VFHVWS

//vfhvws.oncampus.de

Stand 29.01.2018 20:41

Inhalt

VFHVWS ... 9
1 UML - Unified Modeling Language ... 11
1.1 Overview and Learning Objectives UML ... 11
1.2 Unified Modeling Language ... 12
1.2.1 UML Tools ... 13
1.2.2 Diagram Overview ... 14
1.3 History of UML ... 16
1.3.1 UML Structure ... 17
1.3.2 How to use UML? ... 18
1.3.3 Design Flow ... 20
1.4 UML Evaluation ... 22
1.5 Statische Strukturdiagramme ... 22
1.5.1 Verteilungsdiagramm ... 23
1.5.2 Komponentendiagramm ... 25
1.5.3 Paketdiagramm ... 27
1.5.4 Klassendiagramm ... 29
1.5.4.1 Notation ... 31
1.5.4.2 Interfaces ... 32
1.5.4.3 Stereotypen ... 33
1.5.4.4 Assoziation ... 34
1.5.4.5 Vererbung / Generalisierung / Spezialisierung ... 37
1.5.5 Kompositionsstrukturdiagramm ... 37
1.5.6 Objektdiagramm ... 38
1.6 Dynamische Diagramme ... 38
1.6.1 Anwendungsfalldiagramm ... 39
1.6.2 Aktivitätsdiagramm ... 41
1.6.3 Zustandsdiagramm ... 43
1.6.4 Sequenzdiagramm ... 45
1.6.5 Interaktionsdiagramm ... 47
1.6.6 Kommunikationsdiagramm ... 47
1.6.7 Zeitdiagramm ... 48
1.7 Zusammenfassung UML ... 49
1.8 Wissensüberprüfung UML ... 50
1.9 Literatur UML ... 50
2 AJV - Advanced Java ... 52
2.1 Überblick und Lernziele AJV ... 52
2.2 Regular Expressions ... 53

2.2.1 Zeichenliterale ... 55
2.2.2 Metazeichen ... 56
2.2.3 Gruppen und Klassen ... 58
2.2.4 Lookaround ... 58
2.2.5 Tabellarische Zusammenfassung ... 59
2.2.6 Reguläre Ausdrücke Testen ... 61
2.2.7 Fazit ... 62
2.3 Performance Tuning ... 62
2.4 Debugging and Testing ... 62
2.5 Guava ... 62
2.6 JavaFX ... 67
2.7 Zusammenfassung AJV ... 67
3 MET - Software- und Architekturmetriken ... 69
3.1 Überblick und Lernziele MET ... 69
3.2 Literatur MET ... 70
3.3 Motivation ... 71
3.4 Einfache Metriken ... 72
3.4.1 McCabe Metrik ... 75
3.5 Halstead Metrik ... 77
3.6 Code Coverage / Test Coverage ... 78
3.7 Regelverletzungen ... 81
3.8 JDepend ... 87
3.8.1 JDepend-Metriken MET ... 88
3.8.2 Zyklenanalyse als Teil des Buildzyklus ... 90
3.9 Kopplungsmetriken ... 94
3.9.1 Coupling / Kopplung ... 95
3.9.2 Die ACD Metrik ... 96
3.9.3 Zusammenfassend ... 100
3.10 Technical Debt MET ... 101
3.11 Werkzeuge ... 103
3.12 Zusammenfassung MET ... 108
3.13 Wissensüberprüfung MET ... 109
4 CCD - Clean Code Development ... 112
4.1 Überblick und Lernziele CCD ... 112
4.2 Einleitung zu Clean Code Development ... 114
4.3 Fundamentals ... 117
4.4 Coding Basics ... 118
4.4.1 Coding Source ... 123

4.4.2 Coding Conditionals ... 124
4.4.3 Coding Principles ... 126
4.5 Code Quality ... 131
4.5.1 Source Code Konventionen ... 132
4.5.2 Automatisierte Unit Tests ... 133
4.5.3 Code Coverage Analyse ... 134
4.5.4 Zuerst Testen ... 137
4.5.5 Mockups ... 138
4.5.6 Funktionale Techniken ... 139
4.5.7 Lange Funktionen aufteilen und Reviews ... 139
4.5.8 Messen von Fehlern ... 141
4.5.9 Statische Codeanalyse ... 143
4.5.10 Der Technical Debt Deines Produktes ... 143
4.6 Architecture und Class Design ... 146
4.7 Packages ... 148
4.8 Produktivität ... 152
4.9 Management ... 154
4.10 Zusammenfassung CCD ... 159
4.11 Wissenüberprüfung CCD ... 160
4.12 Literatur CCD ... 160
5 BUI - Advanced Buildmanagement ... 162
5.1 Überblick und Lernziele BUI ... 162
5.2 Prinzipien des Buildmanagements ... 164
5.2.1 Buildmanagement in den 80er/90er Jahren ... 164
5.2.2 Buildmanagement im XML/ANT Zeitalter ... 166
5.2.3 Prinzipien: Convention, Dependencies und Plug-In ... 168
5.2.4 Prinzip: Top DSL ... 169
5.2.5 Weitere Buildmanagementsysteme ... 172
5.3 Praktische Übung mit Maven ... 174
5.4 Maven Schnellreferenz ... 177
5.5 Praktische Übung mit Gradle ... 177
5.6 Gradle Schnellreferenz ... 178
5.7 Vor- und Nachteile ... 184
5.8 Zusammenfassung BUI ... 185
5.9 Wissensüberprüfung BUI ... 185
6 DVC – Distributed Version Control Systems Vertiefung ... 187
6.1 Überblick und Lernziele DVC ... 187
6.2 Einführung und Literatur DVC ... 188

6.3 Einstieg in Git ... 190
6.4 Grundlegende Arbeit mit Git ... 193
6.5 Git Kommandos en Detail ... 196
6.5.1 Projekt anlegen ... 196
6.5.2 Status und diff ... 197
6.5.3 Add und commit ... 200
6.5.4 Push und pull ... 202
6.5.5 Remove und move ... 203
6.5.6 Mehr diff Kommandos ... 204
6.5.7 Log ... 206
6.5.8 Branch und merge ... 209
6.6 Zeitreisen in Git ... 209
6.7 Git Glossary ... 211
6.8 Mercurial ... 212
6.9 Kommandoübersicht Git ... 214
6.10 Zusammenfassung DVC ... 215
7 CID - Continous Integration / Continous Delivery ... 216
7.1 Überblick und Lernziele CID ... 216
7.2 Einführung CID ... 217
7.3 Deployment Pipeline und Continous Integration ... 218
7.3.1 Typische Fehler im Deployment-Prozess ... 220
7.3.2 Feedback von Entwicklern ... 221
7.3.3 Saubere Konfigurationsverwaltung ... 222
7.4 Konzept des Continous Integration ... 223
7.4.1 Testen im Continous Integration ... 224
7.4.2 Prozesskette des Continous Integration ... 226
7.5 Praktische Übung: Jenkins einrichten ... 229
7.5.1 Praktische Übung: Projekt erstellen ... 231
7.5.2 Den CI Prozess stabilisieren ... 232
7.6 Travis-CI ... 233
7.7 Zusammenfassung CID ... 234
7.8 Wissensüberprüfung CID ... 235
7.9 Literatur CID ... 236
8 AOP - Aspect Oriented Programming ... 237
8.1 Überblick und Lernziele AOP ... 237
8.2 Das Problem der Komplexität von Methoden ... 238
8.3 Praktische Übung mit AspectJ ... 241
8.4 Praktische Übung mit AspectWerkz ... 246

8.5 Begriffe der AOP ... 249
8.6 Besonderheiten der AOP ... 250
8.7 Zusammenfassung AOP ... 253
8.8 Wissensüberprüfung AOP ... 254
8.9 Literatur zu AOP ... 254
9 FPR - Funktionale Programmierung ... 255
9.1 Überblick und Lernziele FPR ... 255
9.2 Paradigmenvergleich ... 257
9.3 Funktionale Paradigmen in Clojure ... 260
9.4 Funktional: Clojure ... 265
9.4.1 Basics ... 269
9.4.2 Basistypen ... 271
9.4.3 Collections ... 274
9.4.4 Functions&Bindings ... 278
9.4.5 Destructuring ... 280
9.4.6 Namespaces ... 283
9.4.7 Program-Flow ... 284
9.4.8 Java Interoperabilität ... 287
9.4.9 Metadata ... 288
9.4.10 I/O ... 289
9.4.11 State-Management ... 289
9.4.12 Types and Protocols ... 291
9.4.13 Multimethods ... 291
9.4.14 Macros ... 291
9.4.15 API ... 291
9.4.16 MISC ... 293
9.4.17 Weiterführendes ... 296
9.4.18 Stdlib ... 298
9.5 Hybridsprache: Scala ... 298
9.6 Kurzübersicht: Haskell & Erlang & Co ... 305
9.7 Zusammenfassung FPR ... 305
9.8 Wissenüberprüfung FPR ... 306
9.9 Literatur FPR ... 306
10 LPR - Logische Programmierung ... 308
10.1 Überblick und Lernziele LPR ... 308
10.2 Logik ... 309
10.3 Prolog ... 311
10.4 Core.logic ... 315

10.4.1 Einrichten der Umgebung ... 317
10.4.2 Unification, conde und Disunification ... 318
10.4.3 Fresh, membero und distincto ... 320
10.4.4 Everyg, lvar, conso und resto ... 322
10.4.5 Beispiele ... 323
10.5 Zusammenfassung LPR ... 323
11 DSL - Externe und interne DSLs ... 324
11.1 Überblick und Lernziele DSL ... 324
11.2 Einführung in DSLs ... 325
11.3 DSLs tiefer betrachtet ... 327
11.4 Die erste DLS: Expression Builder ... 331
11.5 Fluent Interfaces ... 334
11.6 Mächtigere Sprachen ... 337
11.7 Formen interner DSLs ... 340
11.7.1 Embedded DSLs ... 341
11.7.2 Generative DSLs ... 342
11.8 Formen externer DSLs ... 344
11.8.1 String Manipulation ... 345
11.8.2 Formate wie XML, JSON Transformieren ... 346
11.8.3 DSL Workbench ... 346
11.8.4 Parser Generatoren ... 348
11.8.5 Parser Combinators ... 349
11.9 Wann welche DSL? ... 350
11.10 DSLs mit Xtext ... 351
11.10.1 Xtext Grundlagen ... 352
11.10.2 Beispiel mit Xtext ... 354
11.10.3 Eine eigene DSL ... 357
11.11 DSLs mit ANTLR ... 360
11.11.1 Einfaches ANTLR Beispiel ... 361
11.12 Zusammenfassung DSL ... 364
11.13 Wissensüberprüfung DSL ... 365
11.14 Aufgaben ... 365
11.15 DSLs Literatur ... 365

Anhang

I Literaturverzeichnis ... 367
II Abbildungsverzeichnis ... 368
III Tabellenverzeichnis ... 376

IV Aufgabenverzeichnis 377
V Glossar 379

VFHVWS

Online Studienmodul

Verfahren und Werkzeuge moderner Softwareentwicklung

Lerneinheiten

VFHVWS
1 UML - Unified Modeling Language
2 AJV - Advanced Java
3 MET - Software- und Architekturmetriken
4 CCD - Clean Code Development
5 BUI - Advanced Buildmanagement
6 DVC – Distributed Version Control Systems Vertiefung
7 CID - Continous Integration / Continous Delivery
8 AOP - Aspect Oriented Programming
9 FPR - Funktionale Programmierung
10 LPR - Logische Programmierung
11 DSL - Externe und interne DSLs

Über den Autor

Prof. Dr. Stefan Edlich

Hochschule:

Beuth Hochschule für Technik Berlin

Tätigkeitsbereich:

NoSQL - Big Data / Data Science - Software Engineering

Kontakt:

FB VI / Labor Online Learning

Luxemburger Str. 10

13353 Berlin

1 UML - Unified Modeling Language

Autor: Prof. Dr. S. Edlich

1 UML - Unified Modeling Language
1.1 Overview and Learning Objectives UML
1.2 Unified Modeling Language
1.3 History of UML
1.4 UML Evaluation
1.5 Statische Strukturdiagramme
1.6 Dynamische Diagramme
1.7 Zusammenfassung UML
1.8 Wissensüberprüfung UML
1.9 Literatur UML

1.1 Overview and Learning Objectives UML

Some of you may have already had this learning unit in your bachelor's degree (e. g. in software engineering). This session is intended for you as a repetition and introduction to the UML task you have to submit. For all others who have not yet got to know UML during their studies, this course is an important basis which you will need, among other things, to solve the UML submission task.

Lernziele

Learning Goals The aim of this session is to familiarize you with UML. In addition to the introduction, static structure diagrams and dynamic diagrams (behavioral diagrams) are presented.

You should be able to judge the appropriate use of UML and apply it to your own project. The various elements of UML are to be used and the advantages and disadvantages of this representation form are to be recognized and described. The critical use of this industrial language should definitely be mastered.

As a result of the practical exercise, you will be able to assess which UML diagrams you use in which order to achieve your modeling goal.

Gliederung

Outline

- Introduction
- Background and history of UML
- Presentation of static structure diagrams
- Presentation of dynamic diagrams
- Knowledge check and exercises

Zeitumfang

Time Requirements
To work through the learning unit, you will need approx. 120 minutes and for the processing of the exercises approx. 170 minutes.

1.2 Unified Modeling Language

Definition

Unified Modeling Language
The Unified Modeling Language is a set of notation elements that can be used to develop models for software systems. This concerns the analysis, design and in general the presentation and documentation of the software elements or the software behavior.

The UML was developed by the OMG (http://omg.org) and is specified in more than one hundred pages. It has established itself as the most important notation / language for the specification of software systems. One of the most important tasks is to make the design and the associated architectures transparent for other project participants or outsiders. It is important to note that UML does not have to be used dogmatically.

The aim of UML diagrams or descriptions is essentially that other people understand the statement. There is, therefore, no body that decides on the correct and wrong UML. UML itself can be interpreted in many places.

It is certainly better to put energy into good design than 100% correct UML. The use of representations and methods that are not UML are important for this. For example, decision tables are often required for situations. These are not part of the UML, but they are extremely helpful.

Experiment with UML as well as with Non-UML and talk to people who have already had experience in the field. For example, many architects believe that a good design session on a whiteboard without strict UML can do more than just a lot of UML diagrams on paper.

The OMG also manages other interesting standards such as CWM, MOF and the important UML exchange format XMI.

1.2.1 UML Tools

In order to be able to test everything you need a UML tool.

More than one hundred UML tools can be found at http://www.jeckle.de/umltools.htm The German company OOSE also has a good tool page. It will take some time to find' your' tool. Allow time for this! Often, some settings get jammed, and the tools don't show what you want or even generate incorrect Program Code.

Here are just a few tips on which tools have proven themselves here and there:

- **Together** www.borland.com - Flexible and powerful tool
- **Omondo** www.ejb3.org - Good Eclipse integration. Might now have another name
- **Rational Rose** www.ibm.com - Not free of charge, powerful, integrated into the rational Toolsuite, has the reputation to be a bit clumsy.
- **Magic Draw** www.nomagic.com - Feed Required
- **Argo UML** argouml.tigris.org
- **BlueJ** www.blueJ.org - Has emerged as a teaching tool, is closely interwoven with development and can only be used for class diagrams.

- **Fujaba** www.fujaba.de - A well known free tool of the Uni-Paderborn
- **Astah** www.astah.net - From Japan
- **ObjectIF** www.microtool.de from Microtool from Berlin!
- **Innovator** www.mid.de - different versions for different views (Db, architect, developer etc.)
- **Enterprise Architect** www.sparxsystems.com - Very powerful and comparatively inexpensive modeling tool for UML and other languages.

More Recommendations:

- http://www.topcased.org
- http://www.umlet.com

But the Master is:

- http://plantuml.com UML as a textual description! Versionable!
- http://planttext.com for live demos and live generation

1.2.2 Diagram Overview

Structure diagrams and architecture diagrams are also referred to as static diagrams. Behavior and interaction diagrams are often referred to as dynamic diagrams.

Package Diagram	Shows the concrete structure of the software in namespaces or packages with their dependencies.
Class Diagram	Represents the classes of the respective programming language and their relationships.
Object Diagram	Shows objects - i. e. in which state the classes can occur.

Structure Diagrams

Deployment Diagram	Shows where the (mostly hardware) systems and their components are installed in their parts on computer nodes.

Component Diagram	Displays the component structure with the relationships among each other.
Composition Structure Diagram	Shows the components with their specific ex- and imported interfaces.
Subsystem Diagram	Architectural correlations of components.

Architecture Diagrams

Use-Case-Diagram	Presents the actors and their cases of application (quasi their desires).
Activity Diagramm	Shows how the program works by using actions, transitions and branches.
State Diagram	Object states are displayed here. There are transitions between the states. An automat can specify the sequence of transitions even more precisely.

Behavior Diagrams

Sequence - Diagram	Represents how classes or components interact / communicate by means of messages.
Interaction Overview	A combination of activity and sequence diagram.
Communication Diagram	Arranges the exchanged messages graphically to display them appropriately.
Timing- Diagram	Describes the chronological sequence of the object states (often used in telecommunications and control engineering).

Interaction Diagrams

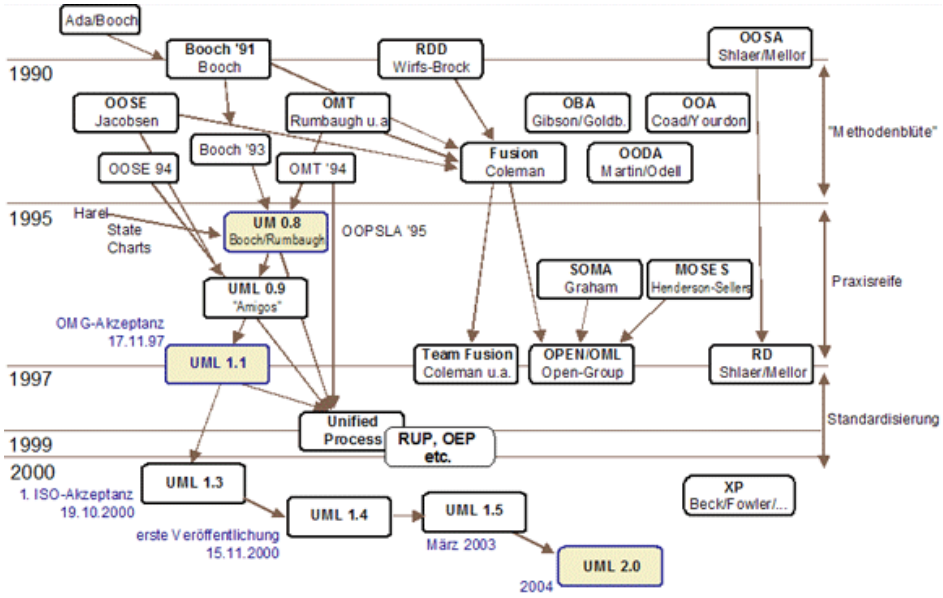
1.3 History of UML

It was not until the 1990s that the work on the subject of object-oriented analysis / design gained momentum. Authors who made a name for themselves here early on were Adele Goldberg, Grady Booch, Peter Coad, Edward Yourdon, James Rumbaugh, Bertrand Meyer and Ivar Jacobsen.

Early on, the methods of James Rumbaugh and Grady Booch became established. However, both competed.
Rumbaughs
method seemed to be better for the analysis and
Boochs
method was probably more advantageous in design. James Rumbaugh was also hired by the Rational Software Cooperation in 1994, where he was also responsible for the further development of the IBM Toolsuite.

United with the forces of Ivar Jacobson,
Rumbaugh
and
Booch
then tried to bridge their differences and create a unified language. This, of course, is also due to pressure from the industry. Ultimately, UML-based tools and process models made it possible to earn money.

Ivar Jacobsens
Objectory was then bought by Rational and the Unified Method (UM) was born from the unification of the methods.



History of the UML

Booch, Rumbaugh
and Jacobsens the three Amigos were called from now on. From the time of their former violent battles for methods, the discussion process and the difficult negotiations, a self-critical statement of
Jacobsens
is known, which is gladly quoted over and over again:

However, the pressure to develop a non-proprietary language based on UM has increased: The Unified Modeling Language (UML). An international consortium developed the UML in 1996 and then handed it over to OMG. This is how the UML 1.1 was created in 1997. It developed in steps to version 2.0, which was the last downloadable version until 2006. At the moment version 2.5 from June 2015 is available in the download area of the OMG.

1.3.1 UML Structure

The latest UML specification can be downloaded from the OMG website (http://www.uml.org) - older versions are also available. It is worthwhile to have a look around on the UML websites. The specification is divided into the following parts:

Vertiefung

UML 2.0

- UML 2.0 Infrastructure Specification: The basic elements of UML are presented here, which are a prerequisite for the following specifications. Elements such as class, association or multiplicity are introduced.
- UML UML 2.0 Superstructure Specification: Here, the actual known elements of UML are defined and presented with use cases and deeper concepts.
- UML 2.0 Object Constraint Language: The last document specifies boundary conditions for some types of graphs. It is used, for example, for class diagrams (chapter 1.5.4) or sequence diagrams (chap. 1.6.4) important. You can specify restrictions with invariants, pre-/post-conditions or guards. For example, it is very helpful to specify preconditions and post-conditions for methods. Or to note down boundary conditions for fields (e. g. never be < 0). These boundary conditions are then also important for the automatic further processing of UML, for example.
- UML 2.0 Diagram Interchange: A fourth part specifies the layout of the diagrams. With this specification, the layout should no longer be lost with the exchange of diagrams across tools, since each diagram is described by means of a graph (node, edges, sheets).

1.3.2 How to use UML?

UML may be a nice notation, but many people will not be able to see which areas of application there are at all.

Communication and Abstraction

The real purpose of UML is to communicate ideas! The question is always, how can you best and fastest bring the design of a system across and document it. It is clear that UML may not always be the best tool and may not be complete.

Therefore, there are many critics who say that UML is too big and too complex. UML tries to please everyone. Nevertheless, UML has established itself against all its previous notations. It is the language that has become established in the industry and which you should definitely master, but still take a critical look at.

Another important benefit of UML is that it allows you to hide details and refine them later with or without UML. In many cases, you have to approach the system Top-Down and specify specific parts or components from the vision / analysis.

How do they succeed? Here UML diagrams can help to define the rough structure/ requirements and then to define them in finer diagrams. Often, however, you simply have to talk to other designers, analysts, stakeholders, project managers or clients who sometimes have no idea about IT. How do you present the system to them? Although many of the people in the community do not have comprehensive IT knowledge, most of them are familiar with UML and the meanings of some diagrams. You will soon be able to explain the meaning of a use case or deployment diagram well enough to convey the basic structure of your system.

Reverse Engineering

One of the possibilities for using UML is Reverse-Engineering. You may already have a system that needs to be further developed and you want to get an overview of the system. Good tools can analyze the existing code and generate various UML diagrams from it. It can be very helpful to get out of a mountain of code, first of all, some diagrams (like. B. class, package or component diagrams) and thus to understand the program' meta'.

Reverse engineering is a not to be underestimated approach in the industry when it comes to achieving competitive advantages. A well-known example of reverse engineering is the Lotus 1-2-3 spreadsheet, the program has been' decomposed',' understood' by Microsoft and rebuilt as MS Excel. This is also reverse engineering.

Sketch- / Blueprint-Mode

In addition to the written documentation of an architecture by architects, UML is often used more informally as a sketch in team discussions. In sketch mode, informal UML is used and the UML diagrams shown mostly show only the interesting sections of the system that is being discussed.

In the first mentioned Blueprint mode, the architect of the system is usually busy with more or less completely depositing the design of the software (blueprint = blueprint; a term from the architecture).

Sketch and blueprint mode are not synonymous to use. Sketch is a small discussion about clippings. Blueprint is a larger architectural specification.

Programming language / MDA

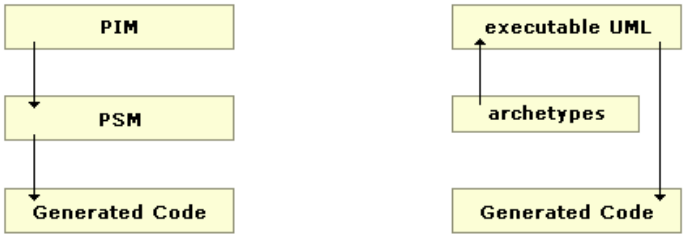
An important element of UML is that it can be processed by machines. The UML and the associated languages such as XMI (XML Metadata Interchange) are read by Frameworks, which can also generate code.

The development of software is relatively expensive, especially European IT companies need to see how you can be competitive and fast, qualitative (!) high quality software.

This can be done by automating the generation of code. If UML diagrams describe the components, packages, classes and also the behavior of software, why not generate as much code as possible? In this case, the UML itself would be something like a higher quality meta programming language.

This approach leads us to the research area of the MDA (Model Driven Architecture), for which there are already an astonishing number of powerful frameworks. Unfortunately, many of these toolkits are hidden in the industry. There are also some good free toolkits available. B. under AndroMDA (http://www.andromda.org/docs/index.html).

MDA is part of the module "Model-Based Software Design" and is treated more intensively there.



Model Driven Architecture (MDA)

The most important branch is shown on the left. You model in UML. You can create a model - the Platform Independent Model (PIM) - which is more concrete, but still independent of the underlying technology.

In the next step, you would generate a PSM (Platform Specific Model) (all with MDA tools) that contains concrete and executable code for a platform. Z. B. Code for a. NET or J2EE platform. This means that there are indeed frameworks that can generate executable code. Sometimes even executable applications that work a lot with templates or prefabricated building blocks.

Another approach is executable UML. The intermediate step of the PSM to be created is skipped and the code is generated immediately afterwards. Archetypes help to describe how the encoding step from UML to code should be done.

1.3.3 Design Flow

When dealing with design, the initial difficulty is often knowing what the concrete procedure looks like. It is important to clarify which diagram should be selected first and why.

Which order is best?

Of course, this question is difficult to answer and depends 100% on the system to be designed. Nevertheless, here is a suggestion based on project experience and considering the abstraction level of the diagrams:

Beispiel

Abstraktionsgrad von Diagrammen

1. Use Case D
   The use cases are certainly among the most abstract. What the user wants from the system should usually be known first and should be part of the analysis rather than the design.

1. Deployment D
   As here is shown, where which coarse component is running, this is also very abstract. This is about physical locations for large projects. So server X here, many clients there, a billing server there, etc. This is abstract and usually belongs far forward.

1. Package D
   Packages usually contain components that consist mainly of a collection of classes. Therefore, component diagrams must be created before class diagrams.

1. Class D
   For this reason, class diagrams should be created according to packages and component diagrams.

1. Composite-Structure D, Subsystem D, Object D
   The diagrams mentioned here are not so often used and are often scattered throughout the entire design phase.

1. Activity D
   The activities of an application are usually specified much earlier in this order to clarify the flow structure of the application. This can often be the prerequisite for the designer to get the components clear. Sometimes even before the deployment diagram but after the use cases. For the sake of clarity, it is located here according to the static diagrams in the dynamic diagrams.

1. Sequence D
   They show the behavior of components or classes. Generally, components or classes must exist before the sequence diagram can be modeled.

1. State D
   The state of the application, components or classes can also be modeled at any time during the design phase. Usually, after the use cases, deployment and activity diagrams have been created."
1. [Interaction-Overview D, Timing D]
   These not so often used diagrams can also be used everywhere if required.

So it would be ideal now if you take your own motivating (private) project and model it and accompany it over a longer period of time. Perhaps the above-mentioned orientation in the design process and the explanations of the concrete diagrams in Chapters 1.4 and 1.6 may be helpful.

1.4 UML Evaluation

The meaning of UML is much more controversial in reality than it seems.

In the academic field, UML is taught in many places, as there are still many proponents on the university side. It is not uncommon for industry to accept this uncritically.

Nevertheless, there are companies and movements that rate the usefulness of UML much less. Especially in the agile environment, other techniques are therefore often used. This can be seen in the meanwhile quite "old" XP model, which does not include extensive UML artifacts. In some major companies, such as e. g. Google is even banned UML!

This does not mean that UML is completely prohibited. If two developers want to "talk" to UML, this is no problem. However, the power of a prototype or prototypical architecture in corporate culture is seen as much more efficient than a UML paper.

It is therefore crucial for every software engineer to know the spectrum of tools or communication tools and to be able to judge for himself to what extent UML is a suitable tool or other methods are better suited.

1.5 Statische Strukturdiagramme

Title: Static Diagrams

This section presents the static structure diagrams:

- **Deployment Diagram**: How are the software components physically distributed?
- **Component Diagram**: What units / components does the software system consist of?

- **Package Diagram**: Which concrete packages / namespaces are used to hierarchize the components?
- **Class Diagram**: Which classes are there (in a namespace)?
- **Composition Structure Diagram**: How are components related?
- **Object Diagram**: Which objects exist at a certain point in time?

1.5.1 Verteilungsdiagramm

Title: Deployment Diagram

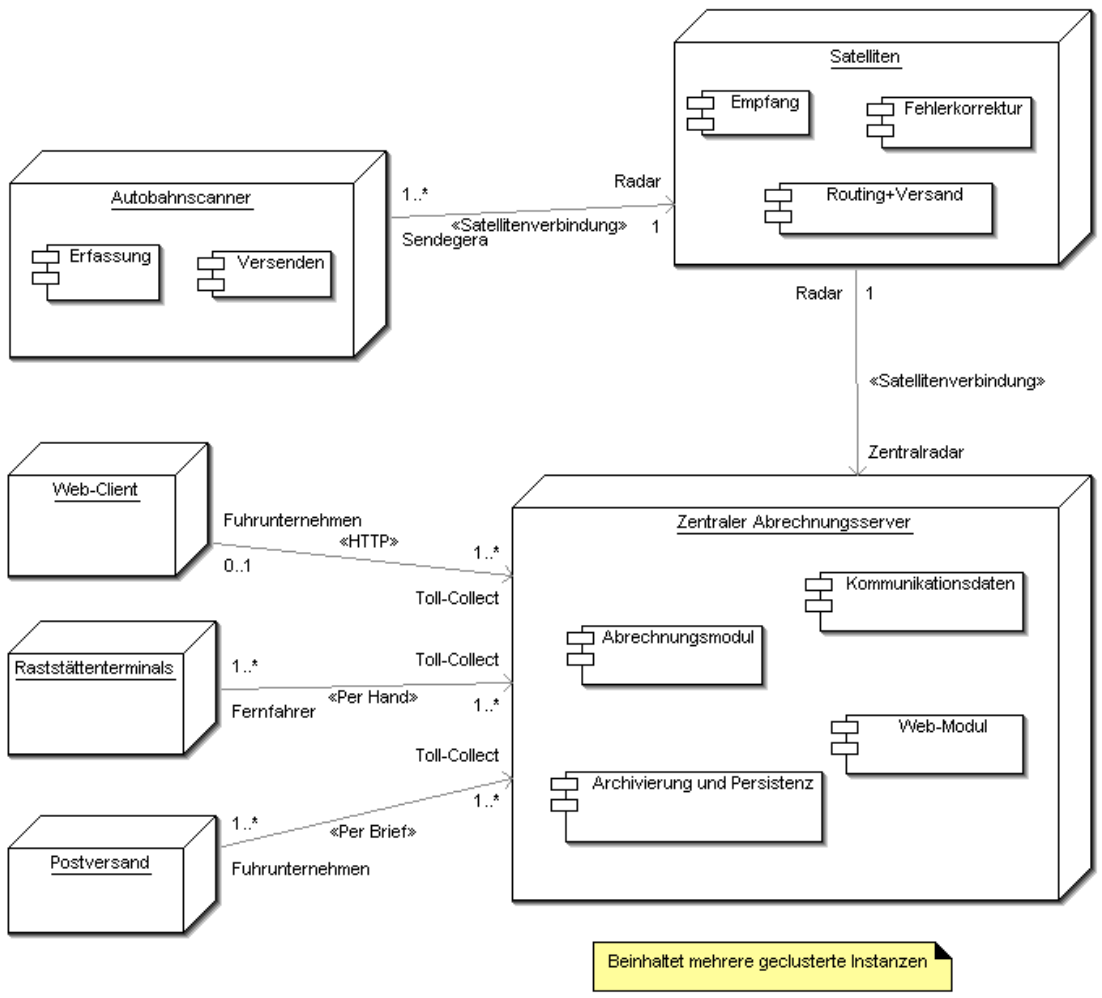
The distribution diagram Deployment Diagram) is specified relatively early. Relatively soon after the Use Cases. This diagram is about showing the physical structure of the entire application. This means which components are running on which physical devices and where.

This is often difficult for students to understand, because you usually only write small projects that run on one computer.

For example, the TollCollect project includes various data centers, satellites, equipment on motorways and equipment in motorway service stations. So it runs very different software on very different computers in many places.

The aim of the distribution diagram is to present the physical distribution conclusively.

Beispiel



Verteilungsdiagramm des Systems Toll-Collec

We see here an example of the German toll system Toll-Collect. Trucks are registered at the motorways. This data is then transmitted to the control center by satellite. The components in the central system show which tasks the components perform. For example, users can either go to the service stations for payment or pay by web or by mail. This example is intended to help you understand the physical distribution of such large systems as the toll system.

Notation

The individual physical locations/nodes (if necessary Servers) are represented by boxes/cuboids. The boxes are connected by simple lines and represent the communication between them. Components or artifacts can be displayed in the boxes. For example, artifacts can be files of different formats such as: jar, dll, war.

The connections of the nodes describe the type of communication connection. You can also specify multiplicities. For example, you can specify that up to 1000 clients can connect to the server at the same time.

Hinweis

Hint

1. For practice, it is advisable to simply model a larger system than what you later want to build or specify more precisely. This results in a more extensive distribution diagram. Or plan your application as a more complex distributed system.
1. Imagine the system administrator comes to you - he knows only that there is work coming to him - and wants to know which hardware he has to provide. The distribution diagram helps him to do this. With the additional components in it, he also has the possibility to understand what the system should do roughly and can also estimate the costs.

1.5.2 Komponentendiagramm

Title: Component Diagram

The Component diagram is specified relatively early. Relatively soon after the use cases. This diagram is about showing the physical structure of the entire application. This means which components are running on which physical devices and where.

This is often difficult for students to understand, because you usually only write small projects that run on one computer.

For example, the TollCollect project includes various data centers, satellites, equipment on motorways and equipment in motorway service stations. So it runs very different software on very different computers in many places.

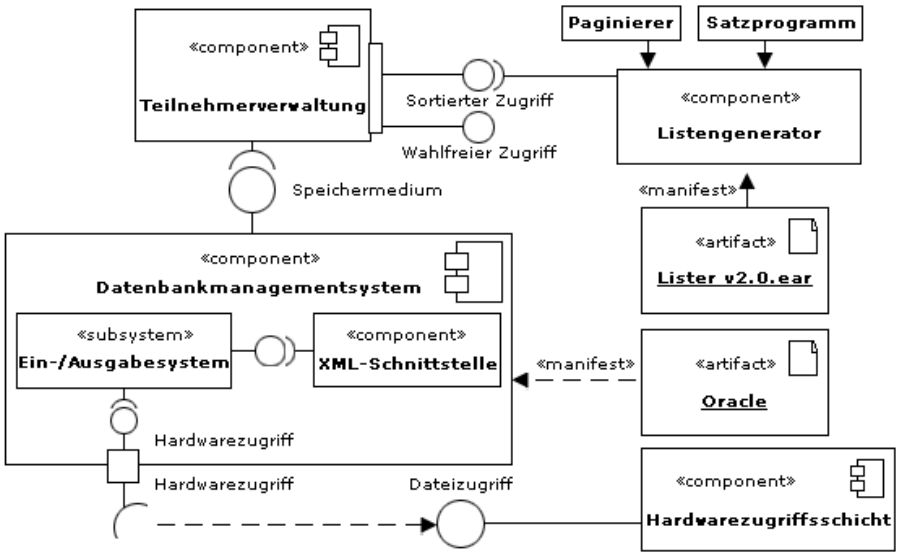
The aim of the distribution diagram is to present the physical distribution conclusively.

Bernd Österreich
writes: *"A component can be instantiated similar to a class and encapsulates complex behavior. It is used to form units with a high degree of professional coherence. In contrast to a class, the principle of interchangeability (substitutes) is also aimed at for a component. A technical platform for components, for example, is Enterprise Java Beans (EJB)."*
[Oes98]

Notation

The component is a rectangle with the typical' Lego symbol' in the upper right corner. It contains the keyword
<<component>>
, that is often not shown for reasons of clarity.

It is allowed to insert other elements - such as objects or components - into the object, but this can easily lead to very complex diagrams (packable element).



Beispiel Komponentendiagramm

What does this diagram show?

- Some Components Komponenten (with the Lego-Symbol)
- On the right some artifacts. This can be paper or software.
- Interesting are the balls, into which the semicircle reaches. The ball is the provided interface. The semicircle is the required interface.
- Showing small squares there are still 'ports' that allow access to the component, which is not based on interfaces.

In diagrams, the keywords from different UML variations still haunt the keywords
<<realize>>
or
<<reside>>
(from which class this component is implemented) or
<<implements>>

. The latter describes the concrete interface that this component also implements.

Anmerkung

Again, one can imagine that this diagram is not only used for internal communication, but also for dialogue with developers and clients.

A few examples:

- The client wants to know how the new components interact with the old ones.
- The customer wants to know and then decide for himself which components (or modules) he can buy when and how they fit together.
- The junior developer wants to know in which components the z. B. Dismantle the full-text indexing system. He asks how that works, how to build it. You draw a diagram with components such as parsers, configuration of the search engine, configuration of the external modules, activation of the engine, handling and preparation of the results, etc. Result: The junior developer understands and has a vision of the structure.

Often Design Patterns as MVC are reflected here.

Wichtig

In this context, it is important to note that in UML there is also the stereotype Subsystemwhich is often used in component diagrams. Here, only a container for other components is built up - represented as a simple box with the stereotype Subsystem.

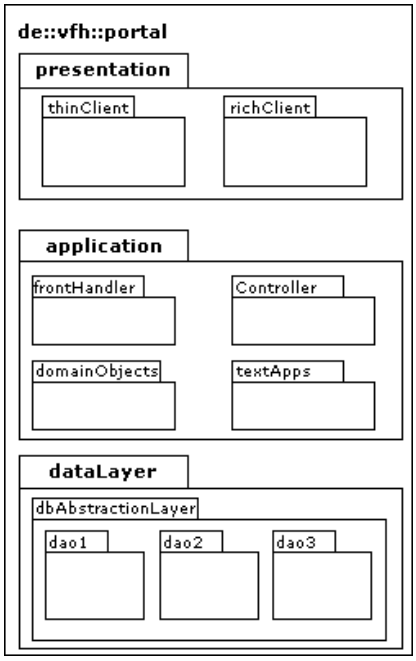
1.5.3 Paketdiagramm

Title: Package Diagram

The package diagram structures the namespace of packages, components, or classes.

It usually corresponds with the packages in Java or with the namespaces in C#. For example, de.vfh.portal.database which includes the class AccessLayer. It is important to have a hierarchical structure that allows the unique identification of the model element.

Beispiel



Paketdiagramm

In the example image you can see that all drawn packages should be placed under the de::vfh::portal namespace.

At the top you can see a presentation package. It contains a package for a rich client, such as a Swing GUI and a thin client, such as a JSP application.

In the middle you can see the application package, which contains four sub-packages. A front-handler for communication with the GUIs, a controller for flow control, the domain objects (i. e. the business objects) and a package for management with external applications.

Below you can see different layers for the database connection.

The package diagram shows the hierarchical structure of the model elements, showing layers and component blocks.

In the notation there are still package merges, which are not explained here and are not very important in practice. Try to find something yourself.

Anmerkung

Why are package diagrams so important?

- In practice, package diagrams are important because the developers want to ask for a "playground" early on and develop test code.
- Packages can reflect the real architecture of the system like component diagrams. Design patterns and layers in the application become transparent here. It is important that transparency is provided at an early stage, that there is sufficient discussion about it from the outset or that experienced designers are employed.

Package diagrams are most important because they allow you to model dependencies. So try to draw the package and ask yourself: Who is using whom? Use arrows to draw this into the diagram. This is shown in the' Usage Structure'. This usage structure is extremely important for good design and architecture.

Is there a completely confused structure here, who is using whom? Or is there a clear flow from top to bottom? The latter is very important for a changeable architecture.

For example, in a package diagram that contains 'uses' relationships, you can also see very nicely whether there are cycles in use. So package A uses package B and vice versa? If this is the case, a redesign is often necessary.

In the last chapter, we see that there are many tools that automatically analyze the usage relation between packages and point out problems.

But experience this for yourself by designing a package diagram in such a way that there is a hierarchical usage relationship in addition to the hierarchical namespace. Your code becomes more maintainable and insensitive to change.

1.5.4 Klassendiagramm

Title: Class Diagram

A class diagram visualizes classes in different ways. For this purpose, the classes themselves are displayed once, if necessary their relations with each other and possibly with each other. also their internal structure. Usually there are two goals:

Representation of classes and relationships: Here different classes of a package or component are displayed in the diagram. It is important to show which classes exist at all and which classes are used by others. The latter is called associations.

Representation of the inner structure: The author of the class diagram then wants to show or define which attributes or methods the classes have.

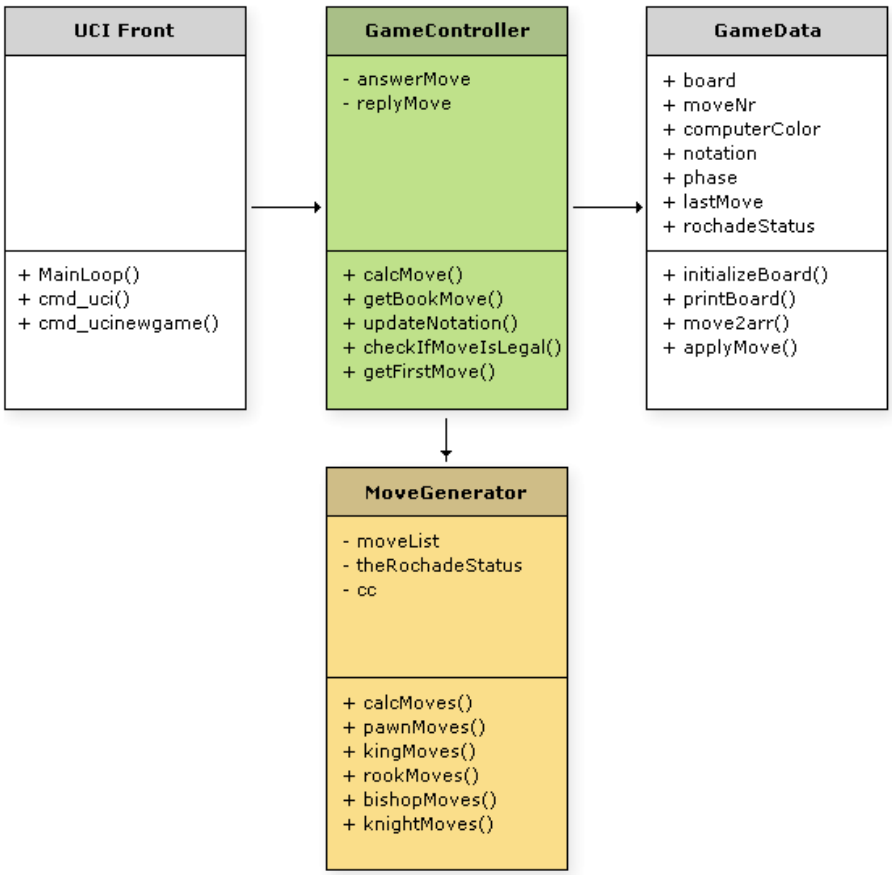
Both forms are often combined with each other and then form a detailed class diagram. But there's more information in the class diagram:

1. What stereotype does the class correspond to? For example, a display class, a controller class, and so on. This will be explained later.

1. Of which structure are the attributes and methods? Which signatures do the methods have? So this is about visibility, like public / private, the types, like int or string or the parameters, like int b, string c.

Beispiel

The following example shows a simple class diagram showing attributes and methods with their visibility (+ and -) but no more extensive associations.



Example Class Diagram

In UML 1, there was a strict distinction between an operation and a method - as the implementation of an operation. However, this is no longer the case since UML 2.

1.5.4.1 Notation

Title: Notation

The class itself is represented in a rectangle containing 1-3 areas. The class name is at the top, which is usually represented in bold type. Below this, the attributes first and then the methods. Class names are capitalized in almost all languages.

Hinweis

It makes sense to specify the responsibility of classes as a comment, i. e. what the class itself is supposed to do, unless this is shown trivially in the class itself.

The following are the attributes that can be specified with or without parameters or visibility. Visibility, types, parameters, constraints or initial values are also optional.

Class attributes are displayed in the same way and belong to each class instance once. This means that all instances can access this attribute.

A Class in the Class Diagram

Abstract classes can be marked in italics or bold. Most of the time
{abstract}
will be written under the class name.

As **Visibilities** we do have

- public **+**
- protected **#**
- private **-**
- und package **~**

These are noted with the symbols above.

Derived attributes can be defined in the UML with a forward slash "/".

It is interesting to note that - unlike in most programming languages - the name is displayed first and then the type (as in Scala or Nim). Don't let this confuse you. An example:
person: String = "Katharina"

This also applies to the parameters of an operation / method! So, that
getMwst (betrag: Integer)

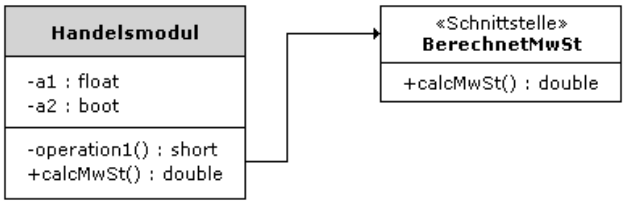
is a valid notation for a method.

1.5.4.2 Interfaces

Title: Interfaces

Yu can mark classes as Enumerations or Interfaces. Enumerations will be denoted as <<enumeration>>. Enumeration quantities (e. g. red, green, blue, etc.) can then be found in the body of the enumeration.

More important we have
Interfaces
, that specify a contract to be implemented in the form of method specifications. They are specified with the keyword
<<interface>>
above the class name.



Interface Example

It is said that the trading module realizes or implements the interface that calculates VAT.

Attributes are not written down in interfaces (but it is not forbidden) because it is about an access contract and not about the inner structure of the class.

Since interfaces are strict contracts, you will find more detailed specifications such as exceptions, invariants, preconditions and post-conditions.

Another possibility to write down interfaces is the representation as sphere and semicircle. The sphere implements the interface and the semicircle is the interface to which it is "docked".

Interface as a Circle and a Semicircle

In addition, you can describe the semicircle with one word, which property this class implements (e. g. B. serializable, or sortable).

1.5.4.3 Stereotypen

Title: Stereotype

Definition

Stereotype
A stereotype is a consistent pattern that occurs or can be reused. The term actually originates from psychology and sociology and refers to the same behaviour patterns of people or groups.

Translated to the UML and classes, this means that you can mark/classify a class or other UML elements with a stereotype. They enable the use of platform or domain-specific notation, which in a way marks them out to implement a certain behavior. For example, for
<<Stateless>>
for a stateless session bean.

Since UML 2, classes can contain several stereotypes. Most UML tools allow you to assign stereotypes directly to elements such as classes, dependencies, components or packages. Stereotypes are often displayed in the same colors.

The UML already specifies a number of so-called "standard stereotypes" that are available for your modeling. The following are some of the relevant standard stereotypes:

span class="courier"><<focus>> states that the classes identified in this way define the central logic or control flow.

- <<auxiliary>> implement secondary logic or secondary control flow and thus support other classes that are central to the model.
- <<<utility>> define a collection of static attributes and class operations that can be accessed from all classes.
- <<create>> can be one of two predefined stereotypes, of which

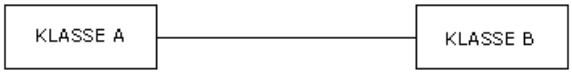
one can be applied to dependency relationships and one to operations.

- <<destroy>> is inverted to create and says that the operation destroys a form of the classifier in which it is contained.
- <<call>> indicates that a dependency relationship marked in this way has operations or classes with operations at its ends, to which the association refers.
- <<instantiate>> means that operations of the consumer generate characteristics of the provider.
- <<interface>> is an interface. It cannot generate any values, it does not contain any attributes, and its operations cannot be implemented in methods.
- <<send>> is applied to dependency relationships and says that the source of the relationship (an operation) sends a signal.
- <<type>> is attached to a class and says that this class groups a set of objects with respect to the operations that can be applied to these objects.
- <<responsibility>> implies an obligation or a contract between two model elements.

1.5.4.4 Assoziation

Title: Associations

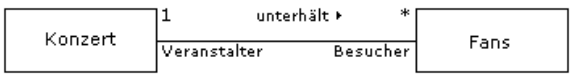
The association describes a relationship between classes in which one object uses the other.



Assoziation

Here is an example. Let's say a customer is the class. Several invoices can be assigned to this class. The Customer object must be able to access invoice objects. Special cases of association are aggregation and composition.

The simple association with a simple line is shown.



Qualified Association

The multiplicities with which you specify the number of links between the two objects are important. Examples are:
0... 1, 0... 42, 1... 1, 1... 42, 0... *, 1... *

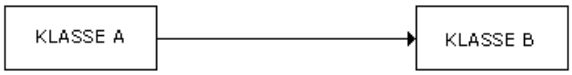
E/R diagrams are often referred to as
1:1, 1: N
or
N: M
- relationships. In general, relations are therefore important hints when specifying objects, database schemas, or database mapping information, for example.

Furthermore, associations can be qualified by marking the line with roles and properties.

And finally, there are multipartite associations in which several classes are linked. In this case, the lines converge into a small diamond.

Directed Assoziation

The directed association describes that you can get to the target class via the source class. But not the other way around. It is therefore a unidirectional relationship.

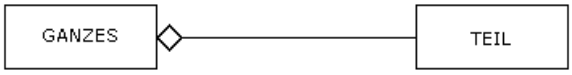


Directed Association

If you want to be able to navigate in both directions - i. e. bidirectional - you can combine two arrows into one. These can then also be provided with roles and properties.

Aggregation

An aggregation is a special case of an association. It is called a part-entire relationship.



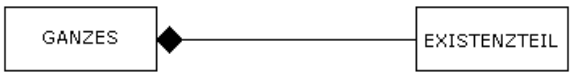
Aggregation

Therefore, you often write "consists of" as a property on the arrow. An example would be a "party" as a whole, which consists of many participants. If the "party" and thus the aggregation is dissolved, the participants are still able to survive. You don't have to have the party. However, the party only results from the aggregation of the party participants.

If you are in doubt as to whether aggregation exists, you can use an association without making a big mistake. However, a clear award is always more readable and useful.

Conposition

In contrast to aggregation, a composition is drawn with a filled rhomb.



Composition

It is the strongest form of aggregation. The parts depend on the existence of the whole thing. The best-known example is the invoice with its invoice items. If the invoice is deleted, the invoice items cease to exist.

Differently with an association (aggregation) to its members. If an association is deleted from the official register, the existing members would like to and can still live on.

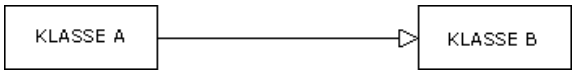
Abhängigkeit

One class requires another. Only then can it fulfill its task.

1.5.4.5 Vererbung / Generalisierung / Spezialisierung

Title: Inheritance / Generalisation / Specialisation

In inheritance, all attributes and methods are also available in the subclass. This is noted in UML with an arrow that is not filled in.



Inheritance

Linguistically, a distinction is made between generalization and specialization.

Generalization refers to the more general properties of the superclasses.

Specialization refers to the subclasses that add further special properties (attributes) and methods to those of the superclass.

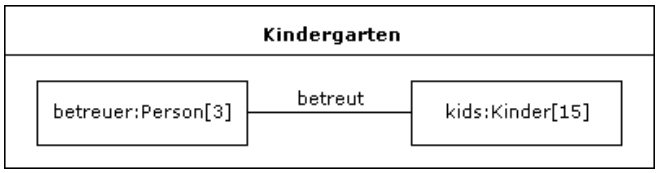
In many cases, the delegation makes more sense than inheritance, as it does not violate the principle of secrecy. In many cases - such as e. g. For example, with graphics libraries - inheritance is useful. Remember to always analyse this yourself.

Multiple inheritance is allowed in some programming languages. In this case, arrows can be combined - or if several subclasses are derived from a superclass (upper class).

1.5.5 Kompositionsstrukturdiagramm

Title: Composition Structure Diagram

Often the available diagrams are not sufficient to show which elements make up a class or component.



Composition Structure Diagram

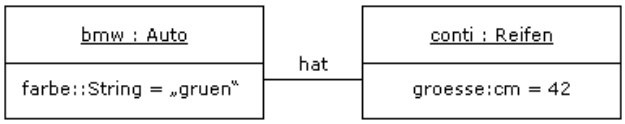
This diagram is drawn with the parent component in a box. This contains two sub-elements. In these elements, the name of the part (part) can be displayed, followed by a colon and a type of the part.

The **multiplicities** are given either in square brackets or superscripted (like the square). The parts can, in turn, be connected to a role.

1.5.6 Objektdiagramm

Title: Object Diagram

The object diagram shows the exemplary state of one or more objects in a system. It is like a photograph of the system in which the objects may have particularly interesting values. It is an instance diagram in which concrete attribute values can be specified.



Object Diagram

This picture shows classes in a certain state.

Just a few more points.

When defining the object class or attribute type, you are very free. These elements can also be simply omitted.

This diagram can also be useful if OCL expressions are specified. This would result in e.g.:
contect tires inv: size > 20
(therefore too small tires are not allowed).

As shown above, links can also be drawn between objects. These are then the associations and are only refined with the role.

1.6 Dynamische Diagramme

Title: Dynamic Diagrams

Now we continue with dynamic diagrams:

- **Use-Case Diagram** → What are the Use Cases?

- **Activity Diagram** → What is a way through the application?
- **State Diagram** → What states can the program be in?
- **Sequence Diagram** → How do classes / objects communicate?
- **Interaction Diagram** → An Activity Diagram which replaces some activities with interactions.
- **Communication Diagram** → Represents complex communication processes
- **Time Diagram** → State lines for theEmbedded- or the electronic

field.

1.6.1 Anwendungsfalldiagramm

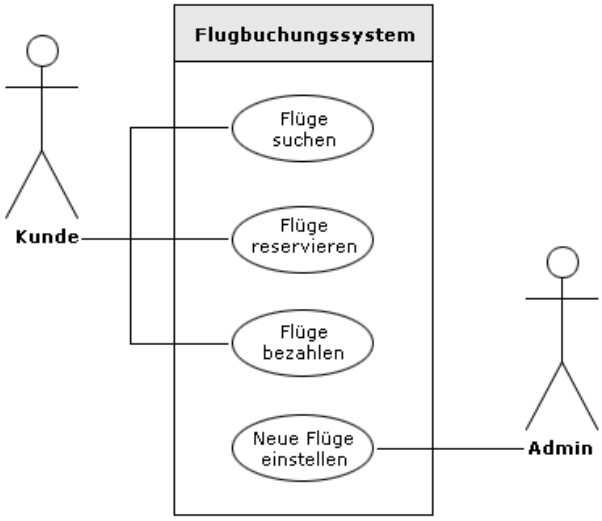
Title: Use Case Diagram

This diagram is quite important and is almost the first UML diagram to be drawn in projects.

The goal is to show what you want to do with the software system to be built. Which cases of application there are therefore! From the analysis phase, it is known that application cases need to be documented. This describes the workflows in the system that should be possible. In the UML, this is now to be supported even more easily with simple graphics.

The diagram shows so-called actors as stick figures. These can be people such as customers, administrators or any other users.

From there, strokes go to the respective use cases, which are written into an ellipse. To group them together, one notes related use cases in a box. Let me give you a very simple example:



Anwendungsfalldiagramm

The use case diagram is intended to communicate the use cases as simply as possible - for example, to management. Therefore, it is usually sufficient to look at such a diagram, and the viewer can immediately develop a vision of the system himself or suggest his own ideas. This is often more efficient than reading extensive application case documentation.

The strokes between the actors and use cases can be named as in the class diagram and provided with cardinalities. For example, only two pilots at a time want to have programmed the "Autopilot switch off" application.

Note in the above example that you have the feeling that the actions take place from top to bottom. But application case diagrams are not directed! Activity diagrams are available for the chronological sequence. Mixing the diagram types is a typical bug of UML beginners. Connecting complex use cases in a Use-Case diagram with many arrows is not correct.

Weitere Anwendungsfallbeziehungen

1. A dashed arrow indicates that there is a realization. This means, for example, that the use case "Go to reporting point" can be realized by two cases connected with a dashed arrow, which can be called "drive to reporting point by car" and "walk to reporting point". Together, the two of them make up the main application case.

1. If you write

   <<include>>

at the dashed arrow , the case of application to which it is pointed is included. It's like programming languages do. Thus, if the application consists of three internal parts A, B, C, then part A can be outsourced as an application and replaced with

<<include>>

the dashed arrow to show the important include again.

1. With

   <<extends>>

   and a dashed arrow, it is said the other way round that an application can be extended with another one. The arrows always go from the external / include object to the main object that uses it.

1. A specialization arrow (analogous to that in the class diagram) indicates that several "sub-application cases" constitute a more general use case. For example, a use case can be the "standard authentication", which is carried out either with an "electronic ID card authentication" or with a "mobile phone number authentication".

1.6.2 Aktivitätsdiagramm

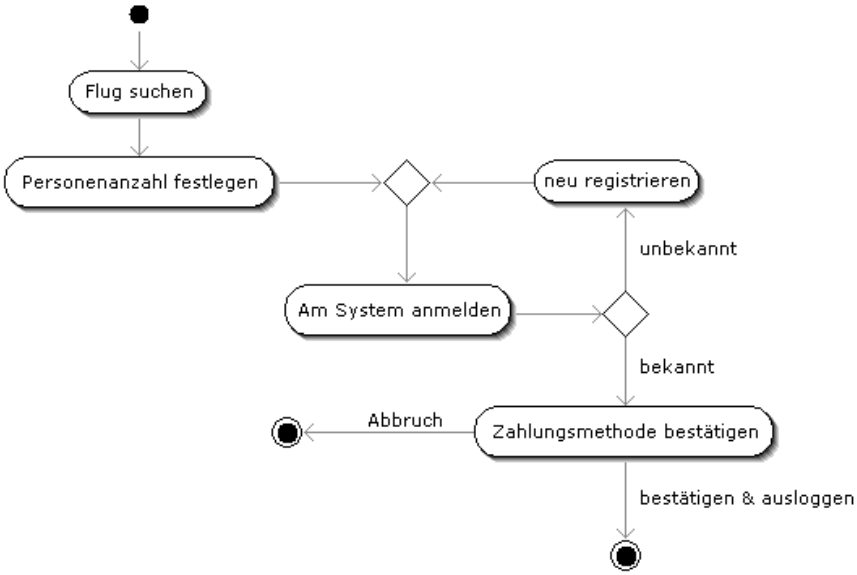
Title: Activity Diagram

The Activity Diagram is one of the most complex diagrams with its many elements. Not all elements can be played here, but they are easy to look up in the UML overview of OOSE. We are confining ourselves to the most important elements.

uml-2-Notationsuebersicht-oose.de.pdf
(304 KB)

The activity diagram is usually used early in the modeling phase and shows the sequence in which the user is guided through a possible path of the application. There are action nodes, object nodes and control nodes. The latter allow you to branch out and make decisions.

Activity diagrams visualize and explain the program's flow, which is often very difficult to visualize with the natural language or the written word.



Aktivitätsdiagramm

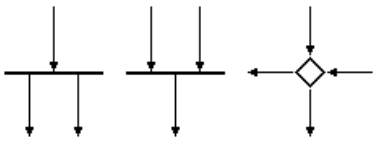
In this diagram you can see some action nodes (called activities at times of UML 1). Furthermore, a start node and an end node. On the right there is a branching. Object nodes are not drawn. In this way, you could insert objects in various places, which are then available at this point.

For example, a flying object that is selected by the customer after a successful search. Or an object representing the entire flight including customer and payment data.

As in many UML 2 diagrams, it is possible to place diagrams in a box and thus illustrate that an activity can consist of several substeps.

Kontrollknoten

Branches, merges or decisions are represented by control nodes. For merges, conditions can be specified and for branches, conditions can be written to the outgoing arms.



Kontrollknoten

For merging nodes, conditions can be formulated like this:

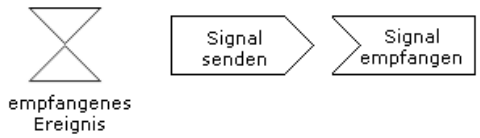
{joinSpec=(X or Y) and (M or N)}

.

For branching, conditions can be specified in square brackets: <span class="quellcode"> [X<0]</span>.

Weiterführendes

1. **Parameter Pins Object Knots**: Pins represent In and Out parameters for objects.
2. **Swimlanes**: can represent areas, as can be the case in sequence diagrams.
3. **Signale**: Signals can be embedded in the control flow of many diagrams. These are shown as shown below.



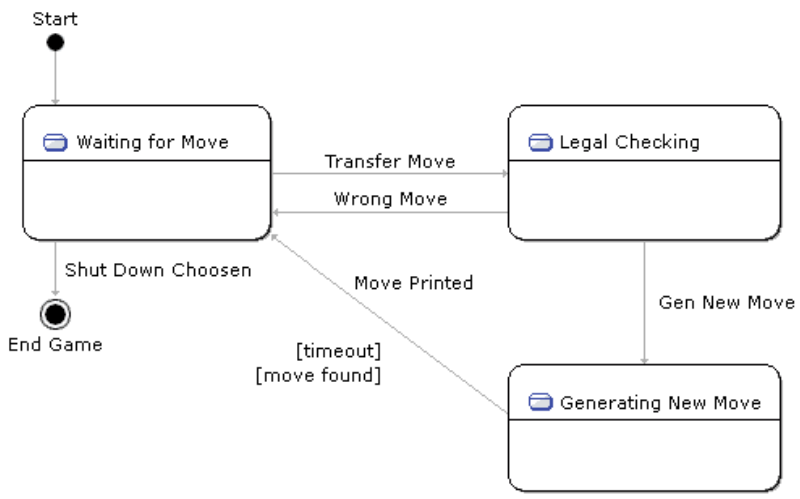
Signale

1.6.3 Zustandsdiagramm

The state diagram describes the behavior of a system in response to certain events. Such a state machine can be traced back to old modeling attempts of Mealy and Moore machines.

It contains:

- States (including compound states and whole substates)
- Transitions
- Regions / Areas
- Start and end state



State Diagram

The notation possibilities differ greatly from application to application. In the simplest case, there is only one box for each state in which the state is described. In this case, the horizontal line should be removed (unlike in the picture here). The three simplest states of a (chess) program are shown here, which can have three states.

Each diagram is usually framed by a box in which the name of the whole diagram is shown at the top.

Furthermore, it is possible to refine use cases in the use case diagram by using state machines. This is often done during authentication, for example.

Notationselemente

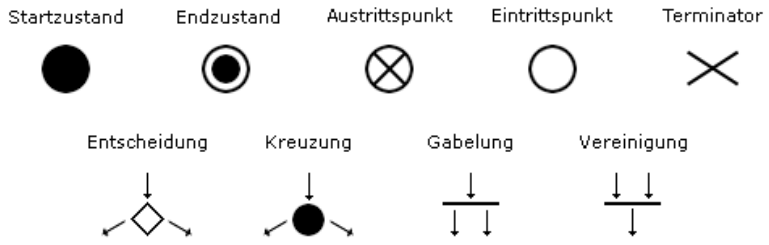
A state is usually (but optionally) given a comprehensible name in the upper box.

The following information can be given in this state:

1. **entry** / Activity
2. **exit** / Activity
3. **do** / Activity
4. **Trigger** / defer, we distinguish:
    - SignalTrigger
    - CallTrigger
    - TimeTrigger
    - ChangeTrigger and
    - AnyTrigger for all

A **Guard** is a condition that can be specified at a transition.

Other elements are already known from other diagrams (such as activity).



Diagrammelemente

1.6.4 Sequenzdiagramm

Title: Sequence Diagram

Sequence diagrams describe a dynamic view of the system. You use it to show how classes interact.

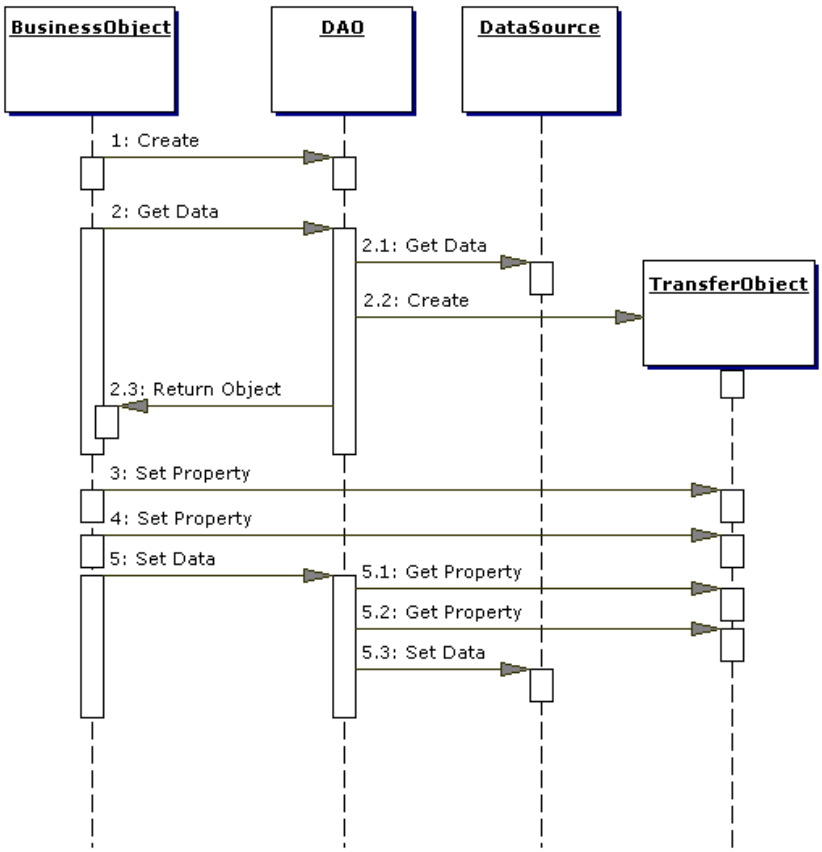
Since the method calls between classes are also referred to as messages that are sent, the diagram displays the messages in chronological order. At the top of the screen (underlined) objects are shown, but classes or even coarser units are also possible. They have lines pointing downwards (so-called swimlanes) or lifelines.

On these lines you can see in blocks how objects are instantiated. Messages from them then go to other objects (blocks). These are displayed as vertical arrows. Messages can have return arrows. The diagrams allow the display of synchronous (filled arrows) and asynchronous messages (non-filled arrows).

At the same time, signals can be received as in the previous diagrams, or branching, iterations and recursions can be displayed. For this purpose, boxes are drawn into the diagram and then, for example, loop or alt (if) are drawn.

An initial found message can start the sequence diagram.

The following diagram of SUN Microsystems shows one of the best and most typical applications for sequence diagrams - Design Patterns. A good architecture for data access is shown here.



Sequence Diagram

What's happening here? The DAO pattern is shown. It contains four objects. Let's take a quick look at it. Not to understand the pattern, but to show why sequence diagrams are useful.

1. A business object requires data from the persistence layer (the data access layer, i. e. ultimately from the database).
2. It is addressed to a DAO object responsible for all database connections.
3. This DAO knows which data access is currently active and gets the database connection via the DataSource.
4. A data transfer object then contains the data in a suitable form (e. g. better than ResultSets) and this is returned to the business object in 2.3.

What follows from 3 to 5.3 is the reversal, which doesn't have to be of particular interest to us in terms of content. For the sake of form: Instead of writing back data via JDBC, for example, it is now state-of-the-art to change the transfer object and then save it via the DAO.

You have seen the scenario to see how helpful sequence diagrams are. The communication of four important objects in the data retention layer becomes more transparent through the display. Together with the Active Record Design Pattern, the example shows the most important design patterns of the data retention layer.

It's interesting to note that sequence diagrams often have two types of communication scheme:

1. **Centralized Control:** One object does the work (calculates or holds the threads like a spider in hand) and all the others close.
2. **Distributed Control:** All classes / objects are equal and "calculate" as it were. There's no central control.

In many cases, the second structure is more advantageous and allows more object-oriented concepts (polymorphism, overwriting, etc.).

For more information, see the following tutorial:

Tutorial: Sequenzdiagramme http://www.tracemodeler.com

1.6.5 Interaktionsdiagramm

Title: Interaction Diagram

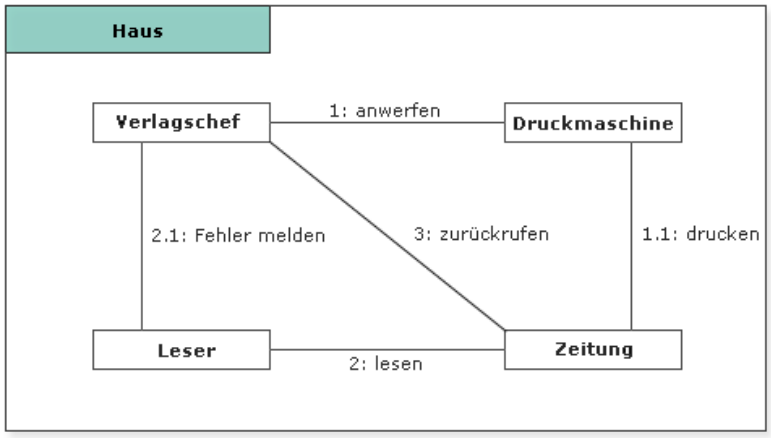
An interaction diagram is a hybrid form of an activity diagram in which individual activities are explained in more detail by an interaction diagram.

In concrete terms, an activity diagram is therefore available in which individual states are to be explained in more detail. These states are then magnified in the diagram and an interaction diagram is embedded, i. e. displayed. This is used whenever states in activity diagrams are very complex.

1.6.6 Kommunikationsdiagramm

Title: Communication Diagram

The communication diagram represents more complex communication processes.



Communication Diagram

What we see here is the attempt to depict a complex sequence in the correct order.

The first communication line is the newspaper's start-up and printing by the head of the publishing house. The buyer reads the newspaper and reports a big mistake. The head of the publishing house pulls the newspaper out of circulation.

In computer science, such a diagram is used whenever the contents of states, transitions and data are not important but the elements should be represented in a diagram. This means, for example, that the tasks or data of the press are not important, but only that they exist and how they interact.

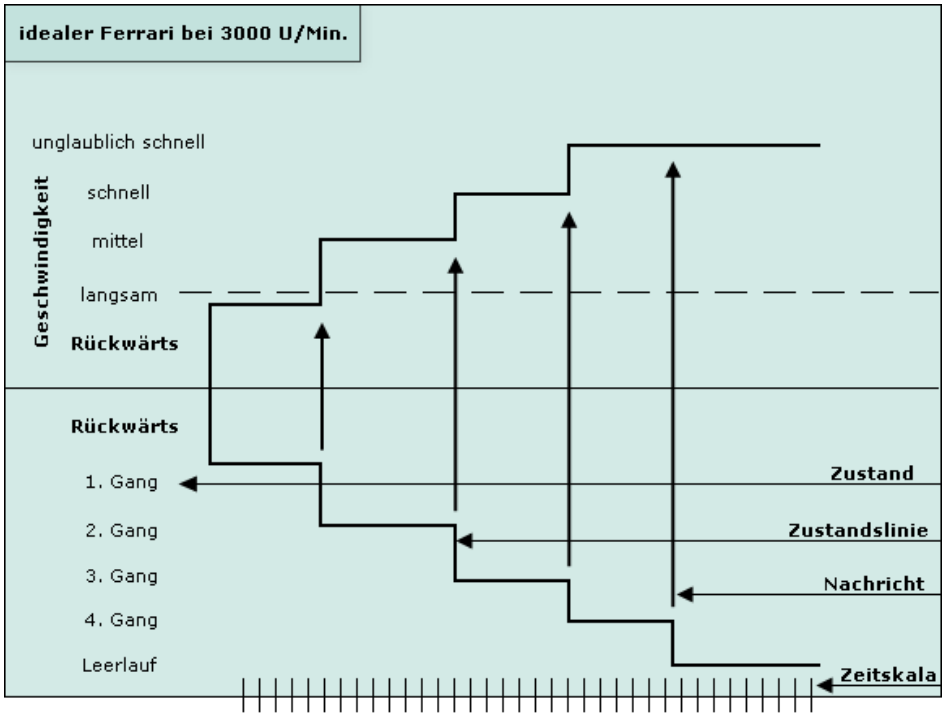
1.6.7 Zeitdiagramm

Title: Timing Diagram

Time diagrams are mostly used for embedded applications or electrotechnical applications. Only in these areas are there such precise processes or procedures that they have to be represented in time. In the otherwise customary modeling of software engineering processes - such as an activity diagram - the chronological sequence does not matter. A booking process can take seconds or even hours.

The diagrams look like signals on an oscilloscope and contain:

1. Time lines
2. Terms and conditions
3. Messages that can lead to a change of state.



Timing Diagram

1.7 Zusammenfassung UML

Title: UML Summary

- As a set of interaction elements, UML serves as a method for developing models for software systems.
- UML helps determine the design process.
- The fathers of UML are Grady Booch, James Rumbaugh and Ivar Jacobsen. They are also called "The Three Amigos".
- UML is not a dogma and is not absolutely necessary for the development of a software system.
- The purpose of UML is to communicate ideas. By using various diagrams, it is possible to illustrate situations that would be difficult to present in writing or in language
- In the UML, different diagram types are used for display. Here is a subdivision into:
- ○ **Static Structure Diagrams**

(Zur Darstellung der Struktur von Modulen eines Softwaresystems. Die UML kennt sechs Strukturdiagramme: Klassendiagramm, Kompositionsstrukturdiagramm, Komponentendiagramm, Verteilungsdiagramm, Objektdiagramm und Paketdiagramm.)

- ◦ **Dynamic Diagrams**, also behaviour diagrams
  (For displaying function sequences. The UML uses seven behavioral diagrams: Activity diagram, use case diagram (also known as use case diagram), interaction overview diagram, communication diagram, sequence diagram, time history diagram and state diagram.

- There are a number of tools that support modeling with UML. However tools describing the UML on a meta level (not drawing) as plantuml have many advantages.

1.8 Wissensüberprüfung UML

Title: UML excercise

Task UML-01: Specification

Download the UML specification from OMG and read / browse through it a bit.

Time needed: 20 Min

Task UML-02: Tools

Make sure you have your favourite UML tool at hand. Evaluate some of the tools mentioned in the session. Allow some time for this. Often you are not satisfied with the first tool. For example, one does not like the generated code or a diagram is missing. If a tool suits you, practice with it and make sure you feel fit in it!

Time needed: 60 Min

1.9 Literatur UML

Title: UML Books

When designing, working with UML tools and in project work it is extremely important to use good literature to look up the subtleties and meaning of the notation elements correctly. The books listed here also contain many examples.

Three works that were also helpful in the creation of the module are worthy of special mention:

Martin Fowler:

"UML Distilled"

Pearson Publishers

Jeckle, Rupp, Hahn, Zengler, Queins:

"UML 2 glasklar"

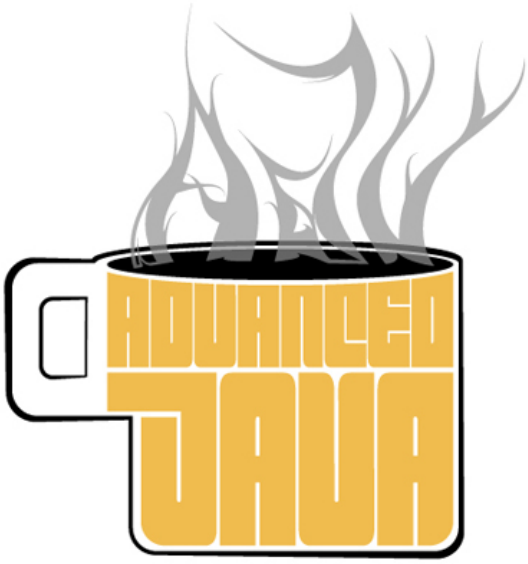
Hanser

Bernd Österreich:

"Analyse und Design mit UML 2.1"

Oldenbourg

2 AJV - Advanced Java



Autor: Prof. Dr. S. Edlich

2 AJV - Advanced Java
2.1 Überblick und Lernziele AJV
2.2 Regular Expressions
2.3 Performance Tuning
2.4 Debugging and Testing
2.5 Guava
2.6 JavaFX
2.7 Zusammenfassung AJV

2.1 Überblick und Lernziele AJV

Lernziele

Lernziele

Diese Lerneinheit hat zwei Ziele:

1. Viele von Ihnen sind keine Kerninformatiker, sondern beispielsweise Medieninformatiker. Da wir jedoch im Bachelor die Programmierung mit Java voraussetzen müssen, gibt ihnen diese Lerneinheit Gelegenheit, sich mit dem Programmieren und Java im Allgemeinen wieder vertraut zu machen.

2. Sie bekommen die Gelegenheit, sich mit Konzepten, Tools und Frameworks vertraut zu machen, für die im Grundstudium keine Zeit blieb.

Gliederung

Gliederung

- Regular Expressions
- Performance Tuning
- Debugging
- Guava
- JavaFX

Zeitumfang

Zeitumfang

Zum Durcharbeiten der Lerneinheit benötigen Sie ca. 30 Minuten und für die Bearbeitung der Übungen ca. 60 Minuten.

References:

- Survey by Zeroturnaround:

2.2 Regular Expressions

Bücher

O'Reilly hat viele gut Bücher im Programm wie Mastering, Introducing, Cookbook, Pocket Reference, etc.
Wrox und Apress haben auch zwei empfehlenswerte Bücher im Programm.

Weblinks

- Premier Regexp Website: mit Reference
- Deutsches Tutorial
- User Guide
- Learn RegExps **in 55 Minutes** (Top!)
- Interactive Tutorial

Interaktive Tools

- http://www.regexbuddy.com
- http://gskinner.com/RegExr
- RegEx Tester
- Vizualise them

Einführung

Reguläre Ausdrücke (Engl. *regular expressions*) sind ein mächtiges Werkzeug, um eine Datenmenge automatisiert zu überprüfen, zu durchsuchen oder zu manipulieren. Heutzutage beherrschen die meisten modernen Programmiersprachen, Textprozessoren und Datenbanken reguläre Ausdrücke. Da man nicht nur in der Programmierung, sondern auch in vielen anderen Gebieten (z.B. der Datenbankprogrammierung) immer schnell auf den Bedarf der regular Expressions stößt, wird hier das grundlegende Handwerkszeug vermittelt.

Reguläre Ausdrücke beschreiben Text. Genauer gesagt, sie sind eine allgemeine Notation, die für die Beschreibung von Textmustern verwendet wird. Die Bandbreite der Verwendung von regulären Ausdrücken umfasst verschiedenste Anwendungsfälle wie „Suche und Ersetzte“, bis hin zu komplexeren Aufgaben wie der Textverarbeitung. Weitere typische Aufgaben für reguläre Ausdrücke wären beispielsweise:

- Finde die Verdoppelung von Worten (wie „das das“)
- Prüfe korrekte Feldeingaben in einem Formular (wie E-Mail Adresse oder Telefonnummer)
- Überprüfe das Zeit- oder Datumsformat

Regular Expressions sind traditionell in der Web-Programmierung (PHP, Perl, etc.) oder in Unix-Umgebungen zu finden. Außerdem, bieten fast alle Programmiersprachen (beispielsweise Java, .NET Sprachen, Perl, Ruby, Python, Scala, Clojure, etc.) Interfaces, die die Arbeit mit regular Expressions ermöglichen oder bei denen String-Funktionalität gleich mit regulären Ausdrücken verfügbar ist. Auch die meisten Datenbanken enthalten Strings oder Textmanipulationsmechanismen mit regular Expressions. Diese stellen elegante und wirkungsvolle Methoden dar, um die Texte im Umfeld der relationalen Datenbanken zu verarbeiten oder abzufragen.

Hier behandeln wir die Grundlagen regulärer Ausdrücke. Es wird im Folgenden die Grundstruktur der „regular expressions“ vorgestellt wie Metazeichen, Quantoren und Literale sowie Zeichenklassen und Gruppen. Zahlreiche Beispiele werden die wesentliche Arbeitsweise mit regulären Ausdrücken veranschaulichen. Anschließend wird eine Zusammenfassung über die regulären Ausdrücke gegeben und es werden Tools vorgestellt, die eine effiziente Arbeit mit diesen ermöglicht.

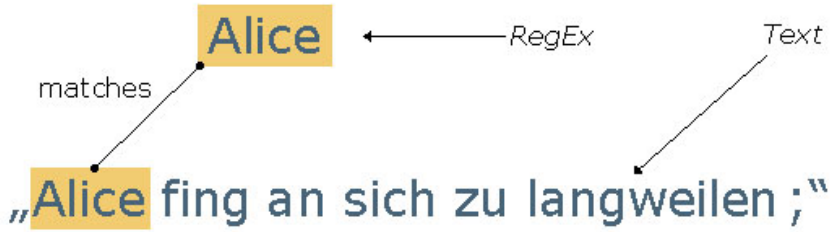
2.2.1 Zeichenliterale

Eine gute Analogie zu den regulären Ausdrücken sind Legosteine, die aufeinander aufgebaut werden. Genauso wie Lego, bestehen reguläre Ausdrücke aus mehreren Komponenten: Metazeichen und Zeichenliterale. Zeichenliterale sind die einfachsten regular Expressions. Dies sind alle Buchstaben und Zahlen. Der einfachste reguläre Ausdruck wäre dann nur ein Zeichen, z.B. das [A]. Wenn man diesen regulären Ausdruck auf den Text „Alice fing an sich zu langweilen;“ anwenden würde, dann würde sich der erste Buchstabe im Wort Alice als Treffer herausstellen.

Bei der Anwendung einer regular Expression auf einen String, werden grundsätzlich folgende Schritte ausgeführt:

1. Der reguläre Ausdruck wird kompiliert.
2. Die Regex-Engine wird auf den Stringanfang positioniert.
3. Der Suchstring wird Zeichen für Zeichen abgearbeitet.
4. Treffer werden festgestellt oder regular Expressions schlagen fehl.

Wenn man nach einem bestimmten Wort im Text sucht, dann verwendet man genau dieses Wort als regular Expression. Sucht man beispielsweise nach dem Wort „Alice“, sieht der reguläre Ausdrück genauso aus: Alice. In dem obigen Beispiel wird dementsprechend das erste Wort „Alice“ gefunden:



RegExcp
Einfacher Regulärer Ausdruck

Angenommen das Wort „Alice“ wird als „Alise“ geschrieben. Sucht man nach jedem Vorkommen dieses Wortes, sei es falsch oder richtig geschrieben, lässt sich der Regex wie folgt formulieren: Ali[cs]e. Dies bedeutet also wörtlich: „Suche nach dem Vorkommen der Buchstaben ‚A‘ und dann ‚l‘ und dann ‚i‘ gefolgt von entweder ‚c‘ oder ‚s‘, und danach einem ‚e‘“. Die Struktur […], wird Zeichenklasse genannt und gibt eine Zeichenmenge an, nach der gesucht wird. In dem angegeben Beispiel wird nach ‚c‘ oder ‚s‘ gesucht.

Die Anzahl der gesuchten Zeichen wird meistens durch die Kardinalität bestimmt. Regular Expressions bieten drei Quantoren an: ?, + und

•

. Der ? Repetition-Quantor bedeutet „kein oder einmal“ und das entsprechende Zeichen ist nur einmal oder gar nicht erlaubt. Der +-Quantor wird für mindestens ein Treffer vorgesehen, wobei der

•

-Quantor für „Null oder mehr Hits“ steht.

Wir betrachten drei Beispiele: A?, A+ und A*. Im ersten Fall muss mindestens ein Zeichen ‚A‘ oder gar keines vorkommen. Im zweiten Fall werden mindestens ein oder mehr ‚A‘-Zeichen erwartet. Im dritten Fall ist das Zeichen ‚A‘ mehrfach oder überhaupt nicht präsent, d.h. ‚A‘ darf auch im Text abwesend sein.

Folgendes Muster AB[0-9]+ bedeutet: „Suche nach einem Zeichen ‚A‘ und dann ‚B‘ gefolgt von mindestens einer Ziffer“. Für diesen regulären Ausdrück würde dann der Text „AB5“ passen. Möchte man die genaue Trefferzahl bestimmen, verwendet man die Curly-Brace Syntax. Es existieren zwei Varianten der Curly-Brace Syntax: Die {n}- und {n,m}-Notation.

{n} Syntax bedeutet, dass ein bestimmtes Zeichen oder eine Zeichengruppe genau n-Mal vorkommt. {n,m} Notation setzt wiederum eine Minimum und Maximum-Grenze, d.h. ein Zeichen oder ggf. eine Gruppe von Zeichen darf mindestens n-mal und Maximum m-mal vorhanden sein, damit das regular Expression mit dem Text übereinstimmt. Beispielsweise passt der reguläre Ausdruck AB[0-9]{3} zu dem Eingabetext „AB837“. Das Muster AB[0-9]{3,5} würde dann den Text „AB58607“ entdecken.

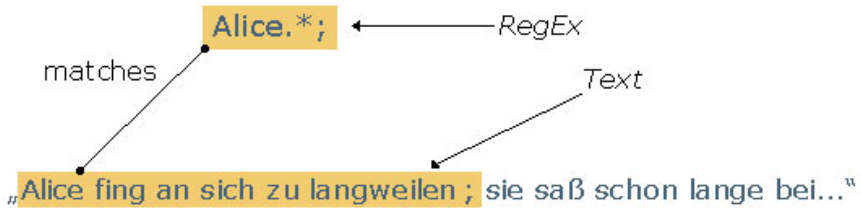
Möchte man nur eine Untergrenze bestimmen, verwendet man dann die folgende Syntax: {n,}. Das heißt, dass man mindestens das n-malige Auftreten eines bestimmten Zeichens erwartet wird, wobei die obere Grenze offen bleibt. Alle hier vorgestellten regulären Ausdrücke sind am Ende nochmals in einer „Reference Card“ zusammengestellt.

2.2.2 Metazeichen

Die zweite wichtige Komponente der regulären Ausdrücke sind Metazeichen. Gerade sie machen reguläre Ausdrücke zu einem mächtigen und effizienten Tool. Wie der Name sagt, erfüllen diese Spezialzeichen eine beschreibende Rolle in regular Expressions. Diese Metazeichen bestimmen beispielsweise die String- oder Wortgrenze,

Gruppierung, Zeichenklassen, Alternation, Lookahead und Lookbehind, sowie bedingte reguläre Ausdrücke und Kommentare. Die bereits besprochenen Repetition Quantoren gehören ebenfalls zu den Metazeichen.

Zu den meist benutzten gehört der Punkt ., was für alle mögliche Zeichen steht, sowohl für Buchstaben als auch für Zahlen, aber nicht das Newline. Wichtig zu wissen ist, dass der Punkt nicht nur bei Klein- und Großbuchstaben des Englischen Alphabetes zu finden ist, sondern auch alle Zeichen aus den nicht-Englischen Alphabeten trifft. Als Beispiel wird der folgende reguläre Ausdrück angewendet:
Eingabe: „Alice fing an sich zu langweilen;
sie saß schon lange bei ihrer Schwester
am Ufer und hatte nichts zu tun.“
Regex: Alice.*;
Beschreibung: „Suche nach dem Text
der mit dem Wort ‚Alice‘ anfängt,
wenn dies gefunden wurde,
suche nach beliebig vielen Zeichen,
bis der Zeichen ‚;‘ vorkommt“
Ergebnis: „Alice fing an sich zu langweilen;“



RegExps
RegExp Anwendung

Möchte man aber die Spezialzeichen wie ., [,],

•

, usw. als Teil des Musters verwenden, sollte man diese mit \ zusammen maskieren. Beispielsweise \. oder *. Zu den Metazeichen gehören auch solche, die Leerzeichen oder Wort-Bestandteil bestimmen: \s bzw. \w. Wobei das letztere sich als Zeichenklasse [a-zA-Z0-9] darstellen lässt. Wichtig bei dieser Kategorie der Metazeichen ist, dass sie mit \ „escaped“ werden sollen.
Eingabe: „Alice fing an sich zu langweilen;“
Regex: Alice\s+\w+
Beschreibung: „Suche nach dem Text

der mit dem Wort ‚Alice' anfängt,
wenn dies gefunden wurde,
suche nach mindestens einem Whitespace-Zeichen,
danach finde alle Bestandteile nächsten Wortes"
Ergebnis: „Alice fing"

Weitere Metazeichen sind in den nachfolgenden Seiten gelistet.

2.2.3 Gruppen und Klassen

Zwei weitere Aspekte der regulären Ausdrücke sind Zeichenklassen und Gruppen. Klassen werden in […] angegeben und bestimmen, nach welchen Zeichen gesucht werden soll. Mit einer Zeichenklasse [a-zA-Z] wird nach einem Zeichen von ‚a' bis ‚z' oder ‚A' bis ‚Z' gesucht, und dies entweder klein oder groß geschrieben.

Um eine Gruppe zu bilden, werden runde Klammern (…) verwendet. Wenn man das Regex Beispiel (Wunder)\w* auf den Eingabetext „Alice im Wunderland" anwendet, ist das Ergebnis dann „Wunderland". Wobei dieses Ergebnis zwei Gruppen enthält: Die erste Gruppe besteht aus „Wunder" und die zweite Gruppe aus „land". Auf diese Weise ist in den bereitgestellten Daten eine verfeinerte Suche möglich.

2.2.4 Lookaround

Grundsätzlich arbeiten reguläre Ausdrücke von links nach rechts. Bei dem Lookaround Feature kann der String in beide Richtungen (d.h. nacht links und nach rechts) untersucht werden. Das Schlüsselwort hier ist „untersuchen", weil Lookaround „konsumiert" keinen Text. Lookaround ist in zwei Formen vorhanden: Lookahead und Lookbehind. Lookahead ist eine Art von Lookaround, wo der String von links nach rechts untersucht wird. Der reguläre Ausdruck ist dann gefunden, wenn der Unterausdruck im Lookahead erfolgreich war. Lookahead wird als spezielle Klammerung geschrieben (?=…). Ein Beispiel soll Lookahead verständlicher machen:
Eingabe: „Alice im Wunderland"
Regex: (?=Wunderland)Wunder
Beschreibung: „Suche nach einem Wort
‚Wunder', passt nur wenn dieses Wort
ein Teil des Wortes ‚Wunderland' ist"
Ergebnis: Wunder

Bei Lookbehind ist die Arbeitsweise ähnlich wie bei Lookahead, bis auf, dass hier der String in Rückwärtsrichtung untersucht wird: Von rechts nach links. Für Lookbehind wird die Klammerung (?<=...) angegeben. Am besten wird dieses Feature an einem Beispiel veranschaulicht:
Eingabe: „Alice im Wunderland“
Regex: (?<=Wunder)land
Beschreibung: „Suche nach einem Wort
‚land‘ nur wenn man rückwärtsschauend
vorne ein ‚Wunder‘ steht“
Ergebnis: land

Anmerkung

Lookaround wird oft für die Ermittlung der Position im Text verwendet. Dies ist bei der Textmodifikation wie Substitution oder Ergänzung notwendig.

2.2.5 Tabellarische Zusammenfassung

Die folgenden Tabellen 1-3 sind eine Sammlung von regular Expressions mit deren Beschreibungen. Diese regulären Ausdrücke sind in den meisten Regex-Engines implementiert.

Metazeichen	**Beschreibung**
.	Irgendein Zeichen
\	Ist ein Zeichen ein Metazeichen, dann escaped mit \ bedeutet dieses das Zeichenliteral.
[...]	Zeichenklasse. [abc] bedeutet ein Zeichen aus der Liste ‚a‘ oder ‚b‘ oder ‚c‘
[^...]	Kein Zeichen aus der vorgegebenen Liste.
?	Kardinalitätsquantor. Bedeutet „passt einmal, aber optional“.
+	Kardinalitätsquantor. Bestimmt mindestens ein Treffer, mehr Treffer sind erlaubt.

*	Kardinalitätsquantor. Bedeutet, dass beliebige Trefferanzahl erlaubt (auch null)
{n, m}	Kardinalitätsquantor. Bestimmt die untere Trefferanzahl-Grenze – n, m-Treffer sind erlaubt.
|	Alternation. `a|b` passt entweder auf ‚a' oder ‚b'
(...)	Gruppieren von Treffer.
^	Definiert den Zeilenanfang.
$	Definiert das Zeilenende.

Metazeichen

Sonderzeichen

Spezielle Zeichen	Beschreibung
\n	Neue Zeile.
\t	Horizontaler Tabulator.
\v	Vertikaler Tabulator.
\b	Backspace. Wichtig bei dem Zeichen \b ist, dass in Java dieses Zeichen eine andere Bedeutung hat: Wortgrenze.
\f	Formfeed.
\r	Carrige Return.

Spezielle Zeichen

Zeichenklassen

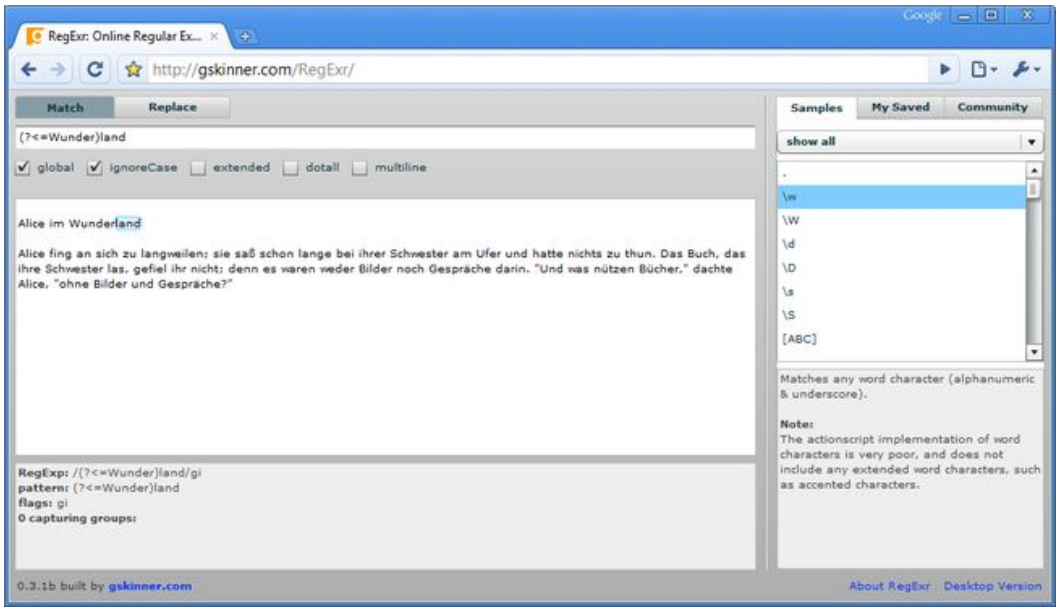
Zeichenklassen	Beschreibung
\d	Eine Ziffer oder das Gleiche wie [0-9]
\D	Nicht-Ziffer-Zeichen oder [^0-9]

\s	Leerzeichen (Whitespace).
\S	Nicht-Leerzeichen oder [^\s]
\w	Bestandteil eines Wortes oder [a-zA-Z0-9]
\W	Nicht-Wortzeichen oder alle Zeichen außer [a-zA-Z0-9]
\X	Unicode Zeichen für kombinierte Zeichen.

Zeichenklassen

2.2.6 Reguläre Ausdrücke Testen

Einige Tools und Sprachen, die reguläre Ausdrücke unterstützen, wurden bereits am Anfang dieser Lerneinheit erwähnt. Doch bevor man diese verwendet, macht es durchaus Sinn, die regulären Ausdrücke vorher auszuprobieren und zu testen. Besonders wenn man noch wenig Erfahrung mit regular Expressions hat, würde das lohnen. Es existieren mehrere kommerzielle / freie Tools und Plug-Ins für Entwicklungsumgebungen, um reguläre Ausdrücke zu lernen und zu testen. Die meisten davon sind leistungsfähig, z.B. das kommerzielle Tool RegexBuddy. Als freier und besonders für Anfänger geeigneter Regex Editor lässt sich RegExr empfehlen:



Tools
RegExp Test Tools

2.2.7 Fazit

Obwohl das Erlernen und Verstehen von regulären Ausdrücken eine gewisse Lernkurve und Übung verlangt, zahlt sich dies meistens aus. Hat man in dem Programmierer-Alltag mit String-Manipulationen oder Suche im Text zu tun, dann sind reguläre Ausdrücke von großer Bedeutung. Eine korrekte regular Expression hilft nicht nur den Code sauber und kurz zu halten, sondern bringt auch oft Performancesteigerungen mit sich. Es gibt unterschiedliche Benchmarks, die feststellen, wie schnell die Ausführung von regular Expressions ist. Man sollte diese Ausführungszeit immer gut im Auge behalten. Das ist besonders wichtig, wenn z.B. „Backtracking“ involviert ist. Ob eine Optimierung notwendig ist, hängt dann meistens von der angewandten Programmiersprache oder Tools ab.

2.3 Performance Tuning

- visualVM
- yourkit
- Caliper MicroBenchmarks

2.4 Debugging and Testing

References:

- Michael Nygard, "Better then unit Testing", link

2.5 Guava

References:

- Original Reference:
- Guava API:
- Original Documentation, "Guava Explained"
- Remove the fluff
- 5 Reasons to use Guava
- Presentation

Herkunft

Google Guava hieß ursprünglich **Google Collections** und stammt initial von zwei Google Entwicklern aus dem Jahre 2007. Grund war die mangelnde Anpassung von anderen Bibliotheken wie Apache Commons an Generics. Hilfe bekamen die Autoren von Joshua Bloch (Effective Java) und Doug Lea (Java Concurrency).

Bestandteile

Guava untergliedert sich in drei wesentliche Bestandteile:

1. **Basisfunktionen**: Hier sind Dinge zu finden wie die Behandlung von Null-Objekten, Pre- und Postconditions, equals, hashCode, toString, und viele weitere kleine Teile.
2. **Collections**: Hier gibt es z.B. immutable Collections, Multimaps, Maps für Listen und die vereinfachte Verarbeitung / Aufruf von Collections.
3. **sonstige Pakete**: Für Caching, die funktionale Programmierung, Range Objekte, etc.

Einiges wurde durch Java 7 und 8 obsolet. Viele Teile sind aber weiterhin extrem hilfreich.

Übersicht

Im Folgenden eine vereinfachte Übersicht aus :

1. **Basic Utilities**
    1.1. Using and avoiding null *
    1.2. Preconditions *
    1.3. Common Object Methods
    1.4. Ordering
    1.5. Throwables
2. **Collections**
    2.1. Immutable Collections *
    2.2. New collection types
    2.3. Powerful collection utilities
    2.4. Extension utilities
3. Caches
4. Functional Idioms
5. Concurrency
    5.1. Listenable Future
    5.2. Service
6. Strings
7. Primitives

8. Ranges
9. I/O
10. Hashing
11. EventBus
12. Math
13. Reflection

1.1 Using and avoiding Null

- Hier lesen !

Neben copy-und-paste sind null Werte im Code mit die häufigsten Fehler in Softwarecode. In der Regel bedeutet Null, dass etwas nicht vorhanden ist. Was liegt also näher, in diesem Fall nicht Null zurückzugeben, sondern ein spezielles Objekt?!

Dies hätte zwei Vorteile:

1. Es tritt nicht zwingend sofort eine Null-Pointer Exception auf (im schlimmsten Falle eine andere).
2. Viel wichtiger ist aber, dass man sich als Programmierer darüber Gedanken macht, wie man mit nicht vorhandenen Objekten umgeht! Und das, indem Null einen Namen bekommt.

Google stellt für diese Fälle ein Objekt
optional<T>
zur Verfügung, welches ein Wert zur Verfügung stellen kann oder mit der abgefragt werden kann, ob ein Wert vorhanden ist oder nicht.

- Link zur API von Optional:

Hierzu ein komplettes Praxisbeispiel:

```
package de.edlich.gtest;

import java.util.HashMap;

import com.google.common.base.Optional;

public class AvoidNull11 {
```

```
 HashMap<String, Integer> mymap = new HashMap<String,
Integer>();

public static void main(String[] args) {
(new AvoidNull11()).runner();
}

public void runner(){
mymap.put("Butter", 2);
mymap.put("Mehl", 4);
mymap.put("Eier", 12);

System.out.println("Mehl=" + getMapItem("Mehl"));
System.out.println("Schrauben=" + getMapItem("Schrauben"));
   System.out.println("Schrauben im Kuchen? " +
getMapItem("Schrauben").isPresent());
// sollte man natürlich vorher testen...
}

public Optional<Integer> getMapItem(String param) {
          Optional<Integer> possible =
Optional.fromNullable(mymap.get(param));
return possible;
}
}
```

Optionals
Die Verwendung von Optional statt null

1.2 Preconditions

- Die Referenz von Google Guice zu Preconditions ist eigentlich selbsterklärend

Dennoch drei Anmerkungen:

1. Natürlich sollten die Methoden nicht nur für Preconditions, sondern auch für Invarianten habe ich in von Invatianten in Invarianten geändert. Bitte ins Glossar aufnehmen/Erklärung einfügen. verwendet werden und natürlich auch für Post-Conditions (das geht selbst wenn man Optional verwendet.
2. Wie beim Mocken (mit z.B. Mockito) sollten die Tests statisch importiert werden.
3. Jede der nachfolgenden Methoden kann nach Aussage der Dokumentation auch noch mehr Argumente beinhalten:

3.1. Basis: checkXY(logischer-Ausdruck)
3.2. Mit Objeckt: checkXY(logischer-Ausdruck, "String oder Objekt das in die Exception kommt")
3.3. Mit Formatstring: checkXY(logischer-Ausdruck, Formatstring, [FormatArgumente]*)

Codebeispiel:

```
package de.edlich.gtest;

import static com.google.common.base.Preconditions.*;

public class Preconditions12 {

 public static void main(String[] args) {
 (new Preconditions12()).runner(null);
 }

 public void runner(String myStr) {
 checkNotNull(myStr, "String Args ist null!");
 System.out.println("String.length=" + myStr.length());
 }
}
```

Preconditions
Google Guava Preconditions

Aufgabe

Probieren Sie auch mal die anderen Methoden und Signaturen der Klasse com.google.common.base.Preconditions aus!

2.1 Immutable Collections

Wie wir noch später in der LE funktionale Programmierung lernen werden, haben unveränderliche Datenstrukturen immense Vorteile (siehe dazu auch https://code.google.com/p/guava-libraries/wiki/ImmutableCollectionsExplained]):

- In einer Multi-Threaded Umgebung ist die Arbeit mit diesen Collections sicher.
- Die Verwendung ist relativ effizient
- Andere Bibliotheken können diese Datenstrukturen sicher verwenden.

Dazu ein komplett lauffähiges Beispiel, welches die Kern-Struktur der Guava Dokumentation verwendet:

```
package de.edlich.gtest;

import com.google.common.collect.ImmutableSet;

public class ImmutableDS21 {

 public static final ImmutableSet<String> COLOR_NAMES =
 ImmutableSet.of(
 "red","orange", "yellow", "green", "blue", "purple");

 public static void main(String[] args) {
 (new ImmutableDS21()).runner();
 }

 public void runner() {
 System.out.println(">>" + COLOR_NAMES.toString());
 ImmutableSet<String> EDL_COLORS =
 ImmutableSet.<String>builder().addAll(COLOR_NAMES)
 .add("black").build();
 System.out.println("new>>" + EDL_COLORS);
 }
}
```

Preconditions
Google Guava Immutables

2.6 JavaFX

2.7 Zusammenfassung AJV

3 MET - Software- und Architekturmetriken

Title: Software- and Architecture Metrics

Autor: Prof. Dr. Stefan Edlich

3 MET - Software- und Architekturmetriken
3.1 Überblick und Lernziele MET
3.2 Literatur MET
3.3 Motivation
3.4 Einfache Metriken
3.5 Halstead Metrik
3.6 Code Coverage / Test Coverage
3.7 Regelverletzungen
3.8 JDepend
3.9 Kopplungsmetriken
3.10 Technical Debt MET
3.11 Werkzeuge
3.12 Zusammenfassung MET
3.13 Wissensüberprüfung MET

3.1 Überblick und Lernziele MET

Title: Overview and Goals MET

The "Software Metrics" learning unit is a supplement to the topic of software quality, which also includes the "Testing" learning unit. The point is to make missing software quality or software complexity visible. Violations of rules or problems in the code

are listed and optimally visualized. With this help, developers, architects or project managers can set the course for new quality better.

Lernziele

Lernziele

This session has two objectives:

1. They learn about the importance of metrics as a measure of quality and how to assess them in practice.
2. You'll learn the basics and how to calculate them.
3. You'll learn about code metrics and how to use the tools.
4. You'll learn about architectural metrics and use tools to visualize them.

Gliederung

Gliederung

- After a short introduction to metrics, simple metrics such as McCabe metrics are first treated. After the Halstead metrics and code coverage, the different types of rule violations are discussed, which are aggregated by many tools as metrics.
- In the second part, JDepend, coupling metrics and the question of guilt - the Technical Debt - follow.
- Finally, current available tools for Java are listed and compared.

Zeitumfang

Zeitumfang

It takes about 120 minutes to work through this session. The application of a tool such as XRadar, Sonar or SonarJ to an existing project takes three hours each.

3.2 Literatur MET

Links

- The Classis: http://www.laputan.org/mud/
- http://akvo.org/blog/the-ball-of-mud-transition/
- http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/
- http://blog.sei.cmu.edu/post.cfm/field-study-technical-debt-208

3.3 Motivation

Metrics is anything measurable. This refers to **measures or ratios**, which occur in all fields such as music, physics, mathematics, etc.

Definition

Softwaremetric
A software metric is a measure that says something about the software.

Definition

"A software metric is a function that maps a software unit into a numerical value. This calculated value can be interpreted as the degree of fulfillment of a quality characteristic of the software unit."
(IEEE Standard 1061,1992)

Beispiel

A simple metric would be, for example, the number of code lines or the number of classes in an OO project. But of course, the topic of metrics goes much further - right up to architectural metrics that tell us something about the state of the software system's architecture.

Since a good software architecture means maintainability, changeability and testability, this topic is also referred to as "software stability". It describes how stable the software system reacts to changes, i. e. to its further development.

Basically, there are many types of metrics related to software development. A software project consists of many elements, for example:

- Sourcecode
- Binary Code
- external Libs
- Buildinformation
- Versioning Information
- Forms of meta- and additional Information (e.g. Annotations, JavaDoc, etc.)

This also means that many metrics can be measured. In this session we will look at some of them, such as:

- Complexity metrics
- Test metrics
- Architectural metrics
- Coupling metrics
- Style metrics

In computer science there are more types of metrics than just software or code metrics. Many metrics are particularly well known from project management, such as the function-point metrics or COCOMO for effort estimation.

We will first concentrate on the simple software metrics and then get to know the architecture metrics.

Hinweis

But: Good metrics are not synonymous with good quality. Good quality does not necessarily mean good metrics!

Experience from many projects also shows that the metric size and quality characteristics correlate quite often with each other!

As today's large projects are being developed and quality has to be checked, the visualization of quality metrics is becoming more and more important. This will be dealt with towards the end of the module.

Vertiefung

Interview with Jonathan Aldrich on the topic of Static Analysis (45 min):

http://www.se-radio.net

3.4 Einfache Metriken

Title: Simple Metrics

It is often helpful to have an overview of very simple code metrics. These include the following metrics:

Lines of Code

The number of lines of code - also known as LOC - is a known measure when it comes to comparing project sizes. For example, checkstyle has 22,369 lines of code and Apache Tomcat already has 159,364 lines of code, which makes it one of the larger projects. It is important to know whether all lines of code are included. If all file lines are included, Tomcat will have 314.461 lines, which can be due to documentation or configuration files.

Code lines can have completely different expressions in different languages. This is the expressiveness of a language. 1 is often referred to as C, modern dynamic languages often have an expression of 7 and more. This means that coding can possibly be done on average seven times as efficiently.

For example, compare the code line in Clojure with a possible implementation in an old basic dialect:

```
(filter (fn [x] (= 1 (rem x 2))) (map (fn [x] (* x x)) (range
 10)))
```

Hinweis

The function returns all odd squares between 0 and 100. For code lines, the expressiveness of the language should always be taken into account.

Comments

Number of lines describing the program. As a rule of thumb, this number should be between 30% and 60% of the code lines.

Methods/ Functions

The number of methods or functions of a program or project.

Classes

The number of classes used is of course also a measure of complexity. Around the year 2000, Germany's largest Java software project (Cheops) had about 30,000 classes, which is quite a lot.

Packages

The number of packages or namespaces are simply calculated here.

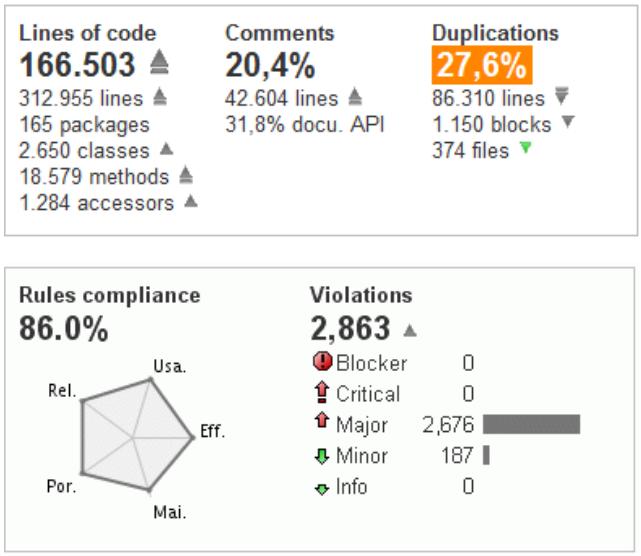
Files

Files are self-explanatory, but unfortunately they do not provide a good measure for the complexity of a software. For example, there are Ruby programs that are packed in a file but are still very complex (e. g. Kirbybase).

Duplications

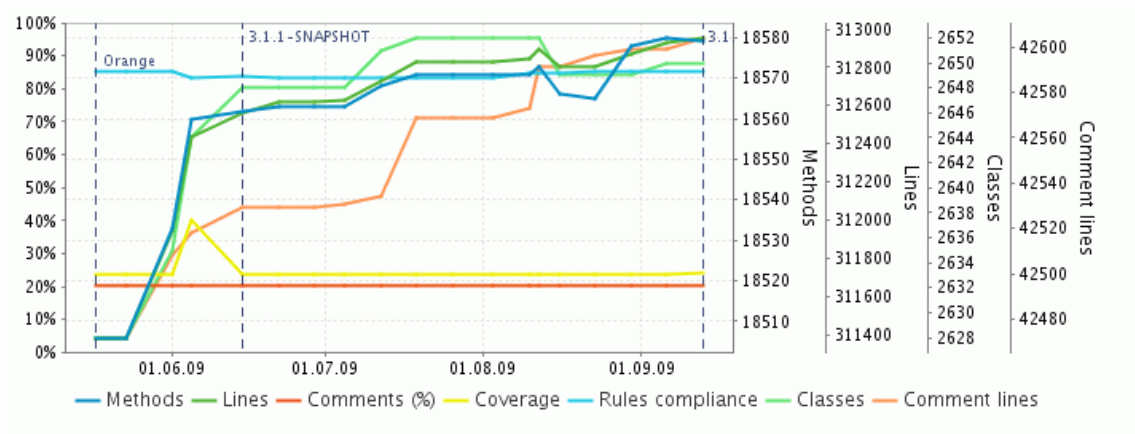
Some good tools search the code for similar building blocks. This can be done on different levels that search for identical lines of code, similar blocks or even similar files. In addition, this provides good information as to whether refactoring could be carried out or not. The DRY principle (Don't Repeat Yourself) was often violated.

For example, the same blocks can be combined in one method, which usually increases the maintainability of the code considerably. Here is an example of an automatic analysis of basic metrics:



Analysis of Basic Metrics

It is helpful if all metrics in the history can be tracked. With good tools, this is easily possible and you get graphically prepared output like the following figure.



Basis Timeline

Some tools also offer a graphical overview in which the metrics are displayed as weighted areas. With a click on the surface (similar to the tools for hard disk usage) you can "drill" yourself into the region or the error.

This list will never be complete. In the code there will be more and more characteristics that can be examined or transformed into metrics (number of to-do marks, annotations, etc.).

3.4.1 McCabe Metrik

Title: McCabe Metric

One of the best-known metrics is the McCabe metric named after Thomas J. McCabe. This is a pure code metrics that is relatively easy to determine. The idea is to automatically determine how complex the code of a software unit (e. g. method or class) is, and this can be done over 10,000 classes.

The basic assumption is that complexity can be calculated by the number of potential pathways that can be taken in a program. In this case, this means how many control flows are possible in the program graph.

Definition

McCabe Cyclomatic Complexity

Every conditional construct (whether it's a loop or an if/else/switch) increases the complexity by 1 (therefore it should be called Conditional Complexity).

The equation is:

Formel

$$M = E - N + P$$

Legende:

- M = the Cyclomatic Complexity,
- E = the number of Edges in the graph,
- N = the number of nodes in the graph and
- P = the number of exit points (return, last command, exit, etc.)

As a rule, exit point P is reconnected to the entry point and counted as an edge.

```
if(a1) code1 else code2 end if(b2) code3 else code4 end
```

Aufgabe

Cyclomatic Complexity How high is the Cyclomatic Complexity in the example above?

Draw the graph!

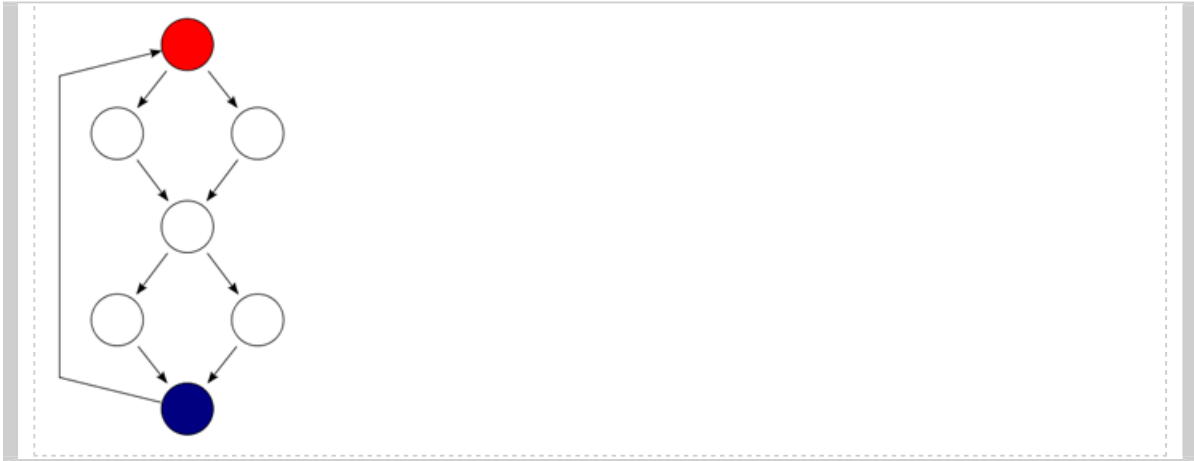
Try to solve the problem yourself before looking at the sample solution.

Time needed: 10 min

Solution

M = 9 - 7 + 1 = 3

Usually, code with a higher McCabe complexity also has a lower cohaesion (see following chapters).



3.5 Halstead Metrik

Title: Halstread Metric

One of the lesser-known metrics is the Halstead metric. Nevertheless, it is implemented in many metric tools because it is easy to calculate by machine. It is also a static process in which the code is not executed, but the source code is analyzed.

This metric can be calculated as follows:

- $\mathbf{n_1}$ => the number of different' *operators* used in the code. So these are orders like that:

```
! != % %= & && || &= ( )
* *= + ++ +=
, - -- -= ->
. ... / /= : ::
< << <<= <= = == > >= >> >>=
? [ ] ^ ^= { } | |= ~ (für C++)
```

and break, case, return, try, but also private, friend, static. But no types.

- $\mathbf{n_2}$ => the number of different' *operands* used in the code.

So type declarations (int, bool, void) and all constants, type names and identifiers that are not reserved names.

- $\mathbf{N_1}$ => the total number of operators in the code.
- $\mathbf{N_2}$ => the total number of operands in the code.

From these four values, the following figures can then be calculated:

Program length:	$N = N_1 + N_2$
Amount of the vocabular:	$n = n_1 + n_2$
Halstaed-Volume:	$V = N * \log_2 n$
Halstaed-Lengh:	$L = n_1 * \log_2 n_1 + n_2 * \log_2 n_2$
Difficulty:	$D = (n_1 / 2)* (N_2/n_2)$
expense:	$E = D * V$

measure figures

The sources even list an estimate for the number of expected bugs.

This metric provides only lexical dimensional data. The actual complexity is therefore only estimated. In particular, dynamic programming languages that introduce or reduce extreme complexity with constructs such as currying, fibers, yield and closures are certainly avoiding the attempt to measure complexity accurately.

However, the absolute number is not necessarily relevant. Whether one module has a D of 65.25 and another module has a D of 67.78 is not so important. It is more important to get a feeling for these numbers and to be able to correctly evaluate the exceeding of orders of magnitude as an alarm signal.

Vertiefung

http://www.virtualmachinery.com/sidebar2.htm

Artikel: Komplexität und Qualität von Software von XAVIER-NOËL CULLMANN und KLAUS LAMBERTZ verifysoft.com mscoder.pdf (737 KB)

3.6 Code Coverage / Test Coverage

There are two different areas of coverage:

1. The cover where the code will pass through during the expiration.
2. The test coverage: which controls/checks how well the tests, for example, to record the methods used.

If coverage is spoken of, software engineers usually refer to test coverage. Nevertheless, one should specify exactly what is meant.

Code Coverage

How often a method or piece of code is run through is also investigated by means of a metric. This can be very useful, for example, to check processes or to debug.

Beispiel

Example from the documentation of the programming language D

Here is an example from the documentation of the programming language D, which has built such a feature directly into the compiler. After executing the program, defined code points are output with the specification of how often the line was run through:

```
|/* Eratosthenes Sieve prime number calculation. */
|
|import std.stdio;
|
|bit flags[8191];
|
|int main()
5|{ int i, prime, k, count, iter;
|
1| writefln("10 iterations");
22| for (iter = 1; iter <= 10; iter++)
10| { count = 0;
10| flags[] = true;
163840| for (i = 0; i < flags.length; i++)
81910| { if (flags[i])
18990| { prime = i + i + 3;
18990| k = i + prime;
168980| while (k < flags.length)
| {
149990| flags[k] = false;
149990| k += prime;
| }
18990| count += 1;
```

```
 | }
 | }
 | }
 1| writefln("%d primes", count);
 1| return 0;
 |}
sieve.d is 100% covered
```

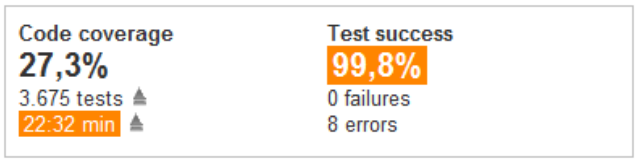
(Quelle: http://www.digitalmars.com/d/2.0/code_coverage.html)

Test-Coverage

Test Coverage has already been addressed in the TST Testing session. It is an important metric that often refers to unit tests - but of course tests can also refer to other units:

- Code blocks - have ifs, switches or loops covered or tested code blocks?
- Have larger units such as classes or packages been sufficiently tested? If so, what percentage?

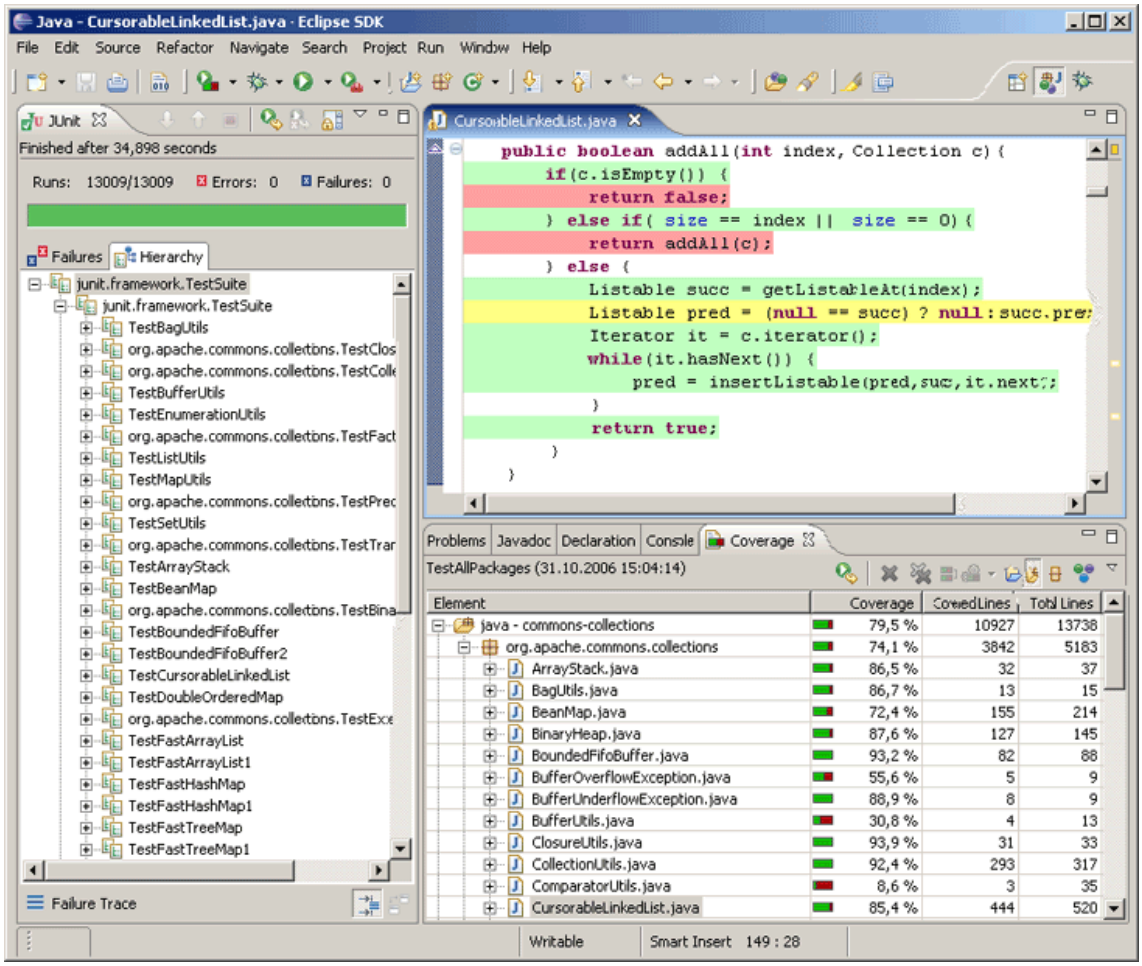
Such tools can be ideally built into a build management tool (Ant, Maven) and then provide meaningful reports:



Code-Coverage Report

While working on the code, such a metric can be very helpful. Therefore, there are plug-ins in every known development environment that provide these results.

For example, the inclined student is advised to install Emma or the eclemma (http://www.eclemma.org) plug-in. This can be done in a few minutes and you will get the first readings as shown in the picture:



Coverage-Metrik in Eclipse

3.7 Regelverletzungen

Title: Rule Violations

For many years, there have been excellent tools for checking code and programming style. Violations of the rules are usually displayed as aggregated metrics. The developer can list them individually and then edit the problems afterwards.

There are many types of rule violations, such as violating the framework of a style guide or violating rules that the community or you have defined as valid.

For Java are the most popular tools:

- Checkstyle
- PMD and
- Findbugs

An overview of other tools can be found on the Internet at:

http://java-source.net/open-source/code-analyzers

Here are some examples of what can usually be checked by the tools:

Documentation in "JavaDoc":

- Is there a package.html?
- Is the JavaDoc Code well formatted?
- Are methods / Variabls well commented?

Java **Naming Coneventions**:

- Do they conform to the naming conventions regexp "^[a-z]+(\.[a-z][a-z0-9]*)*$"?

Is the "header" and "imports" okay?

- Avoidance of imports with asterisks ("de. vfh. swt. *")
- No specification of wrong imports
- No redundant imports
- No unused imports

Specification of "size restrictions":

- The number of executable code pieces can be limited (e. g. the number of methods)
- File size and number of lines can be limited
- The length of the method can be limited. Note: This test forces the refactoring Extract Method.
- A limit for the number of parameters can be set.

Check for WhiteSpaces or tabs,

Check the **order** of the **Modifier** in classes: For example, for Java the following applies

1. public
2. protected
3. private
4. abstract
5. static
6. final
7. transient
8. volatile
9. synchronized
10. native
11. strictfp

There must be no "empty blocks".

Many other "coding standards" are checked:

- Avoidance of empty statements.
- Local variables that should not change their value should be final.

Identical naming within blocks (shadowing) must be avoided.

- Don't use Magic-Numbers. Avoid the use of superfluous and therefore ineffective numbers (except -1,0,2) if possible, and use primarily declarations in final variables that can be supplemented by explanations.
- Missing default in switch construction
- Control variables should not be modified

```
for (int x = 0; x < 1; x++) {x++; }
```

- Unnecessary throws-declarations
- Find unnecessary code:

```
if (b == true), b || true, !false, etc.
```

- Correct name definition for JUnit 3 tests.
- Are there equal strings in the code?

A good class design can also be checked:

- Visibility of class variables
- Use final
- Is the class geared to inheritance?

It is not only possible to set any regular expressions for the analysis of a line. Any number of rules can be implemented and extended.

Beispiel

" *Checkstyle example for inefficient code*'
As you can see here, today's tools also find inefficient code. Here is another example from Checkstyle:

```
if (valid())
 return false;
else
 return true;
```

This example should be replaced with:

```
return !valid();
```

Some tools such as Checkstyle contain the following metrics:

1. Number of **boolsch equations** as &&, ||, &, | und ^ in a command block
2. **ClassDataAbstractionCoupling**: Hint: This is the **Ce** from JDepend!
3. **ClassFanOutComplexity**: Hnt: This is the **Ca** from JDepend!
4. **CyclomaticComplexity**: Well we know this.
5. **NPathComplexity**: The number of possible processes. This corresponds to the McCabe metric.
6. **JavaNCSS**: Number of lines of code. With restrictions on method=50, Klasse=1500, FileMax=2000

Question: Which values do you think are useful for point 6 and the other metrics?

Résumé:

The power and usefulness of style analysis tools should not be underestimated for code and system quality. As an experienced developer, you're usually the first to be hit in the head when many thousands of error messages from these tools are displayed at the beginning. Often the wish to "comment out" some tests in the configuration file appears, which turns them off. It would be better to stop here and think about this carefully. Usually it is better to edit the cause of the error in the code rather than reducing the error check.

Tools such as checkstyles should always be included in the build initially or, if necessary, run as a plugin at the touch of a button. This can be difficult at the beginning, but in the long run this approach pays off, especially with large projects. Particularly because style analysis tools also recognize things that promote good architecture.

Beispiel

Practical Example for Checkstyle

The following shows the configuration and results of Checkstyle. At the beginning, you should consider whether this should be done under ANT (or a build management system) or as a plug-in in your preferred IDE.

- Download of **checkstyle-all-*.jar**
- Declaration of the task:

```
<taskdef resource="checkstyletask.properties"
 classpath="lib/checkstyle-all-5.0-beta01.jar" />
```

- Definition of the targets:

```
<target name="CHECKSTYLE">
 <checkstyle config="docs/sun_checks.xml">
 <fileset dir="src" includes="**/*.java" />
 <formatter type="plain" />
 <formatter type="xml" toFile="checkstyle_errors.xml" />
 </checkstyle>
</target>
```

- Including the analysis definition. E. g. sun_checks. xml (possibly take out the things that don't work...). The file has approximately the following content:

```
<module name="Checker">
...
 <module name="AvoidInlineConditionals"/>
 <module name="DoubleCheckedLocking"/>
 <module name="EmptyStatement"/>
 <module name="EqualsHashCode"/>
 <module name="HiddenField"/>
 <module name="IllegalInstantiation"/>
 <module name="InnerAssignment"/>
 <module name="MagicNumber"/>
```

```
 <module name="MissingSwitchDefault"/>
 <module name="RedundantThrows"/>
 <module name="SimplifyBooleanExpression"/>
 <module name="SimplifyBooleanReturn"/>
...
```

As you can see here, concrete tests are defined for test terms. All tests are usually well documented and well-founded (http://checkstyle.sourceforge.net/checks.html).

Analysis of the result file checkstyle_errors. xml or the output on the console.

The XML file then looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<checkstyle version="5.0-beta01">
    <file    name="C:\EDLCHESS\EDLENGINE\src\de\edlchess\cmds
\PositionCommand.java">
 </file>
   <file   name="C:\EDLCHESS\EDLENGINE\src\de\edlchess\consts
\Info.java">
 </file>
 ... and more files...
 <error line="44" column="35" severity="error"
 message="&apos;71776119061217280L&apos; should be defined by
 a constant."
source="com.puppycrawl.tools.checkstyle.checks.coding.MagicNumberCheck"/
>

 <error line="10" column="9" severity="error"
 message="Javadoc-Comment missing."
source="com.puppycrawl.tools.checkstyle.checks.javadoc.JavadocMethodCheck"/
>

 <error line="10" column="29" severity="error"
  message="The parameter arenaCmd should be declared as
 &apos;final&apos;."
source="com.puppycrawl.tools.checkstyle.checks.FinalParametersCheck"/
>
```

```
<error line="33" column="9" severity="error"
message="The Method &apos;identify&apos; has not been designed
for inheritance - must be abstract, final or empty."
source="com.puppycrawl.tools.checkstyle.checks.design.DesignForExtensionCheck"/
>
```

Vertiefung

Sun Code Conventions Checkstyle PDM Findbugs

3.8 JDepend

One of the oldest and most famous metrics was developed in the 90s by Mike Clark. He provided a tool for code analysis, which can be found in many other frameworks, IDEs and tools in many languages.

The word Depend is already contained in the term JDepend, which is translated as Dependency. It is therefore primarily a question of analysing the dependency of the components of a mostly larger software system.

According to Mike Clark, JDepend's task is to do the following:

JDepend traverses a set of Java class and source file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to effectively manage and control package dependencies. Package dependency cycles are reported along with the hierarchical paths of packages participating in package dependency cycles."

Again, the source code has to be parsed and metrics have to be extracted from each class, which are presented below. These metrics allow you to measure and assess the extensibility, reusability and maintainability of the code. Better said, the metrics may indicate that the code may have is not easily expandable, reusable or maintainable.

In the field of research of software technology, the term "software stability" is also used in this context, i. e. how stable the software as a whole is.

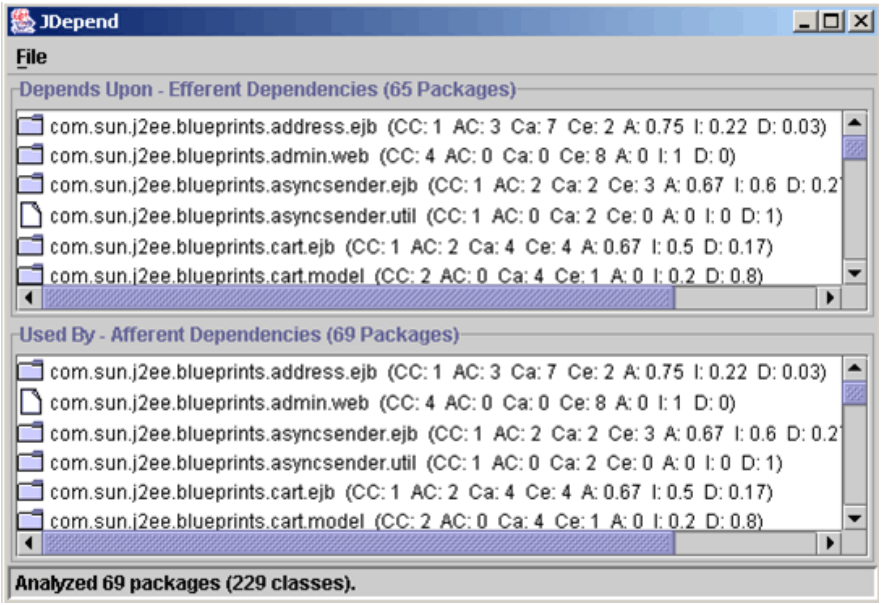
What are the objectives of the JDepend metrics?

1. JDepend promotes Design by Contract. It motivates you to develop stable packages - in the form of interfaces or abstract classes. The goal is that the unstable (frequently changing) classes are dependent on stable classes and not on other concrete

implementations. In this case, different people can also program in parallel / simultaneously against stable interfaces.

2. The independence of the packages is thereby promoted. This applies to both your own packages and external packages. The dependencies to external packages (e. g. commons. jar) can be made visible and, if necessary, isolated. This means that the external functionality can be encapsulated with stable components (e. g. interfaces or abstract classes for facades on the library). Independent, internal packages can also be developed much better in parallel - possibly even in a separate release cycle.
3. JDepend makes cycles visible. This will be discussed in more detail later. Cycles are components that depend on each other cyclically. These can be classes or packages, for example. For the changeability and maintainability of code, practical experience shows that cycles are "deadly".

Let's have a look at the metrics generated by JDepend:



Ausgabe der JDepend-Metriken

As you can see, JDepend initially refers to packages, that is, package dependency analysis. However, the view can be enlarged and reduced in size, i. e. you can also analyze at class level. As a rule, however, large projects are usually carried out at package level.

3.8.1 JDepend-Metriken MET

Title: JDepend Mectrics MET

Let's look at some of the metrics of JDepend.

The total number of concrete classes, abstract classes and interfaces:
This number alone is an indicator of whether the package is already too large, for example.

Dependency/Afferent Couplings (Ca):. Afferent couplings, which can be translated as "bringing", provide information about how many other packages depend on the package currently being considered. To put it bluntly: how many other packages will it "bang", if I change something in this package. Of course, the aim is to keep this figure as low as possible. J. Clark even writes here about the responsibility of this package for its services to others.

Efferent Couplings (Ce): Leading out / leading away "(e. g. from the biology of nerve cells). Here you can specify how many other packages my classes need from the packages considered. To put it bluntly: how sensitive I am when other classes change. How strong will it bang when other classes change? This seems to be a measure of the package's independence.

Abstractness (A): This number reflects a ratio and the values are between zero and one (including). A is defined as AC/ (CC+AC). This means: What is the ratio of abstract classes to all classes? Therefore, a package with metric 1 is a completely abstract package, which only defines but does nothing really. If the metric is zero (which unfortunately is too often the case), there are only concrete classes. In connection with the instability applied to axes, the points of the packages would then "stick"to the axes.

Instability (I): The instability is a little more interesting. It is calculated according to the formula I = Ce / (Ce + Ca). But what does this mean? The denominator is the total number of all dependencies aka Ce + Ca. So both how sensitive other packages are to changes in me and how sensitive I am when other packages change (where I am the package under consideration). Here a 0 would be a stable package and a 1 would be a completely unstable package.

Distance from the Main Sequence (D): This means distance to the main line. The main line is an idealized line with A+I=1 (see also pictures on the next page). This value is attributed to the property of describing how much the package is balanced between abstractness A and stability. Optimally balanced would be a package which lies exactly on this line, i. e. the distance d is minimized. The completely abstract package A=1 is also completely stable (I=0). On the other hand, a package that is only concretely very unstable, because it does not depend on interfaces or abstract classes.

Package Dependency Cycles: The output of cycles is also very interesting and important. If packages contain many cycles, class changes cause chain reactions to further changes. The developers are then only debugging and are not making any progress as regards content. An often observed phenomenon in many companies.

There are many cases of cycles:

- Cycles can be between classes. As a rule, classes of different packages are considered here. However, cycles between classes within a package are also dangerous. For example, a class X from package A requires a class Y from package B. This is a "direct cycle".
- Cycles can also occur between**'' more than two**' different packages.
  Z. B. A -> B -> C -> A. This means that all three classes are involved in one cycle.
- Finally, there are indirect cycles. If there is still a class P in the above example that requires package A, it is also indirectly dependent on this cycle and is marked accordingly by JDepend / Analysis tools.

3.8.2 Zyklenanalyse als Teil des Buildzyklus

Title: Cycle analysis as part of the build cycle

Tools that determine such metrics should of course be included in the build cycle to be available to developers, project managers or quality assurance within the framework of the continuous integration process. This means that the entire JDepend process (or other tools that determine the same or similar metrics) can be started and, for example, a *. txt report can be generated for shipping. Calling JDepend from build management tools such as ANT is therefore relatively simple:

```
<target name="jdepend">
 <jdepend outputfile="docs/jdepend-report.txt">
 <exclude name="java.*"/>
 <exclude name="javax.*"/>
 <classespath>
 <pathelement location="build" />
 </classespath>
 <classpath location="build" />
 </jdepend>
```

```
</target>
```

Source: http://clarkware.com/software/JDepend.html

Beispiel

Some other tools allow you to insert the cycle search as JUnit tests. Here is an example:

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class DistanceTest extends TestCase {

 private JDepend jdepend;

 public DistanceTest(String name) {
 super(name);
 }

 protected void setUp() throws IOException {
 jdepend = new JDepend();
 jdepend.addDirectory("/path/to/project/ejb/classes");
 jdepend.addDirectory("/path/to/project/web/classes");
 jdepend.addDirectory("/path/to/project/thirdpartyjars");
 }

 /**
 * Tests the conformance of a single package to a
 * distance from the main sequence (D) within a
 * tolerance.
 */

 public void testOnePackage() {

 double ideal = 0.0;
```

```
 double tolerance = 0.125; //project-dependent

 jdepend.analyze();

 JavaPackage p = jdepend.getPackage("com.xyz.ejb");
 assertEquals("Distance exceeded: " + p.getName(),
 ideal, p.distance(), tolerance); }
 /**
 * Tests the conformance of all analyzed packages to a
 * distance from the main sequence (D) within a tolerance.
 */
 public void testAllPackages() {

 double ideal = 0.0;
 double tolerance = 0.5; //project-dependent

 Collection packages = jdepend.analyze();

 Iterator iter = packages.iterator();
 while (iter.hasNext()) {
 JavaPackage p = (JavaPackage)iter.next();
 assertEquals("Distance exceeded: " + p.getName(),
 ideal, p.distance(), tolerance);
 }
 }
 public static void main(String[] args) {
 junit.textui.TestRunner.run(DistanceTest.class);
 }
}
```

Source: http://clarkware.com/software/JDepend.html

In the example, the sources are read in the upper area, then JDepend is called completely and the system checks whether cycles are included. These can then be output completely using data structure p. On the JDepend website there are even more examples of how to test for d, for example (you will find an explanation later on). There are tools available on the market for many languages that do this (such as Macker for Java).

The following figure shows how instability has been measured many years ago in community tools using javaforge.com.

SCM Commits | Browse Repository | Browse Commits | SCM Settings

Yesterday & Today	This Week	Last 30 Days	All
3	3	3	3

Source Code Summary

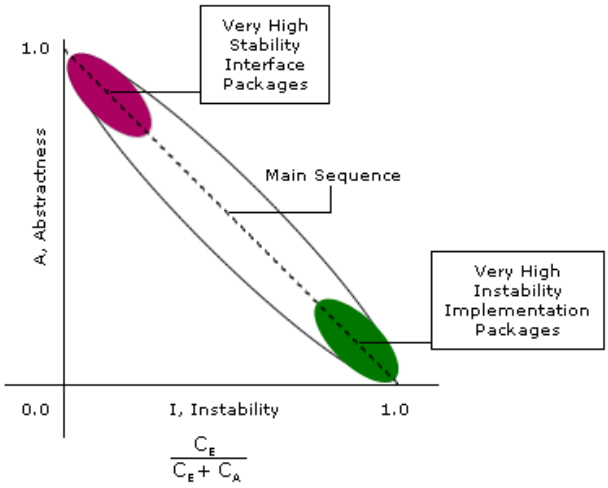
Category	Files	Lines	Bytes
Summary:	0	0	0

Source Metrics Summary

Action	Description	Value
Trends | Details	Package instability ratio	0%
Trends | Details	Percentage of duplicate code files	0%
Trends | Details	Total number of duplicate lines	
Trends | Details	Total number of violations	
Trends	Total number of copy paste blocks	
Trends | Details	Total number of packages	
Trends | Details	Number of checkstyle errors	

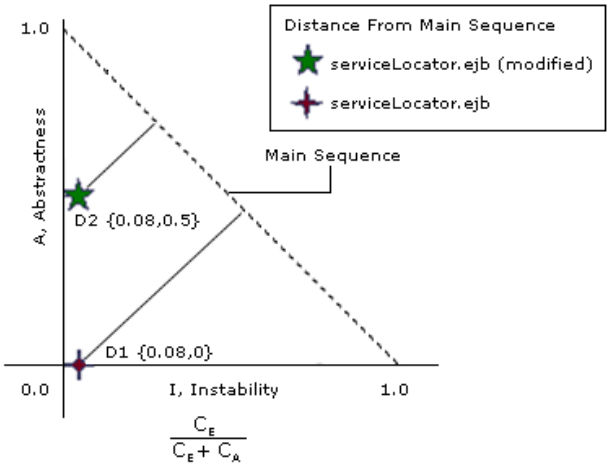
Measurement of the Instability

Analysis of the graphics



Main Sequence Graphics

In the first figure you can see the Main Sequence, which traces the line **A+I**. This shows that packages cannot be completely abstract (' *A=1*) and completely stable (*I=1*) at the same time. There should always be a balanced amount of concrete classes on the one hand and abstract classes / interfaces on the other.



Distance from Main Sequence

(Star: Class A) de.vfh.swt.metriken.cpool; Cross: B) Clas sde.vfh.swt.metriken.media)

In the second picture we see a package A, which is quite far left upside. From this it can be concluded that there are obviously many abstract classes and a high stability (I is small).

The second point has few abstract and too many concrete classes. There is a high instability here (I is large). From all points a d, i. e. the distance to the "Main Sequence", can now be calculated. Afterwards, this should be considered individually and summed up if necessary.

Packages with low abstraction should depend on packets with high abstraction!

In an ideal world, there would only be implementation classes that depend on abstract classes. In the real world, however, you have to make compromises...

JDepend measures the distance of the package to the "Main Sequence" D.

Hinweis

> The greater the distance D between the package and the main sequence, the greater the probability that the package can tolerate a review or refactoring.

These are, of course, only indications that need to be examined in concrete terms and, in special cases, can be devoid of any basis. Nevertheless, experience shows that packages with a large D can certainly tolerate refactoring. Similar to the indication of good test coverage.

3.9 Kopplungsmetriken

Title: Coupling Metrics

Another very interesting metric area is the coupling. This refers to the degree to which components are linked with each other.

There are two different areas:

Cohesion: translates approximately context / binding power

Coupling: is about coupling / connection. Often referred to as the dependency.

Cohesion examines, for example, the methods of a class / component and how they relate to each other. The goal is that one class / component performs a task and not several. If this is the case, there is a high degree of cohesion. These classes are more maintainable, changeable and of course easier to understand. On the other hand, there is usually a small coupling on the other hand.

The opposite example would be a class with low cohesion. Here, the class and its methods would fulfil many tasks. Most of the time there is a large coupling, since more external components need this class. The present component is therefore much more difficult to change and understand. Refactoring should redesign the class so that it only fulfils one task.

A core element of cohesion is whether the component can be meaningfully cut into independent parts and whether the set of both parts provides the same service as before.

Further material is available on this topic here:
http://www.aivosto.com/project/help/pm-oo-cohesion.html

3.9.1 Coupling / Kopplung

Title: Coupling Example

There are different types of measurement with the coupling, because you can look at different coupling variants and these can be weighted differently. For example, the communication of the modules such as data exchange, parameter transfer, the use of further data, etc. can be considered.

Here is one example.

Pressmann *Pr09* gave the idea to do the following:

- d_i: Amount of Input Paramters Data
- c_i: Amount of Input Parameters Control Structures

- d_o: Amount of Output Parameter Data
- c_o: Amount of Output Parameter Control Structures
- g_d: Amount of global used Data
- g_c: Amount of global used Control Structures
- w: Amount of called Modules (this is the Ce from our JDepend)
- r: Amount of Modules, that call the current module (this is the Ca from our JDepend)

This metric also considers control data, which means that when a module transmits control information to another module, this is weighted more strongly than when only simple processing data is considered.

Pressmann defined coupling like this:

Formel

$$C = 1 - 1/(d_i + 2*c_i + d_o + 2*c_o + g_d + 2*g_c + w + r)$$

A side-effect free method would be a method with an input parameter (d_i=1) and an output parameter (d_o=1) such as the calculation of an interest value. Since this is called by a module, (r=1) the denominator must be three in the above formula. As a result, the result is usually a minimum C of 0.67. Of course, this value does not exceed one.

Beispiel

Let's take another example:

There are three input parameters and one output parameter, a control parameter and no global variables are used. The module uses two other modules and is called at four other places in the code.

It follows:

$$C = 1 - 1 / (3 + 2*1 + 1 + 2*0 + 0 + 0 + 2 + 4) = 0{,}917$$

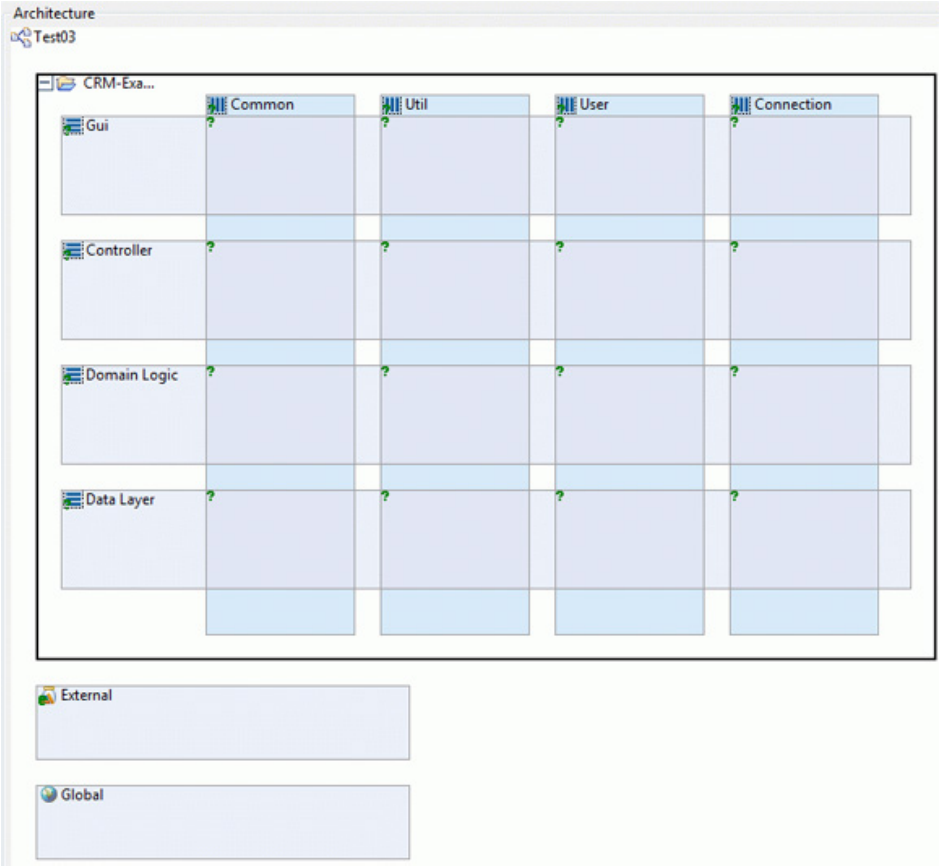
Surely, one can argue about whether the given weighting is reasonable. Other metrics might weight Ca and Ce more heavily than simple input parameters (data).

3.9.2 Die ACD Metrik

Title: The ACD Metric

Another - perhaps even more accurate - metric is measured by SonarJ and comes from John Lakos[*Lak96*]. We will dig a little deeper here.

An architectural approach has already been mentioned in which components were entered into a grid and logically grouped. For example, an axis can refer to the layer model: show the view at the top and the data retention layer at the bottom. Additional layers or logical domain objects can be placed on another axis. Let's take a look at the following figure.



Definition of an architekture

For architects, it is important to know how the "call flow" is determined at this point. Can each component call up any other component according to the "spaghetti style"? Or is a top-down with left-to-right permitting much more meaningful? Of course, the latter makes more sense, which is why there are many tools - such as SonarJ or Macker - that test this kind of thing.

If you take out a component and look at which other components are called up, you can create a graph. This graph can refer to packages in large systems as well as classes in small systems.

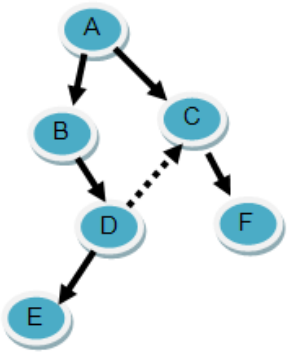
Hinweis

As already explained in the session Design, the following shows that packages should not access each other unstructured. Access to Package B from Component A (or, for example, Package en.vfh.swt.A) is better done via one or a few interfaces (quasi Facade Interfaces), which provide the functionality bundled together.

The recorded graph is itself already a measure of the degree of coupling. A large graph with many cycles is logically much harder to maintain and change, since many more code points are affected. Therefore, a small and cycle-free graph is always useful.

Beispiel

Calculation of the ACD metric How is the ACD metric calculated? Let's look at an example:



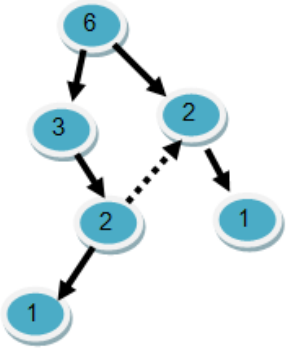
Component Graph

The figure shows some components from A to E, which can represent both classes and packages. A uses B and C. B uses D and indirectly also E. C uses F. In the first variant there are no cycles. In the second variant, D also uses C again, which creates a cycle.

ACD stands for Average Component Dependency and describes the mean component dependency according to Lakos. However, there is no distinction between control and data flow.

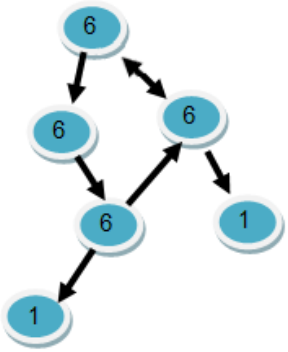
After the ACD method, each component is initially assigned an A since it implicitly depends on itself. Paint the above graph on paper and enter the numbers. Then you start from the bottom and add the sum of the underlying / dependent numbers in each component.

The following result is obtained for the first cycle-free variant:



The ACD metric is calculated by dividing the sum of all values by the number of nodes! In our example, **ACD = 15 / 6 = 2.5**. This means that on average, each node depends on itself (=1) and another 1.5 nodes.

If we now add a critical cycle from D to C and from C to A, the numbers change significantly. We obtain the following graph:



We can see here that in the cycle, each account actually depends on each other. The circle in question therefore receives a value of 6 on each node. If the new mean dependency is calculated, a completely different impression is created: "ACD = 26 / 6 = 4.33". This is a much higher value, which certainly indicates refactoring.

What happens when you compare two graphs? Let's take our graph above with a cycle and a value of 4.33. We want to compare it with a graph that looks similar, has 100 nodes but does not have a cycle.

In this case, the 100knotige graph might have a much higher ACD value, simply because of the absolute number of nodes. Nevertheless, this graph could be "cleaner" than the lower-order small cycle graph. To obtain comparable values, we have to divide by the number of nodes again. Only then do we get a relative value that is independent of the number of nodes. So we call it "rACD" and define it:

ACD = number of dependencies / number of nodes **rACD** = ACD / number of nodes

We obtain:

Graph 1: rACD1 = 2,5 / 6 = 0,4167 Graph 2: rACD2 = 4,33 / 6 = 0,7267

It shows that a maximum value of 0.5, i. e. 50%, can be reached for cycle-free graphs. However, a lower value would be better.

Übung MET-01

Cycle free graphs

Use examples to illustrate why it is impossible to achieve a value greater than 50% in cyclic-free graphs as the number of nodes grows!

Processing time: 15 minutes

Experience has shown that a rACD value for larger graphs is a very likely indication of cycles in the graphs considered.

The sonar team (after Zitzewitz, Java-Magazin 2.2007, p. 30) comes to the conclusion that further formulas can be derived from experience, which provide good values for rACDs - depending on the node size:

At 200 knots, the rACD should be less than 15%. For 1000 knots, the rACD should be less than 7.5%.

The following useful connection can be established:

```
<FORMEL>
 rACDmax=1,5*(1/(2^log5(n))) where n > 200
</FORMEL>
```

3.9.3 Zusammenfassend

Title: Summary

The coupling metrics presented here automatically provide a good overview of the "interlocking" of components. Cycle-free components with the lowest possible rACD are helpful.

These metrics are quickly determined in modern tools and can be displayed in suitable colours. Developers and architects can immediately see which packages or components are critical so that refactoring can be carried out.

Ideally, such metrics should be used right from the start and sound the alarm with the tools, as this promotes the natural growth of a good architecture.

3.10 Technical Debt MET

Title: Technical Debt

Back in 1992 Ward Cunningham came up with the idea of describing the number of errors, rule violations and complexity as **Technical Debt**. Freely translated, this means that you are guilty because something can be improved on the technical side.

There are several reasons for this debt. For example:

- bad architecture / design
- many code metric violations
- too much complexity
- too large coupling
- cyclic dependencies

Some errors may be easier to fix. However, there is also an intrinsic complexity that is usually more difficult to resolve or reduce. Other problem areas are of a long-term nature (architecture) and some are of a short-term nature (missing JavaDoc), as this can be solved more easily.

The entire problem addresses a topic that is often found in industry. There is the pressure to get a product onto the market quickly. Therefore, a solution " *Quick-and-Dirty*" is implemented. The product is shipped with its guilt, including the dirty part that is in the code. A loan has been taken up in the same way as in Financial Accounting (debt), and now interest is to be paid. Interest rates are paid at a higher or lower level because the product - as a rule - has to be further developed. However, the higher the number of injuries, the harder it will be to develop and modify the program. For debugging, the developers now have to invest much more time.

The ideal goal for the software vendor is to deliver the product with a debt of zero. The buyer's expectations are to buy a product with the lowest possible technical debt.

This topic is often discussed in forums and other forums. It is also interesting that modern tools try to express the debt as monetary value. This means that all metrics are converted into man-days. How many man-days are needed to eliminate the number of code and architecture metrics errors? Man-days can be converted into monetary value - depending on the wage level.

Hinweis

The decisive factor is that software costs are not only made up of development costs, but that they also result in much higher maintenance costs, training costs, change costs, etc., which have to be taken into account!

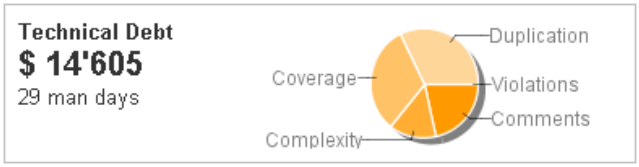
This is done with the Sonar tool as follows:

Debt (in man days)	cost_to_fix_duplications + cost_to_fix_violations + cost_to_comment_public_API + cost_to_fix_uncovered_complexity + cost_to_bring_complexity_below_threshold

Where we have:

Duplications =	cost_to_fix_one_block * duplicated_blocks
Violations =	cost_to fix_one_violation * mandatory_violations
Comments =	cost_to_comment_one_API * public_undocumented_api
Coverage =	cost_to_cover_one_of_complexity * uncovered_complexity_by_tests (80% of coverage is the objective)
Complexity =	cost_to_split_a_method * (function_complexity_distribution >= 8) + cost_to_split_a_class * (class_complexity_distribution >= 60)

Sonar – Cost Calculation



Sonar - Calculation of development days

The number is intended to motivate those responsible or developers to invest in quality. This is particularly useful if the Technical Debt does not result in insignificant "interest payments" (expenses).

Of course, it is clear that this absolute number is very likely to have nothing to do with reality. There may even be cases where a high technical debt is not so bad. And here too it is like with the JDepend metrics (e. g. the d). It's not the absolute number that counts, but the experience gained with such graphics and numbers (which can be further detailed at any time with a simple click).

Such metrics and tools sensitize for the quality factor. This is usually just as important as being the first to be on the market with a product. Quality is one of the top priorities when it comes to dominating the market in the long term and asserting itself against the competition.

Vertiefung

Software Best Practices Blog Matrin Fowlers Comment Announcement bySonar Heise Artikel by Eberhard Wolff Cognitive Horizon

3.11 Werkzeuge

Title: Tools

The tools mentioned so far mostly offered pure code metrics. To improve the architecture, there are programs that integrate the low-level analysis of Checkstyle or PMD. On this page you will find examples of such tools for the Java language. For. NET and other languages like Ruby or Python there are similar tools.

The tools are divided into two license types: commercial and free programs. The free people are divided into three categories according to their abilities:

1. All-purpose tools,
2. Purely physical dependency analysis and
3. Dependency analysis against a logical structure.

The commercial tools listed here all have a graphical user interface.

- **SonarJ**: The most expensive and at the same time most powerful tool comes from the German company Hello2morrow (weblink www.hello2morrow.com). The architecture is divided horizontally and vertically to form subsystems in the levels. Java artifacts (classes and packages) are linked to it. For developers, there is an Eclipse plugin that displays architecture violations as a problem at development time. Class relationships can be virtually removed, so that the effects of refactorizations are immediately displayed. SonarJ works on class files. The source code is included to explain dependencies. An Ant Task generates HTML reports and stops the build in case of architecture violations. Some metrics are listed in the interface and in the reports.

- **Lattix LDM**: Lattix, an American company, manufactures LDM. This organizes the structure in matrices according to the DSM12 approach (http://en.wikipedia.org/wiki/Dependency_Structure_Matrix). Since no logical structure can be defined, LDM does not find any architectural violations. Consequently, automation is not possible. Packages can be used to set up rules that can be accessed. UML diagrams in the exchange format XMI should be able to be imported. In connection with MDA projects, the structure should be analysable. However, importing simple diagrams from MagicDraw in XMI format seems to be problematic.

- **Structure101**: The program was developed by the Irish company Headway Software. Like Lattix, Structur101 follows the DSM approach and also offers more visualizations of the packages. A logical model is based on the Java package hierarchy. Structure101 assumes that packages at the same level in the package hierarchy have the same logical meaning. Due to the strong reference of the logical to the physical structure, most views focus only on classes and packages. At each packet level, it identifies areas that are not related to other packages. It uses only a few metrics. You can use the command line to install it in a build system. An Eclipse plugin indicates architecture violations at the time of development.

- **STAN**: Dhe manufacturer Odysses Software[Ody] produces STAN (Structure Analysis for Java). It is available as an Eclipse plug-in and as a separate application. STAN does not allow the definition of logical structures, which is why all analyses are performed at packet level, to which various metrics are applied. Extensive reports with diagrams can be generated. With the trial version, only one test project can be reviewed.

Except for JDepend, none of the following free tools have a graphical user interface. They are designed to run with the build and generate a report, usually in HTML format. The free general-purpose tools examine architecture in different ways. Both subsequent tools include a variety of other free tools.

- **Sonar**: Sonar (http://sonar.codehaus.org/) is called "Code quality management platform". A tool for the analysis of dependencies is still missing, but it is on the request list. Many metrics are used, some of which are displayed graphically. A special feature is that each analysis result is persisted in order to display it later on timelines. Sonar runs as a stand-alone server and comes with a web application as a user interface. The project to be investigated has to be configured for Maven. The graphical presentation of Sonar is very informative and appealing. Sonar is the only platform that calculates the technical debt and displays it in the form of a monetary debt.

- **XRadar**: XRadar (http://xradar.sourceforge.net/) uses two tools, JDepend and DependencyFinder, to investigate dependencies. A logical structure can be defined in layers and their subsystems. XRadar generates extensive reports (spider diagram) and checks for architecture violations during the build with Maven or Ant. The logical structure and other diagrams are provided graphically. It offers various metrics for which limits can be configured. The configuration for a test project other than the one supplied is not easy, since error messages are lost in the long log. XRadar evaluates the reports from several reports to give a historical overview. This includes various diagrams about the development of metric results.

- **JarAnalyzer**: JarAnalyzer considers a Jar file as a component and analyzes the dependencies to other Jar files (http://www.kirkk.com/main/Main/JarAnalyzer). The result is a report and a dot file. The dot file can be converted into a graphic that displays jar files with their dependencies.

- **JDepend**: It analyzes the class files and creates a report with dependency metrics (http://clarkware.com/software/JDepend.html). All specifications refer to the package level, including the used and dependent packages. In addition, a diagram can be generated via the detour of a dot file. It displays each package and its relationships.

- **DependencyFinder**: It (http://depfind.sourceforge.net/) provides a set of investigative tools. Only a few are applied to metrics. A complete examination takes a relatively long time and results in an XML file of 100 MB. HTML reports are generated from it.

The free dependency tools of logical structure checking only deliver results in text form:

- **dependometer**: (http://dependometer.sourceforge.net) Eine logische Struktur kann in Schichten und funktionalen Abschnitten angelegt werden. Ein Lauf bricht bei maximaler Speicherzuweisung wiederholt ab und hat 4 GB Daten auf die Festplatte geschrieben.
- **Architecture Rules**: (http://dependometer.sourceforge.net) Die Konfiguration der logischen Struktur ist nicht gut dokumentiert. Folglich wird kein Ergebnisbericht erzeugt. In den Editionen der Architecture Rules fehlen brauchbare Details.
- **SA4J**: SA4J (Structural Analysis for Java) was developed by AlphaWorks / IBM (http://www.alphaworks.ibm.com /tech/sa4j), but development was discontinued in beta 2004. The installation of the program is complicated.
- **Macker**: (http://innig.net/macker) Macker only checks for violations during the build and creates an HTML report. A logical structure can be created in components. The configuration of package patterns is powerful, but not well documented for component structures. The report and a check against the logical structure run quickly. In the event of violations, the build system receives an error message with the dependencies that are not allowed.
- **japan**: (http://japan.sourceforge.net/) You can define the logical structure at package level without specifying patterns. Each dependency must be described individually. Besides Ant, IntelliJ is supported. There is no report. The expenditure is not recoverable.
- **Classycle**: (http://classycle.sourceforge.net/) An HTML report provides a detailed view of the dependencies. The logical structure can be defined in layers and components. The report and a check against the logical structure run quickly. In the event of violations, the build system receives an error message with the dependencies that are not allowed.

Without taking into account the costs, SonarJ would probably be the best tool. Sonar, XRadar, Classcycle and JarAnalyser are recommended for further testing.

The following two tables summarize the results. Thereby Kom. for commercial software and FOSS for Free Open Source Software. The line Try gives an impression of the effort required to run the tool successfully (++ = high success).

Kriterien/ Produkt	**SonarJ**	**Lattix LDM**	**Structure 101**	**STAN**	**Sonar**	**XRadar**	**JarAnalyzer**

Version	4.0	5.0	3.2	1.0.3	1.9	1.0	1.2
Ausgabejahr	2009	2009	2009	2009	2009	2008	2007
Lizenz	Kom.	Kom.	Kom.	Kom.	FOSS	FOSS	FOSS
Definition logischer Struktur	++	--	O	--	--	+	--
Vergleich logisch / physisch	++	--	+	--	--	++	--
Visualisierung der Abhängigkeit	--	+	+	+	--	O	+
Zyklen finden	++	+	--	O	--	+	O
Metriken	+	O	O	+	++	++	+
Berichte	O	+	O	+	++	+	+
Ergebnis im Quellcode	+	-	+	O	++	--	--
Ergebnis im Quellcode	++	--	O	+	O	++	++
Unterstützung OSGi	--	--	--	--	--	--	--
Kosten	--	-	-	-	++	++	++
Versuch	++	O	O	-	O	O	++

Software-Werkzeuge I

Kriterien/ Produkt	**JDepend**	**Dependency Finder**	**Architecture Rules**	**Macker**	**Sonar**	**Macker**
Version	2.9.1	1.2.1	2.1.1	12	0.2.4	1.3.3

Ausgabejahr	2004	2008	2008	2003	2005	2008
Lizenz	FOSS	FOSS	FOSS	FOSS	FOSS	FOSS
Definition logischer Struktur	--	--	O	O	-	+
Vergleich logisch / physisch	--	--	--	+	-	+
Visualisierung der Abhängigkeit	O	--	--	--	--	--
Zyklen finden	+	O	--	--	--	+
Metriken	+	O	--	--	--	-
Berichte	+	-	--	-	--	+
Ergebnis im Quellcode	--	--	--	--	--	--
Ergebnis im Quellcode	++	+	+	++	++	++
Unterstützung OSGi	--	--	--	--	--	--
Kosten	++	++	++	++	++	++
Versuch	+	O	-	-	-	++

Software-Werkzeuge II

3.12 Zusammenfassung MET

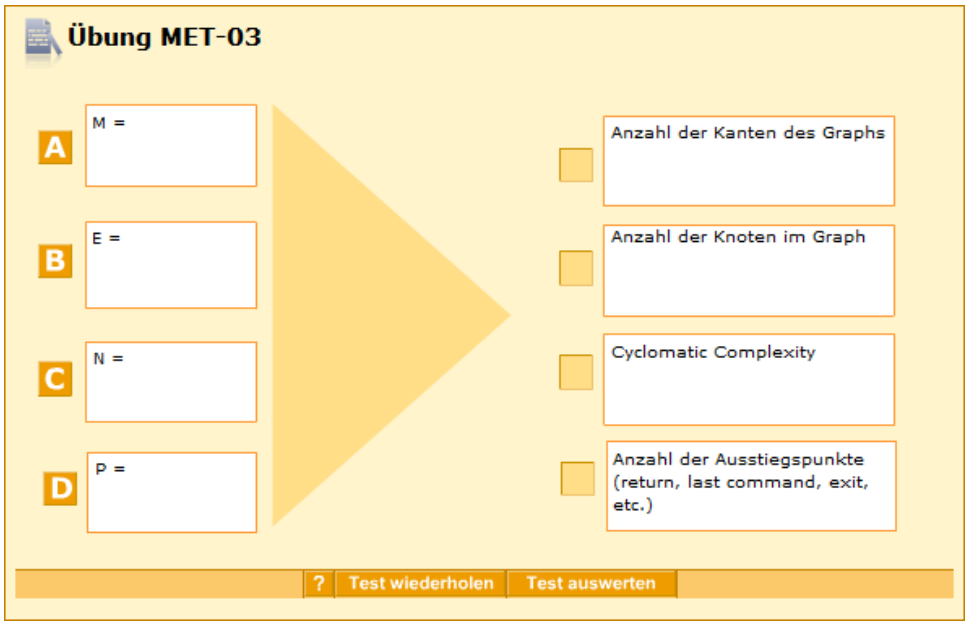
Title: Summary MET

- Metrics are an important tool for quality assurance. Most interesting is not the absolute metrics, but the reference to a problem and the relations of the metrics to each other.
- Even simple metrics like McCabe or Halstaed can be used to calculate interesting complexity metrics in relation to the code.
- Metrics for rule violations in the code are another important area. There are a variety of tools that can look over your shoulder.
- When it comes to cycles, abstract programming and package dependency analysis, JDepend is one of the most important basic tools. Many of the principles have been integrated into other tools.
- In the architecture analysis, there are tools that allow you to check accesses and thus force a better design (e. g. only top-down accesses).
- With other metrics such as the rACD measure, intrinsic module complexity can be measured. These metrics can provide architects with valuable clues to design decisions that are sometimes disadvantageous.
- With the Technical Debt, new interesting metrics are now available to measure the entirety of all violations - in the area of code metrics and architecture metrics - and state them as expenditure debt (in $ or €).

3.13 Wissensüberprüfung MET

Übung MET-02

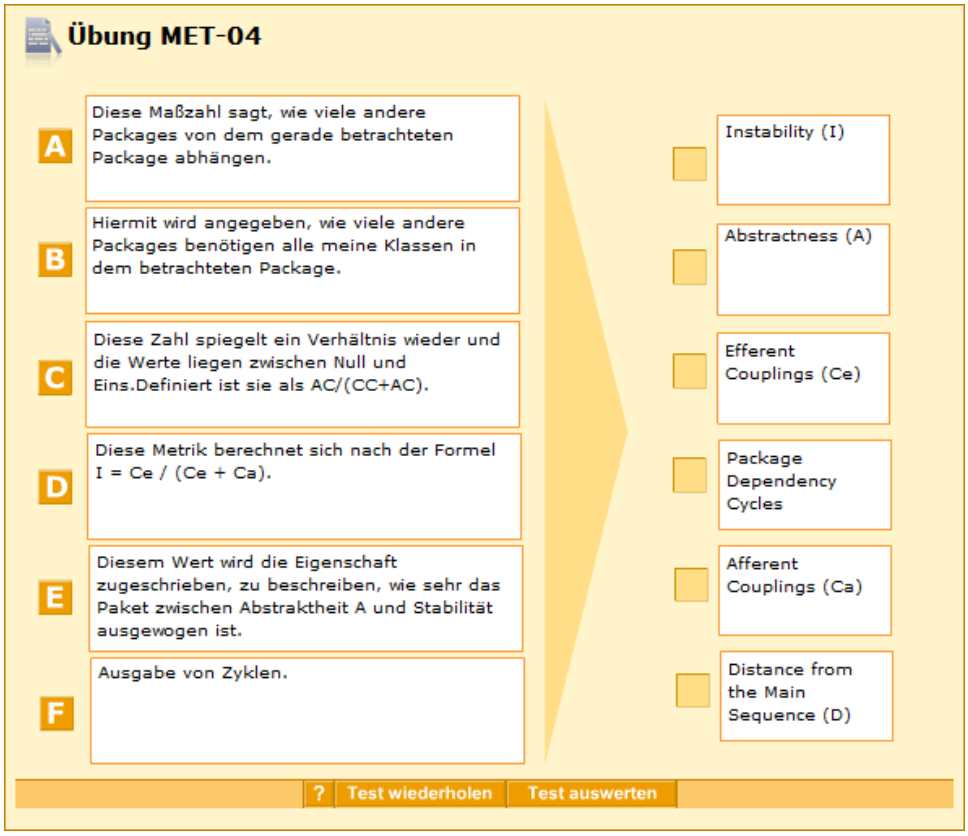
Anfang Druckversion



Ende Druckversion

Übung MET-03

Anfang Druckversion



Ende Druckversion

4 CCD - Clean Code Development

Autor: Prof. Dr. Stefan Edlich

4 CCD - Clean Code Development
4.1 Überblick und Lernziele CCD
4.2 Einleitung zu Clean Code Development
4.3 Fundamentals
4.4 Coding Basics
4.5 Code Quality
4.6 Architecture und Class Design
4.7 Packages
4.8 Produktivität
4.9 Management
4.10 Zusammenfassung CCD
4.11 Wissenüberprüfung CCD
4.12 Literatur CCD

4.1 Überblick und Lernziele CCD

The following learning unit is to be understood as a kind of checklist for the responsible handling of code and the design of good code / good software.

The community now has a global understanding of best practices, rules, bad-smell and coding guidelines that ensure good, modifiable, sustainable and readable code.

This knowledge is not coherent, but exists selectively. For example, there are overlaps in class design, but most of the rules mentioned here are relatively independent of each other.

The student's goal is therefore not only to understand them, but also to create his or her own kind of "cheat-sheet" and to follow the guidelines in his or her practical work.

Lernziele

Learning Objectives In this learning unit, the main point is not to code blindly, but to observe oneself on a kind of meta-level. Ideally, the person should split into two persons - as in the XP procedure model: A) A person who is coded and pays more attention to the current syntax and the specific algorithm, and (B) a person who observes as a meta instance and gives clues.

In this module all the skills are taught to be a perfect B-person or to check yourself as a B-person. In addition, this B-person learns all the principles, practices and "smells" to be able to propose and implement competent improvements at all levels. From the pure code level to architecture and management.

Gliederung

Gliederung This learning unit is divided into the following parts:

- Introduction
- Coding Fundamentals
- Coding Basics
- Code Quality
- Architecture / Class-Design
- Packages
- Productivity and
- Management

Zeitumfang

Time Requirements
It takes about 60 minutes to work through the learning unit and about 130 minutes to complete the exercises.

4.2 Einleitung zu Clean Code Development

Title: Introduction to Clean Code Development

Despite the development of better MDA tools (Model Driven Architecture) and programming languages on an ever higher level, it will still be hand coded in many years to come.

And that good code is important and that good quality features should be fulfilled, is also signed by everyone. Nevertheless, there are already many companies that had to file for bankruptcy due to bad code. Everyone knows the problem, but no one fights it at the root. Everyone of us has seen foreign code before and surely got to see a lot of bad code.

How can this all happen?

1. On the one hand, from the ignorance of the developers
2. Secondly, from the pressure of the market to offer "something" (a prototype?!) as quickly as possible.

In addition, the code complexity and thus the maintainability, readability, testability and modifiability of code increases with time and increasing bad code. Often, this can no longer be controlled. A significant number of companies around the world are involved in debugging and maintaining their software. Software maintenance is estimated to account for more than 80% of the cost of software. Managers, companies and often developers are not aware of the importance of a high technical debt.

Anmerkung

Technical Debts
This refers to dangers/problems/consequences resulting from badly written software.
More information:

1. #
2.

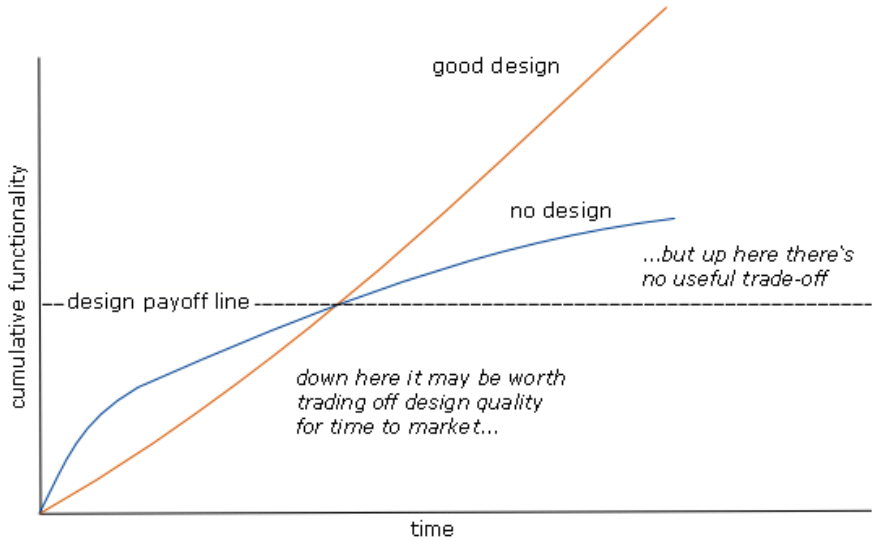
Managers often consciously make the decision:

' First develop quickly and earn money.
Repair and build in quality can be done later.'

In practice, however, it is all too often the case that there will be no future in which money and time will rain.

In the following graphic you can see a comparison of two projects which differ from each other by "good design" and "no design". At the beginning,"no design" with a steeper upward curve surpasses the functionalities of "good design". However, productivity is reduced over time due to poorer maintainability and modifiability. For example, this is about the effort to introduce new features. At the same time, the slope of the curve decreases and the time it takes to earn money from the product decreases. Therefore, there is usually a break-even point at which it is no longer worthwhile to enter Technical Debt. But where is this break-even point. After an in-depth discussion with experts such as Martin Fowler, it is concluded that this point comes much earlier than managers would suggest.

Please have a look at the folliwing graphics: (Quelle: Bliki):

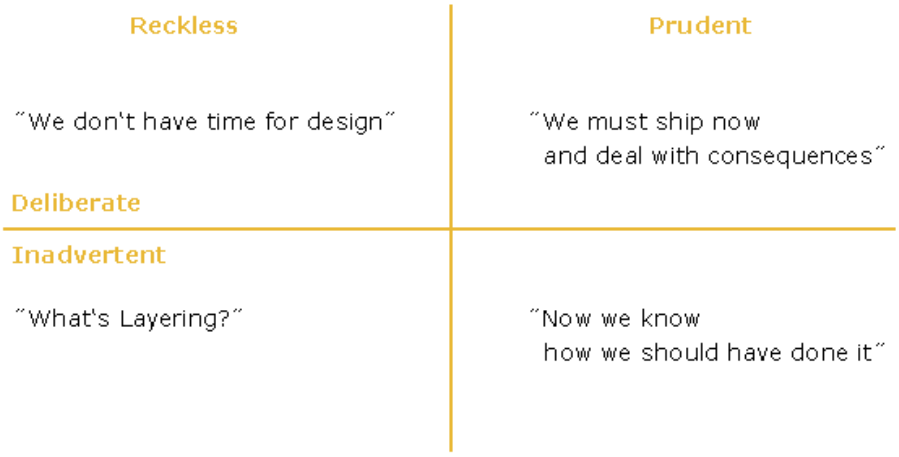


Comparison of two projects: 'no design' und 'good design'

The goal is to encourage developers to develop sustainable code whose quality is so good that technical debts are minimal. On the other hand, there must be a counterbalance to managers who prefer fast money and unsustainable value.

Of course, it is also legal to say that you can take the initial money with you and then let the company go bankrupt. Only then should you tell the developers from the beginning. However, since this is not an option for most companies, the question arises as to why and on what basis decisions are made against good design/good code and thus for implicit technical debts.

Here is a good overview by Martin Fowler (Quelle: Bliki):



Technical Debt Quadrant

The axes are labeled as follows:

- Reckless (rücksichtslos / unverantwortlich)
- Prudent (vernünftig / besonnen)
- Deliberate (absichtlich / bewusst)
- Inadvertent (unbeabsichtigt / ungewollt)

The graphic can be interpreted as follows when transferred to the Technical Debt and this learning unit:

1. **The Reckless / Deliberate** are the ones who want to take the fast money with them and deliberately and ruthlessly accept a code debt. However, it is deliberately not to be aware of a later failure, to be able to display or to take into account it. With "We don't have time for design" (or even good code or zero technical debt) it is expressed that the speed and hardness of the market does not permit any other action. So you have to act in this way and, once again, you are not actually responsible.

1. *'The Reckless / Inadvertent* are here rather the ignorant ones, who can't understand quality, code quality, design or technical debts. Mostly these are people from other disciplines who don't want to deal with them at all. What's Layering "not only expresses the question of what layer architectures (such as N-Tier or MVC) are, but also somehow a form of ignorance according to the motto:" I don't care anyway ".

1. **The Prudent / Deliberate** persons are well aware of the danger of bad code' somehow'. Nevertheless, the problem is - similar to the pyramid scheme - displaced and pushed backwards. Motto: Either it won't be so bad or we will get it under

control somehow. Therefore,"We must ship now and deal with the consequences" applies here. Delivering is therefore still the number one priority.

1. **The Prudent / Inadvertent**' people are the ones who have consciously, perhaps even from the very beginning, the quality in mind, who recognize errors and now at least hope to have time and money to address the problems. Nevertheless,"Now we know, how we should have done it" shows that despite good will, things have gone wrong.

So developers should do their job with the knowledge:

- I have to convince the managers that sustainability = low technical debt is an advantage. Fortunately, you can talk to most (IT) managers and often find a solution in the sense mentioned here.
- I know that 90% of the time code is read and not only written by me.
- I know that the cost of code maintenance and enhancement is more than 80% of the total software cost.

According to
Robert C. (Bob) Martin and Maartin Fowler
it is our duty to defend the / our code with passion!

Die nachfolgenden Auflistungen sind daher:

- **Principles / Practices**: Best practices rules that have proven themselves. Many of these rules are design patterns or good manners among classes / packages.
- **Smell**: This is about what happens if the rules are not followed. So a rather inverse wording, from which the rule is derived.

4.3 Fundamentals

Code is read 5-8 times more than written. And code maintenance is one of the major problems of the industry. Code that is still modern today will be' legacy' tomorrow. Even for the developer himself, speaking clear and logical code is a help. Everyone knows the effect of having to read their own code months later and having a hard time getting in. To code with a lot of technical debts you lose the "contact" much faster and to understand and expand / change it is also more difficult.

Another major problem of the Technical Debt is that managers often rely on the lock-in effect. Once a customer has bought a product, he usually "hangs" in it for 10 years. This is the argument that developers face when it comes to eliminating technical debts. Often

it only helps here if you argue with hard numbers. For example, now 2 developer weeks and then in 2 years no 2 developer months.

References:

- Reduce the Cognitive load

4.4 Coding Basics

Kill Useless Code

Kill Useless Code

Under developers it is sometimes common practice to leave code without a task in the source file. Usually this is also code found by Coverage Tools and cannot be found by any test.

It is assumed that this code belongs to a later to-do. Then a "ToDo:" (e. g. as Eclipse tag) should also be displayed here. And of course it has to be ensured that the corresponding item ends up in backlog lists somehow.

Delete Useless Comments

Delete Useless Comments

This refers to the old discussion: A camp is of the opinion that everything should be documented (i. e. every command line, however understandable it may be). On the other hand, there are those who argue that the code must be written so clearly that no comment is needed. The reader can thus recognize

1. The meaning of the code line and

Information about the actual implementation

However, there is agreement that code should be sufficiently documented at least at the package or class level in order to get an abstract introduction to the sources. This applies in particular to API design.

Two Examples:

Code

```
// Get to know how much was deleted.
x = tmpObj.getDelI();
```

```
// Get to know how much was deleted.
itemsDeleted = Repository.getLastDelItems();
```

Übung CCD-01: Diskussion

Discuss this topic with your fellow students or lecturers. Make yourself a few key points.

- Referenz: "5 reasons to avoid code comments"

Invest in a Precise Naming

Invest in a Precise Naming

This is not only about variables, but also about fields, methods, interfaces, classes or packages. Here too, it should be borne in mind that

- the code is mostly read and therefore has to transport as much information and context as possible.
- On the other hand, however, it must not be too long, so that manual refactoring or enhancements do not take up too much time or make the line unreadably long.

Here it is really worthwhile to find the precise description that represents the optimum of both poles. Names should be self-explanatory. The naming must be consistent. The aim is to achieve symmetry.

Here are some general rules:

- Try to find the best level of abstraction.
- Interfaces should describe the functionality they abstract (Clonable has . clone () method)
- Classes describe the type of implementation. (SSDStream implementations IStream)
- Methods describe what they do but never how they do it.
- Do not encode any type or scope information into the name.
- If something has a side effect, this should be described in the name.

Anmerkung

Scope

dt. = Bereich, means the area of visibility

The last point is coded in some modern languages like Ruby or D as extra characters. In Ruby there are the methods
capitalize (String)
and
capitalize! (String)
. Here, the rules make it immediately obvious which of the two methods has a character that changes from the beginning.

There are standards for all languages, but these differ widely (e. g. Java, Ruby, D, Clojure) but must be applied consistently.

The same applies to methods: you should complete a task. See also point:

→ Single Level of Abstraction (SLA)

Code should read like normal English text / prose:

Code

```
if(plane.altitude() < 100 * tolerance)
 generiereWarnung("Proximity Alert!");
```

Avoid Magic Numbers

Avoid Magic Numbers

Have a look at the following example:

Code

```
MoveList ml[50];
```

In extremely many programs you will find' Magic Numbers'. In 50% of the cases you might even be able to find out with a little effort what this number means and why 40 or 70 was not taken here. In many cases, you will never be able to find out.

On the other hand, the following is much more enlightening:

Code

```
// (in the sourcefile consts)
immutable MAXDEPTH = 50; // For dynamic.d how far will we search
//in chess at maximum

// (now in the file dynamic.d)
MoveList ml[MAXDEPTH];
```

The problem is that very few developers have enough empathy to imagine that another developer spends a lot of time trying to figure out the cause of the magic number. And it is not only about time, but also about the inner attitude and acceptance of third parties to the currently written code.

Prefer Polymorphism to if/else or switch/case

Prefer Polymorphism to if/else or switch/case

But what is the problem? Every developer is used to
if/else
write chains and find nothing. And of course, code fragments are difficult to polymorphise when it comes to pure numbers and / or arithmetic.

Nevertheless, with
switch-case
behavior is often hard wired. In many cases, variants are much more elegant in which the behavior called after
if/else
or in
switch-case
constructs can be called much better by a polymorphic behavior.

No Sideeffects

Ensure that "Changes have Local Consequences"

Besides the classic NullPointer exceptions, side effects are probably one of the main reasons for errors in large software systems.

A typical example is that other modules or methods rely on variables such as age, but they can then assume negative values by mistake. Often, counters are increased for certain actions. Then also during testing. Then a counter that is too high can have negative effects again, etc.

From this burning problem, new research areas and programming languages have developed. In research, the researchers are trying to prove the correctness and' side-effect freedom'. Programming languages such as Haskell or Clojure are inherently almost free of side-effects (unless you misuse some constructs of the language). In functional programming, a method must not have side effects. It should be pure. For this reason, variables have almost been abolished there. If programs have to change the state' global', this is done via different identities of a variable and strictly controlled and threadsafe.

Fields shall define state

Fields shall define state

Fields are often misused to hold temporary data. Fields should represent strict states of the object.

It helps here,

1. local variables,
2. or extract another class that can calculate or maintain temporary states.

Correct Exception Handling

Think about Exception Handling

Exceptions should always be collected as specifically as possible. Only catch all possible types and only exceptions if necessary. Exceptions only catch exceptions if you are on the right level, i. e. you can do something decent with them. Otherwise, the next higher level should do something about it. Never abuse exceptions for regular control flow. Exceptions must be dealt with properly. Otherwise, the system is threatened to remain in an inconsistent state.

Know and apply refactoring patterns

Every developer should:

1. the Refactoring Catalogue (weblink) and have read the relevant literature on this subject.
2. know the refactoring menus from the current IDE inside out and have gained practical experience with them.

Only with this knowledge, these tools and a version control system underneath can code be changed efficiently.

4.4.1 Coding Source

Lokale Deklaration

Vertical Separation

If possible, local variables should always be declared where they are used. A collective declaration at the beginning of the block must be avoided. The declaration should be done before or in the first use and then have as small a scope as possible.

Questions:

1. Why are collective declarations at the beginning of a block usually not useful?
2. What does the length of the scope of a variable depend on?
3. Should variables be explicitly destroyed?

Explanatory Variablen

Explanatory Variables

As long as the program does not have to be 100% optimized for performance (and this is rarely necessary), you should implement the optimal number of explanatory variables. This makes the individual steps understandable, easy to read and maintain.

Beispiel:

Code

```
float net = gross * 0,85;
float tax = net * 0,75;
float fee = tax * 0,1;
```

versus

```
result = input * 0,85 * 0,1 * 0,75;
```

Nesting

Nesting

Deep nested code should take on ever more specific and detailed tasks. The abstraction level or the call probability should be higher at higher levels.

Separate Multi-Threaded Code

Separate Multi-Threading Code

Multi-threaded code should not be mixed with' normal' code. A thread should (and often must) always be encapsulated in its own class at startup.

Code Example:

Code

```
public class SwtThread extends Thread {
 public void run() {// here it happens
 }
}

class X {
 Thread thread;
 void foo() {
 thread = new SwtThread ();
 thread.start();
 }
}
```

4.4.2 Coding Conditionals

Encapsulate Conditionals

Encapsulate Conditionals

Reading complex logic operations is always much more time-consuming than reading a good, explanatory method.

It should therefore always be considered whether it makes sense to encapsulate complex logic.

Example:

Code

```
if (this.ShouldBeDeleted(timer))
```

liest sich einfacher als

```
if (timer.HasExpired && !timer.IsRecurrent)
```

(C) CC Cheat Sheet

Avoid Negative Conditionals

Avoid Negative Conditionals

Negative logical expressions are more difficult to read than positive expressions. Avoid it if possible.

Example:

Code

```
not > 0 versus <= 0 oder
not positive(x) versus nevative(x) // if 0 does not play a role
```

ncapsulate Boundary Conditions

Encapsulate Boundary Conditions

Boundary conditions for e. g. loops like
while / do-loop / loop-do
should be clearly marked and listed in one place. Under no circumstances be hidden in the core code of the loop. For example, if the loop needs to be changed, refactoring would be more difficult.

Example:

Code

```
nextLevel = level + 1; // gefolgt von
```

```
this.callLevel(nextLevel);
```

4.4.3 Coding Principles

Don´t Repeat Yourself (DRY) (rot)

Don´t Repeat Yourself (DRY)

Analyses have shown that copy-and-paste programming is one of the most common sources of error!

This practice and code repetitions involve a variety of problems:

You are guaranteed to forget to turn a screw at some point, because it's not really thought of. The code no longer needs to be maintained because changes to the basic structure of the algorithm must be made everywhere.

What you should do instead is to encapsulate or parameterize the algorithm in a class. This is based on the DRY principle: avoiding the same code points where possible, whereby this principle does not only apply to code points. DRY is one of the most important principles of Ruby on Rails: definitions have to be done only once. In contrast, there are many specifications in which names, code or configurations must occur several times in different places. This is then a pleasant field for code or configuration generators. But of course, the automatic and invisible generation in the program is always better than repetitions (with generator help) considering the DRY principle. The EJB 2 specification is a nice negative example, since descriptors and interfaces and classes often have the same information content.

Example:

Code

```
code code
```

Einfachheit als oberstes Prinzip (rot)

Keep it simple, stupid (KISS)

Schaut man sich die Liste der Gründe an, warum IT Projekte laut IT-Managern angeblich scheitern, so ist von Komplexität fast nie etwas zu hören. Dennoch zeigen auch hier Studien, dass die intrinsische Komplexität ein nicht zu unterschätzender Faktor ist. Menschen sind anscheinend nicht unbedingt perfekt darauf ausgelegt, die Architektur

von tausenden Klassen zu überblicken und diese inhärente Komplexität zu beherrschen ("komplexity kills").

Anmerkung

Unter der intrinsischen Komplexität versteht man die enthaltene, d.h. quasi naturgemäße Komplexität eines Problemes. So ist z.B. die intrinsische Komplexität der Lösung der Goldbachschen Vermutung einfacher als das Sieb des Erestotenes (Primzahlen finden).

Aus diesem Grunde ist es auch ein bekanntes Pattern, Komponenten so aufzuteilen, dass diese dann über ein Protokoll kommunizieren und nicht mehr via "direct call". Falls dies von der Performance her funktioniert, wird so Komplexität verringert.

Aber das KISS Prinzip ist auch mit dem XP Prinzip der (Vermeidung der) "goldenen Wasserhähne" verwandt. Also um Features, die noch keiner braucht. Es geht hier also darum, nicht nur die Menge, sondern auch die Struktur einfach zu halten und erst später zu erweitern.

KISS hilft den Code zu verstehen. Besonders von anderen Entwicklern.

Aber auch dieses Prinzip ist nicht auf Code beschränkt. Es geht auch hier um Datenstrukturen, Code, Komponenten- und Paketstrukturen und Architekturen.

Beispiel:

Code

```
function quicksort('array')
 if length('array') = 1
 return 'array' // an array of zero or one elements is already
 sorted
 select and remove a pivot value 'pivot' from 'array'
 create empty lists 'less' and 'greater'
 for each 'x' in 'array'
 if 'x' = 'pivot' then append 'x' to 'less'
 else append 'x' to 'greater'
return         concatenate(quicksort('less'),         'pivot',
 quicksort('greater'))
// two recursive calls
versus
```

```
DataProcessor.sort(myList);
```

Dieses Prinzip hängt auch mit dem YAGNI Prinzip zusammen (bzw. ist oft äquivalent).

Code soll bitte "kommunizieren, einfach und flexibel sein"

Code Communication, Simplicity, Flexibility (Beck).

Viele dieser Regeln wurden schon von
Kent Beck
in seinem Buch "Implementation Patterns" (*Addison-Wesley, 978-0321413093*) zusammengefasst.

Die Punkte sprechen die folgenden Inhalte an:

- Code Kommunikation => Selbsterklärender Code für Dritte, der Kommentare überflüssig macht
- Simplicity => KISS Prinzip
- Flexibility => Änderbarkeit und damit auch Wartbarkeit des Codes

Vorsicht vor Optimierungen! (rot)

Avoid Early Optimizations

Frühe Optimierungen neigen dazu die Dinge

1. Zu stark zu komplizieren
2. Zu viel Zeit zu investieren
3. Einen Optimierungsdruck anzunehmen, der so vielleicht gar nicht existiert.

Möglichkeiten der Optimierung: Hardware / Software (Profiler), Datenstrukturen / -typen

Lesen sie dazu:

http://www.codinghorror.com/blog/2008/12/hardware-is-cheap-programmers-are-expensive.html

Darin wird verwiesen auf (M.A. Jackson):

Hinweis

Rule 1: Don't do it.
Rule 2: (for experts only): Don't do it yet.

"Doing" versus "Calling" Code(orange)

Single Level of Abstraction (SLA)

Worüber Entwickler oftmals nicht nachdenken ist, auf welcher Abstraktionsebene der geschriebene Code liegt.

So unterscheidet man grob gesagt zwischen

A) "doing" Code: d.h. Code der wirklich etwas tut (z.B. berechnet) und

B) "calling" Code: dies ist Code, der konkreteren Code aufruft.

Entscheidend ist, dass der Developer sich klar sein sollte, welchen Code er in der Methode oder in der Funktion aufruft. Auch hier gilt es sich in eine lesende Person hineinzuversetzen und dieser die Entscheidung zu überlassen, ob man abstrakten Code oder konkreten Code lesen möchte. Man erhöht damit stark die Geschwindigkeit, in der der Code verstanden wird.

Beispiel:

Code

```
// mit Vermischung!
public int calculateSpecificTax(int income, int rate1, int
 rate2,
int amortization){
 int tmp = income * rate1 + 10;
 tmp = tmp * rate2,
 tmp = calculateAddOn(tmp); // andere Ebene!
 tmp = tmp * 0,97;
 return tmp;
}



// ohne Vermischung!
public int calculateSpecificTax(int income, int rate1, int
 rate2,
int amortization){
 int netGain = calculateNetGain(income, rate1);
 int netReturn = calculateOffsetRate(netGain , rate2);
 int grossResult = calculateAddOn(netReturn);
```

```
 return calculateFinalTax(grossResult);
}
```

Keep Configurable Data At High Levels

Keep Configurable Data At High Levels

Configurations or default values should be located at the correct level. So it is better to be read at a higher level of abstraction and then read or transferred to deeper functions. Configurations or default settings are often hidden in low-level functions, so searching for them takes a lot of time.

Configurations: be careful

Over-Configurability

Good configurations are a blessing in the first place. With property files or the numerous configuration frameworks, changeable parameters can be extracted from the code.

On the other hand, many large programs now have hundreds to thousands of setting options. And not infrequently, installations fail because of "over-configurability". This means that too much configurability is often stored externally. The complexity increases here too, and no one can see through.

There are many possibilities how to:

1. Working with default values and subsequent adjustment
2. The program can detect the optimal configuration (e. g. number of cores, availability of data / files).
3. Remove configuration parameters
4. etc.

Often it is advantageous to be able to agree on at least one framework (such as Commons Configuration, Spring or Obix).

Hidden Temporal CouplingRule

Hidden Temporal CouplingRule

If possible, the call of methods should not specify a fixed sequence. If this is the case, you have to make sure that the sequence cannot be called incorrectly.

Beispiel

Example:
kein
read();
vor
open();

Don't Be Arbitrary

Don't Be Arbitrary

Guidelines for coding or structuring the code must be logical and maintained. The structure must be consistently communicated by the code.

If there are fluctuations and the guidelines are not adhered to, others feel encouraged to change the code.

4.5 Code Quality

Use Constraints

Use assertions! Pre / Post Conditions

Code quality must be performed at the lowest level in the form of simple tests.

There are basically three forms of this:

1. **Pre-conditions:** Test whether all parameters contain correct values.
2. **'Post-conditions:'** Test whether all return parameters are correct.
3. **'Assertions:'** Test whether variables and objects in the function code are correct.

Besides copy and paste, return null is one of the most common programming errors. The problem with empty and unset return values is that these empty objects are not easily traced back and can be passed up. And that will be expensive.

A) The simplest approach is to check variables, parameters and return values in this way:

Code

```
int myFunc(int x, object param1) {
 if(param1 == null)
throw new IllegalArgument("myFunc arg is empty!");
```

```
 ... // code: age=x*5;
 assert(age > o);
 ... // code: result = x * param1.factor;
 if(result == null)
throw new IllegalArgument("result is null!");
 return result;
}
```

Although the success of testing is guaranteed, there are still several problems:

Coding if chains in the function is a nuisance. It pollutes the code because it is orthogonal functionality (an aspect) that has nothing to do with the actual function task.

B) This can be remedied by various validation frameworks (e. g. Apache Commons or Validation) or essays forContract-Programming.

These provide at least macros to simplify the queries.

C) Of course, it is even better if the language itself supports contract programming. There are a number of programming languages like Eiffel or D.

One exampe for this:

Code

```
long square_root(long x){
 in{assert(x >= 0);}
 out(result){assert((result * result) <= x && (result+1) *
 (result+1) >= x);}
 body{return cast(long)std.math.sqrt(cast(real)x);}
}
```

D) Another important possibility is, of course, to completely remove the aspect of testing from the function. This can be done well with AOP, for example. At least for entering and exiting the function, you can define jointpoints. See also the unitAOP - Aspect Oriented Programming.

4.5.1 Source Code Konventionen

Title: Source Code Conventions

Check your SourceCode Conventions

Nowadays, it is easy to verify source code conventions. A large number of tools are available for this purpose.

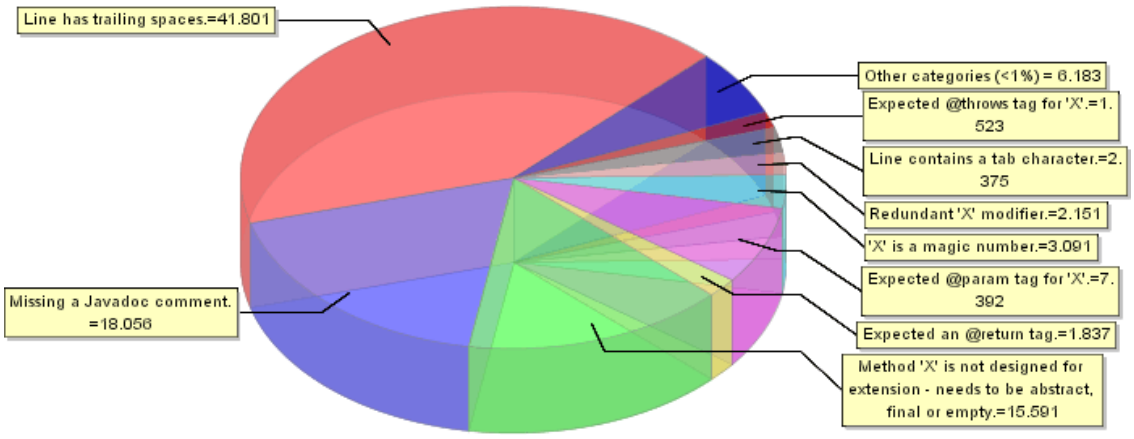
There are some well-known tools for Java alone:

- **Checkstyle**
- **PMD**
- **FindBugs**

The real problem is this:

Integrate this seamlessly into the IDE so that errors are immediately visible and Configure these tools correctly.

For the latter, gigantic XML files often have to be edited in order not to be confronted with thousands of error messages every time.



Percentage distribution of error messages

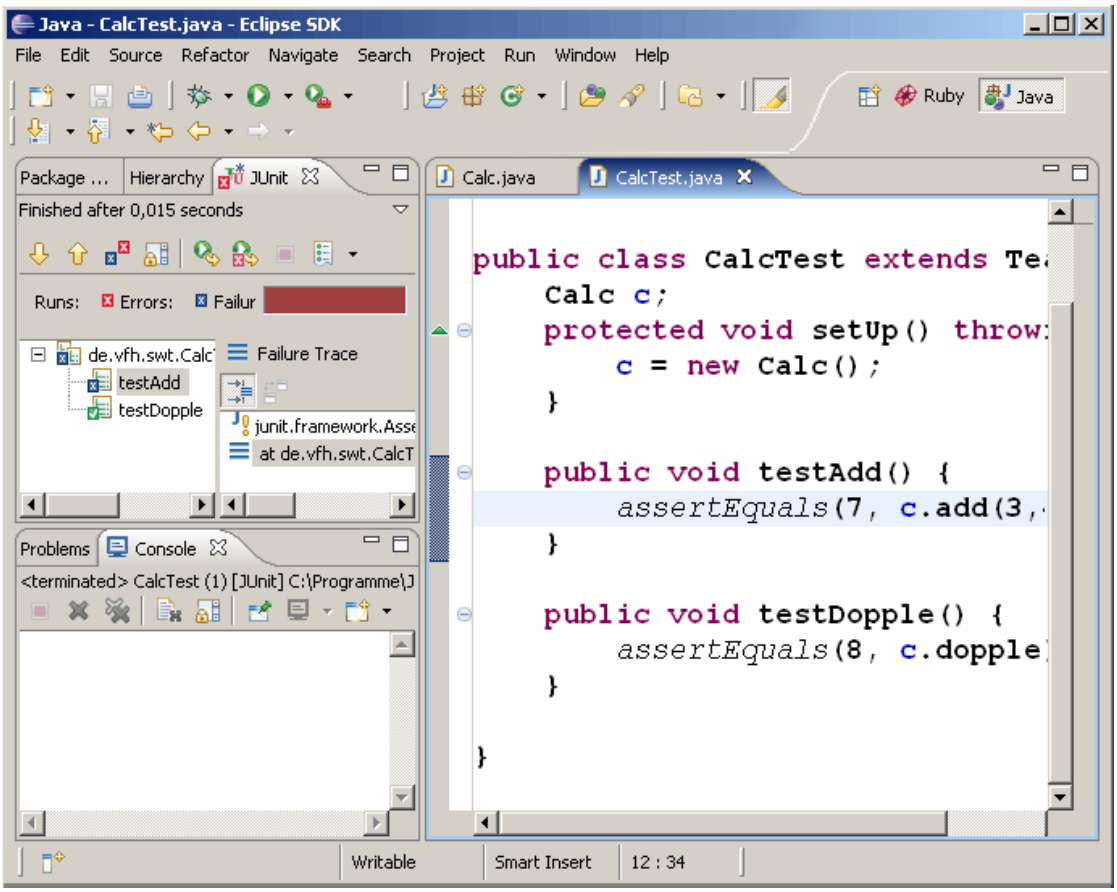
4.5.2 Automatisierte Unit Tests

Title: Automated Unit Tests

Automated tests offer double benefits:

- You save time
- They take fear

The more a codebase is subject to change, the more time is saved. Where code changes, new and old (regression tests) have to be tested again and again. Automation simply saves time. And the more complex the code, the greater the reduction in fear. If complex code needs to be changed - to add functionality, optimize it or simply correct it - there is a high risk of inadvertent errors being introduced. However, small steps of automated tests reveal these, so there is no reason to be afraid of "making things worse".
(C) CCD Westphal / Lieser



Testen mit JUnit

4.5.3 Code Coverage Analyse

Title: Code Coverage Analysis

Code Coverage is the test coverage through test cases. Code Coverage also shows the untested code points.

In practice, a test coverage of 80% is a desirable goal. A high test coverage has become an important marketing tool and plays an important role in awarding and acceptance of orders. There are excellent (also free tools) that can automate the process of code coverage analysis and perform it quite comprehensively.

Übung CCD-02: Discussion „Much helps a lot?“

A high test coverage indicates high code quality!

Would you agree with that statement? Write down briefly arguments that speak both for and against this statement. Keep your remarks ready for the next synchronous meeting.

Processing time: 10 minutes

A good and in-depth introduction to Code Coverage can be found on the Internet under:

http://www.bullseye.com/coverage.html

There are numerous code coverage tools for Java. On the Internet, for example, you can find them at the address:

http://java-source.net/open-source/code-coverage

The most important tools are:

- Clover (commecial), test version available
- Emma (emma.sourceforge.net), Open-Source
- Cobertura (cobertura.sourceforge.net/), Open-Source

Other tools are: Quilt, NoUnit, InsEct, Hansel, Jester, JVMDI Code Coverage Analyse, Grobo, jcoverage, JBlanket,...

(Please do research by yourself and find the leading tools!)

Example from Clover

Beispiel

3 Types of Measurement:

Statement Coverage

Number of statements that pass through a class

Conditional Coverage

How to run through conditional constructs (e. g. if/else). Ideally, the test should cover everything.

Method Coverage

How many methods of a class are called.

Formel

The entire coverage is derived from a more complex formula:

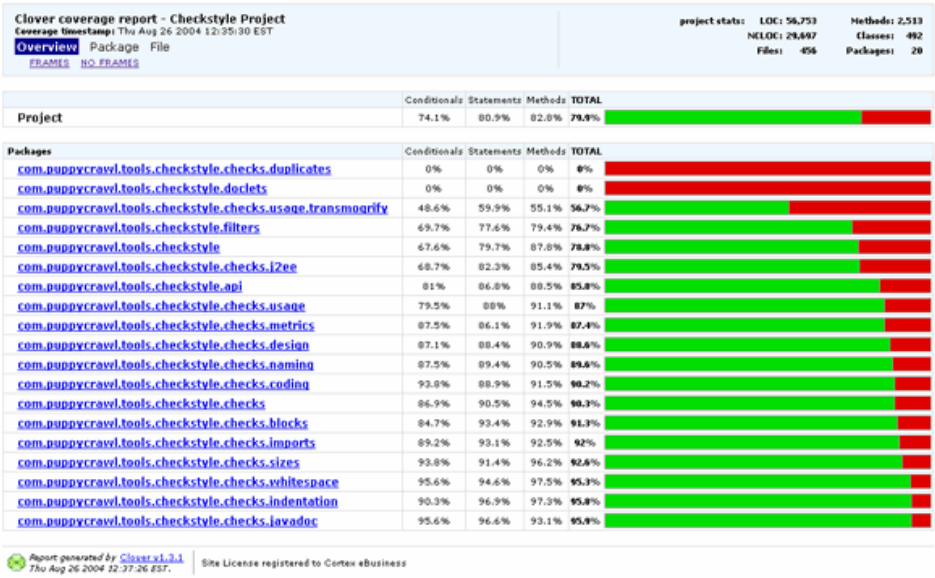
TPC = (CT + CF + SC + MC)/(2*C+S+M)

CT = conditionals that evaluated to „true“ at least once

CF = conditionals that evaluated to „false“ at least once

SC = statements covered MC = methods entered

The most important thing for the project manager/management is to get an overall overview of the coverage:



Clover coverage report - Checkstyle Project
Coverage timestamp: Thu Aug 26 2004 12:35:30 EST
Overview Package File
FRAMES NO FRAMES

project stats: LOC: 56,753 NCLOC: 29,697 Files: 456 Methods: 2,513 Classes: 492 Packages: 20

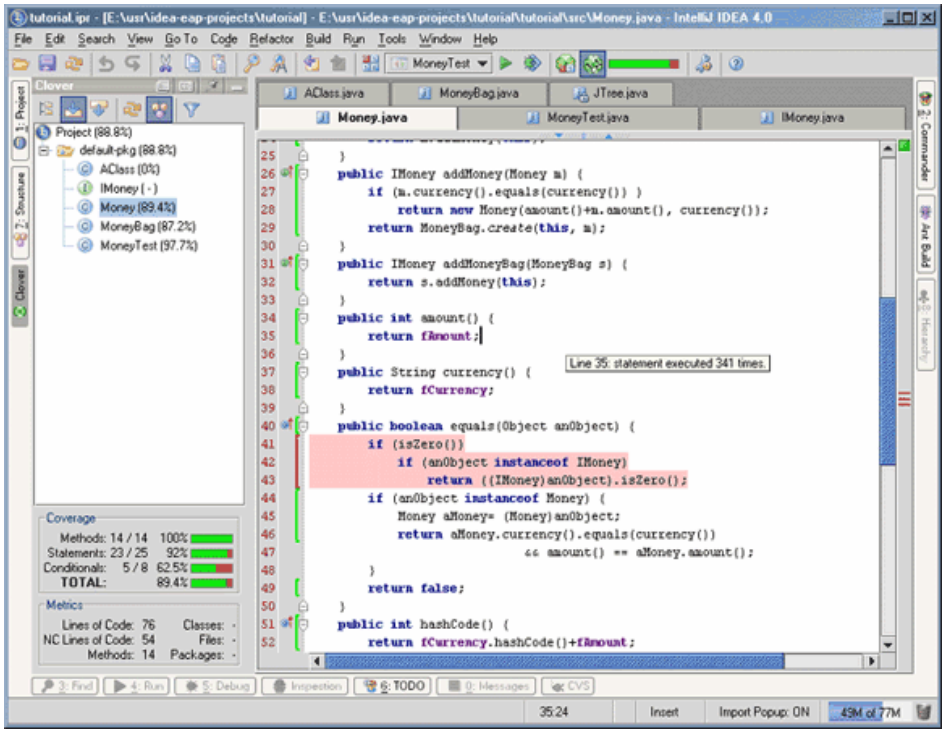
	Conditionals	Statements	Methods	TOTAL
Project	74.1%	80.9%	82.8%	79.9%

Packages	Conditionals	Statements	Methods	TOTAL
com.puppycrawl.tools.checkstyle.checks.duplicates	0%	0%	0%	0%
com.puppycrawl.tools.checkstyle.doclets	0%	0%	0%	0%
com.puppycrawl.tools.checkstyle.checks.usage.transmogrify	48.6%	59.9%	55.1%	56.7%
com.puppycrawl.tools.checkstyle.filters	69.7%	77.6%	79.4%	76.7%
com.puppycrawl.tools.checkstyle	67.6%	79.7%	87.8%	78.8%
com.puppycrawl.tools.checkstyle.checks.j2ee	68.7%	82.3%	85.4%	79.5%
com.puppycrawl.tools.checkstyle.api	81%	86.8%	88.5%	85.8%
com.puppycrawl.tools.checkstyle.checks.usage	79.5%	88%	91.1%	87%
com.puppycrawl.tools.checkstyle.checks.metrics	87.5%	86.1%	91.9%	87.4%
com.puppycrawl.tools.checkstyle.checks.design	87.1%	88.4%	90.9%	88.6%
com.puppycrawl.tools.checkstyle.checks.naming	87.5%	89.4%	90.5%	89.6%
com.puppycrawl.tools.checkstyle.checks.coding	93.8%	88.9%	91.5%	90.2%
com.puppycrawl.tools.checkstyle.checks	86.9%	90.5%	94.5%	90.3%
com.puppycrawl.tools.checkstyle.checks.blocks	84.7%	93.4%	92.9%	91.3%
com.puppycrawl.tools.checkstyle.checks.imports	89.2%	93.1%	92.5%	92%
com.puppycrawl.tools.checkstyle.checks.sizes	93.8%	91.4%	96.2%	92.6%
com.puppycrawl.tools.checkstyle.checks.whitespace	95.6%	94.6%	97.5%	95.3%
com.puppycrawl.tools.checkstyle.checks.indentation	90.3%	96.9%	97.3%	95.8%
com.puppycrawl.tools.checkstyle.checks.javadoc	95.6%	96.6%	93.1%	95.9%

Report generated by Clover v1.3.1 Thu Aug 26 2004 12:37:26 EST. Site License registered to Cortex eBusiness

Example Report by Clover

The most important thing for the developer is to get feedback in a short period of time about which methods are not covered:



Example Report by Clover

Both graphics can be produced in just a few minutes with the Emma Eclipse plug-in Eclemma.

(Source: LE SWT Bachelor)

4.5.4 Zuerst Testen

Title: Test first

The Test First approach means to write the test first and then the code.

This has important side effects:

- First of all, you have to think about interfaces / interfaces and signatures. * From the point of view of the user of the' Unit'.
- The code is easier to read.
- The code is automatically written with better quality.
- The time it takes to find errors is drastically reduced (fast feedback - see[CIC - Continous Integration / Continous Delivery]].
- Faster error detection means enormous cost savings.

Conversely, units that are implemented first are usually much more difficult to test. Test First changes the design.

Übung CCD-03: Code-Katas

Try this with a simple code-catas.

Time needed: about 35 minutes

Anmerkung

Code-Kata
The term **Kata** means practice and is used in the context of karate. For a simple programming problem, a solution should be found within 30 minutes (up to a maximum of one hour). The solution will then be presented. The goal is to train and improve one's own skills.

4.5.5 Mockups

Use Mocks

Translation programs provide the following explanation for mock objects: dummy, counterfeit, imitation, pseudo-, sham, fake and deceive (see also the learning unit "Testing" from the bachelor's degree).

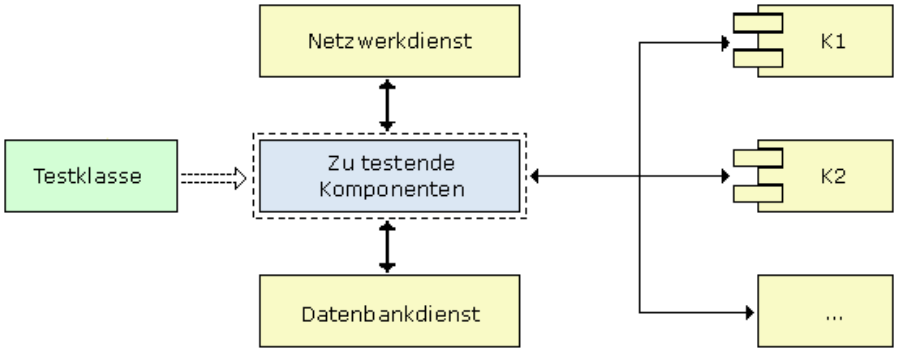
In too many cases, it is not easy to test a unit because the component has too many dependencies. Therefore, it would be desirable for the tests to have test objects that simulate a real environment for a component to be tested. These objects are called mock objects. For example, an object that pretends to be a database. This would then return any result sets, but this is not relevant for the test.

Known mock tools for Java are:

1. mockito
2. EasyMock
3. jmock

Mock tools are available for all languages. These are particularly simple for dynamic languages (e. g. FlexMock in Ruby).

The application of a test or test class to individual service components poses a problem. Service components are usually dependent on other components or services. Mock objects, on the other hand, can isolate the component to be tested and simulate a real environment. The figure below illustrates dependencies between K1, K2, database services, network services and the component to be tested.



Dependencies for a component to be tested

Having a mock at hand makes testing much more efficient. And even if you don't want to test, you can emulate functionality or behavior.

Here are two good links about mocks and stubs and their effects on testing: (thx MC!)

- Martin Fowlers Blog
- Stackoverflow Eintrag

4.5.6 Funktionale Techniken

Title: Functional Techniques

With functional programming techniques, safe programs can be developed, since, for example, freedom from side-effects is important.

See also the learning unitPPD - Programming paradigms.

Important techniques are:

- Use of first-class /Higher Order Functions: e. g. functions as argument or return value; e. g. generation of functions
- Pure Functions: Use of Recursion (or Tail-Recursion)
- Immutable Data: The use of data structures that cannot be changed.

and many other concepts such as lazy evaluation, strict evaluation, type systems, etc.

Often, performance and memory consumption are exchanged here in favor of error-free code or unclear data status. However, this is usually irrelevant as the Bottleneck is located somewhere else. However, it must of course be decided on a case-by-case basis.

4.5.7 Lange Funktionen aufteilen und Reviews

Title: Long functions and Reviews

Split long methods

Method Object

Of course, there are many refactorings at the code level. But only a few have as much impact on readability and changeability as the Method Object Pattern.

Usually, for long functions, the first thing to look for is whether the Extract Method Pattern can be used. However, this is often not simply due to the parameter lists and the naming. Here
Kent Beck
("Implementation Patterns", ISBN-13:978-0321413093) indicates the following procedure:

1. Create a class named exactly like the method: e. g. complexCalculation () becomes the class ComplexCalculator.
2. Create a field in the class for each parameter, variable, and field used in the original method. Leave the names the same.
3. Build a constructor for these variables or parameters
4. Copy the old
   complexCalculation ()
   method to this class. The parameters now become references to the fields!
5. Now replace the body of the original method with code that creates an instance of this class.

Code

```
complexCalculation(){
 new ComplexCalculatior().calculate();
}
```

1. If there were fields in the old method that were used, set them after the call:

Code

```
complexCalculation(){
 ComplexCalculator calculator = new ComplexCalculator();
 calculator.calculate();
 mean = calculator.mean;
 variance = calculator.variance;
}
```

Now check if the code still works. But after Beck, the fun only really starts now, because refactoring is now easy. Extract Method works wonderfully now, since all reference fields are in the class. Fields can often be transformed into local variables. And of course, you can now also delete fields and pass them as individual parameters into the new extracted functions to make them clearer and easier.

Reviews

Code reviews should be an integral part of a quality concept. These are already integrated in XP based procedure models as pair programming. Unfortunately, pair programming is not part of every project.

Therefore, there are often review processes at the end of the work step. Google reports that five' code approvals' are often required for the code to go into production.

Important effects here are important:

- Learning Effects / Continuing Education
- Often 90% of the errors are found, which are later found via expensive debugging. This saves a huge amount of money!
- Code ownership is changing positively.

4.5.8 Messen von Fehlern

Title: Measuring and Tracking Errors

Errors occur during software development. These occur in all phases: misunderstood or unclearly formulated requirements lead to errors as well as incorrect implementations. In the end, everything is a mistake, which leads to the customer receiving software that does not meet his requirements. Iterative approach and reflection are two building blocks that serve to improve the process. However, in order to determine whether an

improvement is actually taking place, there must be a measure of the extent to which a development for the better can be seen.

The measurement of the errors can be carried out by counting or by time measurement. The focus is not on precision as long as the measurement method provides comparable data. The tendency to develop over several iterations should become apparent. Furthermore, it is not a question of clarifying the responsibility for an error. In the end, it doesn't matter who caused the mistake, as long as the team learns from it and improves its process.

Which faults are to be measured? It is not the errors that occur during development. These are unavoidable and hopefully lead to a faultless product being delivered at the end of an iteration. Rather, it is about the errors that are reported back after an iteration by the customer or his deputy (e. g. productowner or support. These are errors that hinder the implementation of new requirements. The errors to be measured are therefore those that occur when you believe that they should not exist.

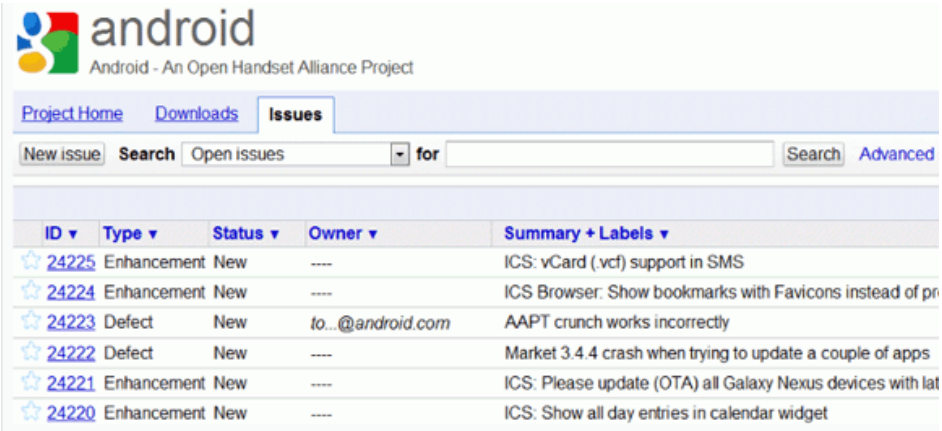
When a team reaches this point in the process and curses, because there is such a flaw in the rest of the work, is to be determined individually for each team.

(C) CCD

Issue tracking systems (or defect tracking) are widely used. All modern code hosters like Google Code, github, Jira Cloud, etc. have integrated these systems.

These systems are available out-of-the-box as a service. So there is no longer any obstacle to deployment.

Here is an example where Android OS is hosted on Google Code:



Issue Tracking System

4.5.9 Statische Codeanalyse

Title: Static Code Analysis

Static Code Analysise (Metrics) / Measure Complexity (green)

The results of a static code analysis are usually extremely informative. These are usually, for example, complexity metrics that provide strong evidence of shortcomings in code design or architecture. For example, there are the following metrics:

- Basic metrics (lines, methods, classes, packages, files, duplicates, etc.)
- McCabe Cyclomatic Complexity
- Halstead metric
- Code Coverage (as pure' visit metrics')
- Test coverage
- Style Injuries
- JDepend
- and coupling metrics

belong to the latter:

- Pressman metrics
- ACD / rARD metric

Anmerkung

rARD Metric
With this metric, each module is initially given the weight 1. Then the weight on which it depends is added for each module.

In addition to the test and readability of high-quality code (which has good metrics), Westphal also describes' evolvability' as a central element of the results of a code analysis or the subsequent refactoring.

4.5.10 Der Technical Debt Deines Produktes

Title: The Technical Debt of a Product

Measure your Technical Debt

Back in 1992

WARD CUNNINGHAM
came up with the idea of describing the number of errors, rule violations and complexity as "technical debt". Freely translated, this means that you are guilty because something can be improved on the technical side.

There are several reasons for this guilt. For example:

- Bad architecture / design
- many code metrics violations
- too much complexity
- over-coupling
- cycles

Some errors may be easier to fix. However, there is also an intrinsic complexity that is usually more difficult to resolve or reduce. Other problem areas are of a long-term (architecture) and some are of a short-term nature (missing JavaDoc), as this can be solved more easily.

The entire problem addresses a topic that is often found in industry. There is the pressure to get a product onto the market quickly. Therefore, a solution "Quick-and-Dirty" is implemented. The product is shipped with its guilt, including the dirty part that is in the code. A loan has been taken up in the same way as in Financial Accounting (debt), and now interest is to be paid. Interest rates are paid at a higher or lower level because the product - as a rule - has to be further developed. However, the higher the number of injuries, the harder it will be to develop and modify the program. For debugging, the developers now have to invest much more time.

The ideal goal for the software vendor is to deliver the product with a debt of zero. The buyer's expectations are to buy a product with the lowest possible technical debt.

This topic is often discussed in forums and other forums. It is also interesting that modern tools try to express the debt as monetary value. This means that all metrics are converted into man-days. How many man-days are needed to eliminate the number of code and architecture metrics errors? In turn, man-days can be converted into monetary value - depending on the wage level.

The decisive factor is that - as already mentioned in the introduction - software costs do not only consist of development costs, but that in individual cases they result in much higher maintenance costs, training costs, change costs, etc., which have to be taken into account.

This is done with the Sonar tool as follows:

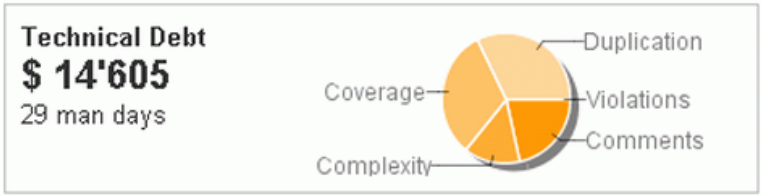
Debt(in man days) = cost_to_fix_duplications + cost_to_fix_violations + cost_to_comment_public_API + cost_to_fix_uncovered_complexity + cost_to_bring_complexity_below_threshold

As well we have:

Duplications	= cost_to_fix_one_block * duplicated_blocks
Violations	= cost_to_fix_one_violation * mandatory_violations
Comments	= cost_to_comment_one_API * public_undocumented_api
Complexity	= cost_to_split_a_class * (class_complexity_distribution >= 60)

Impact Factors for the Technical Debt Calculation

The number is intended to motivate those responsible or developers to invest in quality. This is particularly useful if the Technical Debt does not result in insignificant "interest payments" (expenses).



Sonar - Calculation of Man-Days

Of course, it is clear that this absolute number is very likely to have nothing to do with reality. There may even be cases where a high technical debt is not so bad. And here too, it's like the JDepend metrics. It's not the absolute number that counts, but the experience gained with such graphics and numbers (which can be further detailed at any time with a simple click).

Such metrics and tools sensitize for the quality factor. This is usually just as important as being the first to be on the market with a product. Quality is one of the top priorities when it comes to dominating the market in the long term and asserting itself against the competition.

Daily reflections

Good developers are characterised by being able to look at themselves from a bird's eye view. And asking yourself a lot of questions:

- What did I encode?
- How did I code?
- Did I set myself realistic goals?
- Did I implement Clean Code?
- Why could some things not be implemented?
- Did I learn?

4.6 Architecture und Class Design

Besser Instantiieren als Vererben (rot)

Favour Composition over Inheritance (FCoI)

Text will follow...

Pro Komponente nur eine Aufgabe (orange)

Single Responsibility Principle (SRP)

Text will follow...

Aufgabentrennung(orange)

Separation of Concerns (SoC)

Text will follow...

Text 1 (gelb)

Interface Segregation Principle (ISP)

Text will follow...

Geheimnisprinzip durchsetzen (gelb)

Information Hiding Principle

Text will follow...

Inversion of Control Container (grün)

Use Dependency Injection

Text will follow...

Verwende Dependency Injection (Framework!) (gelb)

Dependency Inversion Principle

Text will follow...

Text 2 (gelb)

Liskov Substitution Principle

Text will follow...

Text 3 (gelb)

Komplexe Refaktorisierungen

Text will follow...

Text 1 (grün)

Open Closed Principle

Text will follow...

Text 2 (grün)

Tell, don´t ask

Text will follow...

Text 3 (grün)

Law of Demeter

Text will follow...

Prüfe die Anwendung von Basispatterns

Check all Basic Patterns: Pluggable Selector, DSL pattern, ...

Text will follow...

Entwurf und Implementation überlappen nicht (blau)

Design and Implementation must not overlap

Text will follow...

Implementation spiegelt Entwurf (UML2Wall) (blau)

Implementation corrosponds Design

Text will follow...

Visualisiere, Messe & Constraine die Architektur (Sonar, SonarJ, JDepend,..)

Visualize, Measure and Constrain your Architecture

Text will follow...

Prüfe den Einsatz von Einterprise Patterns

Check Enterprise Patterns

Text will follow...

4.7 Packages

Put together what belongs together

Common Closure + Common Reuse Principle (CCP, CRP)

This principle means that a class should be designed for this:

A) not to be changed

B) but may indeed experience extensions

Beispiel

(ToDo)

Class: Price calculation

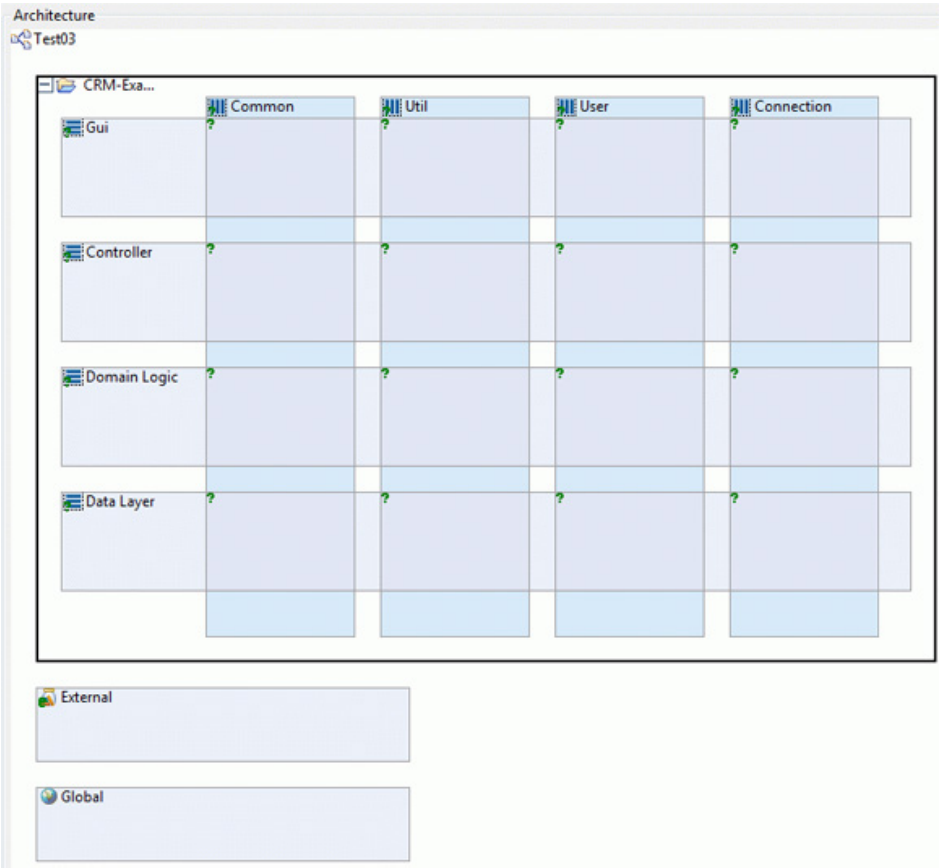
Class: Price calculation incl. Net amount with interface before

Reference: http://www.objectmentor.com/resources/articles/ocp.pdf

Visualize Package Dependencies

Dependency graphs of packages

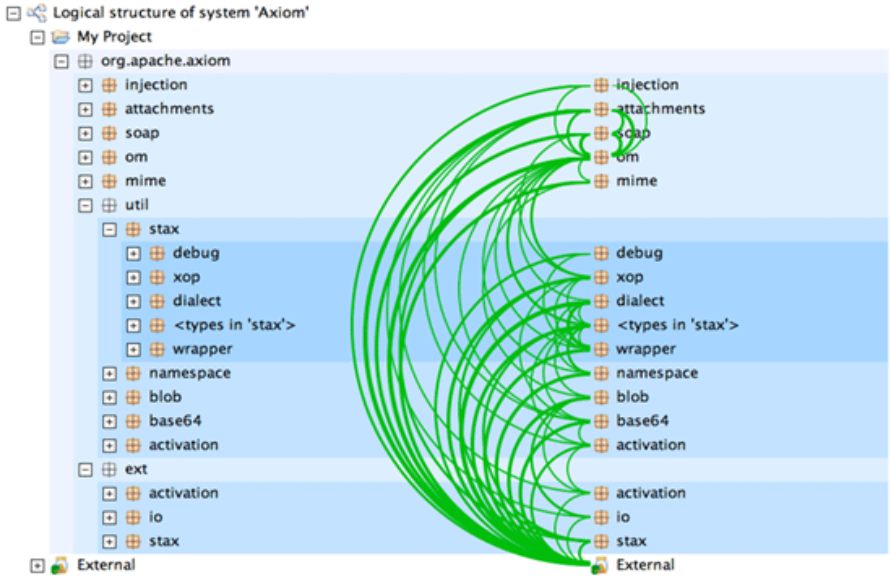
The representation of components as grids is common and helpful for the visualization of components. This display can be used to check whether packages and components really only access down or to the right! Access to the top or possibly also to the left should be avoided, as there is a risk of unchangeable spaghetti code.



Definition of an Architecture

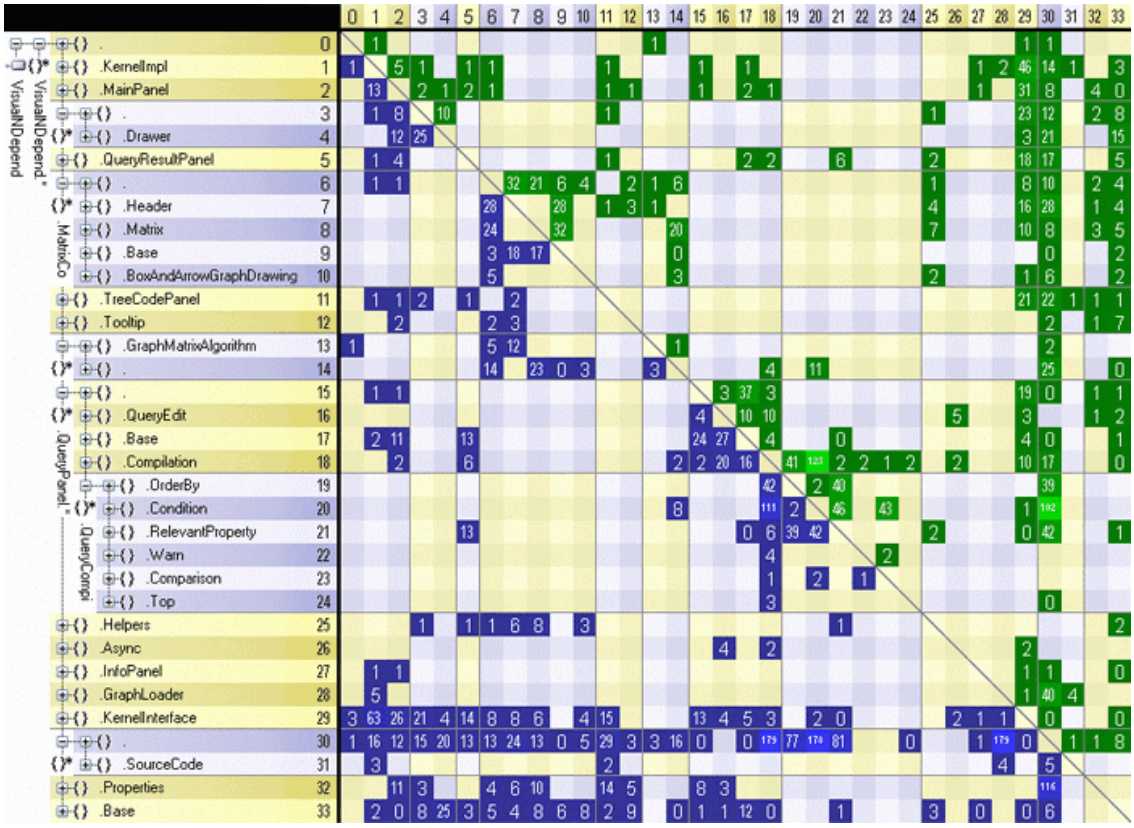
Of course, there are many forms of representation for components and their references.

Below is a graphic created with SonarJ specifically for packages:



Sonar

Below is a picture of NDepend:



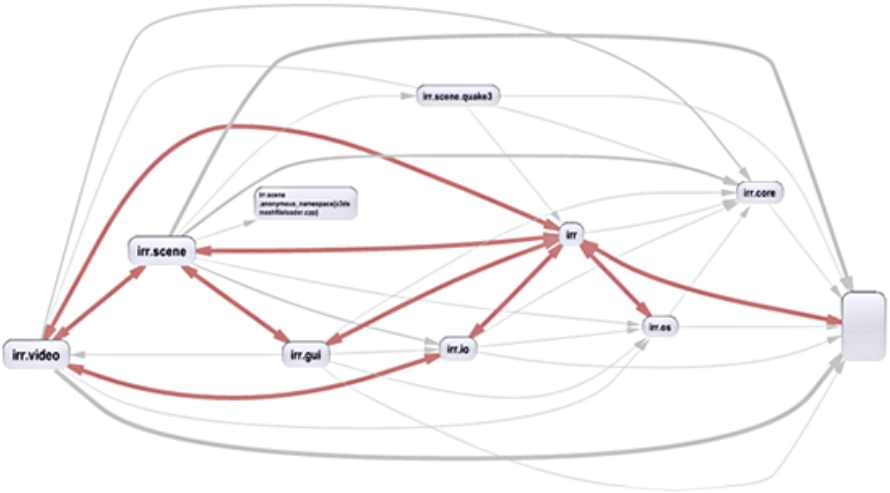
Dependency

This representation is almost equivalent to the second representation in terms of content.

The Dependencies of Packages mus be free of Cycles

Acyclic Dependencies Principle (ADP)

According to the principles of JDepend, all packages must be cyclic free. Experience has shown that the changeability and evolvability of large software with cycles decreases dramatically.



cycles

The graph shown here contains many cycles. He would certainly have a rACD metric of 50%-100%.

Anmerkung

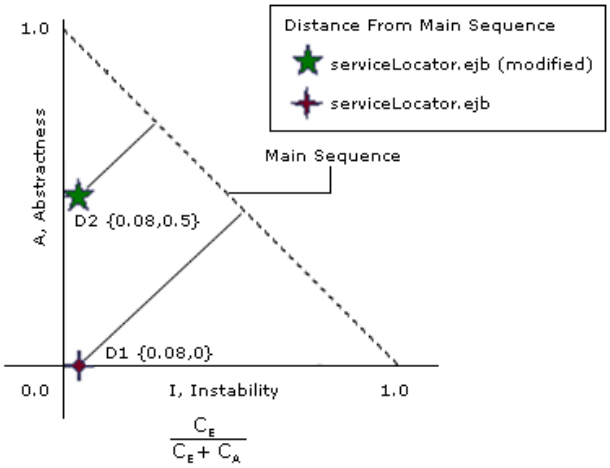
rACD (Relative mean component dependency)
The ACD metric is calculated by dividing the sum of all values by the number of nodes. This value is divided again by the number of nodes (or their square) and results in the rACD value.

In this case, it may be advisable not only to carry out a visual analysis, but also to include the freedom from cycles in the form of tests. Tools such as Macker (http://innig.net/macker/) provide Ant Tasks to enforce architecture enforcements.

A) Unstable packages depend on more stable packages / Stable Dependencies Principle (SDP)

B) The more stable, the more abstract the packages or components / Stable Abstractions Principle (SAP)

Please refer to the JDepend metrics for background information on this topic.



JDepend

In these principles, it is therefore important to look at the Ca and Ce metrics and calculate the values for the packages from them:

- Stable Packages: Usually more abstract packets, i. e. A = AC/CC+AC is high.
- Unstable packages: Are usually completely concretely implemented packages, i. e. I = Ce / (Ce + Ca) is also high.

4.8 Produktivität

Die Pfadfinderregel beachten (rot)

The Scout Rule

besser hinterlassen / broken windows!

Bekämpfe Ursachen. nicht die Wirkung (rot)

Root Cause Analysis

Text will follow...

Komponentenorientierung (blau)

Think and create components

Text will follow...

Keine goldenen Wasserhähne (blau)

You Ain´t Gonna Need It (YAGNI)

Text will follow...

Prinzip der Kleinstmöglichen überraschung (gelb)

Principle of Least Astonishment

Text will follow...

Ein Versionskontrollsystem einsetzen (rot)

Use a Version Control System

Text will follow...

Issues und Fehler werden (öffentlich) Verwaltet (orange)

Issue & Defect Tracking

Text will follow...

Effektives Buildmanagement

Use a Buildmanagement Tool

Research and industry today agree that building management is necessary from just a few classes if a serious program is to be created and more than one developer is working on it. Documentation generation, testing, creating the binaries, etc. cannot be done manually any more and can no longer be done consistently in the IDE after two developers.

Tools like:

1. Ant
2. Maven
3. rake
4. Gradle
5. (or commercial tools)

are therefore mandatory.

Automatisierte Integrationstests (orange)

Use automatic integration tests

Text will follow...

Continuous Integration / Delivery (grün / blau)

Have a continous Integration+Delivery System

ToDo...

Lote aus, ob ein DSL-Einsatz sinnvoll sein kann

Check if DSL can be useful

The coupling of components can take place in many ways:

- with direct function call
- via a communication protocol (e. g. REST)
- Event based
- and much more

However, if developers themselves or modules are to use other components with parameterized parameters, the use of aDSL can also be useful.

There are many examples of this. LINQ is - like SQL itself - such a query language, which is embedded directly into the system or programming language (like C#).

With the increasing ease with which dynamic programming languages such as Ruby or other tools (such as those of IntelliJ) can be used to create or generate DSL, it's definitely worth taking a look at.

The consideration is: how does module A want to use or manipulate module B?

4.9 Management

Kill Distraction

Kill Distraction

As a qualified developer, you also have to think about possible distractions. Nowadays, concentrated work over a defined period of time is hardly possible any more. The multitude of e-mails and events on the social web demand constant distraction and often an immediate reaction. There are many literature and references on this topic on the Internet.

The book below is highly recommended. It is about working concentrated over a defined period of 25 minutes and eliminating all distractions during this period.

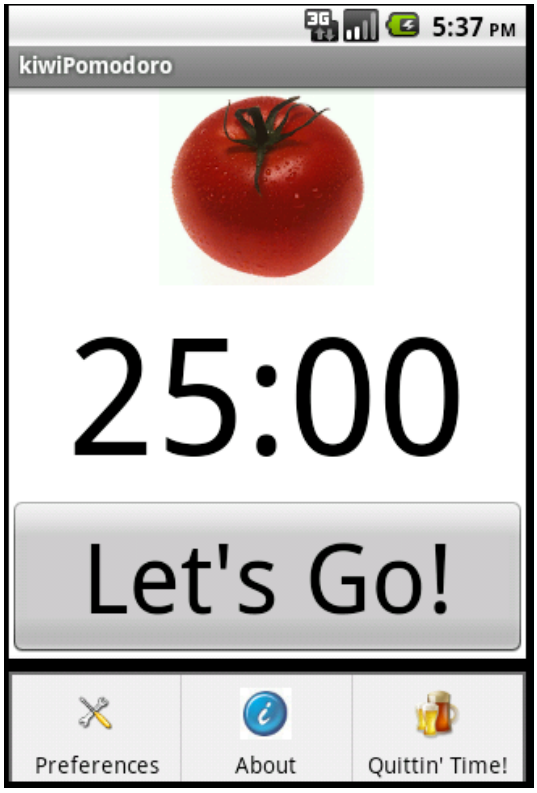
Pomodoro Technique Illustrated: **"Can You Focus - Really Focus - for 25 Minutes?"** Pragmatic Bookshelf, ISBN-13: 978-1934356500

Referenz: http://www.pomodorotechnique.com

Übung CCD-04: Google „Focus“

Read more about the topic on the given website or google. You may even be able to download an app for it.

Time needed: 10 Minuten



App: Pomodoro

Fast Launcher (Ford)

Fast Launcher (Ford)

From Neil Ford there is the hint to always have a fast app launcher at hand.

This certainly applies to both web links and applications.

Become a wizard in Shortcuts

Use Keys & Shortcuts

Some studies on workplace efficiency have shown that a continuous change or break between the mouse and keyboard significantly disrupts the workflow and costs time. Therefore, it is always worthwhile to learn the shortcuts of the current development environment by heart.

Exercise CCD-05: Shortcuts

Which IDE do you use? Which shortcuts do you use to do the following:

1. Format the code?
2. Launch the application?

Stick a label on your monitor to help you with these most important shortcuts.

Time needed : 15 Minuten

Multiple Screens or at least Virtual Desktops

Multiple Screens or at least Virtual Desktops

Similar to the previous example, there are also investigations that the visual switching between the applications costs quite a lot of time. It is not without reason that stock exchange traders have therefore for several dozens of years already begun to use various monitors to distribute the information meaningfully across the entire field of vision. A good 27 inch monitors are already available for little money.

However, if the money is missing, a virtual desktop is a must to be able to switch quickly between completely set up screens.

Become a Scripting Guru

Love Macros & Shell Scripts

Good developers are often lazy people. You try to automate repetitive work steps.

There are many tools for the automation of work steps:

- Build tools / CI/CD tools (e. g. Ant)
- UNIX-Shell & Power Shell
- Scripting languages

Of all these three areas, a developer should have sound knowledge. This does not only include the tools mentioned here under one and the shell of the operating system. Nowadays, scripting languages are just as important. Therefore, the developer should have at least one dynamic scripting language, such as: Python of Ruby.

Love UNIX

Use Unix

Many professional developers work under UNIX. Why are they doing this?

UNIX is much better suited to the work of developers in many respects. Many tools - such as Git - are implemented more efficiently for UNIX than other operating systems. Of course, there are many programs with graphical gimmicks that have been ported to Windows. However, there are professionals who are much faster and more efficient when using Git or Mercurial, for example, if the corresponding commands are executed with the keyboard.

Use the best Editor and IDE

Perfect Editor, Best IDE

Every developer's goal must also be to be as efficient as possible using the best development environment. Here you should ask yourself whether the right tool is used for the current problem. Often more than 1000 € is spent on hardware, but a few Euros for the appropriate software are sometimes no longer available.

In the area of software development with Java, the world with three vendors (Eclipse, NetBeans, IntelliJ) is still quite manageable.

However, it looks different when a text with a certain format (

- . txt, *. xml, *. json, *. cvs

) has to be edited. There are a lot of editors here, which are wonderfully extensible and adapt to the context of the text (UltraEdit, Jedit, notepad++, emacs, vim, etc.).

Know Libraries / Don't Reinvent the Wheel

Know Libraries / Don't Reinvent the Wheel

The title of this tip already says it. As developers, we must not constantly reinvent the wheel. This also includes knowing the most important one or two dozen libraries and using them in the project. Of course, it has to be clarified whether a new library can be added to the project. However, if you have good dependency management, this should not be a problem.

It is exactly this topic that we will address in one of the last sessions. This is a quick look at the most important libraries in the Java environment and why this should be part of your own portfolio (e. g. the old Guava).

Iterative Entwicklung z.B. Scrum

Develop Iterative; e.g. SCRUM

For most of today's projects, iterative development with the appropriate agile project management tools is mandatory.

Unfortunately, however, it is still far too often seen today that projects are carried out without the appropriate tools or visions in terms of agility and iterative approach. What remains is often a management via Excel or web wiki.

Software development is in any case a learning process in which feedback is the most important element to get as close as possible to the final goal.

Lesen, Lesen, Lesen & Weiterbilden

Cultivate your Knowledge

If you have been working in development for a long time, it is often easy to make yourself comfortable in the given work area. You know your area, sit in a (really) safe position and see no need to look beyond the edge of your plate.

Unfortunately, this view is becoming increasingly dangerous and insecure. In a time when everything is getting faster and faster, technology is advancing and people are competing with jobs from all over the world (globalization), the pressure on each individual increases. Why can't a qualified person anywhere else in the world fill his or her own job competently?

In order to survive in this competitive environment, every developer should cultivate many of these sources and pay attention to continuing education:

1. Information through web content (RSS, hackernews, infoQ, dzone, etc.)
2. Information from trade journals
3. Information from good books, eBooks
4. Information from conferences

Visit Events and Learn from the Masters

Visit Events and Learn from the Masters

This point refers to point four of the previous list. No one is perfect in all areas. But learning is more efficient the more you learn from or engage in dialogue with experts. Therefore, the attendance of some specialist conferences and regional groups (JUG) is obligatory.

Share your Knowledge

Share your Knowledge

The open source principle has also played a pioneering role here: in reasonable companies without competitive pressure, it should be a matter of course to pass on special or learned knowledge. And this does not only add value for the company, but also for the learner by enriching their knowledge. Even the process itself, in which experience or knowledge is passed on, allows only a deep penetration of the material and also helps to qualify oneself as a knowledge mediator or trainer.

An important prerequisite for this is an open corporate culture without any power struggles. The latter suffocates every attempt to grow together and achieve common goals.

4.10 Zusammenfassung CCD

Title: Summary CCD

- The term Technical Debt and the consequences resulting from it were introduced.
- You have learned about coding basics (delete code that no longer has authorization, avoid Magic Numbers) and the subcategories:
  - Coding Source (local declaration, explanatory variables, code nesting and separation of multi-threaded code)
  - Coding Conditionals (capsules of complex logic operations, avoid negative logic operations and reveal constraints in one place)
  - Coding Principles (No repetitions alias code copy, simplicity as a guiding principle, ensuring implicit sequences)
- Furthermore, you have learned about various procedures for testing code quality, such as code convention checking, automated unit tests, code coverage analysis, the Test First approach, the use of mock tools and static code analysis (metrics).
- You now know what principles of clean code development exist in relation to software architecture, class design and packages such as task separation and dependency injection, visualize dependency dependencies of packages.
- It was explained which basic principles you should observe with regard to productivity (e. g. no golden taps, effective build management).
- You have learned about ways to work better and more efficiently than software developers (e. g. eliminating distractions, Unix is important, training).

- You now know that it is important to have a good eye for quality at all levels: from source code to architecture or working methods.

4.11 Wissenüberprüfung CCD

Übung CCD-06: Clean Code Development

1. What is the meaning of Clean Code?
2. What is the Technical Debt?
3. Please read www.clean-code-developer.de
4. Create your personal checklist for Clean Code and attach it to the wall at home via your computer. ;-)

Time needed: 60 min

4.12 Literatur CCD

Robert C. Martin:

"Clean Code"

Prentice Hall

Andrew Hunt / Dave Thomas / Ward Cunningham:

"The Pragmatic Programmer"

Addison Wesley

Neal Ford:

"The Productive Programmer"

O'Reilly

Kent Beck:

"Implementation Patterns"

Addison Wesley

Martin Fowlers Bücher über:

Patterns, E-Patterns, Refactoring, Continous Integration, Beyond Architecture, etc.

Joshua Bloch:

"Effective Java"

Prentice Hall

Buschmann, Henney, Schmidt:

"Pattern-Oriented Software Architecture"

Wiley

Weitere Referenzen:

www.clean-code-developer.de

(An deren Entstehung und Verbreitung der Autor beteiligt war)

12 Things Every Programmer Should Know

http://www.javacodegeeks.com

http://www.javacodegeeks.com/2010/12/things-every-programmer-should-know.html

97 Things Every Programmer Should Know

http://programmer.97things.oreilly.com/wiki/index.php/Contributions_Appearing_in_the_Book

O'Reilly

Weitere Weblinks:

- Artikel aus PHP Magazin
- 100 Tools for Developers
- Reducing Incidental Complexity in Our Code and in Our Teaching
- How to be a great Software Developer

5 BUI - Advanced Buildmanagement

Autor: Prof. Dr. Stefan Edlich

5 BUI - Advanced Buildmanagement
5.1 Überblick und Lernziele BUI
5.2 Prinzipien des Buildmanagements
5.3 Praktische Übung mit Maven
5.4 Maven Schnellreferenz
5.5 Praktische Übung mit Gradle
5.6 Gradle Schnellreferenz
5.7 Vor- und Nachteile
5.8 Zusammenfassung BUI
5.9 Wissensüberprüfung BUI

5.1 Überblick und Lernziele BUI

Title: Overview and Learningobjectives

Lernziele

Learningobjectives There are certainly dozens of build management systems on the market. Some are universally applicable and others are specially designed for specific languages. For example, it is noticeable that almost every modern programming language has its own build language. A well-known example is Ruby with the build DSL rake.

Also in the next few years since the first creation of this learning unit (2012) many new systems will emerge and struggle for supremacy in the build area. Presenting the newest and best system in real time is almost impossible and maybe not even necessary. The core element of this learning unit is therefore to present the core principles of the most important systems and to compare them with each other!

As a rule, even experienced software engineers have learned only one build system and use it in daily operation. Other systems are well known from hearsay, but there are only vague ideas.

In addition to a fair comparison of the performance of such systems, it is therefore also the aim of this learning unit to transfer practical experience to at least three systems. Most people are already familiar with the standard Ant system from their bachelor's degree. Gradle and Maven are two more strongly industry-relevant systems, which are presented in more detail here and are also suggested in the exercise.

However, the student is free to include other systems such as rake or. NET based build systems in the tutorial / submission task. Other students could then also benefit from this.

Gliederung

Gliederung

- Introduction
- History and principles of build management
- Practical exercise with Maven and Gradle
- Presentation of the advantages and disadvantages of Maven and Gradle
- Knowledge review

Zeitumfang

Time Requirements

Reading the session takes less than an hour.

The exercise takes at least one hour for each build system.

REFERENZEN:

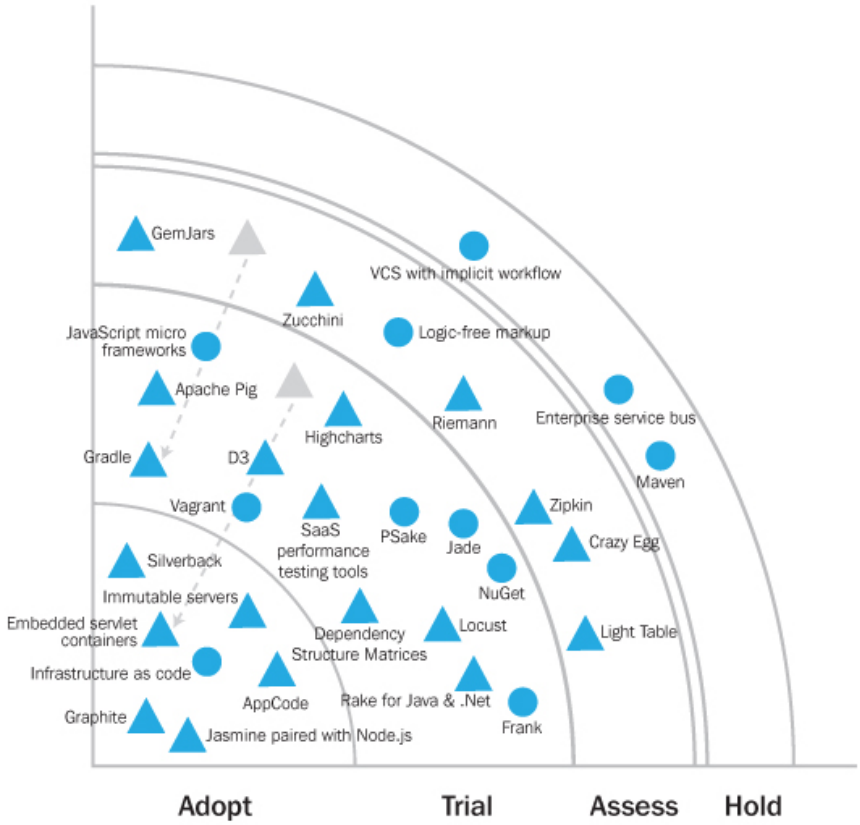
- Old Comparison

5.2 Prinzipien des Buildmanagements

Looking at the history of build management, one of the questions that arises from this is: what principles can be derived from it?

Tools and APIs are subject to a continuous change process. They can hardly always be introduced here. In the following chapters, the most important principles are therefore outlined and a selection of tools are briefly introduced and evaluated.

Remark: Which build management systems are relevant?



Radar
The Technology Radar by Thoughtworks from October 2012
© 2014 ThoughtWorks, Inc.

As you can see, Ant is no longer included as an industry standard. Maven is mentioned, but he's on hold. Gradle wanders from Assess to Trial.

In hindsight we see that Maven and Gradle made it. But due to performance reasons, many companies as Google or Facebook created their own new buildmanagement systems. Have a look: https://bazel.build

5.2.1 Buildmanagement in den 80er/90er Jahren

Title: Buildmanagement in the 80th and 90th

As there were not many languages with a high 'Expressiveness' people built Script alike DSLs (Domain Specific Languages).

Here is an example from the 80s:

Example:

Code

```
all: hello

hello: main.o factorial.o hello.o
 g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
 g++ -c main.cpp

factorial.o: factorial.cpp
 g++ -c factorial.cpp

hello.o: hello.cpp
 g++ -c hello.cpp

clean:
 rm -rf *o hello
```

Obviously, this makefile already contains some things that were considered important in the 90s:

- **Targets** (Goals that describe a product / artifact to be created)
- **Dependencies**

Why isn't that enough? What's missing?

The language, i. e. the closed DSL is not expandable. The language is not powerful. A lot has to be done by hand.

Nevertheless, even today, large systems with make are created. Take a look at the makefile of Redis on GitHub.

5.2.2 Buildmanagement im XML/ANT Zeitalter

Title: Buildmanagement in the XML/ANT age

With the penetration of XML since 1998, ANT has established itself as the leading platform for a decade.

Vertiefung

Tipp:
Further inforamtion on Ant will be given later and examples with screenshots too.

Reference to the manual: http://ant.apache.org/manual/index.html

Es simple example from the manual:
Code: © ant.apache.org

Code

```
<project name="MyProject" default="dist" basedir=".">
 <description>
 simple example build file
 </description>

 <property name="src" location="src"/>
 <property name="build" location="build"/>
 <property name="dist" location="dist"/>

 <target name="init">

 <tstamp/>

 <mkdir dir="${build}"/>
 </target>
```

```
 <target name="compile" depends="init"
 description="compile the source " >

 <javac srcdir="${src}" destdir="${build}"/>
 </target>

 <target name="dist" depends="compile"
 description="generate the distribution" >

 <mkdir dir="${dist}/lib"/>


      <jar       jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
 basedir="${build}"/>
 </target>

 <target name="clean"
 description="clean up" >

 <delete dir="${build}"/>
 <delete dir="${dist}"/>
 </target>
</project>
```

Let us also analyse the advantages and disadvantages here.

Advantages:

- Well machine readable
- Good extensible

But there are enough **Disadvantages**:

- It is *verbose*, i. e. chatty and therefore not very easy to read by people.

Even with the help of assistants, you'll always be somewhere with forgotten / or ".

- Ant is not powerful; i.e. not turing complete.

Der letzte Punkt bedarf einer Erläuterung:

Ant is of course powerful in that any new tasks can be defined in a simple way. This is done in Java and is packed in XMl. This makes the tasks themselves Java-powerful, which is not bad.

Nevertheless, the language - which Ant defines - is limited to the XML schema. With the AntStructure task, you can generate a DTD (Document Type Definition) yourself, but this is also incomplete because attributes often depend on parameters. This makes it clear that a DTD power is far from the power of a programming language, i. e. not even Turing complete. There are the contributed tasks if / else, but no jumps or loops. And properties are not variables, but final constants.

Nevertheless, Ant is still a powerful industry standard that will last for a long time to come.

The following section will therefore explain which build languages can have the power to saw at the Ant (or NAnt) chair.

5.2.3 Prinzipien: Convention, Dependencies und Plug-In

Title: Principles

Maven likes to be portrayed as a much more powerful tool than a *bare* build tool. In addition to the standard tasks clone, compile, package, test, deployment, there are more project-supporting tasks in the foreground, such as the working framework, bringing the team together, reporting, etc..

This already defines the unique selling propositions:

1. Standard frame
2. Dependency Management
3. Project lifecycle
4. Plug-in architecture

All this is controlled by the Project Object Model (POM).

Principle: Convention over Configuration

The principle says, *Use default settings, then you have no configuration work.* Only in the case of justified deviations shall these be specified. However, deviations are always possible.

Ruby on Rails from David Heinemeier-Hansson has taken this principle to its peak. Here, an MVC environment is automatically generated, to which the developer must adapt. This is standard for a lot of tools today; especially web-frameworks.

Maven also has tests, resources, program code and the result in defined folders.

This is partly also referred to as standard frames. Not only locations but also processes - such as tests - are standardized.

Maven Plug-Ins

Maven itself is just an XML parser that contains only a few build-in plug-ins. All functionality is then integrated with the plug-ins. All this plug-in functionality, but also the dependencies are downloaded from central repositories (see e. g. ***http://search.maven.org***). The plug-in knowledge, the structure, dependencies and the location of the libraries are stored there.

Dependency Management

With unique identifiers for the dependency group, artifact and version number, the current version and the corresponding dependencies can be automatically determined and downloaded from the repository.

Archetypes

Archetypes are templates for well-known tasks such as J2EE or web development. They help to establish a correct and uniform project structure in companies and can be adapted.

Conclusion:

Many projects rely on Maven instead of Ant to keep configuration effort to a minimum. Maven POM files are usually much smaller than ANT files.

5.2.4 Prinzip: Top DSL

Title: Principle: Top DSL

Gradle is based on a Groovy DSL and was created by Hans-Dockter.

It also includes some of Maven's principles:

- extensive dependency management
- defaults = Convention over Configuration

Let's have a look at a hello world:

Hello World:

Code

```
task hello << {
 println 'Hello'}}

task world(dependsOn: hello) << { print " World"}
```

- You can already see here that it is quite flexible and extensible due to a syntax similar to programming languages.
- It provides many plug-ins for Java, Groovy, Scala, Web-Projects and similar.
- It integrates well with Maven, Ant or Ivy.

In addition, there are other interesting features:

- Incremental build management
- Parallel execution of UNIT tests
- A Deamon
- Tasks are dynamic
- Ant tasks are first-class Citizens in Gradle. A complete Ant-Script can be imported and the tasks can be called transparently from Gradle.
- a GUI

A few examples of the last point.

Gradle and Ant:

Code

```
task hello << {
 String hi = "Ant says Hi!"
 ant.echo(mesage: hi)
}

task zip << {
 ant.zip(destfile: all.zip) {
 fileset(dir: src) {
 exclude(name: **.class)
```

```
 include(name: **.java)
 }
 }
}

ant.importBuild 'build.xml'
```

Furthermore, many source directory sets can be defined for which different dependencies, documents and even different JDKs can be defined.

Under Ant, the definition and referencing of files was not always trivial. Under Gradle, this is not shorter, but it is clearly understandable:

Code

```
File configFile = file(project1/coredata.json')

FileCollection collection =
   files('res/dateiA.txt',new   File('res/dateiB.txt'),['res/
dateiC.txt',
 'res/datei.txt'])
```

Finally, a comparison to Ant:

Code

```
ant.copy(todir: 'build/img') {
 fileset(dir: 'resources/img')
}
task copyTask(type: Copy) {
from 'src/de/edlich/swt/build‘
 into 'build/src/de/edlich/swt/‘
 include '**/*.rb‘
 exclude { details ->
 details.file.name.endsWith('.yaml') &&
```

```
details.file.text.contains('beta')
 }
}
```

Which projects use Gradle?

Groovy, Grails, Hibernate, Spring (Security, Integration), Canoo, Griffon, Gant, Aluminium, etc.

So you are in good company.

What plug-ins do exist?

- Java, Groovy, Scala, Antlr
- ear, jetty, maven, osgi, war
- checkstyle, codenarc, eclipse, findbugs, jdepend, pmd, sonar
- and many more...

5.2.5 Weitere Buildmanagementsysteme

Title: Some more Buildmanagementsystems

Rake

Rake had been build by Jim Weinrich. Rake is also an interesting DSL in Ruby, which is strongly dependent on Dependencies. Unfortunately, there are no rake books. But there are some rake tutorials.

An Example:

Code

```
task :turn_off_alarm do
 puts "Turned off alarm. Would have liked 5 more minutes,
 though."
 end
task :groom_myself do
 puts "Brushed teeth."
 puts "Showered."
 puts "Shaved."
```

```
 end
desc "I am describing this task"
task :make_coffee do
 cups = ENV["COFFEE_CUPS"] || 2
 puts "Made #{cups} cups of coffee. Shakes are gone."
 end
task :walk_dog do
 puts "Dog walked."
 end
task :ready_for_the_day
 => [:turn_off_alarm, :groom_myself, :make_coffee, :walk_dog]
 do
 puts "Ready for the day!"
 end
```

(C) Jason Seifer

This means you have the full power of the language at hand.

Apache Ivy

Apache Ivy is a subproject of Apache Ant. The goal of this custom build tool is to help with dependencies and jar dynamically reload libraries. So it's a Dependency Manager.

It can be well integrated in Ant. Can also manage all kinds of dependencies on their own.

As special features you can still find here:

- Dependency Reports
- Erweiterbar
- Transitive Dependencies
- and so on

Two examples from the Apache Ivy tutorial
(C)
:

The dependencies:

Code

```

```

```
<dependency>
 <groupId>commons-lang</groupId>
 <artifactId>commons-lang</artifactId>
 <version>2.0</version>
</dependency>
```

The Buildfile:

Code

```
<project  xmlns:ivy="antlib:org.apache.ivy.ant"  name="hello-
ivy" default="run">

 ...


 <target name="resolve" description="--> retrieve dependencies
 with ivy">
 <ivy:retrieve />
 </target>
</project>
```

With the declaration of the namespace all Ivy tasks become known. With retrieve, the resolved dependencies are drawn from the Maven repositories.

The future

Nowadays nearly every language has its own build language as e.g. Scala with SBT.

Nevertheless the future is on the new systems as Google Bazle or Facebooks Buck!

- https://bazel.build
- https://github.com/facebook/buck

Please have a look and try to understand the concepts.

5.3 Praktische Übung mit Maven

Title: Practical Excercises with Maven

Maven install & run

Maven requires Java, of course. A 1.5 installation should be available at least.

Testing the installation:

```
C:\Users\Edlich> java -version
java version "1.7.0_03"
Java(TM) SE Runtime Environment (build 1.7.0_03-b05)
Java HotSpot(TM) 64-Bit Server VM (build 22.1-b02, mixed mode)
C:\Users\Edlich>
```

Download the Maven Version (check for the correct version) is done from this adress: http://maven.apache.org/download.html

The installation details for all operating systems are easy to google.
The package is unpacked and then the correct paths are set.

For Windows:

```
F:\CODING> set M2_HOME=F:\PROGRAMME\apache-maven-3.0.4
F:\CODING> set PATH=%PATH%;%M2_HOME%\bin
```

Then Maven should be ready.

Test:

```
F:\CODING>mvn -v
Apache Maven 3.0.4 (r1232337; 2012-01-17 09:44:56+0100)
Maven home: F:\PROGRAMME\apache-maven-3.0.4
Java version: 1.7.0_03, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_03\jre
Default locale: de_DE, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

Now you can generate a Template (Archetype), if you have internet!

```
F:\CODING> mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Now Maven downloads myriads of libraries, which can take a few minutes.

Here one already suspects a common point of criticism from Maven: Maven loads something out of the net forever and can hang itself thereby also times, if resources are not available. Here you have to do a little bit of work to run Maven in offline mode.

The following structure is then created:

File-Structure

A Java template is also generated:

```
App.java
package de.vfh.vfhvws.maveme;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Java Template

A corresponding test file is also generated.

Now you can easily start with

> mvn compile // (ca. 32 sec)
> mvn package // (ca.29 sec)

to compile and create a jar. A target directory is created, which contains the classes and other folders.

The whole thing can then be done:

```
F:\CODING\my-app\target>java -cp my-app-1.0-SNAPSHOT.jar
de.vfh.vfhvws.maveme.App
Hello World!
F:\CODING\my-app\target>
```

Finally you can clean up with the following line

```
> mvn clean
```

. Das target directory will be deleted.

5.4 Maven Schnellreferenz

Title: Maven Quick References

References:

Maven: The complete Reference
http://www.sonatype.com/books/mvnref-book/reference/

E-Book: Better Builds with Maven
http://www.scribd.com/doc/238927/Better-Builds-With-Maven

Video Tutorial Maven
http://www.youtube.com/watch?v=al7bRZzz4oU

5.5 Praktische Übung mit Gradle

Title: Practical Excercises with Gradle

Gradle "up and running"

First of all, make sure that Java ≥ 1.5 is installed and can be found in the path.

After downloading *gradle-VERSION-all. zip* from www.gradle.org, Gradle must be unpacked.

Then GRADLE_HOME/bin must be added to the PATH environment variable. (For Windows we have path editors like hier)

If PATH has been extended accordingly, it must be tested whether the installation was successful and Gradle is found:

```
F:\CODING>gradle -v
```

Gradle 1.0-rc-1
Gradle build time: Mittwoch, 11. April 2012 11:13 Uhr UTC
Groovy: 1.8.6
Ant: Apache Ant(TM) version 1.8.2 compiled on December 20 2010
Ivy: 2.2.0
JVM: 1.7.0_03 (Oracle Corporation 22.1-b02)
OS: Windows 7 6.1 amd64

Unlike Ant, Gradle only knows projects and tasks. But not the term "targets" that correspond to Gradle tasks.

Now we edit a Hello World:

Code

```
Dateiname: build.gradle

task hello { // Task
 doLast {
 println 'Hello SWT-Master!'
 }
}
```

build.gradle
is the filename analog to the
build.xml
from Ant.

We execute this with
gradle hello
or with gradle -q hello.
-q(quiet) supresses system messages from Gradle.

5.6 Gradle Schnellreferenz

Title: Gradle Quick Reference

Quelle: (C) Gradle User Guide

Quicker Definition

Code

```
task hello << {
 println 'Hello world!'}
```

The println is added here dynamically with the following code:

Code

```
hello.doFirst { // add
 println 'Hello Venus'}
hello.doLast { // add
 println 'Hello Mars'}
hello << { // add
 println 'Hello Jupiter'}
```

would yield
Hello Venus
,
Hello world!,
Hello Mars
und am Ende
Hello Jupiter ausgeführt.

List of all Tasks

A list of all self-defined (and build-in) tasks can be obtained by calling gradle tasks.

Example:

F:\CODING\GrTest>gradle tasks

tasks

All tasks runnable from root project
Help tasks

dependencies - Displays the dependencies of root project 'GrTest'.
help - Displays a help message
projects - Displays the sub-projects of root project 'GrTest'.
properties - Displays the properties of root project 'GrTest'.
tasks - Displays the tasks runnable from rot project 'GrTest' (some of the displayed tasks may belong to subprojects).
Other tasks
copyfile
delfile
To see all tasks and more detail, run with --all.
BUILD SUCCESSFUL
Total time: 2.574 secs
F:\CODING\GrTest>

Using Groovy Code

Code

```
task upper << {
 String someString = 'mY_nAmE'
 println "Original: " + someString
 println "Upper case: " + someString.toUpperCase()
}


task count << {
 4.times { print "$it " }
}
```

Obviously, all variables can be defined and reused somewhere. This would have killed global and local properties. System and command line properties are addressed later.

Dependencies

Code

```
task hello << {
 println 'Hello world!'}
task intro(dependsOn: hello) << {
 println "I'm Gradle"}
```

Dynamic Tasks

Code

```
4.times { counter ->
 task "task$counter" << {
 println "I'm task number $counter" }
}
```

Is called with:

> gradle -q task1 // oder task0, task2, task 3, but not task 4 !

Tasks can be created at runtime!

Each task is an object and has extra properties

The attributes of tasks can be accessed as follows:

Code

```
task hello << {
 println 'Hello SWT!'}


hello.ext.geschmack = "Banane" // meine Properties
hello.doLast {
 println "Greetings from the $hello.name task with flavour
 $hello.ext.geschmack."}
```

Ant Integration

Ant is wonderfully integrated. So you have all the ant-tasks at your fingertips! Here are two examples:

A) The Ant-Task Loadfile loads a file. Ant often has properties in there. But it can be anything. We read in the file and output the file name and content:

Code

```
task loadfile << {
 def file = file('Textfile.txt')
 ant.loadfile(srcFile: file, property: file.name)
 println "Filename = $file"
 println "Properties = ${ant.properties[file.name]}"}
```

What's interesting here?

We call the Ant Task with ant. open. The parameter attributes are available with colon. For example, e. g. srcFile:: and after that comes the value. Just like in the Ant documentation.

Now something new. E.g. **copy**:

Code

```
task copyfile << {
 ant.copy(file: 'Textfile.txt', tofile: 'IchKopie.txt')
 println "Done!"
}

task delfile << {
 ant.delete(file: 'IchKopie.txt')
 println 'IchKopie.txt getoetet'
}
```

So if the file actually exists, it can be copied with gradle copyfile and deleted with gradle delfile

.

Interestingly, you can define the file in
tofile value
in different ways: //tofile: tofile // mit def tofile = file('IchKopie.txt')
//tofile: tofile.name // with the definition above
//tofile: 'IchKopie.txt'

More Power with Methods

Code

```
task methoddemo << {
 println "Ergebnis: " + doubleMe("Sahne")}

String doubleMe(String arg) {
 arg + "+" + arg
}
```

So I can use any method anywhere in any task.

Default Tasks

This is how to define the default tasks:

Code

```
defaultTasks 'clean', 'run'

task clean << {
 println 'Default Cleaning!' // will run}

task run << {
 println 'Default Running!' // will run}

task other << {
 println "I'm not a default task!" // will cry}
```

Gradle GUI

> gradle --gui

Features:

- Tree view
- Launcher
- Favourites (for long buildfiles)
- Pass parameters
- Browse new buildfiles, loglevel, filter, etc.

Eclipse

Unfortunately, there doesn't seem to be a nice Eclipse plug-in. So you can only use the GUI or Gradle as external tools under Eclipse:Weblink: Running Gradle from Eclipse
.

The task is transferred here in this configuration via Prompt. A tree view is not available in Eclipse, but might be now.

But IntelliJ Idea has Gradle support possibly long before 2017.

5.7 Vor- und Nachteile

Title: Advantages and Disadvantages

Even though Ant is still the owner of many build systems, it's time to look at the advantages and disadvantages of the new systems:

Maven:
+ Whoever does not deviate from the default has an easy life
+ Good ecosystem
+ At least two free books

- Reading and understanding XML POMs is no fun
- Expressiveness only via XML parameter
- Better with a running internet (even if offline works...)

Gradle:
+ Yery expressive Language
+ Great docs
+ Many additional features (Deamons, GUI, parallel, dynamic Tasks, etc.) + At least 5 books

- Weak Eclipse integration (2013)

In one sentence: (and simplified in a striking way)

- XML Haters will be happy with Gradle.
- XML Lovers feel at home in Maven.

And of course it makes sense to look at the specific build systems of a language if you stay completely in that language.

Like rake for the Ruby ecosystem.

5.8 Zusammenfassung BUI

Title: Summary BUI

- You have received a brief overview of the history of build management and know the basic principles of build management systems.
- In practical exercises you have met Maven and Gradle.
- You now know the pros and cons of Maven and Gradle.

5.9 Wissensüberprüfung BUI

Übung BUI-01: ANT

Welche Vor- und Nachteile hat ANT?

Lösung zeigen

Vorteile:

- Gut maschinenlesbar
- Gut erweiterbar

Nachteile:

- Es ist verbose, d.h. geschwätzig und daher nicht besonders gut von menschen lesbar.
- Selbst mit Hilfe von Assistenten, wird man immer irgendwo mit vergessenen / oder " hängen.
- Es hat eine geringe Mächtigkeit.

Übung BUI-02: Gradle

Auf welchen Prinzipien basiert Gradle? Nennen Sie zwei.

Lösung zeigen

- umfangreiches Dependency Management
- Defaults = Convention over Configuration

Übung BUI-03: Maven

Auf welchen Prinzipien basiert Maven?

Lösung zeigen

- Convention over Configuration, d.h. z.B. vordefinierte Locations und Einstellungen
- Dependency Management: Das System (die Referenz) kennt alle Abhängigkeiten und kann diese selbst auflösen.

6 DVC – Distributed Version Control Systems Vertiefung

Autor: Prof. Dr. S. Edlich

6 DVC – Distributed Version Control Systems Vertiefung
6.1 Überblick und Lernziele DVC
6.2 Einführung und Literatur DVC
6.3 Einstieg in Git
6.4 Grundlegende Arbeit mit Git
6.5 Git Kommandos en Detail
6.6 Zeitreisen in Git
6.7 Git Glossary
6.8 Mercurial
6.9 Kommandoübersicht Git
6.10 Zusammenfassung DVC

6.1 Überblick und Lernziele DVC

Title: Overview and Goald DVC

In the bachelor's degree in general and here in the software engineering module, the basics of version management procedures have already been laid and all systems with relevant market share have been presented.

This learning unit is now all about mastering the leading system and making optimal use of the associated PaaS service.

After an introduction, the work with Git will be discussed in more detail. Afterwards it will be shown how this is used in combination with GitHub.

This is a conscious departure from the principle of teaching principles and practicing systems. However, due to the overwhelming market share of GitHub - the so-called social coding - the handling of these systems is becoming more and more important.

Therefore, the reference to PaaS and the above-mentioned social coding is also made in this learning unit.

Lernziele

The goal of this learning unit is to master Git and GitHub so confidently that software development can be started immediately and efficiently in a team.

Gliederung

Gliederung The learning unit is therefore divided into the appropriate levels:

- Introduction to Git
- Basic work
- Branching in git
- Distributed environments with Git
- Tools and Git

Zeitumfang

Zeitumfang
You will need approx. 60 minutes to work through the learning unit and approx. 110 minutes to work through the exercises.

6.2 Einführung und Literatur DVC

Title: Introduction and References DVC

Definition

> Git is a version control program that can track, manage and share complete directory trees. GitHub is a web service that acts as a git-host (sometimes free of charge) and can share it with interested parties.

At the beginning of the millennium after make, RCS, CVS was for a long time the market leader in subversion (33.4% in 2009 according to Forrester Research), Git has taken the lead.

With GitHub and GitLab - to name just two of many hundreds - PaaS Services (Platform as a Service) services have been created, which usually offer free storage space and thus allow you to start immediately and free of charge. Bitbucket even offers both protocols since 2012 Git and Mercurial to avoid the question of a better system.

The advantages of development with Git and under the PaaS GitHub have been underlined by several facts:

- In 2012, GitHub received $100 million in venture capital funding

Countless companies and projects rely on GitHub: Facebook, Twitter, Microsoft (!), vmware, redhat, LinkedIn, Mozilla, Android, Debian, Eclipse, jQuery, Lunix, PostgreSQ, RubyOnRails

Auszug aus typical Programmer, der beschreibt, was GitHub heutzutage ist:

TP: "What do you think of github?"

Linus Torvalds: "It started as a place for mothballing unmaintained and unnecessary projects, and that is still most of what is hosted there. But it's turned into a kind of World of Warcraft universe for programmers, where they are ranked by their commits and which projects they have trunk privileges on. I read about a recruiting company built around the idea that GitHub reputation means something, so I guess if you aren't committing to GitHub you won't be getting a job at the coolest startups. The good old days of writing FizzBuzz and moving Mt. Fuji during your interview are over."

GitHub has become the Facebook for coders, where you can collaborate, track, evaluate or contribute.

References

- Git
- "git help"
- Github http://www.github.com
- Reference Documentation

- Book and Videos
- Videos und
- Git Magic auf Deutsch
- Git Cheat Sheet
- Heroku Cheat Sheet
- Git Visual Reference
- Wiki
- Googles Comparison of Git and Mercurial
- (Markdown Reference)
- A successful Git Branching Model
- 8 Reasons to switch to Git
- Git Concepts simplified
- Git Happens :-)
- Git in 600 Words

Books

- Travis Swicegood, "Git", The Pragmatic Bookshelf, 2010, ISBN-13: 978-1-934356-72-2
- Scott Chacon, "Pro Git" (Frei + Online)

Online References

- Ten Things You Didn't Know about Github and Git

6.3 Einstieg in Git

Title: Introduction to Git

Unlike many other version control systems, Git does not work completely on the principle that only the differences in file changes are saved. Instead, Git takes a **snapshot** of files much more often. This has a number of important advantages over the old approach, especially in terms of branching.

Furthermore, Git is strongly focused on security and integrity. Unlike Mercurial, for example - where synchronisation points make life easy - Git calculates a SHA-1 hash value forCommits. These ensure integrity. Subsequent changes or even cover-ups are no longer possible. A restoration also works with these hash values.

The most pleasant advantages of Git are surely:

- except for a few commands, you always work locally. This is an advantage for which you are particularly grateful when travelling (or without the temporary Internet).
- To have the functionality of a time machine.

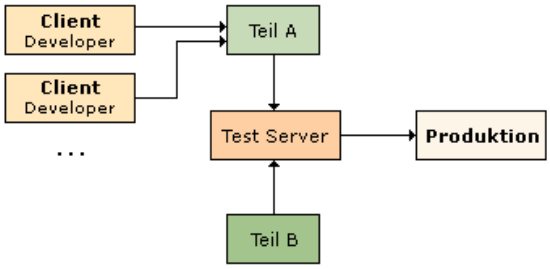
It has already been pointed out in previous sessions that aggressive and courageous programming is important today. Brave changes must be tested and then easily accepted or discarded.

Hinweis

It is just as important to fork courageously! So create branches. This is also excessively promoted with git.

Repository Hierarchy

Git repositories usually form a hierarchy if they belong to the same topic. A tree, then. Usually there are many clones on which development takes place. Then Z. V. a master repository that collects all changes and branches. But this repository can also be part of a project and point to a' MasterMaster' repository. There are no limits to the building structure.



Repo-Architecture
Repository Linking

Anmerkung

Tipp:
If you would like more information about the concept of Git, you can find it under Git-Grundlegendes Konzept.

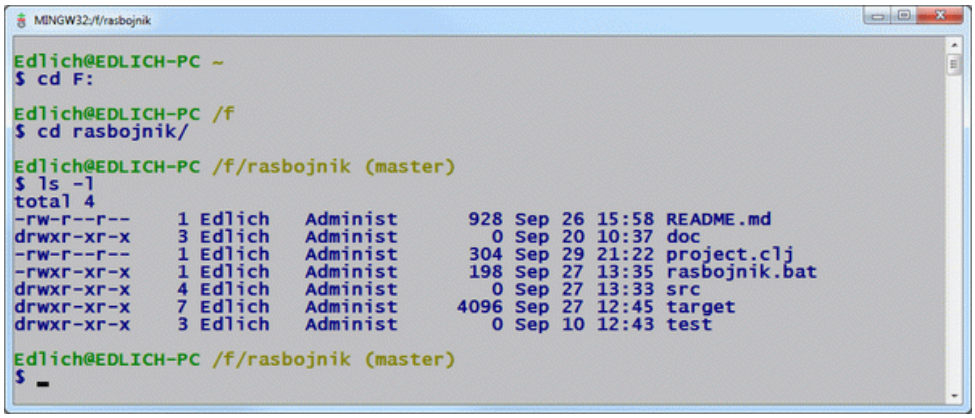
Installation

Git clients are available for native download for Mac, Windows, and Linux. In addition, there are countless other tools such as Graphical Tools (e. g. for Windows by GitHub itself), Eclipse PlugIns, etc.

This is especially easy for Unix / Ubuntu users as is should be preinstalled.

$ apt-get install git-core

The following is therefore based on the assumption that you are command line oriented working with Git. For security reasons, there is a UNIX shell named Git Bash under Windows, which makes it easy to work with. This option is always easily available for other environments / computers.



Use and output of Unix commands

As you can see in the figure, the usual UNIX commands are also available under Windows in the shell:

- with cd will change to the directory you want to manage with git
- with ls and ls -l you can view the contents again

What is now missing are the commands to work with Git. We will have a look at them in the next chapter.

In the following it is assumed that the other UNIX commands for work (e. g. with the file system like mkdir, rm, etc.) are known.

Git und GitHub einrichten

Git himself would like to have some information. With

git config --list

you can display which system properties are already set.

These can be set manually in the ~/. gitconfig file. These can also be changed in the shell as follows:

git config --global PROPERTYNAME "propertyvalue"

The Property user.name should point to "Stefan Edlich". People must be easily identifiable in the team. This is set, for example, with

git config --global user.name "Stefan Edlich"

This can be done with = or without equal sign. Without an equal sign, git then simply takes the rest according to the user. name key.

The e-mail:

user.email=edlich@gmail.com

The Github Username:

github.user=edlich

The Github Token:

github.token=bo39wzgq39tznzc4p95a8q4fe8eb

Used to be required for authentication. Please refer to the current security settings (SSH / OAuth, etc.).

Der Editor, der von Git gestartet werden soll:

core.editor=vi

Which editor should be started at merges:

merge.tool=kdiff3

How do I deal with Unix and Windows line endings?

core.autoclrf=true

6.4 Grundlegende Arbeit mit Git

Title: Basic Git Commands

As with all version control systems, there are two initial startup methods:

- Use existing own local code
- Use an existing remote repository and clone it.

Of course, both can also be combined in a sensible way. Often you create an emptyRepository on the server like GitHub, clone this, copy files into it, add them to the index, commit and then you can easily synchronize changes with a remote server.

A) Start with a localen repository With this command, you create a Git Repository:

$ git init

Excercise DVC-01: Git Repository

See which files are created in . git! What meaning could they have?

Work duration: 5 min

Now you can create, copy, and synchronize files with servers using the commands explained later. Here is an example:

$ git add README.md
$ git -m commit "First Description"
$ git remote add origin https://github.com/edlich/rasbojnik.git
$ git push -u origin master

It is important that I use remote addto create a unidirectional pointer that points to the Server = Master Repository!

B) Clone an existing or empty repo The advantage of course is that the repository already' knows' which master repository it is connected to after cloning.

Therefore, after adding and committing files, a

$ git push

Cloning is easy with the following command line:

$ git clone https://github.com/edlich/rasbojnik.git

But where do you get to know git clone? Simple: It is displayed on the GitHub page of the project. In the text window to the right of HTTP. There is also a button to download the whole repository as a zip file (Note: It sounds great, but you hardly ever need...).

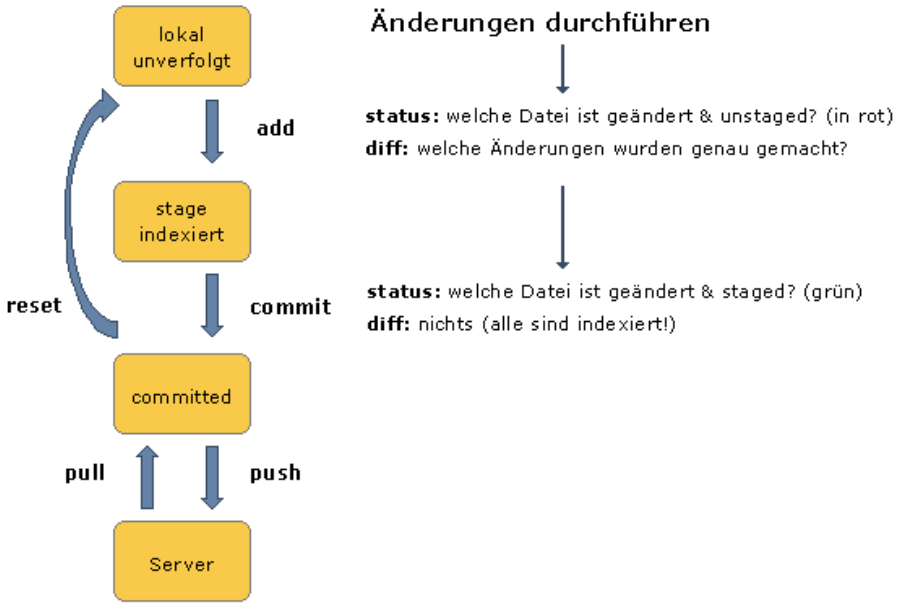
And so you can easily create any number of copies of the data and delete them if necessary. Because they are safe on GitHub.

Here, too, I implicitly used cloning to make sure that my local repository points to the remove server repository. And I can synchronize remotely with a command (and without specifying an address).

Understand how Git works

Git works slightly different to other DVCS like Bitbucket. Therefore, it is very important to understand the four levels of git that can contain data and directories.

The following graphic illustrates this:



Git Stages
Kommandos für Transitionen

There are therefore four levels of location and status:

1. **Local untraced**: Files may be fresh in the folder. They can be changed here. These files or folders have not yet made an acquaintance with Git. Git doesn't know them or chase them.
2. **stage indexed**: This step is important and a common cause of errors, since it is a multi-command (one with several meanings). At this stage Git knows the files. Git indexed it! So they are being followed. This applies to both brand new and modified files! With git add[filepattern] tell Git: take care of a file, a directory (recursive) or ask for everything with .. The current directory will be searched recursively and all files and folders that have not yet been tracked will be included (in the . gitignore file you can write down things you don't want Git to manage).
3. **' committed**: After you have made a lot of changes in the files and included them with add in the' stage' (=the index), you want to tell Git at some point:"That's it!". Update my local repository to ensure that all changes become valid and a top-level

synchronization point is set. So you call git commit (or -m "commit message" if you don't want to start an editor.

4. **push**: In the last step the locally tracked data can be synchronized to the connected server with git push and if necessary'gemerged'.

If you want to skip the stage step, you can do:

$ git commit -a -m "Stage and Committe all!"

in add all files in the Stage to the index and commit all in one step.

Of course, all this can also be done backwards with pull, reset und checkout. But the commands are described in detail later.

6.5 Git Kommandos en Detail

Title: Git Commands en Detail

All important commands are described below. Of course you always assume git in the shell.

- **create project** (with the init and clone) and help use
- **' status and diff** (display file changes and when to use which command)
- **' add and commit** (adding files)
- **push and pull** (exchange with external repositories)
- **remove and move** (remove and move files)
- **More diff commands** (effective comparison with other diff commands)
- **log** (display project history)
- **branch and merge** (work with different branches of development)

6.5.1 Projekt anlegen

Title: Create a Project

help

Hardly used, often forgotten and yet so helpful. A command like

$ git help add

usually even opens the browser and displays the original documentation page of the command (here add) as html in the browser. Of course you can also do this by handgit-scm. com. However, you are usually in the shell anyway and then it's fast.

init

Ensures that a directory is quasi' Gitified'. This means all necessary files under . git are created so that the git command can work.

In case

$ git init "source code

will provide the current directory with git commands.

In case

$ git init mydir

a directory mydir will be created, which is then ready.

clone

We create a local copy of the server repository:

$ git clone https://github.com/edlich/rasbojnik.git

Übung DVC-02: Ein Repository clonen

Try cloning this repository or even better, find an interesting project from Git! E. g. jQuery, rails, redis, linux, etc.
. Consider here what you might need public or private repositories for in the future. Everything was on your hard drive and actually well public could be heard on git! (provided it is interesting for auditing purposes. So no pictures, music, etc., but binaries are another topic for github). Duration: 20 Minuten

Optionally you can specify here:

- A directory after the repository with which an alternative name can be specified.
- Branches (branches) can still be specified

6.5.2 Status und diff

status

Before executing any commands with files, it is important to know status and diff.

git status shows which files have been modified and in which state they are (tracked or untracked).

If we add a completely new file, Git recognizes this and marks this red:

```
Edlich@EDLICH-PC /f/rasbojnik (master)
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Status I
Git recognized a new file

And thankfully we get the message that we should add git add to this file.

It looks similar when a file is modified:

```
Edlich@EDLICH-PC /f/rasbojnik (master)
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   project.clj
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Status II
Git detects a modified file

Git helps here too and says that with git add or git commit -a we have to make sure that everything gets committed or evenly committed.

If the file has been' gestaged', this is also indicated with green:

```
Edlich@EDLICH-PC /f/rasbojnik (master)
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   project.clj
#
```

Status III
Git staged the file and put it to the index

diff

git diff as the UNIX command it lists alle the changes in detail.

```
Edlich@EDLICH-PC /f/rasbojnik (master)
$ git diff
diff --git a/src/rasbojnik/utils.clj b/src/rasbojnik/utils.clj
index 2ad182c..0a8190f 100644
--- a/src/rasbojnik/utils.clj
+++ b/src/rasbojnik/utils.clj
@@ -8,5 +8,3 @@
   (when-let [fun (ns-resolve *ns* (symbol nm))]
     (apply fun args)))

-(defn doppel [x] "Doubles a number" ;; just for testing purposes. Will vanish s
-  (* x 2))
```

Diff I
What changes have been made?

If something was changed within the line, old and new would appear in red (= disappeared) and green (newly added).

It is important to note that diff always compares to the last commited version.

It will be exciting if the file is then added to the stage again with add. What does git show then?

```
Edlich@EDLICH-PC /f/rasbojnik (master)
$ git diff

Edlich@EDLICH-PC /f/rasbojnik (master)
$ _
```

Diff II
No changes visible after add!

The result is a bit unexpected! git diff shows only those changes that are not staged. If everything is done, no difference can be displayed. Even if there are differences between the last stage and commit!

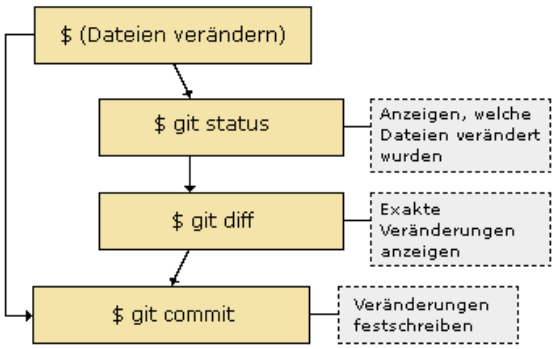
This means that only non-staged changes against last commit are displayed. Of course, you can also display changes between commits.

Übung DVC-03: Differences between commit

Find out how to display differences between commits in both the shell and GitHub.

Duration: 15 min

The following figure shows the status and diff again:



Git: change files

We will adress more compley diff's later!

6.5.3 Add und commit

Title: add and commit

add

With $ git add file (s) I put files or directory trees under the supervision of Git. I track her. This means both new files and changed files.

With this command, Git prepares everything for the next commit: the snapshot / index update.

Wichtig

> Important: No commit without a previous add! Usually even many adds. If you don't add committed and then pushed, then nothing is visible in GitHub. And they accuse Github. Github himself sometimes has problems emptying his order queues, but it's usually not his fault. Herestatus. github. com you can check Github.

The crucial thing about this command is that a file pattern can be passed as a parameter. Here are a few examples:
$ git add README.md // just add a file
$ git add *.java // add all java files
$ git add SecretFolder // add a folder: recursiv
$ git add . // Find everything I touched!

The last option is also the simplest and is often used, but can have side-effects if you have changed things unintentionally.

Ad mentioned above you can leave out git add and do all in one with:

$ git commit -a -m "Nachricht"

But it is recommended to use add because you should be explicit in order to avoid surprises!

Interesting is perhaps still the option -i:

```
$ git add -i .
           staged     unstaged path
  1:    unchanged        +4/-1 Cake-Folder/Kaesekuchen.txt
  2:    unchanged        +2/-0 Cake-Folder/Sacher.txt

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
```

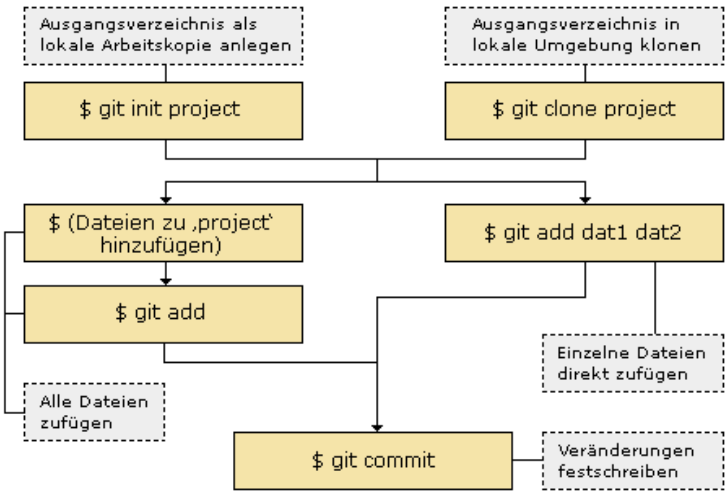
git add -i .
interactive add

Here you can edit files selectively.

Finally, the changes can also be viewed with -e in the editor.

commit

As written you insert all staged changes with commit.



Basiskommandos Git

They then take effect. The stage is emptied.

Only the following options are used as options:

- -a: I forgot to add. Search and addee all files
- -m: I want to specify the message here. Please do not start an editor for the log message.

But -u is also interesting to recognize all the files that are not yet included in Git / in the index.

And finally, Git also has the file option, which means that files or patterns for files can be specified.

Anmerkung

What if the commit was wrong? With --amend you can change the commit before.

This would be roughly equivalent to:

$ git reset -- soft *HEAD*
$ // everything one again correct
$ git commit -c *CORRECT_HEAD*

6.5.4 Push und pull

Title: push and pull

push

With push all local commits are synchronized with the remote server.

There is the following option:

git push <repository>

Normally the repository is stored in . git/config. This means that normally you only execute git push.

Wichtig

When working in a team, you should always think of git pull early on in order to be aware of changes that others in the team have already made and sent to the server with git push!

With

$ git pull --rebase <remote name> <branch name>

you can work on a remote branch and get it.

The credentials are always causing problems here. Especially because the Git protocol is constantly changing:

If you push over the https protocol, Git asks for a password every time. With this command you can cache the password (at least under UNIX).

$ git config --global credential. helper cache

Or even cache for a longer period of time:

$ git config credential. helper' cache --timeout=3600'

For Windows there is Credential helper.

This applies to all versions after 1.7.9. For earlier versions you have to encode name and password in the URL, which of course does not seem to be secure:

$ _https://you:password@github.com/you/example.git "source code

Alternatively, you can also define the credentials for the SSH protocol.

pull / fetch

This compares the changes on the server with the local branch.

Of course, you can also enter a repository address here. This command becomes interesting if branches are to be specified or merged. But more about that later.

Just like git push this command works normally quiet. The address of the repository is read from the configuration file.

pull and fetch are similar. The difference is that pull equals merged and not fetch.

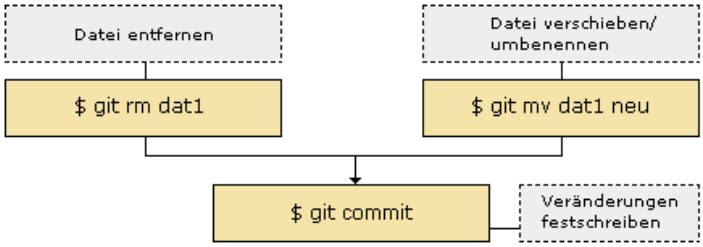
6.5.5 Remove und move

Title: remove and move

Remove files is always done in two steps: Physically remove the file (with DEL or rm file)

1. git rm useless. md

Of course, this only becomes effective after the correct commit.



Git: remove or move files

Hinweis

Often you forget to set files in . gitignore. The above procedure is then appropriate. To delete the file from the index only, you only need to delete it.
$ git rm --cached nostageplease. md

remove also accepts wildcards in the file pattern again.

Git itself still provides a command to rename files. However, this is internally mapped as rm (delete) and add (add) of the file.

Hinweis

Git makes this mv really physical. A ls -l and a git status prove that this one command is sufficient.

6.5.6 Mehr diff Kommandos

Despite all the nice commands so far, there is still a lot missing to be able to compare really effectively, switch to any previous versions and create branches. This will be addressed below.

So far we only got to know the normal diff:

$ git diff

This works wonderfully before staging. Only after staging, we can't see anything here.

What would be interesting is of course the difference between the Stage (= the index) and the last commit. This goes with it:

$ git diff --cached

Which gives us the desired output:

```
Edlich@EDLICH-PC /f/Google Drive/rasbojnik (master)
$ git diff --cached
diff --git a/src/rasbojnik/core.clj b/src/rasbojnik/core.clj
index a5f53b4..2f5396a 100644
--- a/src/rasbojnik/core.clj
+++ b/src/rasbojnik/core.clj
@@ -36,12 +36,17 @@
   "Next search from a different game. So initialize here."
   ) ;; initialize here!

-(defn position [line]
+(defn position [line] ; varargs inhibits destructuring here?!
   "Receives 'position startpos moves c2c4' or 'position fen <fenstring>'"
-  (let [nline (rest line)]
-   (if (= (first nline) "startpos")
-     (println "Apply move" (last nline))
-     (println "fen not implemented"))))
+   (if (= (second line) "startpos")
+     (println "Apply move" (last line))
+     (println "fen not implemented")))
+
+(defn go [line] ; "go depth 4"
+  "Start searching the actual position. Different args can follow."
+  (case (second line)
+    "depth" (println "I call the gamecontroller with" (rest line))
+    (println "This go option is not yet implemented!")))

 (defn ? [line]
   "The debug command by Arena."
```

Diff cached
Stage index against 'last commit'

Hinweis

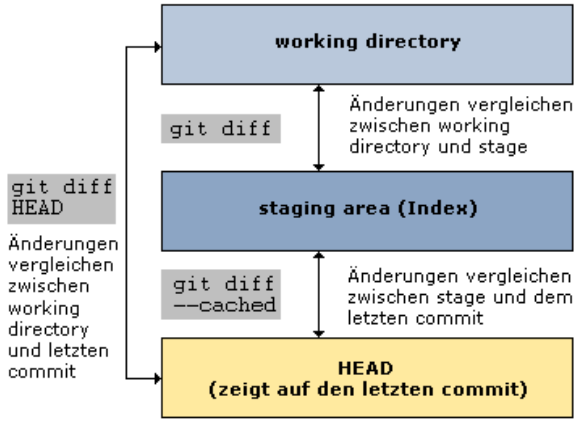
> But here we get only Staged (= indexed) against HEAD (thus the last commit!) Everything that has been edited and not staged is still missing!

So it is still the command that shows us all changes compared to the last HEAD. So staged + unstaged against HEAD:

$ git diff HEAD

In case of doubt, there is always a little bit more than with --cached. Hackers who already have -a in their blood and do this regularly can compare against HEAD.

The following graphic gives you an overview of the above-mentioned diff commands and what is compared in each case:



diff-Kommandos im Vergleich

6.5.7 Log

One of the most verbose commands is:

$ git log

Which shows the history:

```
Edlich@EDLICH-PC /f/Google Drive/rasbojnik (master)
$ git log
commit 48bbf56c0489767045286c0eb1cd3f384a8f9747
Author: Stefan Edlich <edlich@gmail.com>
Date:   Thu Oct 4 14:08:35 2012 +0200

    Inserted 'go' command

commit 16049fd7ea7836df348f04b0377cbffb12cc3ea9
Author: Stefan Edlich <edlich@gmail.com>
Date:   Wed Oct 3 19:34:13 2012 +0200

    passed the line to attack position command

commit cc0879fe78076869b3f3f2299329d02011a1091f
Author: Stefan Edlich <edlich@gmail.com>
Date:   Tue Oct 2 21:42:25 2012 +0200

    deleted a line. only split out please

commit 9bddccb6fcd40eca4820de573300201a795b4bd3
Author: Stefan Edlich <edlich@gmail.com>
Date:   Tue Oct 2 21:39:48 2012 +0200

    Reduced all to core.clj. deleted utils.clj

commit 2fadb0fb8773f3aa72e053a4c5918b588e135903
Author: Stefan Edlich <edlich@gmail.com>
Date:   Tue Oct 2 14:08:24 2012 +0200

    test
```

Log
Commit history

Die ist praktisch der gleich Output wie auf GitHub die Commit Seite. Es enthält aber praktischerweise noch mehr:

- Name des Autors
- E-Mail Adresse
- Die Commit Message
- Die SHA-1 Checksumme

Anmerkung

Mit letzteren SHA-1 Hashs kann man offensichtlich ideal ein altes commit referenzieren! Idealer wären zwar echte Revisionsnummern, aber dies ist in komplett dezentralen Systemen schwieriger.

What else is there:

$ git log -p [-2]

Returns the exact changes of all commits. This is of course far too much, so you can look back into the past. Or with -2, i. e. minus number to specify the number of entries in the past. Try this and compare the hash sums of the commits. Exit the display with q.

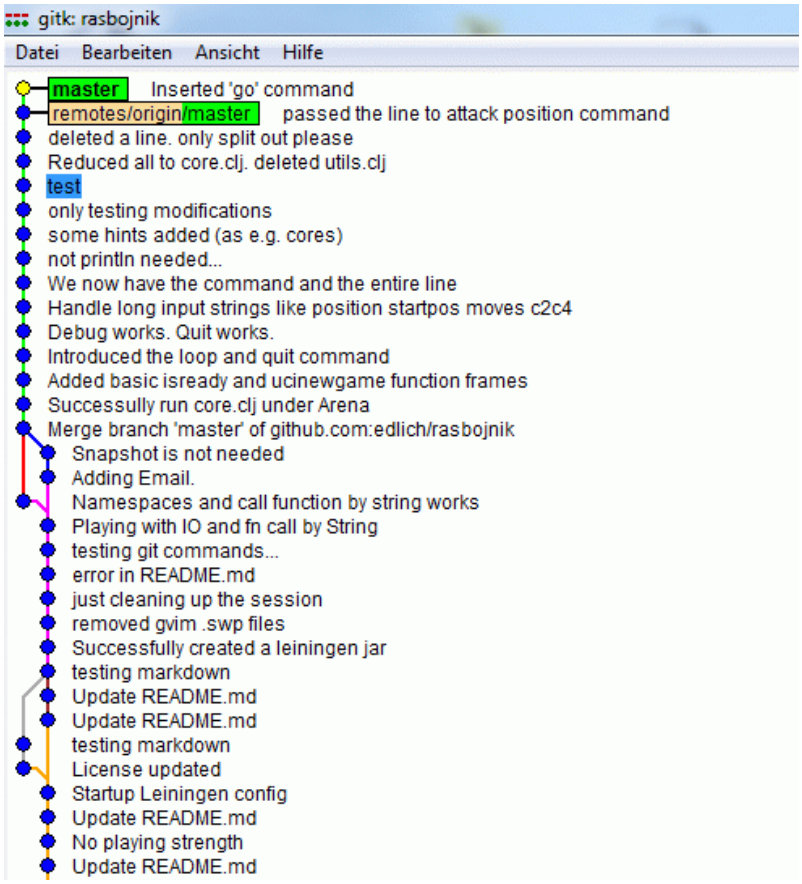
Übung DVC-04: Try Git Commands

Task: Play around with the following commands.

git log --pretty=oneline
git log --stat

Duration: 20 Minutes

But this is certainly easier with GitHub or the Tcl/Tk Vizualizer' *Gitk*, which is included in Git:



Graphical visualization with Gitk

(To do so, go to the Git Shell gitk). Here you can easily view all changes by clicking on the commit in any branch in other windows (not shown here)!

6.5.8 Branch und merge

Title: branch and merge

The basic command for working with different branches of development is $ git branch.
.
.

List of available branches: $ git branch
Create a new branch: $ git branch (branch name)
Create a new branch and switch there: $ git checkout -b (sector name)
. Delete branch: $ git branch -d (branch name)

For merging different branches, Git offers $ git merge[branch name]. In doing so, Git independently checks whether conflicts arise when merging the branches and lists them if necessary.

pull requests

Of particular interest is the merging in git with a so-called "pull request".

Here is an extensive description: The intention is: If I *forked* a project, and before I *pulled* / *cloned* it and changed it, then I can send a *request* in GitHub, to insert my*changes* into the main branch of the original project *merged* werden!

6.6 Zeitreisen in Git

Title: Time travelling in Git

We begin carefully:

IN the first example we want to **unstage** changes which have been **staged**. $ git reset $ git reset HEAD
$ git reset HEAD README.md
$ git reset HEAD -- README.md

Of course, the actual file remains unchanged.

What if you have a new gestaged' te file, but you don't want to include it in the commit yet: $ git rm --cached notYet.txt Normally rm, but in this case only from the stage cache.

(In process! scenario with checkout cannot be verified.) $ git checkout
$ git checkout -- README.md

To completely reset to the latest HEAD version you can of course also choose the' hard' method, delete the old directory and create a fresh repository with git clone:

$ git clone https://github.com/edlich/education

Time Travelling via Tagging

In software development it is normal that an application is subject to constant changes. Be it by correcting errors, adding functions or structural improvements in the form of refactoring. No program is perfect at first go, if there is anything like a perfect program at all. To keep track of changes, it makes sense to work with version numbers. If I want to jump to a certain version, I can easily do this with Git via
$ git checkout <commit-hash>
.

The problem is: How do I know which git-commit belongs to which version number? This is well done in Github, but tagging is much clearer.

One possibility is to search for a hint in the commit messages via $ git log. A prerequisite for this is, of course, that the committing party has thought of making a note to this effect. into the message. But that is not the only problem. Imagine that there is an unmanageable number of commits between two versions. How much time would it take to search them all!

Tagging is a remedy. Git offers two types of tags that differ in the content of their information. With $ git tag -a <tag-name> -m <tag-message> you can save a reference with the name of <tagname> to the last commit. In addition, the entered message, tagger name and date are saved. If you do not want to store this information, it is sufficient to use $ git tag <tagname>. Example: $ git tag -a v0.1 -m "My version 0.1".

With $ git tag you can now display a list of all available tags. More information about a tag can be found via $ git show <tag-name>.

A checkout is now possible via $ git checkout <tag-name>.

Important: Tags are not automatically transferred to a remote via $ git push. To share a local tag, you have to push it explicitly with $ git push <remote-name> <tag-name>. Alternatively, you can also use
$ git tag push <remote-name> --tags
to push all local tags.

Hint: Tags do not have to be limited to certain versions only. Every commit that is supposed to be quickly found later deserves a day.

Further information about tagging: https://git-scm.com/book/en/v2/Git-Basics-Tagging.

An alternative is the GitHub releases, which are also tagged to the last commit. Usually version numbers like 0.6-beta, 0.9,1.0 etc. are used.

6.7 Git Glossary

- **repo, repository**: The Git database location with which you work.
- **index, staging area**: As described inBasics, the changes are indexed here, i. e. prepared for the commit.
- **clone**: A copy of a repository. Created with git clone <address>.
- **commit**: A good snapshot of the current project status. All commits form a chain (which is also visible on GitHub!) and point to their associated files.
- **branch**: A branch where to work on. For example, locally on the disk.
- **dirty**: A branch is dirty if it still contains uncommitted changes.
- **a head**: The main line of the project.
- **HEAD**: Usually refers to the current branch. Refers to the header line in the repository
- **tag**: Usually the message is the -m the tag follows, i. e. the description of the commit.
- **patch**: A change that is introduced not via Git but somehow (e. g. via text).
- **changeset**: Synonym for the content of a commit.
- **object**: An object managed by Git (file, messages, etc.).
- **refspec**: An illustration of how, for example, you want to map remote to local.
- **history**: The Commit History.
- **upstream**: The term upstream refers to the original repository where the local repository is pointing. In Git this is called Origin. So English' the source'.
- **tree**: Either the current directory tree of the branch. Occasionally, all branches from branches to a project are also called trees.

- **rebase**: Einen Branch mit dem Inhalt einer anderen Base so verändern, dass dann die ursprüngliche Base auf das neue Ergebnis gesetzt werden kann. Entspricht z.B. dem setzen eines Repositories auf den Inhalt eines anderen.

6.8 Mercurial

Title: Mercurial

Referenzen

- Good Tutorial by Joe Spolsky
- Quick Start
- Mercurial Usage
- Original Quick Start
- Pics of all Commands

Bookds:

- Mercurial: The Definitive Guide by Bryan O'Sullivan free!

Vertiefung

Tipp:
For more information on the Mercurial concept, seeMercurial Basic Concept.

Sidemarks

Parallel to the git initiated by Linus Torvalds, of course, more DVCS software has been created.

Amongst many others, special mention should be made:

- Mercurial
- Bazaar
- Monotone

Particular attention is paid to the Mercurial protocol with the corresponding software.

Reasons for this are:

- Many find Mercurial and the commands easier. Especially the numbering of the changelogs in the log.

- With Bitbucket. org there is a portal (purchased from Atlassian) which also allows free repositories.
- Unlike GitHub, private portals are free there!
- The commands work directly in the shell (without additional UNIX shell)
- Atlassian invests heavily in Bitbucket. For example, bitbucket also speaks the Git protocol on request!
- Many large companies also rely on Bitbucket such as: salesforce, Paypal, DHL, SecondLife, New York Times, etc.
- A built-in server that can be started immediately on port 8000.

Another big difference is that Mercurial works with "Changesets" and Git works with snapshots.

Comparing Github with Bitbucket is of course difficult, because they change very quickly.

The author of the module once wrote himself down:

Github: + better online editor clearer front screen No private repositories

Bitbucket:

- + unlimited private and public
- + syntax and logs often easier

But this is certainly outdated with time.

Hinweis

Make your own judgement as to which system suits you best. The best thing, of course, is to work with both systems and then always use the right one for the project. It's a nuisance to learn two systems, but there's a lot to learn!

Wichtig

By the way, everything that is public should be hosted from your hard drive on a DVCS provider. And everything that is private, but versioned on your hard drive (such as your elaborations or master works (!) must also be stored on a DVCS repository like GitHub or Bitbucket!

Commands

The following is a summary of the commands in English:

- username = Stefan Edlich <edlich@gmail.com> // in ~edlich/.mercurial.ini
- hg => common commands (help = all cmds)
- hg init => creates a Repo
- hg add => adds file to Repo; no arg = all; // -I *pattern* (=include); -X *pattern (=exclude);// -S (recursive subs); -n (no action)
- hg commit -m message => adds all or given files or finds changes! // -u edlich@gmail.com // -I ; -X; -S like above
- hg log => what happened?!
- hg revert => back to last committ // -- all;

  -I

  ;

  -X

  ; -r // revision
- hg status => what has been edited?
- hg diff file => shows the diff // hg diff -r 0:1 B.txt
- hg remove => plans for remove; actually remove when committed!
- hg cat -r 0 B.txt => show the older revision!
- hg update => hg update -r 0 (back to start); hg update -r 1 (Revision 1); hg update(back to last!)
- hg rename a b=> rename
- hg tag => V 0.0.7 (PSEUDOLEGAL) // simply mark it
- hg serve => starts a server on default port 8000; http://joel.example.com:8000/ shows content!
- hg clone https://edlich@bitbucket.org/edlich/edlclj
- hg push => pushes to last cloned address --update or -u // in .hgrc http://user:password@bitbucket.org/edlich/mytest
- hg outgoing => show local chances which would be send to to the remote connected repo! ->
- hg incoming => show remote changes that have to be merged first! <-
- hg pull => get all remote changes to my local // use parent & update then!! // "added 1 changesets with 1 changes to 1 files (+1 heads)" indicates you have to merge
- hg parent => shows if my log head is the latest or if I need to update
- hg update => incorporate all pull changes from the parent changeset (or to the given number)

6.9 Kommandoübersicht Git

Title: Overview of Git commands.

Here all commands for the dayli work:

- **git init** => Create a fresh Repo
- **git clone** => Copy a Repo (check it!)
- **git pull** =>THis is how you should start a day! (ggf. merge needed!)
- *think & hack & think*
- **git add** [.|file|pattern] => only stage it
- **git status** => list all changes
- **git log --pretty=oneline** => history
- **git diff** => what happened in detail? --cached or better immediately --HEAD
- **gitk** => tool for a visual history
- **git commit** [.|file|fpattern] => -a alle Änderungen committen.
- **git push** like an upload to the server / Github
- **git rm/mv file** => Remove from the index! Check if a real remove is needed afterwards. After this use:

git commit -m "removed file CNAME because wrong ;-)"

6.10 Zusammenfassung DVC

Title: Summary DVC

In this unit we have dealt with distributed version control systems:

Git and Github are currently market leaders and significantly more powerful than their predecessor systems such as Subversion

It is crucial that courageous programming is encouraged. Creating new branches, merging or returning to old versions is easy.

The repositories can be compiled to any graph or tree hierarchy and thus contribute to the final product.

- In Git there's a "stage" that acts as a kind of "intermediate ramp". Here everything is collected before the commit.

7 CID - Continous Integration / Continous Delivery

Autor: Prof. Dr. S. Edlich

7 CID - Continous Integration / Continous Delivery
7.1 Überblick und Lernziele CID
7.2 Einführung CID
7.3 Deployment Pipeline und Continous Integration
7.4 Konzept des Continous Integration
7.5 Praktische Übung: Jenkins einrichten
7.6 Travis-CI
7.7 Zusammenfassung CID
7.8 Wissensüberprüfung CID
7.9 Literatur CID

7.1 Überblick und Lernziele CID

Title: Overview and Learning Goals CCD

Lernziele

Learning Goals

In addition to the specialized knowledge about processes and process chains in Continous Integration / Continous Delivery (CI / CD), prerequisites, boundary conditions and problem areas are particularly important. On the practical side, all skills must be acquired to technically implement a CI / CD process.

Gliederung

Gliederung
Accordingly, the learning unit is divided into two parts: A descriptive part in which the process is presented with all its facets and is also critically examined. The second part contains a starting aid for the practical handling of CI / CD software.
This is not a trivial exercise as an individual student, because apart from the installation of many technology components no real company can be modeled. Nevertheless, we will try to illustrate some more complex processes and execute several targets as well as several tests.

Zeitumfang

Time Requirements
The time required to read this session is 45 minutes.

For the practical part, including all surrounding tasks (DVCS, build management command files, tests, code, etc.), approx. 3 hours have to be scheduled.

7.2 Einführung CID

Title: Introduction CID

As a student, you usually only get knowledge and practical experience of how to develop small software. Most of the time there are a few dozen classes and a library - with the result of an executable program.

Later, however, things will be quite different. Then the complexity of software companies in large companies is much higher. It is then about a multitude of people, a variety of software systems, a lot of libraries as well as probably several servers and many executable programs.

So it's no wonder that nowadays you can see companies that spend a lot of their time and money on managing the software system itself and especially its configurations. For example, there are usually many people who have specialist knowledge of a particular

part of the system. Exactly these persons must then contribute to the construction of the system as a whole. Libraries come in a variety of versions, the individual sources are distributed across different version control systems, and the software creation process is not automated.

In a company, the all-important question is often:

„How long does it take to integrate a single new line of code into the system in such a way that the end product can be immediately rebuilt and demonstrated again?."

This is referred to as cycle time.

Definition

Cycle Time
The time it takes to get a fully operational product after a change.

It turns out that it is important to know how fast a software system can be built nowadays - with the ever-increasing number of software systems. The aim is to reduce this time so that you get within the range of hours and minutes. The measures taken to achieve short cycles play a decisive role in mastering the complexity of large software systems.

Anmerkung

Good to know!
Studies have shown that software maintenance accounts for more than 50% of the total IT budget. Many companies without a small build cycle fail much earlier in this maintenance window and spend more money than would be necessary.

The fascination of CI / CD is therefore particularly in the way a conductor can synchronize an entire company (or its code fragments) at the touch of a button and create a living application. Just like software tests where green bars point to positively tested classes, modern CI software tools also have graphically appealing dashboards and timelines for feedback. They also calculate the average stability of the builds and make a quality measure transparent. It makes success - also of the CI / CD - mostly visible. In the last part, you will find corresponding illustrations.

7.3 Deployment Pipeline und Continous Integration

Title: Deployment Pipeline and Continous Integration

Usually, creating software consists of the following steps:

- cleaning up
- check out
- compilation
- testing (small)
- merging
- deployment
- integration tests

In reality, the points listed are even more extensive and complex. This is known as a deployment pipeline. Ideally, it should only require the use of a single button to initiate this whole process.

Each of these phases is well described. However, the interplay and best practices on all these points are hardly documented.

Over the past 10 years, the concept of Continous Integration has evolved into a new term. etabliert.

Definition

Continous Integration
Continuous integration is a method of developing software. Every developer often contributes his or her work - if possible on a daily basis - which leads to the integration of several times a day. These integration builds are checked by the automated build (for example, through tests) to detect errors as early as possible. Many teams have reported that this approach significantly reduces integration problems.

Continous integration describes a subset of the Continous Delivery discussed here. At the same time, a whole set of tools has been established to automate this integration and the delivery of the software product by means of a "click".

Die bekanntesten Werkzeuge sind:

- Cruise Control – the oldest tool
- Jenkins / (formerly) Hudson – the most famous and most deployed tool
- Bamboo – advanced tool by Atlassian
- Apache Continuum
- CodeBeamer
- TeamCity

- Go - by ThoughtWorks
- Microsoft Team Foundation Server

and many more as a service (see later).

In the last part of this session and in the practical task we will take a closer look at one of these tools.

Requirements for CI / CD tools are very well summarized in this (not quite up to date) matrix.

Hinweis

Tipp
Evaluate tools for yourself! Maybe you like other tools than Jenkins better. Maybe there are trial versions of Bamboo.

7.3.1 Typische Fehler im Deployment-Prozess

Title: Typical Errors in the Deployment Process

Typische Fehler im ganzen Deployment-Prozess sind:

- It is manually deployed. Deployment knowledge is widely spread across many people.
- Documentation on the deployment process is growing rapidly.
- Strongly increased communication effort for a release. Many questions to developers and teams to readjust any screws.
- Multiple deployment environments that differ in configuration.
- Creating the release takes far too long. The tests are carried out at the customer's premises.

An automated Continous Delivery helps here in many ways.

Definition

Martin Fowler: *"Continuous Delivery is about keeping your application in a state where it is always able to deploy into production."*

The process is simple and inexpensive. Most importantly, the process is clearly documented graphically in scripts and through the tool. This means that it is easy for everyone to understand. In addition, all teams are forced to work well together and communicate to ensure that scripts are working properly in this process.
Humble et. al
describe this process as:

„…low-risk, frequent, cheap, rapid, and predictable…"

Humble et. al

Übung CID-01: Further Disadvantages?

Are there any other disadvantages that are known from your company in dealing with the creation of a software product?
Task Time: 10 Minuten

7.3.2 Feedback von Entwicklern

Title: Developer Feedback

In environments where there is a manual build process, it is often underestimated that changes to a single small part of the software system can have an impact on many parts of the overall system. Furthermore, it is impossible to predict which part of the software system will be affected by changes and which persons need to be informed.

A central aspect of continuous delivery is therefore that any change to the code base or configuration of the system forces feedback from all possible stakeholders, developers and departments.

If a change is made during manual deployment and the entire system is not tested immediately, errors are usually detected much later. Maybe even days or weeks later, perhaps even during deployment at the customer's site. This means that errors, feedback and solutions are often so far apart from each other that an assignment is no longer possible. The effort to correct the error is higher by orders of magnitude than if an immediate feedback would occur. If errors are only found after days or weeks, the person who caused them must first find their way back to the concept and possibly even touch down on the environment of the time. This is time-consuming and creates stress.

Feedback too late

If an error is found, it must be distributed in a kind of broadcasting procedure so that all parties involved are informed. Many companies have monitors or traffic lights that show the status and other background information. Based on past experience, this has had an extremely positive impact on the company's productivity in IT projects. There is also a shared responsibility for the deployment process.

Feedback-Status

With such a procedure, the developers know that the system is tested immediately and are more careful with check-in. So-called Broken Windows are less common in code.

7.3.3 Saubere Konfigurationsverwaltung

Title: Advanced Configuration Management

Configuration management usually only means version control of the current source code in the current programming language.

In fact, for a continuous delivery build, all artifacts must be stored in the current version control. Only this allows central access for the build. In addition, this measure enables you to track all changes and access past statuses. For example, configuration files and environment settings are most often forgotten. It is also important to consider whether it can be helpful to save binaries in version control. In some companies, it is common

practice to store the complete configuration of the virtual machine and make it available to all developers. Making this available in version management is certainly an extreme case.

What should be avoided in any case, however, is to store the result of transformation processes - including binaries - in version control. Anything that can be generated dynamically should be created dynamically.

A clean configuration management, therefore, requires the following:

- Management of all software configuration files (typically Java properties, YAML / XML files, etc.).
- Management of all environment parameters.
- Understandable commit messages. Especially if - as explained on the previous page - the originator of a change inadvertently generates an error that has to be processed efficiently and separately from another developer, a high-quality description of the change is indispensable.
- An administration of the required libraries, in all their versions.

There are also a variety of tools for application or environment configuration, such as Nagios, Open NMS or Puppet.

7.4 Konzept des Continous Integration

Title: Continous Integration Concepts

According to observations by Humble & Farley, software systems are most of the time in a state in which an executable cannot be produced. The reason for this seems obvious: nobody is interested in testing the application until a release or sub-release has been completed.

It becomes clear that this is a fatal error when one considers that during this time a large number of changes are made which can cause a multitude of errors. If only the release is tested late, then there is not only a long time lag, but the confusion of errors can lead to unpleasant dependencies. This then becomes exponentially harder to unknot over time.

Hinweis

Historie
In his book *Extreme Prorgamming Explained* (1999),
Kent Beck

pointed out the importance of continuous integration for the first time in one of the XP guidelines. See the corresponding learning unit "Process Models" from Softwaretechnik Bachelor.

Essential prerequisites for Continuous Integration are:

1. The use of a (powerful) version control system with regular and up-to-date check-ins to prevent errors from accumulating. This also means being prepared to jump back to older versions.
2. An automated build process with extensive testing. For example, Selenium tests for web applications.
3. The entire CI process should not take more than 5-10 minutes. This depends on the number of stages (see explanation below).
4. The agreement of the entire team/company that such an integration process is important.

The third point is perhaps even more important than the specific tool.

Nevertheless, today's tools for continuous integration are easy to install and very powerful.

Many other best practices are known from works such as "The Pragmatic Programmer" by Dave Thomson (ref: Andy Hunt, David Thomas, 1999, Addison-Wesley):

- Do not leave broken windows: Do not check in incorrect code. If the bugs cannot be repaired within a few minutes, you should return to the old version.
- First run tests on the local repository (first stage) and then commit.
- And finally, of course, the legendary Test Driven Development.

Definition

Stages
Stages are stages on which the growing application is tested.
For example: Developer = stage 1, between the core teams = 2, in the whole company = 3

7.4.1 Testen im Continous Integration

Title: Continous Integration and Testing

Let's look at the different test levels:

1. Unit Tests

2. Integration tests
3. Deployment tests
4. Acceptance tests
5. Other stress tests: performance, load, security, etc.
6. All metrics: Test Coverage, Code Reviewer, Technical Debt, etc.

Let's assume that there are all the tests mentioned here for the project - not only the unit tests mentioned in the first test level. Now it is to be decided how many stages there are.

Usually, there are at least two stages in a company (and we stick to it for simplicity's sake):

1. Unit tests by the developer
2. All other tests on the next higher stage

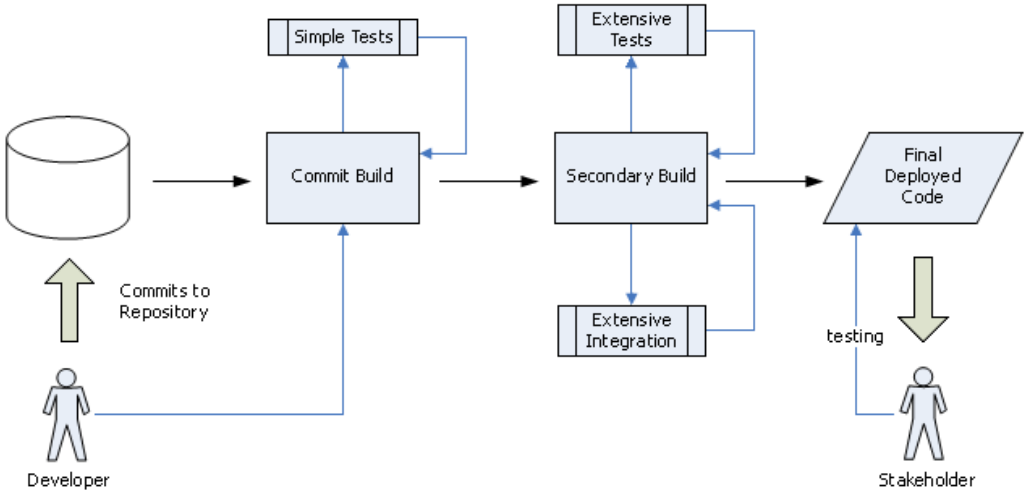
To the level / Stage 1: In the meantime there are even tools that allow checking in the code only if all unit tests are green and running successfully. Even without this feature, modern CI tools even recognize that the repository has been re-published and can then start the CI process automatically. But even without these tools it should be self-evident to clone against the remote server only with complete green.

To level / Stage 2: Here the CI server works and runs all tests (level 1+2).

The 6th test level lists processes that are only indirect tests:

How high is the test coverage? Which one can I live with (and why?) Code analyzers are also tests, but for code quality. And in theTechnical Debt (Technical guilt) all other metrics such as cohesion or dependency analysis are subsumed.

Especially these last areas are often much more interesting for management than the question of whether the system can be built or not. In particular, much too little importance is attached to the area of technical debts - i. e. areas in the code that need improvement, expressed in $, € or man-weeks - although the tools are simple and freely available.



Stagebuild

Excercise CID-02: DVCS Systems

Why are DVCS systems (Distributed Version Control System) better suited for multiple stages than central version control systems?

Excercise time: 5 min

7.4.2 Prozesskette des Continous Integration

Title: Process Chain of Continous Integration

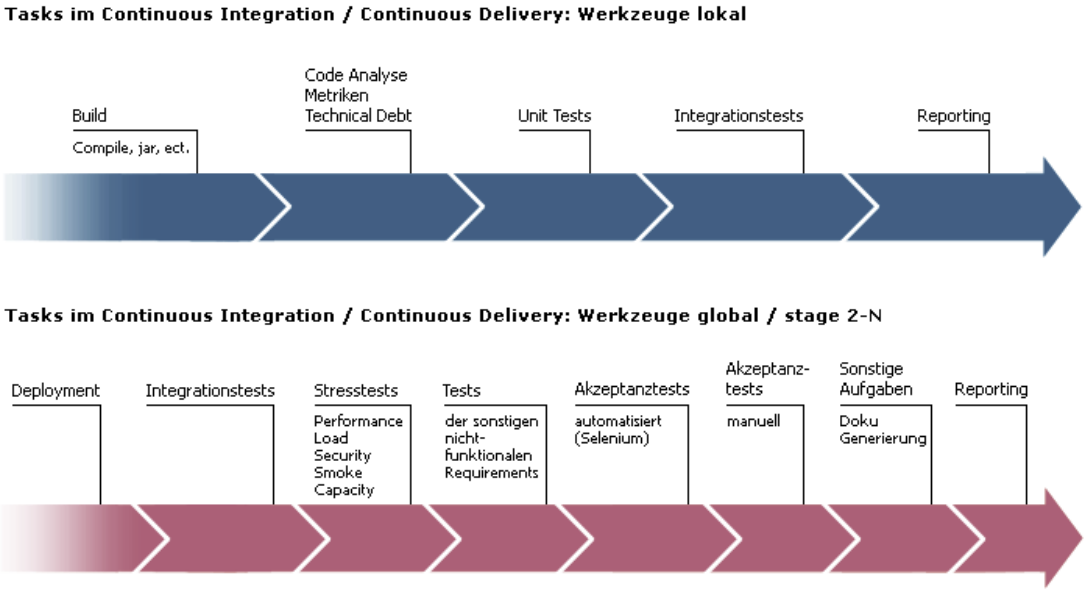
The aim of the entire process chain is to make the CI / CD cycle as simple as possible and to save time and money through this simplicity - by means of *fast feedback*. It has already been mentioned that this is important not only for the developers, but also for the management. The cycle time in which the software is tested alone is an interesting measure of the *production agility* of the software. But also, of course, when the last build was created and what quality and quantity changes occurred.

The process chain of the CI is referred to as the "Deployment Pipeline" in English and in the following. These are all the steps / processes that can or should be done in the exercise with the CI / CD tool (such as Jenkins).

For a process chain, all levels of the test module (plus additional tasks) must now be checked and, if necessary, executed:

Task in CI / CD local tools
• **Build (Compile, jar, etc.)** • **Code Analyse, Metrics, Techical Debt** • Unit Tests • Integration tests • **Reporting**
Task inCI / CD global tools / stage 2-N
• Deployment • Integration tests • **Stresstests: Performance-, Load-, Security, Smoke-, Capacity-, ...** • **Tests pf the other non-functional requirements** • **Automatic acceptence tests (Selenium)** • **Manual acceptence tests** • **Other Tasks (generate doku)** • **Reporting**

Procee Chain of the CI



Process Chain of the CI

For the second or final part of the global stage builds (behind the developer), another central repository is an advantage. However, this can consist of the cloned sub-repositories of the sub-areas.

For the actual developers it is then important that

you can easily see and verify the results, the results, libraries and executables are easily accessible,

not only for developers, but also for general managers or release managers.

The following helpful best practices are important:

- Binaries should only be created once per stage.
- Libraries and binaries should be as independent of the environment as possible.
- The final deployment should go to the same environment as the production environment. A copy of the system with the current version / release is always available. This is important because you can see what the production environment is missing, for example.
- In the build process, make sure that the process does not change any system states after each cycle (idempotent operations)

Only use relative paths in the build process

- Do not install any manual steps. This prevents 1-click deployment. For example, the ant-task "input".
- At each point of the test and CI process, in case of failure (which leads to aborting), helpful messages should be issued that reduce the identification time.
- All results / reports of the process should be stored persistently and also the history should be easily accessible. This can be the version control system itself, a server, blackboard, blog, RSS or other artifact repository.
- Avoid asynchronous tests. Many long asynchronous tests can tear up the 1-Click CI process.

Furthermore, there are many tools that formalize tests and acceptance tests and support them in such a way that they can be described and executed.

The formalization is usually as follows:

Expression	**Meaning**
GIVEN	pre-Condition
WHEN	Events / State

THEN	Result Events / State

Formalization

Therefore, different DSLs are mostly used here to implement the requirements as own Domain Specific Language. Typical systems are:

1. Cucumber
2. JBehave
3. concordion
4. FitNesse, Twist
5. selenium
6. maveryx
7. RSpec, RCov
8. Shoulda, Factory Girl

Many of these are from the Ruby / Rails community.

7.5 Praktische Übung: Jenkins einrichten

Title: Practical Ecxercise - Set Up Jenkins

Jenkins is one of the best-known continuous integration systems, which is also one of the market leaders in the industry. It was developed by Kohsuke Kawaguchi and originally supported by Sun Microsystems under the name Hudson. As a result of Oracle's acquisition of Sun, there were differences that led to a split-up of the systems. The author (and most of the community) decided to change the name because Oracle had acquired the naming rights for Hudson with the acquisition.

However, the two systems had not yet developed far apart in 2011, so that Hudson's literature published before 2012 is almost entirely relevant to Jenkins. The system can be found at https://jenkins.io/index.html Recommended literature:

Simon West

"Continous Integration mit Hudson"

dPunkt Verlag, 2010, ISBN-13: 978-3898646901

Schritt 1: Pre-Conditions

Convince all employees and stakeholders that CI brings added value!

Provide all employees with quick access and information (traffic light + reports) to the integration result and artifacts.

Schritt 2: Central Repository

Set up a central repository (Git, Mercurial, etc.) where all artifacts are stored. Provide the address for access.

Schritt 3: Installation of Jenkins

The installation / start of Jenkins is done trivially by the following call:

java -jar jenkins. war

Alternatively, the war - which was downloaded directly from the Jenkins website - can also be deployed in a servlet container or application server. After the start, Jenkins usually reports in:

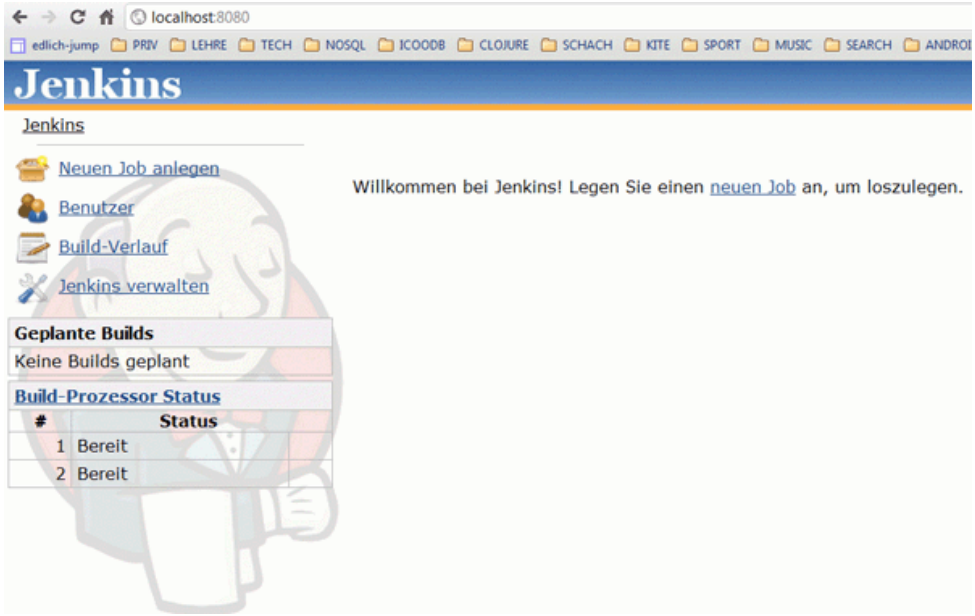
INFO: Jenkins is fully up and running

Schritt 4: Set Up and Configuration

The
war
file from Jenkins contains a servlet container that can now answer the user's web requests.

You can reach the console under
. http://myServer:8080

For local work under:
http://localhost:8080



Jenkins

By default, Jenkins writes to . /jenkins. However, this can be configured with JENKINS_HOME.

7.5.1 Praktische Übung: Projekt erstellen

Title: Practical Excercise - create a Project

After setting up a new (free-style) project with New Jobs, things like version control system and type of build management system can be set up.

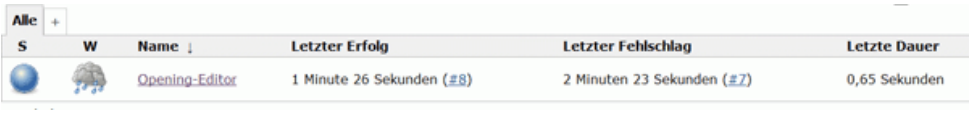
For example, if Ant is used, it has to be installed on the computer and the path Jenkins has to be specified (
Jenkins
→ manageJenkins&rarr>→
"courier" >ConfigureJenkins
→ Ant).

Jenkins must also know the path to the build script. This can be stored here under Windows, for example.

C:\Users\{yourname}\.jenkins\jobs\Opening-Editor\workspace

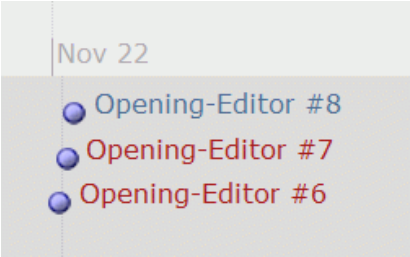
Now all steps of the build, tests, reporting, metrics, etc. can be created and saved in a project.

After some unsuccessful attempts this result should be visible:



Result

And a timeline of all builds:



Timeline of all Builds

There are of course countless other plug-ins for Jenkins (e. g. Git) and a variety of configuration options such as:

Time control, build after SCM commit

Start or result via e-mail

All this can be found on the website and in the Jenkins Wiki:

- http://jenkins-ci.org
- http://wiki.jenkins-ci.org

7.5.2 Den CI Prozess stabilisieren

Title: Stabilizing the CI Process

After all build steps have been completed, the process needs to be stabilized as follows:

Establishing the CI stages

Determining the perfect trigger or time period for builds

Archiving of all results (statistics, reports, etc.)

Collect feedback from all project owners about the process and improve the process

Usually, developers are quickly identified in the company who have responsibility for parts of the code, but who have often been torn out of the project or have to work on other projects at short notice. For this reason, the errors of the CI report can often not be processed within a reasonable time.

The problem of the CI / CD process is therefore mostly social / interpersonal rather than technical. Here we have to pay attention to how the CI / CD results can help these *blockers* to improve their work. There are many possibilities:

Putting disturbing work behind for other projects

Remove features

Getting rid of the work for annoying projects

Provide additional resources The effort for these healing measures is usually always much less than the result of a fast build - therefore the effort for a CI-free troubleshooting is always much greater.

7.6 Travis-CI

Travis-CI is a service from Berlin, which offers continuous integration as a service. Due to its proximity to GitHub and the establishment of a professional company (start-up), Travis-Ci has quickly become successful. It was born from the desire of the Ruby Community (or the Travis-Ci founders) to offer tests as a service!

Why should I do everything locally on the computer, if the code is on GitHub anyway? A service therefore sounds like a good idea.

It reads YAML files and then executes the tests. The YAML file specifies the environment, actions and then calls the test. It supports many languages such as: C, C++, Clojure, Erlang, Go, Groovy, Haskell, Java, JavaScript, Perl, PHP, Python, Ruby and Scala.

Also interesting are the hooks, i. e. actions / triggers, which can be called. For example, Travis-Ci can be started in GitHub whenever something is committed.

Example file for Clojure:

```
# travis-ci build script for clojure, leiningen and ArangoDB
# Warning: GitHub hook not yet set. Untested!
language: clojure
lein: lein2
script: lein2 test
jdk:
```

```
- oraclejdk7
# before_script:
# - "setup_arangodb_1.4.sh"
```

Sample files for Java and other languages are in the documentation, or can be easily found on the Internet.

Other Systems

The market for systems in this direction is currently exploding. There are also other systems in the cloud, such as:

- Amazon AWS Codepipeline
- Codeship https://codeship.com
- CircleCi https://circleci.com

The entire area of Pipelies as a Code is of great importance here. Thoughtworks writes in his radar Vol 16 from the 4th quarter of 2017:

"Teams are pushing for automation across their environments, including their development infrastructure. Pipelines as code is defining the deployment pipeline through code instead of configuring a running CI/CD tool. LambdaCD, Drone, GoCD and Concourse are examples that allow usage of this technique. Also, configuration automation tools for CI/CD systems like GoMatic can be used to treat the deployment pipeline as code—versioned and tested."

The mentioned tools are here:

- http://www.lambda.cd
- https://www.gocd.org bzw. https://github.com/gocd-contrib/gomatic
- http://concourse.ci

7.7 Zusammenfassung CID

Title: Summary CID

- You now know the advantages of Continous Integration / Continous Delivery (fast feedback, early detection of errors, minimizing time to eliminate them, less debugging and constant availability of a current deployed system in the production environment).
- And you know the requirements for CI / CD:
  - Version control system
  - Good automated tests

  - Small CI / DI cycles after each check-in
  - Availability of a CI / DI system
  - Availability of results
  - Code and system ownership of all parties involved
- The Continous Integration process chain was introduced.

In a practical exercise, you will learn how to use the best-known Continous Integration System Jenkins and how to create your own project.

- Furthermore, it was explained how to stabilize Continous Integration.

More Links:

- http://www.eworks.de/research/2010/02/CI-Hudson/
- http://jayflowers.com/joomla/index.php?option=com_content&task=view&id=26

7.8 Wissensüberprüfung CID

Übung CID-03: CI und CD

What does CI and CD stand for?

Lösung zeigen
Continous Integration und Continous Delivery

Excercise CID-04: Tools for Continous Integration

What tools are available for continuous integration?

Show solutions
Cruise Control, Jenkins, Hudson, Bamboo, etc. (siehe Kapitel 9.2 - Einführung)

Excercise CID-05: Tasks in the CI / CD Prozess

Nennen Sie typische Aufgaben / Targets, die im CI / CD Prozess durchgeführt werden.

Lösung zeigen

Build, unit tests, integration tests, stress tests, metrics collection, end product building, reporting, etc.

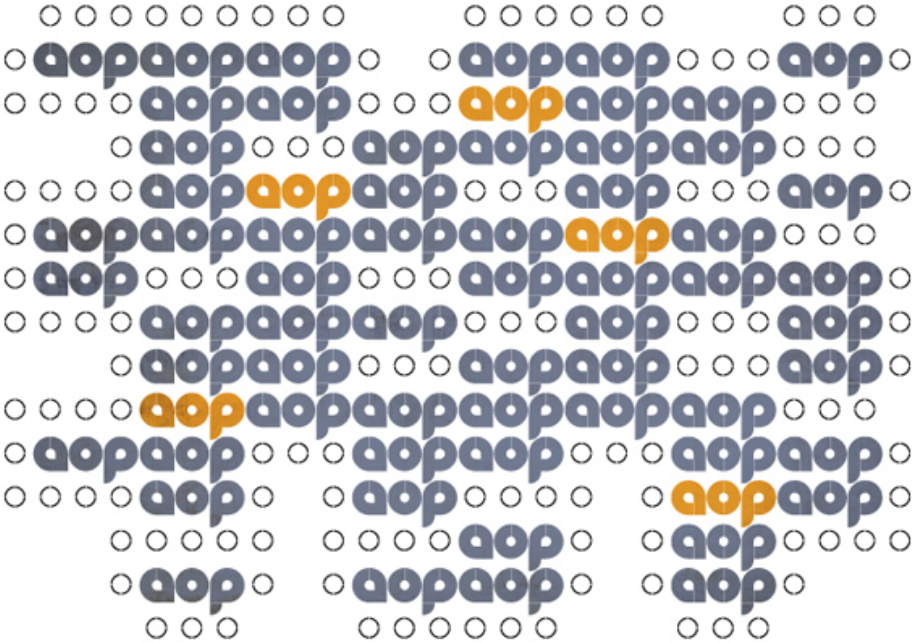
7.9 Literatur CID

- Jez Humble, David Farley, **"Continous Delivery"**, 2011, Addison-Wesley
- Michael T. Nygard
  , **"Release It!"**, 2007, Pragmatic Bookshelf
- Jared Richardson, William A. Gwaltney
  , **"Ship It!"** , 2005, Pragmatic Bookshelf
- Paul M. Duvall
  , **"Continous Integration"**, 2007, Addison-Wesley
- Ben Rady, Rod Coffin
  , **"Continous Testing"** , 2011, Pragmatic Bookshelf

Links

- **Travis-ci.org** (Continous Integration for Open-Source Projects !)
- John Ferguson Smart, ***"Real-world strategies for for Continous Delivery with Maven and Jenkins"***,

8 AOP - Aspect Oriented Programming



Autor: Prof. Dr. Stefan Edlich

8 AOP - Aspect Oriented Programming
8.1 Überblick und Lernziele AOP
8.2 Das Problem der Komplexität von Methoden
8.3 Praktische Übung mit AspectJ
8.4 Praktische Übung mit AspectWerkz
8.5 Begriffe der AOP
8.6 Besonderheiten der AOP
8.7 Zusammenfassung AOP
8.8 Wissensüberprüfung AOP
8.9 Literatur zu AOP

8.1 Überblick und Lernziele AOP

Lernziele

Lernziele Aspect Oriented Programming (AOP) ist eines der zentralen Themen für die Organisation von Code und beispielsweise auch die Architektur von Container-Services.

In dieser Lerneinheit geht es weniger darum, die gesamte Stoffmenge zu vermitteln, als darum:

- ein Hands-On zu erreichen
- die zentralen Grundbegriffe zu vermitteln
- den Einsatz abschätzen zu können

Entsprechend ist die Vorgehensweise. Nach einer Motivation erfolgt der Einstieg in ein einfaches Beispiel aus dem sich der Student alle anderen Anwendungsfälle ableiten kann. Erst danach wird eine kleine Vertiefung vorgenommen.

Gliederung

Gliederung

- Einführung
- Praktische Übungen (mit AspectJ und AspectWerkz)
- Vorstellung wichtiger Begriffe und Besonderheiten des AOP
- Wissensüberprüfung

Zeitumfang

Zeitumfang
Zum Durcharbeiten der Lerneinheit benötigen Sie ca. 30 Minuten und für die Bearbeitung der praktischen Übungen ca. 45 Minuten.

8.2 Das Problem der Komplexität von Methoden

Bei größer werdenden Projekten macht man in der Industrie - mit und ohne Kenntnis von AOP - irgendwann eine interessante Erfahrung, was die Größe und Komplexität von Methoden betrifft.

Schauen wir uns diese beispielhafte Methode an:

```
meineMethode {
    Zielcode (z.B. fill shopping Card)
} endeMethode
```

Einfache Methode

Sie zeigt, was der Entwickler beabsichtigt hat. Die Methode soll nur die eigentliche Aufgabe erfüllen für die sie gedacht ist und wofür der Name der Methode gewählt wurde.

Wenn nun in Projekten weitere funktionale und besonders nicht- funktionale Anforderungen auftauchen, müssen diese - wenn keine Containerarchitektur verwendet wird - an vielen Stellen im Code berücksichtigt werden.

So können in Methoden viele weitere Anforderungen auftauchen und den Code so aussehen lassen:

```
meineMethode {
    Parameter Check
    Logging
    Security
    Pool Management
    Zielcode (z.B. fill shopping Card)
    Pool Management
    Return check
} endeMethode
```

Verschmutzte Methode

Was hier passiert ist, ist typisch für Architekturen die keine Container verwenden oder keine aspektorientierte Programmierung einsetzen. Eigentlich sind das gar keine Architekturen. Viele kleine und unangenehme Aspekte werden so dem Code hinzugefügt und von diesen Aspekten kann es Tausende geben.

Definition

Aspekt
Unter einem Aspekt versteht man ein Codestück bzw. dessen Aufgabe, die es zu erledigen gilt. Dieser wird auch "Concern" genannt.

Concerns werden unterschieden zwischen:

- **Code-Level-Concerns**: Kernfunktionalität, die meistens funktionale Anforderungen betrifft.
- **System-Level-Concerns**: Sonstige - meist nichtfunktionale - Funktionalität.

Worin liegt nun das Problem?

Nun, durch obige Grafik ist dies offensichtlich: Der eigentliche Code wird umgangssprachlich gesprochen "verschmutzt". Dies hat schwerwiegende Auswirkungen:

- Der Code wird schlecht **lesbar**
- Der Code wird schwieriger **wartbar**
- Die **Wiederverwendung** (reuse) von Code wird extrem erschwert
- Die **Produktivität** sinkt
- Der Code kann schwerer "**evolutionieren**"

Ohne ausführliche und unmissverständliche Kommentare wird es für externe Personen vielfach schwerer, diesen Code zu verstehen und zu ändern.

Das beschriebene Problem ist in der Softwaretechnik seit Langem bekannt. Bisher gibt es dafür zwei Lösungsansätze.

- Container Architekturen
- AOP = Aspektorientierte Programmierung

Es gibt zwar noch weitere Sonderlösungen wie "Mix-Ins" oder bestimmte "Design Patterns", diese werden hier aber nicht betrachtet.

Container Architekturen

Container Architekturen wie JEE haben das Problem früh erkannt und versuchen derartige Aspekte herauszuziehen und in einem Container (wie z.B. JBoss, Glassfish, etc.) abzuhandeln. In J2EE 2 beispielsweise wurde "Security" in einer XML-Datei konfigurierbar gemacht, so dass bestimmte Security-Einstellungen für bestimmte Komponenten konfiguriert werden konnten. Dies hat zu Beginn mehr schlecht als recht funktioniert.

Viele Unternehmen begaben sich in eine "XML-Hölle" hinein, die zwar gut zu versionieren war, aber dafür wenig verständlich. Mittlerweile hat sich die Lage verbessert, weil Vieles einfacher wurde. So konnten bestimmte Aspekte nun auch über eine Annotation konfiguriert werden.

Aspektorientierte Programmierung

Wenn schwergewichtige Container wie JBoss oder Glassfish nicht in Frage kommen, bleibt nur noch direktes AOP.

(Weshalb "nur"? Es gibt doch auch die Sonderlösungen, siehe oben!

Was ist "direktes" AOP? Ist nicht erklärt. Vielleicht doch
...bietet sich die Aspektorientierte Programmierung an.)

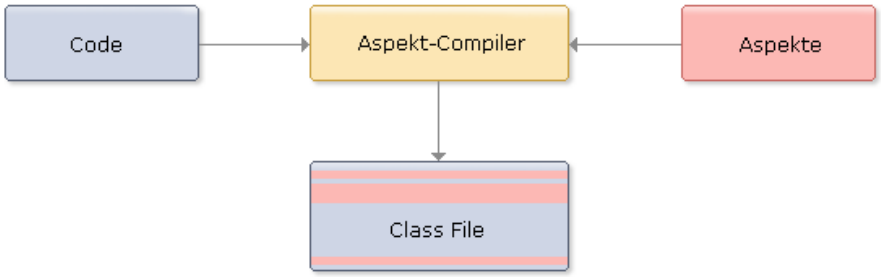
Wie kann man sich das vorstellen?

Hier ein möglicher Ablauf: Der Entwickler schreibt die Methoden genauso, wie er dies immer tut und kümmert sich erst einmal nicht um weitere störende Aspekte. Tauchen dann neue Aspekte auf, werden diese in einer neuen eigenen Methode codiert und möglichst so benannt, dass die zugehörige zentrale Methode identifiziert werden kann.

Weiterhin wird geeignet spezifiziert, an welchen Stellen im Code die Aspekte hineingehören. Hier wünscht man sich eine geeignete DSL (Domain Specific Language) mit geeigneten Wildcards über die alle Aspekte in den eigentlichen Code hineingezaubert werden.

Das Erfreuliche an dieser Vorgehensweise ist: diesen "Zauber" gibt es bereits!

Es ist der Aspekt-Compiler (oder AspectJ Compiler) der diese Aufgabe übernimmt. Entsprechend einer Integrationsanleitung in der beschrieben ist, an welche Stelle welcher Aspekt gehört, "webt" der Compiler die zusätzlichen Aspekte in den Code bzw. in die class files (s. Abbildung).



Die Funktionsweise von AOP

Das Programm kann danach ausgeführt werden, alle Aspekte werden berücksichtigt, aber der Code wird nicht mehr verschmutzt

Die Aspektorientierte Programmierung hat viele Vorteile aber auch einige wenige Nachteile, die in den folgenden Abschnitten besprochen werden. Doch zunächst ein praktisches Beispiel.

8.3 Praktische Übung mit AspectJ

In diesem Teil wird ein schneller Einstieg in die AOP Praxis vermittelt.

Ursprünglich gab es einige AOP Projekte und Werkzeuge. Die bekanntesten sind:

- **AspectJ**: http://www.eclipse.org/aspectj/
- **AspectWerkz**: http://aspectwerkz.codehaus.org/

Beide hatten 2000-2005 ein starkes Momentum. Danach hat sich die Entwicklung und der Hype um die Aspektorientierte Programmierung auch wieder etwas beruhigt.

Seit einigen Jahren ist das führende Werkzeug AspectJ unter dem Dach von Eclipse gehostet.

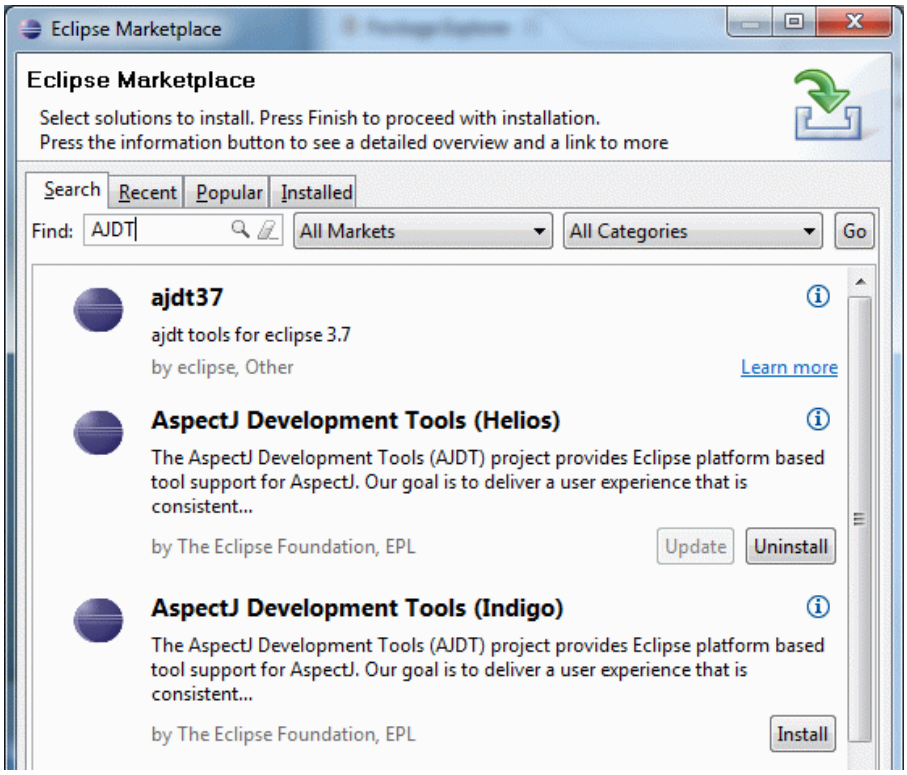
Schauen wir uns die Arbeit mit beiden Werkzeugen kurz an:

AspectJ

Grundsätzlich gibt es verschiedene Möglichkeiten mit AspectJ zu arbeiten. So ist es möglich, die Bibliothek und den Compiler direkt herunterzuladen und zu verwenden.

Im Folgenden wird der komfortablere Ansatz beschrieben, dies gleich in Eclipse zu tun (siehe dazu auch das Eclipse Manual für AJDT). Siehe dazu .

Dazu wurde ein wunderbares Eclipse Plug-In entwickelt worden, welches ganz einfach über den Eclipse Marketplace bezogen werden kann. (Falls Sie Eclipse Marketplace noch nicht installiert haben finden Sie hier eine kurze Anleitung: Eclipse Marketplace).



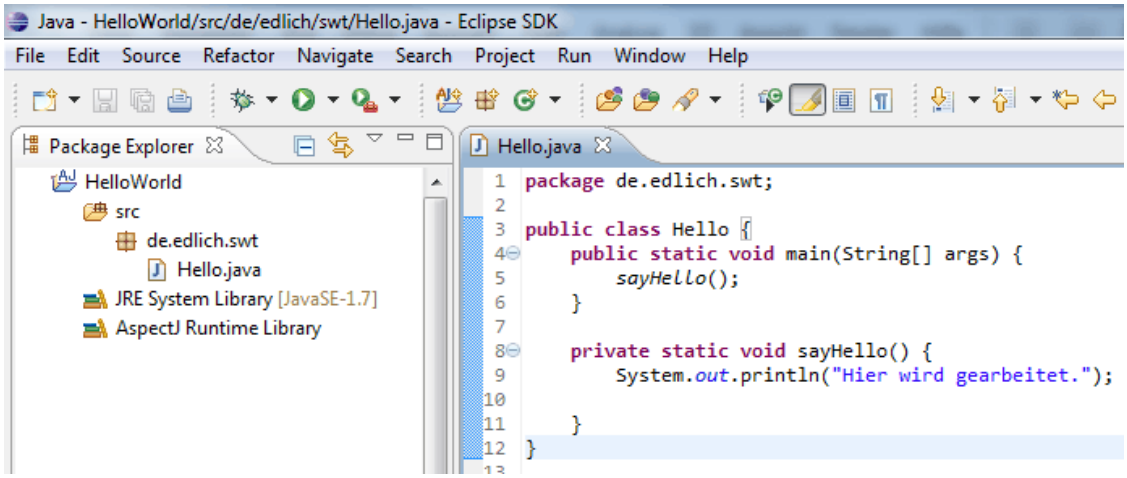
Installation AspectJ

Wie auf der Abbildung zu sehen ist, handelt es sich um das AspectJ Development Tools, kurz AJDT.

Nach der Installation wird wie folgt damit gearbeitet:

Schritt 1

Zuerst wird ein **AspectJ Projekt** erstellt. Sollte dies im new Menü nicht erscheinen, dann ist die Installation fehlgeschlagen.



Schritt 1: Ein neues AspectJ Projekt

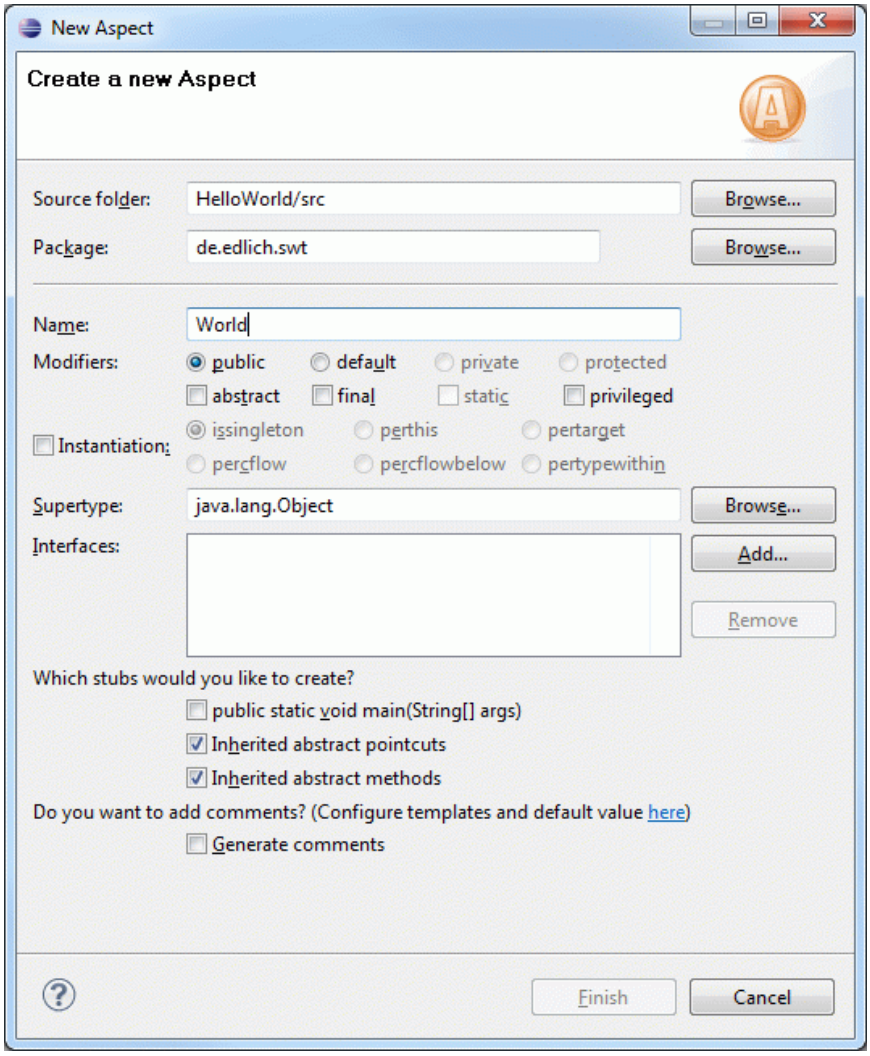
Wir fügen dann in das main eine Methode ein, in der irgendetwas getan wird. Hier lediglich ein System.out.println. In der Realität würde hier beispielsweise wie genannt ein Shopping Cart gefüllt werden.

Tragen sie daher den oben gezeigten Code in die Hello Klasse ein.

Schritt 2

Jetzt muss der **Aspekt angelegt** werden.

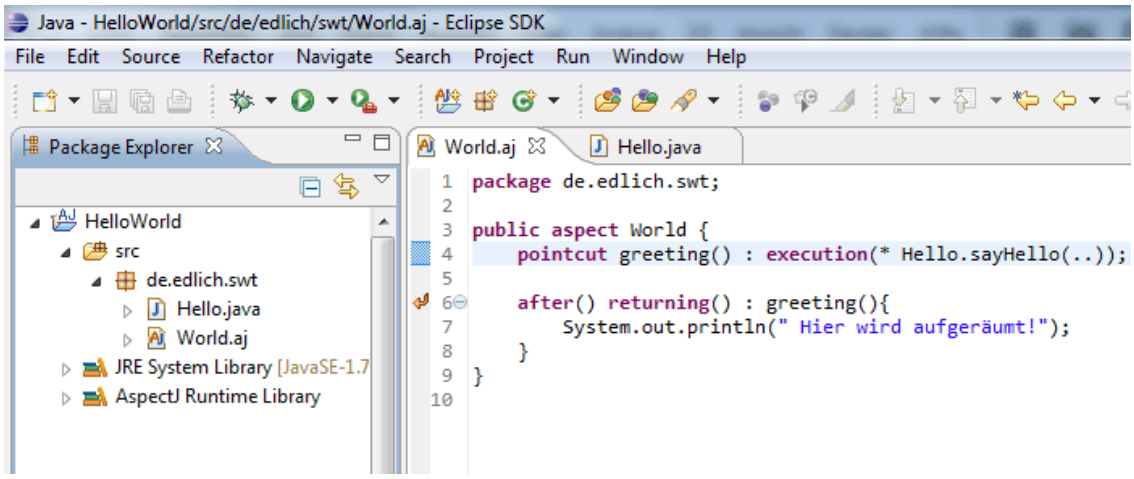
Man öffne daher auf dem Kontextmenü seines Packages New → Aspect und erstelle die Aspekt-Klasse.



Schritt 2: Aspekt anlegen

Schritt 3

Nun gilt es, sowohl den Konfigurationscode, als auch den konkreten Aspekt einzufügen:



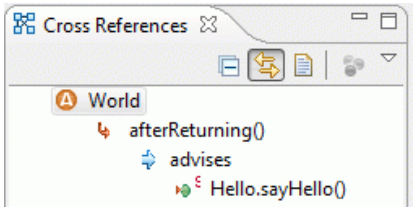
Schritt 3: Hinzufügen des konkreten Aspekts

Man versteht hier intuitiv was passiert:

1. Es wird ein pointcut definiert (was auch immer das ist. Dazu später mehr), welcher definiert, auf welche Klasse und Methode der Aspekt angewendet werden soll.
2. Danach wird definiert, was getan werden soll: Nämlich A) wieder ein System.out und B) wie es getan werden soll. Nämlich *nach* (after) der *Rückkehr* (returning) der Methode sayHello() wird die Methode greeting() aufgerufen.

Man sieht auch hier schon, dass es viele weitere Schrauben gibt, an denen man drehen kann. Die Parameter von sayHello, die Wildcard im Pointcut oder die Begriffe in der konkreten Definition der Aspekt-Methode.

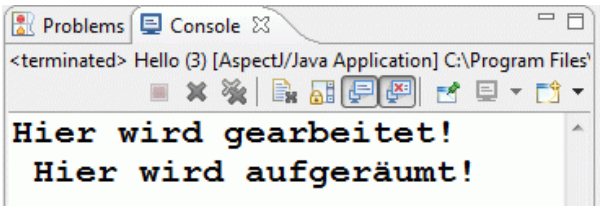
Wir müssten jetzt weiterhin eine View sehen, die uns den Aspekt zeigt. Mit den kleinen braunen Pfeilen in beiden Codedateien kann man nun sehr schön zwischen dem Code und den Aspekten navigieren. Probieren Sie es aus!



Code und Aspekte

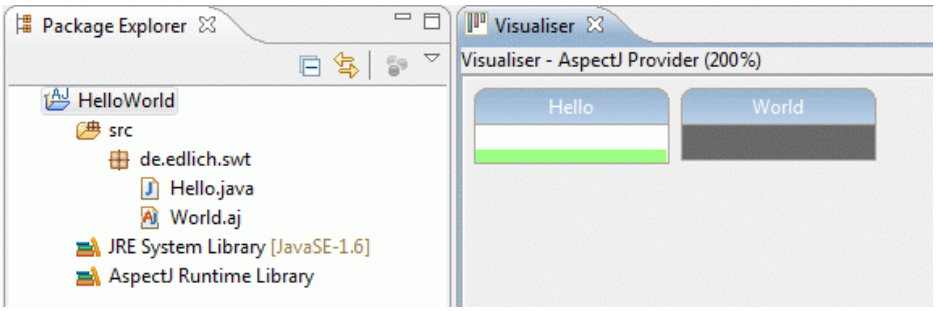
Schritt 4

Nun kann das alles ausgeführt werden. Ruft man also auf der Hello Datei mit der rechten Maustaste Run As → AspectJ/Java Application auf, dann wird im Hintergrund der Aspekt hineingewoben und der ausführbare Code inklusive Aspekt aufgerufen. Es erscheint daher:



Schritt 4: Ausführen

Schön ist, dass das AJDT Werkzeug einen Modus zur Anzeige aller Dateien hat, die mit Aspekten in Berührung kommen.



Aspect Visualization Ansicht

Diese Perspektive finden Sie unter Window →
Open Perspective
→
Others...
→
Aspect Visualization
.

8.4 Praktische Übung mit AspectWerkz

AspectWerkz

Bitte beachten: Dieses Projekt gibt es nicht mehr. Codehaus auch nicht mehr. Man kann es daher nicht mehr selbst so machen. Wir lassen den Code aber hier stehen, da er durchaus interessant ist und zeigt, worum es geht!

Die Arbeit mit AspectWerkz ist ähnlich intuitiv (siehe AspectWerkz Dokumentation . Sie können AspectWerkz hier herunterladen: http://aspectwerkz.codehaus.org/releases.html

Denken Sie daran, dass Sie die heruntergeladen Libraries auch noch einbinden müssen, bevor Sie sie verwenden.

Gegeben sei eine Testdatei / -klasse analog der obigen:

Code

```
package testAOP;

public class HelloWorld {

 public static void main(String args[]) {
 HelloWorld world = new HelloWorld();
 world.greet();
 }

 public void greet() {
 System.out.println("Hello World!");
 }
}
```

Weiterhin benötigen wir den Aspekt, den wir später hineinweben wollen:

Code

```
package testAOP;

import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspect {

 public void beforeGreeting(JoinPoint joinPoint) {
 System.out.println("before greeting...");
 }

 public void afterGreeting(JoinPoint joinPoint) {
 System.out.println("after greeting...");
 }
}
```

Dieses Codestück muss speziell kompiliert werden:

javac -d target -classpath $ASPECTWERKZ_HOME/lib/aspectwerkz-1.0.jar MyAspect.java

Das angenehme an AspectWerkz ist, dass man nun die Konfiguration des Hineinwebens sowohl in XML, als auch mit Annotations vornehmen kann.

Im Falle von XML sähe unsere Konfigurationsdatei so aus:

Code

```
<aspectwerkz>
 <system id="AspectWerkzExample">
 <package name="testAOP">
 <aspect class="MyAspect">
 <pointcut name="greetMethod"
 expression="execution(* testAOP.HelloWorld.greet(..))"/>
    <advice    name="beforeGreeting"    type="before"    bind-
to="greetMethod"/>
    <advice    name="afterGreeting"    type="after"    bind-
to="greetMethod"/>
 </aspect>
 </package>
 </system>
</aspectwerkz>
```

Alternativ können Annotations direkt im Code angegeben werden:

Code

```
package testAOP;

import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspectWithAnnotations {
```

```
 /**
 * @Before execution(* testAOP.HelloWorld.greet(..))
 */
 public void beforeGreeting(JoinPoint joinPoint) {
 System.out.println("before greeting...");
 }

 /**
 * @After execution(* testAOP.HelloWorld.greet(..))
 */
 public void afterGreeting(JoinPoint joinPoint) {
 System.out.println("after greeting...");
 }
}
```

Dies muss wieder speziell kompiliert werden

java -cp $ASPECTWERKZ_HOME/lib/aspectwerkz-1.0.jar org.codehaus.aspectwerkz.annotation.AnnotationC . target

wobei eine minimale XML Datei immer vorhanden sein muss:

Code

```
<aspectwerkz>
 <system id="AspectWerkzExample">
 <aspect class="testAOP.MyAspectWithAnnotations"/>
 </system>
</aspectwerkz>
```

Jetzt kann dies sofort mit einem AspectWerkz Tool gestartet werden, oder so übersetzt werden, dass es alleine lauffähig ist. Dann sieht das Ergebnis wie erwartet so aus:
before greeting...
Hello World!
after greeting...

8.5 Begriffe der AOP

Wir unterscheiden die folgenden wichtigen Begriffe:

Advice: Dies ist der auszuführende Code – der System Level Concern – der in die Core-Level Methode hineingewoben wird.

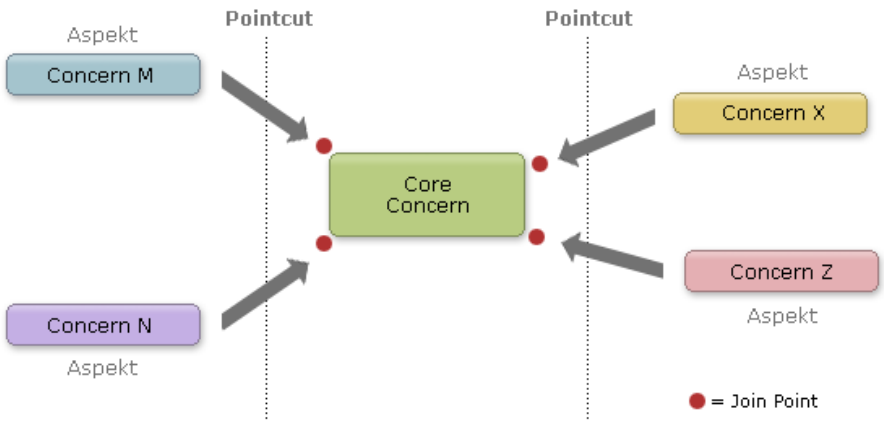
Pointcut: Dies ist die Definition der Orte, an denen tatsächlich hineingewoben wird. Hier kann wie bei einer where Klausel in SQL die Anwendung bestimmt oder eingeschränkt werden.

Join-Point: Dies ist der konkrete Punkt, an dem das Aufrufereignis stattfindet .

Weaving: Der Vorgang des Hineinwebens der fachfremden Concerns in den Zielcode. Vorgenommen wird dies konkret vom Weaver. D.h. dem Programm, welches z.B. .class Dateien neu verdrahten / ändern kann.

Was wollen wir: fachfremde Methoden (**Advices**) definiert (**Pointcut**) konkret (**Joint-Point**) hineinweben, und so nur die Core Concerns sichtbar lassen.

Dazu eine Grafik, die dies erläutert:



Zusammenspiel von Pointcut, Jointpoint und Core Concerns

8.6 Besonderheiten der AOP

Der zentrale Punkt bei AOP ist:

"Separation of concerns"

Also eine Richtlinie, die wir schon ganz allgemein für Klassen und Methoden aus der Lerneinheit CCD - Clean Code Development kennengelernt haben.

Auch dort war einer der Grundsätze: Klassen und Methoden sollen nur das tun, wofür sie verantwortlich sind! In der Praxis ist dies einfacher gesagt als getan, weil System Concerns bzw. nichtfunktionale Anforderungen Ärger machen.

Im Sinne des Separation of Concerns Prinzips kann man zwar fremde Funktionalität auslagern, aber oftmals muss man sie dann in der eigentlichen Klasse oder Methode wenigstens aufrufen. Und schon dies alleine ist oftmals lästig.

Einwand: Man könnte ja für jeden joint-point eine Methode integrieren und dort dann viele Aspekte bzw. Advices aufrufen lassen.

Beispiel:

Code

```
myMethod (params,...){
 aspectBundleA(); // calling many other concerns
 // real code
 aspectBundleB(); // calling many other concerns
}
```

Natürlich geht dies und es stellt wahrscheinlich auch nicht wirklich ein großes Problem dar.

Vorteile von AOP sind hier aber:

- Die Pointcuts können hier viel einfacher und flexibler definiert werden. Eine Concern kann via DSL / REGEXP hundertfach automatisch hineingewoben werden!
- AOP macht dies einfach ordentlich, so dass man wirklich nichts mehr sieht.
- Architekturentscheidungen können mit AOP später getroffen werden, da diese mit AOP leichter später ins System integriert werden könne.
- Die Crosscutting Concerns erzeugen echte Modularität und sorgen dafür, dass wirklich nur minimal gekoppelt wird.

Frage: Welche Parallelen gibt es zwischen AOP und Dependency Injection?

Nachteile von AOP:

- Man sieht nicht mehr, das etwas und was überhaupt passiert. Man darf also seine Advices und Pointcuts nicht vergessen.

- Tools wie AspectJ können nur selbst compilierte Klassen bearbeiten. Nicht aber fertige Klassen wie z.B. solche aus Bibliotheken: z.B. Collections.
- Das Debugging wird erschwert.

Letzteres ist dann auch eines der wichtigsten Argumente der AOP Gegner, um AOP aus Unternehmen zu verbannen. Prüfen sie aber selbst, ob die Vorteile die Nachteile überwiegen oder nicht.

Welche Jointpoints sind denn nun generell möglich?

Aus den vorhergehenden Erläuterungen könnte man voreilig den Schluss ziehen, dass nur der Methodenaufruf mit Aspekten umkleidet werden kann.

Dem ist jedoch nicht so. Wie die folgende Liste zeigt, gehen die Möglichkeiten viel weiter:

- Methodenaufrufe / -ausführung
- Die Ausführung von Konstruktoren (alias Klassenaktivierung)
- Das Lesen und Schreiben von Feldern / Attributen der Klasse
- Objektinitialisierung
- Aktivierung des Exception Handlers

Gerade in einer umfangreichen Exception-Hierarchie sind Aspekte extrem hilfreich. Mit dem geschickten Einsatz sind hier viel bessere Information über die Ursache der Ausnahme zu erhalten, als nur mit dem normalen Exception-Output!

Gehen wir nochmals die Anatomie der Pointcuts durch:

poincut callCalc() : call(public static void HelloWorld.call*(..));

1. pointcut: was folgt ist die Definition eines Pointcuts
2. callCalc(): ist der Name des Pointcuts. In den Klammern könnten noch zusätzliche Parameter angegeben werden.
3. call: Nach dem Doppelpunkt wird mit call angegeben, dass der Aufruf der Methode gemeint ist. Hier soll abgefangen werden.
4. public static void HelloWorld.call: Dies ist die Signatur der Methode. Durch mehrere Pointcuts mit verschiedenen Signaturen können überladene Methoden unterschiedlich behandelt werden. Angegeben ist hier wie üblich die Sichtbarkeit, statisch oder nicht, Rückgabe void, die Klasse und die Methode.
5.
   -

   : "Stern" ist hier ein Wildcard / Regular Expression, die den Name der Methode matchen soll. Gemeint sind hier daher alle Methoden, die mit dem Namen call beginnen.

6. (..): Was folgt sind die Parameter. Offensichtlich zeigt hier ".." an, dass die Parameter egal sind. Alternativ können hier weitere Einschränkungen gemacht werden.

Weitere Schlüsselwörter in Pointcuts sind:

- execution: Für Konstruktoren
- get / set: Für den Zugriff auf Felder
- handler: Für das 'Intercepten' von Exception handlern
- staticinitialization: Für die initialisierung von Klassen. Optional auch ihrer Subklassen

Des Weiteren gibt es eine Menge von Auszeichnungen, um mit Pointcut Mengen umzugehen und diese auch mit logischen Operatoren zu versehen.

Welche Anwendungsgebiete gibt es noch für AOP?

Neben den bekannten und hier schon genannten wie Logging, Security, Persistenz, etc.

- **Eventprogrammierung**: Listener können mit AOP recht einfach integriert werden
- **Frameworkentwicklung**: In der Frameworkentwicklung ist AOP recht beliebt, da die Architektur modular erweitert werden kann. Viele bekannte Frameworks wie JBoss oder Spring bedienen sich AOP-Techniken oder nutzen dieses als zentrales Konzept.
- **Performance Test**: Nichts ist einfacher, als einen Aspekt zu erstellen, der System.current.TimeMillis() enthält und diesen als Pointcut um eine Methode zu definieren.
- **Testen im Allgemeinen**: Mit AOP wird auch das Überwachen privater Attribute möglich. Dies ermöglich dann auch wieder eine ganz andere Herangehensweise in Bezug auf White-Box-Tests. Dazu muss der Aspekt lediglich als privilegiert markiert werden.

8.7 Zusammenfassung AOP

- In dieser Lerneinheit haben Sie gelernt wann und warum Sie aspektorientierte Programmierung einsetzen sollten.
- Sie haben an Hand von praktischen Übungen AspectJ und AspectWerkz kennengelernt und verwendet.
- Die wichtigsten Begriffe im Zusammenhang mit der AOP wurden eingeführt und erläutert, diese sind: Advice, Pointcut, Joint-Point und Weaving.

- Sie kennen nun die Vorteile (z.B. Architekturentscheidungen können später getroffen werden, Crosscuting Concerns erzeugen echte Modularität) und Nachteile (z.B. das Debugging wird erschwert, Tools wie AspectJ können nur selbst compilierte Klassen bearbeiten) von AOP.

8.8 Wissensüberprüfung AOP

Übung AOP-01: Aspektorientierte Programmierung

1. Welches Problem löst AOP?
2. Welche Begriffe sind damit im Zusammenhang wichtig?
3. Wir funktioniert AOP technisch?

Bearbeitungsdauer: 15 Minuten

8.9 Literatur zu AOP

Oliver Böhm:

Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP

Dpunkt Verlag, ISBN-13: 978-3898643306

Ramnivas Laddad:

AspectJ in Action

Second Edition, Manning Publ., 2009, ISBN: 1933988053 (Tipp Ihres Coaches!)

Adrian Colyer , Andy Clement, George Harley , Matthew Webster:

Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ

Addison-Wesley, 2004, ISBN-13: 978-0321245878

Russ Miles:

Aspectj Cookbook

O'Reilly Media, 2004, ISBN-13: 978-0596006549

Joseph D. Gradecki, Nicholas Lesiecki:

Mastering AspectJ: Aspect-Oriented Programming in Java

2003, Wiley, ISBN-13: 978-0471431046

9 FPR - Funktionale Programmierung

Autor: Prof. Dr. S. Edlich

9 FPR - Funktionale Programmierung
9.1 Überblick und Lernziele FPR
9.2 Paradigmenvergleich
9.3 Funktionale Paradigmen in Clojure
9.4 Funktional: Clojure
9.5 Hybridsprache: Scala
9.6 Kurzübersicht: Haskell & Erlang & Co
9.7 Zusammenfassung FPR
9.8 Wissenüberprüfung FPR
9.9 Literatur FPR

9.1 Überblick und Lernziele FPR

In den 90er Jahren waren viele davon überzeugt, dass Gott ebenfalls objektorientiert programmiert. Das Entwickeln in Klassen und Objektinstanzen erlebte seine Blütezeit . Programmiersprachen wie C++, Java und im Jahre 2001 auch C#. Und sogar Ruby und Python wurden Mitte und Anfang der 90er entwickelt. Andere Sprachen mit

anderen Paradigmen wie Lisp oder Prolog dümpelten vor sich hin und konnten keine nennenswerten Marktanteile gewinnen.

Doch es mehrten sich Zweifel, dass OO schon der Weisheit letzter Schluss sei. Insbesondere mit der Reife der Java VM und der .NET CLR explodierte dann ab ca. 2005 die Welt der Programmiersprachen. Einen eigenen Compiler zu schreiben, der auf diese Ziel-Plattformen übersetzt ist damit plötzlich ganz einfach geworden, auch wenn wichtige Features wie z.B. *Tail Recursion* oder *Closures* in Java viel zu lange gefehlt haben. So erschienen sogar Bücher, die gleich einer Bauanleitung gleichen, seine eigene Programmiersprache zu entwickeln:

Terrence Parr, ***"Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages"***, 2009, Pragmatic Programmer, ISBN: 978-1-93435-645-6

Und so findet sich jetzt im berühmten TIOBE Index (*http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*) eine viel buntere Vielfalt der Sprachparadigmen, als noch in den 80er und 90er Jahren, die von imperativen und objektorientierten Sprachen beherrscht waren.

Lernziele

Lernziele

Ziel dieser Lerneinheit ist es, die gewohnten Pfade der objektorientierten und imperativen Programmierung zu verlassen und etwas komplett anderes zu lernen!

Gliederung

Gliederung

- Einführung
- Vorstellung verschiedener Paradigmen
- Wissensüberprüfung und Übungen

Dabei wird bewusst auf radikale Konzepte gesetzt (Clojure, Haskell, Prolog) um den geistigen Gewinn zu maximieren.

Viel Spaß dabei!

Referenzen:

- How to Become Skilled at Learning Programming Languages,

9.2 Paradigmenvergleich

Gerade in der Hochschullehre mit der begrenzten Zeit ist es schwer, die Vielfalt der Paradigmen zu lehren, die in der Programmiersprachenwelt herrschen. Unter Paradigma wird hier daher die grundsätzliche Vorgehensweise verstanden, wie ein Problem gelöst, bzw. eine Arbeitsanweisung für den Computer / Compiler / Interpreter formuliert wird. So bleibt in der Regel in der Lehre nur Zeit für ein Paradigma. Und zwar genau das populärste oder das in der Wirtschaft am gefragtesten. Dennoch lassen sich Probleme oft viel besser mit verschiedenen Paradigmen bzw. den entsprechenden Werkzeugen lösen. Diese Vielfalt zu vermitteln ist das Anliegen dieser Lerneinheit.

Referenzen: Uncle Bob SkillsMatter: "The last programming language",

Typsystem

Man unterscheidet:

- stark (Java)
- schwach (C)
- statisch (Java)
- dynamisch (Ruby)

Die Ausprägung des Typsystems hat starke Auswirkungen auf die Sprache und wie diese sich anfühlt. Damit einher geht der Streit der Produktivität.

Anhänger der starken und statischen Typisierung argumentieren, dass man viele Programmierfehler hier schon zur Compilezeit findet und dass dies Zeit spart und sicherer ist. Auf der anderen Seite argumentieren Verfechter von dynamischen Programmiersprachen, dass alleine schon das Testen einfacher ist und man so schneller ist und besser Fehler finden kann, als bei statischen Sprachen.

Programmiermodell

Neben den populären Modellen wie die objektorientierte und funktionale Programmierung, gibt es prozedurale und logische Sprachen, sowie hybride Ansätze. Mit Io (einer Programmiersprache, die von Smalltalk inspiriert wurde) gibt es ebenfalls das Prinzip 'prototype'.

Dennoch scheint es der Trend zu sein, möglichst viele Paradigmen anzubieten und so um die Gunst der Entwickler zu buhlen. Clojure ist ein bekannter Vertreter, bei dem Verfechter immer wieder beweisen, wie leicht fast jedes Programmiermodell integriert werden kann (z.B. Clojure mit Multimethods).

Interaktion

Es gibt viele Varianten:

- Interpretiert (Ruby)
- Kompiliert (CPP)
- Mit VM (Scala, Clojure, Java)
- Ohne VM (D, CPP)

Core Data Structures und Ablaufkonstrukte

Hier gibt es z.B. die folgenden Fragestellungen:

1. Wie sehen die Collections der Sprachen aus?
2. Wie sieht Pattern-Matching aus?
3. Welche Kontrollstrukturen gibt es?

Gerade bei dem letzten Punkt kann man erstaunt sein und denken, alle Sprachen müssten *if*- und *while*- ähnliche Statements haben. In Clojure denkt man beim Ablauffluss jedoch nie in *while* und selten in *if*.

Core Features, die die Sprache einzigartig machen

Viele Sprachen haben Alleinstellungsmerkmale oder Besonderheiten, die die Einzigartig ausmachen. Ein paar Beispiele:

- In Erlang ist dies sicherlich das Message Passing zwischen Millionen von Agenten. Und damit auch das Design for Crash / Failure.
- Bei Clojure ist das gewiss die Idee, das Concurrent Programming in den Griff zu bekommen.

Vergleich

Objektorientiert

Hier wird versucht den Status in Form von Variablen - zusammen mit Methoden, die diesen Status bearbeiten - zusammenzufassen. Das Ergebnis ist eine Klasse mit der Instanz eines Objektes.

Grundlegende Idee ist *Encapsulation*. Das Prinzip der Datenkapselung erlaubt es dem Entwickler harte Auflagen an die Möglichkeiten der Manipulation von States zu stellen. Da viele Methoden allerdings immer noch auf die Daten im Objekt zugreifen können, ist das Einhalten von *Enforcements* nicht trivial. Zudem beziehen sich Constraints leider immer nur auf ein Objekt.

Weiterhin gibt es entweder Objektidentität mittels ihrer Attributgleichheit zu definieren (extrinsisch) oder aber eine eigene Objektidentität einzuführen (unabhängig vom Zustand der Attribute).

Zusammenfassend kann gesagt werden, dass OOP zwar an einigen Stellen versucht, die Manipulation von Zuständen zu kontrollieren oder zu sichern. Dies gelingt aber nur teilweise.

Funktional

Grundlage ist das Lambda Kalkül, welches ebenfalls völlig zustandslos arbeitet. Klassiker sind hier Haskell und ML.

Herausragendes Argument ist die völlige Vermeidung von sich verändernden Zuständen. Dies schließt jede Art von Seiteneffekten aus. Der Code ist deterministisch. Das Ergebnis ist durch die Parameter bereits definiert.

Dies löst zwar nicht alle Probleme des Testens, aber grundlegend sind funktionale Programme viel einfacher zu testen.

Allerdings besteht die Gefahr, die Verwendung von Zuständen dadurch zu lösen, dass zusätzliche Funktionen eingeführt werden, und diese neue Parameter bekommen, welche dann für die Manipulation einer "Variablen" verwendet werden.

In vielen Fällen kann die funktionale Variante komplexer aussehen, als eine zustandsbehaftete imperative Lösung. Vor allem wenn modularisiert werden soll.

Mit der funktionalen Variante bekommt man:

- bessere Testbarkeit
- höchste Sicherheit
- geringere Komplexität durch definierten Ablauf
- und vor allen Dingen Zustandslosigkeit

In der anderen Variante bekommt man kleineren Code, der all die obigen Dinge als Gegenteil / Nachteil beinhaltet.

Logisch

Hier programmiert man in einem deklarativen Stil, der besagt, was es zu tun gilt, aber nicht, wie es zu tun ist.

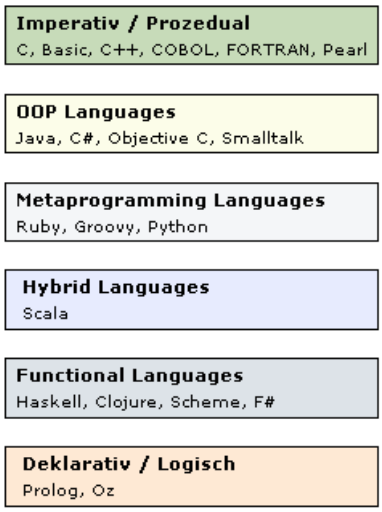
Es werden also Regeln oder Axiome formuliert, die ein Problem behandeln und transformieren, so dass im Zusammenwirken eine Lösung erreichbar ist.

Prolog ist hier das Paradebeispiel. Aber auch Oz hatte dafür interessante Ansätze.

Die Regeln / Axiome verwenden keinen veränderbaren Zustand, was schon einmal von Vorteil ist.

Dennoch ist die Ablaufkontrolle intern im System verborgen und kann zu Halteproblemen führen.

Die nachfolgende Grafik enthält eine Übersicht über verschiedene Programmierparadigmen.



Programmierparadigmen

Es gilt zu beachten, dass die Grenzen fliessend sind und sehr viele Sprachen in verschiedene Kategorien fallen. So ist D beispielsweise sehr maschinennah, implementiert aber auch OO und einige wenige Funktionale Konzepte. Sehr viele Sprachen versuchen auch zunehmend Multiparadigm-Sprachen zu sein, indem alle möglichen Konstrukte im Nachhinein hinzugefügt werden. Dementsprechend gibt es noch sehr viele weitere Kategorien wie scripting, stack-based, compiliert / interpretiert, assembler, etc.

Einen grafischen Überblick mit weiteren Programmierparadigmen finden Sie hier: http://www.info.ucl.ac.be/~pvr/paradigms.html

9.3 Funktionale Paradigmen in Clojure

Referenzen zur Funktionalen Programmierung im allgemeinen:

- Slava Akhmechet (defmarco.com), "***Functional Programming For The Rest of Us***",

- defmacro, ***"The nature of Lisp"***,
- John Hughes, ***"Why funnctional Programming Matters"***,
- Aus Java-Script Magazin:
- Kleiner Blogpost von Uncle "Bob" Martin
- Deconstructing Functional Programming
- (Build your own Lisp!)

Die Clojure Philisophie

Die Architektur und das Design von Clojure sind stark von folgendem hochinteressanten Paper beeinflusst:

Ben Moseley, Peter Marks

"Out of the Tar Pit"

6 Feb 2006, SPA 2006 Functional Relational Programming

(Interessant auch http://awelonblue.wordpress.com/2012/10/21/local-state-is-poison!)

Der nachfolgende Abschnitt orientiert sich an diesem Paper und zeigt die wichtigsten Beobachtungen auf.

Wichtig

> Es lohnt sich unbedingt dieses Paper zu lesen, da es an den Fundamenten rüttelt und die Basis für neue Paradigmen, Sprachen und sogar Datenbanken darstellt. Und tatsächlich war dieses Paper auch Basis für Clojure und sogar Datenbanksysteme wie Datomic.

Die wichtigsten Kernaussagen sind die Folgenden:

Brooks identifiziert für große Softwaresysteme die folgenden Probleme:

1. Complexity -> Komplexität
2. Conformity -> Conformität (Zugehörigkeit / Ähnlichkeit)
3. Changeability -> Änderbarkeit
4. Invisibility -> Unsichtbarkeit

Moosley und Marks argumentieren, dass sich die Punkte 2, 3 und 4 auf die Komplexität zurückführen lassen.

Die wichtigsten Ursachen für das Kernproblem der Softwarekrise **Komplexität** sind:

- State -> Zustand

- Code Volume -> Codeumfang
- Flow of Control -> Programmablauf

Anmerkung

Bei den Gründen für das Scheitern von Softwareprojekten steht Komplexität kaum in der Hitliste. Meistens stehen Anforderungen, Kommunikation, etc. im Vordergrund. Wissenschaftliche Untersuchungen haben jedoch gezeigt, dass die inhärente Komplexität ein ganz wichtiger Faktor ist, der fast immer übersehen wird..

Dazu zwei Zitate von Backus aus dem Jahre 1978:

"I conclude that there are two ways of constructing a software design: One way is to make it so simple, that there are obviously no deficiencies and the other way is to make it so complicated that there are no abvious deficiencies. The first method is far more difficult."

und

"Simplicity is Hard"

Programmcode zu verstehen kann auf zwei Arten geschehen:

- Testen
- *Informal Reasoning* (Überlegen / Verstehen)

Implizit gehen wir davon aus, dass wir ordentlich testen und dass wir Programme verstehen können. Dabei überschätzen wir oftmals unsere Fähigkeiten.

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
Edsger W. Dijkstra

Gemeint sind hier allerdings klassische Methoden der Unit- und Integrationstests. Glücklicherweise gibt es neue Ansätze wie z.B. das generative Testen.

Komplexität durch Zustand

Wir alle sind viel öfter mit diesem Problem konfrontiert:

- "Versuche es nochmal"
- "Versuche das Dokument neu zu laden"
- "Starte das Programm neu"
- "Reboote den Computer"
- "Installiere das Programm neu" oder sogar

- "Installiere den Computer neu und dann das Programm"

sind alles alltägliche Auswüchse von zustandsorientiertem Programmieren.

Anmerkung

Die zustandsorientierte Programmierung mit Variablen oder das Arbeiten mit zustandsorientierten Datenbanken entstammt den 60er / 70er und 80er Jahren, als Systemressourcen beschränkt waren. Die damals geltenden Voraussetzungen sind lange nicht mehr gültig. Doch nur wenige denken über entsprechende Konsequenzen nach. (siehe dazu auch die Keynote von Rich Hickey / GOTO 2012)

"From the complexity comes the difficulty of enumerating, much less understanding, all the possible of states of the program, and from that comes the unreliability."

"Computers [...] have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do."

Generell wird hervorgehoben, dass ein Test nur einen Zustand abdeckt. Die zu testende Einheit aber um viele Größenordnungen mehr Zustände annehmen kann, die dann nicht getestet werden.

Jedes Bit eines zusätzlichen Status verdoppelt die Anzahl der möglichen Zustände, so dass in üblichen Programmen eine unüberschaubare Anzahl von Zuständen entsteht.

Ein weiteres Interessantes Beispiel aus Moseley / Marks ist der Einfluss der Kontrolle auf Komplexität:
a := b + 3
c := d + 2
e := f * 4

Offensichtlich werden in diesem Codestück zwei orthogonale Dinge vermischt:

1. Die Variablen und Konstanten stehen in Relation zueinander
2. Gleichzeitig wird aber auch eine Aussage über den Ablauf getroffen, der so unter Umständen gar nicht gewollt oder nötig war.

Wie bekannt ist, neigen große Codemengen dazu exponentiell komplexer zu werden.

Wichtig

> Wir unterscheiden ***Essential Complexity***, also die dem Problem inhärente Komplexität. Für einen Quicksort muss man nun mal bestimmte Schritte ausführen. Auf der anderen Seite ist ***Accidential Complexity*** alles das, was nicht Essential Complexity ist. Also unnötige Verwirrung durch Randbedingungen wie die Programmiersprache (30.000 Klassen), Performanceaspekte (Bitfelder, Bloom-Filter, etc.) oder sonstige Dinge wie Infrastrukturzwänge.

Weiterführendes:

- Simple Made Easy, Rich Hikey at Strageloop, Oct 2011
- The future of programming

Funktionale Paradigmen

Daraus ergeben sich die folgenden **allgemeingültigen** *Funktionalen Paradigmen*:

- Statt mit vielen Zustände zu arbeiten, arbeitet man hier mit unveränderlichen Datenstrukturen (und einfachen Abstraktionen). Die Idee ist, lieber dreifach berechnen zu lassen, als Berge von Zuständen mitschleppen, die später keiner mehr versteht.
- Wir nehmen Higher Order Functions und zaubern (*composition*) die lieber in Richtung deklarativer Programmierung zusammen, als mit imperativen und iterierenden Kontrollstrukturen zu arbeiten.

Entsprechende Ansätze gibt es auch in Guava (siehe Lerneinheit AJV - Advanced Java) und in http://functionaljava.org

Hier einmal ein Beispiel aus Emrick, was alles passieren kann wenn z.B. (sogar anders als in Java!) Strings mutable sind, was sie in Ruby sind!

```
>> s = "hello"
=> "hello"
>> s << "*"
=> "hello*"
>> s == "hello"
=> false
```

Ruby
Mutable Strings in Ruby

Sie ahnen also, was passiert wenn viele Threads Referenzen haben und jeder fleißig ändert...

Dazu noch ein Beispiel:

```
>> h = {[1, 2] => 3}
=> {[1, 2]=>3}
>> h[[1,2]]
=> 3
>> h.keys[0] << 3
=> [1, 2, 3]
>> h[[1,2]]
=> nil
```

Ruby
Mutable Strings in Ruby

Von dieser Sorte gibt es unendlich viele Beispiele. All dies veranschaulicht hoffentlich, dass die Komplexität von Programmcode mit der Anzahl der Variablen exponentiell (2^n) steigt.

Übung PPD-01: Lösung mit Clojure

Aufgabe: Wie würde das letzte Problem in Clojure aussehen (oder sogar das erste)?

Bearbeitungszeit: 15 min

9.4 Funktional: Clojure

This all moves to https://github.com/edlich/langRefs and the complete Clojure part will be deleted!

Konzepte & Ideen

- Functional => pure / first-class / Higher Order Functions
- Immutable => sichere und korrekte Funktionen
- Hohe Expressiveness => smart mit weniger Klammern
- DSL ready / Konzepte von OO oder logischer Programmierung leicht realisierbar
- Konzepte vieler Sprachen abbildbar
- Auf Java gehosted => Rund doppelt so schnell wie Ruby 1.9 (Nach Language Shootout)

- perfekte Java Integration
- Angepasst auf Multicore / multi-CPU mit MVCC daher immer Thread Safe
- Starkes Makrokonzept (Scala wird das erst einbauen)
- dynamisch und stark typisiert
- viel abstrakteres Programmieren
- Homioiconic => Code = Data

Übrigens: loops und klassische Design Patterns sind Code-Smells!

Literatur

- Emerick, Carper, Grand, *"Clojure Programming"*, OReilly, 2012, ISBN 9781449394707
- Halloway, Bedra, *"Programming clojure"*, Pragmatic BookShelf, 2012, ISBN 9781934356869
- Fogus, Houser, *"Joy of Clojure"*, Manning, 2011, ISBN 9781935182641
- Rathore, *"Clojure in Action"*, Manning 2011, ISBN 9781935182597
- VanderHart, Sierra, *"Practical Clojure"*, Apress, ISBN 9781430272311
- Kamphausen, Kaiser, *"Clojure: Grundlagen, Concurrent Programming"*, dPunkt, 2010 ,ISBN 9783898646840
- Brian Marick, "Functional Programming for the Object-Oriented Programmer",
- Dominik Herzberg, "Funktionale Programmierung mit Clojure", *frei!*
- Clojure Cookbook, Vanderhart, Neufeld, OReilly oder frei hier:

Weblinks

- Homepage:
- Clojure Api:
- Clojure Doc Apis:
- Clojure Doc:
- API als Baum:
- Ein Online repl:
- Alle Sprachen interaktiv:
- Learn Clojure in 15 Minutes
- Die Google Gruppe:
- Clojure Blog:
- Steward Halloway Blog:
- **Cheat Sheet**: ***

- **QuickRef**:
- **Crash Course**:
- **From the ground up**:
- Quick Reference von jf (thx A.Martin)
- Introduction to Clojure Part I
- Programming Guide: http://blackstag.com/blog.posting?id=5
- Functional programming in Clojure [Functional programming in Clojure]
- Volkman Guide: http://java.ociweb.com/mark/clojure/article.html and more Resources
- Wiki: http://en.wikibooks.org/wiki/Clojure_Programming
- Clojure Tutorial for the non lisp programmer http://moxleystratton.com/clojure/clojure-tutorial-for-the-non-lisp-programmer (thx A.Martin)
- Clojure Dev Team: http://dev.clojure.org
- getting started: http://dev.clojure.org/display/doc/Getting+Started
- Learn Clojure: http://learn-clojure.com/
- **Clojure for the brave and true**: http://www.braveclojure.com/
- Langref: http://langref.org/clojure (Viele Sprachen!)
- Clojure Toolbox: (list of libs) http://www.clojure-toolbox.com
- Labrepl (learning env): https://github.com/relevance/labrepl
- (Marks List: http://java.ociweb.com/mark/programming/Clojure.html)
- Nette Rätsel mit vielen Lösungen: http://clojure-euler.wikispaces.com
- Code worth reading: https://news.ycombinator.com/item?id=4011359
- STM and Clojure: http://java.ociweb.com/mark/stm/article.html#Overview
- 10 one liners to impress your friends ;-) http://www.verious.com/article/10-clojure-one-liners-to-impress-your-friends/
- Clojure from the ground up
- Decompiling Clojure

Videos

- LinkedIn Learning hat ein paar Clojure Kurse
- Stuart Halloway, Simplicity ain't easy
- Rich Hickey on Reducers
- Chas Emrick, "What Sucks about Clojure...and Why You'll Love It Anyway", infoQ from Clojure/West.
- Stuart Sierra, "Clojure in the Large"

- Neil Ford, "The Curious Clojurist"
- Interview with Rich Hickey, "Comping close to Clojure",

Foren

- Google Group International
- Google Group Deutschland
- IRC Kanal: #clojure (der Dozent ist dort auch manchmal und dort auch gerne jederzeit zu einem Treffen bereit)

Training

- 4Clojure
- Clojure Koans

Artikel

- Uncle Bob, **"Why is it called functional"**

Entwicklungsvorgehen

- OO Development Style

Zu Komponente / Klasse springen, berechnen und lokale oder andere Daten verändern.

- Clojure Pure Development Style

Aufbau eines Zwiebelmodells. Vollständiges verzichten auf persistente Daten!

- Clojure let / final Style

Definiere Top-Level Task. Unterteile in Sub-Tasks. Jeder Sub Task kann final Daten mit let binden (final-'freezen') und diese zu Anfang berechnen. Danach wird weiterberechnet.

Wer nutzt Clojure?

Zum Beispiel:

- BigML
- Storm
- Cascalog
- Nokia
- Twitter
- Akamai
- Factual
- Prismatic!!

- Groupon
- Heroku!! Als PaaS!
- und 1000 mehr...

Zusammen mit Scala auf Platz 1 im ThoughtWorks Tech Radar 3/12.

Wie lerne ich Clojure am besten?

- Reales Projekt machen / Problem lösen.
- Das Oreilly Buch durcharbeiten.
- In Stuard Halloways Buch schmökern.
- Das Cheat-Sheet immer parat haben und lernen.
- Immer jemanden Fragen können: IRC, Edlich (!), ...

Vorträge

- Howard Lewis Ship, ***Towards the essence of programming***

9.4.1 Basics

Aufruf

java -cp C:\Programme\clojure\clojure.jar clojure.main %1 ;

oder

java -jar C:\Programme\clojure\clojure.jar %1; für run file

Ohne %1 wird dann der REPL aufgerufen

IDE

- Clockcounterwise für Eclipse (gut z.B. Bracket Modes, selten Probleme)
- IntelliJ La Clojure (guter Editor und gut für Java Integration)
- Emacs (für Gurus, d.h. 68% aller Clojure Hacker)
- gvim + ?
- Leinigen Build-Tool (siehe unten, keine IDE) + Editor
- Light Table, relativ junge IDE mit innovativem Bedienkonzept

HelloWorld

(println "Hello World!") ; Ich bin ein Kommentar (mit #_ forms ausschalten!)

Quoting

(println '(1 2 3) "no eval here!") ; quote = kein eval (keyword too)

Basic Function

```
(defn HelloPow [name val] ; Basic Function
   „Liefert das quadrat des Wertes val zurück“ ;
 Funktionsbeschreibung
 (str „Hi „, name, „!\nThe power of „, val, „ is „, (* val val)))
(println (HelloPow „Stefan“ 5)) ; Der echte Aufruf
```

Basic Function

Bindings

(def message "Hello Stefan!") ; bind String to final keyword

Ausführung

- Entweder direkt über obiges Script
- Im REPL via

(load-file "c:/hw.clj")

Coding Conventions

- Referenz:

1. Dateiendung ist .clj
2. Worttrenner ist ein - (Unterstrich nur bei Dateien wenn unbedingt nötig)
3. Funktionen mit Nebeneffekten bekommen ein ! (Bang)
4. Prädikate = true/false Functs bekommen ?
5. Alle schließenden Klammern kommen auf eine Ebene.

Dokumentation

(defn HelloPow [name val] ; Basic Function
„Use (doc HelloPow) to display doc“ ...)
; and (find-doc "*Pow") to find anything

Kommentare

(println "Hi!") ; I am a comment!
(comment
(println "Hi!")
)

(Randbemerkung: Mit #_ gibt es noch weitere comment Makros)

Reader

> (read-string "(+ 1 2)") ; Der Reader ist wichtig!
3

Der Reader kann noch mehr:

- Definition

    1.1. ()

  ; Referenz

    1.1.

  ; dann noch `,~,~@ für Makros und ^ für Metadata.

Beispiele

(+ 1 1) => 2
(+ 3 3 3) => 9
(/ 2.0 4) => 0.5
(mod 5 4) => 1
(/ 16 4 4) => 1
(< 1 2 3) => true
(< 1 3 2 4) => false

Übung FPR-02: Erstellung einer Clojure Datei

Aufgabe: Erstellen Sie eine sinnvolle Clojure Datei und starten Sie diese

1. via Script mit load-file und
2. via REPL.

Bearbeitungszeit: 20 min

Hinweis

Wird in einer Datei eine Funktion a deklariert, die b verwendet und b erst danach deklariert, so ist b noch nicht bekannt. Da hier kein 2-Pass Compiler vorliegt kann man aber mit: (declare b) die Funktion b schon mal bekannt machen und versprechen, dass sie noch deklariert wird.

9.4.2 Basistypen

- Clojure nutzt (und erfand) die edn = Extensible Data Notation

TYP	BESCHREIBUNG
nil	=> wie null in Java ist logisch falsch
Funktion	=> First Class Citizen. Können übergeben und 'returned' werden
Boolean	=> true, false Beispiel: (< 1 0) => false
Char	=> Character: \c, \newline, \space, \tab, \return, \backspace
String	=> "String \n mit Umbruch" (str "Hello" "Stefan") concats. Can span more lines!
Zahlen	=> 23 42,0 (Integer, Long, Double, BigInteger, BigDecimal)
Ratio	=> 24/5 wird als 24/5 gespeichert.
Symbole	=> (def +erlaubt-*123 42) (def 11myfinal) => FALSCH
Keywords	=> (keyword? :white-knight) => true (keyword "black") => :keyword (keyword "gamedata/white") ; keyword white in gamedata Namespace

Grundtypen in Clojure

Auf **Zahlen** sind viele Operationen definiert:
inc, dec, quot, rem, min, max (min, max takes a list!)
<, <=, >, >=, ==, neg?, pos?, zero? (takes lists!) (< 2 5 17/2)

Verwende **type** oder **class**, um den Typ zu bekommen: (type 42.0)! *(type uses more Clojure Metadata and Class more the Java type info)*

(type '(4 6 8)) => clojure.lang.PersistentList

Vertiefung

Das besondere an Keywords: es sind Funktionen, die sich selbst in der übergebenen Collection suchen.

Dazu ein Beispiel:

```
(def mensch {:name "Stefan Edlich"
 :hobby "Kiten & Clojure"})
;= #'user/mensch
(:hobby mensch)
;= "Kiten & Clojure"
```

Keywords
Keywords suchen sich selbst..

Hinweis

Vorsicht: Ratios gibt es nicht in Java!

Strings

- Clojure String API: http://richhickey.github.com/clojure-contrib/string-api.html
- String Utils 1: http://richhickey.github.com/clojure-contrib/str-utils-api.html
- String Utils 2: http://richhickey.github.com/clojure-contrib/str-utils2-api.html

Behandlung analog zu Java
(.equals "Stefan" "Edlich") => false
(.substring "Is this not pung?" 3 7) => "this"

Mehr über Literals

- Regular Expressions: (re-seq #"(...) (...)" "foo bar") (nobackslash escape required)
- Kommentare: Neben

    °

  gibt es noch

    2.1. _

  . Also liefert '(+ 1 2 #_(* 2 2) 8) beispielsweise (+ 1 2 8) zurück. Dann gibt es noch (comment (println "hello")) aber das evalt zu nil!

- Kommas: Kommas evaluieren zu whitespaces: (= [1 2 3][1, 2, 3]). Machen sich aber gut in der Lesbarkeit von z.B. maps.

Typen Konvertierung

String in eine Zahl:
(type (read-string "42"))
=> java.lang.Long

Zahl in einen String:
(str myLong)

Number to Char:
(char 20)

Character to int:
(int \A)

Unbedingt auch into anschauen:
(into [] '(1 2 3 4)) ==> [1 2 3 4]

9.4.3 Collections

Referenzen:

1. Collections and Sequences in Clojure: (!)
2. Fogus, Houser, "An In-Depth Look at Clojure Collections",

- **List**: '(a b :name 12.5) ; interpreted as calls so quote
- **Vector**: ['a 'b :name 12.5]/ ; ...
- **Map**: {:name "Chas" :age 31} ; Hashtable
- **Set**:

  4.1. {1 2 3}

  ; bin voller uniques

Die Sequence Lib ist die generelle Abstraktion. Vielen Funktionen ist es daher zum Glück egal, was sie bekommen. **Collections** Index in Collections immer wie in C. Starten mit 0!

Tipp 1: Versuche immer mit abstrakten Sequences zu arbeiten wenn es auf abstrakten Datenstrukturen sein soll. Dann bleibe in der Sequence World. Wenn Du konkret arbeiten willst, dann bleibe in der Collection world. Nutze dann z.B. pop und eben nicht drop-last.

Beispiel:

(def b [4 5 6])
(conj (drop-last b) "xx") ; ergibt ("xx" 4 5) !

Hier verliere ich die Vecor Eigenschaft. Weil ich mit drop-last die collection Welt verlasse. Mit pop statt drop-last würde ich in der Welt bleiben!

Generic Ops:
(count mycoll) => 42 ;elements

List

Ideal für vorne anhängen oder lineare suche nach hinten. Wie linked list!

Wenn in einemEval Ausdruck, dann wird das erste Element als Operator interpretiert. Um dies zu verhindern muss man quoten.
> (+ 2 4 6) ;=> 12
> (2 4 6) ;=> Error 2 ist keine Funktion (IFN)
> '(2 4 6) ;=> (2 4 6)

Vector

Konstante Zugriffszeit O(n)! Gut für Indexzugriffe + add hinten.
> (type [1 2 3])

clojure.lang.PersistentVector
> (vector 1 2 3)

erstellt einen Vector aus einer Liste: [1 2 3] Index startet mit 0!
> (vec '(1 2 3 "vier"))

1. => Bekommt nur ein Argument und macht daraus einen Vector: [1 2 3 "vier"]

Lesen im Vector
> (def myvec [1 2 3]) #=> #'user/vec1
> (get myvec 2) => 3 und .get von java.util.List geht auch
> (get myvec 5 42) #=> 42 ist default, falls nicht soviele da
> (nth myvec 1) => 2 # geht zur Not auch
> (myvec 2) => 2 und sogar das geht!

Ein Vektor ist auch Funktion! Er erwartet als Argument die Position des Elementes um diese zurück zuliefern!
> (def myvec ["Stefan" "Tom" "Bob"])
> (myvec 2) # => "Bob"

Schachteln wird aufgelöst:
> [1 2 (+ 40 2)] #=> [1 2 42]

> [1 2 '(+ 40 2)] #=> [1 2 (+ 40 2)]

Zugriff auf geschachtelte Tiefen:
> (get-in ["D" ["Berlin" ["Zehlendorf" "Spandau"] "Hamburg"] "US"] [1 1])

1. => ["Zehlendorf" "Spandau"]

> (get-in ["D" ["Berlin" ["Zehlendorf" "Spandau"] "Hamburg"] "US"] [1 1 0])

1. => "Zehlendorf"

Vektoren sind immutable. Eine Erweiterung liefert einen neuen Vektor!
> (assoc myvec 2 42) #=> [1 2 42] # tausche Wert in Vector aus
> (conj myvec 100) #> [1 2 3 100] # Fügt nur an. myvec bleibt immer gleich!

Genauso geht das Austauschen an einer Stelle:
(assoc-in ["D" ["Berlin" ["Zehlendorf" "Spandau"] "Hamburg"] "US"] [1 1 1] "Dahlem")

1. => New Vec Spandau -> Dahlem

Austauschen an einer Stelle mit einer Funktion statt einem Wert:
(update-in ["Apfel" "Birne" [9 5]] [2 0] inc) #=> ["Apfel" "Birne" [10 5]]

Map

```
(def person {:name "Stefan"
 :city "Berlin"})
;= #'user/person
 (:city person)
```

Map Struktur
Erst Key, dann Value

Keywords sind Funktionen die sich in Collections selbst suchen

=> "Berlin"

Aber Collections sind auch Funktionen, die ein Argument-Key auch in sich suchen. Also geht auch:
(person :city) ; => "Berlin"

Hier unbedingt weiterlesen: Thinking in HashMaps

Seq und Sorted Seq

Sonstige Datenstrukturen

Mit Finger-Trees (bitte Googeln) kann man sich selbst Datenstrukturen bauen oder:

- **double-list** is a sequential collection that provides constant-time access to both the left and right ends.
- **counted-double-list** provides all the features of double-list plus constant-time count and log-n nth.
- **counted-sorted-set** is sorted set that also provides log-n nth

Referenz [double-list is a sequential collection that provides constant-time access to both the left and right ends.

counted-double-list provides all the features of double-list plus constant-time count and log-n nth.

counted-sorted-set is sorted set that also provides log-n nth]

Sequence Lib

References:

- Understanding lazy Sequences:
- Sequence Abstractions:

- (**take n coll**) ; gebe die ersten n der collection zurück

user=> (take 3 [1 2 3 4 5 6 7]) => (1 2 3)
user=> (take 3 '(1 2 3 4 5 6 7)) => (1 2 3)

- (first v) (second v) (last v)

- Differenz von (rest v) (next v)

- (nth myvec 5) ; take the nth element of a vector

- (nthnext)

- (concat)

- (seq coll) ; liefert nil wenn leer. Ideal für if oder if-let

Was ist Sequable?

- Strings, maps und alle Iterables. Und alles was Seq implementiert.

MISC
(cycle ["x" "0"]) ;=> erzeugt eine lazy infinite seq

9.4.4 Functions&Bindings

Hier wird eine Funktion definiert, aber nicht gebunden.

```
(fn mk-set [x y] #{x y}) ; function vector 2 list
#(mk-set [x y] #{x y}) ; anonymous
```

Dies ist z.B. nützlich als Rückgabewert.

Funktionen können wie in anderen Sprachen überladen werden, d.h. verschiedene Signaturen haben:

```
(fn
([x] #{x}) ; one arg
([x y] #{x y}) ; two args
```

Um eine Funktion jezt an einen Namen zu binden müsste man dies ja so tun:

```
(def make-a-set ; long verbose variant to bind a function
(fn
([x] #{x})
([x y] #{x y})
(make-a-set 1) ; => #{1}
```

Kürzer geht es daher aber gleich so:

(defn make-a-set ; Create a **short named binding for a function**

```
"Nimm ein oder zwei args und liefere set zurück"
([x] #{x})
([x y] #{x y})
```

Hinweis

Merke: Den parameter-Vector der Argumente kann man genauso für Destructuring verwenden! D.h. sich beliebig tiefe oder komplexe Dinge herausschneiden. Siehe dazu das Kapitel Destructuring.

Anmerkung

Function Literals können einem oft das Leben einfacher machen (oder schwerer ;-). Dazu ein Beispiel: (fn double [x] (* x x)) ist das gleiche wie (def myfunc2 #(* %1 %1)). Beide Funktionen sind gleich, und können übergeben, zurückgegeben oder mit def zugewiesen werden.

Local Bindings:

Aber auch:
(def x 42) ; wie final

Kann aber im REPL neu gebunden werden um "sich" weiterzuentwickeln. Aber eben am besten auch nur dort.

Advanced Stuff:

- defn- ist ein locales binding! Tatsächlich wird hier nur mit ^:private d.h. ^{:private true} ein Metadata Flag hinzugefügt.

let

Local bindings wie funktionslokale *finals*:

```
(let [r 5
 pi 3.1415
 r-squared (* r r)]
 (println "radius is " r
 (* pi s-squared))
```

Local Bindings
final in Funktion

when-let

Binde den Wert wenn nicht nil ist:
(when-let [a nil] (do (println a)) (+ a 5))
=> nil

Wenn die rechte Seite des Bindings nicht nil ist, dann wird gebunden:
(when-let [a 42] (do (println a)) (+ a 5))
=> 42, 47

Referenzen:

if-let

Wie when let aber mit einem else:
(if-let [a :anything] a "no")
=> :anything
(if-let [a nil] a "no")

=> "no"
user=> (if-let [a false] a "no")
=> "no"

Higher order Functions

Funktionen müssen als Argument und als Rückgabewert übergeben werden können. Hier ein Beispiel für ersteres:
(defn call-twice [f x]
(f x)
(f x))

Hinweis

Wie würde man so eine Abstraktion für jede beliebige Funktion in Java schreiben?

Wichtig

Higher Order Functions können selbst Funktionen als Argumente nehmen, um daraus etwas höherwertigeres zu bauen.

Die am meisten verwendete Higher-Order Funktion ist wahrscheinlich map:
(map inc [1 2 3 4 5])
=> [2 3 4 5 6]

Aber auch mehrere Arg-Vectoren kann man verwenden:
(map + [2 5] [7 15])
(9 20)

9.4.5 Destructuring

Definition

Unter Destructuring versteht man die vereinfachte Selektion von Teildaten aus einer Sequence z.B. in Parametern. Wird beispielsweise a und b übergeben und b ist eine Liste die aus zwei Teilen besteht, dann kann man in den Funktionsparametern schreiben: [a [c d]]. Dann wird b automatisch in c und d zerlegt.

Zuerst ein Beispiel in den Parametern:

```
(defn select-from-map [{myage :age}]
 (println "Alter= " age))
; und jetzt ist im REPL
((select-from-map {:name "Stefan" :age "42"})
=> Alter= 42
```

Destructuring Sequences
Hier als Parameter

Genauso gut geht die bei let mit _ wie das auch in Scala möglich ist:

```
(let [[_,_,c] [7,24,137] ; oder nur ein Collection Symbol!
 c) ; und _ hätte jetzt den Wert 24 !
=> 137
```

Destructuring
Mit Blanks

Dazu ein konkretes Beispiel:

```
(defn retAny []
 '(:Wickie :Ilvie))

(defn callMe []
 (let [[a,b] (retAny)]
 (println "A=" a "; B=" b)))

(callMe)
```

Beispiel
Erweitertes Beispiel

Aufgabe

Stellen Sie sich vor statt c würde am Ende _ returned. Welchen Wert hätte _ dann? (Sie ahnen es bereits,_ kann mit dem letzten Binding verwendet werden!

Wir destrukturieren jetzt mal ein Argument welches eine Liste dynamischer Länge ist. Um z.B. das erste Element abzusplitten und den Rest:
(defn myrest[[x & xrest] ... ;; Wir übergeben [1 2 3 4]
;; jetzt ist x=1 und xrest=[234]

Es gibt noch weitere interessante Möglichkeiten, die hier zu finden sind:

Map Destructuring

Mit Maps und Records geht das genauso:

```
(def mymap {:x 42 :y 137 :z 500})
(let [{width :x length :z} mymap]
 (* width length))
=> 137
```

Map Destructuring
beliebige Elemente entnehmen

Als key kann dabei alles genommen werden. Nicht nur Symbols.

Zahlen können dabei als Index verwendet werden.
let [a 5 b 10] seq

entnimmt das fünfte und zehnte Element der Sequenz und bindet sie an *a* und *b*.

Und natürlich kann das ganze

- beliebig geschachtelt werden.
- Wieder mit dem Wildcard _ versehen werden.
- as name

  bindet alles an den Namen [x y as alles].

Falls Werte nicht vorhanden sind, können Default-Werte gesetzt werden:
(let [{p :unbekannt :or {p 42}} m] p)

Hier wird p also zuerst nil und dann auf 42 gesetzt.

Mit dem

keys keyword

können die values aus der Map an in let automatisch verfügbare=passende keys gebunden werden:

(let [{:keys [x y z]} mymap]
(format "%s ist die Informatikerzahl und %s ist die Physikerzahl" x y))

Dies lässt sich auch nach dem Vararg Symbol anwenden, um die Restsequenz auseinanderzunehmen.

Noch ein paar zusammenfassende Beispiele von Michael Fogus (C):

```
(defn destr-seq [[a b & more]] (str "Got " a ", " b ", and
 " more))
(destr-seq [1 2])
;=> "Got 1, 2, and "

(destr-seq [1 2 3 4])
;=> "Got 1, 2, and (3 4)"

;; associative desctructuring
(let [{first-thing 0 last-thing 3} '[a b c d]]
 [first-thing last-thing])
;=> [a d]

;; map destructuring
(let [{p1 :player1 p2 :player2} {:player1 "Mike", :player2
 "Chris"}]
 [p1 :vs p2])
;=> ["Mike" :vs "Chris"]
```

Destructuring

9.4.6 Namespaces

Referenzen:

- All you need to know abot Namespaces: (link lost...)
- Colin Jones, ***"Clojure Libs and Namespaces: require, use, import, and ns"***,

Namespaces sind komplett analog zu Modulen in Ruby / Python oder Packages in Java. Sie werden tatsächlich zu Java Packages kompiliert. Ein Typ Company im Namespace *de.units* wird eine Java Klasse *de.units.Company* erzeugen.

- Der REPL startet im default ***user*** Namespace.

1. Der aktuelle Namespace ist immer an *ns* gebunden
2. **(ns edlich)** schaltet auf einen neuen NSpace um. Im Code und im REPL.

ns ist offensichtlich ein Makro mit Seiteneffekt!

Hinweis

Da java.lang per default importiert ist, sind alle dortigen Symbole sofort vorhanden: String, Integer, List, Socket, ... Das gleich gilt aber auch für alle clojure.core Symbole wie z.B. filter.

Import

Innerhalb eines Makros (:require [project.utils :as utils])

9.4.7 Program-Flow

if

```
(if (= 1 2)
"Math is broken!"
"Math still works!")
```

Wahr ist alles außer nil oder false.

Wenn es kein else gibt liefert falsch nil zurück.

when

```
(when (= 1 1) true)
```

Ideal wenn etwas evaluiert werden soll oder nichts getan werden soll und nil zurückgeliefert werden soll.

if-let / when-let

do

```
do ; returns it's last expression
(println "Hallo!" ; side effect
(apply + [ 2 4 4])) ; prints and returns 10
```

defn, let, fn, loop und try enthalten ein implizites do!

let

(let [x 1 ;local binding that can't change outside the vector scope
y x] ; en implicit do follows with n expressions
y) -> 1

Zwei Beispiele aus Emrick:

```
(defn hypot
 [x y]
 (let [x2 (* x x)
 y2 (* y y)]
 (Math/sqrt (+ x2 y2))))

(let [location (get-lat-long)
 _ (println "Current location:" location)
 location (find-city-name location)]
 …display city name for current location in UI…)
```

Let usage
Bindings in Clojure

Wir lernen:

1. Wir können Daten in let mehrmals rebinden. Hier erst long und lat an location und dann den richtigen Stadtnamen an location.
2. Das Zwischenergebis geben wir aus und binden das result an nix ="_", da uns der Returnwert von println ja nicht interessiert.

(let [[x y z] myvec] ; wird destructured gebunden
(working...)

Ist dasselbe wie:
(let [x (nth v 0)
y (nth v 1)
z (nth v 2)]

In Python heißt das unpacking. Ruby kann das auch.

Dies geht auch *nested*:
(let [[x _ _ [y z] v] ; wird visuell so gebunden
42 "foo" 99.2 [5 12]]; d.h. (+ x y z) ist 59

Weiterhin geht
[[x & rest] v] ; also x = erstes element und rest ist der Rest

und
[[x _ z :as original-vector] v] ; dann ist original-vector = v

case
(case (+ 2 3)
6 "error"
5 "ok")

cond
(cond
(< n 0) "negative"
(> n 0) "positive"
:else "zero")

Loop and Recur

Der Witz an recur ist, es verbraucht keinen Stack Space:
(loop [x 5] ;; statt loop später defn countdown
(println x ";") ; no side effects please...
(if (neg? x)
x
(recur (dec x))))

Weiterhin hat es initiales Vectorbinding ([X 5]). Dies dient dann wieder dazu, die Argumente "im Kreis" zu reichen.

Mit diesem Binding sieht es fast wie eine Funktion aus - und genauso geht es auch mit Funktionen.

Wichtig

Oftmals braucht man loop und recur aber gar nicht. Wenn loop einfacher ist, dann geht auch **doseq** oder **dotimes**. Oder man will sowieso eine Sequence bearbeiten, dann sind map, reduce, sort, filter und das for* (also die List comprehension) noch besser geeignet. Also immer zweimal überlegen!

Übung FPR-03: doseq und dotimes

Welches Ergebnis liefert der erste Code?

(doseq [x [1 2 3]
y [1 2 3]]
(prn (* x y)))

Welches Ergebnis liefert der zweite Code?
(dotimes [n 5] (println "n is" n))

© Code von clojuredocs-org

Bearbeitungsdauer: 5 Minuten

9.4.8 Java Interoperabilität

Operation	Java Code	Sugared Interop
Object Instantiation	new java.util.Array List(100)	(java.util.ArrayList.100)
Static method invocation	Math.pow(2, 10)	(Math/pow 2 10)
Instance method invocation	"hello".sub string(1, 3)	(.substring "hello" 1 3)
Static field access	Integer.MAX_VALUE	Integer/MAX_VALUE
Instance field access	someObject.some Field	(.someField someobject)

Java Interop
Java aus Clojure © Emrick et al

Code

```
(new java.lang.Date) ; create a Java Object
(java.lang.Date.) ; equal and easier to write with the "."
at the end

(.getTime (java.util.Date.)) ; call a method of the object
(.nextInt (java.util.Random.) 10) ; yields 0, 0, 2, 3 in range
of [0,10[

 (.exists (java.io.File. "data.txt")) ;; needs (import
'java.io.File)
```

```
(java.lang.Math/PI) ; => 3.141592653589793 call static Methods
(java.lang.StrictMath/pow 2 5)

(Thread/sleep 100)
```

Mit set! erlaubt Clojure auch das direkte setzen von Attributen!

9.4.9 Metadata

Metadaten sind Daten über Daten. In vielen Programmiersprachen gibt es diese bereits. In Java kennt man diese als Annotations z.B. mit @test. Leider sind Metadaten in den meisten Sprachen nachträglich hinzugefügt worden und nicht *first-class-citizens*.

In Clojure dagegen sind Medataten perfekt von Anfang an integraler Bestandteil der Sprache gewesen. Ideal wäre es doch z.B. auch in Java gewesen, wenn man ein Objekt mit einem Objekt annotieren könnte?! Genau das geht quasi auch in Clojure. Wir geben Metadataten in Clojure in Form von einer Hashmap an. D.h. ein Keyword (z.B. :meinkeyword) und einem beliebigen Value. Dieser Value kann auch eine Funktion oder sogar ein Typ sein. Es ist offensichtlich, dass dies dann viel leichter zu bearbeiten und auszulesen ist, als in einigen anderen Sprachen.

Wichtig ist, dass es

1. normale Metadaten für Daten, Funktionen, ec. gibt und
2. reader Metadaten für den Compiler (typen, Zeile, File, etc.)

Beide werden unterschiedlich notiert!

Definition:

Code

```
^{:key1 val1 :key2 val2 ...}
(meta object) ; get the Meta-Information
```

Typische Use-Cases:

- Doc-Strings (die ja auch nach den Parametern direkt angegeben werden können. Beides ist äquivalent!)
- Für *type* und *return hints* (=Hinweise). Dies ermöglicht dem Runtime Compiler zu optimieren und den Typ nicht zu reflekten.

- Interne Compilerhinweise wie die Arglist, die Zeile einer var, ob eine var eine Referenz auf ein Macro ist. Dies wird aber meistens nicht vom Developer angefasst.

Beispiele:

Code

```
(with-meta [1 2 3] {:mykey "aValue"}) ;; => [1 2 3] direkt
den Vector taggen
(meta (with-meta [1 2 3] {:mykey "aValue"})) ;; => {:mykey
"aValue"}

(def wm (with-meta [1 2 3] {:mykey "aValue"}))
(meta wm) ;; => {:mykey "aValue"}
```

Jetzt Funktionen mit Metadaten versehen:

9.4.10 I/O

Einlesen einer kompletten Textdatei:
(**slurp** "text.txt")

Schreiben in eine Textdatei:
(**spit** "blubber.txt" "test")

Siee oben mit append modus:
(**spit** "event.log" "test 1\n" :append true)

Weitere IO Funktionen:
read-line => ;; einlesen von der Konsole

9.4.11 State-Management

Concurrency

Vars

Vars sind nur Referenzen im Namespace. Eher für Konstanten gedacht!

```
(def x 5)
(var x) wenn nur die Referenz interessant ist!
```

```
(def ^:private x 42) ;; Makro für Metadata, wie defn-
(def a "Beispielwert" 42) ;; Mit docstring
```

Es gibt mehrere special Metadatas wie :private, :doc, :test, :tag

```
(def ^:const endgueltig 42) ;; nicht rebindbar!
```

Dies wird nicht zur Laufzeit sondern in den Code compiliert. Ist also viel schneller.

```
(def ^:dynamic *maxval* 1024) ;; * * ist eine namenskonvention
(earmuffs)
```

Dynamisch meint hier, dass es thread lokal überschrieben werden kann.

```
(binding [*maxval* 2048]) ;; dynamisches überschreiben
(alter-var-root #'x inc)
```

Atoms

Für synchrone und unkoordinierte Zugriffe. synchron = threads blockieren unkoordiniert = wir haben in threads separierte Kontexte.

```
(def sarah (atom {:name "Sarah" :age 25 :wears-glasses?
 false}))
;= #'user/sarah
(swap! sarah update-in [:age] + 3)
;= {:age 28, :wears-glasses? false, :name "Sarah"}
@sarah ;; Den Inhalt dereferenzieren
;= {:age 28, :wears-glasses? false, :name "Sarah"}
```

Definition:

```
(atom wert)
(def aktenkoffer (atom wert)) ; Besser gleich binden
```

Wert setzen:

```
(reset! atom wert)
```

Diese Form wird nur motiviert, wenn der alte Wert vergessen werden kann.

Viel besser ist doch, den Wert über einen Funktionsaufruf zu übergeben!

```
(swap! atom update-fn args*) ; update-fn sollte pure sein =
keine Nebeneffekte!
```

Hier wird der Wert des Atoms quasi als erstes Argument übergeben. Wenn man also z.B. nthrest verwendet, bekommt man ein Argument mehr (zuviel!).

9.4.12 Types and Protocols

References:

- Learn Clojure using records and protocols, Jan Kronquist:

9.4.13 Multimethods

Links:

- Amit Rathore, ***"Multimethods in Clojure"***,

9.4.14 Macros

References:

- Example
- My first Macro
- When do you need Macros?
- Code is Data, Data is Code

9.4.15 API

Ausgewählte Eigenschaften der API. Am besten hier zu studieren: http://clojure.org/cheatsheet

Random Numbers

rand selbst gibt ein Double, im Bereich zwischen 0 und der angegebenen Zahl, zurück:
(**rand** 10) => 9.534711173605547

rand-int gibt im gleichen Bereich ein Integer aus:
(**rand-int** 10) => 9

Strings

Splitted einen String:
(**clojure.string/split** "q1w2e3r4t5y6u7i8o9p0" #"\d+" 5)
=> (clojure.string/split "q1w2e3r4t5y6u7i8o9p0" #"\d+" 5)

Konkateniert eine Liste zu einem String:
(apply str ["abc" "def" "geh"]) => "abcdefgeh"

Conversion

Konvertierung von Listen untereinander:
(into () ["first" "second"]) => ("second" "first");

Hier einen Vector in eine Liste.

Mehr zur Konvertierung von Grundtypen im ersten Kapitel Basics (12.4.1)

Collections
(map * [2 5 7][3 6 9]) => (6 30 63)
(reduce max [-5 0 10 4]) => 10 ist genau wie
(max (max (max -5 0) 10) 4) // ist genau das gleiche expandiert geschrieben
(reduce - 100 [12 4 17 42]) => mit einem initialen Seed. Brauch man oft.

Aufgabe

Was ergibt das letzte Beispiel?

Max ist wie viele Operatoren (+,*, etc.) variadic. D.h. (max -5 0 10 4) geht.
(apply + [1 2 3]) ; ist (+ 1 2 3)

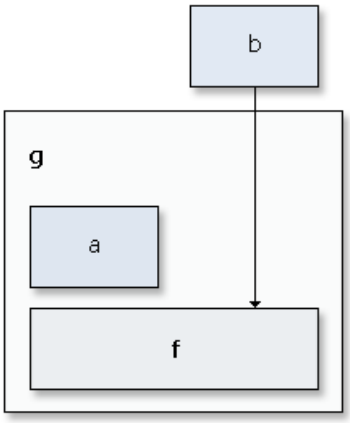
Higher-Order Konstrukte

Apply wendet eine Funktion auf alle Argumente an:
(apply hash-map [:bauer 3 :turm 5 :dame 9]) => {:bauer 3, :dame 9, :turm 5}

Partial (in Scala auch Currying genannt) ist komplexer:
Hier übergibt man nur ein paar Argumente (und nicht alle). Zurück erhält man dann quasi eine gefrorene Funktion, die als Argumente die letzten noch fehlenden zurückliefert. Dies ist ideal für Funktionen, die erst parametrisiert (z.B. Connections oder Filter) und dann später mit verschiedenen Argumenten aufgerufen werden.

Beispiele:
(def add-the-answer (partial + 42))
(add-the-answer 10) => 52
(defn old? [x] (if (> x 60) x))
(def only-oldies (partial filter old?))
(only-oldies '(42 7 66 45 81 77)) => (66 81 77)

(map (partial + 2) [1 2 3]) => (3 4 5)



Partiell angewendete Funktionen

Genauso einfach wie das letzte Beispiel kann aber auch ein reader-Makro verwendet werden:
(map (partial + 2) [1 2 3]) => (3 4 5) ;; mit Partial
(map #(+ 2 %) [1 2 3]) ;; als reader macro / function literal
(map (fn [x] (+ 2 x)) [1 2 3]) ;; die Variante mit anonymer Funktion

Das Funktions-Literal

1. (+ 2 %)

ist gut lesbar, hat aber Probleme mit der variablen Anzahl von Argumenten (siehe Emrick, 67ff).

Hinweis

Aufrufe mit apply, partial und comp können bei langen Argumentlisten (> 3-4) ein wenig langsamer werden (dennoch viel schneller als in Ruby oder Python).

comp ist die Komposition (Vermischung) von mehreren Funktionen zu einer Funktion. Beispiel:
(map (comp - *) [2 4 6] [1 2 3]) ; (-2 -8 -18)

comp etabliert also so etwas wie eine Pipeline. Bedingung: Jedes Argument in der Pipeline muss wieder passen.

9.4.16 MISC

Structs / Objekte

Makro für eine 'enforcte' Map:
(defstruct book :title :author)
(def b (struct book "Programming Clojure" "Stuart Halloway"))
b => {:title "Programming Clojure" :author "Stuart Halloway"}

Wie Konstruktor. Weniger ja. Mehr nein.

Variabler mit struct-map:
user=> (struct-map book :title "Anathem" :farbe "Grün")
{:title "Anathem", :author nil, :farbe "Gr?n"}

Pre und Post Conditions

Nach den Parametern kann eine Map kommen, die :pre und :post enthält. Der Value der beiden ist ein Vector mit einer beliebigen Anzahl von Tests. Die :pre Tests können auf die Parameter zugreifen. Die :post tests können mit % den Return-Value erfassen.

```
(defn constrained-fn [f x]
 {:pre [(pos? x)] ;; first conditions then function comment!
 :post [(= % (* 2 x))]}
 "Comment"
 (f x))

(constrained-fn #(* 2 %) 2) ; Übergebe '''anonyme''' Funktion!
;=> 4
(constrained-fn #(float (* 2 %)) 2)
;=> 4.0
(constrained-fn #(* 3 %) 2)
;=> java.lang.Exception: Assert failed: (= % (* 2 x))
}
```

Funktions Assertions mit Pre und Post
(C) Emrick, Carter, Grand

Man beachte: Die Funktionsbeschreibung muss nach den pre und post Conditions kommen! Sonst kommt eine Fehlermeldung: Symbol % not found!

Mehr über Contract Programming hier:

1. Es geht auch ausserhalb:

Try / Catch

1. Geht wie oben auch ausserhalb:

Documentation / Doc

1. Man kann Codox nutzen.

Varargs

Varargs gehen im let Binding mit dem & Symbol wie in vielen Sprachen:
(let [[firstname lastname & restinfo] infosequence

Und in einer Funktion:
(defn myfunc [a & mehr]
(println "First arg=" a "; Restargs= " mehr))

Ergibt wiederrum:
(myfunc :42 "Stringy" 137 "putty")
First arg= :42 ; Restargs= (Stringy 137 putty)
nil

Regular Expressions

References

- Mark Needham Examples
- Fatvat Blog
- Regular Expressions in Java
- re-pattern
- langref.org
- Good Blog Examples:

Testen

- Artikel:
- core.test mit (is...
- midje mit (fact (1 => 1))

Performance

- visualVM
- Verwende Maps (fast lookup)
- verwende native types
- generic Collections erstellen

- Java Arrays
- Evtl. transients

Was an Clojure am Anfang schwierig ist

- Es gibt noch keine IDE in Java Qualität - auch wenn Counterclockwise (ein Clojure Eclipse-Plugin) geht. So sollte z.B. die ungerade Anzahl von Klammern angezeigt werden.
- Das Debugging ist nicht leicht zu installieren (aber theoretisch leichter als in mutablen Sprachen).
- Leiningen lacks abstraction / language power
- Bei einer Liste wird hinten und bei Vector wird vorne geadded. Weil Liste wirklich linked liste und Vector ist Tree! Siehe Datenstrukturen!
- Viele Operationen auf Typen (z.B. Vector) liefern etwas anderes zurück, z.B. liefert map auf Vector eine Lazy Sequence. Es steckt ein Sinn dahinter, der aber nicht einfach zu verstehen ist. Man soll also entweder in der Clooection world bleiben (z.B. pop) oder in der Sequence World (drop-last)!
- Zudem sind Namespaces, Metadata oder generell Higher Order Functions - Kompositionen als Konzept zum Beginn nicht einfach verständlich. Referenz
- Die vielen Zaubersymbole (!)

Daher hier noch gute Rules:

Klammern nicht! Die veschwinden:

9.4.17 Weiterführendes

Typische Fehler

1. Schließende Klammer vergessen.
2. Can not execute iFN: Es wurde versucht eine Liste zu tief auszuwerten, die als erstes Element aber nur Grundtypen (Zahl, String) hat.
3. def statt defn geschrieben. Dann kommt natürlich wrong args error
4. Hier soll getestet werden, ob der String in seiner Länge größer als 3 ist. Ist das so korrekt? (> 3 (count "g1f3 d7d5 c2c4"))
5. Pre und Post Conditions kommen VOR der Funktionsbeschreibung!
6. Keine "-" oder "_" im Namespace! require überprüfen
7. as statt :as im :require geschrieben
8. empty statt empty? genommen

Zusätzliche Literatur

- Lisp in 3 Minutes
- Neil Ford, ***Why functional Programming is on the rise***
- Write less code
- Getting started with Clojure on Mac
- Stefan Tilkov, "Clojure - A pragmatic Lisp dialect for the JVM",
- Luke VanderHart, Ryan Neufeld, uvm., "Clojure Cookbook", ein OpenSource-Cookbook, geschrieben mit AsciiDoc und gehostet auf Github,

Advanced Stuff

- Core Async Video
- Actor Library
- John Purdy, "Why concatenative Programming Matters!"
- Carin Meier, "Why is a Monad Like a Writing Desk?",
- Domain Driven Design

Useful Stuff

- Übersicht Libs:
- Wie **Refactore** ich Clean Clojure Code!!
- pprint => Prittyprint statt print verwenden
- time für Performanceanalyse
- Kibit @ github für statische Codeanalyse
- Test Coverage
- Noch mehr Marcos wie -> und ->>

Clojure Web-Stack

- The Clojure Web-Stack
- Luminus Web-Framework
- Pedestrial Web-Framework

Parser

- Parsing with Parsley
- Kern, parser combinators
- Growing a DSL with Clojure
- Persatron
- Instaparse (!)

- squarepeg

Reducers

- Stefan K on Heise

Tools

- Bouncer Validation Lib
- Clojure in the Browser

Weitere Videos

- Stuart Sierra, Functional Design Patters

Incanter

- Exploring data with Clojure, Incanter, and Leiningen

9.4.18 Stdlib

Function Definition

- defn ;; Destructuring, multiple arities, type hinting, :pre + :post, metadata annotations, defn- scope to local use etc.
- fn ;; ((fn [x] (* (Math/random) x)) 100) oder (#(* (Math/random) %) 100) mit "#" !
- (identity 4) ;; => 4
- (fn* ...) ;; rarely used for macros with ^:once hint

9.5 Hybridsprache: Scala

Freie Literatur

- Martin Odersky, "Programming in Scala" First Edition,
- Wampler, Payne, "Programming Scala",
- Daniel Spiewak, "Scala for Java Refugees",
- Ted Neward, "Scala Tutorials for Java Programmers", IBM Developer Works,
- Schinz, Haller, "A Scala Tutorial",

Literatur & Videos

- Joshua Suereth D., "Scala in Depth", Manning 2012
- Cay Horstmann, "Scala for the Impatient", Addison Wesley, 2012

- Wampler, Payne, "Programming Scala", OReilly, 2009
- Kurzintro:

Wie Scala lernen?

- Graham's Guide to Learning Scala:

Konzepte

Die Grundidee von Scala ist, in einer Sprache die Entwicklung nach dem funktionalen- und nach dem objektorientierten Modell zu ermöglichen.

Hinweis

Author ist Martin Odersky, der schon bei Nikolaus Wirth an Modula 2 und Oberon mitgearbeitet hat. Später arbeitete er an Pizza und insbesondere an Generics in Java. 2001 begann die Arbeit an Scala, die dann 2004 veröffentlicht wurde. Inzwischen wird Scala vom European Reseach Council als einige der wenigen Sprachen im europäischen Raum gefördert.

M. Odersky

Grundlegende Konzepte sind:

- Statt dem Erstellen einer Klasse, wird in Scala mittels *object* ein Objekt erstellt, welches ein **Singleton** ist.
- Die meisten Datenstrukturen in Scala gibt es sowohl als **mutable** aber auch als **immutable**.
- Scala ist **funktional**, das heißt insbesondere, dass Funktionen Objekte erster Klasse sind. Sie können sowohl Parameter als auch Rückgabewerte von Funktionen sein. **Closures** können auf Variablen ihres Kontextes zugreifen.
- Scala ist **objektorientiert**. Sogar primitive Typen sind ein Objekt.

- Erweiterung von Objekten geschieht über **Traits** als **Mixin** Konzept. Dabei wird Funktionalität, aber nicht die Daten vererbt. Es gibt daher keine Mehrfachverebung im herkömmlichen Sinne.
- Scala unterstützt Nebenläufigkeit mittels **Aktoren**, wie sie bereits in Erlang eingeführt wurden.
- **Pattern Matching** als zentrales Konzept für Polymorphie und die Vermeidung von switch Konstrukten.

Vorteile:

1. Scala integriert gut mit Java
2. Scala hat durch die Community und Förderung vielleicht das stärkste Momentum als leistungsfähigere Alternative zu Java
3. Der Autor von Groovy wird oft mit den Worten zitiert: "Hätte ich früher von Scala gewußt, hätte ich Groovy nicht entwickelt"
4. Die Aktoren sind leistungsfähig
5. Scala ist schnell.

Nachteile:

1. Das Typsystem wird öfters als sehr kompliziert bezeichnet (Kovarianz und Kontravarianz). Viele argumentieren allerdings, dass man nicht zwangsläufig mit komplexen Typen arbeiten muss. Scala ist statisch getypt.
2. Scala macht nicht eines richtig, sondern ist eher ein *Mischmasch*. Es fehlt die Radikalität von Clojure und Haskell.

Wie hier beschrieben gilt folgendes:

- Scala hat eine hohe Expressiveness. Dies bedeutet weniger Codezeilen
- Scala ist insbesondere dann schnell, wenn beispielsweise optimiert werden kann (wie in dem Paper die Loops). In manchen Fällen ist Scala dann sogar schneller als Java.

Scala in Depth

```
object HelloWorld {
 def main(args: Array[String]) {
 println("Hello, world!")
 }
```

```
}
```

Basic Function
HelloWorld.scala

HelloWorld.scala -> scalac -> HelloWorld.class und mehr...

Filename oder DirStruct egal
> scala -classpath . HelloWorld

Was passiert hier?

- Scala kennt keine statischen Methoden. Aber dafür Singleton-Objekte die sich genauso anfühlen.
  - Haben Klasse und Objekt denselben Namen so nennt man sie in Scala „companions“
- Arrays sind Collections mit dem generic Type in eckiger Klammer.
  - Generics gehen sogar wenn Java nur 1.4 ist!
- scala.Predef wird immer statisch importiert.
  - cala und java.lang sind quasi mit drin.

Alles ist ein Objekt

- 1+2*3/x besteht komplett aus Methodenaufrufen: 1.+(2.*(3./(x)))!
- Funktionen sind ebenfalls Objekte und können in Variablen gespeichert werden.

Anonyme Funktionen:

```
object TimerAnonymous {
 def oncePerSecond(callback: () => unit) {
 while (true) { callback(); Thread sleep 1000 }
 }
 def main(args: Array[String]) {
 oncePerSecond(() => println("time flies like an arrow..."))
 }
}
```

Basic Function
Anonyme Funktionen

Klassen

class Complex(real: double, imaginary: double) {
def re() = real

def im() = imaginary
}

- Zwei Accessoren
- Return Type nicht nötig: *inferred automatically*

class Complex(real: double, imaginary: double) {
def re = real // Methoden ohne Args
def im = imaginary // Methoden ohne Args
}

Java Interaktion

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
 def main(args: Array[String]) {
  val now = new Date
  val df = getDateInstance(LONG, Locale.FRANCE)
  println(df format now)
 }
}
```

Basic Function
Java Interaktion

Ein zweites Beispiel:

```
package SwingDemo

import javax.swing.{JFrame, JLabel}

object Main extends Application {
 def getTitle() = "Scala can Swing"
```

```
 val frm = new JFrame(getTitle)
 frm.getContentPane.add(new JLabel("Hello World"))
 frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
 frm.pack
 frm.setVisible true
}
```

Basic Function
Java Interaktion

Currying

Definition

„Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments."
Quelle:

Dazu ein Beispiel:

```
object CurryTest extends Application {

 def filter(xs: List[Int], p: Int => Boolean): List[Int] =
 if (xs.isEmpty) xs
 else if (p(xs.head)) xs.head :: filter(xs.tail, p)
 else filter(xs.tail, p)

 def modN(n: Int)(x: Int) = ((x % n) == 0)

 val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
 println(filter(nums, modN(2)))
 println(filter(nums, modN(3)))
} // Ergebnis = ?
```

Basic Function
Currying in Scala

Aufgabe

Was ist das Ergebnis dieser Funktion?

Hinweis: modN liefert eine Funktion Int => Boolean

Nested Functions

Zitat: "Note that the nested function process refers to variable threshold defined in the outer scope as a parameter value of filter." Quelle:

```
object FilterTest extends Application {
 def filter(xs: List[Int], threshold: Int) = {
 def process(ys: List[Int]): List[Int] =
 if (ys.isEmpty) ys
 else if (ys.head < threshold) ys.head :: process(ys.tail)
 else process(ys.tail)
 process(xs)
 }
 println(filter(List(1, 9, 2, 8, 3, 7, 4), 5))
} // Ergebnis = ?
```

Basic Function
***** in Scala**

Funktionen höherer Ordnung

Der nachfolgende Code wurde der folgenden Quelle entnommen:

```
class Decorator(left: String, right: String) {
def layout[A](x: A) = left + x.toString() + right
}
```

und

```
object FunTest extends Application {
def apply(f: Int => String, v: Int) = f(v)
val decorator = new Decorator("[", "]")
println(apply(decorator.layout, 7)) // Ergibt was?
}
```

Wie bereits erwähnt können Funktionen als Parameter oder Rückgabewerte verwendet werden. Die Methode decorator.layout wird hier automatisch auf den Typ Int => String definiert. decorator.layout ist daher eine Polymorphe Methode, die der Scala Compiler in eine Methode mit richtiger Signatur formen muss. Initial kann daher von der Signatur von Methoden abstrahiert werden.

Polymorphe Methoden

```
object PolyTest extends Application {
 def dup[T](x: T, n: Int): List[T] =
 if (n == 0) Nil
 else x :: dup(x, n - 1)
 println(dup[Int](3, 4))
 println(dup("three", 3))
}
```

Basic Function
Polymorphie in Scala

Methoden in Scala können sowohl mit Werten, als auch mit Typen parametrisiert werden. Werte in runden und Typen in eckigen Klammern. Der Typ kann hier also auch später angegeben und so *infered* werden, das heißt aus dem Kontext abgeleitet.

9.6 Kurzübersicht: Haskell & Erlang & Co

Haskell

Literatur:

- Learn Haskell Fast and Hard

9.7 Zusammenfassung FPR

- Sie kennen nun verschiedene Paradigmen der Programmierung und deren Vor- und Nachteile.
- An Hand von Clojure wurde ein Beispiel der funktionalen Programmierung genauer analysiert und vorgestellt.
    - Sie wissen nun wie Sie eine Clojure-Datei erstellen und ausführen.

  - Sie kennen die wichtigsten Konzepte von funktionalen Sprachen
- Mit Scala kennen Sie nun eine Hybridsprache, die es ermöglicht funktional und objektorientiert zu programmieren.
- Als deklarative Programmiersprache wurde Prolog vorgestellt. Sie konnten mittels Eingaben und Aufrufen die Funktionsweise genauer kennenlernen.

9.8 Wissenüberprüfung FPR

Übung FPR-04: Clojure

In den ersten Seiten dieses Kapitels wurde darauf hingewiesen, dass unter anderem Clojure auf ein radikales Konzept setzt.

1. Welche Konzepte von Clojure sind „radikal“'?
2. Was sind Closures?
3. Was sind „pure functions“?
4. Was ist „immutability“?
5. Was versteht man unter „currying“?
6. Was sind „Agents“?
7. (Was sind „Continuations“?)

Bearbeitungsdauer: 30 Minuten

9.9 Literatur FPR

- Bruce A. Tate, **"Seven Languages in Seven Weeks"**, Pragmatic Programmers, 2010, ISBN-13: 978-1934356593
- Shriram Krishnamurthi, Programming Languages - Application and Interpretation
- Felleisen et al., "How to Design Programs - An Introduction to Programming and Computing"
- Paul Graham, **"On Lisp"**, Prentice Hall, Free Download:

Artikel:

- Ben Moseley, Peter Marks, **"Out of the Tar Pit"**, 6 Feb 2006, SPA 2006 Functional Relational Programming

Links:

- Joe Armstrong , **"Why OO Sucks"**

- Dean Wampler, Slides concerning functional Programming,
- Christophe Grand, **"Logic programming in Clojure: core.logic.introm"**,
- David Nolen, **"Sudoku in 33 lines with logic programming"**,
- Functional programming books review

Videos:

Steward Halloway, **"Evident Code at Scale"**, Euroclojure 12,

10 LPR - Logische Programmierung

Autor: Prof. Dr. Stefan Edlich

10 LPR - Logische Programmierung
10.1 Überblick und Lernziele LPR
10.2 Logik
10.3 Prolog
10.4 Core.logic
10.5 Zusammenfassung LPR

10.1 Überblick und Lernziele LPR

Lernziele

In diesem Modul ist es sehr wichtig, dass die drei großen Paradigmen gleichwertig wahrgenommen werden: Objektorientierte-, Funktionale- und Logische Programmierung. Es gibt sehr viele Gründe, warum funktionale Programmierung ohne Status = Variablen sinnvoll ist. Es gibt aber genauso viele Problemstellungen, in denen weder Status noch Ablauf sinnvoll ist. In diesem Fall kann das Problem einfach durch Regeln, d.h. mit logischer Programmierung beschrieben werden. Sudoku ist hier das einfachste Beispiel, welches in 10-20 weniger Codezeilen mit logischen Ausdrücken beschrieben werden kann. Ziel dieser LE ist es also, das Bild abzurunden, damit wir als Top-Developer immer das richtige Werkzeug zur Hand haben.

Gliederung

Gliederung

Initial wird etwas Mathematik = Logik zur Auffrischung wiedergegeben. Es folgt die Königin der logischen Programmiersprachen: **Prolog**. Danach schauen wir uns **core.logic** an. Ein berühmtes Beispiel aus der Scheme / Clojure Welt, welches es einem ermöglicht, Beziehungen und Regeln mittels Funktionen auszudrücken. Dies ermöglicht es dann, die logische Programmierung komplett in der funktionalen Programmierung zu nutzen.

Zeitumfang

Zeitumfang

Zum Durcharbeiten der Lerneinheit benötigen Sie ca. 60 Minuten und für die Bearbeitung der Übungen ca. 120 Minuten.

10.2 Logik

Referenzen:

- Dirk Hoffmann, "Theoretische Informatik", Hanser Verlag, ISBN-13: 978-3446426399 (ein hervorragendes Buch, das in diesem Kapitel häufig Anwendung finden wird).

Im Bereich der Logik geht es vor allem um zwei Dinge:

- Beschreiben von Fakten mit einer logischen Auszeichnungssprache
- Schlussfolgern alias Deduktion

Noch Anfang des neunzehnten Jahrhunderts hatte man die Hoffnung, alles mit Hilfe von Regeln, Axiomen und Gesetzen beschreiben und so die Welt erklären zu können. Auch die ersten Grundansätze der logischen Programmierung wurden hier gelegt. Leider zeigte sich mit Kurt Gödels Unvollständigkeitssatz, dass dies nicht komplett möglich ist. Und so verschwand gleichzeitig auch die logische Programmierung zugunsten der von Neumann orientierten Programmierung, bei der Speicherzellen manipuliert wurden.

Erst später mit Prolog erlebte die Logik und insbesondere die logische Programmierung wieder einen starken Aufwind und bildet heute die Basis für viele Bereiche:

- den Hardwareentwurf
- Logisches schließen aus Wissensbasen
- Textanalyse und Machine Learning

Syntax der Aussagenlogik

Basisbausteine sind Formeln:

- 0 und 1 (alias falsch und wahr) sind Formeln
- Alle Variablen sind Formeln
- Formeln, die über aussagenlogische Symbole verknüpft sind, sind auch Formeln.

Basisbausteine sind:

- $\neg A$ Negation
- $A \wedge B$ logisches und
- $A \vee B$ logisches oder
- $A \rightarrow B$ Implikation

Dazu gehören weiterhin die Äquivalenz $\leftrightarrow$ und die $\not\leftrightarrow$ und die Antivalenz (XOR!).

Anmerkung

Formeln, die nicht weiter zerlegt werden können beschreiben wir als atomar.

Definition

Üblicherweise bindet die Negation $\neg$ stärker als das logische und $\wedge$. Das logische or $\vee$ bindet wiederum schwächer als die Äquivalenz und Antivalenz $\rightarrow$ bzw. $\not\leftrightarrow$.

Es folgen noch einige grundlegende Regeln:

Ausdruck	Bedeutung
$\neg A \wedge B$	$((\neg A) \wedge B)$
$A \vee B \wedge C$	$(Alor(B \wedge C))$
$A \vee B \rightarrow C$	$((A \vee B) \rightarrow C)$
$A \rightarrow B \nLeftrightarrow C$	$((A \rightarrow B) \nLeftrightarrow C)$
$A \nLeftrightarrow B \rightarrow C$	$((A \nLeftrightarrow B) \rightarrow C)$

Bindungsregeln

Ausdruck	**Bedeutung**
$\neg\neg\neg A$	$\neg(\neg(\neg A))$
$A \wedge B \wedge C$	$(A \wedge B) \wedge C$
$A \vee B \vee C$	$(A \vee B) \vee C$
$A \rightarrow B \rightarrow C$	$(A \rightarrow B) \rightarrow C$
$A \leftrightarrow B \leftrightarrow C$	$(A \leftrightarrow B) \leftrightarrow C$
$A \nLeftrightarrow B \nLeftrightarrow C$	$(A \nLeftrightarrow B) \nLeftrightarrow C$

Kettenregeln

10.3 Prolog

Referenzen:

- A Logic Programming Reading List:
- Sterling, Shapiro, ***The Art of Prolog***
- Friedman, Byrd, Kiselyov, ***The Reasoned Scheamer***
- ISO Standard

Systeme:

- GNU Prolog
- SWI Prolog
- SICTUS Prolog
- Online Interpreter!

Refexion

Bruce Tate in : "You describe the **characteristics of cakes** that you like and let the **baker** pick the ingredients and **bake the cake for you**, based on the rules you provided. With Prolog, you don't have to know how. The computer does the reasoning for you. With a casual flip through the Internet, you can find examples to **solve a Sudoku with fewer than twenty lines of code**, **crack Rubik's Cube**, and solve famous puzzles such as the **Tower of Hanoi** (around a dozen lines of code). Prolog was one of the first successful logic programming languages. You make assertions with pure logic, and Prolog determines whether they are true. You can leave gaps in your assertions, and Prolog will try to fill in the holes that would make your incomplete facts true."

Entwickelt schon 1972 von Alain Colmerauer und weiteren. Es ist die derzeit bedeutendste deklarative Programmiersprache. Spannend an diesem Thema ist, dass es mittlerweile viele Bibliotheken gibt, aus denen man das in ganz vielen Programmiersprachen machen kann. Das ist dann z.B. auch in Java oder anderen Sprachen eine mächtige Waffe. Und viele Probleme sind in logischer Programmierung (z.B. Sudoku , siehe ( ) um Größenordnungen einfacher und komplett Fehlerfrei zu programmieren. Gleichzeitig ist die Performance in der Regel zwar etwas langsamer (Faktor 2-10), aber manchmal sogar schneller, als wäre es imperativ ausprogrammiert!

Grundelemente

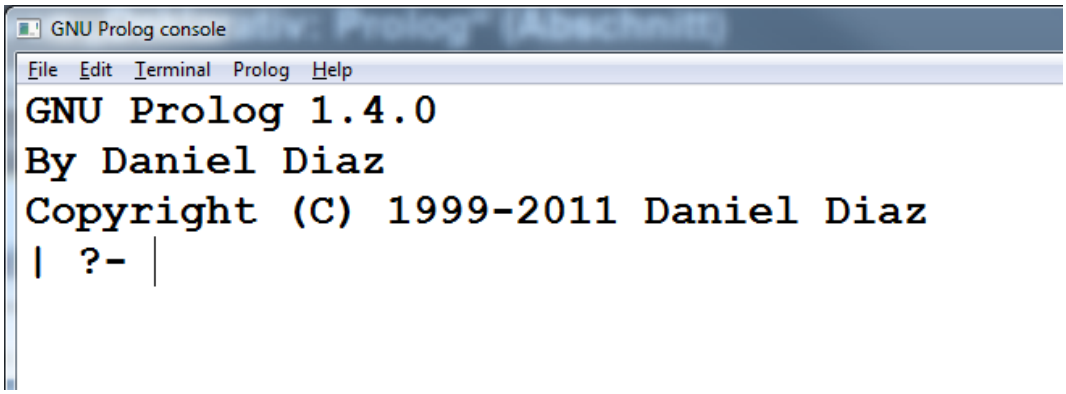
Die Elemente sind:

1. **Facts**: Fakten sind Aussagen über die reale Welt: "Anke ist eine Frau".
2. **Rules**: Regeln sind Aussagen, die Fakten verknüpfen: "Wenn Tom ein Mensch ist mag er Bier".
3. **Queries**: Abfragen auf dieser Menge von Fakten und Regeln: "Ist Linus Torvalds mit Edward Torwalds verwandt?"

Fakten und Regeln werden kompiliert und in einer Wissensbasis verwaltet.

Interaktives Beispiel

Nach dem Download des GNU Compilers kann dieser gestartet werden:



Eingabeaufforderung

Die Shell begrüßt uns mit einem:

| ?-

Die Eingabe von Fakten und Inferenzen wird nun immer mit einem Punkt abgeschlossen. Beendet werden kann das Programm mit CTRL-Z.

Jetzt sollte man im Menü
File|ChangeDir
in das Arbeitsverzeichnis wechseln, welches die Datei mit den Fakten enthält.

Es folgt ein Beispiel nach Tate.

Wir erzeugen eine Datei friends.pl mit dem Inhalt:

```
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).
friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

Basic Function
Arbeiten mit Prolog

Die drei ersten Zeilen sind Fakten. Letzte Zeile ist offensichtlich eine Regel, die drei Aussagen enthält.

Nun laden wir diese Wissensbasis in GNU Prolog mit

['friends.pl'].

GNU Prolog teilt uns nun mit, dass alles compiliert wurde und bereit ist für Abfragen:

compiling F:/CODING/PROLOG/friends.pl for byte code...
F:/CODING/PROLOG/friends.pl compiled, 12 lines read - 926 bytes written, 14 ms

Also tun wir eben dieses:

| ?- likes(wallace, sheep).
no
| ?- likes(grommit, cheese).
yes

Wichtig

Alle Schlüsselwörter die mit kleinem Buchstaben beginnen, sind unveränderbare Atome. Alle anderen, die mit einem Großbuchstaben beginnen (oder _) sind Variablen).

Schlussfolgerungen und Variablen

Das Ausführen der Regel führt zu folgendem Ergebnis:
| ?- friend(wallce, wallace).
no

Dies ist der erste Teil der Regel, wobei "\+" eine Negation bedeutet. Also dürfen Freunde schonmal nicht identisch sein.
| ?- friend(grommit, wallace).
yes
| ?- friend(wallace, grommit).
yes

Weiterhin hat die Regel offensichtlich definiert, dass zwei Subjekte Freunde sind, wenn sie das gleiche mögen (hier offensichtlich Käse).

Auch das Nicht-Freund-sein ist leicht gezeigt:

| ?- friend(wendolene, grommit).
no

Noch ein Beispiel:

```
food_type(gouda, kaese).
```

```
food_type(ritz, cracker).
food_type(dosenfleisch, fleisch).
food_type(wuerstchen, fleisch).
food_type(gerolsteiner, mineralwasser).
food_type(danone, nachtisch).
flavor(suess, nachtisch).
flavor(herzhaft, fleisch).
flavor(herzhaft, kaese).
flavor(suess, mineralwasser).
food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).
```

Basic Function
Prolog Beispiel 2

Wir können jetzt mit Variablen fragen, welches Fleisch es gibt. Dazu schalten wir mit ";" weiter:

| ?- food_type(What, fleisch).
What = dosenfleisch ? ;
What = wuerstchen ? ;
(31 ms) no
| ?-

Ein paar weitere:

| ?- food_flavor(wuerstchen, suess).
no

und

| ?- flavor(suess, What).
What = nachtisch ? ;
What = mineralwasser
(15 ms) yes

Offensichtlich liegt hier eine falsche Regel vor, da Mineralwasser nicht süß ist.

Noch eine tiefere Schlussfolgerung:

| ?- food_flavor(What, herzhaft).
What = gouda ? ;
What = dosenfleisch ? ;
What = wuerstchen ? ;
(16 ms) no

10.4 Core.logic

Core.logic ist eine Bibliothek für die Logische Programmierung unter Clojure. Ähnliche Bibilotheken gibt es in vielen Programmiersprachen.

Hinweis

> Wie Sie sich vielleicht schon gedacht haben: Logische Programmierung ist vergleichbar mit einer Umkehrfunktion. "Das Resultat wird übergeben und gesucht wird nach den Argumenten!" Wenn *result = rules* und *arguments = goals* sollte es wahr sein.

Wir programmieren also rückwärts!

Generelle Literatur

- Dirk W. Hoffmann, **Theoretische Informatik** , Kap 3 Logik und Deduktion, Hanser
- Attila Elci, **Logic Programming, a mathematical introduction**
- SICP, Abelson, 2xSussman, of CompPrgs, S. 402-445
- Sterling, Shapiro, **The Art of Prolog**, MIT Press
- Friedman et al. **The Reasoned Schemer**, MIT Press (das Standardwerk)

Core Logic Links

- core.logic on Github
- Core.Logic Wiki
- Core logic Primer
- Stuart Sierra: **Lojic Part Two** (get up and running)
- Ryan Senior The Magical Island of Kanren
- jrhead **Explorations in core.logic**
- Pepijn de Vos The Magic Square
- Examples from the Zürich Meetup
- Cheat SHeet

Sonstige Referenzen & Videos

- Tutorial: https://github.com/swannodette/logic-tutorial
- Steve Minder, Challenges for Logic Programming(Historic Stuff)
- Ryan Senior q conj11 Practical core.logic
- Edmund Jackson LP on JVM with core.logic
- A logic db

Basis: Introduction von Norman Richards

10.4.1 Einrichten der Umgebung

Erstellen Sie ein neues leiningen Projekt:
lein new logicfun

Fügen Sie folgendes in Ihre project.clj ein:
:dependencies [[org.clojure/clojure "1.5.1"]
[org.clojure/core.logic "0.8.5"]])

oder eben die aktuelle letzte Version (Clojure stable; core.logic)

Danach kann es hilfreich sein das folgende
warning ignorieren

oder
(:refer-clojure :exclude [==])

im Namespace anzugeben.

In Ihrem Programm importieren Sie dann einfach:
(ns your.namespace
(use clojure.core.logic)) ;; no quote needed

Falls es nicht funktionieren sollte, versuchen Sie:
(:require [clojure.core.logic :as xxx]))

Testen Sie mit:
(run 1 [q] (== 4 q))

Dann läuft es hoffentlich!

Und wieder den kompletten File in den namespace:

```
(ns clogic.core
 (use clojure.core.logic)
 (:refer-clojure :exclude [==]))

(defn -main [& args]
 (println (run 1 [q] (== 42 q))))
```

core.logic
src/clogic/core.clj

Hinweis

Man beachte, dass sowohl run, als auch das umfassende run* (liefere mir alle möglichen Ergebnisse zurück) eine lazySeq zurückliefert! Hier ist also erstmal ein first nötig, will man mit einem Wert weiterarbeiten.

10.4.2 Unification, conde und Disunification

Unification

Die einfachste Operation auf logischen Variablen ist die *Vereinheitlichung (engl. Unify / Gleichheit)*, d.h. das gleichsetzen:

```
(defn returnme []
(run* [q]
(== q :hello-world))
;; => (:hello-world)

(defn -main [& args]
(println (returnme))
```

Unification
Gleichheit abfragen

In den folgenden Listings sparen wir uns den Clojure Rahmen und geben nur noch den Inhalt der Funktion an.

Unification funktioniert auch über eine Sequenz:
(run* [q]
(== q [:hello :world]))
;; => ([:hello :world])

Und in der Regel hat man es bei Problemen der realen Welt immer mit Sequenzen / Listen zu tun!

Dies kann auch ohne Schaden mehrmals getan werden:
(defn returnme []
(run* [q]
(== q [:hello :world])

```
(== q [:hello :world]))
;; => ([:hello :world])
```

Verschiedene Werte mit *Gleichheit* an q zu binden funktioniert logischerweise nicht:

```
(run* [q]
(== q [:hello])
(== q [:world]))
;; => ()
```

Hinweis

Sie merken worauf das hinaus läuft. Wir geben alle unsere Regeln in einer (oder später auch mehreren) Funktion an! Der Solver im Hintergrund versucht nun möglichst effizient und korrekt die Lösung zu suchen.

condo

Will man aber nun, dass q aus dem letzten Beispiel beide Bedingungen erfüllt, dann hilft conde:

```
(run* [q]
(conde
[(== q :hello)]
[(== q :world)]))
;; => (:hello :world)
```

Wie bei let erwartet conde eine Menge von *clauses*, die logische Ausdrücke sind und alle erfüllt werden sollen.

Hinweis

Hier kann man ein Muster erkennen: Die normalen Begriffe wie cond, member, distinct, every, cons, rest, etc. sind ja in vielen Programmiersprachen schon vergeben. Daher wurde bei core.logic einfach ab und zu noch ein Buchstabe angehängt. Das ergibt dann conde, membero, distincto, everyg, conso, restero, etc.!

Disunification

Die entgegengesetzte Operation ist die *Disunification*, die mit != angegeben wird:

```
(run* [q]
(conde
[(== q :hello)]
[(== q :world)])
```

```
(!= q :hello))
;; => (:world)
```

q soll hier nie das *unify* erfüllen, d.h. die Disjunktion aus dem bisherigen q und

hello

ist gemeint. Also fliegt

hello

aus der Liste wieder heraus. Tatsächlich können hier die Argumente von != *komplette Terme* sein.

10.4.3 Fresh, membero und distincto

fresh

Der Punkt ist nun: Muss man ein komplexes Regelwerk innerhalb der run oder run* Funktion erstellen, dann baut man sich das q erst einmal zusammen. Um am Ende das q herleiten zu können, brauche ich aber vorne und mittendrin frische und ungebundene Variablen! Dazu gibt es das Schlüsselwort fresh:

```
(run* [q]
(fresh [x y]
(== x :something)
(== y :something-else)))
;; => (_0)
```

Hier werden zwar zwei Variablen gebunden, aber q ist immer noch selbst frisch und ungebunden. Dies wird mit dem Symbol _0 dargestellt!

Möchte man hingegen etwas mit x tun, könnte dies so aussehen:

```
(run* [q]
(fresh [x]
(== x :something)
(== q x)))
;; => (:something)
```

Man beachte, dass die Reihenfolge selbst keine Auswirkung hat:

```
(run* [q]
(fresh [x y]
(== q [x :and y])
(== x :something)
(== :something-else y))
```

;; => ([:something :and :something-else])

membero

membero ist die Memberfunktion:
(membero x l)

hat Erfolg, wenn x ein Element aus l ist.

Die Funktion
(run* [q]
(membero q [1 2 3])

- ₀ => (1 2 3)

liefert also für q alle Möglichkeiten aus dem Vektor l [1 2 3].

Oder umgekehrt:
(run* [q]
(membero 7 [1 3 8 q]))
;; => (7)

Wenn 7 in der Liste enthalten sein muss, dann muss q den Wert 7 annehmen.

Daher liefert das nachfolgende Beispiel alles aus dem zweiten Beispiel von membero aber doppelt:
(run* [q]
(fresh [smurf]
(membero smurf
[:papa :brainy :lazy :handy])
(== q [smurf smurf])))
;; >= ([:papa :papa] [:brainy :brainy] [:lazy :lazy] [:handy :handy])

Interessant wird es, wenn membero mehrmals Erfolg hat:
(run* [q]
(fresh [s1 s2]
(membero s1 [:bmw :mercedes :porsche])
(membero s2 [:vw :audi])
(== q [s1 s2])))
;; => ([:bmw :vw] [:bmw :audi] [:mercedes :vw] [:mercedes :audi]
;; [:porsche :vw] [:porsche :audi])

In diesem Fall wird q mit jedem Paar unified (Kreuzprodukt?!).

distincto

Anders herum funktioniert distincto:

```
(run* [s1 s2 s3]
(membero s1 [:bmw :mercedes])
(membero s2 [:vw :audi])
(membero s3 [:opel :isetta])
(distincto [s1 s2 s3])
;; => ([:bmw :vw :opel] [:bmw :vw :isetta] [:bmw :audi :opel]
;; => [:bmw :audi :isetta] [:mercedes :vw :opel] [:mercedes :vw :isetta]
;; => [:mercedes :audi :opel] [:mercedes :audi :isetta])
```

Hier sind alle Paare, deren Elemente sich unterscheiden. Quasi analog zum mathematische / logischen oder auch SQL Operator.

10.4.4 Everyg, lvar, conso und resto

everyg

everyg sichert zu, dass jedes Element in der Collection ein Ziel erfüllt. D.h. es ist keine Relation im klassischen Sinne:

```
(run* [q]
(fresh [s1 s2 s3]
(== q [s1 s2 s3])
(everyg #(membero % [:bmw :audi :mercedes]) q)
(distincto q)
)
)
;; => Alle Permutationen ohne die Distinct Zeile, d.h. 27 (3^3)
;; => Nur distincte permutationen, d.h. 6 wie [:bmw :audi :mercedes]
;; [:bmw :mercedes :audi], d.h. jedes an jeder Position nur einmal.
```

lvar

todo

conso

todo

resto

todo ...

10.4.5 Beispiele

- Map Coloring
- Rock Paper Scissors
- Logic Puzzles (as the Zebra Riddle)
- Facts and Relations
- Finite Domains
- Sudoku Solver in 13 lines...

10.5 Zusammenfassung LPR

11 DSL - Externe und interne DSLs

Autor: Prof. Dr. Stefan Edlich

11 DSL - Externe und interne DSLs
11.1 Überblick und Lernziele DSL
11.2 Einführung in DSLs
11.3 DSLs tiefer betrachtet
11.4 Die erste DLS: Expression Builder
11.5 Fluent Interfaces
11.6 Mächtigere Sprachen
11.7 Formen interner DSLs
11.8 Formen externer DSLs
11.9 Wann welche DSL?
11.10 DSLs mit Xtext
11.11 DSLs mit ANTLR
11.12 Zusammenfassung DSL
11.13 Wissensüberprüfung DSL
11.14 Aufgaben
11.15 DSLs Literatur

11.1 Überblick und Lernziele DSL

Lernziele

Lernziele

Ziel dieser Lerneinheit ist es, Sie mit DSLs vertraut zu machen. Wir hatten DSLs sogar bereits an vielen Stellen in diesem Modul. Gradle ist ein gutes Beispiel dafür: Gradle ist keine eigene Programmiersprache, deckt aber ein kleines Gebiet gut ab.

Weiterhin ist es wichtig:

- Die verschiedenen Arten von DSL zu kennen und einschätzen zu können
- Ein paar DSLs selbst entworfen, implementiert und getestet zu haben

Gliederung

Gliederung

Nach den Definitionen und Einführungen werden wir uns zuerst mit internen und dann mit externen DSLs beschäftigen.

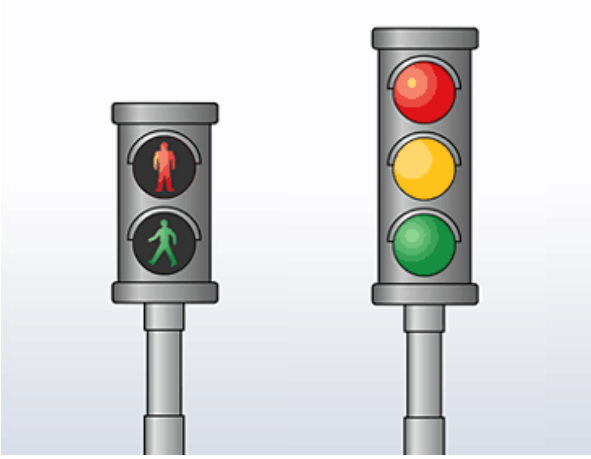
Zeitumfang

Zeitumfang

Zum Durcharbeiten der Lerneinheit benötigen Sie ca. 120 Minuten und für die Bearbeitung der Übungen ca. 120 Minuten.

11.2 Einführung in DSLs

Unter "Domäne" aus Domänenspezifische Sprache versteht man ein Problemfeld oder auch einen Problembereich. Es geht also um eine Kommunikation die auf ein bestimmtes eingegrenztes Problemfeld bezogen ist. Domänenspezifische Kommunikation umgibt uns an vielen Orten. An der Ampel sagen uns domänenspezifische Farben, wie wir uns mit dem Auto verhalten sollen. Seefahrer und Piloten haben ebenfalls einen speziellen Sprachsatz, der auf bestimmte Probleme und Anwendungsgebiete ausgerichtet ist.



Ampel
Eine domänenspezifische Kommunikationsform

Wird eine solche Kommunikationsform in eine Sprache gegossen, dann ist dies eine Domain Specific Language (dt. Domänenspezifische Sprache), kurz DSL genannt. Der Vorteil dieser Sprachen liegt im Wesentlichen in der besseren Abstraktion. Natürlich könnte auch an jeder Ampel eine Person stehen und uns erklären, was wir zu tun haben und was passieren würde, wenn wir gerade jetzt über die Straße fahren würden.

Definition

DSL
Eine DSL ist eine Programmiersprache, die einen kleinen Problemraum abdeckt. Die Syntax und Semantik der DSL entsprechen möglichst der gleichen Abstraktionsebene, wie das Problemfeld selbst.

An vielen Stellen hat sich gezeigt, dass Menschen mit Maschinen oder Computern (oder auch diese untereinander) viel besser über DSLs kommunizieren können. Sie kennen bestimmt die Börsenhändler an der Wall Street, die mit ihren weißen Handschuhen durch die Luft wedeln und domänenspezifische Signale geben.

Auch mit unseren Computern kommunizieren wir domänenspezifisch, um effizienter zu sein. Die Datenbankabfragesprache SQL ist ein wunderbares Beispiel:

```
SELECT [DISTINCT] Auswahlliste [AS Spaltenalias]
FROM Quelle [AS Tabellenalias]
[WHERE Where-Klausel]
```

```
[GROUP BY (Group-by-Attribut)+]
[HAVING Having-Klausel]
[ORDER BY (Sortierungsattribut [ASC|DESC])+];
```

SQL
SQL ist eine DSL

Natürlich sind alle Programmiersprachen DSLs in irgendeiner Art und Weise.

Im Prinzip geht es daher um jede Art von besserer Abstraktion.

Uns interessieren in unserem Rahmen natürlich alle nicht trivialen Sprachen und solche, die wir als Informatiker für unsere Arbeit selbst bauen können.

Weiterhin unterscheiden wir zwei Arten:

Definition

Unter einer **externen DSLs** versteht man eine Sprache, die nicht in eine Programmiersprache eingebettet ist, sondern ihren komplett eigenen Sprachraum beinhaltet. SQL ist ein solches Beispiel: ein abgeschlossener Sprachraum für die Kommunikation mit Datenbanken.

Unter einer **internen DSL** versteht man eine DSL, die in die Programmiersprache eingebettet ist. Man kann also die Programmiersprache ganz normal nutzen, hat aber gleichzeitig eine Menge von Schlüsselwörtern, die die Kommunikation mit einer speziellen Domain / für eine bestimmte Aufgabe erleichtert. Gradle ist hier ein schönes Beispiel: der volle Groovy Sprachraum steht zur Verfügung und man hat eine mächtige Bibliothek für das Buildmanagement an der Hand, mit der man über Schlüsselwörter (task hello << {...}) kommunizieren kann.

Es geht also darum, ein leistungsfähiges Vokabular zu erstellen, welches quasi unsichtbar in die Sprache der Zieldomain konvertiert wird. Dies unterscheidet die DSL von der Programmiersprache. Letztere sind breit angelegt und adressieren alles. DSLs haben den Fokus auf genau einem Problemfeld.

Vorteile des Einsatzes von DSLs sind natürlich:

- Höhere Effizienz durch höhere Expressiveness
- Einigung auf ein gemeinsames Vokabular
- Bessere Beschreibung und besseres Testen

11.3 DSLs tiefer betrachtet

Betrachten wir nochmal einige Beispiele (aus Gosh 2011):

- **HTML**: Sprache für das Web Markup
- **CSS**: Sprache für die Definition von Stylesheets
- **RSpec**, Cucumber: Sprachen für Behaviour-Driven Testing
- **Make**, Ant, Rake, Maven, Gradle: Sprachen für das Buildmanagement
- **YACC**, Bison, ANTLR: Sprache für Parser Generatoren
- **SQL**: Sprache zur Kommunikation mit Datenbanken

Fassen wir nochmals die **Vorteile** zusammen:

1. DSLs arbeiten auf einem hohen Level der Expressiveness und daher abstrakter.
2. DSLs sind präzise und lassen keinen Deutungsspielraum.
3. DSLs können auch unter Menschen leichter verstanden und damit zur Kommunikation verwendet werden.
4. DSLs sind auch für Laien gedacht.
5. Durch die größere Expressiveness ist man effizienter.

Aber natürlich gibt es auch **Nachteile**:

1. Das Erstellen einer guten DSL kostet Zeit und Geld.
2. Eine gute DSL zu designen ist schwer.
3. DSLs fügen einen Layer in das System ein. Dies kann in seltenen Fällen Performance kosten.
4. DSLs müssen gelernt werden und sind 'noch eine Sprache mehr'.
5. DSLs brauchen meistens auch guten Tool Support wie z.B. *Syntax Highlighting*.

Noch ein Beispiel zu verschiedenen Abstraktionsebenen:

Aufgabe

Aufgabe: Was tut der nachfolgende Code?

```
def is_stuff_char (c)
 (c >= ?a && c <= ?z ) || ( c >= ?A && c <= ?Z) ||
 (c >= ?0 && c <= ?9) || c == ?_
end
def find_stuff (str)
```

```
result = [ ] ; i = 0 ; while i < str.length
start = i ; i += 1 while is _stuff_char(str[i])
if start != i && str[i] == '@'
at = i ; i += 1
i += 1 while is_stuff_char (str[i])
result << str [start ... i] if at != i
end ; i += 1 ; end
result ; end
```

C-Code
Was tut dieser Code?

Etwas, was man in Ruby oder anderen expressiven Sprachen in drei Zeilen schreiben könnte:

```
def find:emails(str)
str.scan(/\w+@\w+/)
end
```

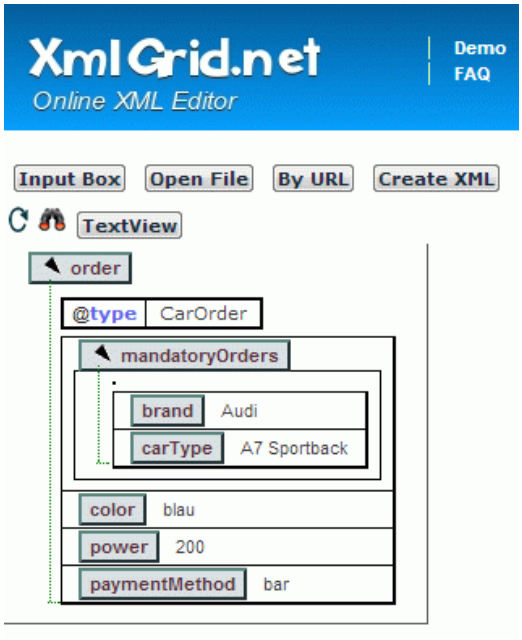
Offensichtlich findet der Code heraus, ob es sich um eine Email Adresse handelt oder nicht. Doch warum ist die zweite Variante um so vieles besser? Weil auch Regular Expressions eine wunderbare DSL sind (siehe LE Advanced Java). Man vergleiche auch hier wieder die Abstraktionsebenen.

XML als DSL

Auch XML kann eine gute DSL sein. Sie ist zwar nur mittelmäßig menschenlesbar, dafür aber gut von Maschinen lesbar und gut visualisierbar.

```
<order type="CarOrder">
 <mandatoryOrders>
  <brand>Audi</brand>
  <carType>A7 Sportback</carType>
 </mandatoryOrders>
 <color>blau</color>
 <power>200</power>
 <paymentMethod>bar</paymentMethod>
</order>
```

XML Code



XML Ansicht des XML-Codes in Baumstruktur

Und für genau dieses Beispiel eines Autokaufes werden wir gleich eine DSL in Java bauen.

Annotations

Aber auch Annotation oder Makros können eine gute Form von DSL sein:

```
class PatientVisit...
@ValidRange(lower = 1, upper = 1000)
private int weight; // in lb
@ValidRange(lower = 1, upper = 120)
private int height; // in inches
@Author(name = "Dante Alighieri",date = "14.09.1321")
class EineKlasse() { }
```

Und mit @Deprecated, @Override, @SuppressWarnings verwenden wir ja bereits schon viele Annotations. @Test, @beforeTest, @afterTest, @beforeMethod, @after, etc. sind auch aus Testframewkroks bekannt.

Annotations unter Java sind relativ einfach DSLs, weil sie leicht zu definieren sind und sogar geschachtelt werden können. Dazu ein paar Beispiele:

```
@Vorläufig
class Parser {
void parseXML();
}
@Autoren({"Backschat", "Edlich"})
```

@AutorenDetails(name = "Stefan Edlich", alter = 42) // ;-)

11.4 Die erste DLS: Expression Builder

Um schnell ein *Hands On* zu bekommenen, schauen wir uns eine DSL an, die nach dem **Expression Builder**-Pattern gestaltet ist. In diesem Fall in Java. Das nachfolgende Beispiel, zeigt dann die Verwendung des Expression Builders mit einem Fluent Interface.

Ein Expression Builder erlaubt es Regeln zu erstellen, um Daten zu sammeln und diese dann in einem Objekt zu bündeln.

Ein Fluent-Interface ähnelt einer Verkettung von Methodenaufrufen, um ein Objekt so zu parametrisieren, dass die Parameter schließlich in diesem Objekt gespeichert werden.

Beide *Design patterns* hängen eng zusammen. Die Begriffe wurden von Martin Fowler und Eric Evans geprägt.

Ein Fluent Interface ist eine einfache Möglichkeit, in einer Zeile Argumente zu übergeben, Nachrichten in Form von Methodenaufrufen zu übermitteln (.barZahlung()) und dabei eine Grammatik zu erzwingen. Beispielsweise darf nur eine bestimmte Reihenfolge von Methodenaufrufen vorliegen.

Ein Beispiel dafür kennen wir bereits aus dem Testen:

when(mb.mydouble(2)).thenReturn(4);

Hier wird in Mockito ein Mock so parametrisiert, das er weiß was zu tun ist. Und das angenehme ist hier, es liest sich wie ein Satz in Englisch: "Wenn die Methode mydouble aus dem Objekt mb mit dem Argument 2 aufgerufen wird, dann gebe 4 zurück". Dies scheint eine gut lesbare DSL zu sein.

Das letztendliche Ziel ist es, sogenannte Value Objets erstellen. Value Objets sind ebenfalls ein Design Pattern, bei dem man ein Objekt mit unveränderlichen Werten erstellt. Die Idee dahinter ist, dieses Objekt herumreichen zu können ohne dass es manipuliert werden kann. Alle Instanzen die das Objekt bekommen verlassen sich also darauf, dass das Objekt immer nur mit final Feldern gefüllt ist.

Schauen wir uns ein Beispiel an (nach Gosh 2011): (*security* bedeutet in diesem Beispiel so viel wie Aktie)

```
package de.edlich.vfhvws.dsls;
```

```
public class Order {
 static class Builder {
  private String security;
  private int quantity;
  private int limitPrice;
  private boolean allOrNone;
  private int value;
  private String boughtOrSold;

  public Builder(){}
  public Builder buy(int quantity, String security) {
   this.boughtOrSold = "Bought";
   this.quantity = quantity;
   this.security = security;
   return this;
  }

  public Builder sell(int quantity, String security) {
   this.boughtOrSold = "Sold";
   this.quantity = quantity;
   this.security = security;
   return this;
  }
  public Builder atLimitPrice(int p) {
   this.limitPrice = p;
   return this;
  }
  public Builder allOrNone() {
   this.allOrNone = true;
   return this;
  }
  public Builder valueAs(OrderValuer ov) {
   this.value = ov.valueAs(quantity, limitPrice);
   return this;
  }
  public Order build() {
   return new Order(this);
```

```
  }
 }
 private final String security;
 private final int quantity;
 private final int limitPrice;
 private final boolean allOrNone;
 private int value;
 private final String boughtOrSold;

 private Order(Builder b) {
  security = b.security;
  quantity = b.quantity;
  limitPrice = b.limitPrice;
  allOrNone = b.allOrNone;
  value = b.value;
  boughtOrSold = b.boughtOrSold;
 }
}

package de.edlich.vfhvws.dsls;

public interface OrderValuer {
 int valueAs(int qty, int unitPrice);
}

package de.edlich.vfhvws.dsls;

public class StandardOrderValuer implements OrderValuer {
 public int valueAs(int qty, int unitPrice) {
  return qty*unitPrice;
 }
}

package de.edlich.vfhvws.dsls;

public class RunMe {

 public static void main(String[] args) {
```

```
  Order o = new Order.Builder()
      .buy(100, "IBM")
      .atLimitPrice(300)
      .allOrNone()
      .valueAs(new StandardOrderValuer())
      .build();
 }
}
```

Builder Pattern
Builder Pattern in Java

DSL-01: Java-Code ausführen und verstehen

Führen Sie den Beispiel-Code aus und verstehen Sie jede Zeile!
Bearbeitungszeit: 20 min.

Hier fällt einiges negativ auf:

- Java scheint keine allzu gute Sprache für DSLs zu sein. Der Code ist umfangreich und enthält zu viel 'Boilerplate' Code (langweilige / umständliche Codebausteine). Moderne dynamische Sprachen könnten dafür besser geeignet sein.

Interessant ist aber:

- wenn man mit buy etwas kauft, dann müssen sowohl die Menge, als auch die Art der Aktie angegeben sein. Ansonsten macht die Order selbst keinen Sinn. Hier werden also zwingend Argumente gekoppelt
- mit atLimitPrice wird ganz normal ein Parameter angegeben. Hier der maximale Preis zu dem man zu kaufen bereit ist.
- Mit allOrNone wird zwar kein Parameter übergeben, sondern indirekt über den Namen der Funktion ein boolscher Wert gesetzt.
- mit dem Interface OrderValuer und der entsprechenden Implementierung dazu, können auch Verfahren übergeben werden (dies ähnelt dem Strategy Pattern).

Es bleibt die Frage ob das a) auch in Java einfacher geht und b) dies nicht in anderen Sprachen noch viel einfacher zu realisieren ist.

11.5 Fluent Interfaces

Das vorher genannte Beispiel eines Expression Builders mit Fluent Interfaces lässt sich auf Kosten anderer Features stark vereinfachen:

```
package de.edlich.vfhvws.dsls2;

public class CarOrder {
 String marke;
 String bezeichnung;
 String farbe;
 int leistung;
 boolean zahlart; // true = Bar, false = Kredit

 public CarOrder(){};
 public CarOrder(String bez, String mar){
  this.bezeichnung = bez;
  this.marke = mar;
 }
 public CarOrder farbe(String f) {
  this.farbe = f;
  return this;
 }
 public CarOrder leistung(int kw) {
  this.leistung = kw;
  return this;
 }
 public CarOrder barzahlung() {
  this.zahlart = true;
  return this;
 }
 public CarOrder kreditzahlung() {
  this.zahlart = false;
  return this;
 }
 public String toString(){
  return "Kauforder: " + this.marke + ", " + this.bezeichnung +
    "\nFarbe: " + this.farbe +
    "\nLeistung: " + this.leistung + " Kw" +
```

```
    "\nBarzahlung: " + this.zahlart;
  }
}

package de.edlich.vfhvws.dsls2;

public class RunMe {

  public static void main(String[] args) {
   CarOrder co = (new CarOrder("Audi", "A7 Sportback"))
     .farbe("blau").leistung(200).barzahlung();
   System.out.println(co);
  }
}
```

Fluent Interface
Ein einfaches Fluent-Interface in Java

In diesem stark vereinfachten Beispiel haben wir auf einiges verzichtet:

- Wir haben kein Value Objekt. Die Felder / Attribute sind nicht final
- Wir haben keine Objekte in den Methoden übergeben (wie StandardOrderValuer()).

Weiterhin haben wir hier keine Möglichkeit, irgendwelche Reihenfolgen fest zu definieren. Dennoch sind die Bezeichnung der Automarke (bezeichnung) und die Marke selbst (marke) verpflichtend. Ohne dieses Basisobjekt können keine weiteren Attribute gesetzt werden.

Dennoch sind wir hier völlig frei, ohne Reihenfolge, Methoden zu setzen oder Werte auch nicht zu übergeben. Letzteres birgt natürlich auch das Risiko, dass Felder nicht gesetzt sind und das Objekt ggf. nicht sinnvoll weiterverarbeitet werden kann.

Der Stil des Beispiels oben wird daher auch oft als Method-Chaining bezeichnet und nicht für gut befunden, da dies fehleranfällig ist.

Hinweis

Versuchen Sie daher selbst Energie in dieses Pattern zu investieren und das Objekt gegen Fehlerfälle, falsche Reihenfolgen oder sonstige Veränderungen abzusichern!

Dennoch ahnen wir bereits, dass es in Sprachen wie Python, Ruby, Groovy, Scala oder Clojure besser gehen kann.

11.6 Mächtigere Sprachen

Einfacher wird dies alles in mächtigeren Sprachen wie Ruby, Groovy, Scala oder Clojure. Diese Hochsprachen mit höherer Expresiveness haben für DSLs mehrere Vorteile:

- Sie erlauben viel besseres Metaprogramming
- Sie unterstützen Closures

Ein typisches Beispiel mit Rake:

```
require 'rubygems'
require 'rake/packagetask'

spec = Gem::Specification.new do |s|
 s.platform = Gem::Platform::RUBY
 s.summary = "Builds in Ruby"
 s.name = 'rake'
 s.version = PKG_VERSION
 s.requirements << 'none'
 s.require_path = 'lib'
 s.autorequire = 'rake'
 s.files = PKG_FILES
end
```

Rake
Rake Beispiel

Rake ist das Buildsystem von Ruby. Entscheidend ist dabei das Folgende: Man eröffnet einen Block. Dies kann durchaus auch wie ein Methodenaufruf betrachtet werden. Wichtig ist aber, dass dieser Block einen Kontext eröffnet. Innerhalb dieses Blockes können jetzt weitere Operationen vorgenommen werden. Wir kennen das bereits aus dem Buildsystem Gradle.

task :make_coffee do
cups = ENV["COFFEE_CUPS"] || 2
puts "Made #{cups} cups of coffee. Shakes are gone."

end

Hier kann wie bei Gradle einfach nur etwas ausgeführt werden, oder aber der umschließende Kontext angereichert werden. Die inneren Operationen können also indirekt Zugriff auf den äußeren Kontext bekommen. Auf der nächsten Seite wird erläutert, wie dies geht (tatsächlich bekommt der äußere Kontext Zugriff auf den inneren.)

Wie würde unser erstes Beispiel (jetzt unter einer solchen Sprache wie Groovy) aussehen:

```
class Order {
 def security
 def quantity
 def limitPrice
 def allOrNone
 def value
 def bs //buy or sell

 def buy(su, closure) {
  bs = 'kauf'
  buy_sell(su, closure)
 }

 def sell(su, closure){
  bs = 'lease'
  buy_sell(su, closure)
 }

 private buy_sell(su, clojure) {
  security = su[0]
  quantity = su[1]
  closure()
 }

 def getTo() {
  this
```

```
 }

 def methodMissing(String name, args) {
      order.metaClass.getMetaProperty(name).setproperty(order,
 args)
 }

 def getNewOrder(){
  order = new Order()
 }

 def valueAs(closure) {
  oder.value = closure(order.quantity, order.limitPrics[0])
 }

 Integer.metaClass.getShares = {-> delegate}
 Integer.metaClass.of = {instrument -> [instrument, delegate]}
}
```

Groovy
Aktienkauf in Groovy (C) Ghosh

Dann können wir die DSL so beschreiben:

```
newOrder.to.buy(100.shares.of('IBM')) {
limitPrice 300
allOrNone true
valueAs (qty, unitPrice -> qty * unitPrice - 500)
}
```

Beispielsweise in einer Datei.

Diese Datei könnte Groovy dann ganz einfach ausführen:

```
def dslDef = new File('ClientOrder.groovy').text
def dsl = new File('order.dsl').text
def script = """
$(dslDef)
$(dsl)
"""
new GroovyShell().evaluate(script)
```

Man beachte, dass die ersten beiden Zeilen dieses Scriptes wieder wie eine DSL aussehen.

In der ersten Zeile des letzten Codes wird die DSL 'Grammatik' geladen. In der zweiten Zeile die eigentliche Order. Danach wird der ganze Code ausgeführt. Jetzt wäre es ein leichtes, das Folgende zu tun:

- Alles in eine Schleife zu packen.
- Alles auszuführen, und am Ende Code anhängen der das fertige Orderobjekt liest.
- Der Code muss dann nur noch die Order dem Handelssystem übermitteln.

Und schon hätte man ein einfaches Handelssystem, mit dem man eine textuelle Order erstellen könnte.

Der weitere Vorteil wäre, dass der Groovy Compiler hier nicht nur die Syntax der menschen-geschriebenen DSL überprüfen könnte. Es könnten weitere Regeln hinterlegt werden, bei denen die Eingaben geprüft werden könnten.

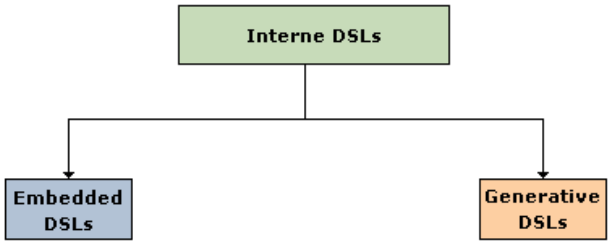
Anmerkung

Man sieht hier schon wieder ein Pattern, welches in der LE Programmierparadigmen weiter entwickelt wird: Je höherwertiger das Programmieren - also je mehr es in Richtung Metaprogramming geht - desto mehr schreibt man nur noch Regeln. Die Verarbeitung und Auflösung dieser Regeln übernehmen dann Subsysteme. Ein Trend nicht nur in der Softwareentwicklung (imperativ -> OOP -> funktional -> deklarativ), sondern auch in der KI allgemein: Die Arbeit mit regelbasierten Knowledgebases, die aber dennoch 'fuzzyness', d.h. Gewichte für nicht-diskrete Statistik erlauben.

11.7 Formen interner DSLs

Man unterscheidet im Wesentlichen zwei Formen von internen DSLs:

- Embedded DSLs
- Generative DSLs



Unterteilung interner DSLs

Auf den folgenden Seiten erfahren Sie mehr über die Formen der internen DSLs und deren weiterer Unterteilungen.

11.7.1 Embedded DSLs

Schauen wir uns die erste Variante an: Hier werden Daten und Typen erstellt, die selbst schon Teil der Sprache sind. Typische Beispiele sind:

- **Smart APIs**: Hier werden - wie beim Fluent Interface - mit bereits vorhandenen Sprachmitteln so Objekte aufgebaut, dass aus diesen später wieder Aktionen ausgeführt werden können. Dies ist mit Sprachen wie Java oder Ruby leicht möglich.
- **AST Manipulation**: AST steht für Abstract Syntax Tree. Dabei wird aus dem gegebenen Code / Objekt ein Baum aufgebaut, der dann leicht transformiert werden kann. Ein Beispiel wäre eine Wurzel mit einem Vergleich der drei Kinder besitzt: 1) größer 2) Variable a und 3) Variable b. Dieses Konstrukt kann man dann leicht in alle Programmiersprachen transferieren, auch wenn die letztendliche Syntax dann völlig unterschiedlich ist. Dies ist mit Bibliotheken oder auch unter Java, Groovy oder Ruby leicht möglich. Mehr zu AST's aber in späteren Kapiteln.
- **Eingebettete Typen**: In Scala oder Haskell kann man beliebige Typen definieren, um diese in eine DSL innerhalb der Sprache zu füllen. Dies ist dann geprüft zur Übersetzungszeit. Ein Beispiel ist in dem nachfolgenden Listing zu sehen. Das geht natürlich nur in statisch getypten Sprachen wie Haskell oder Scala.

```
trait Trade { // (C) Gosh
type Instrument
type Account
def account: Account
def instrument: Instrument
def valueOf(a: Account, i: Instrument): BigDecimal
def principalOf: BigDecimal
```

```
def valueDate: Date
}
```

- **Reflektive Metaprogramming**: Schauen Sie sich noch einmal das Rake-Beispiel der vorigen Seite an. Ein Pattern, welches auch in Ruby intensiv verwendet wird ist, dass in einem Block Methoden aufgerufen werden, die noch nicht bekannt sind. Der äußere Block bekommt die inneren Codeteile als Parameter. Das sind jedoch meistens Methoden, können aber auch Properties oder Daten sein (die Unterschiede verschwimmen hier, da in Ruby alles Function call ist). Da der äußere Block jedoch die inneren Methodenaufrufe nicht kennen kann, ruft Ruby hier die Methode: method_missing auf und verarbeitet den Inhalt. Das bedeutet, der Anwender kann beliebige (DSL) Methoden erfinden (oder nutzen), aber die Sprache kann diese dennoch verarbeiten, weil method_missing einfach das Ergebnis des Methodenaufrufes nimmt und verarbeitet. Dies geht in Sprachen wie Ruby oder Groovy sehr gut.

Hinweis

Die Einführung von method_missing war ein ziemlich genialer Schachzug in Ruby (ohne wäre z.B. Rails nicht entstanden). Es erlaubt beliebige APIs / DSL zu definieren, und die Inhalte dann einfach auszuwerten.

Hier ein Beispiel aus der Definition eines Active Records:

```
class Client < ActiveRecord::Base
has_one :address
has_many :orders
has_and_belongs_to_many :roles
end
```

Oder Abfragen in Rails:

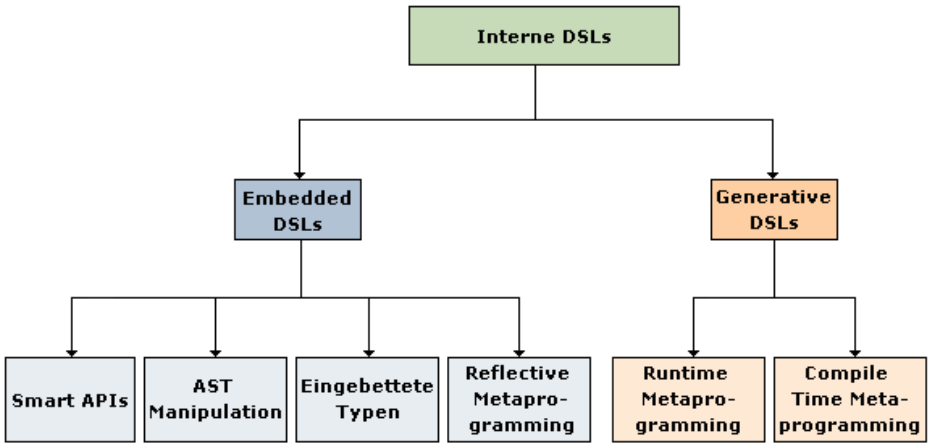
```
Client.where(:created_at => (params[:start_date].to_date)..(params[:end_date].to_date))
Category.joins(:products).merge(Product.cheap)
Album.find_or_initialize_by_name('I Got a Woman', :release_year => 1955)
Album.where( :release_year => 1960..1966 )
Album.where(:release_year => 1960..1969).order(:name).reverse_order
```

11.7.2 Generative DSLs

Wenden wir uns nun der zweiten Klasse der internen DSLs zu: Den Generativen DSLs. Hier treffen wir auf zwei Formen:

1. Runtime Metaprogramming
2. Compile Time Metaprogramming

In der nachfolgenden Grafik finden Sie zur Übersicht die Unterteilung der internen DSLs.



Unterteilung interner DSLs

Beim **Runtime Metaprogramming** werden dem Code Methoden hinzugefügt, die wie Klebstoff fungieren und weitere Methoden generieren, die dann die eigentliche Aufgabe übernehmen. Hier werden also mit diesen Hilfsmethoden schlussendlich Objekte oder weitere Codeteile generiert.

Das Codestück der vorherigen Seite ist so ein Fall:
newOrder.to.buy(100.shares.of('IBM'))

Die Schlüsselwörter shares und of fügen hier ja fast nur gut lesbare englische Wörter ein, reichen aber meistens die Argumente nach vorne weiter.

Tatsächlich ist auch das obige Active Record-Beispiel aus Rails ein solches Beispiel. Bei der Definition des Records wird intern eine Menge Code generiert (z.B. aus der Active Record Klasse hinzugefügt), der das Arbeiten mit einem Record wie einem Client stark vereinfacht.

Aber auch die gesamten Rails Methoden gehören in diese Kategorie:
where select group order reorder reverse_order limit offset
joins includes lock readonly from having

Wieder sind hier Ruby und Groovy starke Vertreter dieser Kategorie.

Die letzte Subklasse ist das **Compile-Time Metaprogramming**. Entscheidend ist dabei, dass der Code nicht zur Laufzeit, sondern zur Compile-Zeit generiert wird. Dies ist meistens ein Feature von Lisp ähnlichen Sprachen wie Clojure oder Template Haskell. Aber auch Scala denkt derzeit (Stand Ende 2012) über eine derartige Erweiterung nach. In diesen Sprachen wird dies überwiegend als *Macro* bezeichnet. Der Vorteil ist hier, dass die DSL Elemente wirklich Teil der Sprache werden! Die Elemente können hier schon zur Übersetzungszeit, validiert und optimiert werden. Dies ist effizienter und schneller.

Das einfachste, schon erwähnte, Beispiel sind die aus C und C++ bekannten Makros, die aber in ihrer Mächtigkeit doch stark beschränkt sind. Sie arbeiten nur auf lexikalischer Ebene, d.h. es werden mehr oder (eher) weniger Textersetzungen durchgeführt. In Lisp kann man dagegen komplett auf dem AST operieren, da Code und Daten eine Einheit bilden.

Hinweis

Diese Sprachen werden als Homoiconic bezeichnet. Die Eigenschaft als Homoiconicity. Ein Begriff, der schon von Alan Kay 1969 in seiner Dissertation verwendet wurde. Diese Möglichkeit bieten einige Sprachen wie XSLT, XQuery, Io, Ioke, Joy, Factor, Prolog, R, Tcl, Mathematica und Julia. Lisp Sprachen sind jedoch ein Paradebeispiel für Macros, da sogenannte S-Expressions verwendet werden. Mehr jedoch dazu im Kapitel über Programmierparadigmen (wer dem schon vorgreifen will liest hier). S-expressions

Die (DSL) Makros werden vor dem eigentlichen kompilieren bearbeitet, expandiert und in sogenannte *valid Forms* (in etwa *gültige Schlüsselwörter*) überführt. Dann kennt diese der Compiler und kann das gesamte Programm kompilieren, so dass dem Anwender dann ein zusätzlicher (DSL) Satz an gültigen Schlüsselwörtern (*valid Forms*) zur Verfügung steht.

11.8 Formen externer DSLs

Die internen DSLs haben bisher die Sprachmittel der Sprache verwendet, in der sie eingebettet waren. Natürlich kann man auch immer eine Datei in dieser Sprache einlesen und ausführen lassen bzw. diese interpretieren. Dies geht in vielen Sprachen einfach z.B. in Ruby mit eval:

```
eval("20+22")
```

Dennoch ist die Sprache der Datei immer gebunden und nicht frei erfindbar. Es besteht daher oftmals das Verlangen, sich aus dem Bann der Programmiersprache zu lösen. Und einfach etwas komplett Eigenes und Neues zu definieren, dessen Sprachelemente und Syntax man sich selbst und vollständig überlegt hat.

Vorteil

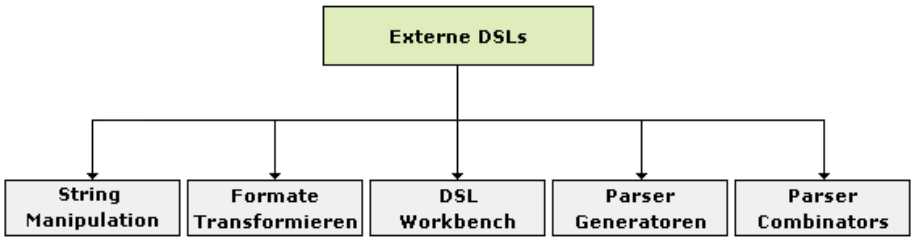
- Die Sprache passt noch besser und geht keine Kompromisse ein.

Nachteil

- Alle Mächtigkeit der Sprache geht verloren

Beispiel: Warum soll eine App für ein Bauern nicht solche Eingaben ermöglichen oder entsprechende Eingaben einlesen können:
Grunz 4 weg // (verkaufe 4 Schweine)
Muuh 2 her // (kaufe 2 Kühe)

Um das zu erreichen gibt es auch hier wieder viele Möglichkeiten die im Folgenden vorgestellt werden.



Formen externer DSLs

- String Manipulation
- XML, JSON, etc. Transformation und Ausführung
- DSL Workbench
- Parser Generatoren
- DSL basierend auf Parser Combinators

11.8.1 String Manipulation

Bei dieser einfachsten Form werden nur Strings eingelesen und mit geeigneten Hilfsmitteln (wie regex Parser) bearbeitet.

Der DSL String
Grunz 4 weg

kann sicher leicht in folgenden Code übersetzt werden:
Order("Pigs").sell.amout(4)

Was so ziemlich in allen Sprachen mit Clojures so oder in ähnlicher Form geht.

Vorteile

- Man muss kein Werkzeug lernen. Man kann fröhlich drauflos hacken.
- Ist gibt keine Abhängigkeit von Tools oder Bibliotheken.

Nachteile

- Eine Syntax-Überprüfung ist zunächst nicht vorhanden.
- Auch eine Hilfe durch einen Editor ist initial nicht da (z.B. Tooltips oder Strg+Space, etc.).

Auf Grund dieser Nachteile setzen Profis meistens auf die nächsten Varianten.

11.8.2 Formate wie XML, JSON Transformieren

Hier ist man bei dem Design der DSL der Meinung, eine bekannte oder maschinenlesbare DSL ist durchaus von Vorteil. Daher verwendet man bei dieser Kategorie beliebige, definierte Formate wie XML, JSON, YAML, etc.

In diese Kategorie fallen dann natürlich Projekte wie Ant oder auch das Dependency Injection Framework von Spring. Im Zusammenhang von Ant, XML und DSLs weist hier der Blog aus dem Teil der Lerneinheit Funktionale Programmierung wieder interessante Zusammenhänge auf . Ant ist ja eigentlich ausführbares XML. Bei Ant werden die XML-Werte Java übergeben und dann ausgeführt.

Vorteile

- Es gibt viele Tools / Viewer.
- Kann von Anwendungen, Libs, Datenbanken etc. eingelesen und verarbeitet werden.

Letzteres dann auch für OLAP, Data-Mining, Predictive Analytics, etc.

11.8.3 DSL Workbench

Normalerweise ist der Ablauf bei externen DSLs wie folgt:

1. Einlesen der externen Datei
2. Parsen der Datei

3. Syntaxanalyse und Überführung in einen AST (Abstract Syntax Tree)
4. Transformation des AST in das Zielformat

Arbeit machen dabei natürlich immer die Schritte 2 und 3. Schön wäre doch das Folgende:

- Eine DSL-IDE steht bereit. In diese muss nur noch die DSL eingegeben werden.
- In einem anderen Fenster kann man leicht die Information zum Parsen (d.h. die Definition des ASTs) eingeben.
- Den Rest macht dann die DSL-IDE.
- Zusätzlich kann die DSL-IDE noch Hilfestellung im Editor geben, da sie ja die Definition des ASTs kennt (analog zu einem *rich-editor*).

Bei letzterem Punkt sind dann Dinge wie Validierung, Code-Assist (Strg+Space), farbliche Hervorhebung, Outline etc. gemeint. Gut wäre es natürlich noch, wenn das auch als Bibliothek exportiert werden könnte, so dass eine Nutzung via Bibliothek außerhalb von der IDE möglich wäre.

Es wäre fein wenn es so etwas geben würde. Und so etwas gibt es tatsächlich.

Xtext aus dem Eclipse Projekt ist so ein Beispiel . Mit einer gelungenen Eclipse Integration editiert man hier schnell und einfach die Grammatik, mit der der Abstract Syntax Tree aufgebauf werden kann.

Unter DSL-Workbench sind daher solche Tools / Editoren gemeint, die den Nutzer in der oben genannten Art und Weise unterstützen.

Xtext wird ein Schwerpunkt dieser Lerneinheit sein und auch eine entsprechende Übung dazu beinhalten.

Die nachfolgende Abbildung zeigt den Editor, in welchem beispielhaft die Grammatik für Xtext definiert wird.

```
DomainModel.xtext
4
5  Domainmodel:
6      elements += Type*
7  ;
8
9  Type:
10     DataType | Entity
11 ;
12
13 DataType:
14     'datatype' name = ID
15 ;
16
17 Entity:
18     'entity' name = ID ('extends' superType = [Entity])? '{'
19         features += Feature*
20     '}'
21 ;
22
23 Feature:
24     many?='many'? name = ID ':' type = [Type]
25 ;
26
27
```

Eine einfache DSL Grammatik

(C) http://www.eclipse.org/Xtext/documentation.html

11.8.4 Parser Generatoren

Parser Generatoren nehmen ebenfalls eine Grammatik, lesen diese ein und produzieren dann Code, der

- diese Grammatik beinhaltet
- die DSL lesen kann
- diese lexikalisch und syntaktisch verarbeitet
- und die entsprechenden Ausgaben vornehmen kann.

Das Ergebnis der Programmierung von Parser Generatoren ist wiederum ausführbarer Code oder eine Bibliothek. In diesem Fall ist es dann sogar möglich, Code in der Zielsprache mit Code der Laufzeitumgebung zu mischen. Oder sich vom Program aus in den Parserzyklus einzuklinken und bei entsprechenden Events eigene / zusätzliche Aktionen auszuführen.

Man spricht hierbei auch von Compiler Compilern. Die bekanntesten sind:

1. ANTLR
2. Lex und Yacc

Letztere sind bereits recht alt und schon seit längerem in UNIX enthalten.

ANTLR ist ein ebenso berühmtes Werkzeug von Terence Parr (ANother Tool for Language Recognition). Es ist Open-Source und in Java geschrieben. Es enthält Schnittstellen, um C, C++, Java, JavaScript, Objective-C und Python zu generieren.

Wir werden uns ANTLR später genauer anschauen und ggf. üben.

Vorteil

- Sehr mächtig

Nachteil

- Recht kompliziert

Hinweis: In vielen Hochsprachen gibt es Berge von Parsern, die sehr mächtig sind. In Clojure z.B. Instaparse (Beispiele anschauen und einfügen!).

11.8.5 Parser Combinators

Die Idee eines Parser Combinators ist, das mächtige Typsystem einer Sprache auszunutzen, um sich quasi ein ANTLR äquivalent selbst in der Sprache zu definieren. Dazu wird das Folgende getan:

- üblicherweise werden viele einzelne Parser definiert
- diese werden aus den einzelnen String Definition für die Syntax gebildet
- all diese kleinen Parser werden dann meistens (zumindest in modernen / expressiven Sprachen) an eine Higher-Order Funktion gegeben, die daraus einen Gesamtparser (eine Funktion) erzeugt.

Man erkennt hier schon, dass dies eine mächtige Domäne der funktionalen Sprachen ist. Diese sind aber nicht zwingend dafür nötig.

Das Ergebnis dieses Prozesses ist dann der Parse-Tree, d.h. der gefüllte AST.

Vorteil

- Man ist wieder in der eigenen Sprache.

Nachteil

- Nicht in allen Sprachen geht das elegant und leicht.

Beispiele:

- Scala: Siehe Spiewak Artikel unten

- Clojure: Imparsonate:
- Clojure: parsley: https://github.com/cgrand/parsley
- F#: FParsec
- Python: Pyparsing
- Java: Jparsec
- JavaScript: JsParse

Weiterführende Lektüre:

- Daniel Spiewak, ***The Magic Behind Parser Combinators***, => **TIP!**
- Debasish Gosh, ***External DSLs made easy with Scala Parser Combinators***,

11.9 Wann welche DSL?

Bisher haben wir ca. sieben Kategorien an DSL kennengelernt. Dazu kommen viele Frameworks, die DSLs unterstützen. Umso schwerer ist die Frage, wie man zu einer Entscheidung kommt, welches DSL eingesetzt werden soll.

Hier gibt es viele Kriterien für die Entscheidungshilfe:

- **Wer ist der Nutzer?**: Kann man dem Anwender die Eingabe in einer Programmiersprachensyntax zumuten oder nicht? Braucht er eine eigene IDE? Kann der Anwender einen einfachen Texteditor verwenden oder muss er dies sogar? Wer ist überhaupt der Nutzer der DSL. In der Regel sind dies meistens entweder Entwickler oder informatikfremde Spezialisten (wie z.B. Börsenhändler), mit jeweils unterschiedlichem Vorwissen in Bezug auf die Basis und den Aufbau einer DSL.

- **Existierende Infrastruktur / Wissen nutzen**: Oftmals werden in Firmen beispielsweise schon Sprachen eingesetzt, die für DSLs wunderbar genutzt werden können wie Scala, Clojure, Ruby, Groovy, Python. Es gibt viele weitere Beispiele: Wird z.B. überall Eclipse genutzt, so kann man eine externe DSL auf dieser Basis verwenden. Soll die DSL in ein Produkt eingebaut werden (in Java?!), so empfiehlt sich beispielsweise ANTLR. Tendenziell wird für externe Personen auch eher eine externe DSL verwendet, weil sie komplett neu definiert und zu 100% auf die externe (sprachenunwissende) Person zugeschnitten werden kann. Interne Entwickler hingegen könnten eher an einer internen DSL Gefallen finden, da sich diese nahtlos in die Programmiersprache integriert. Es muss daher sowohl das Wissen der Anwender als auch der Entwickler der DSL berücksichtigt werden.

- **Expressiveness und Lernkurve**: Es gilt natürlich immer den Aufwand für die Erstellung und den Aufwand für das Lernen des Anwenders zu Analysieren. Es lohnt natürlich nicht, die Entwickler in eine interne Haskell DSL zu zwingen, wenn alle fit in Groovy sind. Es loht auch nicht jemanden auszubilden, wenn überhaupt kein Programmierwissen vorhanden ist. Hier bietet sich wieder eine Workbench an. Auf der anderen Seite gilt es beim Anwender natürlich das Gleiche zu betrachten. Ein unerfahrener Anwender kann keine komplexe interne DSL lernen. Er braucht eine triviale externe DLS, mit hoher Expressiveness, die seiner Domäne bestens entspricht (siehe der Bauer).

Wie üblich sollte man alle diese Kriterien in eine Matrix packen und dann eine Aufwand / Nutzen Analyse durchführen.

11.10 DSLs mit Xtext

Xtext ist ein Projekt, welches unter dem Eclipse Projekt gehostet wird. Es ist ein Teil des Eclipse EMF Projektes .

Xtext definiert sich selbst als Framework für die Entwicklung von DSLs und Programmiersprachen. Ein Aspekt auf den wir bisher noch nicht eingegangen sind. Natürlich können viele mächtige (externe) Werkzeuge wie ANTLR oder Xtext auch sehr gut dafür verwendet werden, komplett neue Programmiersprachen zu erstellen.

Xtext ist ein relativ junges Projekt (2006), welches versucht sich, durch einige zusätzliche Features von Platzhirschen wie ANTLR, abzusetzen.

- Parser, Compiler und Interpreter
- ein komplettes Klassen / Objektmodell als AST
- komplette Integration in Eclipse:
  - Farbliche Syntaxhervorhebung
  - Code Completion (Automatische Vervollständigung) mit Str+Space
  - Validierung der Grammatik
  - Quick-Fixes, d.h. Verbesserungsvorschläge
  - komplette Navigation und Integration in Eclipse wie bei Java
  - und vieles mehr

Hinweis

Übrigens wurde Xtext maßgeblich von deutschen Entwicklern und Firmen entwickelt, und es gab zudem Unterstützung durch deutsche Forschungsgelder.

Intern verwendet Xtext ebenfalls ANTLR. Die von Xtext erstellten Artefakte können aber komplett unabhängig von Eclipse als Bibliothek verwendet werden.

11.10.1 Xtext Grundlagen

Grundlage der Arbeit mit Xtext und ANTLR ist das Erstellen einer Grammatik. Sowohl in Xtext, als auch in ANTLR wird auf die EBNF zurückgefriffen. Die EBNF (Erweiterte Backus-Naur-Form) ist eine Metasprache, um kontextfreie Grammatiken darzustellen.

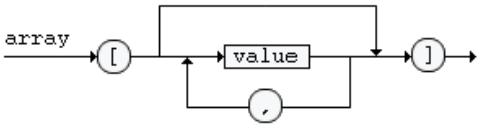
Definition

Wir erinnern uns: In dem Gebiet der formalen Sprachen ist definiert, dass ein Nichtterminalsymbol immer nur genau auf eine Menge von Nichtterminal + Terminalsymbolen abgeleitet wird. Üblicherweise schreibt man
V -> w
. Die Ableitungsregel selbst hängt aber nur davon ab, ob das Nichtterminalsymbol V im Eingabestrom überhaupt vorkommt oder nicht. Es hängt dagegen nicht vom Kontext ab, welche Regel angewendet wird. Die Regeln sind also immer eindeutig:
A -> Pferd, B -> Haus, etc
. Dagegen hängen kontextsensitive Grammatiken dementsprechend immer vom Kontext ab. Hiermit haben z.B. Übersetzer in unserer Sprache wie Google Translate zu kämpfen. Der Satz "man muss die freien Radikale im Mikroskop beobachten" ist sicher schwer zu übersetzen, da der Kontext von umgebenden Wörtern abhängt (hier das Mikroskop für Chemiker). Für Verfassungsschützer (und ohne die Wörter "im Mikroskop") würde man dies eventuell anders übersetzen.
Zur Erinnerung:

- **Typ 3**: Reguläre Sprachen
- **Typ 2**: Kontextfreie Sprachen
- **Typ 1**: Kontextsensitive Sprachen
- und **Typ 0** / alle anderen Sprachen

Anmerkung

Bekannt für die Notation von Grammatiken sind auch die aus den Programmiersprachen bekannten Syntaxdiagramme. Dazu hier ein Beispiel für die Definition von JSON: Das array beginnt mit einer öffnenden eckigen Klammer und wird mit einer schließenden Klammer geschlossen. Die einzelnen Werte werden durch ein Komma getrennt:

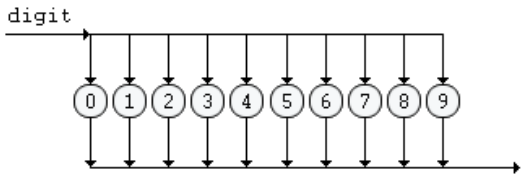


Syntaxdiagramm von array

Üblicherweise definiert man sich für DSLs, Programmiersprachen oder Datenformate wie JSON erst einmal einen Sprachraum:

number = ["-"], digit, { digit } ;
string = '"' , { alphabetic character }, '"' ;
identifier = alphabetic character, { alphabetic character | digit } ;
alphabetic character = "a" | "b" | "c" | "d" | "e" | "f" | "g"
| "h" | "i" | "j" | "k" | "l" | "m" | "n"
| "o" | "p" | "q" | "r" | "s" | "t" | "u"
| "v" | "w" | "x" | "y" | "z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

In der nachfolgenden Grafik sehen Sie exemplarisch das digit-Syntaxdiagramm:



Syntaxdiagramm von digit

Der oben stehende Sprachraum erlaubt nur Kleinbuchstaben, legt aber den Grundstein für viele Symbole in einer DSL oder Programmiersprache: Zahlen, Strings und Bezeichner. Sie bekommen jetzt sicherlich auch ein Gefühl dafür, wie man noch fehlende Dinge (= Regeln) einfügen könnte. Beispielsweise die Syntax für ein Import-Statement oder die Definition einer Klasse in Java
(visibility class identifier "{" content "}"
.

Den Standard kann man auch hier einsehen, um die Syntax der Grammatikbeschreibung näher zu verstehen: . Zum Einstieg nähern wir uns dem ganzen jedoch mit einem einfachen Beispiel.

Referenzen:

- Hilfe wie man EBNFs schreibt:
- EBNF Grammar:

11.10.2 Beispiel mit Xtext

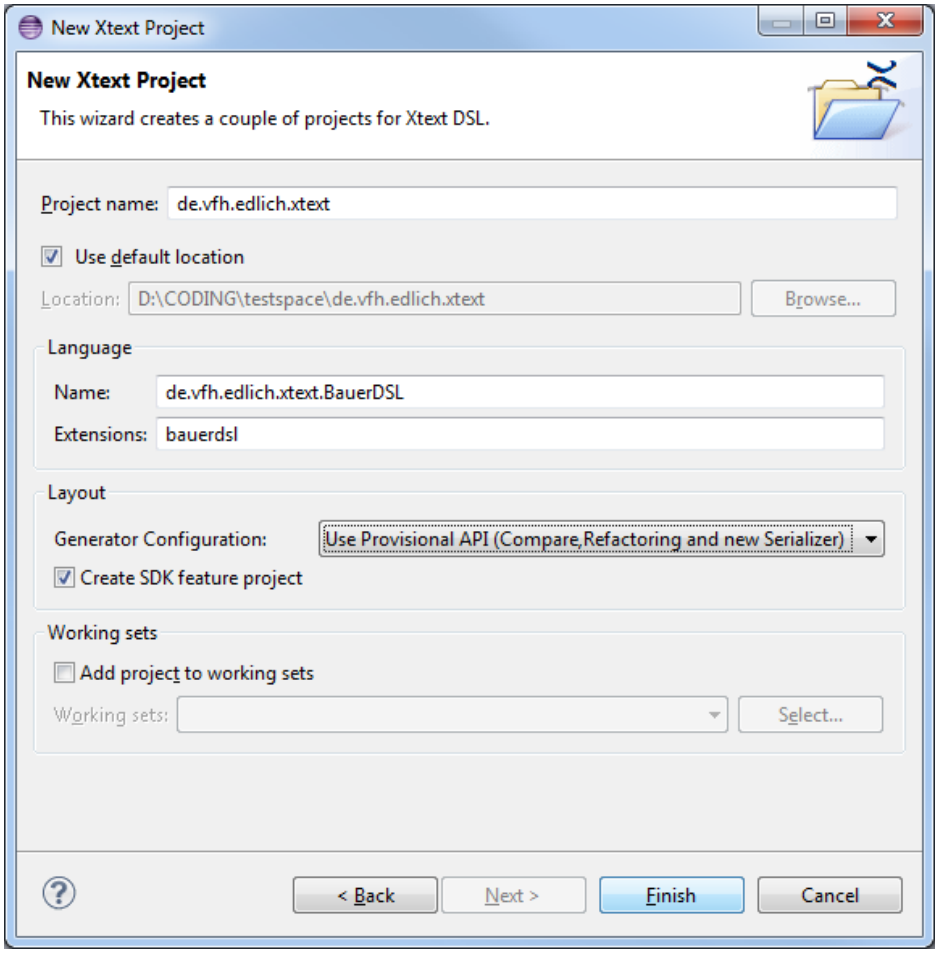
Im Folgenden wird ein einfaches Beispiel mit Xtext beschrieben. Nach einem ersten Test der Laufzeitumgebung (wie er auch in der Dokumentation beschrieben ist), werden wir eine eigene kleine DSL erstellen.

Installation

Die Installation von Xtext kann wie folgt vorgenommen werden:

1. Über Eclipse Marketplace (falls Sie Eclipse Marketplace noch nicht installiert haben finden Sie hier eine kurze Anleitung: Eclipse Marketplace). Hier ist allerdings unklar, mit welchen Eclipse Versionen Xtext erfolgreich zusammenarbeitet. Daher kann hier das Update manchmal durchaus scheitern *(conflicting ressources)*.
2. Alternativ gibt es Pre-Build Distributionen für alle gängigen Plattformen in 32 und 64 Bit. Dies sollte in der Regel klappen, ist allerdings fast 1/4 GB groß.

Nun muss ein neues Projekt erstellt werden. Mit
New | Project | Xtext Project
geht dies analog zum ersten Bild unten relativ einfach:



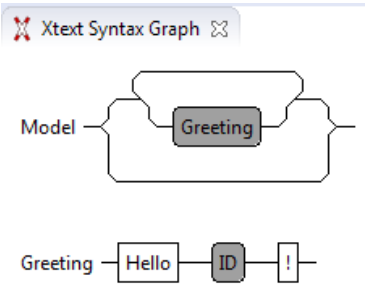
Xtext Projekt
Erstelle ein Xtext Projekt

Hier ist die Extension die spätere Dateiendung der DSL-Datei, die dann von Eclipse / Xtext geparst werden soll.

Er erscheint (hoffentlich) sowohl eine ganz einfache Beispiel EBNF und ein Diagramm der Syntax (
Window | Show View | Others | Xtext > Xtext Syntax Graph
):

```
BauerDSL.xtext
grammar de.vfh.edlich.xtext.BauerDSL with org.eclipse.xtext.common.Terminals

generate bauerDSL "http://www.vfh.de/edlich/xtext/BauerDSL"

Model:
    greetings+=Greeting*;

Greeting:
    'Hello' name=ID '!';
```

Xtext
Die EBNF Definition

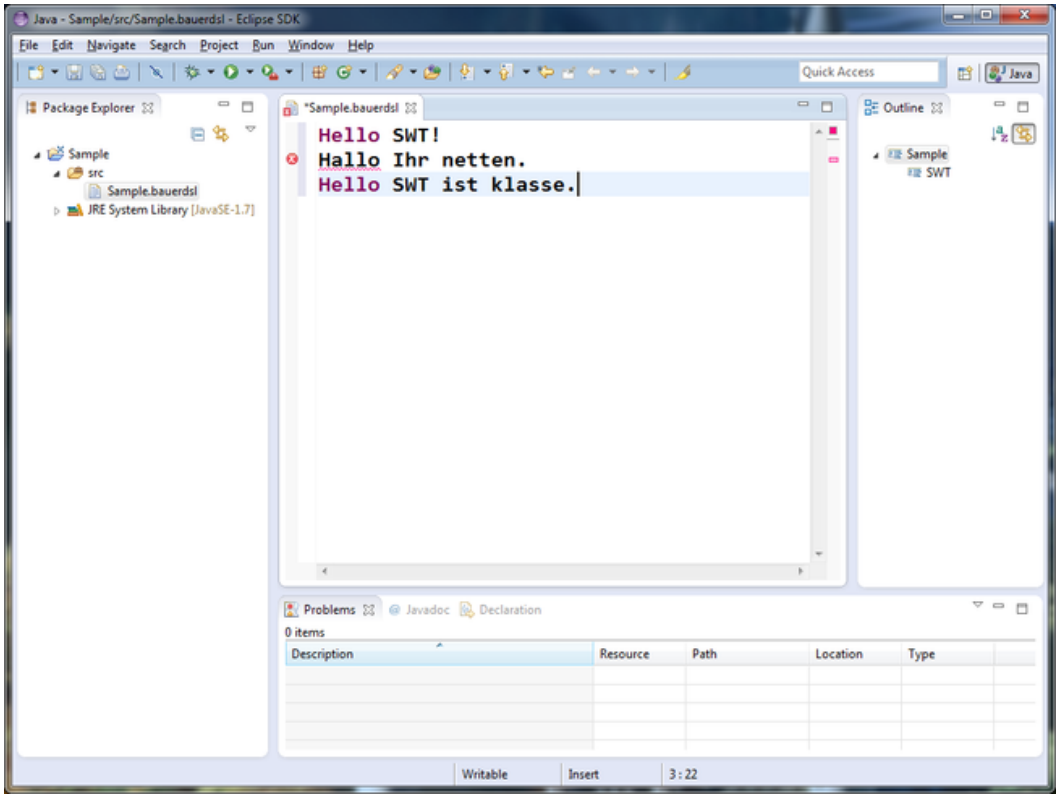


Xtext
Die EBNF als Diagramm

Danach soll das ganze getestet werden. Dazu müssen alle ANTLR Module, das Encore-EMF Modell und die Eclipse Editoren generiert werden. Wir generieren daher die *language infrastructure* über
Run As | generate Xtext artifacts
. Gegebenenfalls möchte Xtext hier ANTLR 3 haben: Dies wird im Konsolen-Fenster erfragt und automatisch nachgeladen!

Wenn alles generiert ist, kann man auf dem Projekt oben mit der rechten Maustaste das Projekt als neues Eclipse Projekt starten (
Run as | Eclipse Application
). Das bedeutet, es wird eine neue Eclipse Instanz gestartet (Vorsicht, das ist Ressourcenhungrig. Den RAM und die Console in dem ersten Eclipse beachten!). Diese neue Instanz hat dann die Bibliotheken für die DSL mitbekommen.

Das bedeutet man kann jetzt ein beliebiges Projekt (z.B. Java) erstellen. Im src Verzeichnis erstellt man jetzt eine Datei mit der Endung, die man beim Erstellen der DSL angegeben hat. Hier *bauerdsl*. Also z.B.
Beispiel.bauerdsl
.



Xtext
Parsen im Editor

Hier sieht man bereits das Folgende:

- Die Datei mit der Endung
  - .bauerdsl

  wird als DSL Datei erkannt
- Es erfolgt eine farbliche Hervorhebung von Schlüsselwörtern wie *Hello*.
- Die erste Reihe entspricht der EBNF Grammatik und wird als korrekt erkannt.
- Die zweite Reihe ist fehlerhaft. Gemäß der Grammatik ist *Hallo* kein Schlüsselwort.

Nach Beseitigung des Fehlers in der zweiten Reihe würde dann auch die dritte Reihe als fehlerhaft erkannt, da auf die ID kein Ausrufezeichen folgt.

11.10.3 Eine eigene DSL

Im Folgenden soll eine etwas komplexere DSL erstellt werden. Dazu nehmen wir einfach eine ausgebaute DSL für landwirtschaftliche Betriebe, die bereits in Kapitel 7.8 (Formen externe DSLs) angedeutet wurde.

Es soll in einer eigenen Datei bzw. im Editor der Workbench die folgende Eingabe möglich sein:

kaufe 20 Schweine max 2000 €
verkaufe 50 Hühner min 250 €

Er soll also kaufen und verkaufen können. Weiterhin muss die Anzahl angegeben werden. Danach folgt die Tierart, wobei wir nur Schweine, Kühe, Hasen und Hühner erlauben. Beim Kauf muss dann das Schlüsselwort 'max' folgen. Umgekehrt natürlich 'min' für den verkauf. Und abschließend folgt die Geldsumme in Euro aber ohne Cent. Kauft oder verkauft er nur ein Tier, dann ist dennoch die Mehrzahl (z.B. Schweine) einzugeben.

Der letzte Absatz ist die umgangssprachliche Beschreibung einer EBNF, die leider meistens nicht eindeutig in eine eben solche hineingegossen werden kann. Dennoch sähe eine mögliche EBNF Definition so aus:

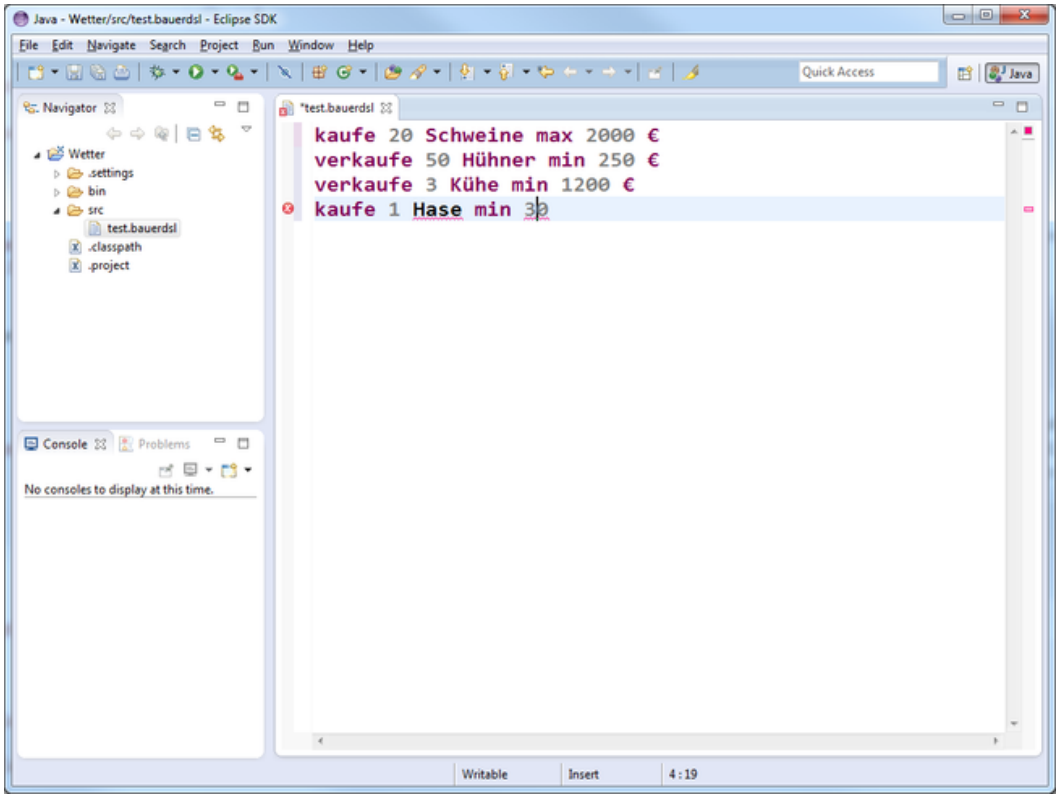
```
Bauerdsl.xtext
grammar de.vfh.edlich.xtext.Bauerdsl with org.eclipse.xtext.common.Terminals

generate bauerdsl "http://www.vfh.de/edlich/xtext/Bauerdsl"

Model:
    trade+=Trade*;

Trade:
    buyTrade | sellTrade;

buyTrade:
    'kaufe' INT animal 'max' INT '€';

sellTrade:
    'verkaufe' INT animal 'min' INT '€';

animal:
    'Kühe' | 'Schweine' | 'Hasen' | 'Hühner';
```

Xtext
Definition der Bauer-DSL

Was passiert hier:

- In der ersten Zeile wird der *default lexer* von Xtext verwendet.
- Die zweite Zeile bewirkt, dass das Metamodell für die bauerdsl generiert wird.
- Danach folgen Rules = Regeln
- Als erstes die üblicher enumeration Regel: Es folgen beliebige *Trades*, d.h. Handelsanweisungen
- Ein Trade selbst besteht aus einem Kauf oder einem Verkauf.
- Ein Kauf ist so aufgebaut: 'kaufe Stückzahl Tier max Preis €'. Wobei das Tier = animal wieder eine Regel ist und Stückzahl und Preis integer Zahlen sein müssen.
- Ein Tier (animal) ist eines der vier erwähnten Tiere.

Nach dem generieren der Xtext Artefakte kann die zweite Eclipse Workbench gestartet werden. Dort sind nun die folgenden Eingaben in einer Datei mit der Endung *.bauerdsl möglich:



Xtext
Eingabe der landwirtschaftlichen Trades

Die obige Datei zeigt, dass Eingaben der Grammatik möglich sind. In Zeile vier erkennt er ebenfalls drei Fehler (falls die vorigen beseitigt sind):

1. Man kann keinen 'Hase' kaufen. Nur 'Hasen'.
2. Ein kauf muss weiterhin mit 'max' ausgezeichnet sein.
3. Und schliesslich muss ein Trade mit einem '€' abgeschlossen werden.

Anmerkung

Es sei angemerkt, dass auch der Modulautor viele Abstürze und Eclipse-Aufhänger zu verkraften hatte. Es scheint sich um ein großes komplexes System zu handeln, welches viele Konfigurationen beinhaltet, stark RAM abhängig ist und anscheinend sehr viele Zustände einnehmen kann. Gute Nerven helfen hier ;-)

Abschließend wäre natürlich noch interessant, wie wir den erzeugten Code verwenden, um nicht nur zu parsen, sondern auch zu generieren. In obigem Beispiel wurde ja nur geparst und daher die Syntax erkannt. Intern steht aber durchaus ein AST zur Verfügung mit dem man extern arbeiten kann.

11.11 DSLs mit ANTLR

Referenzen:

- http://www.antlr.org/
- Terrence Parr, "The definitive ANTLR 4 Reference", Pragmatic Bookshelf, 2013
- http://javadude.com/articles/antlr3xtut/

ANTLR wird seit 1989 von Terence Parr entwickelt (University of San Francisco). Übersetzt bedeutet es 'AN'other 'T'ool 'L'anguage 'R'ecognition.

ANTLR kann aus grammatik / parsing Informationen einen Lexer und einen Parser für kontextfreie Grammatiken LL(k) generieren. Die ANTLR Anwendung selbst ist in Java geschrieben und frei zum download verfügbar (BSD Lizenz).

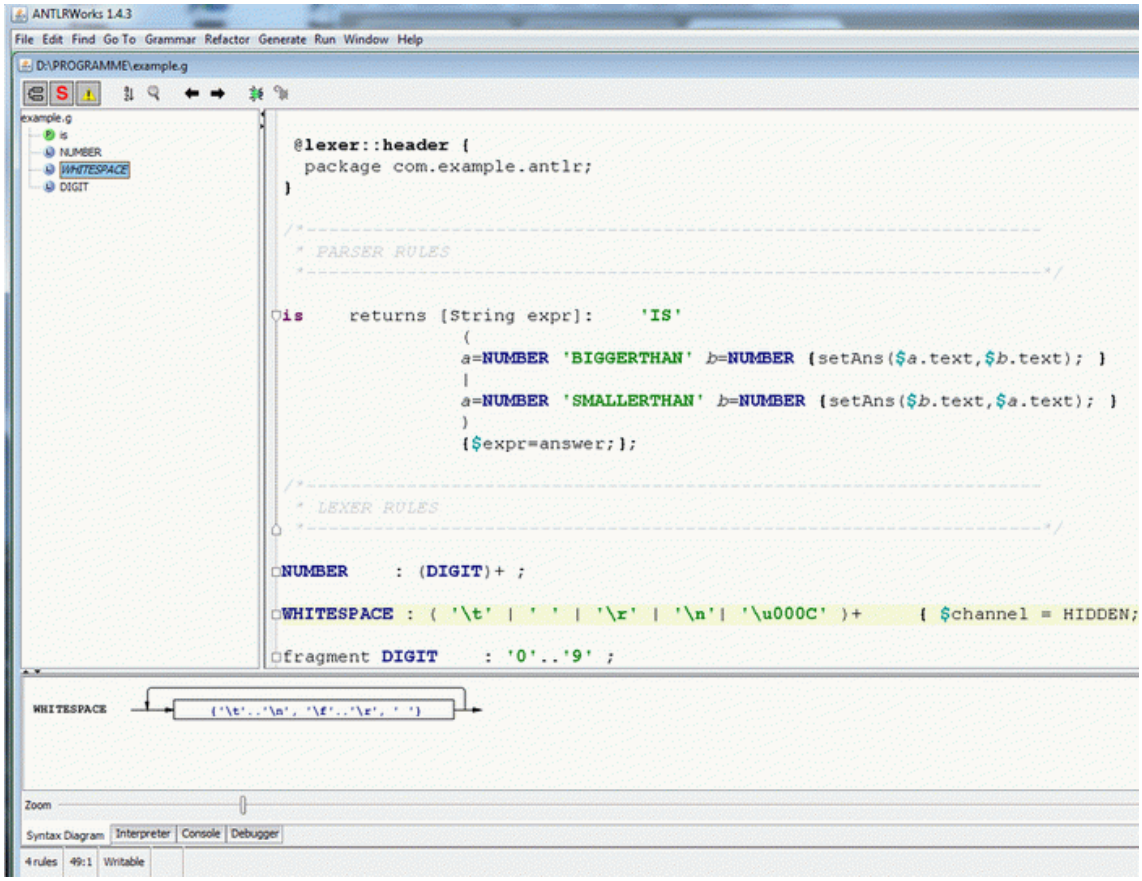
ANTLR selbst wird in unzähligen Tools und Frameworks eingesetzt wie z.B. Hibernate. Es unterstützt als Zielsprache die folgenden Programmiersprachen: Ada, ActionScript, C, C++, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby und ML.

Der Ablauf ist daher immer der Folgende:

- Der **Lexer** übernimmt die lexikalische Analyse und reicht die Tokens weiter
- an den **Parser**. Dieser kann nun entweder **direkt Ausgaben** erzeugen oder
- den **AST** (Abstract Syntax Tree) zur Verfügung stellen.
- Mit diesem AST kann man dann selbst leicht einen **Tree Walke** zur Verfügung stellen, und den Baum selbst entlanglaufen und zur Laufzeit oder dynamisch entscheiden, ob und was ausgegeben / generiert werden soll.

ANTLR liest also selbst initial wie bei Xtext auch nur ein Stream von Characters und erstellt dann daraus sogenannte 'Tokens', die in dem AST abgelegt werden.

Unter dem Namen ANTLRworks gibt es auf antlr.org ebenfalls eine GUI zu finden, die den Prozess der Grammatikerstellung unterstützt.



ANTLRworks
Graphisch Grammatiken erstellen

Hinweis

Es gibt natürlich noch viele weitere Parser und Generatoren, die hier nicht weiter erläutert werden können wie: Bison (oder die alte LEX / YACC Familie aus UNIX), COCO, Rats! aus XTC u.s.w.

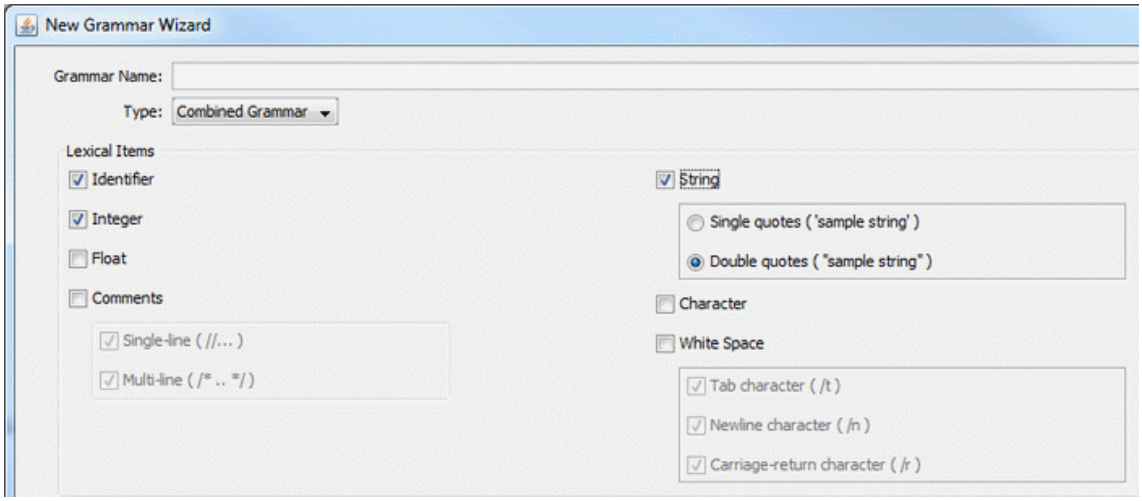
11.11.1 Einfaches ANTLR Beispiel

Installation:

ANTLR gibt es als kompaktes und einzelnes jar zum Download: z.B. antrl-3.4-complete.jar.

Grammatik:

Der ANTRLworks Wizard kann einem bei der Definition initial einiges an Handarbeit abnehmen:



ANTLR works Wizard
Definition der Grundtypen

Erstellt wird dann das Grundgerüst der Grammatik für die Typen, die man verwenden möchte. In unserem Fall Identifier, Integer und Strings:

```
grammar EdlichsTest;

ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

INT :   '0'..'9'+
    ;

STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```

ANTLR works Grammatik
Generierung Grundtypen

Angepasste Grammatik:

Eine einfache Grammatik würde nun so aussehen:
grammar SimpleCalc;
add : NUMBER PLUS NUMBER;
NUMBER : ('0'..'9')+ ;
PLUS : '+';

Offensichtlich kann diese nur einfache Additionsausdrücke lesen.
> java -jar antlr-3.4-complete.jar SuperSimple.g
Erstellt die Dateien:
* SuperSimple.token
* SuperSimpleLexer.java
* SuperSimpleParser.java

Diese lassen sich jetzt einfach in Eclipse importieren und übersetzen. Dazu muss man nur die ANTLR Bibliothek in den Classpath nehmen.

Nun kann mit einem Java Programm auf den Lexer und den Parser zugreifen:

```
package de.edlich.atest;

import org.antlr.runtime.ANTLRFileStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;

public class RunSuperSimple {

 public static void main(String[] args) throws Exception {
      SuperSimpleLexer   lex   =   new   SuperSimpleLexer(new
 ANTLRFileStream(args[0]));
 CommonTokenStream tokens = new CommonTokenStream(lex);

 SuperSimpleParser parser = new SuperSimpleParser(tokens);

 try {
```

```
 parser.expr();
 } catch (RecognitionException e) {
 e.printStackTrace();
 }
 }
}
```

Java Zugriff
Mit dem Parser in Java arbeiten

Schreibt man den nachstehenden Code gleich so in das *.g File, dann ist dieser gleich im Parser enthalten und kann sofort gestartet werden:

```
 @members {
 public static void main(String[] args) throws Exception {
    SimpleCalcLexer    lex    =    new    SimpleCalcLexer(new
 ANTLRFileStream(args[0]));
 CommonTokenStream tokens = new CommonTokenStream(lex);

 SimpleCalcParser parser = new SimpleCalcParser(tokens);

 try {
 parser.expr();
 } catch (RecognitionException e) {
 e.printStackTrace();
 }
 }
}
```

Java Zugriff
Java direkt aus .g generieren lassen

Die Methode, die der Parser dann aufruft ist keine API Methode, sondern entspricht dem Namen der *Rule*, die aufgerufen werden soll!

11.12 Zusammenfassung DSL

- Sie wissen nun, dass eine DSL eine Programmiersprache ist, die einen kleinen Problemraum abdeckt, und dass Syntax und Semantik möglichst auf der gleichen Abstraktionsebene sind, wie das Problemfeld selbst.

- Des Weiteren haben Sie die Vorteile (z.B. DSLs sind präzise und lassen keinen Deutungsspielraum) und Nachteile (z.B. eine gute DSL zu designen ist schwer und kostet Zeit und Geld) von DSLs kennengelernt.
- Es gibt zwei Arten von DSLs:
  - externe DSL (beinhaltet komplett ihren eigenen Sprachraum)
  - interne DSL (ist in eine Programmiersprache eingebettet)
- Des Weiteren unterscheidet man zwei Formen der internen DSLs:
  - Embedded DSLs (Smart APIs, AST Manipulation, Eingebettete Typen, Reflektive Metaprogramming)
  - und Generative DSLs - hier wird unterschieden zwischen Runtime Metaprogramming und Compile Time Metaprogramming
- Bei den externen DSLs gibt es folgende Formen:
  - String Manipulation: Strings einlesen und mit Hilfsmitteln bearbeiten.
  - Formate wie XML, JSON Transformieren
  - DSL Workbench: in die DSL-IDE wird die DSL und die Informationen zum Parsen eingegeben. Es gibt zusätzliche Hilfestellungen z.B. im Editor.
  - Parser Generatoren: die Grammatik wird eingelesen, Code wird produziert, das Ergebnis ist ausführbarer Code oder eine Bibliothek.
  - Parser Combinators: viele einzelne Parser werden definiert, daraus kann mittels Higher-Order Functions ein Gesamtparser generiert werden.
- Als Kriterien für die Entscheidungshilfe, wann Sie welche DSL nutzen sollten wurden Ihnen folgende Punkte vorgestellt:
  - Wer ist der Nutzer (Entwickler oder informatikfremde Spezialisten)
  - Vorhandenes Wissen/Infrastruktur
  - Lernaufwand
- Sie kennen Xtext und ANTLR

11.13 Wissensüberprüfung DSL

11.14 Aufgaben

Initialer Text

11.15 DSLs Literatur

Es gibt derzeit (Stand 2012) nur zwei relevante Literaturreferenzen:

Debasish Gosh

"DSLs in Action"

Manning Inc., ISBN-13: 978-1935182450

Martin Fowler

"Domain Specific Languages"

Addison Wesley Longman, ISBN-13: 978-0321712943

Das Manning Buch ist recht praktisch und betrachtet mit Scala und Clojure mehr Programmiersprachen. Besonders der Appendix ist interessant.

Das Martin Fowler Buch will eher eine komplette Referenz sein und bespricht auch die entferntesten Patterns. Es beschränkt sich weiterhin auf Java, C# und Ruby. Clojure und Scala werden nicht erwähnt.

Weitere Weblinks:

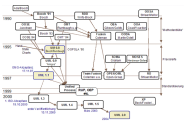
- Xtext Support für Eclipse
- Growing a DSL with Clojure

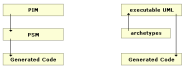
xtc / Rats! merken!

I Literaturverzeichnis

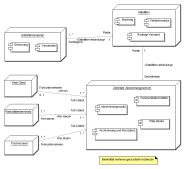
Kam10 Kamphausen, Kaiser. *Clojure: Grundlagen, Concurrent Programming*. dPunkt, 2010. isbn:9783898646840

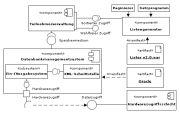
II Abbildungsverzeichnis

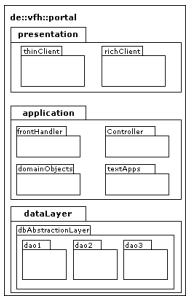

History of the UML 16



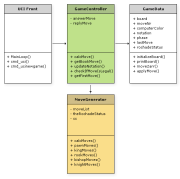
Model Driven Architecture (MDA) 20


Verteilungsdiagramm des Systems Toll-Collec 23

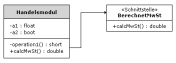

Beispiel Komponentendiagramm 26



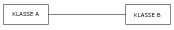
Paketdiagramm 28


Example Class Diagram 30

A Class in the Class Diagram 31


Interface Example 32

Interface as a Circle and a Semicircle 33


Assoziation 34

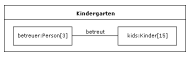
Qualified Association 35

Directed Association 35

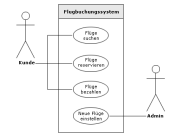
Aggregation 36

Composition 36

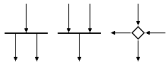
Inheritance 37


Composition Structure Diagram 37

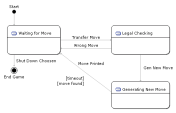
Object Diagram 38


Anwendungsfalldiagramm....39

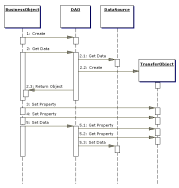
Aktivitätsdiagramm....41


Kontrollknoten....42

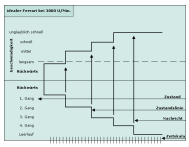
Signale....43


State Diagram....43

Diagrammelemente....45


Sequence Diagram....45

Communication Diagram....47

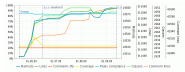

Timing Diagram....48

RegExcp....55

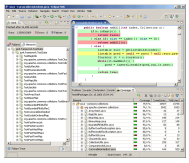
RegExps....57

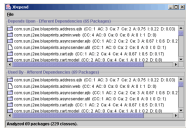
Tools....61

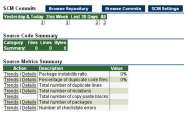
Analysis of Basic Metrics....74

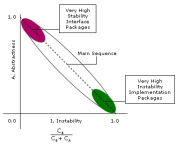

Basis Timeline....74

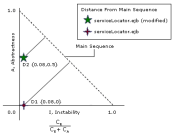
Code-Coverage Report....80

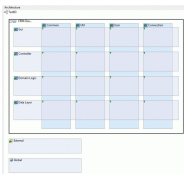

Coverage-Metrik in Eclipse....80


Ausgabe der JDepend-Metriken....88

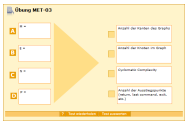

Measurement of the Instability....93


Main Sequence Graphics....93

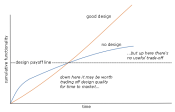

Distance from Main Sequence....93


Definition of an architekture....97

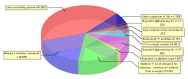
Sonar - Calculation of development days.... 102

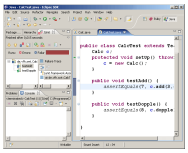

....109

....110

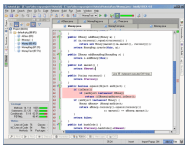

Comparison of two projects: 'no design' und 'good design'....115

Technical Debt Quadrant....116

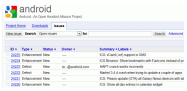

Percentage distribution of error messages....133


Testen mit JUnit....134

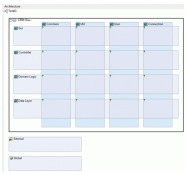
Example Report by Clover....136


Example Report by Clover....136

Dependencies for a component to be tested....139


Issue Tracking System....142

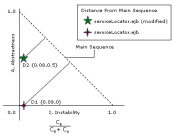
Sonar - Calculation of Man-Days....145


Definition of an Architecture....149

Sonar....149

Dependency....150

cycles....151

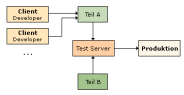

JDepend....152

App: Pomodoro....155

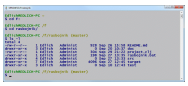
Radar....164

File-Structure....176

Java Template....176



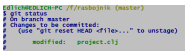
Repo-Architecture....191


Use and output of Unix commands....192

Git Stages....195

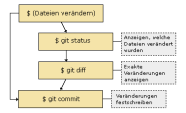
Status I....198

Status II....198

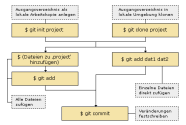

Status III....198

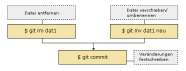
Diff I....198

Diff II....199

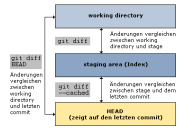

Git: change files....199

git add -i 201


Basiskommandos Git....201

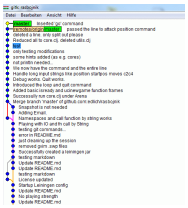

Git: remove or move files....203

Diff cached....204



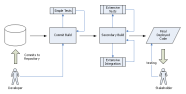
diff-Kommandos im Vergleich....205

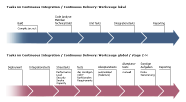
Log....206


Graphical visualization with Gitk....208

Feedback too late....221

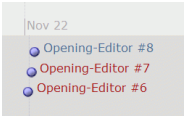
Feedback-Status....222


Stagebuild....225

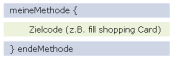

Process Chain of the CI....227

Jenkins 230

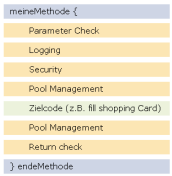
Result 232



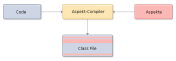
Timeline of all Builds 232

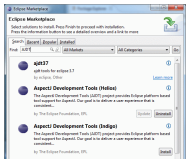


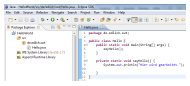
Einfache Methode 238

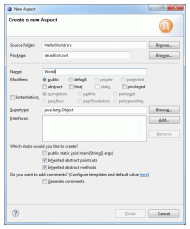


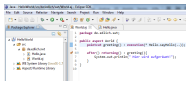
Verschmutzte Methode 239

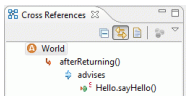

Die Funktionsweise von AOP 241


Installation AspectJ 242


Schritt 1: Ein neues AspectJ Projekt 243

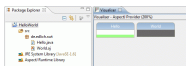

Schritt 2: Aspekt anlegen 243

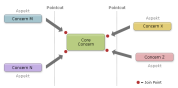

Schritt 3: Hinzufügen des konkreten Aspekts 244



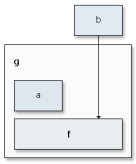
Code und Aspekte 245

Schritt 4: Ausführen 245


Aspect Visualization Ansicht 246

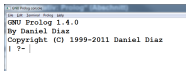

Zusammenspiel von Pointcut, Jointpoint und Core Concerns 250

Programmierparadigmen 260

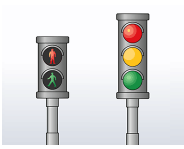


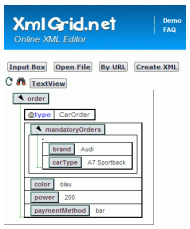
Partiell angewendete Funktionen 293

M. Odersky 299

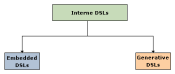


Eingabeaufforderung 313

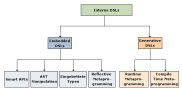

Ampel 325



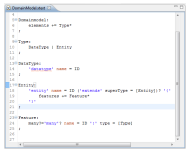
XML 329

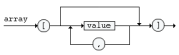


Unterteilung interner DSLs 340

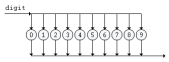

Unterteilung interner DSLs 343

Formen externer DSLs 345

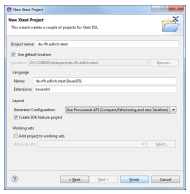

Eine einfache DSL Grammatik 347



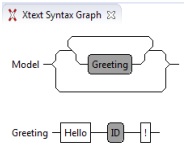
Syntaxdiagramm von array 353



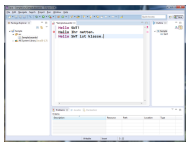
Syntaxdiagramm von digit 353


Xtext Projekt 354

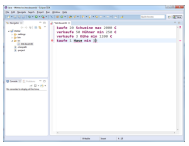
Xtext 355

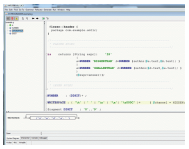


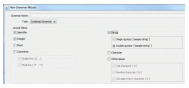
Xtext 356


Xtext 356

Xtext 358


Xtext 359


ANTLRworks 360


ANTLR works Wizard 361

ANTLR works Grammatik 362

III Tabellenverzeichnis

Structure Diagrams....14
Architecture Diagrams.... 14
Behavior Diagrams....15
Interaction Diagrams.... 15
Metazeichen.... 59
Spezielle Zeichen.... 60
Zeichenklassen....60
measure figures....78
....102
Sonar – Cost Calculation....102
Software-Werkzeuge I....106
Software-Werkzeuge II....107
Impact Factors for the Technical Debt Calculation....145
Procee Chain of the CI....226
Formalization....228
Grundtypen in Clojure....272
Java Interop.... 287
Bindungsregeln....311
Kettenregeln....311
Expressiveness von Sprachen....384

IV Aufgabenverzeichnis

- Task UML-01: Specification 50
- Task UML-02: Tools 50
- Übung MET-01 100
- Übung MET-02 109
- Übung MET-03 110
- Übung CCD-01: Diskussion 119
- Übung CCD-02: Discussion „Much helps a lot?“ 135
- Übung CCD-03: Code-Katas 138
- Übung CCD-04: Google „Focus“ 155
- Exercise CCD-05: Shortcuts 156
- Übung CCD-06: Clean Code Development 160
- Übung BUI-01: ANT 185
- Übung BUI-02: Gradle 185
- Übung BUI-03: Maven 186
- Excercise DVC-01: Git Repository 194
- Übung DVC-02: Ein Repository clonen 197
- Übung DVC-03: Differences between commit 199
- Übung DVC-04: Try Git Commands 208
- Übung CID-01: Further Disadvantages? 221
- Excercise CID-02: DVCS Systems 226
- Übung CID-03: CI und CD 235
- Excercise CID-04: Tools for Continous Integration 235
- Excercise CID-05: Tasks in the CI / CD Prozess 235

Übung AOP-01: Aspektorientierte Programmierung....254
Übung PPD-01: Lösung mit Clojure....265
Übung FPR-02: Erstellung einer Clojure Datei....271
Übung FPR-03: doseq und dotimes....286
Übung FPR-04: Clojure....306
DSL-01: Java-Code ausführen und verstehen....334

V Glossar

Abhängigkeiten

Dependencies (Abhängigkeiten)

Dependencies sind Abhängigkeiten zwischen Code oder Modulen. Beispielsweise kann es sein, das die Datei A geändert wurde, und anschließend nicht nur A, sondern auch B neu kompiliert werden muss.

Agiles Modell

Agiles Modell

Das agile Vorgehensmodell versucht mit geringem bürokratischen Aufwand und wenigen, flexiblen Regeln auszukommen. Es wird dabei nur so viel Dokumentation wie nötig verwendet. Das agile Modell ist geeignet für kleine Projektteams und kleine, flexible Systeme.

Analysemodell

Analysemodell

Ein Analysemodell (das Ergebnis der Analysephase) ist ein abstraktes Modell zur Spezifikation eines geplanten Systems oder Produkts, wobei noch keine Einschränkung über die Art der Implementierung enthalten ist.

Analysephase

Analysephase

Zeitraum, in welchem das Problem genau untersucht wird, um die notwendigen Kenntnisse über das Problemumfeld (Domain Knowledge) zu erwerben. Das Ergebnis dieser Phase ist das Analysemodell, welches eine Abstraktion des realen Problems darstellt.

Anforderungen

Anforderungen (requirements)

Anforderungen legen die qualitativen und quantitativen Eigenschaften an ein Softwareprodukt aus Sicht des Auftraggebers bzw. Anwenders fest.

Anwender

Anwender

Als Anwender werden alle Angehörigen einer Institution oder organisatorischen Einheit bezeichnet, die ein Computersystem zur Erfüllung ihrer fachlichen Aufgaben einsetzen. Sie benutzen die Ergebnisse der Anwendungssoftware oder liefern Daten, die die Software benötigt.

Anwendung

Anwendungssoftware (auch Applikationssoftware)

Anwendungssoftware (engl: application software) ist eine Software für die objektkonkrete Nutzung der technischen Mittel zur Bearbeitung eines spezifischen Anwenderproblems. Es sind programmtechnische Lösungen, die in ihrem Inhalt eine Abbildung des Aufgabensystems beim Anwender darstellen. Sie setzt in der Regel auf der Systemsoftware der verwendeten Hardware auf bzw. benutzt sie zur Erfüllung der eigenen Aufgaben.

Anwendungsfall

Anwendungsfall (Use-Case)

Eine abgeschlossene, ununterbrochene Abfolge von Interaktion zwischen einem oder mehreren Akteuren und dem System, die zu einem identifizierbaren, fachlichen Zielergebnis führen

Aspektorientierte Programmierung

Aspektorientierte Programmierung

Die grundlegende Idee der Aspektorientierten Programmierung (AOP) ist, sogenannte Querschnittsaspekte getrennt zu entwickeln und erst später und transparent in die Anwendung zu integrieren.

Querschnittsaspekte sind Aufgaben wie Security, Logging, Transaktionalität oder Ähnliches. Dies sind Aufgaben, die unter Umständen in fast allen Komponenten verwendet werden müssen. Um nun nicht den Code "zu verschmutzen" sollen diese möglichst transparent, also unsichtbar, in den Code eingewoben (weaving) werden.

Commit

Commit

Ein commit ähnelt dem einer Datenbanktransaktion und bezeichnet einen check-in. In diesem Fall werden die Dateien des lokalen Sandboxverzeichnisses auf den Server übertragen und die Änderungen und Anmerkungen verwaltet.

Compiler

Compiler

Der Compiler ist ein Programm, das Quellcode komplett in Objektcode bzw. in ein ausführbares Programm übersetzt. Erst das vollständig übersetzte Programm wird ausgeführt.

Constraint

Zusicherung (constraint)

Mit einer Zusicherung können Regeln bestimmt werden, die als Vereinbarungen mit dem Programmierer fungieren.

Container

Container (Behälter)

Im softwaretechnischen Alltag wird der Begriff Behälter metaphorisch verwendet. Die charakteristische Eigenschaft eines Behälterobjekts, andere Objekte enthalten zu können, wird in die Welt der Software-Konstruktionen übertragen.

Continous Integration

Continous Integration (CI)

Continuous Integration ist eine Vorgehensweise, Software zu entwickeln. Dabei steuert jeder Entwickler häufig - wenn möglich täglich - seine Arbeit bei, was dazu führt, das mehrmals täglich integriert wird. Diese Integrationsbuilds werden von dem automatisierten Build überprüft (z. B. durch Tests), um Fehler so früh wie möglich festzustellen. Viele Teams haben berichtet, das dieses Vorgehen die Integrationsprobleme bedeutend reduziert.

Contract-Programming

Contract-Programming

Contract-Programming (Synonyme: Design-by contract oder auch Programming by Contract) ist ein Konzept der Softwareentwicklung. Darunter versteht man, dass Programmmodule so präzise definiert werden, ihre Ein- und Ausgabeparameter geprüft werden.

Daten

Daten

Unter Daten (Singular: Datum) versteht man Informationen, die nach einer eindeutigen Vorschrift in ein computergerechtes Format überführt wurden. Daten bestehen aus Zeichenfolgen.

Datenbank

Datenbank

In einer Datenbank werden große Datenbestände dauerhaft und effizient gespeichert. Neben den Nutzerdaten enthält eine Datenbank auch Beschreibungsdaten für die interne Datenorganisation. Zum Entwerfen, Aufbauen und Nutzen einer Datenbank dient ein Datenbankmanagementsystem. Datenbank und Datenbankmanagementsystem fasst man zum Datenbanksystem zusammen.

Datenstruktur

Datenstruktur

Neben den einfachen Datentypen gibt es die Möglichkeit mehrere Daten unter einem gemeinsamen Namen zusammenzufassen. Diese komplexen Einheiten bezeichnet man als Datenstrukturen

Dependency Injection

Dependency Injection (DI)

Dependency Injection ist ein generelles Entwurfsmuster, welches der Entkopplung von Abhängigkeiten zwischen Systemkomponenten dient. Verantwortung für das Erzeugen und die Verknüpfung von Objekten wird an ein extern konfigurierbares Programmiergerüst übertragen. Dependency Injection dient dazu Softwaretests zu vereinfachen, Abhängigkeiten zu reduzieren und Implementationen leichter austauschbar zu machen.

Designphase

Designphase (Entwurfsphase)

In der Designphase, welche der Analysephase folgt, geht es um die Erstellung einer Software-Architektur und deren Spezifikation von Systemkomponenten. Das Ergebnis der Designphase ist ein abstraktes, aber implementierbares Modell im Lösungsraum. (Siehe auch Elaboration Phase).

Diagramm

Diagramm

Ein Diagramm ist eine grafische Darstellung zur Veranschaulichung von Zahlen, (statistischen) Daten, Größen und logischen Größenbeziehungen.

Domain Specific Language

Domain Specific Language

Domain-Specific Language (DSL) (dt. domänenspezifische Sprache) ist eine Programmiersprache, die einen kleinen Problemraum abdeckt. Die Syntax und Semantik der DSL entsprechen möglichst der gleichen Abstraktionsebene, wie das Problemfeld selbst. Kennzeichnend für diese Sprache ist, dass sie exakt und optimal auf das Problem angepasst ist und auch leicht von Menschen zu erstellen oder zu erlernen sein soll.

Elaboration Phase

Elaboration Phase (Ausarbeitungsphase)

Die Elaboration Phase ist der zweite Teilprozess eines Software-Entwicklungszyklus, ausgerichtet auf Gestaltung und Architekturentwurf. (Siehe auch Entwurfsphase).

Entwurfsmuster

Entwurfmuster (design pattern)

Entwurfsmuster sind ein Konzept aus der Software-Architektur, in der immer wieder gleichartige Aufgabenstellungen (wenn auch oft mit unterschiedlichen Programmiersprachen) gelöst werden müssen. Entwurfsmuster beschreiben auf hoher Abstraktionsebene jeweils einen Ansatz für eine solche Lösung.

Ereignis

Ereignis

Ein Ereignis ist das Eintreten eines betriebswirtschaftlich relevanten Zustandes eines Informationsobjektes (Datenobjektes), der den weiteren Ablauf des Geschäftsprozesses steuert oder beeinflusst. Ereignisse können Auslöser oder Ergebnisse von Funktionen sein.

Expressiveness

Expressiveness

Unter Expressiveness versteht man die Ausdrucksstärke einer Sprache. Mit 1 wird oft die Sprache C bezeichnet, moderne dynamische Sprachen haben oft eine Expressiveness von 7 und mehr. Gemeint ist damit meistens die Anzahl der Codezeilen im Vergleich zu C gemeint.

Hier eine Gegenüberstellung verschiedener Sprachen:

Language	Statements ratio	Lines ratio
C	1	1
C++	2.5	1
Fortran	2	0.8
Java	2.5	1.5
Perl	6	6
Smalltalk	6	6.25
Python	6	6.5

Expressiveness von Sprachen

Quelle: http://en.wikipedia.org/wiki/Comparison_of_programming_languages

Moderne Sprachen wie Scala oder Clojure haben manchmal oft sogar bis zu 10 oder 20!

Framework

Framework

Ein Framework (deutsch: Rahmenstruktur, Fachwerk) ist ein Programmiergerüst, welches in der Softwaretechnik insbesondere im Rahmen der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen verwendet wird. Ein Framework ist selbst noch kein fertiges Programm, sondern stellt den Rahmen, innerhalb dessen der Programmierer eine Anwendung erstellt, zur Verfügung, wobei u. a. durch die in dem Framework verwendeten Entwurfsmuster auch die Struktur der individuellen Anwendung beeinflusst wird.

Funktionen

Funktionen

Betrachtet man eine Funktion im Zusammenhang mit einem Anwendungssystem, dann bewirkt sie die Transformation der Eingabe- in Ausgabedaten bzw. sie bewirkt eine Veränderung des Inhalts oder der Struktur von Informationen. (Siehe auch Methode).

Geschäftsprozess

Geschäftsprozess

Ein Geschäftsprozess ist eine zusammenhängende Abfolge von Unternehmensverrichtungen zum Zweck einer Leistungserstellung. Ausgang und Ergebnis des Geschäftsprozesses ist eine Leistung, die von einem internen oder externen Kunden angefordert und abgenommen wird.

Hardware

Hardware

Als Hardware werden alle physikalischen Teile eines Computers bezeichnet.

Higher Order Functions

Higher Order Functions

(dt. Funktionen höherer Ordnung) sind im Kontext der Informatik Funktionen, die selbst Funktionen als Argument haben oder Funktionen als Rückgabewert übergeben.

Homoiconic

Homoiconic

Als Homoiconic werden Sprache bezeichnet bei denen Daten als Code verstanden werden können und Code die Form von Daten hat. So ist z.B. bei Lisp (LIST Processing) der Code ebenfalls eine Liste und kann bequem verarbeitet und transformiert werden.

(C) http://en.wikipedia.org/wiki/Homoiconicity

Immutable

Immutable

Im Kontext der Programmierung versteht man unter immutable (dt. nicht veränderbar), dass die Werte der Datenstrukturen nach der Erstellung nicht veränderbar sind. Beispiel: 50% der Datenstrukturen von Scala und 100% von Clojure sind immutable (wie Vector, List, Map und HashMap).
Im Gegensatz dazu gibt mutable Datenstrukturen also veränderbare.

Implementierung

Implementierung (Implementieren)

Der entworfene Lösungsweg wird in eine Programmiersprache übertragen und codiert. Damit beginnt der eigentliche Editiervorgang, bei dem entsprechend der Programmiersprache die Anweisungen ausgearbeitet und niederschrieben werden.

Informationen

Informationen

Die zweckbestimmte Interpretation von Daten durch den Menschen. Daten bestehen zunächst nur aus Fakten und werden erst dann zu Informationen, wenn sie im Kontext betrachtet werden und eine Bedeutung für den Menschen übermitteln. Computer verarbeiten Daten, ohne deren Bedeutungsgehalt in einer beliebigen Form zu verstehen.

Intrinsisch

Intrinsisch

Synonym für innewohnend, immanent. Eine Eigenschaft, die ein Gegenstand, Lebewesen oder System von sich aus aufweist.

Klasse

Klasse

Eine Klasse ist eine Definition der Attribute, Operationen und der Semantik für eine Menge von Objekten. Alle Objekte einer Klasse entsprechen einer Definition.

Komponente

Komponente

Unter einer Komponente versteht man eine Einheit, die eine definierte Schnittstelle besitzt. Sie hängt nur von sich selbst ab und kann daher beliebig mehrfach eingesetzt und erweitert werden

Kopplung

Kopplung

Die Kopplung schaut sich die Benutzt-Relationen an, also den Grad der Abhängigkeit zwischen Komponenten. Dies sind meistens Assoziationen aber auch Datenabhängigkeiten. Grundsätzlich ist eine schwache Kopplung erstrebenswert.

Mealy-Automat

Mealy-Automat

Zustandsautomat, bei dem die Ausgaben bzw. Aktionen an den Zustandsübergang gebunden sind.

Merge

Merge

Ein Merge bringt verschiedene Software-Versionen zusammen. Beispielsweise den "Branch" (Zweig) 1.3.12 mit der Version 1.4.2

Methode

Methode (Programmierung)

Eine Methode ist im objektorientierten Datenmodell eine Prozedur, die in einem Objekt vereinbart ist und anderen Objekten zur Verfügung gestellt wird, damit diese die Dienste des Objekts in Anspruch nehmen können. Die meiste Kommunikation zwischen Objekten findet über Methoden statt. (Siehe auch Funktionen).

Model Driven Architecture

Model Driven Architectur

Model Driven Architecture (deutsch: Modellgetriebene Architektur) bezeichnet einen modellgetriebenen Softwareentwicklungsansatz, der auf einer klaren Trennung von Funktionalität und Technik beruht.

Modell

Modell

Unter einem Modell versteht man in der Softwaretechnik die Abbildung eines Systems, wobei nur die in Hinblick auf den Modellzweck als wichtig beurteilten Systemkomponenten und Beziehungen zwischen den Komponenten abgebildet werden. Jeder Modellbildung liegt ein Abstraktionsvorgang zugrunde.

Moore-Automat

Moore-Automat

Zustandsautomat, bei dem die Ausgaben bzw. Aktionen an den Zustand gebunden sind.

Namensraum

Namensraum (Programmierung)

Über einen Namensraum (engl.: Namespace) sollen Namen von Modellelementen (Komponenten) eindeutig unterscheidbar sein, womit eine eindeutige Zuordnung bzw. Identifizierung zwischen Namen und Modellelement möglich sind. Ein Namensraum enthält eine Gruppe von Modellementen, die zwei Bedingungen erfüllen müssen. Erstens müssen sie einen Namen haben. Zweitens muss jedem Element eine der Sichtbarkeiten public, private, protected oder package zugeordnet sein.'

Objekt

Objekt

In der objektorientierten Programmierung versteht man unter einem Objekt eine Instanz oder ein Exemplar einer Klasse. Ein Objekt enthält Daten und Methoden (nach dem Muster der Klasse) sowie eine eigene Identität.

Objektorientierung

Objektorientierung (OO)

Der Objekt-orientierte Zugang zur Softwareentwicklung stellt die Kapselung von Daten und Funktionen zu Software-Objekte in den Mittelpunkt der Modellierung. Elementare Einheit der OO-Software-Modellierung ist die Klasse. Jede Klasse K spezifiziert die Attribute (Daten) und Funktionen (Methoden/Operationen) und damit die Struktur von Objekten. Die Klasse K legt somit den Typ von K-Objekten fest, die zur Laufzeit erzeugbar sein sollen. Die zur Laufzeit erzeugbare K-Objekte sind strukturidentische Kapselungen von Attribut-Werten (Objektzustand zu einem Zeitpunkt x) und aufrufbare Funktionen auf diesen.

OMG

Object Management Group (OMG)

1989 gegründeter, internationaler Verband, der offene Standards für objektorientierte Anwendungen fördert. OMG war auch an der Definition von OMA ("Object Management Architecture", deutsch: Objektverwaltungs-Architektur) beteiligt, einem Standardobjektmodell für verteilte Umgebungen.

Operator

Operator

Aus der Mathematik bekanntes Operationssymbol, wie z. B. "+", für die Addition. In Programmiersprachen gibt es verschiedene Arten von Operatoren, wie Operatoren für logische Vergleiche oder für die Verkettung von Zeichenketten. In Java gibt es Operatoren, die durch ein Symbol oder zwei Symbole (z. B. "++") dargestellt werden.

Package

Package

Ein Package, oder auch Paket, fasst Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene auszudrücken. Pakete können im Paketdiagramm dargestellt werden.

Programm

Programm

Eine nach den Regeln einer Programmiersprache zusammengestellte Folge von Anweisungen, die zur Erledigung einer oder mehrerer Aufgaben dient.

Programmiersprache

Programmiersprache

Programmiersprache (engl.: programming language) ist eine formale Sprache zur Formulierung von Computerprogrammen. Programmiersprachen sind die Kommunikationsschnittstelle zwischen Mensch und Computer. Neben der Maschinensprache gibt es höhere Programmiersprachen, die verschiedene Programmierkonzepte unterstützen.

Projekt

Projekt

Ein Projekt ist ein komplexes Vorhaben, das unter Berücksichtigung aller Umstände (wie z.B. Zeit, Kosten und Ressourcen) ein klar definiertes Ziel erreichen soll.

Quellcode

Quellcode (Quelltext, Programmcode)

Ein Quellcode (engl: source code) ist ein in einer höheren Programmiersprache geschriebenes und von Menschen lesbares Programm, das von einem Compiler in ein Zielprogramm übersetzt bzw. von einem Interpreter ausgeführt werden soll.

Repository

Repository

Das Repository ist der Ort, an dem die Dateien gespeichert werden (vergleichbar einem Archiv oder einer Sammelstelle). Dies ist in der Regel ein Server.

Ressourcen

Ressourcen

Unter Ressourcen (Einsatzmittel) versteht man die Gesamtheit an Personal und Sachmittel, die zur Durchführung von Handlungen, Vorgängen oder Projekten benötigt werden.

Reverse Engineering

Reverse Engineering

Eine Vorgehensweise, mit der versucht wird, ein altes Softwaresystem anhand des laufenden Systems und der vorhandenen Unterlagen zu verstehen. Fehlende Informationen, Dokumente und Modelle werden durch manuelle und/oder automatische Analysen abgeleitet. Das Altsystem wird in seiner Funktionalität und Semantik nicht verändert.

Schnittstelle

Schnittstelle (Interface)

Eine Schnittstelle ist ein definierter Übergang oder eine Verbindungsstelle zwischen zwei unterschiedlich arbeitenden Teilprogrammen oder -systemen, über die der Austausch von Daten, Signalen u. a. erfolgen kann.

Software

Software

Software umfasst Programme, zugehörige Daten und Dokumentationen, die in ihrem Zusammenwirken gestatten, mit Hilfe eines Computers Prozesse zu unterstützen bzw. Aufgaben zu lösen.

Software-Architektur

Software-Architektur

Unter Softwarearchitektur versteht man die Beschreibung der grundlegenden Komponenten eines komplexen Softwaresystems und deren Interaktion.

Softwareprodukt

Software-Produkt

Bei der Verwendung des Begriffs Softwareprodukt betrachtet man Software von außen, aus der Benutzer- oder Auftraggebersicht, während ein Softwaresystem die innere Sichtweise, die des Softwareentwicklers darstellt.

Softwaresystem

Software-System

Ein Softwaresystem ist ein System, dessen Systemkomponenten und Systemelemente aus Software bestehen. Ein System ist aus Teilen (Systemkomponenten oder Subsystemen) zusammengesetzt, die untereinander in verschiedenen Beziehungen stehen können. Systemteile, die nicht weiter zerlegbar sind oder zerlegt werden sollen, werden als Systemelemente bezeichnet.

Spezifikation

Spezifikation

Eine Spezifikation definiert die Merkmale und Funktionsweise eines Produkts, eines Systems oder einer Dienstleistung.

Standard

Standard

Eine gesetzlich gültige, technische Richtlinie (Norm), die von einer staatlichen oder einer nicht gewerblichen Organisation befürwortet wurde, um in einem bestimmten Bereich der Entwicklung von Hardware oder Software eine Einheitlichkeit zu erzielen. Der Standard ist das Ergebnis einer formalen Prozedur, basierend auf Spezifikationen, die von einer kooperativen Gruppe oder einem Komitee nach einer ausführlichen Untersuchung der vorliegenden Verfahren und Ansätze sowie technologischen Tendenzen und Entwicklungen erstellt werden.

Strukturdiagramm

Strukturdiagramm (auch Statisches Diagramm)

Ein Strukturdiagramm ist ein grafisches Darstellungsmittel zur Beschreibung von Funktions- bzw. Systemstrukturen mit allen darin vorhandenen Elementen, sowie deren Beziehungen und Abhängigkeiten zueinander. Strukturdiagramme in der UML sind: Klassendiagramm, Komponentendiagramm, Kompositionsstrukturdiagramm, Objektdiagramm, Paketdiagramm und Verteilungsdiagramm.

Subklassen

Subklasse (Unterklasse)

Subklassen sind Spezialisierungen von übergeordneten Superklassen, von denen sie Methoden und Attribute erben.

Superklassen

Superklasse (auch Basis- oder Oberklasse)

Superklasse ist ein Begriff aus der objektorientierten Programmierung und bezeichnet eine Klasse, die den von ihnen abgeleiteten Klassen (Subklassen) ihre Methoden und Attribute vererben.

Swimlanes

Swimlanes

Durch Swimlanes kann man die Darstellungsfläche eines Aktivitätsdiagramms in mehrere senkrechte Streifen unterteilen. Sie grenzen Verantwortungsbereiche ab. Eine Swimlane kann einem Akteur, einer Klasse oder einem Package zugeordnet werden.

System

System

Jede Sammlung von Einzelelementen, die zur Ausführung einer Aufgabe zusammenarbeiten. So stellt ein Computer ein Hardware-System dar, das aus einem Mikroprozessor, den zugehörigen Chips und Schaltkreisen sowie einem Eingabegerät, einem Ausgabegerät und beliebigen Peripheriegeräten besteht. Ein Betriebssystem besteht aus einem grundlegenden Satz von Programmen zur Verwaltung der Hardware und der Datendateien. Ein Datenbank-Managementsystem besteht aus Komponenten zur Verarbeitung bestimmter Arten von Informationen.

Systemsoftware

Systemsoftware

Unter Systemsoftware (auch Systemprogramme) versteht man die Software, die die unmittelbare Schnittstelle zur Hardware, also dem Computersystem, darstellt. Hierzu zählen das Betriebssystem sowie Compiler und Interpreter.

Technical Debt

Technical Debt

Frei übersetzt bedeutet Technical Debt, dass man eine Schuld eingeht, weil auf der technischen Seite etwas verbessert werden kann. Diese Schuld kann verschiedene Ursachen haben. Beispielsweise: schlechte Architektur / Design, viele Codemetrikverletzungen, zu hohe Komplexität, zu große Kopplung, Zyklen.

Top-Down-Vorgehen

Top-Down-Vorgehen

Ein Entwurfskonzept für Programme, bei dem man zunächst die Funktionalität eines Programms (als eine Folge von Tasks) auf der höchsten Ebene festlegt und dann schrittweise jeden Task in Funktionsblöcke auf der jeweils darunterliegenden Ebene aufteilt

Transition

Transition (Zustandsübergang)

Eine Transition verbindet einen Ausgangs- und einen Folgezustand. Sie kann nicht unterbrochen werden und wird stets durch ein Ereignis ausgelöst. Ausgangs- und Folgezustand können identisch sein.

UML

Unified Modeling Language (UML)

UML ist eine Notation zur grafischen Darstellung objektorientierter Systeme.
Zur grafischen Darstellung gehören Klassen-, Objekt-, Kollaborations- und Sequenzdiagramme.

Variadic

Variadic

Unter dem Begriff variadic (dt. variadisch) versteht man im Kontext der Programmierung Operatoren, die beliebig viele Argumente aufnehmen können, d.h. bei der Vereinbarung steht die Anzahl der Argrumente noch nicht fest, es können daher beliebig viele Argumente (varargs) angegeben werden.

Vererbung

Vererbung

Unter Vererbung versteht man das Ableiten von Unterklassen (Subklassen) durch übergeordnete Oberklassen (Superklassen). Vererbt werden Datenstruktur (Attribute) und Verhalten (Methoden) der vererbenden Klasse.

Vorgehensmodell

Vorgehensmodell

Unter einem Vorgehensmodell versteht man in der Softwareentwicklung einen Plan, der den Entwicklungsprozess in unterteilte Phasen organisiert. Das dient dazu das Erstellen und Warten von komplexer Software übersichtlicher und geordneter zu gestalten. Vorgehensmodelle spiegeln das theoretische, konzeptionelle und methodische Verständnis von den Prozessen und Produkten der Software-Entwicklung wider.

Wartung

Wartung

Nachdem ein Softwareprodukt zur Anwendung freigegeben wurde, beginnt die Wartung und Pflege, d. h. es gilt alle nach Inbetriebnahme auftretenden Fehler zu beseitigen und das Anwendungssystem an veränderte Bedingungen anzupassen und es bei Vorliegen neuer Anforderungen weiterzuentwickeln.

White-Box-Tests

White-Box-Tests

Ein White-Box-Test hat die Kenntnis über den internen Ablauf der zu testenden Komponente. Üblicherweise wird dieser Test von der gleichen Person geschrieben. Ein bekanntes Beispiel ist der Test auf eine bestimmte Exception. Hier wird also ein Test geschrieben, ob die Komponente beispielsweise bei Übergabe eines Null-oder besonderen Wertes auch die richtige Exception liefert.

Zustandsautomat

Zustandsautomat

Der Zustandsautomat besteht aus einer endlichen Anzahl von internen Zuständen. Zwischen den Zuständen gibt es Übergänge, die in Abhängigkeit von Eingaben oder Ereignissen durchgeführt werden. Eine Ausgabe oder Aktion kann beim Zustandsübergang erfolgen (Mealy-Automat) oder in einem Zustand (Moore-Automat).