with the Prolog based tool. Remembering, the Singleton pattern mentioned already in Section 2.2 where the testability design guidelines were introduced.

Before writing the first analysis implementation the question have to be clarified which characteristics are useful to identify the Singleton pattern in a Java code base. The common way of the Singleton implementation is, to declare the constructor of the class as private. This prevents to create instance from outside of class. Second, a method has to be declared as public that has the return type of the class itself. A private field declaration with type of Singleton class is the last characteristic to identify this pattern. Summarized, these are the a characteristics in a nutshell:

- private constructor
- public method which return the class and
- a private field of type is equal to class

Listing 3.3 show a simple analysis implementation using the different PEFs available in JTransformer factbase.

Listing 3.3: Singleton pattern analysis implementation

```prolog
:- module('singleton.analysis', [
  classMethodReturnsOwnInstance/3
]).

classMethodReturnsOwnInstance(Type, Method, Field) :-
  methodT(Method, Type, _MethodName, [], Type, _, _, _),
  modifierT(_, Method, static),
  fieldT(Field, Type, Type, _FieldName, _Init),
  modifierT(_, Field, static),
  containsFieldAccess(Method, Field).


containsFieldAccess(Method, Field) :-
  fieldAccessT(_ID, _Parent, Method, _Receiver, Field, _Type),
  ! .
```

The term `classMethodReturnsOwnInstance/3` returns the unique identifiers of the three distinguishing characteristics of a Singleton, after the unification of the variables during the SWI–Prolog process: the class itself (`Type`), the method which returns instance of the Singleton (`Method`) and the private field, that store the Singleton (`Field`). The defined predicate can be already called interactively via the associated Prolog console. The result of the query are the unified unique identifiers which are not very helpful and hard to read by a human. Therefore JTransformer offers a specific analysis api with two important predicates to use: `analysis_definition/5` and `analysis_result/3`. Both are used in Listing 3.4 to add the Singleton analysis implementation to the JTransformer GUI.

Listing 3.4: Attach analysis definition to JTransformer

```prolog
:- use_module('singleton.analysis').

analysis_api:analysis_definition(
  'singleton-pattern', onSave, info,
  'Testability', 'Singleton pattern detection').

analysis_api:analysis_result('singleton-pattern', _, Result) :-
  classMethodReturnsOwnInstance(Type, Method, Field),
  (
    make_result_term(class(Type), 'Singleton class', Result);
    make_result_term(instanceMethod(Method), 'The instance method', Result);
    make_result_term(instanceField(Field), 'The instance field', Result)
  )
```

After adding an analysis definition to JTransformer, as seen in line 3 of the listing, a new entry with the label “singleton-pattern” in the analysis list appears. Second parameter of definition says that the analysis should always run after save action is triggered in Eclipse. Third parameter determines which marker in Eclipse IDE should be displayed on matching elements (e.g. info, warning or error). With the last two parameters the developer can define the group label where the analysis belongs to and a simple description of the anlysis. Both are displayed in the Control Center of JTransformer. Finally, line 7 shows the `analysis_result/3` predicate which wraps up the results of analyses in order to display them in the JTransformer GUI. The first parameter determines the name of the analysis, the results belongs to and have to be the same as defined in `analysis_definition/3` clause. In the body of the `analysis_result/3` clause the implementation of the singleton pattern detection is called and the unified ast-element-identifiers are wrapped into a term using the `make_result_term/3`. How the control center looks like when analysis pattern matched is shown in Figure 3.3

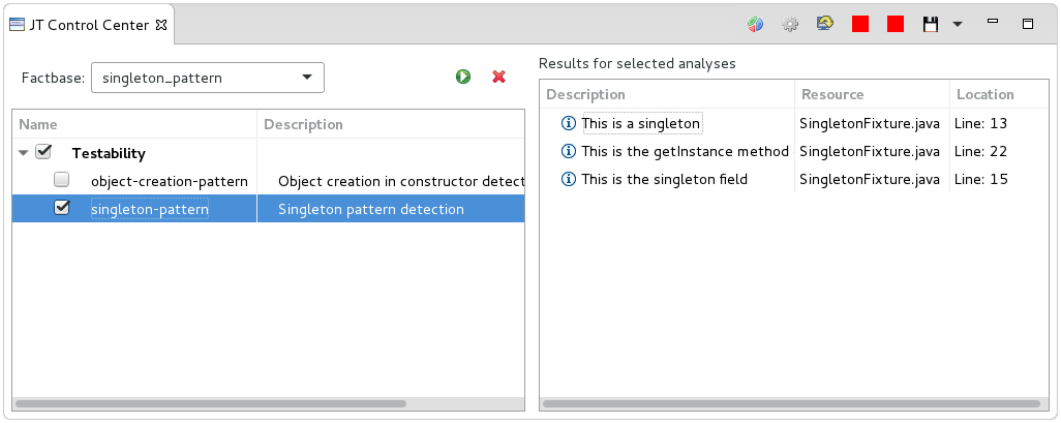

Figure 3.3.: Control Center contain result info from Singleton pattern analysis

This knowledge about how JTransformer works and analysis definitions, as well as their results could be attached to the graphical user interface is important for the current approach. Later on, developers should have the possibility to run the automatically generated analysis from rules and review the outcome of the analyses via the Eclipse plug–in. The integration of YaCI into JTransformer is part of Section 6.5.

4. The YaCI Rule Language

The definition what does testability of software components in the world of object–oriented programming languages mean is introduced in Chapter 2. There are also important design patterns determined which help to write code that is testable with less effort. Because of the goal, that not only developers can implement their own analysis pattern for the examination part of the analysing tool, there is a more abstract and general language necessary to phrase analysis rules. A specification of a natural language similar to the English once is required.

For this purpose, the current chapter introduces the *YaCI Rule Language* and its related grammar. Parsing phrased rule sentences – represented as simple strings – requires a closer look into Definite Clause Grammar and quasi quotations techniques which are available in SWI–Prolog. They are suitable to process strings token by token. Result of parsing is the generalised internal representation of rules which will be introduced in Chapter 5.

4.1. Grammar Specification

Phrasing a valid YaCI Rule assume that two main parts are always used in the descriptive sentence. The rule obligatory starts with the keyword “when”, followed by a sequence of allowed words described later. This section of the rule is called the *rule premise*. As the closing of a valid rule wording, there has to be a section introduced with the keyword “then”, that induces a logical implication of the sentence – the *conclusion*. Finally, the dot closes the rule sentence. This form is known as the grammatical structure named *conditional sentences*. Figure 4.1 shows the general form of the YaCI Rule definition, that can be handled by the implementation of the rule parser.

$$
\begin{aligned}
\langle \text{rule} \rangle &\models \text{when } \langle \text{premise} \rangle \text{ then } \langle \text{conclusion} \rangle \text{ .} \\
\langle \text{premise} \rangle &\models \textit{specified later} \\
\langle \text{conclusion} \rangle &\models \textit{specified later}
\end{aligned}
$$

The word sequence after "when" are expresses a hypothetical situation, or factual implication, which is also known as premise. It is possible to concatenate different premisses or only name one of these. In contrast of the premise sequence the "then" keyword can only be followed by a single conclusion. The introduced conditional sentence form will be used as the structure for phrasing testability rules in the current approach. In order to give end–users the capability to phrase valid testability rules, a specification of a grammar is necessary. Transformation of the string sentence into internal representation is also done during the parsing process using the grammar. Therefore the premise and conclusion must be written accordingly to the grammar we introduce in Section 4.1.1 and 4.1.2.

In Section 2.2.2 the introduced testability design guidelines shall be expressible within the rule premise. At this point we want to be one step ahead and provide a flexible language specification that allows to phrase a wide variety of rules and not only to be limited to the context of testability. The conclusion of the rule takes part as what happens when the defined premise fails while examination against the abstract syntax tree.

A formal grammar type is necessary to write down a specification of the YaCI Rule language. For the proposed approach we use a *context–free grammar* (CFG) to define a set of production rules, in order to describe all possible sequences of words for phrasing rules. CFGs consists of a set of rules which describe possible replacements. Available components for the body of a production rule are:

- *Terminals* which represent symbols of the alphabet of the language being defined,
- *Non–terminals* that represents different type of clause in the grammar sentence and
- a *start symbol* is a variable that represents the complete language being defined. The start symbol is always the first variable in a grammar specification.

The reason for using a conditional sentence as the basic structure of rule wording is the fact that these are already fulfilling the requirements for the given approach of phrasing testability rules. In the premise the end–user is able to describes the specific design pattern and how it is reflected in object–oriented programming languages. The conclusion part permits to define the desired action when the premise is evaluated to true. Especially for rule based systems these conditional sentences are established and usually used. Randall Davis and Jonathan J. King define in [RK84] that rules in a pure production system can also be viewed as simple conditional statements. Furthermore, they specify that one side (the premise of the rule) is evaluated with

reference to the data base and, if succeeds, then the action (the rule conclusion) is executed.

4.1.1. Premise of the Rule

The core part of the rule language is the *premise*. Within it the user can describe the pattern, respectively coding quirks, the analyse tool shall find in source code under test. A simple, but powerful description language for this part is necessary to achieve a flexible way of examine the abstract syntax tree of the software component. First version of the YaCI Rule language must contain all essential conditions and language–based quirks to phrase the testability design pattern with the language specification defined in the next step. As a short reminder, the analysis tool in its first version is restricted to the object–oriented programming language Java. Therefore the grammar is adjusted to general structure and language depended characteristics of object–orientation, especially in the context of Java. A short assessment, if the specified grammar is compatible or can be adapted to other programming languages will be part of future work section later in this work.

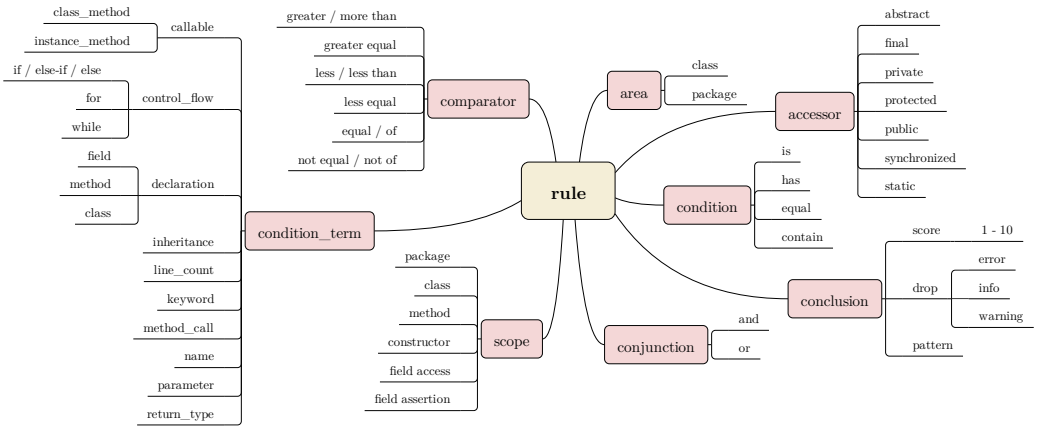

Figure 4.1.: Overview to the different components available for rule phrasing

The structure of every rule is based on components which can be combined in a restricted but flexible way. An overview of the available components and their relations are shown in Figure 4.1. The premise itself starts always with one of the available scopes. Possible scope elements are for example `class`, `method` or `constructor` and are adapted from Java. After the scope, a condition construct is expected by the grammar specification. Conditions are useful to describe more in detail, what pattern, keyword or maybe some other language quirk has to be considered in order to fulfil the condition in respect to the selected scope. The condition can be introduced with one of the five available keywords: `is`, `has`, `contain`, `equal` or `not`

`equal`. Therefore the basic structure of phrasing simple rule for a single scope and condition is established. To satisfy the requirement that developer can describe more complex design pattern the support of chaining scopes and conditions must be defined. In general, this mean that every rule premise can handle both, a single scope and a list of scopes, chained with allowed logical conjunctions: `and` respectively `or`. Furthermore the scope also can be composed of a single or chained condition. The logical conjunction to chain scopes are also available for conditions. A brief overview gives the simple example shown bellow:

(1) *when* ⟨*Scope*1⟩ *and* ⟨*Scope*2⟩ *then* ⟨*Conclusion*⟩.
(2) *when* ⟨*Scope*⟩ ⟨*Condition*1⟩ *and* ⟨*Condition*2⟩ *then* ⟨*Conclusion*⟩.

At this point, the premise of the rule can consists of a scope and the associated condition. In order to complete the conditional part, a detailed specification of the misbehaviour in source code is necessary. Therefore the condition can be specified by a wide variation of allowed terms in respect of the comparison against a predefined integer or a string value. Details on how to phrase valid conditional parts can be derived from the grammar (see Listing 4.2 near to the production rule *condition*). A simple example condition on the scope of a method can be as follows:

when method contain line_count greater 3 then score 1.

While examination of the AST this condition will match on every method, when the body of the related method include three lines of code or more. It is very easy to restrict the condition on the given scope a little bit more. This can be realised by adding another condition to the related rule scope using an available conjunction:

when method contain line_count greater 3 and contain control_flow more than 3 times then score 1.

This extension of the condition results in a more restricted result set after running the analysis. At least, this rule expresses that only methods are relevant whose body covers more than three lines and additional there have to be a control flow more than three times in the method implementation (e.g. if/else, while, for). The premise including the two compound conditions is simple phrased as seen above.

In the rule wording the premise part is the most complicated section because of the huge variety of coding quirks and the availability of different scopes and condition terms. Next, it is described in detail what conclusions are possible and how to phrase them within a YaCI Rule.

Figure 4.2.: Specification of the YaCI Rule premise grammar

⟨rule⟩ ⊨ when ⟨premise⟩ then ⟨conclusion⟩ .
⟨premise⟩ ⊨ ⟨scopes⟩ ⟨conditions⟩
⟨scopes⟩ ⊨ ⟨scope⟩ | ⟨scope⟩ ⟨conjunction⟩ ⟨scopes⟩
⟨scope⟩ ⊨ class | constructor | field | method | package
⟨conditions⟩ ⊨ ⟨condition⟩ | ⟨condition⟩ ⟨conjunction⟩ ⟨conditions⟩
⟨condition⟩ ⊨ contain ⟨contain_term⟩ | is ⟨accesor⟩ | has ⟨has_term⟩
⟨conjunction⟩ ⊨ and | or
⟨contain_term⟩ ⊨ access of ⟨accessor⟩ ⟨scope⟩ | call of ⟨callable⟩ ⟨comparison⟩ | control_flow ⟨comparison⟩ | declaration ⟨scope_combi⟩ ⟨comparison⟩ | keyword ⟨keyword⟩ ⟨comparison⟩
⟨has_term⟩ ⊨ declaration of ⟨scope_combi⟩ | inheritance of depth ⟨numbers⟩ | line_count ⟨comparison⟩ | name ⟨string_comp⟩ ⟨chars⟩ | paramater of type ⟨chars⟩ | return_type ⟨comparator⟩ ⟨chars⟩
⟨callable⟩ ⊨ class_method | instance_method | ⟨scope_combi⟩
⟨comparison⟩ ⊨ ⟨comparator⟩ ⟨numbers⟩ | λ
⟨scope_combi⟩ ⊨ ⟨accessor⟩ ⟨scope⟩
⟨accessor⟩ ⊨ abstract | final | private | protected | public | static | synchronized
⟨comparator⟩ ⊨ greater | more than | greater equal | less | less than | less equal | equal | of | not equal | not of
⟨string_comp⟩ ⊨ equal | not equal
⟨keyword⟩ ⊨ new
⟨numbers⟩ ⊨ ⟨numbers⟩ | ⟨number⟩ ⟨numbers⟩
⟨number⟩ ⊨ 0 | 1 | 2 | ... | 8 | 9 | 10
⟨chars⟩ ⊨ ⟨char⟩ | ⟨char⟩ ⟨chars⟩
⟨char⟩ ⊨ [a-z] | [A-Z]

Figure 4.3.: Specification of the YaCI Rule conclusion grammar

$$
\begin{aligned}
\langle \text{rule} \rangle &\models \text{when } \langle \text{premise} \rangle \text{ then } \langle \text{conclusion} \rangle . \\
\langle \text{premise} \rangle &\models \textit{already specified} \\
\langle \text{conclusion} \rangle &\models \text{score } \langle \text{number} \rangle \mid \text{drop } \langle \text{severity} \rangle \mid \text{pattern} \langle \text{chars} \rangle \\
\langle \text{number} \rangle &\models 0 \mid 1 \mid 2 \mid \ldots \mid 8 \mid 9 \mid 10 \\
\langle \text{severity} \rangle &\models \text{error} \mid \text{info} \mid \text{warn} \\
\langle \text{chars} \rangle &\models \langle \text{char} \rangle \mid \langle \text{char} \rangle \langle \text{chars} \rangle \\
\langle \text{char} \rangle &\models [\text{a-z}] \mid [\text{A-Z}] \mid [\text{0-9}]
\end{aligned}
$$

4.1.2. Conclusion of the Rule

To paraphrase a complete and valid YaCI Rule there is also a conclusion necessary. When the premise of the rule is matched while abstract syntax tree examination, the conclusion has to be evaluated. In the first implementation of the grammar it can be process three different consequences, when the premise failed:

- Severity
- Scoring
- Pattern detection

Listing 4.3 illustrates the specified grammar definition for the conclusion part of the rules. The reason to provide YaCI with different rule consequences is because of the various possible applications in the analysis area. The main goal in the current work is to show different severity in an Integrated Development Environment like Eclipse using problem markers. But later on, the integration in the continuous life cycle of the software development process is conceivable, too. For this purpose using different score values as the consequence is maybe the more convincing way.

Severity: Herewith, the level of feedback in modern Integrated Development Environment tools can be defined by using different problem marker levels. For example in Eclipse, when compiler found unused variables in class declaration, then the marker with the level *warning* is displayed in the editor view (see Figure 4.4). The severity levels in the specified grammar are adapted from the problem markers available in the Eclipse editor. These are *info*, *warn* and *error*. To advise the analysis tool to create the marker using the different levels of severity the rule must conclude with a conclusion contain “drop” and the associated severity level.

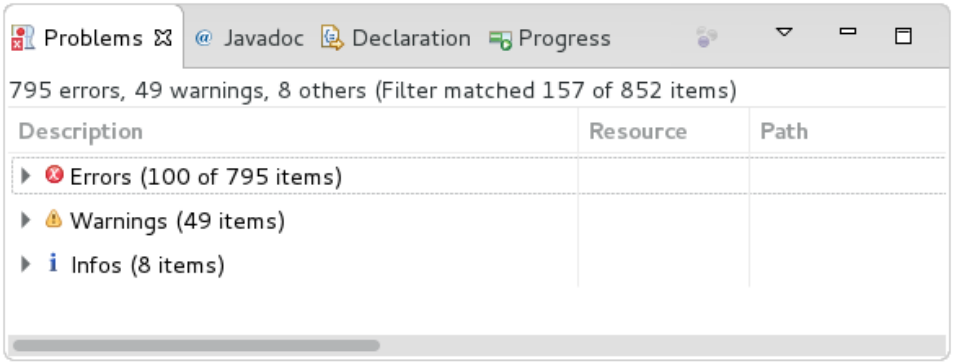

Figure 4.4.: Available severity problem markers in the Eclipse IDE

Scoring: The grammar for conclusions also provide the specification of a variable score value that has to be an integer between 1 and 10. Because of the actual development situation of the YaCI tool the scoring is mapped back to severity, as described above. A meaningful usage of score values can be the integration into the continuous integration life cycle of software components. But this approach will be left out for the present and is not considered in the current work. The mapping from scoring to severity is defined in the following way:

- from **0** to **3** → info marker
- from **4** to **6** → warning marker
- from **7** to **10** → error marker

Pattern detection: The third and last possible conclusion of a YaCI Rule can induce the matching of a design pattern. In order to examine a pattern the rule must conclude with *"... then pattern "⟨Name⟩"."*. The result of this rule covers the different indicators in source code belongs to the phrased pattern. The well-known Gang of Four introduce in their book *"Design Patterns: Elements of Reusable Object–Oriented Software"* [GHJV95] different recurring solutions to common problems in software design. One example of a possible pattern is the *Singleton* (envisaged in Section 2.2) to restricts the object creation of a class to only one available instance at runtime.

4.1.3. Description of the Rule

The YaCI Rule language provides an additional description written as free text. This simply allows the user to describe the rule in his own words. Another point is to improve the readability of the set of rules collected in a `*.rules` file. Later, the rule

Figure 4.5.: Specification of the YaCI Rule description grammar

⟨rule⟩ ⊨ ⟨description⟩ @ ⟨rule⟩ .
⟨description⟩ ⊨ ⟨long desc⟩ ⟨short desc⟩ | ⟨short desc⟩
⟨long desc⟩ ⊨ /* ⟨multi line⟩ */ ⟨return⟩
⟨short desc⟩ ⊨ ⟨chars⟩
⟨multi line⟩ ⊨ ⟨chars⟩ | ⟨chars⟩ ⟨return⟩ ⟨multi line⟩
⟨chars⟩ ⊨ ⟨char⟩ | ⟨char⟩ ⟨chars⟩
⟨char⟩ ⊨ [a-z] | [A-Z] | [0-9] | “ ”
⟨return⟩ ⊨ \n

file is used by the JTransformer YaCI integration to read the testability rules in and display it in the JTransformer Control Center (see Section 3.2). The JTransformer Eclipse plug–in provide two types of description: a short and a long one. Listing 4.5 shows the grammar specification of the rule description, while Listing 4.1 gives a short instruction on how to provide a YaCI Rule with a description.

Listing 4.1: Different ways to provide YaCI Rules with a description

```
Short description @ when method is private then drop warning.                RULES
/* Long, one-liner description */
Second rule @ when constructor has line_count greater 10 then drop error.
/* Long description
over to lines or more */
Another rule @ when class is final drop info.
```

4.1.4. Example Rule

In the last sections the available rule components were introduced. As a consequence of this an essential knowledge on how to structure and write valid rules based on the YaCI Rule grammar is acquired. For a better overview on how to construct YaCI Rules a brief example shall be introduced next. Ahead of writing an own rule there have to be two questions answered by yourself: *“What kind of bad smells in code is expected as a resulting match by the rule?”* and *“How to identify this circumstance in source code?”*

For example, a simple rule shall identify all methods which are private and named “getInstance”. This thought sets up the requirements on writing a valid YaCI Rule and is all what is necessary for the beginning. As a short reminder: every premise starts with a scope followed by a single condition or chaining of different conditions.

The instruction *"identify all methods"* indicates that the scope of the rule has to be of type *"method"*. Towards the grammar specification after the scope a condition is followed. The requirement says there have to be two conditions necessary to fulfil it related to the named scope:

- only methods declared as `private` and
- the plaintext name of the method specified in source code

This results in an logical and conjunction contains the first condition ("is private") to observe only private declared methods and the second one determining the method name ("has name 'getInstance' "). The final rule that is specified using the introduced rule grammar looks like the following conditional sentence:

when method is public and has name "getInstance" then drop info.

The purpose of this small example is to demonstrate the flexibility and cardinality of the defined rule grammar to write an arbitrary set of rules not only limited to the context of testability. As an introduction and to start testability analysis from scratch, the testability design patterns – which are presented in Section 2.2.2 – are already available as YaCI Rules. An overview to the phrased rules is available in Section 4.3.

4.2. Parsing Natural Language

The domain–specification language of phrasing YaCI Rules shall be as closely as possible to the English language. To realise this purpose a grammar is specified in Section 4.1 to determine the frame of wordings for testability rules. Before these sentences can be used to run the analysis process they have to be parsed using the specified grammar. This step will transform them into the internal rule representation that can be used as input of the YaCI Analysis tool. Using an internal representation has the advantage regarding to different capabilities in the community. This aspect removes the limitation to only use outcome of the parser within the YaCI toolchain. Section 5.2 describes detailed why it is useful to introduce an adaptive representation of the parsed rules. The input that is passed through the parser is in form of the conditional sentence listed in a simple text file. Achieving the goal to handle a set of rule strings the parser reads the file line by line and applies the grammar to the token sequence. A short introduction on parsing English language sentence using SWI–Prolog is already introduced by Alan Bundy and Lincoln Wallen in [BW84]. For this purpose the *Definite Clause Grammar* (in the following named as DCG) is used. Matter of the next section is to explain in short

what a DCG is and how it is used to define a grammar in SWI–Prolog. Afterwards the implementation is shown more detailed in context of the YaCI Rule parser.

4.2.1. Definite Clause Grammar

The starting point of solving the parser problem is to handle the specified rules shown in Section 2.2.2. Each rule is represented as a simple string in the rule file. This can be a single line for each rule, if there is only a short description available, or multiple lines, if a long description is attached to the rule. The parser processes the sequence of tokens according to the grammar specification that is already introduced. Especially in the context of the logical programming language Prolog, using DCG is a well–known and proved approach to write rules for the parsing process. They also improve the ease to specify a context free rule grammar in a more readable notation. In order to use YaCI Rule Parser and the related DCG as well as the quasi quotations, a local installation of SWI–Prolog 6.4.0 or newer is necessary. An additional DCG library *basics.pl*[13] is also available through the official package source. This package adds basic functionality and different general utility predicates which help to implement Definite Clause Grammar in SWI–Prolog. Loading it via `consult/1` into the Prolog file it will frees up the different utility clauses.

The DCG exists on a set of Prolog structures in the ordinary form of "`head --> body`". Instead `:-` the body is separated from the head using the special notation `-->`. As gathered from the official DCG documentation, the body of a rule can contain three different types of terms:

- a simple reference to another DCG grammer rule,
- code, that have to be interpreted as native Prolog code (must written between `{...}`) and
- a list, that is interpreted as a sequence of literals.

Listing 4.2 illustrates a possible implementation of the basic YaCI Rule grammar in a simplified way using DCG notation in SWI–Prolog. The detailed implementation will be part of the next section. There are two different rules specified: Line 1 to 5 implements the grammar for rules contains a description before *"when ... then ... "* part is expected. The other way round in line 7 to 14 the grammar for rules without description is specified. Another important difference is the head of the rule. Within the second one the normally unbound variable `Rule` will be unified during the parsing process and returns the unification of the variable. Using this programming style

[13] `http://www.swi-prolog.org/pldoc/doc/swi/library/dcg/basics.pl`, accessed on 20th July 2016

will help to realise the planned internal rule representation. As well, the example above contains all three available variations of terms in the rule body. A reference to another DCG grammar rule `rule_description//1` is illustrated in line 2. Instead of the conventional list notation in Prolog (e.g. `[itemA, itemB, ...]`), the sequence of characters is embraced by quotation marks and represents a valid list of characters. For instance, the internal representation of the quoted string `"when"` is simple a list notation like `["w","h","e","n"]`. Finally, line 10 to 13 explains the usage of code in the grammar body that shall be interpreted in native way by the Prolog process. Especially for generating the outcome from the parsing process this notation will be helpful to create the internal Prolog structure of YaCI Rules which consists of name and different arguments. These arguments themselves can be again literals and atoms.

Listing 4.2: Simple Definite Clause Grammar implementation in SWI–Prolog

```
1  rule -->                                                    PROLOG
2    rule_description, "when", rule_premise, "then", rule_conclusion, "." .
3
4  rule(Rule) -->
5    "when", rule_premise, "then", rule_conclusion, {
6      % build up rule representation
7      Rule = rule(when(), then())
8    }, "." .
9
10 rule_description --> % ...
```

The basic knowledge about DCGs and how to use them in Prolog helps to understand the implementation of the YaCI Rule parser. The following section takes a closer look at the technical implementation of the parser and how to specify a grammar using the previously described approach in Prolog.

4.2.2. Implementation

At the starting point of the grammar parser there is a local rule file containing a set of testability rules, in order to parse them into the internal YaCI Rule representation (see Chapter 5). As described earlier each rule can consist of three parts:

1. optional, long description
2. required short description
3. and the rule sentence itself according to the grammar specification

The indicator for the end of a rule is simply a dot followed by a line break. Processing the file is done using an input stream that reads the file token by token until the end of the stream is reached. The result of parsing the text file will be a list of rule terms transformed into the internal structure.

The set of rules in the current work are provided through a local file. There is already a helpful library available from official SWI–Prolog package source called *Pure I/O*[14]. It deals with processing input streams from local resource using grammar rules. New predicates are provided with the package which can be used in Prolog. For the current approach `phrase_from_file/2` is used to read text from file sequentially and parse them according to the DCG definition which is passed through the first argument. The second argument is used to determine the absolute path where the rule file is located on local hard disk. Prolog will open an input stream for the given file. There is also an implementation of `phrase_from_file/3` available which allows to determine different options that have to be recognised within the read and write operations.

Before we get more in detail on the implementation of the Definite Clause Grammar there shall be introduced some general Prolog notation hints. When talking about DCG in Prolog this means in general a set of rules which are used to parse a given string. Defining DCG rules is equal to the ordinary clauses syntax are known from Prolog using `:-/2` for separating head and body of the clause. Instead of `:-/2` the head and the body of a DCG rule is separated by `-->/2`. The Prolog preprocessor uses `expand_term/2` to translate DCG rules into ordinary Prolog clauses. Two additional arguments are added to the converted terms which represent the difference lists[15] of the parsed string. The arity of a DCG rule is declared by `//` after the name. For instance, the grammar rule `rule(Result) -> ..` is declared using the predicate `rule//2`.

Back to parsing the read-in rule sentences. Entry point of the grammar specification is the `parse_rules//1` predicate. It has one argument that is unified after successfully parsing the text file and holds all created rule predicates as the result of the parsing process. As seen in Listing 4.2 the `parse_rules//1` references to `blanks//0` and `list_of_rules//1`. The referenced `blanks//0` rule is a helper from the basics.pl utility library that holds various general utilities for processing strings. For instance `blank//0` and `blanks//0` used to skip white-space characters in the sequence of tokens. For processing a set of rules the `list_of_rules//1` is referenced. Two different scenarios are available during traversing through the input stream. Firstly, if

[14] `http://www.swi-prolog.org/pldoc/man?section=pio`, accessed on 10th July 2016

[15] A difference list is a sorted pair of lists d(L1, L2) where the second list L2 is completely included at the end of list L1. Simplified, this means that the last elements in the first list are reflected by the elements of the second list.

there is only one rule generated from input stream, then line 10 is used and returns a list with one parsed rule. Secondly, if there are two or more rules built up from tokens, then the DCG rule in line 12 is used and called recursively until the end of file is reached or no valid rule can be generated any more. In Listing 4.2 there are also different definitions of the `rule//1` predicate – see line 16, 22, 27 and 34. This one specifies the available and valid sequence of words for the general wording of the YaCI Rules.

Listing 4.3: Definite Clause Grammar rules which are used to process local file

```
1  % read rule file and pass them through the dcg                    PROLOG
2  read_file(File, Result) :-
3    phrase_from_file(parse_rules(Result), File).
4
5  % entry point of the YaCI grammar
6  parse_rules(Rules) -->
7    blanks,
8    list_of_rules(Rules).
9
10 list_of_rules([SingleRule]) -->
11   rule(SingleRule).
12 list_of_rules([SingleRule|Rules]) -->
13   rule(SingleRule),
14   list_of_rules(Rules).
15
16 rule(Rule) -->
17   when, blank, blanks, rule_premise(Premise),
18   then, blank, blanks, rule_conclusion(Conclusion), ".", ( "\n" ; eos ),
19   { /* create rule term and unify with "Rule" */ }.
```

The already introduced component–based structure of the natural language for paraphrasing testability rules is used for the grammar implementation, too. To get a quick overview, Listing 4.4 contains all necessary DCG rules to parse the simplest YaCI Rule containing a scope, a simple "is" condition term and the conclusion to return the severity level "error" as result of the analysis when rule is failed. The example testability rule looks like:

> *"Simple Example @ when method is private then drop error."*

Listing 4.3 gives a basic introduction how to free up the testability rule file in SWI–Prolog. This includes the creation of an input stream to read–in the content of the file as well as the entry point in form of a recursive clause definition (see line 10 and 12) that calls `rule/1` until the end of the token sequence is reached or no valid rule was found. Much more interesting is for now, how the implementation on parsing each single rule by the YaCI parser looks like. In other words, what set of DCG rules

is necessary to specify the wordings of a valid YaCI Rule and how it is structured in the current implementation.

The complete rule parser exists of a set of DCG rules which are process the input stream in form of tokens. Line 1 in Listing 4.4 shows the general `rule//1` clause as the starting point for parsing the conditional sentences. Its body lists the different terms necessary to specify a valid YaCI Rule. If all terms are evaluated to true, then the parsed character sequence is a valid rule and the generated internal representation during parsing process is unified to the provided argument variable and added to the list of rules. The body of `rule//1` specifies that a rule has to start with a description. The wording of these description is defined in the clause `rule_description//1`. Next, the token stream must contain the keyword "when" that introduces the rule premise, followed by one or more white–space characters. Furthermore, the grammar expect the premise itself that is signified by a reference to the DCG rule `rule_premise//1`. After the premise there has to be the keyword "then" followed by white-spaces as well and the reference to the `rule_conclusion//1`. Finally the grammar except a "." (dot) as the closing of a valid rule statement. After that a line break indicated the start of another rule or the end of the stream is reached (eos).

Listing 4.4: Basic DCG implementation to parse YaCI Rule

```prolog
rule(Rule) -->
  rule_description(Description),
  when, blank, blanks, rule_premise(Premise),
  then, blank, blanks, rule_conclusion(Conclusion), ".", ( "\n" ; eos ),
  { Rule = rule(when(Premise), Conclusion, Description) }.

rule_description(Description) -->
  string_without("@", ShortDescription), blanks,
  "@", blank, blanks,
  { /* create description term and unif with "Description" */ }.

rule_premise(Premise) -->
  rule_scope(Scope), { Premise = Scope }.

rule_scope(ScopeResult) -->
  single_rule_scope(Result), { ScopeResult = Result }.

single_scope_condition(Structure) -->
  condition(ConditionTerm), blank, blanks,
  condition_term(ConditionTerm, TermResult), !,
  build_structure(ConditionTerm, TermResult, Structure).

condition_term(is, Result) -->
  accessor(AccessorTerm), blank, blanks, { Result = AccessorTerm }.
```

The example rule above is the simplest available phrasing through the YaCI Rule grammar. For instance, the chaining of scopes or conditions are not considered yet. But exactly this element increases the strength of the grammar immense in order to observe several conditions in a single rule premise. This advantage is implemented in the grammar specification using three different `rule_premise//1` clauses – see listing below. The first listed clause describes a premise that only consists of a single scope. If the token sequence only hold one scope, then the clause `rule_scope//1` is used to parse the sentence and unifies the outcome to the unbound variable *Premise*. More interesting will be an example with the following YaCI rule wording:

> *"when method ⟨Condition⟩ and constructor ⟨Condition⟩*
> *then ⟨Conclusion⟩."*

This sentence can not be parsed using the first clause of Listing 4.5. For this purpose the clauses in line 4 and 8 are necessary. These allow to parse sentences with two or more scopes in the rule premise chained with the conjunctions "and" respectively *or*. The way of proceeding in SWI–Prolog is to try each of the three DCG rules. If the currently used rule failed during process of the string then go back to the last correct branch and tries an alternative path using one of the other available rule definitions. The chars "and" causes the break using first clause of `rule_premise//1`. because it expects the keyword "then" to initiate the rule conclusion. In fact of the extended clause definitions the SWI–Prolog interpreter is able to use one of the other available rules. The variation starts in line 4 contain the keyword "and" in its specification which allows to parse the sentence above without problems. Third clause is necessary to cover rule wordings using the or conjunction between two scopes. SWI–Prolog uses for the explained behaviour the technique called *backtracking*[16].

Listing 4.5: Implementation of chaining scopes in rule grammar

```
1  rule_premise(Premise) -->                                          PROLOG
2    rule_scope(Scope), { Premise = Scope }.
3
4  rule_premise(Premise) -->
5    rule_scope(FirstStatement), and, blank, blanks, rule_premise(SecondStatement),
6    { Premise = and(FirstStatement,SecondStatement) }.
7
8  rule_premise(Premise) -->
9    rule_scope(FirstStatement), or, blank, blanks, rule_premise(SecondStatement),
10   { Premise = or(FirstStatement,SecondStatement) }.
```

[16] Backtracking uses the trial and error principles in order to solve a problem while extending a reached partial solution to a comprehensive solution. If the partial solutions can not be resolved than the algorithm goes one or more steps backwards and tries alternative ways.

It is almost impossible to publish a stable application within its first release. Often the requirements has changed several times during the development process. Starting from scratch also means that the developed tool at the beginning contains a limited basis of functionality. This includes also the potency of the rule grammar in order to write complex rules and transform testability design patterns into valid rule wording. In order to reduce the additional expenditure on refactoring and extending the rule grammar, the related component definitions are extracted into separate Prolog files. Each file contains simple DCG rules that determine the component and the string value of it. This implementation allows to quickly add new components or keywords to the rule grammar. Listing 4.6 shows the declaration of available accessors through the *accessor.pl* file.

Listing 4.6: DCG component "accessor.pl" of the rule grammar

```prolog
accessor(abstract) --> "abstract".
accessor( final ) --> "final".
accessor(private) --> "private".
% ...
```

The chosen component–based file structure which hold the different grammar definitions reflecting each component to keep the DCG rule specification clean and allows the extension of the keywords in an easy way. The body of the rule only contains the list of characters that represent the parsed component string. The single argument in the head is used to return a named term that will be used to create the internal rule representation during parsing process. All other rule components are divided into separate files and are specified equal to the accessor example above. The complete list of accessor definitions named before and an additional declaration file of grammar components is shown in Appendix C.1.

4.3. Specification of Testability Rules

In Section 2.2 we already picked out general testability design guidelines from literature. These principles target on the object–oriented programming language Java and help to write test suites with less effort. With the YaCI Rule Language developers are free to phrase their own rules. It is not mandatory to stay at the context of testability or extend these basic rule specifications. Because of the selected approach on examine testability of source code, a basic set of testability rules is provided by default with the YaCI analyser tool. Later in the evaluation, these basic testability rules are used to interpret the correct operating principle of the implemented analysis toolchain, includes rule parsing, examination of rules against the AST and

the feedback to developers via IDE. Below, there are the rule specification for the various literature design pattern guidelines introduced.

#1: Prevent complex, private methods

Scope	Method
Symptom	1. Declared as private 2. To many control flows 3. To many lines in body
Solution	Break down complex private method into several small methods which are easy to understand and lose complexity
YaCI Rule	*"when method is private and has line_count greater 10 and contain control_flow more than 3 times then drop info."*

#2: Prevent final or static methods

Scope	Method
Symptom	1. Declared as final 2. Declared as static
Solution	Do not declare methods as final or static
YaCI Rule	*"when method is final or method is static then drop warning."*

#3: Do not use keyword new to often

Scope	Method
Symptom	1. Used keyword new to often
Solution	Prevent to create to much class instances in a single method instead split them into separate but logical related methods
YaCI Rule	*"when method contain keyword new more than 4 times then drop warning."*

#4: Prevent constructors with complex logic

Scope	Constructor
Symptom	1. Object instantiation with keyword new 2. To many control flows 3. To many lines in the body
Solution	Pass objects via parameters into constructor (dependency injection), implement init methods instead of control flows and tons lines of code

YaCI Rule	*"when constructor contain keyword new more than 0 times and contain control_flow more than 3 times and has line_count greater 15 then drop error."*

#5: Prevent the Singleton pattern

Scope	Constructor, method, and field
Symptom	1. Constructor is declared as private 2. Public method with return type of class 3. Private field with type of class
Solution	Implementation of Singleton pattern is in most cases unnecessary. If it is not avoidable the wrap Singleton implementation with testable wrapper class.
YaCI Rule	*"when constructor is private and method has return_type of class and is static and contain access of static field then drop warning."*

#6: Prefer composition pattern opposite inheritance

Scope	class
Symptom	1. Depth of inheritance hierarchy of a class is to depth
Solution	Instead of inheritance prefer the composite pattern. Therefore the additional dependencies on calling `super()` of the super class does not exist.
YaCI Rule	*"when class has inheritance of depth greater 3 then drop warning."*

#7: Wrap external libraries

Scope	Constructor, method
Symptom	1. call method on class, not on instance 2. called method not in package of project
Solution	Calls to class methods of external libraries can not be stubbed out during unit testing. A solution could be to implement a wrapper class for the library which can be easily faked.
YaCI Rule	*"when method contain call of class_method more than 0 times not in package then score warning."*

#8: Prevent service lookups or static method calls

Scope	Method
Symptom	1. call method on class, not on instance

Solution	Implement a wrapper class which wrap all calls to an external library behind it. The wrapper class instead can be easily stubbed out during unit tests.
YaCI Rule	*"when method contain call of class_method more than 0 times then drop info."*

#9: Prefer interface-based design

Scope	Constructor, and method
Symptom	1. Type of parameters not of type interface
Solution	Replace invalid parameter type with definition of an interface
YaCI Rule	*"when method is public and has parameter not of type interface or constructor has parameter not of type interface then drop info."*

#10: Do not declare class as final

Scope	Class
Symptom	1. Declaration of class is final
Solution	Change accessor to public because there is normally no reason to declare class as final
YaCI Rule	*"when class is final then drop info."*

Table 4.1.: Available testability design patterns and their YaCI Rule

As already mentioned, the user is able to extend and adapt these rule specifications to his own requirements. Phrasing new rules is therefore not limited to the context of testability. Remember: the conclusion of a rule can be also to match a pattern defined through the premise.

5. The YaCI Rule Representation

The YaCI toolchain includes different modules in order to fulfil each partial objective set for the current work. Focus of this section will be the outcome of the rule parser after processing the predefined testability rules. The necessary grammar for parsing various conditional rules is already introduced in Chapter 4. Establishing a more generalised intermediate step on specifying an internal rule representation is useful in order to improve the flexibility and the use of single YaCI modules in their own analysis work flow with external tools. Later on, Section 5.2 present an outlook on the obtained flexibility through the intermediate step using a abstract representation.

5.1. Internal Structure in Prolog

The implementation of the rule parsing and the analysis module are both written in SWI–Prolog. This fact makes the decision on how to design the internal rule representation as the outcome of the YaCI Rule parser. It will be a complex structure of nested Prolog terms including all the extracted rule information while parsing the sequence of words. Because all other YaCI toolchain modules are also written in SWI–Prolog it will be useful to design the exchange format in a Prolog notation. Using another proven exchange format like JSON[17] or XML[18] does not make much sense at this point. This will result in additional expense that is not necessary. On the basis of the chosen modularised architecture it is technically possible to specify an own representation of the parser outcome. For more details, look onto Section 5.2.

Each processed rule sentence consists of a single `rule/3` literal[19]. Possible arguments of the basic rule structure can be the literals `when/1`, `then/1` and an optionally argument `description/2`. The first two literals are available in each rule term. Only the description can be left out and is missing when the rule string does not include

[17] The JavaScript Object Notation is a compact data format with less overhead and is used for exchange data between applications. The notation is also easily readable by humans and helped to make JSON popular [JSO13].

[18] The Extensible Markup Language (XML) is another format to exchange data between applications. It is used to represent hierarchical and structured data in form of text files. Instead of JSON, the XML format contain more overhead caused by opening and closing tags [BSMP+04].

[19] A sequence of characters that have the same syntactical form as facts but have no closing dot at the end. Literals are atomic statements.

a description sequence. Listing 5.1 gains an overview to a valid rule representation after processing the related rule sentence using the grammar parser.

Listing 5.1: Output of the YaCI Rule Parser after parsing a rule sentence

```
?- string_codes('Prevent final methods @ when method is final                SWI-Prolog
      then drop warning.', Rule),
    phrase(parse_rules(Result), Rule).
Rule = [80, 114, 101, 118, 101, 110, 116, 32,  102|...],
Result = [rule(
    when(method(is(final))),
    then(drop(warning)),
    description(short('Prevent final methods'))
)] ;
false.
```

The term of the parsed rule premise is dependent on the scopes and the associated conditions listed in the premise. If there is no conjunction of scopes or conditions available, then the created term is according to the name of the currently parsed scope tokens. For instance, the premise "method is private" is represented as

```
method(is(private)).
```

In case of the combination of two rule scopes the `and/2` respectively `or/2` literal encloses the scope terms. The term itself contains two arguments: both describes the related scope and the additional condition term. This is the first scenario when the premise only exists of two scopes. In the second one, there are three ore more scopes listed in a single rule combined by conjunctions. As already mentioned, in this case the internal representation is nesting the different incidence of the scope. The first argument of the term is a scope and the second one the next conjunction term and so on. Listing 5.2 gives a short overview on the generated representation of combining two or more scopes respectively conditions. The decision to nest the conjunction instead of using a simple Prolog list is taken because nested terms are easier to handle within the YaCI Rule Analyser and break them down into their single terms, later on.

Listing 5.2: Nesting of scopes and conditions by conjunctions

```
?- string_codes('when method is private or constructor is private        SWI-Prolog
      and class is final then drop warning.', Rule),
    phrase(parse_rules(Result), Rule).
Rule = [119, 104, 101, 110, 32, 109, 101, 116,  104|...],
Result = [rule(
  when(or(method(is(private)), and(constructor(is(private)), class(is(final))))),
  then(drop(warning)), description())] ;
false.
```

For now, the general rule structure and the associated scope and condition representation is introduced. The chaining of scopes or respectively their related conditions is represented, too. Next, the specification of the condition term is necessary to complete the parsed rule representation. It is easier to understand the transformation by introducing a simple example which exists of a rule scope and the associated condition term:

1. *when method has line_count greater 3 then score 1.*
2. *when constructor contain keyword new more than 3 times then drop warning.*

The scope term is followed by "has" and the concrete classification of the condition term. In this case it is phrased to observe the line count of the scope body declaration and validate it against the given numeric value using the comparator "greater". This is only one valid constellation of the condition term. The grammar implementation has to take care of the different wording of rule conditions in order to provide a most natural language spelling for the YaCI Rules. In the second example above another rule is listed which validates the usage of the new keywords in scope of a constructor. The outcome of the parser is approximate the same instead of the different phrasing above. Let's have a look at the resulting internal representation shown in Listing 5.3.

Listing 5.3: Representation of condition term and the related information

```
?- string_codes('when method has line_count greater 3                    SWI-Prolog
      then score 1.', Rule),
    phrase(parse_rules(Result), Rule).
Rule = [119, 104, 101, 110, 32, 109, 101, 116,  104|...],
Result = [rule(when(method(has([line_count, greater, 3]))), then(score(1)), description())] ;
false.

?- string_codes('when constructor contain keyword new more than 3 times
      then drop warning.', Rule),
    phrase(parse_rules(Result), Rule).
Rule = [119, 104, 101, 110, 32, 99, 111, 110,  115|...],
Result = [rule(when(constructor(contain([keyword(new), greater, 3]))), then(drop(warning)),
     description())] ;
false.
```

In the described implementation the important keywords, extracted during parsing process, are transformed to the Prolog fact named after the keyword. The argument of the condition term is a list containing all associated information. They are simply defined as Prolog atoms[20]. In the second example above the first item in the list

[20] An atom in Prolog is a character string starts with lower–case letter or is enclosed by quotation marks.

is another literal in Prolog and specifies the condition term and the related keyword. All available transformations of rule conditions through the parsing step are summarised in Listing 5.4. Previously the internal representation of chained rule elements are determined to nesting terms instead of lists. The decision on representing condition terms as list elements is explained because it results in a more flexible way of collecting the information related to the conditional term.

Listing 5.4: Internal term structure derived from substring of a rule sentence

```
"is {accessor}" := is({accessor})

"has {term} of depth {comparator} {number}" := has([{term}, {comparator}, {number}])
"has {term} {comparator} {number}" := has([{term}, {comparator}, {number}])
"has {term} {comparator} {string}" := has([{term}, {string} ])
"has {term} {comparator} type {type}" := has([{term}, {comparator}, {type}])

"contain {term} {keyword}" := contain([keyword({keyword}), greater, 0])
"contain {term} {keyword} {comparator} {number}" :=
    contain([keyword({keyword}), {comparator}, {number}])
"contain {term} of {accessor} {type}" := contain({term}([{accessor}, {type}]))
"contain {term} of {callable} {comparator} {integer} times" :=
    contain({term}([{callable},{comparator},{interger}]))
"contain {term} {comparator} {number}" := contain([{term}, {comparator}, {number}])
```

The illustrated term representation of the parsed rule string is created during the parsing process according to the YaCI Rule grammar. Changes on this internal structure are possible by adapting the corresponding term creation in the implementation of the DCG rules. How to use the parser result in order to run analysis during the YaCI Rule analyser is part of Chapter 6, where the architecture and functioning of the analysis is described in detail.

5.2. Interchangeable Representation

The additional step on defining an exchangeable representation of parsed rules is not necessary to reach the imposed goals for the current work. For simplification, the parsing result have to be tightly meshed to the analysis implementation of YaCI. In this case, the intermediate step on specify an additional representation is not necessary. The tightly coupled grammar parser and the analysis step would certainly work well together, but for this, the given approach will not have to be such a flexible toolchain in order to reuse on other research fields or within other applications. On the basis of the current modularity of YaCI, the usage of each single component is conceivable in a broader community.

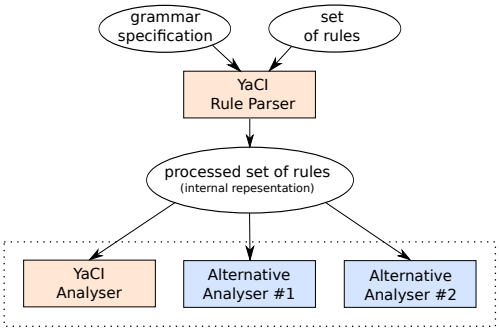

Figure 5.1.: Workflow of the YaCI Rule Praser and its in– and outcome

In a closer look at the grammar parser and the analyser module the advantages of the introduction using a generalised rule representation get more clear. As shown in Figure 5.1, there is now the possibility to replace the analyse task which is done in this case by the YaCI tool with another implementation of an analyser. It is only necessary that the used analysis tool is able to handle the specified internal rule representation of the parsing process generated from the rule parser. The other way around, it is also possible to replace the used rule grammar from the developed YaCI toolchain. The implemented version of the YaCI core module – the analysing part – uses the internal representation as input, too. For instance, in order to use another natural language parser, the outcome of the parsing step only has to be changed to the internal structure of the rules, defined in Section 5.1.

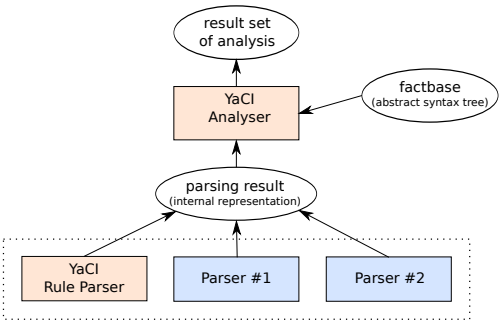

Figure 5.2.: Workflow of the YaCI Analyser and its in– and outcome

The modular design architecture allows a more flexible usage of each single component. Especially for the grammar parser and the analysis step this is a great advantage. With the chosen design approach it is possible for developers of other research fields to use different parts of the YaCI toolchain. This can be an own grammar for parsing other rule structures instead of conditional sentences or an-

other analyse module for examining the AST derived from source code. Figure 5.2 gives an overview to this obtained flexibility. A possible scenario can be for instance the extension of the toolchain in order to support other programming languages to bet not only limited to Java.

6. YaCI - Yet another Code Inspector

Different modules are introduced in the previous chapters. This includes a grammar specification for phrasing conditional sentences as well as the parser which implements the defined grammar and processes the rules. We often talk about modularisation and different components who shall work together to finally realise a complete analysis work flow in one toolchain. At this point of the thesis the big picture of the realised tool shall be get more clear. The complete work flow of the YaCI toolchain and its components is shown in Figure 6.1.

Furthermore in this chapter the two core components of YaCI are introduced, too. On the one hand, the analyser module that do the main analysis process in order to find rule violations in the abstract syntax tree of Java source code, on the other hand the result generator module. Its responsibility is the transformation of a rule match to a representation that can be used to display the results in the Eclipse JTransformer Control Center. To reduce the effort on installation and integration of the YaCI tool a packaged version is necessary. For this purpose we plan to publish a library of the complete toolchain including the grammar parser, the core analysis tool, the result generator, and the interface to the JTransformer Eclipse plug–in. Finally, the use of YaCI in a developers daily business is presented and a short future prospect to integrate the analysis tool into an already existing CI process.

6.1. Architecture

As already mentioned, the YaCI toolchain is developed as a software which exists of different modules. All together, a full working analysis tool is implemented, starting from phrasing conditional sentences (better call it rules) in a convenient and well–known natural language, over to parsing those sentences into an internal rule representation which is used for analysis purpose and examination of the abstract syntax tree of Java projects. At least the JTransformer Eclipse plug–in is adapted to display the analysis results in a GUI. The different components are coupled together using the implemented interfaces of the modules and the related exchange format to pass data from one module to the next one. Figure 6.1 not only shows the available components. It also gives an overview on which modules are interconnected and what data they share together. The implementation of the analysis toolchain

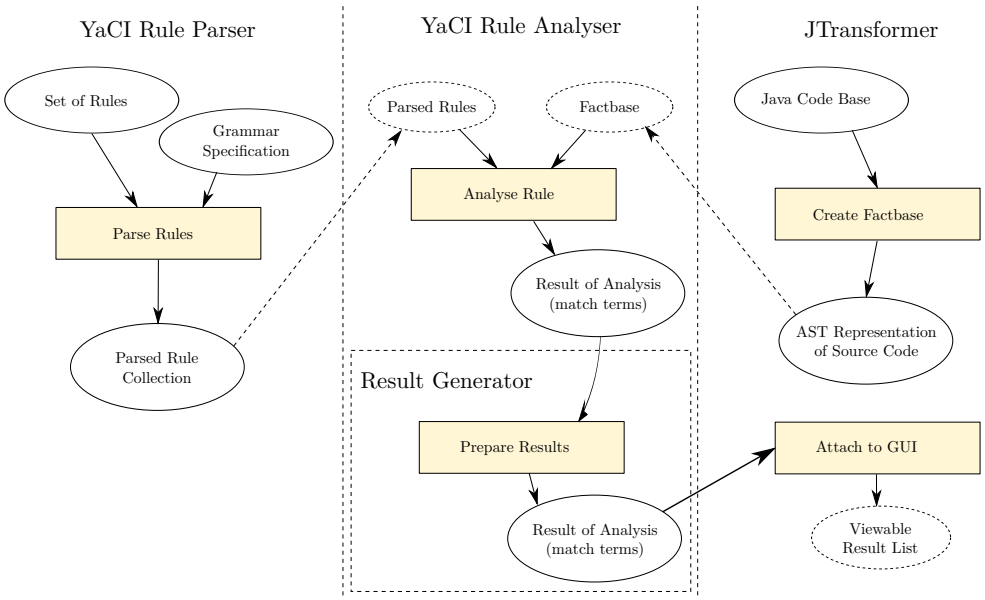

Figure 6.1.: YaCI components and their interactions

is based on version 7.2.3 of SWI–Prolog. It is recommended to use this version or newer in order to provide the necessary built-in predicates regarding to DCGs and quasi quotations. These are used by the rule grammar specification and the related parser implementation.

The rule parser is focus of Chapter 4 and the creation of the abstract syntax tree is explained in Chapter 3. In the figure above, on the left side there is the internal rule representation shown as the exchange format between the rule parser and the analysis tool. Its characteristic and benefit is explained in Chapter 5. Creating the factbase of the related Java project is done via the available JTransformer integration for Eclipse. A closer look at the created AST and the structure of the derived Prolog factbase is shown in Section 3.1.

There are only two modules missing and not yet observed. One of those is the core module of the complete YaCI toolchain: the analyser. Inside this component all the examination of the abstract syntax tree using the generated rules from grammar parser take place. The result will be a generalised match result term containing detected unique fact identifiers affected by the rule. Because of the planned approach to visualise these results in the JTransformer Eclipse plug–in another simple component is necessary: the result generator. It simply takes the matching results and transform them into a JTransformer compatible representation. In this way it is possible to attach the result information – like what class, method or constructor declaration are failed during analysis process – to the JTransformer Control Center.

6.2. Rule Analyser

The analytical part of the YaCI toolchain shall be also implemented in the logical programming language SWI–Prolog. The analyser expects two inputs in order to fulfil the imposed demands. Firstly, the factbase is necessary which hold the abstract syntax tree in form of Prolog–based logical facts generated from a Java project. The second input will be the collection of parsed rule sentences represented in the internal structure that was introduced in Section 5.1. Generating the abstract syntax tree using JTransformer was already described more detailed in Section 3.1.

All modules of the Yet another Code Inspector are implemented in SWI–Prolog which is a huge advantage on building up a complete toolchain. There is no more additional work expected to transform output from one module in order to use it with the subsequent module realised in another programming language. In fact of this advantage the exchange format between the different YaCI modules are always nested Prolog terms. The core analysis component is able to consult the generated factbase resulting from JTransformer. This step makes the factbase available to the SWI–Prolog process that represent the core analysis implementation. To start the examination of the AST against the predefined set of rules it is only necessary to hand over this collection to the analysis module.

Entry point of the analysis is the clause `analyse_rule/2`, defined in the module as the public interface to pass data into it. The already mentioned AST factbase is consulted when the SWI–Prolog process is started and from now available to query it. Remind: the database consists of facts specified as Program Element Facts (see JTransformer manual[21] as a reference of available facts). The starting point of the analysis expects only the rule premise. Conclusion and the third term – the description labels – are not important for analysing the provided rule. Generally, the parsed rule sentence illustrated in the internal representation looks like this:

```
rule(when(method(is(private)), then(⟨Conclusion⟩),
description(⟨Desc⟩))).
```

Using the rule premise as the input of the analysis term and call the `analyse_rule/2` clause will start the process on validating the rule against the AST. As seen in the short example above, the premise is well structured and follows the assembling of a scope, followed by the condition and the related term containing specific information on how to satisfy the condition term. To validate the different parts of the premise term the analyser has to dismantle it using different Prolog clauses implemented by the YaCI Rule Analyser. Because of the generated abstract syntax tree includes not

[21] `https://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/4.1/java_pef_overview`, accessed on 18. July 2016

only own implemented classes of the Java project it must filter unnecessary facts out in order to analyse only the relevant once.

In Listing 6.1 the entry point of the analysis module is shown and its related implementation. The `find_compilation_unit_id/2` clause in line 2 is responsible for finding all class identifiers which are under `/src` in the Java project. This excludes all files in `/test` and prevent the examination of its associated facts during the analysis process. An important naming convention of the PEFs apparent in line 1 and 6. All types related to the Java AST ending with a capital `T`, instead of the language–independent facts that represents the organisation of source code in the AST. These types are ending with a capital `S`.

Listing 6.1: Filter only necessary compilation units from factbase

```
1  analyse_rule(when(Premise), Result) :-                                    PROLOG
2    find_compilation_unit_id(_, CompilationClassID),
3    apply_premise(Premise, CompilationClassID, PremiseResult),
4    Result = PremiseResult.
5
6  find_compilation_unit_id(_, ClassID) :-
7    projectS(ProjectID, _Name, _LocalPath, _, _),
8    sourceFolderS(_FolderID, ProjectID, _FolderName), !,
9     fileS (FileID,  ProjectID, FileName),
10   \+ sub_string(FileName, _, _, _, '/test/'),
11   compilationUnitT(_, _PackageID, FileID, _ImportsArray, [ClassID]).
```

In detail, the clause iterates over all available `projectS/5` facts in the factbase and unifies the project id to the variable `ProjectID`. Next, an available `sourceFolderS/4` is selected from the factbase whose second parameter can be unified with the already bound variable from line above. Line 4 bind the `Filename` to an available string represented by a fact of type `fileS/3`. Because all unit test files shall be ignored by the analysis tool it left out `fileS/3` facts whose filename contain the substring `/test`. At least, the variable `ClassID` is unified in the last line using the `compilationUnitT/5` fact. Result of the clause will be an unique identifier of an available `classT/5` term. Via backtracking the analyser will investigate one class identifier after another and collect possible rule matches in a list.

Understanding the example above on filtering only necessary compilation units from the factbase helps to understand all the remaining implementation and working method of the analysis module. To provide a better understanding for using the existing factbase in order to observe the YaCI Rule, the example premise above shall be picked up again:

```
when(method(is(private)))
```

Similar to the rule parser which iterates through a sequence of words, the analyser goes through the nested premise terms in order to examine the given rule. The `apply_single_scope/3` clause is called by the already introduces `analyse_rule/2` and contain two input parameters and the third parameter as the outcome of the clause. First the clause on applying the rule scope search for a `methodT/8` term that second argument can be unified with the `Context` variable. This variable represents the class identifier received from the `find_compilation_unit_id/2` clause. To ensure that the method id is a direct child element of the class the helper `is_direct_child/2` is called. When a fitting method fact is found then the condition is examined using the clause in line 4. The remaining lines of `apply_single_scope/3` create a Prolog structure that represents the result of the examination containing the scope of the rule, the id, and name of the matched method in the factbase, and all hint elements which are related to the rule validation. In this simple case it is only the matched accessor of the method.

Listing 6.2: Validate example rule premise against AST

```prolog
apply_single_scope(method(Condition), Context, Result) :-
  methodT(MethodID, Context, MethodName, _, _, _, _, _),
  is_direct_child(MethodID, Context),
  apply_condition(Condition, Context, MethodID, ConditionResult),
  Result = method(
    id(MethodID),
    name(MethodName),
    hints(ConditionResult)
  );
  fail.

apply_single_condition(is(Modifier), Element, ModifierResult) :-
  modifierT(ModifierID, Element, Modifier),
  ModifierResult = modifier(ModifierID).
```

The Listing 6.2 also includes the `apply_single_condition/3` clause that validates the related rule condition, if all requirements are met. It expects three arguments. The first is the rule condition which is necessary. The condition is taken apart by the term `is(`⟨`Modifier`⟩ where the variable `Modifier` is unified with the content of the condition term – in the example above the atomic string “private”. The second parameter is unified with the unique id of the parent element (in this case the unique scope id of the method) received from the factbase. Checking, if the modifier of the method fact is according to the rule definition, in line 14 the `modifierT/3` tries to unify the variable `ModifierID` by using the already bound variables `Element` (the method identifier) and `Modifier`. If the modifier id can be matched in the factbase, then a result term is generated and unified to `ModifierResult`. The result

is returned to the calling clause which uses the intermediate result on generating the final term of the scope examination.

Described above is the general work flow of examining the abstract syntax tree against the YaCI Rules. It only depends on the different components the rule premise term can be exists of. Summarised the analysis process can be explained in a short way during the following steps:

1. Start analysis by calling `analyse_rule/2` clause of YaCI analyser.
2. Find relevant compilation units in order to omit unnecessary unit tests classes or basic Java class files.
3. Select element fact from factbase according to the current scope.
4. Apply the condition which is defined in the rule related to the scope and return the validation result.
5. Go back and build up the result term related to the current scope.
6. If rule contain more than one scope (conjuncted by “and” respectively “or”) start again at (3) until all scopes are analysed.
7. Return the match result term.

Listing 6.3: Result term of the YaCI Rule Analyser

```
1 ?- analyse_rule(when(method(is(private))), Result).                    SWI-Prolog
2 Result = match(class(28595), scopes([method(id(28600), name(initialize),
3 hints([modifier(28613)]))])) ;
4 Result = match(class(28595), scopes([method(id(28601), name(appendToLog),
5 hints([modifier(28676)]))])) ;
6 false.
```

The general output of the `analyse_rule/2` clause is specified in an abstract level. This gives the flexibility to use the result of the analysis process in collaboration with another graphical user interface to display the results. For the current approach the Eclipse IDE is used as the UI in combination with the JTransfomer plug–in. In Listing 6.3 a match term is represented that is generated during the analysis process of the simple example rule *“when method is private then drop info.”*. Via backtracking all possible matches for a single YaCI Rule will be generated during the analysis process. Each of the results contain the compilation unit item with its unique identifier and a list of scopes that will be matched by the rule. The scope itself contains its own identifier as well as general information – like the name of the method. Each scope term hold as the last information a term called *“hints”*. This term contains all elements that are identifying the pattern described in the

given YaCI Rule. In case of the previously introduced simple rule example, only the modifier of the method is a helpful hint on matching these rule in the factbase. Within this additional information the developer is able to see which element in the source code is affected on evoking the matching rule.

In the next section we want to describe how the analysis results can be used to display the extracted information from the abstract syntax tree in a graphical user interface. Therefore an additional result generator module is implemented that transform the match terms into a JTransformer compatible result term that can be used to attach them to the Control Center.

6.3. Result Generator and JTransformer Integration

In Section 3.3 the JTransformer tool and its analysis API[22] is already introduced. Using the predefined analysis interface of JTransformer allows developers to add own implementations of analysis tasks to the Control Center in order to run and view this tasks. In order to add the YaCI Rules – representing the analysis definitions of the YaCI toolchain – a JTransformer integration is necessary that uses these API. The correct usage of the interface gives the opportunity to use additional functionality like to jump in to related position in source code by double click on the analysis results in the Control Center. A specific term structure is necessary in order to have the availability to add the analysis results of the YaCI Rule Analyser to JTransformers graphical user interface. The transformation of the result terms (received from the YaCI Rule Analyser) is done by the Result Generator module which is described later in this section. First a short overview to the JTransformer YaCI Integration is given in the following.

Attaching the analysis implementations defined by the YaCI Rules is done by using the corresponding interfaces which are provided by JTransformer. The general usage of the analysis API is already shown in Section 3.3 where the Singleton Pattern analysis task is implemented and attached to the Control Center in order to analyse given factbase against it. First there have to be an analysis definition asserted to the current SWI–Prolog process which can be used by the UI to display a list of available analysis tasks. Exactly the same must be done for the analysis result interface which is also available through the JTransformer API. This steps are automated by the YaCI JTransformer Integration when the library is consulted during the bootstrapping process. As a short summarise, the following actions are executed when

[22] An **a**pplication **p**rogramming **i**nterface allows using functionality of a programm through the defined interfaces.

the YaCI integration module is consulted in order to use the analysis toolchain via Eclipse IDE:

1. Read in the YaCI Rule file in order to parse the conditional sentences using the specified DCG.
2. Attach parsed YaCI Rules to the JTransformer Control Center using the analysis definition API.
3. Initialise the analysis result API in order to show the result terms in the Control Center after running selected analysis tasks.

As already mentioned, to attach analysis tasks to the graphical interface YaCI has to define for each parsed rule the `analysis_api:analysis_definition/5` and assert the term to the current factbase of the running SWI–Prolog instance. The YaCI JTransformer Integration module runs for each rule the clause in line 1 with one argument to pass in the generated rule from the Rule Parser. The necessary information to assert the analysis definition are unified to the associated variables shown in line 2 to 4. Asserting the analysis definition is done by using the `assert/1` clause of SWI–Prolog. All helpful information to complete the JTransformer Control Center UI are passed into `analysis_api:analysis_definition/5`.

Listing 6.4: Add YaCI analysis definitions to the Control Center

```
1  add_single_definition(rule(_,then(Conclusion),description(Short, Long))) :-          PROLOG
2    Short = short(ShortLabel),
3    Long = long(LongLabel),
4    Conclusion = drop(LogLevel),
5    assert(
6      analysis_api:analysis_definition(
7      ShortLabel,        % Rule name
8      onSave,            % Trigger
9      LogLevel,          % Severity level
10     'Testability',     % Analysis group
11     LongLabel          % Long description
12   )
13 ).
```

After adding the YaCI Rules to the UI as explained before, there is still the implementation of the analysis result API missing. Listing 6.5 shows the specification of the `create_single_analysis_result/2` Prolog clause which asserts the `analysis_api:analysis_result/3` rule to the currently running SWI–Prolog process. For each YaCI Rule, also the clause will be called in order to add the associated analysis result. When starting the process on running all defined analysis tasks via Control Center and there are available matches for the related rule, then the

`analysis_result/3` clause is invoked. The terms in the body of the rule will be called via backtracking until all possible results are attached to the result view in the Control Center. The listing also shows in line 10 and 11 invoking the YaCI Rule Analyser and the result generator module of the toolchain. In line 13 the different matched AST elements are marked in the related resource (the class in a Java project where the rule is matching). The implementation of `mark_match/2` will be explained next.

Listing 6.5: Analysis result API used by YaCI

```
1  create_single_analysis_result(Premise, Description) :-                    PROLOG
2    Description = description(short(RuleID), _),
3    assert(
4      analysis_api:analysis_result(
5        RuleID,
6        ResultGroup,
7        Result
8      ) :-
9      (
10       analyse_rule(Premise, AnalysisResult),
11       generate_match_term(AnalysisResult, MatchTerm),
12       MatchTerm = result(ResultGroup, RuleElements),
13       mark_match(RuleElements, Result)
14     )
15   ).
```

As seen in the listing before, there are to helper modules necessary in order to add the analysis results to the UI. Firstly, the already mentioned result generator which applies the transformation from the match term generated by the YaCI Rule Analyser to the result term. Secondly, another small helper that creates the Eclipse IDE problem markers in order to highlight affected elements in the editor view of Eclipse. The result generator flatten the match term in order to generalise the affected AST elements to provide a more smoother way to add these elements using the JTransformer analysis API. Listing 6.6 shows the transformation result of the `result/2` term. The `result_group/1` term and the included unique identifier represents the Java class which contains the rule match.

Listing 6.6: Transformed YaCI Rule Analyser match into result term

```
1  ?- generate_match_term(match(class(28595),scopes([method(id(28600),          SWI-PROLOG
2         name(initialize), hints([modifier(28613)]))])), Result).
3  Result = result(result_group(28595), elements([method(28600), method_modifier(28613)])).
```

At least, there is the `mark_match/2` clause that will be invoked until adding analysis results to the Control Center result view. The first argument is an already bound

variable to pass in the matching elements which were examined during the analysis process. The variable `RuleElements` contain a list of available element types to attach a marker to the Eclipse IDE editor view. Listing 6.6 already introduce a small range of available element types, e.g. `method/1` or `method_modifier/1`. The helper module implements a set of Prolog rules to cover all possible element types returned from the analysis task. JTransformer provide the `make_result_term/3` in order to create a valid Prolog term that can be displayed in the Control Center window. It expects as the first argument an unique fact id from the existing factbase, and second a short description which is displayed in the result view of the Eclipse IDE. The third argument represents the bound variable **Result** during the unification of the `make_result_term/3` term.

Listing 6.7: Implementation of the `mark_matcher/2` Prolog clause

```
1  find_element(List, Pattern, Element) :-                    PROLOG
2    member(Pattern, List),
3    Element = Pattern.
4
5  mark_match(elements(ElementList), Result) :-
6    find_element(ElementList, method(_), Element),
7    make_result_term(Element, 'Method caused rule violation', Result).
8
9  mark_match(elements(ElementList), Result) :-
10   find_element(ElementList, method_modifier(_), Element),
11   make_result_term(Element, 'Modifier of matched method', Result).
```

For now, there are all necessary prerequisites made in order to use the YaCI toolchain in combination with the Java–based Eclipse IDE and the JTransformer plug–in. How to integrate YaCI into the daily business of a developer will be discussed in Section 6.5.1 later on.

6.4. SWI–Prolog Package

In order to support an easy as possible usage of the YaCI toolchain there shall be a packaged version of the tool which can be used as a SWI–Prolog library. This package includes the grammar specification, the related rule parser, the analysing module, the result generator and the JTransformer integration for Eclipse. Packaging all these components will provide a more easier installation and usage of the tool on other development workstations. The YaCI SWI–Prolog package is created according to the tutorial[23] available on the official website.

[23] `http://www.swi-prolog.org/howto/Pack.html`, accessed on 18th July 2016

Creating the Prolog library of YaCI is realised by using a simple shell script. In the first step it copies all necessary resources to a temporary distribution folder. The resources are mainly the implemented Prolog files and the required `pack.pl` that contain general information of the package like the name of the author, the title of the package and the version. The current version number of the package is readout from the `pack.pl` file and used for the archive file name. The created `yaci-⟨version⟩.tgz` can now be used for distribution purpose. Installing the package on another workstation is done by using the following command in a running SWI–Prolog instance:

```
pack_install('path/to/pack/yaci-⟨version⟩.tgz').
```

The `pack_install/1` clause extract the package and copy it to the local library folder of SWI–Prolog. It can be found in `/home/⟨user⟩/lib/swipl/pack`. When the package is locally installed then it can be referenced in a Prolog file using `use_module/1` in the following way:

```
use_module(library(yaci)).
```

6.5. YaCI in Development Work Flow

In order to make the *Yet another Code Inspector* available for developers the first step is to build a SWI–Prolog package that can easily be used in own Prolog programs. The procedure on packaging YaCI is described in the section before. But how can developers integrate the tool in their local development process to validate their source code by running the analysis tool. And what about internal build process – like continuous integration – that create the Java project to release it for customer use. In the following, the usage of YaCI on a local development machine using Eclipse as the Integrated Development Environment shall be described. Furthermore, a short outlook on how to integrate it into the CI process is given.

6.5.1. Eclipse Integration

To use YaCI on a local machine there are some installation requirements to met. First of all the IDE Eclipse is necessary as well as the JTransformer plug–in for Eclipse and the *Prolog Development Tools (PDT)*. Both are available via the official JTransformer update site[24] and can be installed using the *New Software* dialogue in Eclipse. Just an executable SWI–Prolog installation is required. The installer

[24] `http://sewiki.iai.uni-bonn.de/public-downloads/update-site-jt`, accessed on 19th July 2016

can be downloaded on the official website or using the *swivm* command line tool. Using swivm is the preferable way because it allows the handling of different local installations of SWI–Prolog. A helpful documentation for swivm is available in the official GitHub repository[25].

If all prerequisites are finalised, then YaCI can be used according to the following instruction in order to analyse Java projects. First of all a new project in the current workspace of Eclipse have to be created. This can be done using the *New Project wizard* that is available in *File* → *New* → *Project...*. The name of the project is arbitrary and freely selectable. For instance, we named it "YaCI". From now the project is available via the *Project Explorer* in Eclipse. Next, a new file `load.pl` is added to the project that is used as the entry point of the YaCI project. This file simply load the packages library including the actual implementation of the rule parser, the analysis core module and the result generator. Figure 6.2 shows an overview to Eclipse with the YaCI project and the opened `load.pl` file. Before the module is consulted for the first time, the variables `BasePath` and `RuleFile` have to be unified to the current circumstances on the local environment first. The rule file – holding the phrased conditional sentences – can be placed in the Eclipse project as seen in the figure bellow or changed to a valid path on the local disk.

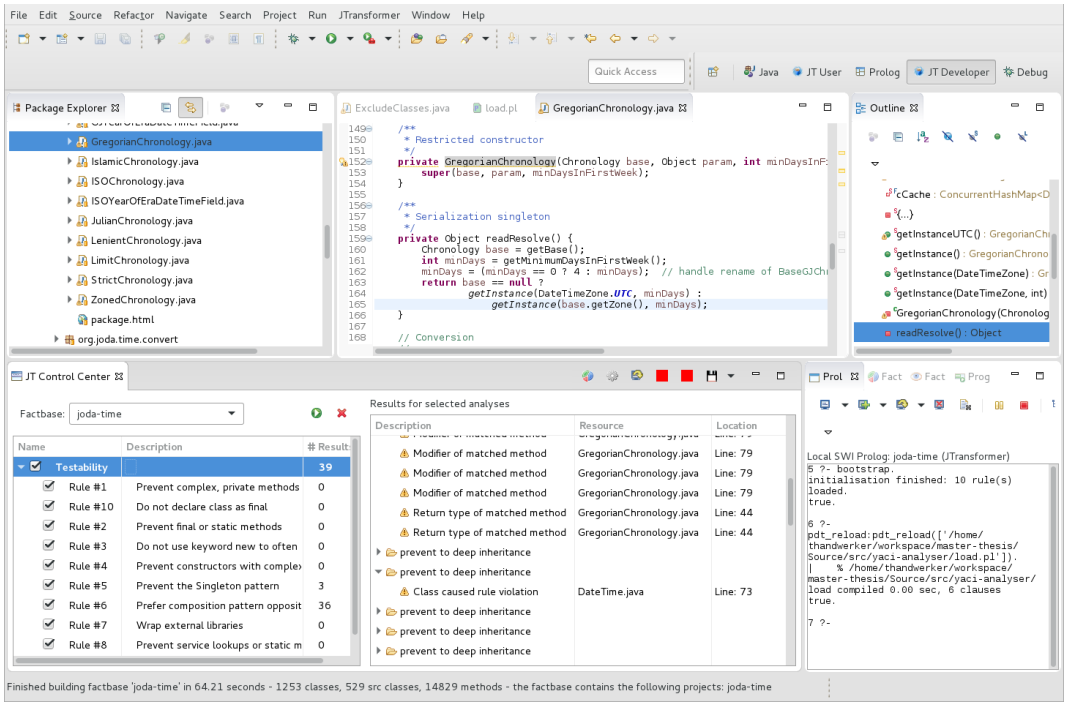

Figure 6.2.: Eclipse and the consulted YaCI Rule Analyser library

[25] `https://github.com/fnogatz/swivm`, accessed on 19th July 2016

If not already done, open the Developer View of the JTransformer Eclipse plug–in to display the Control Center. In this view the desired factbase of the Java project under analysis has to be selected. After selection, consult the load.pl file by right clicking on the YaCI project and select *Prolog Development Tools → (Re)consult*. A new Prolog process is started and interacting with it can be done using the *Prolog Console* view. Start initialisation of YaCI is done by executing the *bootstrap/0* clause via console view. After successfully finished the init process there comes a short log message that shows a success message. (Re)Consulting of the `load.pl` at this point is necessary to refresh the Control Center in order to display the available analysis definitions for JTransformer.

Running the analysis process of the selected rules is started by clicking the *Run all enabled analyses* buttons. After processing the analyses the results are attached to the Control Center and can be considered in the result table view. Jumping to marked elements in source code is also available. Simply make a double–click on the desired entry in the list and Eclipse will jump to the point in source code and open the correct file in editor, if necessary.

6.5.2. Continuous Integration Lifecycle

Another important point of utilising the analyses functionallity is the integration into an already available continuous integration lifecycle of a software deployment process. The technical implementation of the CI integration will not be part of the current work. Leastwise, through its important attitude in today's software development it shall be shortly suggested. Continuous integration is the process on building deliverable software in a progressively and repeatable way. This mean that compiling Java classes, running unit tests, packaging the necessary resources, etc. is a specified and automated process. For this purpose there are different build systems available that helps to characterise one or more build process. For example, Jenkins[26] and Travis CI[27] are well–known in this section and are widely used. The prominence of Travis CI is likely to be, because of the good integration into GitHub which is used for a huge amount of open source projects in the community.

As described in the previous section, the integration of YaCI into the local development process is possible at all. The next logical step is to integrate the analysis tool into an existing build system. This integration will help to keep the source code of a Java project testable and ensure that only "good" and clean code will be shipped to the customer. Local integration of YaCI is a good choice to help developers to analyse their written line of codes against the predefined coding guidelines (illustrated

[26] `https://jenkins.io`, accessed on 19th July 2016
[27] `https://travis-ci.org`, accessed on 19th July 2016

through the rules are used for YaCI) just in time on their local machine. But this will not prevent developers from committing code into the version control that not follow the rules. This behaviour can send to deliver an undesirable software to the customer.

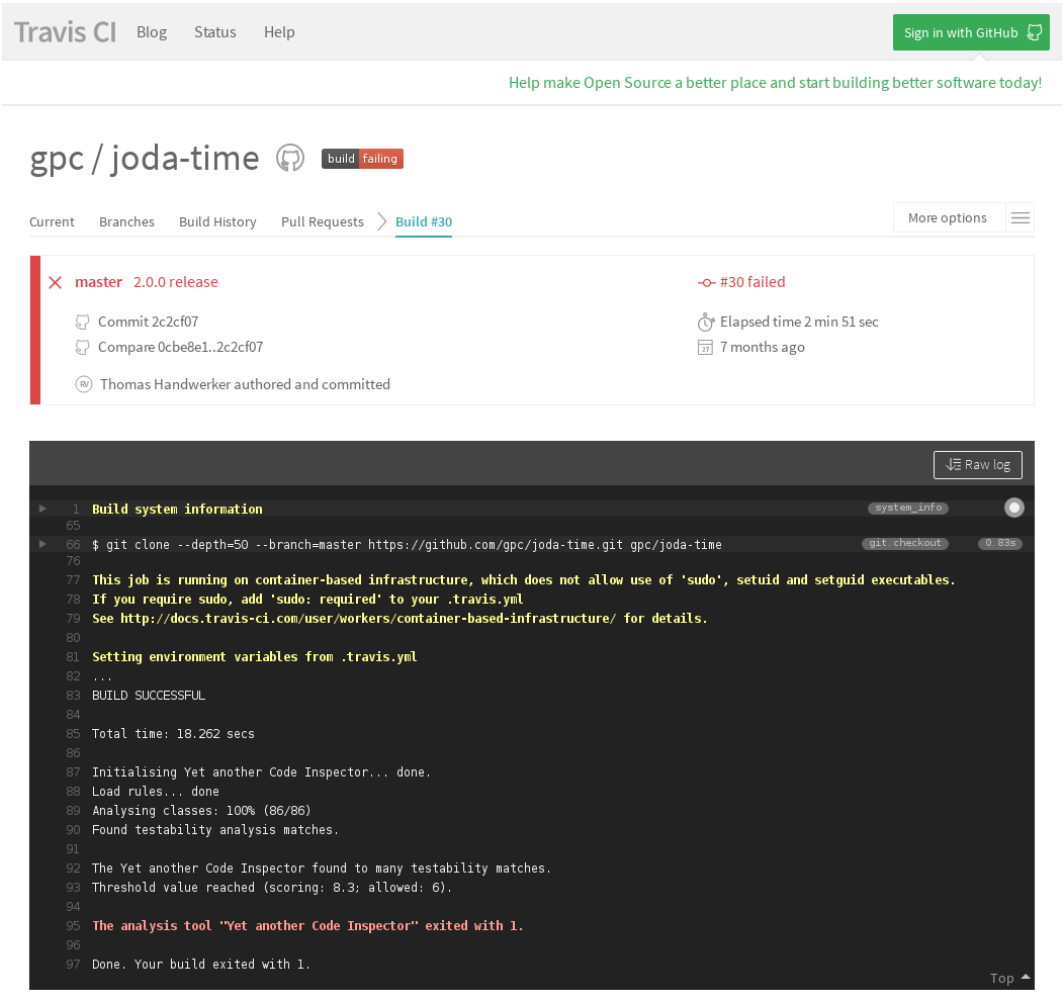

Figure 6.3.: Exemplary integration of YaCI into Travis CI

The integration of YaCI into the build process can prevent this misbehaviour and is able to stop building the delivery premature when the YaCI analyser returns errors during analysis. This ensures that only stable and well analysed code will be delivered to the customer. To integrate the realised code analyser into the build process the YaCI rule grammar already support conclusions like

when method is private then score ⟨*Integer*⟩.

By using a rule condition of this type, a simple integer value can be provided which determine the severity of violating this rule. A appropriate way on using these integer

values in the context of a build process can be as follows. If a predefined value in the build system is exceeded by the score value of a rule, then stop building the software immediately and return corresponding log message. Another – and maybe a better – solution can be the definition of a threshold value configured server–side on the build server. This approach prevent the situation, if only one rule with a high score value is matched during analysis process and therefore stops the remaining build. Using a threshold will only stop building when the averaged value of the matched analysis rules will exceed the predefined value.

The extension of existing build systems shall be able through their plug–in architecture. This allows to extend the functionality in order to use external tools during the build process, for example. The already named open source build tools Jenkins and Travis CI also include such a plug–in system. Using the YaCI analysis tool during the build process can be realised by implementing an own extension for considerable build tools. The requirements that have to be met on the build server is nearly exact the same as on local machines. A global installation of SWI–Prolog is necessary as well as the JTransformer tool which creates the abstract syntax tree of the Java projects. The AST factbase is essential for the analysis part of the YaCI tool. How to use JTransformer in order to create the factbase without running Eclipse on the build server have to be examined in the run–up of the plug–in implementation.

7. Evaluation and Results

In this section the functionality of the implemented analysis tool shall be evaluated. The goals of this approach – defined in Section 1.4 (Implementation Goals) – are used for evaluation and prove if these are reached in an acceptable coverage. As a short reminder, the following goals are set in scope of the current work:

- Search for available testability design pattern in literature and collect them in order to rewrite them in conditional sentences for the analysis tool.
- Specify a rule grammar and the related parser using DCGs in SWI–Prolog.
- Realise internal rule representation as the outcome of the parser in order to provide a flexible usage of the different components of the analysis tool.
- Implementation of an analysis tool which examines the parsed conditional sentence rule against the abstract syntax tree of a Java project.
- Result of the analysis tasks shall be used to generate feedback for developers and help to find inconsistency in source code matched by the testability rules. A graphical user interface shall be used to show the generated feedback.
- Implement this toolchain in the logical programming language Prolog (respectively its implementation SWI–Prolog)
- Evaluation of the YaCI toolchain using the real world Java–based software project "Joda Time".

In order to check, whether the objectives of the current work have been achieved, the Yet another Code Inspector shall be run in an environment which is as close to the real world as possible. For this purpose the developed toolchain is used to examine the Java date time utility library "Joda Time". Its source code is published as open source under the Apache 2.0 licence and is available on GitHub[28]. The decision to use an open source project was made because a transparent, reliable, and repeatable evaluation shall be obtained. Analysing the abstract syntax tree of a Java software helps to improve its testability with regards to writing unit tests with less effort. Therefore, the analysis focuses on finding design patterns in source code

[28] `https://github.com/JodaOrg/joda-time`, accessed on 21th July 2016

which are bad in the point of view form testability in object–oriented programming languages.

The way of proceeding is structured as described in the following: First, a list of rules is necessary which is stored in a local `*.rules` file and must be available for the YaCI toolchain. The set of rules contains all testability rules which are extracted from literature and coding guidelines (see Section 2.2). According to the specified YaCI Rule grammar the patterns are reworked to the conditional sentence structure *"when* ⟨*Premise*⟩ *then* ⟨*Conclusion*⟩*."* and are listed in the local rule file. In the next step, this file acts as the input of the YaCI Rule Parser. The parser processes the set of rules and transforms them into the internal representation which is also the output of the module. The list of the parsed testability rules is passed on to the YaCI Rule Analyser that is responsible for the core analysis. As second input parameter the analyser expects the AST of the Java software which shall be examined. The syntax tree is represented by a set of Prolog facts in its logical representation which is generated from the JTransformer Eclipse plug–in. During the analysis process the YaCI Rule Analyser applies each testability rule to the AST and builds up the related match terms. These terms contain all necessary information to display the result of the analysis process in a graphical user interface. Helpful information extracted by the analysis can be for example the affected resource (in Java it means the class where the result is matched) or different hints that indicate a match of the rule in the AST. The created result terms of the YaCI Rule Analyser are then passed on to the YaCI Result Generator. It transforms the outcome of the analyser into result terms which are used by the YaCI JTransformer Integration in order to display the analysis results in the associated Control Center.

7.1. YaCI Modules

In this section, the general functionality of YaCI and its various modules shall be observed in detail. This starts with using the Rule Parser to process conditional sentences followed by the Rule Analyser to examine the AST of a Java project. In order to provide a visualisation of the results in a graphical user interface the components Result Generator and JTransformer Eclipse Integration is necessary and shall be evaluated next. The provided set of rules are to extensive for carrying out a concrete evaluation. That is the reason why the decision to limit the evaluation on only one testability rule has to be made. This single rule will be used during the following sections to determine, if the objectives of the current work have met or not. A far–reaching testability design pattern will be #4: "prevent constructors with complex logic". The wording in form of a YaCI Rule is defined as follows:

"when constructor contain keyword new more than 0 times and contain control_flow more than 3 times and has line_count greater 15 then drop error."

7.1.1. Rule Parser

The specification of the YaCI Rule grammar is realised by the usage of DCGs in SWI–Prolog. With the help of this technology the implemented parser is capable to parse a stream of tokens according to the specified grammar. During the parsing process the parser collects all necessary information from the rule and returns the result as a term in the well–known Prolog syntax. The rules are so–called conditional sentences in the form of *"when* ⟨*Premise*⟩ *then* ⟨*Conclusion*⟩*."*. The output of the parser will be used later as the input for the YaCI Rule Analyser. In Listing 7.1 the correct functionality of the rule parser is demonstrated. The entry point of the Rule grammar is the built–in predicate `phrase/2` of SWI–Prolog. It expects as the first parameter the specified DCG grammar and as the second one a list which represents the string tokens. For demonstration purpose the above rule "prevent complex logic in constructor" is used as the input token stream. The result of the parsing process using the DCG specification `parse_rules//1` of the YaCI rule grammar is also shown in the listing bellow.

Listing 7.1: Result term of the YaCI Rule Analyser

```
?- string_codes('/* Prevent complex logic in constructor */\nRule #1          SWI-Prolog
    @ when constructor contain keyword new more than 0 times and contain
    control_flow more than 3 times and has line_count greater 15 then drop error.',
    Rule),
   phrase(parse_rules(Result), Rule).
Rule = [47, 42, 32, 80, 114, 101, 118, 101,  110|...],
Result = [rule(
  when(constructor(and(contain([keyword(new), greater, 0]), and(
    contain([control_flow, greater |...]) , has([line_count, greater |...]) )))),
  then(drop(error)),
  description(short('Rule #1'), long('Prevent complex logic in constructor'))
)] .
```

Generating the internal rule structure is implemented as simple as possible. Each scope and each condition produces a Prolog term containing its relevant information parsed from the token stream, unifies them with a variable and give it back to the calling clause. Maybe in the future or in association with the use of other analysis tools it is required to produce another result exchange data structure. Another can be JSON, XML or other possible formats, for instance. With less implementation

expenditure it is possible to adapt the grammar parser in order to generate the required output structure. There only have to be adapted the related locations in the code. The specification on how to parse the token stream will not be affected by these changes. This fact is a good point of flexibility in order to use the grammar parser in a complete other area or research approach.

A huge disadvantage of the current parser implementation is the missing error handling during the parsing process. In the actual version the YaCI Rule parser returns a result if the provided sentence – the testability rule – is a valid wording according to the specified grammar is introduced in Section 4.1. If a YaCI Rule contains invalid conjunction of components or uses not supported keywords, the parsing process is stopped immediately. The behaviour is the same when the rule includes a simple spelling mistake – like transposed letters, etc. Instead of returning the correct phrased rule sentences at least, the rule parser stops and returns `false`. This described behaviour is shown in the listing bellow. In the second line there is a typical example of transposing two letter (cotian instead of contain). A better solution would be a detailed error handling that advises the developer what kind of failure has happened and at which position in the string. Especially when the rule set is a bigger one, it is hard to find the invalid location in the sentences.

Listing 7.2: Parsing failed caused by a misspelling in the rule

SWI-Prolog

```
?- string_codes('/* Prevent complex logic in constructor */\nRule #1
    @ when constructor contian keyword new more than 0 times and contain
    control_flow more than 3 times and has line_count greater 15
    then drop error.', Rule),
   phrase(parse_rules(Result), Rule).
false.
```

Regarding the objectives on the rule grammar set at the beginning, the developer shall be able to phrase rules which cover the testability patterns introduced in this thesis. This goal is obtained as seen in Section 4.3 where all design pattern and guidelines are transformed into conditional sentences starting with “then” and ending with an according conclusion. All these sentences can be parsed with the current grammar specification and returns the result as expected. In order to extend the grammar respectively adapt them to other Java language characteristics this can be done by implementing the related DCG rules or by adding the keywords for the scope, conditions or something else in the appropriate Prolog files. For instance, a simple expansion can the attach of a new accessor to the `accessor.pl` component when in a future release of the Java language specification a new keyword is added to the set of available modifiers (e.g. private, public and so on). If new components or a variation of the predefined wordings is desired, this can be realised by changing

the related location in the grammar or add new DCG rules in the grammar implementation. In summary, the YaCI Rule Grammar is flexible and can be varied in different ways as described previously.

7.1.2. Rule Analyser

Within this approach, one objective is to realise code analysis by using conditional sentences as the description of an analysis task. These rules shall be used to identify the described characteristics in a software components code base. In order to run those analyses it assumes that two prerequisites have met. On the one hand, the already named rules and a parser which implements the grammar specification, on the other hand the AST of a software component to examine which represents the structure and hierarchical information of the code. Another objective is to realise this analysis tool using the logical programming language Prolog. The YaCI Analyser module is completely implemented in SWI–Prolog regarding to the predefined requirements. JTransformer and its available Eclipse plug–in helped to realise the examination because of the provided feature to create the AST factbase from a Java–based software component. These facts – in context of JTransformer named as PEFs – in order to query the AST and validate its structure against the rule sentences provided by the YaCI Rule Parser.

Listing 7.3: Derived match term from Rule Analyser

```
?- analyse_rule(when(constructor(and(contain([keyword(new),greater,0]), and(     SWI-Prolog
      contain([control_flow,greater,1]), has([line_count,greater,1])))),  Match).
Match = match(class(28597), scopes([constructor(id(28604), name('ComplexConstructor'),
     hints([new(1, [28645]), control_flows(2, [28624|...]), line_count(8)]))]))  ;
Match = match(class(28630), scopes([constructor(id(28701), name('ExampleMatcher'), hints([
     new(2, [28655|...]), control_flows(5,  [28611|...]) , line_count(11)]))]))  ;
false.
```

The listing above illustrates the performed analysis task using the testability rule that aforesaid to prevent complex constructors and logic instructions (testability rule #4). To start the analysis process the premise of the parsed rule sentence is passed into the `analyse_rule/2` clause of the YaCI Rule Analyser. Before running analyses against the AST, it is necessary to consult the factbase which contains the generated AST. Later on, when using the analysis tool, these prerequisites are carried out by the YaCI toolchain itself.

An additional objective for the analysis module is to provide the general infrastructure which ensures that the testability rules can be processed by the YaCI Analyser. This requirement is fulfilled and will be proven in Section 7.2 when the tool is used

to analyse the Joda Time open source framework. Caused by the grammar specification developers are allowed to phrase a wide range of rule sentences not only limited to the context of testability. It is possible to extend the analysis module of YaCI in order to provide the compatibility for those rules. These additional wordings of rules are not yet implemented and can be done in a future work approach.

Listing 7.4: No match can be derived from AST by the analyser

```
?- analyse_rule(when(constructor(and(contain([keyword(new),greater,10]), and(    SWI-Prolog
      contain([control_flow,greater,15]), has([line_count,greater,50]))))), Match).
false.
```

In Listing 7.3 a successfully generated match term is returned by the Rule Analyser. The term `match/2` contains all necessary information to reproduce the affected AST elements which are the description of the code characteristic. In case of the "Prevent complex logic in constructors" rule these different elements are identified: the unique id of the constructor itself, its name (ComplexConstructor) that also represents the class name where the constructor belongs to and a list of the different hints. Calling the constructor of another class (usage of keyword new), different control flows and the count of line numbers in the body of the constructor are the core hints of the appeared testability rule. When more than one match can be derived in the AST factbase, then SWI–Prolog returns this terms one after another via backtracking. The YaCI Rule Analyser simply returns `false`, if the rule sentence does not fit the given structure in the AST. Listing 7.4 demonstrates the described behaviour.

Finally, an important point is here that the current implementation of the YaCI Rule Analyser is evaluated during the development process on explicit created Java source code fixtures to evoke the necessary hints to match the different rules. Furthermore, improvements on the robustness of the analyser reached during the examination against minor projects received from the practical bachelor course on the University of Würzburg and the internal projects of MULTA MEDIO written in Java. The closing evaluation of the analysis module is done in Section 7.2 when examining a real world open source project.

7.1.3. Result Generator

The YaCI Result Generator is used as the connecting link between the rule analyser evaluated before and the integration of YaCI into the Eclipse IDE. During the technical implementation of the analysis toolchain the need of an additional module has arise. This helps to prepare the analysis results in order to use them with the preferred integration module for a desired graphical user interface or IDE.

At the moment the toolchain and the analyses are restricted to Java projects and the associated programming language. For future approaches the result generator gives the opportunity to handle another analysis tool and adapt them to the JTransformer Eclipse plug–in. Or the other way around use the analysis tool in order to attach the result onto another IDE.

Listing 7.5: Match term transformed into JTransformer compatible result term

```
?- generate_match_term(match(class(28597), scopes([constructor(id(28604),      SWI-Prolog
    name('ComplexConstructor'), hints([new(1, [28645]), control_flows(2, [28624,28641]),
    line_count(8)]))])), Result).
Result = result(result_group(28597), elements([constructor(28604), new_keyword(28645),
    control_flow(28624), control_flow(28641)])).
```

To come back to the introduced example rule for evaluating the general functionality of the different YaCI modules, Listing 7.5 shows how the result generator transforms the match term into the according result term. This generated structure fits to the required input of the match marker for the Eclipse IDE integration of the YaCI tool. Flexibility and the capability to adapt this module for using in another approach is important, too. For this purpose the implementation of the Result Generator is as simple as possible and therefore the output of the module can be easily adapted. This allows to use the Result Generator in association with other GUIs in a future approach. If the output of the YaCI Rule Analyser will be extended or the structure changed, then the generator module must be adjusted in order to maintain the compatibility between these two modules, too. The Result Generator contains no additional logic. It simply splits up the match term of the analysis process and generates a new structure for the parsed information.

7.1.4. JTransformer Eclipse Integration

The JTransformer Eclipse Integration module completes the YaCI toolchain. It is responsible for the integration of the core analysis module and its results into the JTransformer Eclipse plug–in. This gives developers the ability to run and observe the analysis tasks specified through the graphical user interface of the Eclipse IDE. One objective is to attach the testability tasks in form of conditional sentences and their description dynamical to the JTransformer Control Center view. The same is planned with the related results after running each analysis as well as the functionality to jump into the location of the rule violation by double clicking on the result entry in the Control Center. These described functionalities are already supported by the JTransformer plug–in. Section 3.3 demonstrates the basic usage of the API based on the Singleton pattern detection analysis in order to attach the definition

and the related results to the user interface. This is done by using the advised way of implementing the JTransformer interfaces. The JTransformer Eclipse Integration will do it in a dynamical manner.

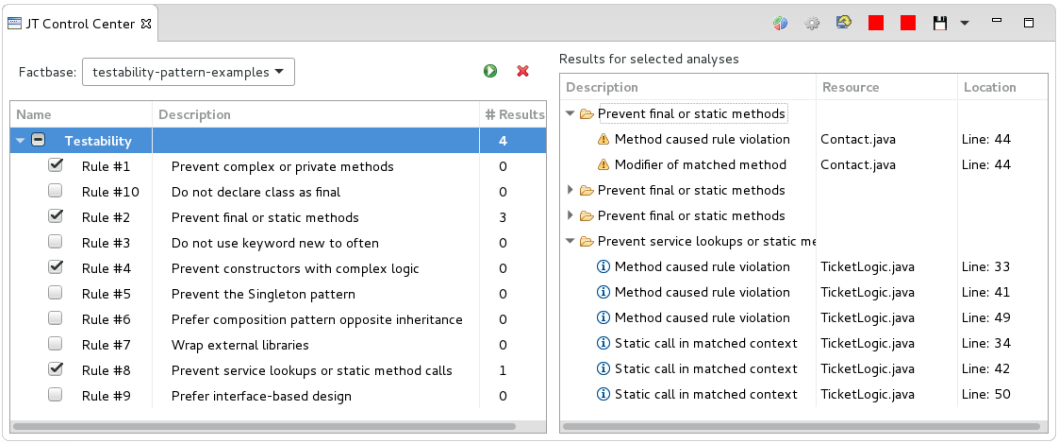

Figure 7.1.: JTransformer Control Center filled with content from YaCI

The previously described objectives are achieved with the first version of the YaCI toolchain. A simple predefined Eclipse project in the current workspace of the project is necessary to initialise and connect YaCI with the JTransformer integration. The selection of an available factbase in JTransformer (represents the AST of a Java–based project) starts an own Prolog process coupled with the graphical user interface of the Eclipse plug–in. Consulting the Prolog file – contained in the YaCI Eclipse project – includes the library of the realised analysis tool and allows the developer to bootstrap the YaCI integration by calling `bootstrap/0` via the Prolog console. A short success message informs that the rule definitions are attached to the JTransformer UI. Within the current approach it is not possible to refresh the Control Center automatically after YaCI has been successfully initialised. The user has to (re)consult the Prolog file in the Eclipse project in order to refresh the Control Center view. Finally, the loaded analysis definitions created by the rule set is shown in the Eclipse perspective.

With the exception of the automatic refresh relating to the JTransformer Control Center, all objectives have satisfied within this approach. Figure 7.1 above shows the related perspective of Eclipse which contains the information extracted from the rule sentences respectively the data received from the analysis results. In the list of available analysis definitions (on the left side of the figure) the developer can select single tasks and run them separately. Showing the affected locations in the source code works as expected, too. The additional feature to jump into the code can be used by the developer when double click on the result entry. The YaCI JTransformer integration runs stable and does not crash during test analyses. Imagine, when the

AST analysis gets into an infinite loop caused by the YaCI Analyser. In this case the Eclipse integration will not react on user interactions and the running Eclipse process has to be killed via command line respectively the task manager.

7.2. Analysing "Joda Time" Library

In the previous sections the basic functionality of the single YaCI toolchain modules are proven. Therefore the general objectives set for the current thesis are met and realised within the analysis tool implementation. The result of the previously performed evaluation should be taken with a pinch of salt. This steps were done by using smaller source code examples in order to give guarantee on the basic functionality of the tool. Because it is important to run YaCI against a real world project to have a more meaningful evaluation result. As the next step the open source date time utility library Joda Time is used to examine a real software component and give short overview on the quality of analysis results. During the evaluation these questions shall be answered more detailed in the following:

- Is it possible to analysis such a huge project without crashing YaCI?
- Are the returned results correct and the described design pattern in the rules are matched?

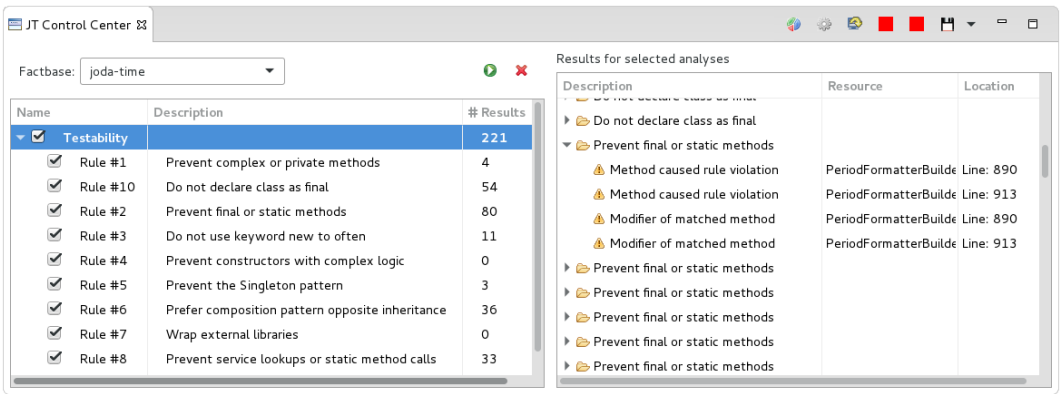

Figure 7.2.: Analysis results derived from Joda Time examination

The answer to the first question is simple: yes, the YaCI toolchain is able to analyse real word Java projects. After successfully creating the AST of the related software component the code analyser is ready for examination. In order to show the correct functionality, all available testability rules are enabled within the JTransformer Control Center before starting the analysis tasks. After the duration of round about 1 minute and 8 seconds the analysis finishes and shows the results in the Control

Table 7.1.: General factbase information of examined projects

Fact Type	**Joda Time**
Entire factbase	356106
callT	69556
classT	529
commentT	7396
fieldAccessT	16156
foreachT	26
forT	188
ifT	2292
methodT	9134
newT	9336

Center result table perspective. These are grouped by their rule id into separate result sets which can be investigated by developers. The runtime of YaCI depends on two factors: the size of the created factbase derived from the Java source code and the complexity of the premise in the phrased analysis rule. The more complex the premise is the more complex are the analysis tasks to examine the AST of the code base under examination. A short overview to the factbase of Joda Time is given in Table 7.2. It contain a narrow range of created facts. For instance, the library contains 529 class declarations and 9134 method definitions.

After the evaluation of the general serviceability of the YaCI toolchain, a closer view onto the received results from the examination is necessary. Therefore, the open source Joda time library – written in Java – is used for analysis purpose. As seen in Table 7.2 below, altogether the YaCI analyser found 222 matches within the 10 testability analyses tasks. This verifies the correctness and feasibility of the code analyser which is implemented in the current approach. Next to the received results, developers can use them to quickly jump into the affected location in the sources and fix the examined design pattern violations. The problem markers which are provided through the Eclipse IDE are working as expected and highlight the location in the code editor. The severity of the problem markers are determined by the conclusion of each YaCI Rule.

Based on this evaluation, the core functionality of the YaCI toolchain and its working order is proven. The received results from analysis tasks are visualised in a

Table 7.2.: Results of Testability analyses

#	**Analysis Description**	**Matches**
1	Prevent complex, private methods	4
2	Prevent final or static methods	80
3	Do not use keyword new to often	11
4	Prevent constructors with complex logic	1
5	Prevent the Singleton pattern	3
6	Prefer composite pattern opposite inheritance	36
7	Wrap external libraries	0
8	Prevent service lookups or static method calls	33
9	Prefer interface-base design	0
10	Do not declare class as final	54
-	*Sum of all*	*222*

graphical user interface. This allows developers to work with them and get a quick overview. In what way the predefined testability rules help to improve the source code in order to write test cases with less effort is not an general objective of the current work. Basically, within this approach we want to show that static code analyses can be realised by the logical programming language Prolog (respectively its implementation SWI–Prolog). Furthermore, the analysis tasks shall be describable through conditional sentences in a more natural wording instead of implementing own analysis tasks. The evaluation shows that these objectives are achieved and a general analysis toolchain is established during the current work.

8. Management Summary

In this chapter the chosen approach and the realised tool will be summarised in order to present the big picture of the implemented toolchain and the achieved objectives. Finally, we want to discuss possible improvements of the YaCI toolchain for the future. A critical view on the achieved approach is already done in the previous chapter where the practicality of the realised analysis tool is evaluated.

8.1. Summary

In modern software engineering approaches the existence of testing support is a necessary prerequisite to deliver stable and tested software. Different modern process models like TDD or BDD guarantee that basically first a test case has to be written before the first line related line of productive code is implemented. A large number of projects are not provided by such a good test support. Adding a set of test suites later on in the project cycle is complex and maybe impossible. The hardness on writing tests for productive code later is the grown structure and raised code design. The investigation of legacy code in fact of its testability will be the focus of the current work and the related static code analysis tool shall be realised within this thesis. Starting point of the analysis tasks shall be different descriptions of preferred design pattern in object–oriented programming languages in order to improve the testability. The planned analysis tool shall be implemented completely in the logical programming language Prolog – respectively its implementation variation SWI–Prolog.

Altogether, ten principles on writing better testable code and implementing test cases with less effort are collected from literature research and different experiences of various leading authorities on this area. These principles describe different patterns and preferable design guidelines in the context of object–oriented programming languages in order to provide less effort on writing associated test suites. After the specification of an own YaCI Rule Language, we are able to paraphrase these guidelines in conditional sentences. Using the YaCI Rule Parser the rule sentences – represented as strings – can be parsed according to the grammar and transformed

to the internal rule representation of YaCI. The grammar specification and the accompanying parser implementation are successfully realised in SWI–Prolog by using the built–in functionality of Definite Clause Grammar and quasi quotations.

To accomplish analysing source code a representation is necessary which contains information about structure and hierarchy of it. Examine a simple `*.java` file will not fit for purpose because it contains only a stream of tokens without additional knowledge about it. Therefore, the AST must be derived from source code during a parsing process. Creation of the AST is realised by using the JTransformer tool from the University of Bonn. This provides the necessary functionality to create the AST and generates the associated factbase that contains all structural and hierarchical information necessary for analyses.

In the next step the core module implementation of the toolchain is realised. It uses the parsed rule sentences in order to examine the given AST of a Java software component. The input is represented as nested Prolog term specified by the internal rule representation generated by the YaCI Rule Parser. The result of an analysis task is a match term that includes all affected AST elements. These hint elements are defined by the wording of the provided analysis rule. The basic usability of the core module is achieved during the current thesis. It is also confirmed during the evaluation process where we analyse the open source "Joda Time" library according to the predefined testability rules. At this time the analysis results can be observed via command line interface. But it is hard to read and understand for humans because the result term only contains identifiers (unique integer value for each element) of the matched AST elements.

In order to visualise the analysis results in a readable and understandable nature a graphical user interface is necessary. The mentioned JTransformer plug–in for the Eclipse IDE already provides a GUI for analysis results written in the preferred JTransformer manner (means to implement the analysis definition directly in Prolog). In our approach the JTransformer perspective is used to attach the analysis results of the generated result terms derived from the YaCI Rule Analyser. The integration of a GUI into the work flow of the toolchain is also achieved within this work. Developers are able to observe the attached result terms via the JTransformer Control Center perspective in the Eclipse IDE.

Finally, all set objectives are achieved within this thesis. The first runnable version of the analysis toolchain works as expected and accomplishes the basic functionality. This includes paraphrasing analysis rules in form of conditional sentences and the examination of Java source code using its AST. The rule wording specifies the characteristic of the analysis task and the obtained result of the examination. As well as the integration of the derived results which are attached to a suitable GUI.

Outcome of the current approach is a “ready for use” toolchain to analysis software components according to simple “when ... then ...” rules implemented in a logical programming language.

8.2. Future Work

In the first version of YaCI we focus on the basic functionality and the general objective to realise a runnable analysis tool on the basis of Prolog. Due to this, there are various improvements rather that will increase the range of functions for YaCI. In the following, this possible enhancements are discussed in the scope of future work.

At the moment the possibility to give feedback during the rule parsing process is missing in the YaCI Rule Parser implementation. If rule sentences contain spelling mistakes or invalid components which are uncovered by the grammar specification, then the parsing process fails without additional error information. This behaviour does not allow any conclusions regarding to the occurred failure. In the next version of YaCI some kind of error handling for parsing the rule sentences will be helpful during paraphrasing further analysis rules. Furthermore, the behaviour of the parser can be improved when an invalid rule is included in the collection of rules. At the moment the parser failed and nothing will be returned. A preferable behaviour will be skipping invalid sentences and return only the correct paraphrased rules at least.

The grammar specification allows wording of analysis rules that can not be analysed with the current implementation of the YaCI Rule Analyser. This is justified by the missing implementation in the analysis module to examine the AST according to the described characteristics based on the rule wording. Within this the performance of the analysis step is a good point to investigate. First and foremost, YaCI will be a proof of concept toolchain. Therefore the functionality and general usability and not the performance was centre of attention.

Integrating the analysis tool into an existing continuous integration life cycle is already mentioned inside the current thesis. Currently the static code analysis is only usable on a local workstation in contextually with the Eclipse IDE. The integration of YaCI into a automatic build environment will be a useful feature in order to run defined analyses each time before the software component is delivered to the customer. Because of the modular architecture of YaCI only the Eclipse JTransformer integration module must be exchanged by a specific implementation that connects the analysis tool with the build server.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bin94] Robert Binder. Design for Testability in Object–Oriented Systems. *Comm. ACM*, 37(9):87–101, 1994.

[BLS02] Benoit Baudry, Yves Le Traon, and Gerson Suny'e. Testability Analysis of UML Class Diagram. In *Proceedings of the Metrics Symposium*, pages 54–63, June 2002.

[BSMP+04] Tim Bray, Michael Sperberg-McQueen, Jean Paoli, François Yergeau, and Eve Maler. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[BVD04] Magiel Bruntink and Arie Van Deursen. Predicting Class Testability Using Object-Oriented Metrics. In *Source Code Analysis and Manipulation. Fourth IEEE International Workshop*, pages 136–145. IEEE, 2004.

[BW84] Alan Bundy and Lincoln Wallen. Definite Clause Grammars. In *Catalogue of Artificial Intelligence Tools*, page 26. Springer Berlin Heidelberg, 1984.

[CM84] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (2nd Edition)*. Springer-Verlag, New York, USA, 1984.

[Cre97] Roger F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara*, volume 97, page 18, 1997.

[Fow11] Martin Fowler. *Domain–Specific Languages*. The Addison-Wesley Signature Series. Addison–Wesley, 2011.

[Fre91] Roy S. Freedman. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Hev08] Misko Hevery. Guide: Writing Testable Code. *The Testability Explorer Blog*, November 2008.

[HML03] Dirk Heuzeroth, Stefan Mandel, and Welf Lowe. Generating Design Pattern Detectors from Pattern Specifications. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 245–248. IEEE, 2003.

[Hof08] W. Dirk Hoffmann. *Software–Qualität*. EXamen. press series. Springer Berlin Heidelberg, 2008.

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std. 610.12-1990*, pages 1–84, Dec 1990.

[ISO01] ISO/IEC. Software Engineering – Product Quality (ISO/IEC 9126). Standard, 2001.

[JSO13] The JSON Data Interchange Format. Technical Report Standard ECMA–404 1st Edition, ECMA, October 2013.

[Kay93] Alan C. Kay. The Early History of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL–II, pages 69–95, New York, USA, 1993. ACM.

[Kos13] Lasse Koskela. *Effective Unit Testing: A guide for Java developers*. Manning Publications Co., 1st edition, 2013.

[LR09] Bernhard Lahres and Gregor Rayman. *Praxisbuch Objektorientierung: Das umfassende Handbuch*. Galileo Press, Bonn, 2 edition, 2009.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2008.

[MS96] John D. McGregor and Satyaprasad Srinivas. A Measure of Testing Effort. In *Proceedings of the USENIX Conference on Object-Oriented Technologies, COOTS'96, Toronto, Ontario, Canada, June 17–21*, 1996.

[Mul07] Emmanuel Mulo. Design for Testability in Software Systems. Master Thesis, Delft University of Technology, July 2007.

[Osh09] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[PW80] Fernando C.N. Pereira and David H.D. Warren. Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[RK84] Davis Randall and Jonathan J. King. The Origin of Rule-Based Systems in AI. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, pages 20–52, 1984.

[SRK07] Daniel Speicher, Tobias Rho, and Günter Kniesel. JTransformer – Eine logikbasierte Infrastruktur zur Codeanalyse. *Softwaretechnik–Trends*, 27(2), 2007.

[SS06] Péter Szabó and Péter Szeredi. Improving the Prolog Standard by Analyzing Compliance Test Results. In *International Conference on Logic Programming*, pages 257–269. Springer, 2006.

[WH13] Jan Wielemaker and Michael Hendricks. Why It's Nice to be Quoted: Quasiquoting for Prolog. *CoRR*, abs/1308.3941, 2013.

[Wie16] Jan Wielemaker. SWI–Prolog Implementation, off. GitHub Repository. `https://github.com/SWI-Prolog/swipl-devel`, 2016.

[WMV03] Laurie Williams, Michael E. Maximilien, and Mladen Vouk. Test–Driven Development as a Defect–Reduction Practice. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium*, pages 34–45. IEEE, 2003.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

Erklärung

Ich, Thomas Handwerker, Matrikel-Nr. 1995289, versichere hiermit, dass ich meine Master Thesis mit dem Thema

> *Testing Source Code with the*
> *Logic Programming Language Prolog*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Master Thesis zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Universität abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Würzburg, den 30ten Juli 2016

Thomas Handwerker

A. Available Sources

Along with this thesis there will be all sources provided as a CD. This also includes the created implementations which are done to realise the Yet another Code Inspector toolchain. Additional, the sources of the implementation are available at the official GitLab of the University of Würzburg. They are released under the terms of the license specified in the projects repository or if not specified, under the MIT license.

Thesis All necessary LaTeX source files of the thesis are located in the `/thesis` directory. The base document is `thesis.tex`. Before compilation, ensure that all necessary packages are installed:

- `backnaur`
- `longtable`

Distribution The packaged version of the YaCI toolchain is located at the `/distribution` folder at the CD. The SWI–Prolog library can be installed by using the `pack_install/1` clause.

Implementation All source files of the YaCI toolchain are located at the `/implementation` folder of the CD. A detailed installation instruction for YaCI is provided in Appendix D.1. The implementation of YaCI includes the following modules:

YaCI Rule Parser
The parser implementation and the related rule grammar specification is located in the `/implementation/src/grammar` directory.

YaCI Rule Analyser
The core analysis tasks are done by the rule analyser. Its implementation is located in `/implementation/src/analyser` directory.

Result Generator
Result terms derived from the YaCI Rule Analyser are transformed into match terms. This allows the integration of the analysis results into the GUI. Its implementation is located in the `/implementation/src/generator` directory.

JTransformer Integration
In `/implementation/src/jt-integration` the source of the JTransformer integration of YaCI is provided.

YaCI Analyser (Eclipse)
The predefined Eclipse project is located in `/implementation/src/yaci-analyser`.

Test Suites
In `/implementation/test` all available tests for the different YaCI modules are provided.

Joda Time Library During the evaluation of the YaCI toolchain it was recommended to not only examine small and prepared Java projects. Therefore, the open source Java date time utility "Joda Time" was used to run analysis on a real world project. The version used for the evaluation is located in the `/evaluation/joda-time` directory of the CD. The sources are also available at GitHub at `https://github.com/JodaOrg/joda-time`. At the time of the evaluation we analyse the library on top of the commit id `1e86b1eb`.

B. Testability YaCI Rules

In this appendix we want to give an overview to the testability rules which are extracted from literature and different guidelines of object–oriented programming. The patterns are introduced in Section 2.2 and are rewritten as YaCI Rules in the form of conditional sentences. This collection is only a basic set of rules and can be extended according to the specified rule grammar. See Chapter 4 for detailed information related to the YaCI Rule grammar.

Listing B.1: Basic set of testability YaCI Rules

RULES

```
/* Prevent complex, private methods */
Rule #1 @ when method is private and has line_count greater 10 and contain control_flow
    more than 3 times then drop info.
/* Prevent final or static methods */
Rule #2 @ when method is final or method is static then drop warning.
/* Do not use keyword new to often */
Rule #3 @ when method contain keyword new more than 4 times then drop warning.
/* Prevent constructors with complex logic */
Rule #4 @ when constructor contain keyword new more than 2 times and contain control_flow
    more than 2 times and has line_count greater 10 then drop error.
/* Prevent the Singleton pattern */
Rule #5 @ when constructor is private and method has return_type of class and is static and
     contain access of static field then drop warning.
/* Prefer composition pattern opposite inheritance */
Rule #6 @ when class has inheritance of depth greater 3 then drop warning.
/* Wrap external libraries */
Rule #7 @ when method contain call of class_method more than 0 times not in package then
    score warning.
/* Prevent service lookups or static method calls */
Rule #8 @ when method contain call of class_method more than 0 times then drop info.
/* Prefer interface-based design */
Rule #9 @ when method is public and has parameter not of type interface or constructor has
    parameter not of type interface then drop info.
/* Do not declare class as final */
Rule #10 @ when class is final then drop info.
```

C. Code Snippets

This appendix contain different code snippets to present basic implementation details of the YaCI toolchain.

C.1. Definition of Grammar Components

Listing C.1 and C.1 shows the specification of two basic components available in the YaCI Rule grammar. This implementations are encapsulated in order to provide a better extensibility for the rule grammar. New keywords for the different components can be simply append to this files.

Listing C.1: Specification of various available components in the rule grammar

```prolog
:- module('components/accessor', [
  accessor//1
]).
:- use_module(library(dcg/basics)).

accessor(abstract) --> "abstract".
accessor(final) --> "final".
accessor(private) --> "private".
accessor(protected) --> "protected".
accessor(public) --> "public".
accessor(static) --> "static".
accessor(synchronized) --> "synchronized".
```

```prolog
:- module('components/comparator', [
  comparator//1
]).
:- use_module(library(dcg/basics)).

comparator(greater) --> "greater" ; "more than".
comparator(greater_equal) --> "greater_equal".
comparator(less) --> "less" ; "less than".
comparator(less_equal) --> "less_equal".
comparator(equal) --> "equal" ; "of".
comparator(not_equal) --> "not_equal" ; "not of".
```

C.2. Exclusion of Classes in YaCI Analyser

The following listing shows the `exclude_class` module used by the YaCI Rule Analyser. It also includes a short example factbase of classes that shall be ignored during the analysis process. The list can be extended by adding new facts of the type `excluded_class/2`.

Listing C.2: Specification of classes to exclude from analysis

```prolog
:- module('exclude_class', [ is_class_excluded/2 ]).

:- discontiguous exclude_class:excluded_class/2.

is_class_excluded(PackageName, ClassName) :-
  excluded_class(PackageName, ClassName).

% extend this list of facts in order to ignore the specified classes.
% and their package name
% form: excluded_class(PackageName, ClassName).
excluded_class('java.lang', 'Integer').
excluded_class('java.lang', 'String').
excluded_class('java.util', 'ArrayList').
exlcuded_class('java.util', 'HashMap').
excluded_class('java.util', 'Random').
```

D. User Manual

In this appendix the installation of the YaCI toolchain is describe in detail. This allows to set up the analysis tool locally on a workstation and examine software components written in Java.

D.1. Installation

YaCI is a SWI-Prolog based source code analysis tool. It includes different modules which provide a complete toolchain that can be used in the Eclipse IDE. To use YaCI in the local development process there are some prerequisites to met.

SWI-Prolog

First, there have to be a local installation of SWI–Prolog. The recommended version will be 7.2.3 (this version is also used during development and testing the analysis toolchain). Since YaCI use *quasi quotations* which are introduced in version 6.4.0 this release of SWI–Prolog is the minimum requirement because of DCG and quasi quotations built–in predicates are necessary. The installed version can be checked using the command:

```
$ swipl -v                                                    Command Line
```

If SWI-Prolog is not available the `swivm` command line interface is a helpful tool to install and handle local installations.

Eclipse IDE and JTransformer

Creating the necessary abstract syntax tree of a Java project is done by using the JTransformer plugin for the Eclipse IDE. This requires the installation of Eclipse $\geq$ Mars and the available JTransformer Plugin $\geq$ 4.1.0. There is also a local installation of the Java Development Kit necessary. Recommended is JDK 7.

Eclipse IDE can be downloaded via official mirrors. For the current work Eclipse Mars is used and tested. The necessary JTransformer plugin is available via the official update site (`http://sewiki.iai.uni-bonn.de/public-downloads/update-site-jt/`). After installing the JTransformer Eclipse plugin we have to change the location of the `swipl` executable. This allows Eclipse to start a SWI–Prolog process. The configuration can be done within the following steps:

1. Go to *Window* → *Preferences* and select in the tree view the item *PDT* → *Prolog Processes*.
2. On the right side of the dialogue select *SWI Prolog* and click on *Edit...*.
3. Click on *Browse...* in the line of the *Prolog executable* and select the swipl binary in the local installation path of your SWI–Prolog (if swivm is used for installing SWI–Prolog, then the binary is in `/home/<user>/.swivm/versions/<version>/bin/`)
4. Click on *Apply* and close dialogue with *OK*.

Installing YaCI SWI-Prolog library

The YaCI toolchain and all its accompanying modules are packaged in the "yaci" pack that is available through the files attached to this thesis. It can be added to the local SWI–Prolog installation by using the following command in a running Prolog process:

```
?- pack_install('/path/to/yaci_<version>.tgz').                    SWI-Prolog
true.
```

Configure YaCI Eclipse Project

For the usage of the analysis tasks that can be processed by YaCI it is recommended to use the Eclipse IDE. All results of the analysis will be attached to the JTransformer Control Center as well. In order to run analysis with YaCI we need a general project in the current Eclipse workspace. This serves as the bridge between JTransformer Eclipse plugin and the YaCI toolchain. A starter project for Eclipse is already included in the YaCI SWI-Prolog library installed previously. Before importing the project into Eclipse you can copy the project folder into your workspace (e.g. `/home/<user>/workspace`). The prepared project can be found in `/home/<user>/lib/swipl/pack/yaci/eclipse`.

1. Right click to the *Package Explorer* in Eclipse.

2. Select *Import...* to open the Import dialogue.
3. In the wizard select *General* → *Existing Projects into Workspace* and click *Next*.
4. Browse to your local workspace and select the "yaci-analyser" folder which contain the `.project` file for Eclipse.
5. Select the "yaci-analyser" entry in the list of available projects and import the project.

Finally, the `BasePath` and `RuleFile` must be adapted to your local environment characteristic which is defined in `load.pl`. But for now, the YaCI toolchain is installed, the Eclipse project for YaCI is set up, and is ready for usage in the development process. How to run an analysis task to examines a specific Java project is explained detailed in the next section of the manual.

D.2. Run Analysis Tasks

First of all the AST of the Java project must be created by using the JTransformer Eclipse plugin. Therefore, the following steps must be done:

1. Open the Eclipse workspace and right click on the desired project in the *Package Explorer*.
2. Select *Configure* → *Assign JTransformer Factbase* in the context menu.
3. In the appearing dialogue a name of the factbase can be entered and then starts the creation process with *OK*

All generated factbases are selectable in the JTransformer Control Center view in Eclipse. By selecting one of the list items an associated SWI-Prolog process is starting in the Prolog console and consults the selected factbase. Now we are ready to bootstrapping YaCI:

1. Open the JTransformer Developer view in Eclipse.
   a) *Window* → *Perspective* → *Open Perspective* → *Other...*
   b) Select *JTransformer Developer* from the list of available perspectives.
2. In the *JTransformer Control Center* view you can select one of the available factbases which are created before.
3. Consult the YaCI Analyser project:
   a) Right click on `load.pl`

b) Click on *Prolog Development Tools → (Re)consult*

4. In the Prolog console start YaCI by typing `bootstrap.` and press enter. This will read in the rule file, parse the rules and attach the resulting analysis definitions to JTransformer. In the console a short success message will appears like: "initialisation finished: 10 rule(s) loaded."

5. In order to refresh the JTransformer Control Center we have to reconsult the YaCI Analyser (repeat step 3).

The YaCI Analyser is ready to run analysis tasks on the selected factbase by picking all or only single analysis rules. Start analysis by pressing *Run all enabled analysis* (the green play button). After a while the analysis results will appear right to the rule list in the result table. You can jump into the associated line of code by double clicking the result entry.

D.3. Create Packaged Library Version

The repository contain a build script for packaging the Yet another Code Inspector SWI-Prolog library. This is a simple shell script implemented in `build.sh`. Running the script via command line will create an archive that contains all necessary files to publish the YaCI toolchain to the official SWI–Prolog packs website or to use it on another workstation.

The package contain the following source files:

- `pack.pl`: Necessary package informationen for SWI-Prolog packs website
- `YaCI Grammar Parser`: contain grammar specification and parser
- `YaCI Rule Analyser`: the core analyser module of YaCI
- `YaCI Result Generator`: Transforms analysis match term into result term which can be attached to the JTransformer plugin
- `YaCI JTransformer Integration`: Module to integrate YaCI and its features into the Eclipse IDE
- `YaCI Eclipse Project`: Bootstraping project for Eclipse integration

D.4. Run Tests

The development of the YaCI modules are provided by simple test suites. These are written according to the *Test Anything Protocol*. In order to run the test specifications the "tap" library for SWI–Prolog must be installed on the workstation. Simple run the following command via swipl

```
?- pack_install(tap).                                          SWI-Prolog
true.
```

will be install the package to `home/<user>/lib/swipl/pack`. After that you can run the various test suites for the YaCI modules by executing the following commands:

```
# test suite for YaCI Rule grammar                             Command Line
$ swipl -q -t main -f test/grammar/grammar.test.pl
# test suite for YaCI Result Generator
$ swipl -q -t main -f test/generator/generator.test.pl
# test suite for YaCI Analyer (1-9)
$ swipl -q -t main -f test/analyser/analyser_0*.test.pl
```