Elementare Softwaretechnik

In vielen Definitionen von Softwaretechnik wird darauf verwiesen, dass es bei Softwaretechnik um das Konstruieren *großer* Softwaresysteme geht. Und in der Tat treten natürlich viele Fragen und Probleme erst auf, wenn man Software arbeitsteilig in verschiedenen Teams entwickelt.

Allerdings gibt es durchaus Vorgehensweisen der Softwaretechnik, die auch dann angewandt werden können, wenn man als Einzelperson oder in einem kleinen Team Software entwickelt.

Und: wenn Softwaretechnik für große Projekte erforderlich ist, dann müssten doch sinnvoll anwenderbare Techniken auch für kleine Projekte geeignet sein. Was im Großen erforderlich ist, ist im Kleinen doch sicherlich auch vernünftig.

Und in der Tat. In dieser Vorlesung behandeln wir Techniken der Softwareentwicklung, die man *auf jeden Fall* anwenden sollte — ganz gleich, wie groß ein Vorhaben ist: Elementare Softwaretechnik

Codierrichtlinie

Warum Codierrichtlinie?

Programmcode wird öfter gelesen als geschrieben!

Guter Code ist

- leicht zu lesen — Layout, Konsistenz, „Ästhetik“
- durchschaubar — durchsichtige Konstrukte, aussagekräftige Namen
- nachvollziehbar — geeignete Kommentare
- wartungsfreundlich — gemeinsamer Stil im Team erleichtert Zusammenarbeit ungemein
- weiterverwendbar — Muster erlauben Automatisierung siehe z.B. javadoc

Beispiele für Codierrichtlinien

- Suns Richtlinien für Java — Code Conventions for the Java Programming Language, siehe http://www.oracle.com/technetwork/java/codeconv-138413.html

- Doug Leas „Draft Java Coding Standard“ — siehe http://gee.cs.oswego.edu/dl/html/javaCodingStd.html
- Java-Codierrichtlinien für den Fachbereich MNI — siehe http://homepages.thm.de/~hg11260/mat/MNIStandard.pdf
- Physikalisch-Technische Bundesanstalt „Richtlinie für die Programmierung in C++“
- Taligent's Guide to Designing Programs — online http://pcroot.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html

Inhalt von Codierrichtlinien

Festgelegt werden

- Organisation der Quellen in Dateien, Pakete u.ä.
- Kommentierung, Layout der Quellen
- Namenskonventionen
- Spezifikation von Schnittstellen
- Fehlerbehandlung
- auch grundlegende Designregeln

Die Sache geht aber über das Einhalten von (mehr oder weniger) formalen Regeln weit hinaus. Ein gutes Team entwickelt einen gemeinsamen Stil. Es wird auf eine bestimmte Art entwickelt: modularisiert und programmiert. Dies erleichtert das gemeinsame Arbeiten und das Reinkommen in den Code anderer ungemein. Es fördert die Qualität, weil durch erprobte Konstrukte viel weniger Fehler gemacht werden.

Eine gute Codierrichtlinie wird so zu einer Richtlinie für den Entwurf „im Kleinen“ — ein gutes Beispiel dafür ist der erwähnte Taligent Style Guide.

Beispiel für Codierstil

Folgendes Codestück stamt aus dem Open Source Projekt *xalan*, dem XSLT-Prozessor von Apache:

```
public class FuncQname extends Function
{
  public XObject execute(XPath path, XPathSupport execContext,
  Node context, int opPos, Vector args) throws org.xml.sax.SAXException
```

```
  {
    int nArgs = args.size();
    if(nArgs > 0)
    {
      if(nArgs > 1)
      path.error(context, XPATHErrorResources.ER_NAME_HAS_TOO_MANY_ARGS);
      NodeList nl = ((XObject)args.elementAt(0)).nodeset();
      context = (nl.getLength() > 0) ? nl.item(0) : null;
    }
    XObject val;
    if(null != context)
    {
      int ntype = context.getNodeType();
      if((ntype == Node.ATTRIBUTE_NODE) || (ntype == Node.ELEMENT_NODE)
         || (ntype == Node.PROCESSING_INSTRUCTION_NODE))
      {
        String qname = (Node.ATTRIBUTE_NODE == ntype) ?
                       ((Attr)context).getName() : context.getNodeName();
        val = new XString(qname);
      }
      else { val = new XString(""); }
    }
    else { val = new XString(""); }
    return val;
  }
}
```

Eine andere Art, den Code zu strukturieren ist:

```
public class FuncQname extends Function
{
  public XObject execute(XPath path, XPathSupport execContext,
    Node context, int opPos, Vector args) throws org.xml.sax.SAXException
  {
    int nArgs = args.size();
    if(nArgs > 0)
    {
      if(nArgs > 1)
      path.error(context, XPATHErrorResources.ER_NAME_HAS_TOO_MANY_ARGS);
      NodeList nl = ((XObject)args.elementAt(0)).nodeset();
      context = (nl.getLength() > 0) ? nl.item(0) : null;
    }

    // kein Kontext
    if(context == null)
      return new XString("");

    int ntype = context.getNodeType();

    // Attribut
    if(ntype == Node.ATTRIBUTE_NODE)
```

```
      return new XString(((Attr)context).getName());

    // Element oder ProcInstruction
    if((ntype == Node.ELEMENT_NODE)
              || (ntype == Node.PROCESSING_INSTRUCTION_NODE))
      return new XString(context.getNodeName());

    // sonst
    return new XString("");
  }
}
```

Worin besteht der Unterschied? Was sind Vorteile und Nachteile der beiden Arten?

Es ist offensichtlich, dass es vernünftig ist, sich in einem Team oder für ein gemeinsames Entwicklungsprojekt auf einen (geeigneten) gemeinsamen Stil zu vereinbaren!

CodeReview

Ein **Review** ist eine formell organisierte Überprüfung eines Arbeitsergebnisses (in unserem Kontext insbesondere von Quellcode) durch eine Gruppe von Gutachtern.

Man unterscheidet:

- Codeinspektion - Vorbereitetes Überprüfen des Codes und Befragung des Autors - Festhalten der problematischen Stellen
- Walkthrough - Autor stellt den Code vor - Zuhörer verfolgen und überprüfen die Darstellung

Grundlage des Reviews ist die Spezifikation, gegen die der Code überprüft wird.

Wie führt man ein Review durch?

Prüfpunkte – oft anhand einer Checkliste

- Datenfehler (Initialisierung, Längen etc.)
- Steuerungsfehler (Vollständige Bedingungen etc.)
- Ressourcenfehler (Allokation von Speicher, Dateien etc.)
- Fehlerbehandlung

Was kann man durch Reviews erreichen?

Ergebnisse

- Untersuchungen zeigen, dass 60% - 90% der Fehler durch Reviews gefunden werden können
- oft solche, die durch Test schwer zu finden wären
- Nebeneffekt: Wissenstransfer, gemeinsame Idiome

Exkurs: Die Rolle der Psychologie im Review

Selbstverständlich spielt die Psychologie in ein Review hinein. Deshalb ist wichtig, dass die Beteiligten ein Bewusstsein haben

- dass es um die Prüfung der *Sache* geht, nicht um die Person des Entwicklers
- dass es *viele* Wege gibt, Spezifikationen zu erfüllen
- dass *gemeinsame* Anstrengungen nötig sind, die Qualität des Codes zu verbessern
- dass das Entwicklen von Software *schwierig* ist, und es keinen einfachen und eindeutigen Weg gibt
- dass es darum geht, voneinander zu *lernen*

Manchmal ist es sinnvoll, einen externen Mediator einzusetzen, der das Codereview leitet.

Grundlagen des Softwaretests

Man unterscheidet

- Black-Box-Test: das zu testende Element wird von außen, in seiner Funktionsweise betrachtet, kein Blick in die Interna
- White-Box-Test: die Interna sind bekannt, also z.B. der Quellcode einer Klasse oder Methode. Diese Kenntnis wird zum Testen herangezogen

Elementare Testmethoden:

- Bilden von Äquivalenzklassen von Werten, Eingaben (Black-Box)

- Testen an Bereichsgrenzen (Black-Box)
- Überdeckung des Tests überlegen (White-Box)
- Erraten von Fehlern (Beides)

Literaturhinweis: James A. Whittaker gibt viele interessante Hinweise, wie man systematisch, aber auch durch Erraten Fehler in Software finden kann. Er nimmt sich dazu Produkte vor, die wir alle kennen. Das Buch:
James A Whittaker: *How to Break Software: A Practical Guide to Testing.* Addison-Wesley 2003.

Oft wird Test als eine Tätigkeit gesehen, die *nach* dem Entwickeln kommt. Das trifft natürlich einerseits zu, ist andererseits aber viel zu kurz gegriffen:

Integration des Tests in das Entwickeln! Nicht erst nachträglich. Extrem formuliert: Test first (XP).

JUnit eignet sich sehr gut als Werkzeug, das systematische (und halbautomatisierte) Testen von Java-Klassen in die Entwicklung zu integrieren. Einmal eingerichtet und erprobt, wird man schnell feststellen, wie das ständige parallele Testen beim Entwicklen sehr hilfreich ist. Insbesondere kann man recht leicht überprüfen, ob Änderungen den Code tatsächlich verbessert haben und nicht unerwartete Fehler oder Effekte an anderer Stelle zur Folge haben.

Eine Kurzanleitung für JUnit finden Sie auf der WebSeite zur Veranstaltung: http://homepages.thm.de/~hg11260/mat/junit.pdf.

Build-Prozess und Versionsverwaltung

Build-Prozess

Worum geht es beim Build-Prozess?

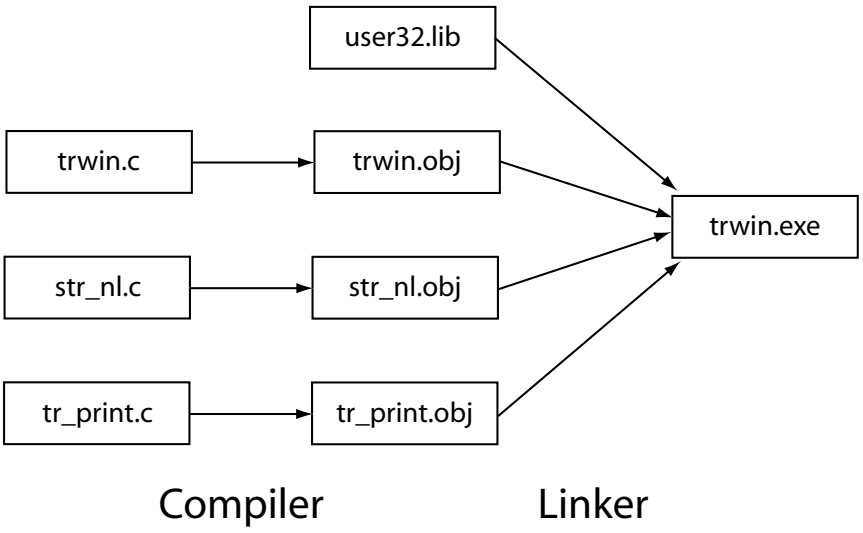
Der Build-Prozess ist das Herstellen aller Bestandteile und ihre Integration zu einem laufenden System

- Kontiniuerliche Prüfung des Entwicklungsstands (statt Big Bang)
- Erstellen testbarer Software, automatisierte Tests (JUnit)
- Inkrementelles Bauen und Ausliefern
- Stichwort „Continuous Integration“ (Microsoft, XP)
- Automatischer Build-Prozess („Daily Build“)

Um den Build-Prozess zu handhaben, setzt man Werkzeuge ein. Die elementaren Werkzeuge sind: make resp. ant oder Maven.

Wie funktioniert make?

Nehmen als Beispiel eines kleines Programms:



Das zugehörige makefile sieht etwa so aus:

```
# Makefile fuer trwin

CC  := cl -c
LD  := link

CFLAGS  = -nologo -W3 -GX -GB -MD -O2 -D"WIN32"
LDFLAGS = /nologo /subsystem:windows /machine:I386
LDFLAGS += /incremental:no /out:$@

%.obj: %.c
  $(CC) $(CFLAGS) -Fo"$@" -Tc $<
%.exe: %.obj
  $(LD) $(LDFLAGS) $(LDLIBS) $^

LDLIBS  += user32.lib kernel32.lib

trwin.exe: trwin.obj str_nl.obj tr_print.obj
trwin.obj: trwin.c
str_nl.obj: str_nl.c
tr_print.obj: tr_print.c
```

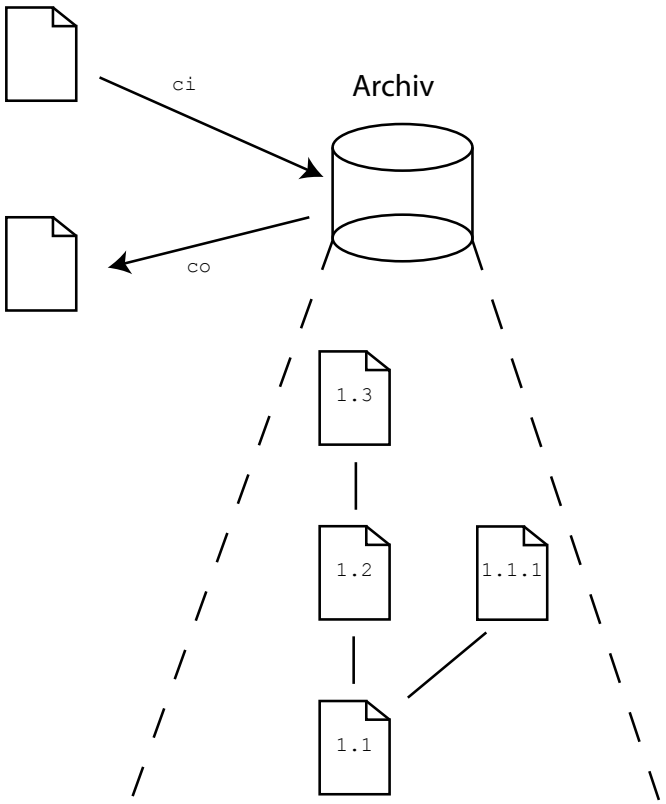
Eine ant-Datei zu einem Programm für einen Taschenrechner, bei dem der Java-Compiler-Compiler javacc eingesetzt wird, sieht so aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- ....................................................................
  Build file for mic

  Copyright (c) 2008
  by Fachhochschule Gießen-Friedberg  University of Applied Sciences.

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with the Alloy Analyzer; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
USA
  ....................................................................
  $Id: ant.lst 376 2010-12-16 08:36:18Z br $
  .................................................................... -->

<!-- .................................................................... -->

<project name="mic" default="compile-j" basedir=".">
  <description>Project MPA MNI Integer Calculator</description>
  <property name="pkg" value="mic"/> <!-- package name for mic -->
  <property name="src" location="${basedir}/src"/>
  <property name="build" location="${basedir}/build"/>
  <property name="dist" location="${basedir}/dist"/>
  <property name="doc" location="${basedir}/doc"/>
  <property name="lib" location="${basedir}/lib"/>

  <path id="project.classpath">
    <pathelement location="${build}"/>
    <fileset dir="${lib}"> <!-- 3rd party libraries -->
      <include name="*.jar"/>
    </fileset>
  </path>

  <target name="init">
    <tstamp/>
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>
```

```
  <target name="compile-parser" depends="init">
    <jjtree target="${src}/${pkg}/parser/MicParser.jjt"
      nodepackage="${pkg}.parser"
      outputdirectory="${src}/${pkg}/parser"
      javacchome="${lib}" />
    <javacc target="${src}/${pkg}/parser/MicParser.jj"
      outputdirectory="${src}/${pkg}/parser"
      javacchome="${lib}"/>
    <jjdoc target="${src}/${pkg}/parser/MicParser.jj"
      outputfile="${doc}/grammar.html"
      javacchome="${lib}"/>
  </target>

  <target name="compile-jjt" depends="compile-parser" />

  <target name="compile-j" depends="compile-jjt">
    <javac srcdir="${src}" destdir="${build}">
      <compilerarg value="-Xlint"/>
      <classpath refid="project.classpath"/>
    </javac>
  </target>

  <target name="dist" depends="compile-j">
    <mkdir dir="${dist}"/>
    <!-- put everything in ${build} into mic.jar file,
         first remove file if exists -->
    <manifest file="${build}/MANIFEST.MF">
      <attribute name="Built-By" value="Burkhardt Renz"/>
      <attribute name="Main-Class"
        value="${pkg}.cli.mic"/>
    </manifest>
    <jar jarfile="${dist}/mic.jar" manifest="${build}/MANIFEST.MF"
      basedir="${build}" />
  </target>

  <target name="clean" description="clean up">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
    <delete>
      <fileset dir="${src}/${pkg}/parser">
        <include name="**/*.jj"/>
        <include name="**/Mic*.java"/>
      </fileset>
    </delete>
  </target>
</project>
```

Versionsmanagement

Software entwickelt sich. Es ist die extrem seltene Ausnahme, dass eine Software einmal entwickelt und dann nie wieder verändert wird. Im Gegenteil: oft steckt ja bereits im Dateinamen eines Auslieferungsdatei die Versionsnummer mit drin.



Also benötigt man eine Verwaltung der Entwicklung der Versionen, man möchte

- feststellen, welche Version eine Quelldatei hat
- eine ältere Version wiederherstellen können
- den Unterschied zwischen Versionen ermitteln können
- einen Kommentar finde, was sich verändert hat, und warum

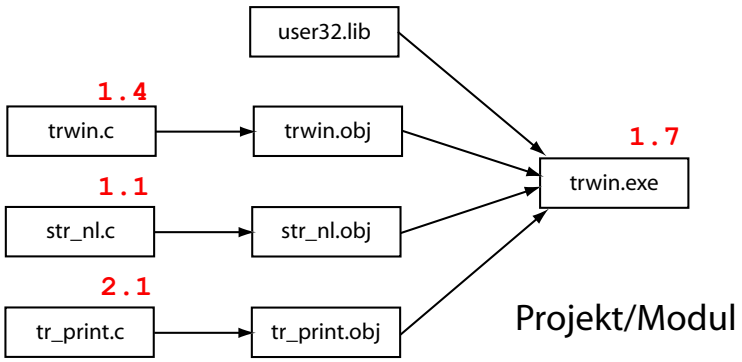
Das Werkzeug für diese Zwecke: rcs Revision Control System

Die Abbildung zu rcs zeigt die auftretenden Fragen

- Archiv für die Dateien
- Synchronisation der Zugriffe der Entwickler
- Evolution durch Folgeversionen
- Verzweigung (*branch*) bei Änderungen älterer Versionen

Doch ein Softwareprodukt oder eine Komponente besteht nicht nur aus einer Quelldatei, sondern aus verschiedenen Quelldateien, Konfigurationsdateien, Bibliotheken usw. usf.

In unserem kleinen Beispiel könnte etwa folgende Situation auftreten:



Was bedeutet das?

Man muss Gruppen von Dateien zu Projekten zusammenfassen können. Man muss wissen, welche Version welcher Datei zum Auslieferungsstand eines Programm gehört.

Ein Werkzeug, Quelldateien zu Projekten zusammenzufassen ist cvs Concurrent Versions System oder auch svn Subversion. Der wichtigste Unterschied besteht darin, dass cvs die Versionsnummer pro Datei vergibt und erhöht, während Subversion *eine* Versionsnummer für ein ganzes Archiv hat und jede Änderung, auch das Verschieben oder Umbenennen von Dateien mitverwaltet. Da in Java der Name von Dateien und der Ort im Verzeichnissystem Bedeutung für die Sprache haben (Dateinamen = Klassenname, Pfad im Verzeichnisbaum = Package), kommt es oft vor, dass die Namen von Dateien geändert werden.

Typische Befehle bei cvs sind (analog bei svn):

`cvs add file.java`	Datei file.java dem Projekt hinzufügen
`cvs ci file.java`	Neue Version archivieren („einchecken“)
`cvs co ...`	Aktuelle Version zur Bearbeitung „auschecken“
`cvs status`	Status des Projekts abfragen
`cvs update`	Dateien mit Archiv synchronisieren

Lektürehinweis zum Versionsmanagement – ein konkretes Beispiel
Claudia Fritsch: *Codemanagement: Entwicklung von Standardsoftware mit Varianten.* http://homepages.thm.de/~hg11260/mat/Codemanagement.pdf

Eine Kurzanleitung für Subversion finden Sie auf der WebSeite zur Veranstaltung: http://homepages.thm.de/~hg11260/mat/svn.pdf.

Kurzanleitung für Git: http://homepages.thm.de/~hg11260/mat/git.pdf.

Defensives Programmieren und Fehlerbehandlung

Man unterscheidet bei der Beurteilung der Qualität von Software den Fehler (*fault*) vom Ausfall (*failure*): Fehler können auftreten, auch im Code der Software sein - die Frage ist, ob sie zu Ausfällen führen.

Die Fehlerbehandlung ist ein Thema, das man nur schwierig nachträglich in den Quellcode hineinbringt – weil *alle* Quellen berührt sind. Deshalb muss man früh in der Entwicklung ein einheitliches Vorgehen für die Fehlerbehandlung vorsehen.

Dazu hilfreich ist es, sich zu überlegen, was eigentlich passieren kann:

Typen von „Fehlern“

- Unerwartetes Verhalten von Anwendern
- Verwendete Ressourcen außerhalb der Kontrolle des Systems
- Kontrollierbare Ressourcen
- Programmierfehler

Frühzeitige Überlegungen zur Strategie

- Fehlerbehandlung ist ein Aspekt „cross concern“
- Gleichartiges Vorgehen an analogen Stellen
- Betrifft das gesamte System — low level bis zur Oberfläche
- Sehr wichtig für Wartung (Tracing)

Defensives Programmieren — Pro & Contra

Defensives Programmieren besteht darin, Bedingungen vorzusehen, die im Programm zu Fehlern führen können und den Code so zu schreiben, dass diese Fehler vermieden werden.

> *Defensive programming* is a way to mitigate the effects of bugs without knowing where they are. Adopting defensive strategies is not an admission of incompetence. Although any large program is likely to have bugs in it, even a bug-free program can benefit from defensive programming.
>
> — Daniel Jackson

> Defensive programming appears in contrast to cover up for the lack of a systematic approach by blindly putting in as many checks as possible, furthering the problem of reliability rather than addressing it seriously.
>
> — Bertrand Meyer

Lektüre:

Joel Spolsky: *The Joel Test: 12 Steps to Better Code*
http://www.joelonsoftware.com/articles/fog0000000043.html

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.0 – 12. März 2012