Software Engineering - Projekte bauen, Testen und Bewerten

Eine allererste Einführung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de
Version: 1.0

Die Folien basieren in Teilen auf Folien von Dr. Helm und Prof. Dr. Hermann.

Alle Fehler sind meine eigenen.

Folien: [HTML] https://delors.github.io/se-build-test-measure/folien.de.rst.html
[PDF] https://delors.github.io/se-build-test-measure/folien.de.rst.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Buildsysteme

Buildsysteme (hier: sbt)

- Buildsysteme automatisieren repetitive Aufgaben
- Sie haben eingebaute Unterstützung oder Plugins für häufige Aufgaben:
  - Code zur einer ausführbaren Datei kompilieren: `sbt compile`
  - Tests ausführen: `sbt test`
  - Kompilierte Dateien entfernen: `sbt clean`
  - Start eines interaktiven Interpreters: `sbt console`
  - Code-Dokumentation als HTML oder PDF ausgeben: `sbt doc`
  - Code formatieren, Code-Stil prüfen: `sbt scalafmt`, `sbt scalastyle`
  - ...
  - Ausführbare Datei (auf Server) publizieren: `sbt publish`
- Buildsysteme werden mit Buildskripten konfiguriert
- Buildsysteme verarbeiten meist nur geänderte und davon abhängige Dateien (Inkrementalität)

Heutzutage bringen fast alle Programmiersprachen eigene Buildsysteme mit oder es gibt etablierte Buildsysteme.

C/C++:	**Make**, **CMake**
Java:	**Maven**, Gradle, ~~Ant~~, sbt
Scala:	**sbt**
Rust:	**Cargo**
...:	...

Minimale Anforderungen an ein Buildskript für Softwareprojekte

1. **Kompilieren** des Codes nach einem frischen Update (bei Fehler Abbruch)
2. **Testen** des Codes:
    1. **Unit-Tests** (bei Fehler Abbruch)
    2. **Integrationstests** (bei Fehler Abbruch)
    3. Systemtests/Abnahmetests (bei Fehler Abbruch)
3. **Packaging** des Projekts (bei Fehler Abbruch)
4. **Deployment** (typischerweise in einer Testumgebung)

Buildskripte für größere Softwareprojekte

Insbesondere bei größeren Projekten kommen häufig noch viele weitere Schritte hinzu:

- Code-Dokumentation erzeugen und veröffentlichen
- verschiedene statische Analysen durchführen, um Fehler zu finden
- zahlreiche Skripte um zum Beispiel Datenbanken zu aktualisieren, Docker-Container zu bauen und zu starten...
- ...

2. Testen von Software

Validierung vs. Verifikation (V&V)

Validierung: „Bauen wir das richtige Produkt?“
Verifikation: „Bauen wir das Produkt korrekt?“

> Zwei komplementäre Ansätze die (V&V) unterscheiden:

1. Software-Inspektionen oder Peer-Reviews (statische Technik)
   Software-Inspektionen können in allen Phasen des Prozesses durchgeführt werden.
2. Software-Tests (dynamische Technik)

Software-Inspektionen überprüfen die Übereinstimmung zwischen einem Programm und seiner Spezifikation.

Ausgewählte Ansätze:

Programminspektionen

Ziel ist es, Programmfehler, Verstöße gegen Standards und mangelhaften Code zu finden, und nicht, allgemeinere Designfragen zu berücksichtigen; sie werden in der Regel von einem Team durchgeführt, dessen Mitglieder den Code systematisch analysieren. Eine Inspektion wird in der Regel anhand von Checklisten durchgeführt.

Studien haben gezeigt, dass eine Inspektion von etwa 100LoC etwa einen Personentag an Aufwand erfordert.

automatisierte Quellcodeanalyse

Diese umfasst u. a. Kontrollflussanalysen, Datenverwendungs-/flussanalyse, Informationsflussanalyse und Pfadanalyse.

Statische Analysen lenken die Aufmerksamkeit auf Anomalien.

Formale Verifikation

Die formale Verifizierung kann das Nichtvorhandensein bestimmter Fehler garantieren. So kann z. B. garantiert werden, dass ein Programm keine Deadlocks, Race Conditions oder Pufferüberläufe enthält.

Zusammenfassung

Software-Inspektionen zeigen nicht, dass die Software nützlich ist.

ausgewählte Testziele

- Test der Funktionalität
- Test auf Robustheit
- Test der Effizienz/Performance
- Test auf Wartbarkeit
- Test auf Nutzbarkeit

Black Box Testen

Wir wollen die Korrektheit zeigen.

Testdaten werden durch die Untersuchung der Domäne gewonnen. Was sind gültige und was sind ungültige Eingabewerte in der Domäne?

Der Test kann (und sollte!) ohne Betrachtung der konkreten Implementierung entwickelt werden.

White Box Testen

Wie auch beim Black-Box Test wollen wir die Korrektheit zeigen.

Testdaten werden durch die Inspektion des Programms gewonnen.

Das heißt im Umkehrschluss, dass wir den Quellcode des Programms benötigen.

Der Umfang eines Tests ist die Sammlung der zu prüfenden Softwarekomponenten.

Unit Tests (Modultest):
Umfasst eine relativ kleine ausführbare Datei; z.B. ein einzelnes Objekt.

Integrationstest: Komplettes (Teil-)System. Schnittstellen zwischen den Einheiten werden getestet, um zu zeigen, dass die Einheiten gemeinsam funktionsfähig sind.

Systemtest: Eine vollständige integrierte Anwendung. Kategorisiert nach der Art der Konformität, die festgestellt werden soll: funktional, Leistung, Stress oder Belastung

Abnahmetests: Tests (durch den Kunden), um zu zeigen, dass das System die Anforderungen erfüllt.

Testpläne

Testpläne beschreiben, wie die Software getestet wird.

Beobachtung

Da die Anzahl der Tests praktisch unendlich ist, müssen wir (für praktische Zwecke) eine Annahme darüber treffen, wo Fehler wahrscheinlich zu finden sind; d. h. die Tests müssen auf einem Fehlermodell beruhen.

Es gibt zwei allgemeine Fehlermodelle und entsprechende Prüfstrategien:

1. konformitätsorientiertes Testen
2. fehlerorientiertes Testen

Entwicklung eines Testplans - Beispiel

Entwickeln Sie einen Testplan für ein Programm, das ...

1. drei ganzzahlige Werte liest,
2. diese dann als die Länge der Seiten eines Dreiecks interpretiert
3. danach ausgibt ob das Dreieck...
   - gleichschenklig,
   - schief oder
   - gleichseitig ist.

Hinweis

Ein gültiges Dreieck muss zwei Bedingungen erfüllen:

- Keine Seite darf eine Länge von Null haben
- Jede Seite muss kürzer sein als die Summe aller Seiten geteilt durch 2

Beschreibung	A	B	C	Erwartetes Ergebnis
Gültiges schiefes Dreieck	5	3	4	Schief
Gültiges gleichschenkliges Dreieck	3	3	4	Gleichschenklig
Gültiges gleichseitiges Dreieck	3	3	3	Gleichseitig
Erste Permutation von zwei gleichen Seiten	50	50	25	Gleichschenklig
(Permutationen des vorherigen Testfalls)	...	...	...	Gleichschenklig
Eine Seite ist Null	1000	1000	0	Ungültig
Erste Permutation von zwei gleichen Seiten	10	5	5	Ungültig
Zweite Permutation von zwei gleichen Seiten	5	10	5	Ungültig
Dritte Permutation von zwei gleichen Seiten	5	5	10	Ungültig
Drei Seiten größer als Null, Summe der zwei Kleinsten ist kleiner als die Größte	8	5	2	Ungültig
Drei Seiten mit maximaler Länge	MAX	MAX	MAX	Gleichseitig
Zwei Seiten mit maximaler Länge	MAX	MAX	1	Gleichschenklig
Eine Seite mit maximaler Länge	1	1	MAX	Ungültig

Die Testfälle sind noch nicht vollständig (zum Beispiel wenn A, B und C alle 0 sind; oder wenn eine Seite die Länge der beiden anderen Seiten addiert hat). Tests zum Beispiel in Hinblick auf objektorientierte Struktur, Fehlerbehandlung, etc. ... fehlen.

(Test-)Überdeckung (eng. (Test-)Coverage)

Frage

Wie wissen wir aber wie “gut” unsere Tests sind?

Definition

Testüberdeckung beschreibt, wie viel des Programms durch die Tests geprüft wird.

Hierbei gibt es verschiedene Überdeckungskriterien, die verschiedene Metriken für die Beschreibung der Testgüte nutzen.

Anweisungsüberdeckung (🇺🇸 *Statement Coverage*)

Alternativ auch Zeilenüberdeckung oder 🇺🇸 *Line Coverage* genannt.

Zu testende Software:

```java
public double compute (boolean includeTax) {
    double result = 1.3;
    if (includeTax) {
        result *= 1.19;
    }
    return result;
}
```

Testfälle:

```java
compute(false);
compute(true);
```

Frage

Wie hoch ist die Anweisungsüberdeckung? Sind beide Testfälle notwendig?

Definition

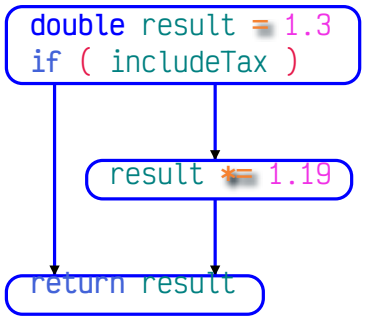
$$\text{Anweisungsüberdeckung} = \frac{\#\text{ ausgeführte Anweisungen}}{\#\text{ aller Anweisungen}} \cdot 100\%$$

Zweigüberdeckung (🇺🇸 *Branch Coverage*)

Zu testende Software:

```java
public double compute (boolean includeTax) {
    double result = 1.3;
    if (includeTax) {
        result *= 1.19;
    }
    return result;
}
```

Kontrollflussgraph:



Frage

Wie hoch ist die Anweisungsüberdeckung? Sind beide Testfälle notwendig?

Definition

$$\text{Zweigüberdeckung} = \frac{\#\text{ ausgeführte Zweige}}{\#\text{ aller Zweige}} \cdot 100\%$$

Pfadüberdeckung (🇺🇸 *Path Coverage*)

Definition

$$\text{Pfadüberdeckung} = \frac{\#\text{ ausgeführten Pfade}}{\#\text{ aller (möglichen) Pfade}} \cdot 100\%$$

Zu testende Software:

```java
public double compute (boolean includeTax,
                       boolean reducedTax,
                       double discount) {
  double result = 1.3;
  if (includeTax) {
    if (reducedTax) {
      result *= 1.07;
    else {
      result *= 1.19;
  } }
  if (discount > 0.0) {
    result *= (1.0 - discount);
  }
  return result;
}
```

Testfälle:

```java
compute(false, false, 0.0);
compute(true, false, 0.0);
compute(true, true, 0.0);
compute(false, false, 0.1);
compute(true, false, 0.1);
compute(true, true, 0.1);
```

Frage

Wie hoch ist die Pfadüberdeckung?

Achtung!

Die Pfadüberdeckung ist in der Praxis oft nicht realisierbar, da die Anzahl der möglichen Pfade exponentiell mit der Anzahl der Verzweigungen wächst.

In der Praxis wird daher oft die Zweigüberdeckung als Kompromiss genutzt.

Weitere Überdeckungskriterien

- (einfache) Bedingungsüberdeckung (🇺🇸 *(Simple) Condition Coverage*)
- Eingangs-/Ausgangsüberdeckung (🇺🇸 *Entry/Exit Coverage*)
- Schleifenüberdeckung (🇺🇸 *Loop Coverage*)
- Zustandsüberdeckung (🇺🇸 *State Coverage*)
  (Erfodert ggf. das ein endlicher Automat modelliert wird.)
- Datenflussüberdeckung (🇺🇸 *Data Flow Coverage*)

Überdeckungsziele

IEEE 29119 “Software Testing”:

100% Anweisungsabdeckung

100% Zweigabdeckung für kritische Module

DO-178B “Software Considerations in Airborne Systems and Equipment Certification”:

Abhängig von der Auswirkung von Systemfehlern. Beispiel: 100% Anweisungsabdeckung bei Verletzungsgefahr von Passagieren.

IEC 61508 “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems”:

100% Anweisungs-/Zweig-/Bedingungsabdeckung je nach Sicherheitsanforderung

ISO 26262 “Road vehicles - Functional safety”:

Abhängig von der Kritikalität der Komponente

Zusammenfassung

Das Erreichen eines Überdeckungsziels erfordert Aufwand, der durch die Kritikalität möglicher Fehler motiviert sein muss.

In vielen Anwendersystemen ist eine hundertprozentige Testabdeckung nicht notwendig. Testabdeckung ist dennoch zu messen.

Tests können nur das Vorhandensein von Fehlern zeigen, nicht deren Abwesenheit.

—E. Dijkstra

JUnit Test Case - Beispiel

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import java.util.Arrays;

public class SimpleCalculatorTest {

  @Test                                          // ⩽ JUnit Test Annotation
  public void testProcess() {

    String[] term = new String[] {
      "4", "5", "+", "7", "*"
    };
    long result = SimpleCalculator.process(term);
      assertEquals(Arrays.toString(term), 63, result); // ⩽ JUnit Assertion
  }
}
```

TestNG - Beispiel

```java
// This method will provide data to any test method
// that declares that its Data Provider is named "provider1".
@DataProvider(name = "provider1")
public Object[][] createData1() {
    return new Object[][] {
        { "Cedric", new Integer(36) },
        { "Anne", new Integer(37) }
    };
}

// This test method declares that its data should be
// supplied by the Data Provider named "provider1".
@Test(dataProvider = "provider1")
public void verifyData1(String n1, Integer n2) {
    System.out.println(n1 + " " + n2);
}
```

Behavior-Driven Development

Das Ziel ist, dass die Entwickler die Verhaltensabsichten des Systems, das sie entwickeln, definieren.[1] Hier mit Hilfe von ScalaTest.

```
1  import org.specs.runner._
2  import org.specs._
3
4  object SimpleCalculatorSpec extends Specification {
5
6      "The Simple Calculator" should {
7          "return the value 36 for the input {“6”,“6”,“*”}" in {
8              SimpleCalculator.process(Array("6","6","*")) must_== 36
9          }
10     }
11 }
```

[1] http://behaviour-driven.org/

The Last Word

A Tester's Courage

The Director of a software company proudly announced that a flight software developed by the company was installed in an airplane and the airline was offering free first flights to the members of the company. "Who are interested?" the Director asked. Nobody came forward. Finally, one person volunteered. The brave Software Tester stated, 'I will do it. I know that the airplane will not be able to take off.'

—Unknown Author @ http://www.softwaretestingfundamentals.com

Übung

Entwickeln Sie einen Testplan für das folgende Programm:

```java
import java.util.Stack;

public class RPN {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: java RPN <expr>");
            return;
        }
        // Main logic
        Stack<String> infix = new Stack<>();
        Stack<Double> ops = new Stack<>();
        for (String arg : args) {
            switch (arg) {
                case "+":
                case "*":
                case "/":
                    if (ops.size() < 2) {
                        System.out.println("Error: / requires two operands");
                        return;
                    }
                    var right = ops.pop();
                    var left = ops.pop();
                    var rightTerm = infix.pop();
                    var leftTerm = infix.pop();
                    switch (arg) {
                        case "+" -> ops.push(left + right);
                        case "*" -> ops.push(left * right);
                        case "/" -> ops.push(left / right);
                    }
                    infix.push("(" + leftTerm + " " + arg + " " + rightTerm + ")");
                    break;
                case "sqrt":
                    if (ops.size() < 1) {
                        System.out.println("Error: sqrt requires one operand");
                        return;
                    }
                    ops.push(Math.sqrt(ops.pop()));
                    infix.push("sqrt(" + infix.pop() + ")");
                    break;
                default:
                    infix.push(arg);
                    ops.push(Double.parseDouble(arg));
            }
        }
        System.out.println(infix.pop() + " = " + ops.pop());

        if (infix.size() > 0) {
            System.out.println("Unused: ");
            infix.forEach(System.out::println);
        }
    }
}
```

3. Metriken

Qualitätsmetriken

Warum?

- Neue Features oder Code-Qualität erhöhen?
- Kann ich Komponente C austauschen? Bzw. wie viel Aufwand ist das?
- Haben die letzten Änderungen das System negativ beeinflusst?
- Welche Systemteile sind besonders sicherheitskritisch?

Ziele

- Mögliche Bugs frühzeitig erkennen
- Systeme quantitativ miteinander vergleichen
- Wartbarkeit einschätzen
- Refactoring planen
- Änderungen bewerten
- Evolution des Systems überwachen bzw. verstehen

Verwendung

- Als *Quality Gates* in Buildsystemen

  z. B. Zeilen pro Methode < 50
- Zur Bewertung von Software

Einfache Metriken

Lines of Code (LOC):
: Alle Zeilen zählen so, wie sie im Quellcode stehen

Source Lines of Code (SLOC):
: Alle Zeilen ohne Leerzeile oder Kommentare

Comment Lines of Code (CLOC):
: Es zählen nur Zeilen mit Kommentaren; dies können ganze Zeilen sein, oder einfach nur Inline Kommentare; auskommentierter Code zählt ggf. auch

Non-Comment Lines of Code (NCLOC) / Effective Lines of Code (ELOC):
: Codezeilen ohne Kommentarzeile oder Zeilen, die nur Klammern enthalten, reine Import Statements oder Methodendeklarationen

Logical Lines of Code (LLOC):
: Zählt nur Anweisungen

Fortgeschrittene Metriken

Zyklomatische Komplexität (McCabe):
: Anzahl der linear unabhängigen Pfade durch den Code

Kopplung: Anzahl der Abhängigkeiten zwischen Komponenten

...: ...

Warnung

Umfangreiche Forschung hat gezeigt, dass es keine fixen Werte gibt, die für alle Projekte gelten. Es hat sich weiterhin gezeigt, dass es keine einzige Metrik gibt, die alleine zur Bewertung der Qualität eines Systems ausreicht. Welche Metriken die Qualität des Systems am besten beschreiben, lässt sich immer nur posthum beantworten.

Metriken sind immer kontextabhängig zu bewerten.

Metriken eignen sich insbesondere um Veränderungen zu bewerten und um die Entwicklung von Software zu überwachen.

4. Softwarequalitätssicherung

Konstruktiv vs. Analytisch

Mechanismen der konstruktiven Softwarequalitätssicherung

- Programmiersprachen (mit Typsystemen)
- Softwareentwicklungsprozesse
- Domain Specific Languages (DSLs)

Ansätze der analytischen Softwarequalitätssicherung

	Wiederverwend-barkeit	Wartbarkeit	Korrektheit	Aufwand
leichtgewichtige statische Analysen		✓	✓	↓-○
Semiformale Methoden			✓	↓
formale Methoden			✓	↑
Strukturanalysen	✓	✓		↓
Stilüberprüfungen		✓		↓

Leichtgewichtige statische Analysen können zum Beispiel Code-Clone erkennen (Maintenance), oder Verletzungen von empfohlenen Vorgehensweisen identifizieren und auf gängige Fehlermuster (*Bug Patterns*) hinweisen.