Validating the Object Calisthenics

Evaluation and Prototypical Implementation of Tool Support

Student Research Paper

for the certification examinations for the

Bachelor of Science

Degree Course Applied Computer Science

Baden-Wuerttemberg Cooperative State University Karlsruhe

by

Fabian Schwarz-Fritz

Publication date:	January 4, 2014
Time required for processing:	12 Weeks
Matriculation number:	212024979
Course:	TINF11B2
Vocational training company:	SAP AG, Walldorf
Reviewer:	Daniel Lindner
Reviewer's company:	Softwareschneiderei GmbH, Karlsruhe

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Validating the Object Calisthenics - Evaluation and Prototypical Implementation of Tool Support*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Diplomarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Sekretariat meines Studiengangs an der DHBW Karlsruhe abzugeben habe. Als Abgabetermin giltbei postalischer Übersendung der Eingangsstempel der DHBW, also nicht der Poststempel oder der Zeitpunkt eines Einwurfs in einen Briefkasten der DHBW.

Karlsruhe, den January 4, 2014

Fabian Schwarz-Fritz

Sperrvermerk - TODO ??? INFORMIEREN. Gehoert das der SAP AG

Die Ergebnisse der Arbeit stehen ausschließlich dem auf dem Deckblatt aufgeführten Ausbildungsbetrieb zur Verfügung.

Abstract

Hier bitte den Abstract Ihrer Arbeit eintragen. Der Abstract sollte nicht länger als eine halbe Seite sein. Bitte klären Sie mit Ihrem Studiengangsleiter ab, ob der Abstract in englischer oder deutscher Sprache (oder möglicherweise sogar in beiden Sprachen) verfasst werden soll.

Contents

Abbreviations

DHBW	Duale Hochschule Baden-Württemberg
OSS	Open Source Software
Sem	Semester

List of Figures

List of Tables

1 Introduction

Jeff Bay's "Object Calisthenics" [Bay, 2008] are nine rules that train the software developer to write better object oriented code. He created concrete rules out of general software principals and patterns. These rules shall be applied in a short exercise, usually about two to four hours. With these concrete rules the trainee doing the exercise can improve his software development skills, which is helping him when applying general software principals and patterns to real world software projects. The exercise and the reasons for it are described in this chapter, in section [**?**].

This paper is divided into three chapters.

Chapter 2 describes Jeff Bay's rule and what let him establish the nine rules. It poses Jeff Bay's reasons for today's problems of software development.

The chapter describes every rule step by step. Within this description, a source code example is given. A bad example firstly exemplifies how the code looks like without the rule. A good example, validating the rule, then shows the advantages of the resulting source code when applying the rule. These examples are described shortly. After describing the examples, it researches concepts behind the rule. It describes the problems that would occur without the rule. After explaining the problems behind the rule and Jeff Bay's rule, a more detailed analysis ensues. In this detailed analysis, patterns and principals are described and ideas and concepts are examined. Section 2.3 describes these principles detailed for every rule, step by step. In the end, the section 2.4 furthermore discusses the rules shortly.

Chapter 3 investigates possible tool support to validate the Object Calisthenics. First, the section 3.1 describes general advantages of tool support. The section 3.3 goes through every rule presents the possibilities to validate the given rule. Advantages and disadvantages of the implementation are explained and a short conclusion of every rule is drawn. Section 3.4 then summarizes the results of the evaluation of rule validation. Lastly, section 3.5 further ideas of rule validation and links for further work.

The last chapter 4 describes a prototypical implementation done during this research. The prototype shows the practicability of the evaluation results.

1.1 Exercising Better Object Oriented Programming Skills: a Concept by Jeff Bay

Jeff Bay's "Object Calisthenics" [Bay, 2008] are an exercise to improve the quality of Object Oriented code. According to him, the chapter "will give new programmers an opportunity to learn best practices while writing their own code." [Bay, 2008, p. 70].

The book "The thoughtworks anthology. Essays on Software Technology and Innovation"[ThoughtWorks in p. 70-79] was released in 2008. The whole paper consists of thirteen chapters discussing various topics and ideas on how to improve software development. The essays discuses problems and ideas on languages, tool support, software principals and software quality. One of the paper's chapter describes the rules of the Object Calisthenics. Furthermore it shortly describes their purpose and outcome and why the author Jeff Bay created these rules.

All essays in the book [ThoughtWorks inc., 2008] are written by developers working at the company "Thoughtworks inc". The company is well known for creating, designing and supporting high quality software. The company is to be said to be one of the most future oriented company in terms of technology and software principals. They describe themselves as "[...] a software company and community of passionate individuals whose purpose is to revolutionize software design, creation and delivery, while advocating for positive social change [...]" [ToughtWorks inc.].

The Object Calisthenics are nine programming rules helping to write good object oriented code. But moreover the Object Calisthenics are an exercise to improve the quality of Object Oriented code. Good Object Oriented Code is hard to learn when coming from procedural code. Many developers think in Object Oriented code – but do they really write good Object Oriented Software? That is the question that Jeff Bay poses in his essay.

Usually the developer doesn't use these rules in real world project but applies them in short two hour exercises in which he designs and implements minimalist software with little requirements. This could be a Minesweeper or a Tic Tac Toe game for example. These training challenges should lead the developer to write better code and be more aware of code quality in real world projects.

With these little training sessions, the Object Calisthenics help to create highly object oriented code in small projects. When applying the rules, the developers automatically fulfill many important software patterns and principals leading to higher code quality than the code would have without the given rules. By training developers to focus the rules they automatically apply various helpful and important software principals and software patterns.

The developer is supposed to "spend 20 hours and 1,000 lines writing code that conforms 100?percent? to these rules" [ThoughtWorks inc., 2008, p. 80]. He is convinced the developer will "break old habits" [ThoughtWorks inc., 2008, p. 80]. Furthermore he promises that the exercise will change the way "that you may have lived with for your whole programming life."[ThoughtWorks inc., 2008, p. 80]. According to Bay, this change is the result of the developer's need to rethink and lateral thinking. He is genuine that by this process the developers perspective on existing code and the way he will write code in the future will change radically. As just described this rethinking is only possible in small, lucid projects. This is for example a project with 1,000 lines of code, as suggested by Jeff Bay.

However, Jeff Bay's idea is that by this process of rethinking the code quality of the code written by the developer will improve. Of course this will only work if the developer recognizes the ideas and principals behind the rules, and hereby accepts the positive outcome of the resulting code. Hopefully he is able accept the positive value of the resulting code. When he is working in real world projects he hopefully remembers parts of the rules, resulting ideas

or principals and concepts behind the rules. Because of the improvement of his software development skill, he is then able to apply the concepts to real world project. The result is a tremendously improved code quality in the real world project.

Improving the quality of software's implementation by little training sessions - that is his idea, basically. Jeff Bay included also this idea in the name of the rules. The word "Calisthenics" indistinguishable describes the approach, the idea and the outcome of the exercise.

As already said, the background of every rule is described in chapter 2.

1.2 Tool Support to Validate the Object Calisthenics

The purpose of the Object Calisthenics was already described. However, in this paper the focus lies on the evaluation and prototypical implementation of tool support, validating the rules of the Object Calisthenics.

Evaluating the possibility to create tool support validating the Object Calisthenics is the main part of this report. It has two main advantages - efficiency and quality.

Providing tool support for the rules of the Object Calisthenics could improve the time the developer uses to conducts the exercise. This efficiency increase simply makes it easier to do the exercise.

Furthermore it might reveal rule violations that where not detected by the developer yet. Therefore it could guarantee that the developer sticks to the rules. This is a quality improvement.

Therefore, the next chapter 2 describes the rules Object Calisthenics to understand the software principals and quality metrics behind the Object Calisthenics.
As already mentioned, the ensuing chapter 3 discusses tool support for every rule. the prototypical implementation is described in chapter 4.

2 Object Calisthenics by Jeff Bay - Patterns and Principals

This chapter describes the patterns and principals behind the Object Calisthenics.

Firstly, in section **??** the advantages of the Object Calisthenics are described. Before explaining the advantages of Object Orientation, the problems of non object oriented code are discussed.

In the second section 2.3, Jeff Bay's rules for better object oriented programming are described step by step. Every rule is described, explained and a good and bad example are given. The examples conduce to understand the problem and the solution given by the rule. Furthermore patterns and principals behind the rules are explained.

2.1 Problems of procedural code

There seem to be problems with procedural, non object oriented code in software. By saying non object oriented software, procedural code is meant. Jeff Bay lists the most important ones in [ThoughtWorks inc., 2008, p.70]. These disadvantages and problems of procedural source code are explained in the following.

2.1.1 Problems of procedural code

In procedural code the behavior is described by step by step operations resulting in an algorithm.

No bundle of data and behavior

The operations, also called commands, in procedural code usually access multiple data structures storing and managing data during the execution time. In procedural code steps can be consolidated by using methods with output and input parameters. Methods can summarize multiple commands to an abstract step that can be executed by a caller. Therefore procedural code provides a solution for structuring algorithms.

But there is another problem remaining. By simply looking at the data structures, the developer cannot see possible operations on the data structure. Of course, he is able to see what is stored in the data structure. However, he is not able to see which operations can be executed on the data structure and how the data structure is supposed to be used. He can only get that information by reading through the algorithm step by step and remembering the possible operations on the data structure. In procedural code there is not clear coherence between the data structure and its problem specific operations. There is no solution for the problem of bundling data and behavior given in procedural software development.

Furthermore there is no encapsulation of the data structures, hiding the implementation from the abstract definition of possible operations. Also, procedural software lacks of encapsulation. The concept of scoping therefore realizing encapsulation is not given in procedural code.

Maintainability

The just described problem of the low cohesion of data structure and problem specific operations makes maintainability of procedural code hard. Source code of long lasting software has to be maintained years after the initial design and the first implementation. In old and large code bases changes are hard to conduct. This is because the cohesion of behavior and data structures and the bad encapsulation between the different data structures is low. To be able to compensate the cohesion, relatively much documentation and much time to read the code is necessary. Therefore the low cohesion makes the conduction of maintenance software changes hard.

Not understandable

Another problem, closely related to the problem of maintainbility, is the understandability of code. As already said, the problem of the low cohesion makes it hard to dedicate the operations to the data structures. The difficulty to dedicate operations to the appropriate data structures makes the understanding of the code hard. By reading through a part of an algorithm, the developer does not know which operations will be executed on which data structure. Secondly, when there are multiple operations on many different data structures, the developer has to remember all these state changes of the data structures. Therefore procedural code is often hard to understand.

No modularity

In modern software development the separation in different modules is a core concept.

Modular programming "breaks down program functions into modules, each of which accomplishes one function and contains all the source code and variables needed to accomplish that function."[About.com]. In [About.com] Juergen Haas also states that the concept of modularity enables the programmer to debug and maintain difficult source code more easily.

A module only contains all necessary information and operations to perform the steps necessary. All other information is not visible by the module, it is hidden in other modules that are responsible for other tasks. However, within one module the developer is not distracted by other operations or data structures that are not related to the module, because they are hidden.

Structure

As already said, procedural code is only structured in different algorithm steps. The data structures are can't be encapsulated. This makes structuring procedural code extremely hard, because there is no mechanism enabling the developer to give structure by hiding information. The only way to give structure is to sort different methods in different files. But even if a data structure and its functions are put in a different file, the data structure is not hidden from the other files. The data structure can be manipulated in every other position of the program.

Overview

This point is closely related to the just explained problem of structuring the program. The only way to sort or separate something can be done by using separate files. But still, this does not solve the problem of information hiding and therefore an overview is not given.

In procedural code it is not possible to put problem related data into a abstract data types.

No reusability

In procedural code it is hard to reuse code. This is a result of the missing encapsulation and the missing modularity that was explained previously.

As already said in procedural code, a method often interacts with many variables of the system that are not encapsulated. A method can interact with all system variables. This leads the developer to not separate the program into different parts with their own responsibility and their own well encapsulated data structures and operations.

When using multiple files in procedural code, at some point the order of inclusion of these files has to be determined. When another file is imported there is no guarantee that the included files does already execute an operation. Furthermore, cyclic dependencies can occur that have to be resolved manually.

However in object oriented software the concept of encapsulation, respectively scoping, of data structures is integrated in the concept of object orientation. In good object oriented software every module and object has a clear scope and is well encapsulated. This makes it easy to reuse the object or module. Firstly, it's dependencies are clear visible. They are not order specific, because they do not determine the order of the inclusion of files, but they determine the types that are used by the class. Secondly, the inner state expressed by data structures is hidden from everything that is not part of the module. The developer does not have to apprehend other modules to change internal states of the module. Thirdly, the operations that can be executed on the object are well defined. The developer can document those interfaces to help developers using the interface to understand how the module or object works.

2.2 Advantages of object orientation

The described problems and disadvantages that where described in the previous chapter 2.1, led to the use of object orientation. All in all it is generally stated that object orientation and its related principals saves us from the problems, described in 2.1 and provides a proper solution.

According to Jeff Bay the change from programming in an procedural way towards programming object oriented style is hard to learn. "Transitioning from procedural development to object-oriented design requires a major shift in thinking that is more difficult than it seems" **??**p. 70]bay2008. According to Jeff Bay, however, the procedural way of thinking is still in the mind of many developers. "Transitioning from procedural development to object-oriented design requires a major shift in thinking that is more difficult than it seems" **??**p. 70]bay2008. "Many developers assume they're doing a good job with OO design, when in reality they're unconsciously stuck in procedural habits that are hard to break" **??**p.70]bay2008. Not only the problem of the "schwierigketi" hardness to learn a different way of thinking and coding when

changing from procedural to object oriented code "ist eine barriere". But also the fact that it is easy for a developer to write procedural code. The disadvantages of procedural code are do not occur immediately. Many of the problems that "come by" an procedural source code occur late. That means that the developer does not get an immediate feedback about the quality of his code.
And even worse, when he is not "erkennen" and "feel" the disadvantages of procedural code, 'there is no motivation for him to change that behaviour'. Furthermore when creating code and not "warten" legacy code it is generally easier to write procedural code than to write object oriented code. The creation and the management of objects and the strict rules of encapsulation that 'auszeichnen' good object oriented software seems harder to do than the creation of procedural software. Therefore it is 'easier' for the developer to write procedural code, than it is to write object oriented code.

Using the concept of object orientation helps to create software that as multiple advantages. These advantages are described later in this section. Before I will explain the advantages or core concept stated by Jeff Bay, I will summarize the "eigenschaften" and compare procedural and object oriented code very shortly.

Procedural code is more like a step-by-step description of "abfolgenden" commands. Seldom information hiding is used in procedural code. Furthermore multiple operations manipulate data, but there is no bundling of data and behaviour in one "abgeschlossen" unit.

In object oriented software however such a bundling in units exits. Data and behavior are put together in classes and packages. A unit using another unit uses the interface of the unit to determine possible operations, and uses it "wie gewuenscht". The actual implementation and the actual data structures are hidden from the calling unit.

In object oriented software tasks that are not "aufgabenbereich" of a unit are always delegated. Otherwise the unit would not do exactly and only what it is supposed to do. To satisfy the definition of a module it need to have a high cohesion and should only do the one thing that is dedicated to that unit. The definition cohesion was already explained in 2.1.1. Therefore the object delegates its work to other units that are dedicated to that well defined and specific task.
There is a third point that "auszeichnet" object orientation. The things that are represented in software not only mapped by representing a lsit of commands, but having concrete representations for the real world. These concrete representations are object that model real world things directly and bundle the behavior and representation of a real world object directly. These "auzeichnungen" of object orientation allow it to separate in different models and make the models reusable. By doing so the "einzelnen" modules are reusable and are simple and easy to use. This led object orientation to the four "saulen" pillars of object orientation: encapsulation, abstraction, inheritance, polymorphism. These pillars are the generally accepted basic advantages of object orientation.

Jeff Bay however enhances the advantages by "core concepts" and "core qualities" that mark good object oriented code. He states that in dann qualities beschreiben

---

OO saves us! –> But ... because
Comparision of procedural versus object oriented programming procedural: step by step,

seldom informaiton hiding, actions mainulate data. Actions are spread all ove rthe programming

oo: "bundle" with capabilities, hides structures, delegates tasks that are not done by the object itself to other objects, objects model real world behaviour, decoupled and separated in different modules. "what" leads to "how": encapsulation, abstraction, inheritance, polymorphism

advantages of oop: modules, reusable maintainable, simplicity (describe each shortly)

Describe each quality, a bit more detailes, but also short and concise:

cohesion

loose coupling

zero duplication

encapsulation

testability

readability

focus

2.3 The Rules and their Background

=== This chapter describes the rules of the Object Calisthenics itself.

=== Reference to introduciton, to make sure the reader know that this is an exercise... The exercise: repeat strict coding standards and stuff from introduction shortly

=== FOR EACH rule/subsection IN rules/subsections: Every rule is described one after the other with the information that is already documented in the commited work of the "research" phase:

- Explain the rule. Use quotes of paper.

- Every chapter gives short explanation, a good and a bad example. These examples are explained shortly.

- Furthermore every chapter describes the software patterns and software principals behind every rule. These are for example design patterns, software principals and best practices.

- Refer to other sources to be able to explain clearly but shortly. Summarize complex patterns instead of explaining every pattern and principal in detail.

- Make sure the principals behind are easy to read: A advanced reader should not be bored by detailed explanation and a beginner reader should be able to understand the main message and idea behind the principal and idea.

- The outcome of every rule is then summarized.

2.3.1 Rule 1: "Use One Level of Indentation per Method"

I am able to add code directly in the text

```
1 Listings
```

in the text. Furthermore it is possible refer to a file:

```
1 package ocanalyzer.rules.noelse;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
```

```
6  import ocanalyzer.rules.general.ValidationHandler;
7
8  import org.eclipse.jdt.core.dom.ASTVisitor;
9  import org.eclipse.jdt.core.dom.IfStatement;
10 import org.eclipse.jdt.core.dom.Statement;
11
12 /**
13  *
14  * This class is used to visit all if {@link Statement}.
15  *
16  * An if {@link Statement} which does have a corresponding else
17  * {@link Statement} is saved and furthermore it is reported to the
       given
18  * {@link ElseValidationHandler}
19  *
20  * @author Fabian Schwarz-Fritz
21  *
22  */
23 public class ElseVisitor extends ASTVisitor {
24
25         private List<Statement> elseStatements;
26         private ValidationHandler validationHandler;
27
28         public ElseVisitor(ValidationHandler validatonHandler) {
29                 this.validationHandler = validatonHandler;
30                 elseStatements = new ArrayList<Statement>();
31         }
32
33         @Override
34         public void endVisit(IfStatement ifStatement) {
35                 if (isSingleElse(ifStatement)) {
36                         Statement elseStatement = ifStatement.
                            getElseStatement();
37                         elseStatements.add(elseStatement);
38                         validationHandler.printInfo(elseStatement);
39                 }
40         }
41
42         private boolean isSingleElse(IfStatement ifStatement) {
43                 return ifStatement.getElseStatement() != null;
44         }
45
46         public List<Statement> getElseStatements() {
47                 return elseStatements;
48         }
49
50 }
```

which is cool!

Furthermore like this class `ElseVisitor` and I also like `SomeOtherClass`. `IfThereIsAStupidLineBreakInTheClassName` - what happens then?

2.3.2 Rule 2: "Don't Use the else Keyword"

asdf

2.3.3 Rule 3: "Wrap All Primitives and Strings"

asdf

2.3.4 Rule 4: "Use Only One Dot per Line"

asdf

2.3.5 Rule 5: "Don't Abbreviate"

asdf

2.3.6 Rule 6: "Keep All Entities Small"

asdf

2.3.7 Rule 7: "Don't Use Any Classes with More Than Two Instance Variables"

asdf

2.3.8 Rule 8: "Use First-Class Collections"

asdf

2.3.9 Rule 9: "Don't Use Any Getters/Setters/Properties"

asdf

2.4 Discussing the Rules

2.4 === This chapter is discussing the rules shortly.

=== Use quotes from Jeff's text: He also gives a short summary of his text in the end

2.4.1 Similarities

=== Are there similarities in the rules? Is it possible to categorize the rules in terms of: - Do they have the same intention? - Are they related to the same "big picture" idea (example: encapsulation or abstraction)

Categorize rules: Are there similarities from perspective of principle? ??? Together with next chapter?

2.4.2 Precedence

=== Make clear that this is my own estimation and not related to Jeff's text. Prioritize the rules.
=== To be determined: How long is this chapter?
// Yada yada yada: Own estimation: what's the most important rule? What do I think? What does the author think? Reason with descriptions and examples given

2.4.3 Conclusion and Outcome

=== Give a short and precise conclusion of the Object Calisthenic: Purpose, exercise, stuff to learn, skill improvement.
=== Also Summarize what most of the rules are about:
RESERACH page 21:
- behaviour and operation oriented
- LoD and loose coupling
- talk to friends
- talk the protocol specified by the object's operation
- Next page: conclusion, no else ,naming,
all in all: Duplication of code and idea ==> "Simple and elegant Abstractions"

3 Tool Support to Validate the Object Calisthenics - Evaluation

3.1 Advantages of Tool Support

=== What is a tool? Why do tools help? What makes tools strong? Why do tools matter for developers? Possible outcome of tool support for the OC's? Already described in the Introduction (chapter 1). Describe this more in detail here if necessary.

3.2 Working Environment

=== Describe AST generally. Say that Eclipse provides types representing the parts of code syntax. This is seen as given.

=== Refer to other references explaining AST.

=== Describe shortly how it is possible to do an AST validation with eclpise.

=== Say that: "Eclipse" terms for AST nodes are quite similar to general terms for java's nodes in ASTs. In this report the standard terms are used. These are not further described in this paper.

–> Example: Describe that "MethodDeclaration" consists of different other nodes representing the declaration of a method. Describe the structure of the child nodes of the node as far necessary. Do not embark on a discussion about what exactly is allowed as method declaration.

=== Therefore: Give reference for questions about "parameter", "type", "class", "expression" or "statement". Say that the validation in the next section is exactly implemented as described. One validation implementation example is explained exemplary in the Prototype chapter.

3.3 Evaluation of Rule Validation

=== Say that the prioritization of the rules (in terms of the rule validation) and the "ranking" is given in the end. The sections of this chapter are "rule specific", even if the next subsections refer to each other.

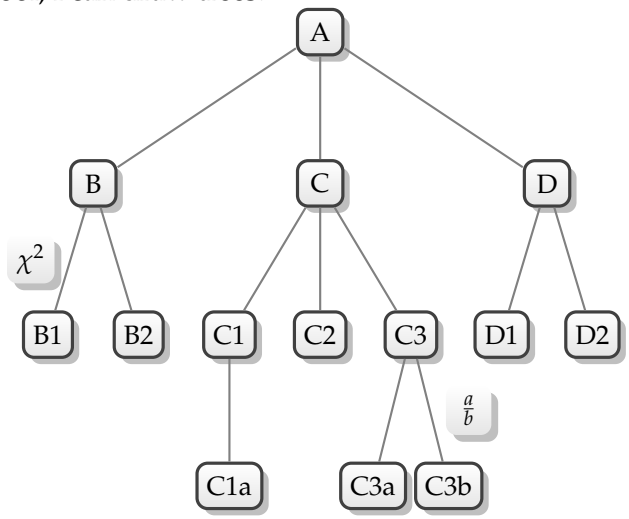
=== FOREACH Rule/Subsection IN Rules/Subsections:

- Similarities found out in description may be similar in this validation? Categorize the rules in groups to form a validation perspective. (E.g.: Validation of rule y is very similar to the validation of rule x that was already explained.)
- Use examples given in description chapter to describe the typical structure of the rule.
- Explain "the positive case": What is the positive structure, satisfying the rule. Where does a rule violation occur when the structure is not fulfilled. (Example: Positive case: maximum of 2 instance variables. If not fulfilled: Discussion Rule violation information occurs on the level of the class or on the level of the third instance variable?
- What checks have to be done for the validation?

- Discussion of 'rule dependencies' within one rule (example: wrapper has to determine possible wrapper classes first before beeing able to indicate the use of a primitive/string in a non-wrapper class...)
- –> solution found/no solution found
- Therefore: Be self-critical: Now, were a "solution" is found (or not), describe the problems that occur with the described implementation
- If there are cases where the rule cannot be validated: Why is it hard to validate. Where does the validation fail? How is it possible to "trick" a good implementation (Example: wrap of primitives can be "tricked" with 'return new Wrapper(instancevariable)'...)

3.3.1 Validation of Rule 1

Cool, I can draw trees:



3.3.2 Validation of Rule 2

asdf

3.3.3 Validation of Rule 3

asdf

3.3.4 Validation of Rule 4

asdf

3.3.5 Validation of Rule 5

asdf

3.3.6 Validation of Rule 6

asdf

3.3.7 Validation of Rule 7

asdf

3.3.8 Validation of Rule 8

asdf

3.3.9 Validation of Rule 9

asdf

3.4 Result of the Evaluation

=== Give a summary on how hard it was to implement the rules. Where does the rule validator fail. Summarize results of the evaluation. Be positive: Do not forget to emphasize the good and working parts of the evaluation tool and accent the positive outcome of them.

3.5 Future Work

=== Be philosophical: What is still to be done in terms of rule validation.

=== If many rules cannot be validated: Future work might be a software where the user can "mark" structures as given. (Example: If it is not possible to determine if a class is a wrapper class, the user could "mark" it as a wrapper class and the validation algorithm might work...)

=== If many rules can be validated: Future work might be the configuration of the rule validator. "pluggable" rules that the user can implement himself? Or "configurable" rule: Use of 2/3/4 instance variables per class.

4 Prototypical Implementation of Tool Support

=== This chapter describes the implementation of the prototype.

4.1 Requirements

=== Describe requirements for the prototype. Depends on how "good" the prototype is....

4.2 Architecture

=== Describe overall architecture.

=== How about a package overview and a short description: What do the classes of the package do and how do they interact in the software?

4.3 Exemplary Rule Validation

=== One example implementation of one rule validation. Idea: Implementation of an ASTVisitor class...

4.4 Resulting Prototype

=== If the protype is really good: swagger and present it as a good software, that might be used in trianings?

=== If not: Be proud of what is there :). This is only a prototype. What could be improved in a better implementation? Is it even possible to do a better implementation in terms of the rule validation (depends on the outcome of chapter 3).

=== Show screenshot and describe UI. What is possible to do with the client

=== Independent form the quality of the prototype: What are ideas that are still out there for the protoype? How could the product improve?

4.5 Outlook and Future Work

=== Possible future work, dependent on what is said in the previous section.

5 Conclusion

=== Conclusion of the result of this work. Do not explain in detail, but refer to the Introduction: What was good, what was bad?

TODO: Have fun writing and stay happy :)

Bibliography

[FoBa03] Foo, John; Bar, Belinda: *Titel : Untertitel*,
Verlagsort: Verlag, Jahr der Auflage. S. 10-20

[Le01] Autor Name: *Titel des Buches*, New York: Penguin Books, 2001.

LITERATUR:

[Bay, 2008] Bay, Jeff: *Object Calisthenics*.
In: ThoughtWorks inc. (eds): *The ThoughtWorks Anthology. Essays on Software Technology and Innovation*.
Raleigh, North California; Dallas, Texas: The Pragmatic Bookshelf, 2008, p. 70–80.

[ThoughtWorks inc., 2008] ThoughtWorks inc. (eds): *The ThoughtWorks Anthology. Essays on Software Technology and Innovation*.
Raleigh, North California; Dallas, Texas: The Pragmatic Bookshelf, 2008.

[Gamma et al., 1994] Gamma, Eric; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software*.
Amsterdam: Addison-Wesley Longman, 1994.

[Martin, 2008] Martin, Robert Cecil: *Clean Code. A Handbook of Agile Software Craftsmanship*.
n.p., Prentice Hall International, 2008.

[Fowler et al., 1999] Fowler, Martin: *Refactoring. Improving the Design of Existing Code*.
Amsterdam: Addison-Wesley Longman, 1999.

[Evans, 2003] Evans, Eric: *Domain-Driven Design. Tackling Complexity in the Heart of Software*.
Amsterdam: Addison-Wesley Longman, 2003.

asdfklsajkflj INTERNET

[Eclipse Documentation] The Eclipse Foundation: *Eclipse documentation - Eclipse Kepler*.
URL http://help.eclipse.org/kepler/index.jsp.

[Wikipedia] Wikipedia. The Free Encyclopedia.
URL http://www.wikipedia.org.

[About.com] Haas, Juergen: *Modular Programming*.
URL http://linux.about.com/cs/linux101/g/modularprogramm.htm.

[ToughtWorks inc.] Thoughtworks inc.
URL thoughtworks.com/about-us (17 November 2013).