

The slide features a blue header bar with the text 'SWT1', 'Dr. Jörg Mielebacher', and '1'. The main content area is white and contains the title 'IMPLEMENTIERUNGSPHASE' in a large, bold, black font. Below the title is a horizontal line, and underneath that line is the text 'Dr. Jörg Mielebacher, mail@mielebacher.de'.

Die folgenden Folien beschäftigen sich mit der Implementierungsphase. Hierzu gehören Implementierungsprinzipien und Werkzeuge dieser Phase. Zu jeder Folie sind Notizenseiten erfasst.

Verbesserungsvorschläge und Fehlerhinweise können Sie gerne an die Adresse mail@mielebacher.de senden.

Rechtliche Hinweise: Die Rechte an geschützten Marken liegen bei den jeweiligen Markeninhabern. Alle Rechte an diesen Folien, Notizen und sonstigen Materialien liegen bei ihrem Autor, Jörg Mielebacher. Jede Form der teilweisen oder vollständigen Weitergabe, Speicherung auf Servern oder Nutzung in Lehrveranstaltungen, die nicht von dem Autor selbst durchgeführt werden, erfordert seine schriftliche Zustimmung. Eine schriftliche Zustimmung ist darüber hinaus für jede kommerzielle Nutzung erforderlich. Für inhaltliche Fehler kann keine Haftung übernommen werden.

SWT1	Dr. Jörg Mielebacher	2
------	----------------------	---

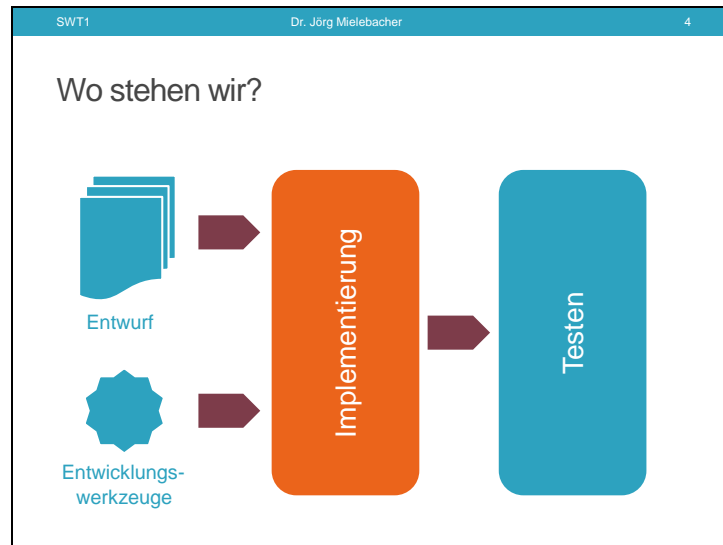
Eckpunkte der letzten Veranstaltung

- Entwurfsprinzipien sollen den Entwurf auf mehreren Ebenen verbessern.
- Strukturierung, Modularisierung, Abstraktion und Geheimnisprinzip sind wesentlich für wartbare Software.
- Wiederverwendung beschleunigt und verbessert die Entwicklung, verursacht aber zusätzlichen Aufwand.
- Das Gesetz von Demeter und die SOLID-Prinzipien sind wichtige Handreichungen für den guten objektorientierten Entwurf.
- Standardisierung ist wichtig, um Fehler zu vermeiden, effizienter zu arbeiten.
- Entwurfsmuster sind bewährte Lösungsansätze für wiederkehrende Probleme der Softwareentwicklung.

SWT1	Dr. Jörg Mielebacher	3
------	----------------------	---

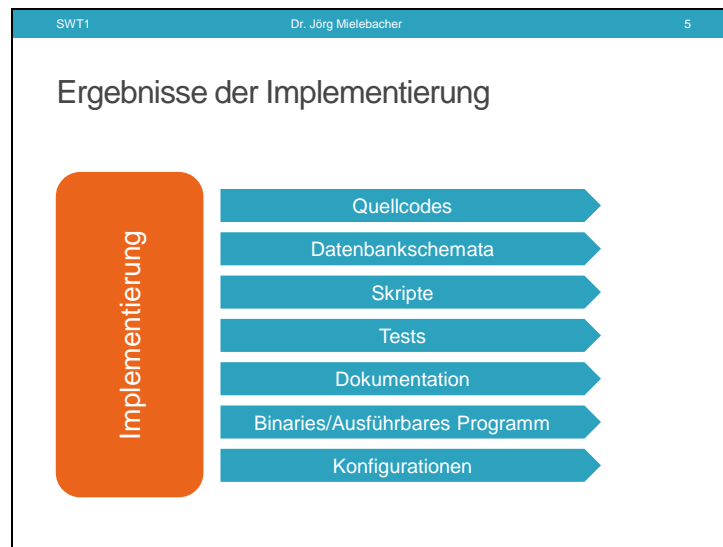
Literaturempfehlungen

- Balzert: Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb
 - Standardwerk, teilweise aber sehr akademisch
- Sommerville: Software Engineering
 - Standardwerk, akademisch, aber z.B. gute FAQ-Kapitel
- Boswell/Foucher: The Art of Readable Code
 - Viele praktische Tipps, sehr empfehlenswert
- Java Coding Standards
 - <http://www.oracle.com/technetwork/java/codeconv-138413.html>
- Git Documentation
 - <http://git-scm.com/documentation>
- Javadoc Documentation
 - <http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html>



Die zurückliegenden Veranstaltungen beschäftigten sich mit der Analyse von Anforderungen und anschließend mit dem Entwerfen der Software. Quellcode war bis zu diesem Punkt nur von untergeordneter Bedeutung. Dies ändert sich mit der Phase der Implementierung.

Nach erfolgter Implementierung folgt üblicherweise die Phase des Testens.



Die Implementierungsphase liefert als Ergebnis vor allem den Quellcode der Software, dazu aber auch die notwendigen Datenbankschemata, Skripte (z.B. Buildskripte, Installationsskripte usw.), Testroutinen sowie die Implementierungsdokumentation. Ergebnisse der Implementierung sind aber auch die sog. Binaries, also das ausführbare Programm sowie die Anfangskonfiguration.

Das Ausführbare Programm liegt am Ende der Implementierung noch nicht in der sog. Release-Version vor, da vor dem Release zuerst noch die vorgesehenen Tests durchgeführt werden müssen.



Für eine erfolgreiche Implementierung sind eine Reihe von Anforderungen wesentlich:

Nachvollziehbarkeit: Die Nachvollziehbarkeit des Quellcodes umfasst ein zweckmäßiges und übersichtliches Codelayout, selbsterklärenden Code, sprechende Bezeichner, klar dokumentierte Verfeinerungen, hilfreiche Kommentare usw. Ein intuitiver Lösungsansatz ist Voraussetzung hierfür, dieser wird aber üblicherweise bereits im Entwurf festgelegt.

Integrierte Dokumentation: Alle notwendigen Informationen sollen möglichst im Quellcode zu finden sein. Mehr dazu im Kontext Dokumentation.

Defensives Programmieren: Defensives Programmieren zielt auf robuste Software ab, d.h. man erwartet Fehler und Falscheingaben und berücksichtigt dies bei der Implementierung. Hierzu gehört vor allem, dass vor dem eigentlichen Verarbeitungsschritt alle notwendigen Voraussetzungen überprüft werden (z.B. Wertebereiche, Formate, Plausibilität usw.). Wichtige Hilfsmittel sind dazu Exceptions, try-catch-Blöcke sowie Assertions. Letztere lassen sich mit dem Schlüsselwort `assert` in den Quellcode integrieren; erst wenn bei der Ausführung des Programms die Option `-ea` gesetzt ist, werden diese Assertions tatsächlich ausgewertet.

Adäquate Werkzeuge: Die Auswahl der Werkzeuge und Programmierkonstrukte ist wichtig, da hierdurch eine effizientere Entwicklung möglich wird, die verhindert, dass Mängel in den Werkzeugen kompensiert werden müssen.

Automatisierung: Zur Fehlervermeidung sollten möglichst viele Schritte der Implementierung automatisiert ablaufen, z.B. Prüfen und Vereinheitlichen des Codeformats, Ausführen von Tests, Versionierung, Dokumentation usw.

Adäquate Kompetenz: Zahlreiche Studien haben gezeigt, dass Erfolg und Produktivität der Softwareentwicklung stark von der Kompetenz der Entwickler abhängen. Bei der Implementierung betrifft dies die Kenntnis und den routinierten Einsatz der Programmierkonstrukte, der verwendeten Klassenbibliotheken sowie den Umgang mit wahrscheinlichen Fehlern und Risiken (z.B. Anwendungssicherheit).



Für die Implementierung stehen zahlreiche und in ihren Eigenschaften mitunter sehr verschiedene Programmiersprachen zur Verfügung. Im Rahmen des Entwurfs wird für das Projekt festgelegt, welche Sprache für welchen Zweck verwendet wird. Beispielsweise könnte die Anwendung selbst in Java geschrieben werden, die Datenbankabfragen in SQL und die Installationsskripte als Bash-Skript.

Programmiersprachen lassen sich entlang unterschiedlicher Dimensionen unterscheiden. So unterscheidet man etwa Hochsprachen von Maschinensprachen; ein in Maschinensprache geschriebenes Programm lässt sich direkt vom Prozessor ausführen, während in Hochsprachen geschriebener Quellcode zuerst in Maschinensprache übersetzt werden muss. Der Hochsprachen-Code ist jedoch verständlicher und enthält abstrakte Konstrukte wie Bedingungen und Schleifen. Zu den Hochsprachen gehören u.a. C, C++, Java, PHP usw. Programmierung direkt in Maschinensprache ist sehr unüblich, in eng umrissenen Bereichen (hardwarenahe Optimierungen, Embedded Systems usw.) setzt man Assemblersprachen ein, die der Maschinensprache ähneln, deren Binärcodes aber durch mnemonische Symbole ersetzen (z.B. MOV).

Daneben gibt es die Unterscheidung in imperative und deklarative Programmiersprachen. Imperative Sprachen beschreiben die einzelnen Verarbeitungsschritte (das „Wie“), während deklarative Sprachen Ergebnisse beschreiben (das „Was“) und den Weg zu diesen Ergebnissen offen lassen. Imperative Sprachen sind wiederum C, C++, Java usw., während zu den deklarativen Sprachen bekannte Vertreter wie SQL zählen, aber auch eher weniger bekannte wie Haskell, Prolog usw.

Unterscheiden lassen sich außerdem prozedurale (Basic, Pascal usw.) und objektorientierte Sprachen (C++, Java usw.), wobei es auch sog. Hybridsprachen gibt – beispielsweise ist C++ keine rein objektorientierte Sprache, sondern erlaubt auch das prozedurale Programmieren, das keine Klassen und dergleichen kennt.

Eine wichtige Unterscheidung betrifft die Übersetzung des Quellcodes: Hier unterscheidet man Sprachen, bei denen der Quellcode zunächst vollständig kompiliert werden muss (C, C++ usw.) sowie interpretierte Sprachen (PHP, Basic usw.), bei denen der Quellcode erst zur Laufzeit schrittweise in Maschinencode übersetzt wird.

Die Frage nach der geeigneten Programmiersprache muss zahlreiche Aspekte berücksichtigen, z.B. die einschlägige Kompetenz der Entwickler, die Unterstützung der Projektanforderungen, die Verfügbarkeit geeigneter Werkzeuge usw.

SWT1Dr. Jörg Mielebacher8

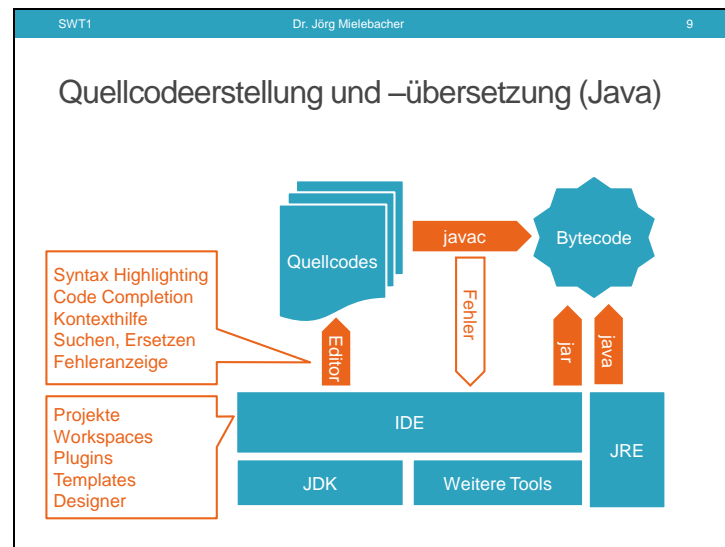
Styleguides / Coding Standards

- Quellcodeformatierung
 - Einrückungen
 - Zeilenlänge
 - Klammerung
- Dateien
 - Zeichencodierung
 - Namen und Ablage
- Namenskonventionen
- Dokumentation
 - Sprache
 - Aufbau von Kommentaren
- Vorlagen
- u.v.m.

Das Schreiben des Quellcodes wird typischerweise durch Styleguides/Coding Standards festgelegt. Diese legen u.a. folgende Aspekte fest:

- Quellcodeformatierung (Art und Größe von Einrückungen, Zeilenlänge, Art der Klammersetzung usw.)
- Dateinamen und Zeichencodierung (z.B. Windows, Unix usw.)
- Namenskonventionen (Benennung von Variablen, Methoden usw.)
- Dokumentationssprache sowie Art der Kommentierung sowie ggf. erlaubte Tags für Dokumentationsgeneratoren
- Vorlagen für Kommentare (z.B. Methodenkomentar), Methoden, Kontrollstrukturen, Dateihheader usw.

Styleguides definieren darüber hinaus oftmals allgemeine Vorgaben für grafische Oberflächen, Schnittstellen und dergleichen.



Für die Quellcodeerstellung nutzt man üblicherweise eine Integrierte Entwicklungsumgebung (IDE). Diese besteht aus einem Editor und aus der Anbindung der notwendigen Werkzeuge. Hierzu zählt vor allem die Integration der Übersetzungswerkzeuge (Compiler, Linker usw.); das bedeutet, dass mit Starten des sog. Build-Prozesses die notwendigen Werkzeuge für die Übersetzung des Quellcodes der Reihe nach aufgerufen werden. Ausgaben dieser Werkzeuge werden wiederum von der IDE angezeigt und ausgewertet (z.B. Fehlermeldungen des Compilers).

Die Editor-Komponente einer IDE ist besonders mächtig. Typische Funktionen dieses Editors sind:

- 1.) Syntax Highlighting (das farbliche Hervorheben von Sprachelementen)
- 2.) Code Completion (Vervollständigung von Eingaben nach den ersten Zeichen)
- 3.) Kontexthilfe (Hilfe zu gerade ausgewählten Sprachelementen)
- 4.) Suchen und Ersetzen in einzelnen oder mehreren Dateien, mit regulären Ausdrücken usw.
- 5.) Hervorheben von Fehlern, z.B. übliche Syntaxfehler (z.B. Klammer vergessen, unbekannte Variablen usw.), vor allem aber die vom Compiler gefundenen Fehler

Weitere Funktionen der IDE sind:

- 1.) Organisation des Quellcodes in Projekten (z.B. projektspezifische Einstellungen, Darstellung als Dateibaum usw.)
- 2.) Organisation von sog. Workspaces (Zusammenstellungen von Projekten und Einstellungen)
- 3.) Unterstützung von Plugins zur Erweiterung des Funktionsumfangs
- 4.) Verwalten von Templates für Codeteile oder Projekte
- 5.) Designer für Oberflächen und dergleichen

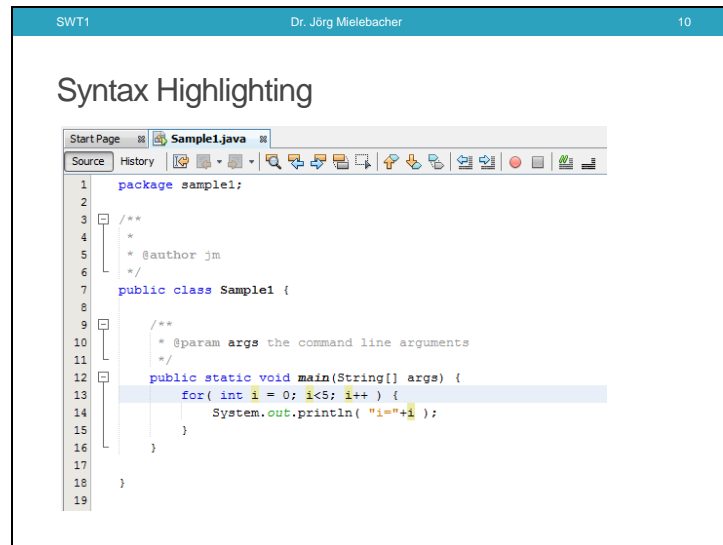
Bekannte Java-IDEs sind Eclipse und NetBeans, wobei beide auch für andere Programmiersprachen genutzt werden können. Unter den kommerziellen Entwicklungsumgebungen ist vor allem das Visual Studio von Microsoft verbreitet.

Für die Entwicklung von Java-Programmen ist eine Sammlung von Werkzeugen wichtig: das Java Development Kit (JDK). Es enthält unter anderem den Compiler `javac`, den Dokumentationsgenerator `javadoc` und den Archivierer `jar`. `javac` übersetzt den Quellcode (Dateiendung `.java`) in sog. Bytecode (Endung `.class`), der wiederum mit `jar` zur größeren Einheiten – sog. Jar-Archiven – zusammengefügt werden kann. `Javadoc` erzeugt aus den Quellcodes Dokumentationsdateien, dazu später mehr. Stellt der Compiler Fehler im Quellcode fest, werden diese nach Ende der Übersetzung an die IDE gemeldet; das Übersetzungsergebnis lässt sich in diesem Fall noch nicht ausführen. Stattdessen muss man zunächst die gefundenen Fehler beseitigen.

Der Java-Bytecode ist ein plattformunabhängiger Zwischencode, der auf der jeweiligen Zielplattform von der dort installierten Java Runtime Environment (JRE) in Maschinenanweisungen übersetzt und ausgeführt wird. Früher geschah dies durch Interpretieren, heute erfolgt dies aus Geschwindigkeitsgründen verstärkt durch Compiler.

Die IDE bindet typischerweise noch viele weitere Werkzeuge ein, z.B. für Fehlersuche, Versionsverwaltung, Build-Management und Tests. Dazu später mehr.

Ein typischer Entwicklerarbeitsplatz erfordert häufig weitere Software. Beispielsweise benötigt man für das Entwickeln einer Web-Anwendung geeignete Webserver bzw. Anwendungsserver. Auch zusätzliche Hardware kann notwendig sein, beispielsweise um Embedded Software zu erproben.

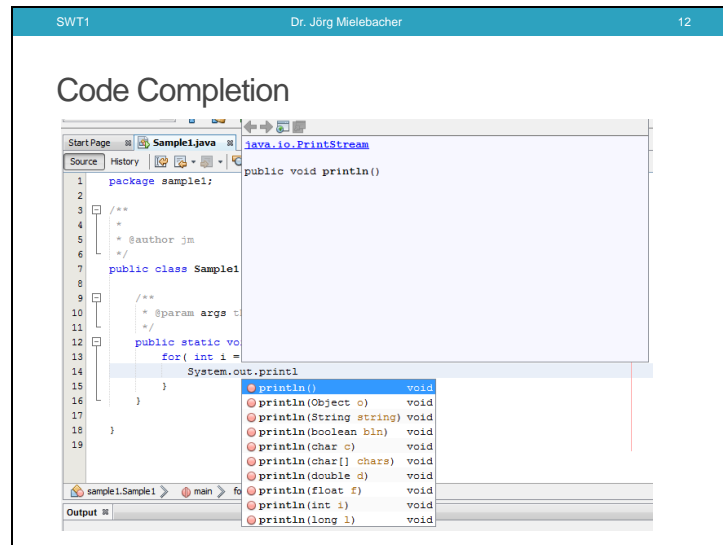


Syntax Highlighting bedeutet, dass die IDE bestimmte Teile des Quellcodes farblich hervorhebt. Hier z.B. die Schlüsselwörter (`package`, `public`, `class` usw.) blau und Kommentare grau. Außerdem werden z.B. zusammengehörige Klammerpaare hervorgehen, wenn der Cursor auf ihnen steht, oder – so wie hier – die Verwendungsorte einer Variablen.

Syntax Highlighting erhöht die Lesbarkeit des Codes, weil man Gleichartiges besser erkennen kann. Es dient aber auch der Fehlervermeidung, da man unmittelbar während der Eingabe eine Rückmeldung erhält, ob Begriffe richtig geschrieben wurden.

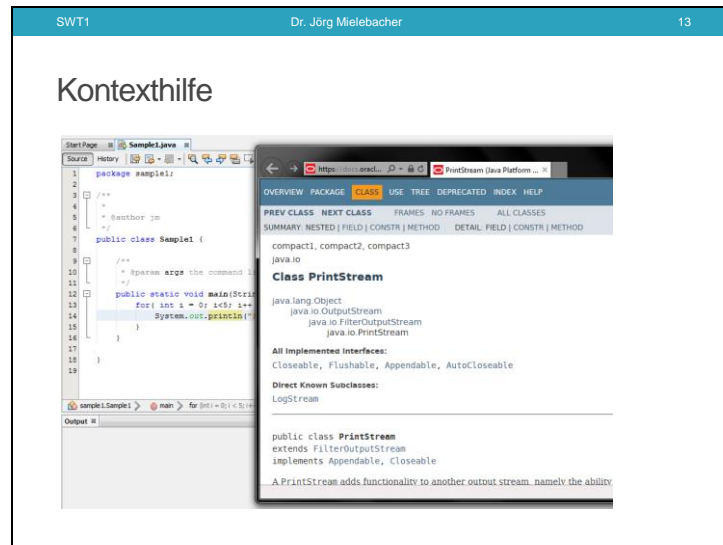


Ein einheitliches Quellcode-Layout macht den Quellcode übersichtlicher und dadurch verständlicher. Die Formatierung folgt dabei einigen grundlegenden Richtlinien, z.B. Einrückung, Formatierung von Klammern usw. Die IDE bietet hierfür i.d.R. eine automatische Quellcode-Formatierung; in Eclipse erreicht man dies mit Strg+Shift+F, in Netbeans mit Alt+Shift+F.



Code Completion bietet – ausgehend von den gerade eingegebenen Zeichen – diejenigen Attribute, Methoden usw. an, die der Benutzer möglicherweise verwenden möchte. Außerdem werden teilweise auch Hilfestellungen angezeigt (z.B. Parameterlisten).

Code Completion hilft, den Code schneller einzugeben, unterstützt dabei, den richtigen Bezeichner einzugeben und dient der korrekten Verwendung der Methoden usw. Die Code Completion ist also einerseits Effizienzwerkzeug, andererseits ein Beitrag zur Fehlervermeidung.



Die Kontexthilfe zeigt bei Bedarf den Javadoc-Eintrag zu einem Begriff an. In Eclipse erreicht man dies über Shift+F2, in Netbeans mit Alt+F1.

SWT1Dr. Jörg Mielebacher14

Lesbarkeit von Quellcode

Always code as if the guy
who ends up maintaining
your code will be a violent psychopath
who knows where you live.
Martin Golding

Eine kleine Motivation, sich mit dem Thema Lesbarkeit von Quellcode zu befassen...

SWT1
Dr. Jörg Mielebacher
15

Lesbarkeit von Quellcode

```

#include <stdlib.h>
#include <stdio.h>

int main(
    int a, int n, double d) char**a; {
    for(d=atoi(a[1])/10*80-
        atoi(a[2])/5-596;n="@NKA\
CLCCGZAAQBEEADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAABAFHJE\
TBdFLDAANEfDNBPPhdBcBBBEA_AL\
H E L L O,   W O R L D! "
        [1++-3];) for(;n-->64;)
        putchar(!d+++33^
                1&1);}

```



Das ist ein besonders nettes Beispiel für unlesbaren Code (Quelle: Isernhagen/Helmke, SW-Technik in C und C++, S.7). Das Programm erzeugt für die Parameter b2 50 50 die gezeigte Ausgabe. Verstehen Sie den Quellcode (Hinweis: Es ist C)?

SWT1	Dr. Jörg Mielebacher	16
------	----------------------	----

Zwischenfragen

- Warum ist das Attribut `gewicht` u.U. schlecht benannt?
- Warum ist der Parameter `grenzel` missverständlich?
- Was ist hier schlecht lesbar?

```
public int berechnung(int a,int b){  
    if(a>a-a)  
        return a;  
}
```

SWT1Dr. Jörg Mielebacher17

Lesbarer Code durch klares Erscheinungsbild

- Einrückungen
- Leerzeilen und Leerzeichen zur optischen Trennung
- Ausrichten von Ausdrücken

```
int average = 5;  
int max     = 100;
```

- Parameterwerte untereinander, ggf. benannt

```
find_level(    0, /* range_min */  
            100, /* range_max */  
            true, /* find_first */  
            false /* find_single */ );
```

Es gibt eine einfache Grundregel: Gleiches sollte gleich aussehen. Hierzu bedient man sich der üblichen Formatierungsregeln, z.B. Einrückungen oder die optische Trennung in sich abgeschlossener Quellcodeabschnitte. Sinnvoll ist aber auch die Nutzung eines Spaltenlayouts, z.B. bei mehreren Wertzuweisungen, bei Methodenparametern usw.

SWT1 Dr. Jörg Mielebacher 18

Lesbarer Code durch geeignete Bezeichner

- Auf allgemeine Bezeichner verzichten
x
berechnen()
- Wichtiges in den Namen einbetten
klartext_passwort
gewicht_kg
- Mehrdeutigkeiten auflösen
 - Bereichsgrenzen eindeutig formulieren → inklusiv oder exklusiv
 - Doppelte Negationen auflösen
!notloggedin → loggedin

Die Wahl geeigneter Bezeichner ist einer der besten Hebel für lesbaren Quellcode. Ziel ist es, möglichst viele Informationen in den Bezeichner zu packen, z.B. Einheiten (kg, ft, h etc.), Verarbeitungszustände (Klartext, verschlüsselt usw.). Außerdem sollte man Mehrdeutigkeiten auflösen, beispielsweise kann man Bereichsgrenzen dahingehend präzisieren, dass man man erste oder letzte Werte benennt, um deutlich zu machen, ob es sich um inklusive oder exklusive Grenzen handelt. Problematisch sind doppelte Negationen; diese sollte man möglichst auflösen, da sie leicht zu Missverständnissen führen.

SWT1Dr. Jörg Mielebacher19

Lesbarer Code durch nützliche Kommentare

→ **Kommentare nur bei unklarem Code**

- Nicht intuitive bzw. unerwartete Lösungswege
- Wichtige Beobachtungen (z.B. lange Laufzeit)
- Annahmen, Vorbedingungen, Nachbedingungen etc.
- Frühere, nicht geeignete Lösungswege
- Ein-/Ausgabebeispiele, Grenzfälle usw.
- Ggf. Zusammenfassungen von Dateiinhalten

Grundsätzlich sollte nicht jede Zeile kommentiert werden, sondern nur Bereiche, die unklar sind. Dies betrifft unerwartete Lösungswege oder unerwartetes Programmverhalten, aber auch wichtige Annahmen, Vorbedingungen, Nachbedingungen usw. Hilfreich – und meist nützlicher als lange Ausführungen – sind Ein-/Ausgabebeispiele (z.B. $\text{ratio}(1,0) = 0$) oder Grenzfälle (0, Randwerte, negative Werte, Definitionslücken usw.). Für die Einarbeitung in eine bestehende Codebasis kann es zweckmäßig sein, Zusammenfassungen von Dateiinhalten an den Dateianfang zu stellen (Was ist in der Datei zu finden? Welche Abhängigkeiten bestehen? usw.).

SWT1Dr. Jörg Mielebacher20

Lesbarer Code durch verständliche Lösungen

- Single Responsibility Principle
- Modularisierung
- Lange Ausdrücke in benannte Teilergebnisse zerlegen
- Methoden frühzeitig verlassen
- Tiefe Schachtelung vermeiden
- ➔ Häufige Sonderfälle sprechen für ungeeigneten Ansatz
- ➔ Von Zeit zu Zeit von oben auf das Problem schauen

Lösungswege verständlich zu formulieren, kann – gerade bei schwierigen Problemen – eine Herausforderung sein. Meist lässt sich das auf zwei Ursachen zurückführen:

1.) Nähert man sich schrittweise einer Lösung, kann der entstandene Lösungsweg ein Stückwerk dieser Teilschritte sein. Meist neigen diese Lösungswege zu unangemessen vielen Sonderfällen und umständlichen Programmabläufen.

2.) Wenn man sich lange mit einer Problem beschäftigt hat, hat man das zu lösende Problem umfassender durchdringen, versteht Annahmen und Abhängigkeiten besser.

Die Verständlichkeit kann man beispielsweise erhöhen, indem man grundlegende Entwurfsprinzipien berücksichtigt, z.B. immer nur eine Aufgabe zur selben Zeit zu lösen, Codeabschnitten (z.B. Klassen) eindeutige Aufgaben zuweisen (und nicht viele verschiedene Aufgaben oder Teile davon), Code in kleinere Einheiten zerlegen (Modularisierung) usw. Aber auch lokal lässt sich viel Verständlichkeit gewinnen, z.B. indem man große Berechnungsausdrücke in kleinere Ausdrücke aufteilt (evtl. mit Variablen für Zwischenergebnisse), indem man Funktionen/Methoden möglichst früh verlässt usw. Die wichtigste Empfehlung ist aber, von Zeit zu Zeit einen Schritt aus dem Problem heraus zu machen und mit einem frischen Blick den gefundenen Lösungsweg zu hinterfragen; meist erkennt man dann nämlich einen einfacheren Ansatz.

SWT1Dr. Jörg Mielebacher21

Zwischenfragen

- Wie finden Sie Fehler in Ihren Programmen?
 - Debugger
 - Paarprogrammierung (einer sagt dem anderen, was er tippen soll)
 - Rubber-Duck-Debug

Wenn Sie programmieren, sind Sie immer mit Situationen konfrontiert, in denen das Programm nicht das macht, was es (ihre Meinung nach) sollte. Wie kommen Sie den Ursachen solcher Fehler auf die Schliche?

SWT1Dr. Jörg Mielebacher22

Fehlersuche während der Implementierung

- Meldungen des Compilers
- Stacktraces
- Assertions
- Debugging
- Logging
- Softwaretests

Während der Implementierung muss man bereits Fehler erkennen und beseitigen, geeignete Ansätze hierfür sind:

1.) Meldungen des Compilers:

Die Fehlermeldungen des Compilers sind wichtig für die Fehlersuche. Sie liefern mehr oder weniger konkrete Informationen über den Fehler vor allem aber beziehen sich die Meldungen auf einen bestimmten Quellcodeabschnitt (Datei + Zeile). Die IDE erlaubt das Anklicken solcher Fehler, um den jeweiligen Codeabschnitt anzuzeigen.

2.) Stacktraces:

Stacktraces liefern genaue Informationen darüber, welche Aufrufe zum Absturz des Programms geführt haben. Sie werden normalerweise auf der Konsole ausgegeben, beispielsweise wenn eine Exception auftritt:

3.) Assertions:

Assertions prüfen, ob bestimmte Zusicherungen zu einem gegebenen Punkt erfüllt sind, wenn nicht, wird eine entsprechende Fehlermeldung ausgegeben.

4.) Debugging:

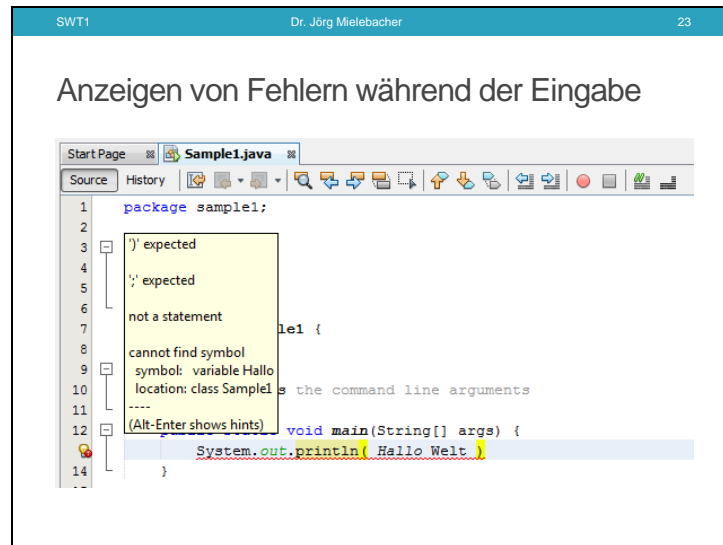
Debugger erlauben es die Programmausführung an Haltepunkten zu unterbrechen, schrittweise auszuführen, Variablen zu beobachten und weitere Informationen abzurufen.

5.) Logging:

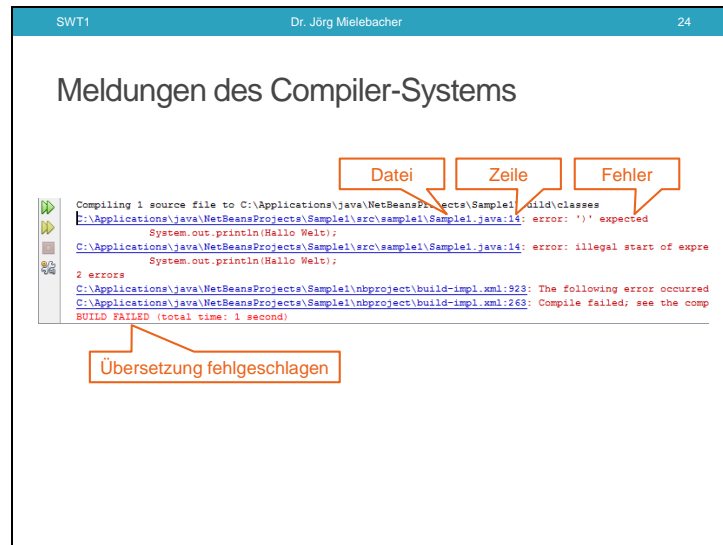
Ist ein Anwendungslog vorhanden, so lassen sich Fehler meist anhand dieses Logs untersuchen.

6.) Softwaretests:

Das Einrichten und Durchführen von Softwaretests ist ein unverzichtbarer Teil der Softwareentwicklung. Dabei werden erwartete Ergebnisse mit den tatsächlichen verglichen und bei Abweichungen als Fehler ausgegeben. Mehr dazu im Kapitel über die Testphase.



Die meisten IDEs zeigen syntaktische Fehler bereits während der Eingabe an. Diese werden dann farblich hervorgehoben; eine Fehlermeldung lässt sich bei Bedarf anzeigen. Diese Hinweise sollen Fehler bereits während der Eingabe vermeiden, d.h. man erhält Meldungen des Compilersystems früher als sonst.



Das Übersetzen des Projekts („Build“) schließt mit ein, dass der Compiler (javac) den Quellcode auf syntaktische Korrektheit, beispielsweise, ob alle verwendeten Variablen auch tatsächlich bekannt sind oder ob die Klammerung korrekt ist.

Stößt der Compiler auf Fehler, so gilt der Übersetzungsvorgang als fehlgeschlagen, d.h. es entsteht kein ausführbarer Bytecode. Alle Fehler und Auffälligkeiten (z.B. Hinweise bei veralteten – deprecated - Sprachkonstrukten) werden in der IDE angezeigt. Zu jeder Meldung werden Datei, Zeile und Fehlermeldung ausgegeben. Durch Anklicken gelangt man zu der betreffenden Quellcodestelle.

Solche Meldungen haben den großen Teil, dass sie klar erkennen lassen, wo genau ein Fehler im Quellcode gefunden wurde.

SWT1 Dr. Jörg Mielebacher 25

Stacktraces

```
public class Sample1 {  
    public static double calc2() {  
        return 1/0;  
    }  
    public static double calc1() {  
        return calc2();  
    }  
    public static void main(String[] args) {  
        System.out.println( calc1() );  
    }  
}
```

Output - Sample1 (run)

run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at sample1.Sample1.calc2(Sample1.java:12)
 at sample1.Sample1.calc1(Sample1.java:16)
 at sample1.Sample1.main(Sample1.java:20)
Java Result: 1


Division durch 0

Stacktraces liefern wichtige Informationen bei Laufzeitfehlern (hier: Division durch 0). Neben der Fehlerbeschreibung zeigen sie, wo der Fehler auftrat und auf welchem Weg diese Stelle des Quellcodes erreicht wurde.

SWT1Dr. Jörg Mielebacher26

Assertions

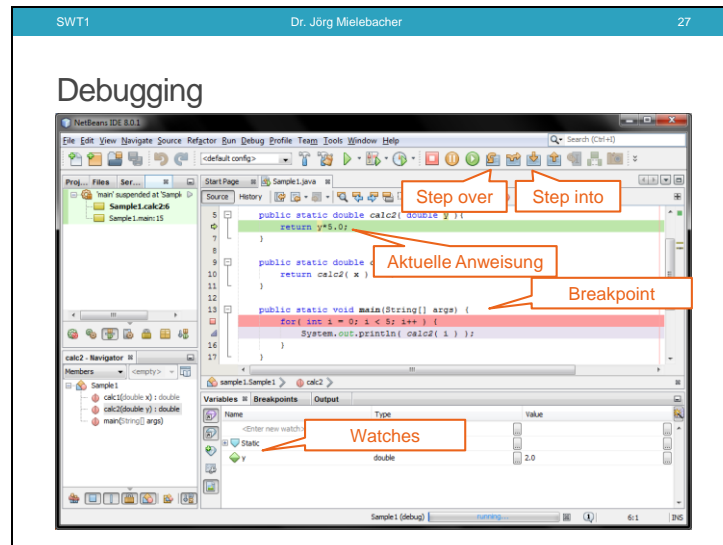
```
assert i >= 0 : "Invalid index"
```



Bedingung

Meldung

Assertions erlauben es, im Quellcode Voraussetzungen zu prüfen. Ist eine solche Voraussetzung (z.B. der erlaubte Wertebereich eines Parameters) nicht erfüllt, bricht das Programm mit einer entsprechenden Meldung ab – aber nur dann, wenn beim Starten von Java die Option `-ea` übergeben wurde. Dieser Aspekt ist besonders interessant, da man Assertions zentral aktivieren und deaktivieren kann. Deaktiviert verursachen sie keinen zusätzlichen Aufwand, verbleiben jedoch im Quellcode und dokumentieren dennoch die Voraussetzungen.



Debugger sind wichtige Werkzeuge für die Fehlersuche. Sie erlauben es, das Programm bis zu Haltepunkten („Breakpoints“) auszuführen, das Programm schrittweise auszuführen („Step over“ verfolgt Aufrufe nicht, „Step into“ springt in aufgerufene Methoden) und Variablenwerte anzusehen („Watches“). Meist bieten sie noch wesentlich mehr Möglichkeiten (Hardware usw.). In Java verwendet man üblicherweise jdb, der von den IDEs eingebunden ist, so lässt sich der Debugger bequem über die IDE steuern.

SWT1 Dr. Jörg Mielebacher 28

Logging

```
Logger log=Logger.getLogger(Sample1.class.getName());  
...  
log.log( Level.SEVERE, "Fehler ..." );
```

The diagram illustrates the logging process. A box on the left lists the logging levels: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, and FINEST. An arrow points from the SEVERE level to a box on the right labeled 'Meldung' (Message). Another arrow points from the 'Fehler ...' string in the code to the same 'Meldung' box, indicating that the message is associated with the log level.

Logging ist wesentlich für Software, da man hierdurch Abläufe auch ohne Debugger (z.B. während des Produktiveinsatzes) protokollieren und untersuchen kann.

Java bietet mit dem Logging-Package eigene Klassen, die für das Logging verwendet werden koennen. Sehr verbreitet sind jedoch Logging-Frameworks, vor allem Log4j.

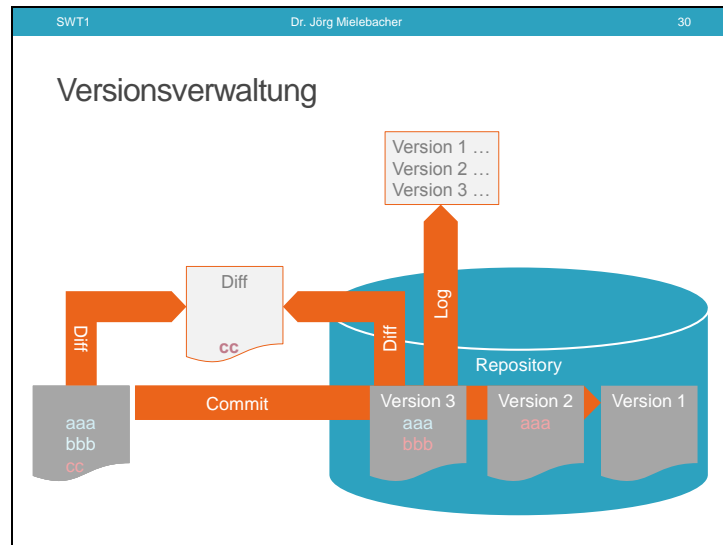
Allgemein bieten solche Logging-Klassen die Möglichkeit, Meldungen auf verschiedenen Kanälen auszugeben, z.B. in Dateien oder in Datenbanken. Dabei kann man beim Erstellen der Meldung festlegen, welchem Schweregrad sie zugeordnet ist; typisch ist die Unterscheidung in SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST. Zusätzlich legt man Fehlermeldungen fest und kann ggf. einen Stacktrace mit ausgeben.

SWT1Dr. Jörg Mielebacher29

Zwischenfragen

- Wie finden Sie heraus, was Sie wann am Quellcode verändert haben?
- Was machen Sie, wenn Sie zu einer früheren Fassung des Quellcodes zurückkehren wollen?

Stellen Sie sich vor, Sie erstellen eine Software. Der Quellcode umfasst mehrere Dateien, die Sie auf Ihrem Computer gespeichert haben. Wie behalten Sie den Überblick, wann Sie was an welcher Datei geändert haben? Was würden Sie machen, wenn Sie bei einer Änderung größere Fehler gemacht haben und zu einer früheren Fassung des Quellcodes zurückkehren wollen?



In Softwareprojekten arbeiten meist mehrere Entwickler zusammen. Nacheinander oder zeitgleich öffnen und verändern sie die Dateien des Quellcodes. Hierbei bestehen meist die folgenden Anforderungen:

- 1.) Änderungen an Dateien sollen nachvollziehbar protokolliert sein (Datei, Benutzer, Datum/Uhrzeit, Änderungen)
- 2.) Änderungen sollen so koordiniert werden, dass sie sich nicht gegenseitig auslöschen. Beispielsweise durch Sperren von Dateien oder durch das Zusammenführen von Änderungen.
- 3.) Alle Entwickler sollen stets Zugriff auf den aktuellen Stand sowie auf frühere Stände des Quellcodes haben und diese ggf. wiederherstellen können (Archivierung).
- 4.) Es muss möglich sein, unterschiedliche Entwicklungszweige (Branches) eines Projekts zu verwalten (z. B. kundenspezifische Anpassungen, Testversionen usw.).

Um diesen Anforderungen gerecht zu werden, setzt man Werkzeuge für die Versionsverwaltung ein. Hierzu zählen Git, Subversion, Visual SourceSafe.

Im Folgenden wird beispielhaft die Verwendung der freien Versionsverwaltung Git vorgestellt. Git unterstützt Protokollierung, Koordination, Archivierung, Wiederherstellung und die Verwaltung von Branches. Besonders ist dabei, dass es sich um eine sog. verteilte Versionsverwaltung handelt. D. h. alle Entwickler haben stets das gesamte Repository, d. h. den gesamten Quellcode mit Änderungshistorie, auf ihrem Computer. Deshalb ist für die meisten Aktivitäten kein Netzwerkzugriff notwendig. Von Zeit zu Zeit werden diese lokalen Repositories synchronisiert, d. h. auf denselben Stand gebracht. Wurde dieselbe Datei geändert, versucht das System, diese Änderungen zusammenzuführen. Gelingt dies nicht, müssen die Entwickler den sog. Konflikt manuell auflösen, indem sie den Quellcode anpassen. Man bezeichnet dieses Vorgehen als „Copy Modify Merge“, d.h. alle Entwickler erhalten eine Kopie, jeder kann daran ändern, anschließend werden die Änderungen zusammengeführt. Andere Versionsverwaltungen setzen

statt dessen auf das Sperren von Dateien, d. h. solange ein Entwickler die Datei zur Bearbeitung gesperrt hat, können andere keine Änderungen vornehmen („Lock Modify Write“).

Git steht unter verschiedenen Betriebssystemen zur Verfügung. Es handelt sich um ein kommandozeilenbasiertes Programm. Allerdings existieren grafische Oberflächen (TortoiseGit). Diese bieten z. B. unter Windows auch eine Integration in den Windows Explorer an, wodurch man geänderte Dateien direkt an einem entsprechenden Symbol erkennen kann. Die Git-Operationen lassen sich dann auch über das Kontextmenü (Rechtsklick) aufrufen.

Wichtige Operationen in Git sind:

Add: Eine neue Datei in die Versionsverwaltung des Projekts aufnehmen

Commit: Änderungen an einer Datei (oder an mehreren) bestätigen und in das Repository übernehmen.

Revert: Änderungen verwerfen und zu einem früheren Stand zurückwechseln

Show log: Änderungshistorie der Datei abrufen.

Diff: Datei mit einem früheren Stand vergleichen und Änderungen hervorheben

Clone: Lokales Repository als Kopie eines entfernten Repositories erstellen

Push: Lokale Änderungen in das entfernte Repository laden

Push: Änderungen des entfernten Repositories in das lokale Repository einfügen

SWT1

Dr. Jörg Mielebacher

31

Beispiel: Lokales Git-Repository anlegen

```
im@PERSEUS /c/test
$ git init
Initialized empty Git repository in c:/test/.git/

im@PERSEUS /c/test (master)
$ git add .

im@PERSEUS /c/test (master)
$ git commit
[master (root-commit) 3aa7c48] Erste Version
2 files changed, 2 insertions(+)
create mode 100644 test1.txt
create mode 100644 test2.txt

im@PERSEUS /c/test (master)
$
```

Leeres Repository anlegen

Alles im Verzeichnis versionieren

Aktuellen Stand sichern

Im hier gezeigten Beispiel soll der Inhalt eines Verzeichnisses (die Dateien test1.txt und test2.txt) unter Versionsverwaltung gestellt werden. Hierzu wechselt man in diesen Ordner und legt mit git init ein leeres Repository an. Diesem fügt man mit git add . alle vorhandenen Dateien hinzu und übernimmt den aktuellen Inhalt mit git commit als aktuelle Version. Dazu muss man eine sog. Commit-Message eingeben, die künftig zu diesem Commit-Vorgang (d.h. zu dieser Version) angezeigt wird.

SWT1 Dr. Jörg Mielebacher 32

Beispiel: Lokales Git-Repository anlegen

```
im@PERSEUS /c/test (master)
$ git diff
diff --git a/test1.txt b/test1.txt
index 496c613..55559ac 100644
--- a/test1.txt
+++ b/test1.txt
@@ -1,3 @@
-Datei 1
\ No newline at end of file
+Datei 1MM
\ No newline at end of file
im@PERSEUS /c/test (master)
$
```

Änderungen zur letzten Version

Letzter Commit

Aktuelle Bearbeitung

Bearbeitet man nun die Dateien (hier: Text in test1.txt eingefügt), kann man sich bei Bedarf mit `git diff` den Unterschied der Dateien zum letzten Commit anzeigen.

SWT1 Dr. Jörg Mielebacher 33

Beispiel: Lokales Git-Repository anlegen

```
joe@PERSEUS /c/test (master)
$ git commit -a
[master 0fdd230] Inhalt eingefügt
1 file changed, 3 insertions(+), 1 deletion(-)

joe@PERSEUS /c/test (master)
$ git log
commit 0fdd2308cf5db2fc50c9e9629c9b47e06e1801a0
Author: Jörg Mielebacher <mail@mielebacher.de>
Date:   Sun Nov 16 21:55:00 2014 +0100

    Inhalt eingefügt

Datei enthält jetzt ...

commit 3aa7c483aef3b4b21a0a0f6f9fdca4d05e758adb
Author: Jörg Mielebacher <mail@mielebacher.de>
Date:   Sun Nov 16 21:40:11 2014 +0100

    Erste Version

    Zwei leere Dateien vorhanden.

joe@PERSEUS /c/test (master)
```

Aktuellen Stand sichern

Bisherige Commits anzeigen

Soll der bearbeitete Stand übernommen werden, ist dies am einfachsten mit `git commit -a` möglich. Dabei werden alle Änderungen in allen Dateien übernommen.

Möchte man die bislang durchgeführten Änderungen ansehen, kann man hierzu `git log` einsetzen.

SWT1 Dr. Jörg Mielebacher 34

Beispiel: Lokales Git-Repository anlegen

```
PERSEUS /c/test (master)
$ git diff
diff --git a/test1.txt b/test1.txt
index 5559ac..d17ae95 100644
--- a/test1.txt
+++ b/test1.txt
@@ -1,3 +1,5 @@
Datei 1

Inhalt...
\ No newline at end of file
msim AM
Inhalt...AM

PERSEUS /c/test (master)
$ git checkout -- test1.txt

PERSEUS /c/test (master)
$ git diff

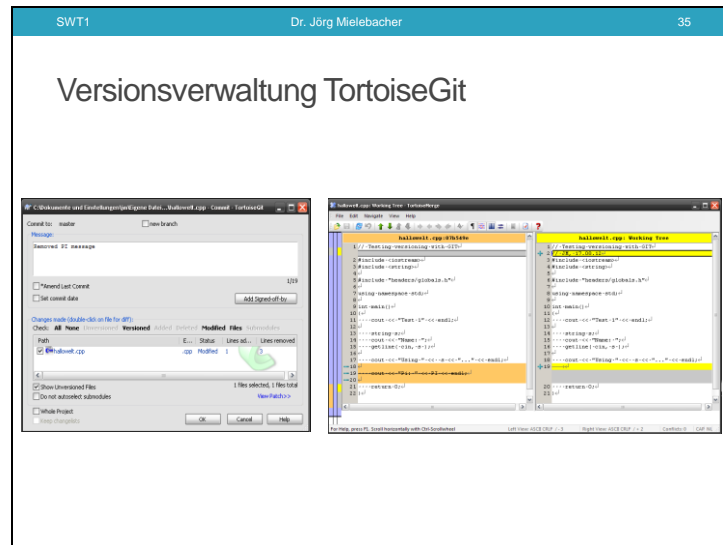
PERSEUS /c/test (master)
```

Änderungen zur letzten Version

Änderungen rückgängig machen

Änderungen sind gelöscht

Stellt man bei der Bearbeitung fest, dass man Änderungen verwerfen möchte (d.h. man möchte zum letzten Commit zurückkehren). Kann man dies mit `git checkout -- <Datei>` erreichen. Allerdings wird dabei der Dateiinhalt unwiderruflich überschrieben. Möchte man die Änderungen dennoch bewahren (um z.B. spätere Vergleiche zu haben oder Lösungsansätze wieder aufzufreien), gibt es geeignetere Lösungen (z.B. Stash).





Die gerade gezeigte Verwendung von Git in der Shell ist für routinierte Nutzer vertraut und bietet darüber hinaus Vorteile, weil Schritte automatisiert werden können (z.B. im Rahmen des Build-Managements). In vielen Fällen ist die Nutzung einer grafischen Oberfläche komfortabler. TortoiseGit ist eine solche grafische Oberfläche für Git. Unter Windows integriert sich TortoiseGit in den Explorer, d.h. man kann Git-Funktionen durch das Kontextmenü aufrufen. Die Screenshots zeigen das Commit-Fenster (li.) und das Ergebnis eines diff-Vergleichs (re.).


Die meisten IDEs bieten darüber hinaus die Möglichkeit, Versionierungswerkzeuge als Plug-in einzubinden.

SWT1 Dr. Jörg Mielebacher 36

Grundsätze der Quellcodedokumentation

 Selbsterklärender Quellcode

 Dokumentation im Quellcode

 Zusammenfassen mit Dokumentationsgeneratoren

Der Quellcodedokumentation liegen heute vor allem drei Grundsätze zugrunde:

1.) Der Quellcode wird möglichst selbsterklärend formuliert:

Hierzu zählen aussagekräftige Bezeichner (Variablenamen, Methodennamen usw., die Inhalt/Verwendung erkennen lassen) und ein einheitliches Quellcode-Layout, z.B. Einrückungen, Abstände usw., aber auch Ordnerstruktur und Dateinamen (z.B. bei Header-Dateien). Intuitiv nachvoll-ziehbare Lösungswege erhöhen die Aussagekraft. Eingebettete Testfälle und Assertions unterstützen das Verständnis von Sonderfällen und kritischen Bedingungen.

2.) Die Dokumentation erfolgt im Quellcode:

Sind Hinweise erforderlich, um den Quellcode zu verstehen – d.h. der Quellcode alleine ist nicht aussagekräftig genug - werden diese in Form von Kommentaren direkt in den betreffenden Abschnitten eingefügt. Gleiches gilt für Aufgaben („To-Do“), Verbesserungsvorschläge oder vermutete Schwachstellen und Fehler, die nicht sofort beseitigt werden können. Die Dokumentation im Quellcode ist schnell zu erfassen und verhindert ungültige Verweise zwischen Dokumentation und Quellcode. Quellcode und Dokumentation werden auf diese Weise immer zeitgleich und damit konsistent verteilt.

3.) Zusammenfassungen werden durch Dokumentationsgeneratoren erzeugt:

Da die technische Dokumentation ohnehin fast immer elektronisch verwendet wird (z.B. PDF- oder HTML-Dateien), setzt man Generatoren ein, um diese Doku-men-tation zu erzeugen (z.B. javadoc oder Doxygen). Sie durchsuchen den Quellcode nach bestimmten Schlüsselworten – meist in Kommentaren - und extrahieren dann die betreffenden Informationen und fassen sie gemäß einer Dokumentenvorlage zusammen.

Die Quellcodedokumentation bündelt folglich eine Reihe wichtiger Informationen:

- Erläuterungen zu Quellcode-abschnitten

- projektbezogene Angaben (Autor, Version, Lizenzen, Aufgaben, Vorschläge usw.)
- Anweisungen für Dokumentationsgeneratoren für Zusammenfassungen

Abhängig von der verwendeten Versionsverwaltung besteht die Möglichkeit, Teile der Quellcodedokumentation aus der Versionsverwaltung zu übernehmen, z.B. die Änderungshistorie, Versionsnummern, Autornamen usw. Dies bezeichnet man als „Keyword substitution“ und wird beispielsweise von Subversion unterstützt.

SWT1
Dr. Jörg Mielebacher
37

Dokumentationsgenerator javadoc

```

/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}

```

Dokumentationsgeneratoren wie javadoc und Doxygen (www.doxygen.org) haben die Aufgabe aus Kommentaren in den Quellcode Dateien Zusammenfassungen zu generieren, z.B. Klassenhierarchien usw. Die Kommentare im Quellcode müssen hierzu einen bestimmten Aufbau besitzen. Der Generator durchsucht dann den Quellcode nach solchen Kommentaren verdichtet sie zu dem gewünschten Dokument.

Doxygen ist sprachunabhängig und unterstützt die gängigen Programmiersprachen gleichermaßen, es kann sogar für weitere Sprachen erweitert werden. Javadoc ist stattdessen eng mit Java verbunden, weil es Teil des JDK ist.

Kommentare für javadoc werden mit `/**` eingeleitet und enden wie gewöhnlich mit `*/`. Der Textblock wird als Beschreibung interpretiert, er kann HTML-Code enthalten, der entsprechend umgesetzt wird (z.B. `<p>`, ``, `<code>`). In die Beschreibung können Verweise zu anderen Dokumentationsinhalten eingefügt werden, hier z.B. `{@link URL}`. Für ausgewählte Informationen stellt javadoc Tags zur Verfügung, die in den Kommentaren verwendet werden können, z.B. `@param` für Methodenparameter, `@return` für den Rückgabewert oder `@see` für Querverweise.

SWT1Dr. Jörg Mielebacher38

Dokumentationsgenerator javadoc

getImage

```
public Image getImage(URL url,
                       String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
`url` - an absolute URL giving the base location of the image.
`name` - the location of the image, relative to the `url` argument.

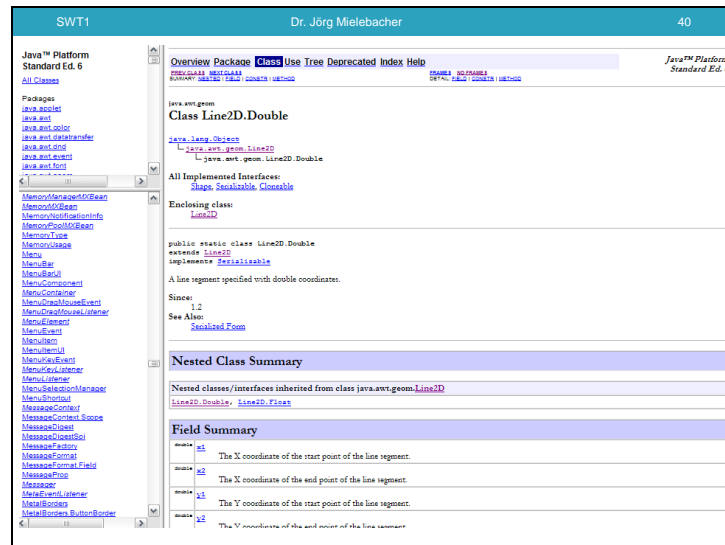
Returns:
the image at the specified URL.

See Also:
`Image`

Hier die entsprechende HTML-Ausgabe zu dem vorangegangenen Dokumentationskommentar. Der Methodenname wird automatisch übernommen, wie auch der Zugriffsmodifikator und die gesamte Methodensignatur. Den `@param`-Tags entsprechen die beiden Zeilen im Abschnitt „Parameters:“, dem `@return`-Tag entspricht der Abschnitt „Returns:“, dem `@see`-Tag „See Also:“.

SWT1	Dr. Jörg Mielebacher	39
Dokumentationsgenerator javadoc		
@author	Name des Autors	
@version	Version von Klassen und Interfaces	
@param	Methodenparameter	
@return	Rückgabewert von Methoden	
@exception	Exceptions, die auftreten können	
@see	Querverweise	

Javadoc unterstützt eine Reihe von Tags; sie sollten in der hier gezeigten Reihenfolge beschrieben werden. Für @param sind auch mehrere Einträge möglich – bevorzugt in der Reihenfolge der Parameter. Auch @author, @see und @exception kann mehrfach notiert werden.



Dieses Beispiel zeigt eine Ausgabe von javadoc – wie Sie sehen, wird javadoc auch zur API-Dokumentation von Java verwendet. Für eine Klasse wird beispielsweise die Vererbungshierarchie angezeigt, die implementierten Interfaces sowie Attribute und Methoden samt Beschreibung. Die HTML-Ausgabe nutzt intensiv die Möglichkeit der Verlinkung.

Andere Ausgabeformate (auch erweiterbar durch sog. Doclets) sind LaTeX, XML usw.

SWT1Dr. Jörg Mielebacher41

Build-Management-Tools

- Apache ANT (Java)
- Apache Maven (Java)
- make (C/C++)

Build-Management-Tools steuern die vielfältigen Aufgaben, die rund um die Übersetzung des Quellcodes notwendig sind, z.B. Übersetzen, Testen, Packen usw. Hierzu definiert man durch Konfigurationsdateien projektspezifische Abläufe - im Wesentlichen die Aufrufe der notwendigen Werkzeuge.

Apache ANT und Apache Maven sind verbreitete Build-Management-Tools für Java, während make (und seine Varianten) für C und C++ sehr verbreitet ist.

SWT1Dr. Jörg Mielebacher42

Weitere Werkzeuge

- Stylechecker
- Codeanalyse
- Zusammenarbeit und Projektmanagement
- Emulatoren
- usw.

Neben den genannten Werkzeugen kommen je nach Art des Entwicklungsvorhabens zahlreiche weitere Werkzeuge zum Einsatz: Stylechecker zur Prüfung des Quellcodeaufbaus, Codeanalyse zur Erkennung typischer Fehler und zur Ermittlung von Kennzahlen, Werkzeuge für die Zusammenarbeit (Chat, Wiki usw.) und für das Projektmanagement (Aufgabenverwaltung, Zeiterfassung, Budgetkontrolle usw.). Emulatoren werden benötigt, wenn Software für besondere Hardware entwickelt wird (z.B. für Mobiltelefone) oder die Zielplattform von der Entwicklungsplattform abweicht (z.B. Entwicklung von Android-Software unter Windows). Solche Emulatoren bilden virtuell eine bestimmte Hard- und Software nach; sie erlauben damit ein erstes Testen von Software.

SWT1Dr. Jörg Mielebacher43

Empfehlungen

- Intuitive Lösungsansätze wählen
- Inkrementell entwickeln
- Übersichtlicher Quellcode
- Übersichtliche Methoden (<30 Zeilen)
- Selbsterklärende Bezeichner
- Keine tiefe Schachtelung
- Konstanten und Enumerations verwenden
- Keine globalen Ressourcen verwenden
- Geeignete Datenstrukturen verwenden
- Schlanke Schnittstellen
- Konstruktoren nutzen, um eindeutigen Objektzustand zu erhalten
- Bewährtes wiederverwenden
- Prozesse mit geeigneten Werkzeugen automatisieren
- Defensiv programmieren (Logging, Assertions, Vor-/Nachbedingungen usw.)

Die hier gezeigten Liste von Empfehlungen ließe sich noch beliebig verlängern. Sie besitzt eine Reihe von Kernaussagen:

- 1.) Bei der Entwicklung geht es immer um Verständlichkeit, da sich hierdurch aufwendiges Einarbeiten und Fehler vermeiden lassen.
- 2.) Bewährtes sollte wiederverwendet werden, was auch bedeuten kann, ein Framework oder dergleichen einzusetzen.
- 3.) Modularisierung und Kapselung sollen auf allen Ebenen eingesetzt werden, um verständliche, testbare und wiederverwendbare Software zu erhalten.
- 4.) Man programmiert defensiv, d.h. die Möglichkeiten der Fehlervermeidung und –suche werden umgesetzt (z.B. Assertions, Logging usw.).
- 5.) Man sollte die Fülle und die Unterstützung der verfügbaren Werkzeugen und wiederkehrende (oder fehlerträchtige) Abläufe automatisieren.

SWT1	Dr. Jörg Mielebacher	44
<h2>Zusammenfassung</h2> <ul style="list-style-type: none">▪ Der Quellcode und damit verbundene Ergebnisse werden auf Grundlage des Entwurfs erstellt.▪ Eine IDE bietet Editorfunktionalität sowie die Anbindung der relevanten Werkzeuge.▪ Styleguides beschreiben den Aufbau des Quellcodes.▪ Eine Versionsverwaltung speichert die Änderungen an allen Quellcodedateien und steuert den Zugriff durch mehrere Entwickler.▪ Quellcode sollte selbsterklärend sein, relevante Dokumentation enthalten und für Dokumentationsgeneratoren vorbereitet sein.		

SWT1	Dr. Jörg Mielebacher	45
<h3>Aufgabe 08.1</h3> <ul style="list-style-type: none">▪ Beschreiben Sie die Aufgaben einer IDE.▪ Beschreiben Sie die notwendigen Schritte, um ein Java-Programm zu erstellen und auszuführen. Welche Programme sind hierfür notwendig?▪ Nennen Sie Funktionen der IDE, die Fehler bei der Implementierung verhindern.▪ Nennen Sie Beispiele für Build-Management-Tools.▪ Nennen Sie Beispiele für Dokumentationsgeneratoren.▪ Geben Sie einen beispielhaften javadoc-Methodenkommentar an und beschreiben Sie die Tags.		

Bitte bearbeiten Sie die Aufgaben.

SWT1	Dr. Jörg Mielebacher	46
<h3>Aufgabe 08.2</h3> <ul style="list-style-type: none">▪ Auf welche Weise könnte man Fehler beim Zugriff auf ein Array erkennen und verhindern?▪ Auf welche Weise lassen sich Fehler untersuchen, die zur Laufzeit beim Kunden auftreten und von diesem gemeldet werden?		

Bitte bearbeiten Sie die Aufgaben.

SWT1	Dr. Jörg Mielebacher	47
<h3>Aufgabe 08.3</h3> <ul style="list-style-type: none">▪ Sie arbeiten zu zweit an einer Quellcodedatei, die von Git verwaltet wird. Was geschieht, wenn Sie beide einen Commit durchführen und Änderungen an derselben Stelle des Quellcodes durchgeführt haben?▪ Was ist der wichtigste Unterschied zwischen Git und Subversion?		

Bitte bearbeiten Sie die Aufgaben.