Zusammenfassung

Software Engineering 2

Michael Wieland and Fabian Hauser

Hochschule für Technik Rapperswil

26. August 2017

Mitmachen
Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter `https://github.com/michiwieland/hsr-zusammenfassungen` forken.

Inhaltsverzeichnis

1. Projektplanung

1.1. Meilensteine und Zwischenresultate

1.1.1. Messbare Zwischenresultate

- Anzahl Seiten Requirements
- Anzahl Items (Teil-Abschnitte, aufgezählte Punkte) in den nicht-funktionalen Anforderungen
- Anzahl Klassen im Domainmodell
- Anzahl User Stories (im Backlog, completed...)
- Development speed (Anzahl Story Points per Sprint; bei Scrum)
- Anzahl & Schwere der offenen Bugs, durchschnittl. Verweildauer, Abarbeitungsgeschwindigkeit
- Anzahl screens (GUI-Entwürfe)
- Alle Code-Metriken (lines of code, Anzahl Klassen, Methoden, Kopplung, etc...)
- Anzahl (nicht)erfüllte Testfälle
- Anzahl Schnittstellen zu Umsystemen, inkl. Komplexitäts-Einschätzung S, M, L, XL

1.2. Dokumentationen in Software Projekten

1.2.1. Kundensichtbarkeit von Dokumentationen

Zwischenresultat	*sichtbar für Kunde*	*für technisch Versierte*	*für Entwickler*
Prototypen, bzw. Releases am Ende jeder Iteration	x	x	x
Quellcode (Sources)	-	-	x
Requirements: UCs, nicht-funktionale Req.	x	x	x
Requirements: Aktivitätsdiagramme, Zustandsdiagramme	-	x	x
Beschreibung der IT Landschaft (Ist-Zustand)	x	x	x
Schnittstellen zu Umsystemen	(x)	x	x
Analyse: Domainmodell	(x)	x	x
GUI-Entwürfe, Wireframes	x	x	x
Software Architecture Document (SAD)	-	x	x
QS-Massnahmen/Review-Plan, inkl, Testplan, Testfälle und Testprotokolle	(x)	x	x
Backlog: User Stories, Arbeitspakete mit Akzeptanz-Kriterien	x	x	x
Infrastruktur-Planung (hauptsächlich: Diagramm der Server)	(x)	x	x
Installationsanleitung, Installations-Skripte/Tools, Deployment Dokumentation	-	(x)	x
Benutzeranleitung	x	x	x

Abbildung 1: Kundensichtbarkeit von Dokumentationen

2. Automation

2.1. Build Tools

Pros:

- Automated, non-interactive process
- Repeatable!
- Independent of your IDE
- Time-consuming tasks can be scheduled

Cons:

- Automated, non-interactive process
- Maintenance and extension is difficult
- Most likely platform-dependent (.sh, .bat, ...)

Benefits:

- Reduce repetitive tasks
- Independence of any IDE
- Reproducible results
- Save time
- Basis for continuous integration!

Features / Wishlist:

- Single command builds (CRISP)
  - Complete: Das Programm lässt sich bei jeder Ausführung komplett builden.
  - Repeatable: Wiederholbar auch für auf älteren Versionen/Tags
  - Informative: Feedback, sprechender Output bei jedem Step
  - Schedulable: Planbar
  - Portable: Plattformunabhängig ausführbar
- Flexibility (run individual tasks)
- Performance Extensibility

2.1.1. Make

Imperatives Build Script, welches als DAG (Azyklischer Graph) umgesetzt wurde. Make ist plattformabhängig und hat kein Dependency Management.

2.1.2. Ant

Ant ist ein imperatives Build Tool mit folgenden Konzepten:

- XML-based "scripting" with built-in tasks (mkdir, jar, condition,...)
- Focus on portability
- Custom tasks can be written in Java

2.1.3. Maven

Maven is a deklarativ build tool, designed for consistency and automated dependency management.

Concepts:

- Declarative (XML-based)
- "Convention over configuration" => Default build => Only specify differences
- Predefined lifecycles = DAG
- Default = build and deployment
- Clean
- Site = generate documentation
- Predefined directory tree
- Automated dependency resolution

Pros:

- More concise => shorter build-files
- Reusable build-logic (plug-ins)
- Automated dependency management (!)

Cons:

- Less general and flexible than "imperative" tools
- Impose a somewhat rigid project / file structure

```
app
|- src/main/java      //production code
|- src/main/resources
|- src/test/java      //tests
|- src/test/resources
|- pom.xml
```

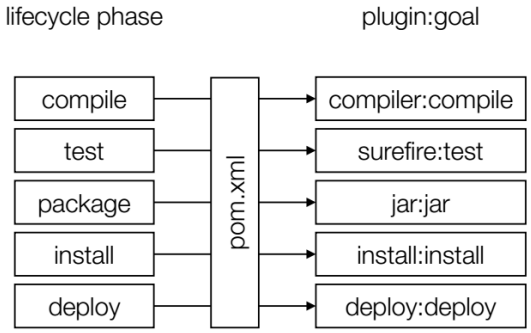

Abbildung 2: Maven Filestructure und Lifecyle

2.2. Continuous Integration

Continuous Integration darf nicht mit Continuous Delivery verwechselt werden! Continuous Integration beschreibt den Prozess des fortlaufenden Erstellens von Software aus mehreren Komponenten. Es stellt die Basis für Continuous Delivery dar, wobei bei Continuous Delivery die getestete Software in sehr kurzen Zyklen auf die produktiven Server deployed wird.

2.2.1. Best Practices

1. Maintain a single source repository
2. Automate the build
3. Make the build self-testing
4. Everyone commits to the mainline every day
5. Every commit to the mainline should be built
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy to get the latest deliverables
9. Everyone can see what's happening
10. Automate deploymentMave

3. SE Practices

3.1. Requirements Practices

- Anforderungen aus Benutzersicht
- Kritisches Hinterfragen: Nach Grund und Ursprung für einen Wunsch Fragen.
- Genügend generell und Abstrakt definieren. Abstraktionsgrad sollte so hoch sein, dass der Programmierer eine konfigurierbare Lösung implementiert.
- Verfolgung des Ursprungs: Person, Grund, Datum.

3.1.1. Vorgehen

1. Dig for Requirements: Ermittlungstechniken:
   - Dokumentenstudium
   - Befragung: Interview, Fragebogen, Workshop, Prototyp Review
   - Beobachtung: Feldbeobachtung, Apprenticing, Bestehender Business Prozess analysieren
   - Kreativität: Brainstorming
2. Make Quality a Requirement
   - Qualität als (NF-)Requirements aufnehmen
     - Performance, Scalability, Security, robustness
   - Möglichst testbare Qualitätsanforderungen
     - Max. Antwortzeiten unter definierten Umständen
     - Min. Unterstützte Datenmenge
     - Prüfung von Daten/Geräte und Verhalten bei Fehler
   - Basierend auf echten Anforderungen
     - Konkrete Benutzer-Erwartungen / harte externe Limiten
     - Quantitative Werte unter definierten Rahmenbedingungen.
   - Qualitätsanforderungen sollten quantitativ, prägnant beschrieben werden, um Fehlinterprätationen zu verhindern.
   - Schwieriger zu ermitteln sind unbewusste Wünsche. Diese müssen mit prägnanten Fragen gefunden werden.
   - Von der geforderten Qualität hängen Architekturentscheide ab. Muss die Architektur während dem Projektverlauf angepasst werden, geht das ins Geld. Deshalb früh Qualitätsanforderungen prüfen.
3. Deal with Changes: Mythos "Stabile Requirements"
   - Ca. 2% ändern sich pro Monat!
   - Requirement-Änderungen Antizipieren: Details nachfragen & abstrakt definieren
   - Design for Change
   - Kurze Iteration mit User-Feedback
   - Change Assessment: Qualität nach jeder Iteration prüfen.

3.2. Design Practices

3.2.1. Don't repeat yourself

Repetition von Information vermeiden, damit keine Inkonsistenzen bei Änderungen entstehen:

- Code Duplikationen: Logiken, Daten, Boilder-Plate Code etc.
- Dokumentation in Code: Redundante Beschreibungen
- Dokumentation separat zum Code: Wiederholung
- Wiederholung wegen Programmiersprache

DRY Techniken:

- Konstanten statt literale Konstantenwerte
- Vermeiden von Code-Snippets
- Code-Kommentare geben nur relevante Zusatzinformationen
- Externe Konfigurationsdaten
- Code-Generierung: Evtl. mit DSL (auch selbstgebaute)

3.2.2. Archieve orthogonality

Ziel: Hohe Kohäsion / Tiefe Kopplung. Man spricht von Orthogonalität, wenn zwei Dinge, die nichts miteinander zu tun haben, voneinander unabhängig sind. Vorteile Orthogonalität:

- Selektive Wiederverwendung eines Aspekts
- Ein Aspekt isoliert änderbar
- Einfacher zu verstehen
- Einfacher zu testen

Zur Reduktion der Kopplung:

- Hierarchische Zerlegung der Komponenten (System, Packages). Allenfalls Information Hiding, um die Verständlichkeit des Architekturdiagramms zu gewährleisten.
- Reduzierte Abhängigkeiten zwischen Komponenten (DAG/Keine Zyklen, Abtrennung Daten und Routinen, Kein Spaghetti Code)

3.2.3. Design to test

Testbarkeit vor Entwicklungszeit betrachten, hat Einfluss auf die Architektur. Schlüsselwörter:

- Klare Interfaces und Kontrakte
- Überlegung der relevanten Testfälle
- Grad der Unit-Testbarkeit und Integration-Testbarkeit
- Evtl. Freiheitsgrade für «Dependency Injection»

  - Erzeugung und Initialisierung von benötigten Objekte ausserhalb des Benutzer Objektes (Wie Strategie Muster oder Fabrik Muster)
  - Isoliertes Testen mit anderen Komponenten («Fakes»)
  - Kann Design komplexer als (produktiv) nötig machen

Schlüsselwörter:

Fake Vereinfachte Implementierung

Mock Auf Testfall zugeschnitten; prüft Reihenfolge und Inhalt der erwarteten Aufrufe

Stub Auf Testfall zugeschnittene Antworten

Dummy Objekte, die nur herumgereicht, aber nie inspiziert werden.

3.3. Implementation Practices

3.3.1. Fix broken windows

Probleme beheben, wenn sie entstehen (kleine sofort beheben, grössere markieren und später beheben). Probleme können gut durch Refactoring erkannt werden. Refactoring beschreibt den Prozess, des Code verbesserns, ohne die Funktionalität zu verändern.

3.3.2. Refactor early and often

- Konstanter Verbesserungsprozess in einem wachsenden Software-Projekt
- Liste über die zu verbesserenden Bereichen führen
- Vorgehen:
  - Refactoring soll keine neuen Funktionen bieten
  - Gute Tests vor Beginn des Refactoring haben
  - Mehrere kleinere Schritte statt einer Riesenänderung.

3.3.3. Program deliberately

Gezieltes programmieren:

- Klares Ziel und Design
- Logisch rigoros analysieren, entwickeln und testen
- Nur auf spezifizierte Features von Libraries verlassen
- Eingesetzte Technologie beherrschen
- Annahmen dokumentieren und mit Assert & Tests prüfen
- Crash early: Alle ungültige Zustände sollen Fehler erzeuge
- Exceptions richtig behandeln, nicht blind unterdrücken
- Debugging: Auf Grund gehen, Fehler verstehen

3.4. Verification Practices

Verifikation besteht aus zwei Komponenten:

- Statische Analyse
  - Design Reviews
  - Code Reviews
- Testing (Dynamisch)
  - Unit Tests
  - Integration Tests

3.4.1. Test rigorously

- Früh, häufig und automatisch testen.
- Für gefundene Fehler sollte immer ein Test geschrieben werden, um diesen in Zukunft zu vermeiden.
- Integrationstests sind mindestens so wichtig wie Unit-Tests!
- TDD und die dadurch resultierende hohe Test Coverage ist anzustreben.
- Nur weil Tests existieren, sollte rigoroses Überlegen beim Programmieren nicht vernachlässigt werden. Auch 100% Korrektheit bedeutet noch keine Korrektheit.

3.4.2. Perform reviews

- Formale Inspektionen durchführen: Durch Experten, sowie durch Entwicklern Design & Code Reviews. Immer festhalten der Ergebnisse (severity, Action, Verantwortlicher)
- Das Ziel der Reviews nicht aus den Augen verlieren: Defekte zu finden. (keine Stil oder Architekturdiskussionen)

4. Test Driven Development

The TDD way to write code is to write tests that don't pass, then make them pass.

1. Red: Schreibe zuerst einen Test, der fehlschlägt, weil das geforderte Feature noch nicht umgesetzt ist. (er sollte aber kompilieren)
2. Green: Ändere den Code bis der Test durchläuft
3. Refactor: Refactore den Code sowie den Microtest bis diese ein kosistentes, gutes Design aufweisen.
4. Integrate (optional): Der geschriebene Code und Tests in die Codebase einfügen. Dieser Schritt kann auch von einem CI-System übernommen werden.

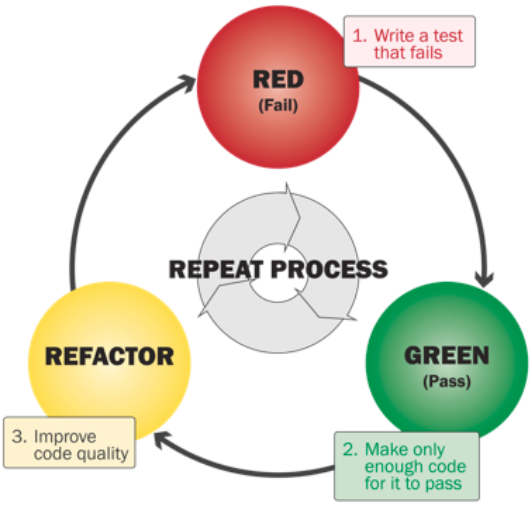

Abbildung 3: TDD Lifecycle

4.1. Produktivität

TDD verbessert wegen den folgenden Punkten die Produktivität:

- Die Fehlersuche wird stark vereinfacht.
- TDD gibt eine relativ hohe Garantie, dass der Code von anderen Entwickler auch noch nach der eigenen Änderung funktioniert. (obschon vorhandene Tests keine Korrektheit garantieren)
- Man verwendet weniger Zeit mit dem Debugger, um unerwartetes Verhalten zu analysieren
- TDD wird Schrittweise entwickelt und so kommt es seltener vor, dass grosse Codeteile auf einmal verstanden werden müssen.
- Unnützer Code wird erst gar nie geschrieben

4.2. Vorgehen

Specify it

1. Essence First: Entwickle das wichtigste zuerst. Achte auf die essenzielle Funktionalität.
2. Test First: Starte mit einem guten Namen für den Test
3. Assert First: Schreibe zuerst den `ASSERT` mit den erwarteten Werten.

Frame it

4. Frame First: Erstelle ein Skelett, damit der `ASSERT` kompiliert. Erstelle nur gerade so viel Code, wie dazu benötigt. Es bietet sich an, dass die benötigten Klassen und Methoden durch die IDE generiert und anschliessend die Parameternamen entsprechend angepasst werden.

Evolve it

5. Do the simplest thing that could possibly work: Möglichst einfach die Testkriterien erfüllen
6. Break it to make it work: Schreibe ein `ASSERT`-Ausdruck der mit dem aktuellen Code fehlschlägt.
7. Refactor mercilessly: Verbessere die Qualität der Test- und Produktionscode fortlaufend und rücksichtslos
8. Test driving: Wiederhole die vorgehenden Schritte für kleine Teilfeatures.

```
1  // class under test
2  public class XmlBuilder { // #4
3     private final String rootTagName;
4
5     public XmlBuilder(String rootTagName) {
6        this.rootTagName = rootTagName;
7     }
8
9     public String toXml() {
10       return "<" + rootTagName + "></" + rootTagName + "a>"; // #5 first version
11       return openTag() + closeTag(); // #7 second version incl. extracted methods
12    }
13
14    private String openTag() {
15       return "<" + rootTagName + ">";
16    }
17
18    private String closeTag() {
19       return "</" + rootTagName + ">";
20    }
21 }
22
23 // test
24 public class XMLBuilderTest { // # 1
25    @Test
26    public void testOpenClosingTag() {
27       XmlBuilder xmlBuilder = new XmlBuilder("a"); // # 3
28       assertEquals("<a></a>", xmlBuilder.toXml()); // # 2
29       assertEquals("<h1></h1>", xmlBuilder.toXml()); // # 6
30    }
31 }
```

4.3. Detail Lifecycle

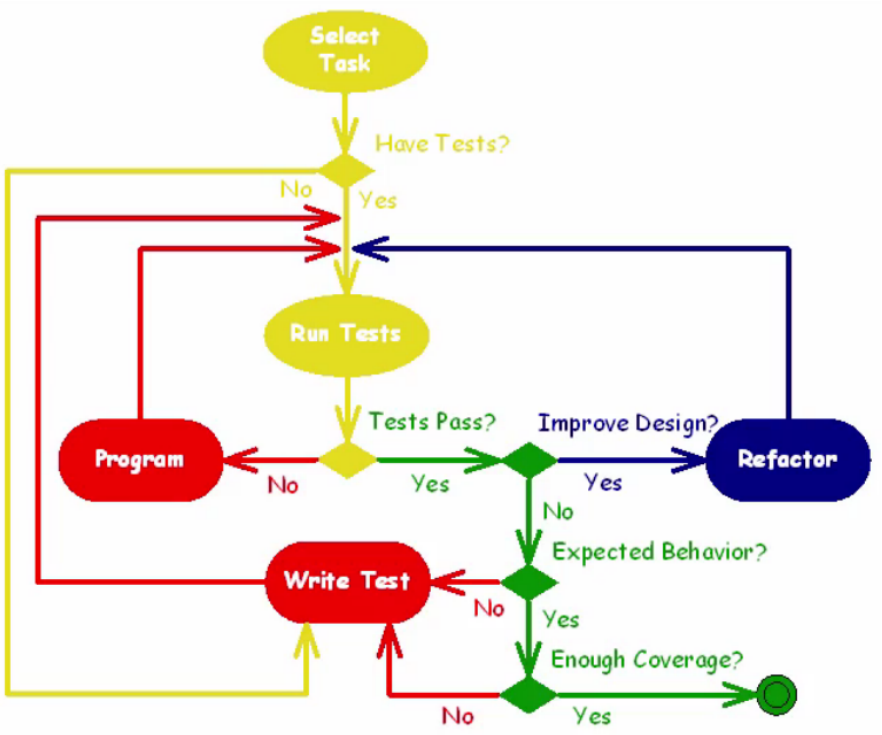

Abbildung 4: TDD Lifecycle in Depth

5. Code Smells

Ein Code Smell ist ein Hinweis auf ein schwerwiegenderes Problem (Design Flaws), dass besser behoben werden sollte. (If it stinks, change it) Wie Weinkenner, haben auch Programmierer ein eigenes Vocabular, um Smells genau zu beschreiben.

5.1. Technical Dept

Technical Dept ist eine Metapher für die möglichen Konsequenzen schlechter technischer Umsetzung von Software. Man versteht darunter den zusätzlichen Aufwand, den man für Änderungen und Erweiterungen an schlecht geschriebener Software einplanen muss.

- Nicht Aktualisieren von Softwaredokumentation
- Fehlende Versionierung, Backup, Build-Tools, CI
- Aufschieben und Verzicht von automatisierten Tests
- Missachtung von TODO und FIXME
- Codewiederholungen und andere Code Smells
- Missachtung von statischer Code Analyse

5.2. Ursachen

- Fachlicher Drucker
- Ungenügende qualitätssichernde Prozesse
- Ungenügendes Wissen / Kommunikation

5.3. Lost Intent

Beim Lost Intent ist die Absicht nicht klar ersichtlich.

```
1  // constructors
2  public Loan(commitment, riskRating, maturity);
3  public Loan(commitment, riskRating, maturity, expiry);
4  public Loan(commitment, outstanding, riskRating, maturity);
5  public Loan(capitalStrategy, commitment, riskRating, maturity, expiry);
6  public Loan(capitalStrategy, commitment, oustanding, riskRating, maturity, expiry);
7
8  // refactorings
9  private Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry);
10
11 public createTermLoan(commitment, riskRating, maturity, expiry);
12 public createTermLoan(commitment, outstanding, riskRating, maturity);
13 public createResolver(capitalStrategy, commitment, riskRating, maturity, expiry);
14 public createRCTL(capitalStrategy, commitment, oustanding, riskRating, maturity,
       expiry);
```

5.4. Inefficient Name

Unverständliche Namen erschweren die Arbeit des Programmierers, da es schwierig ist die Funktionlität bei schlecht gewählten Namen zu eraten. Möchte man die Performance eines Programmcodes verbessern, startet man oft an der Stelle die am meisten aufgerufen wird. Genau gleich sollte es mit der Performance des Programmierers sein. Programmierer verwenden einen grossen Teil ihrerer Zeit für das Lesen und Verstehen von Code. Damit wird die Performance des Programmierer verbessern können, müssen wir den Code lesbar schreiben. Gute Namen beantworten die Frage nach dem Warum sollte jemand diese Methode, Variable, Klasse verwenden.

```
1 squareRootOfSumOfDimensionalDifferences -> distance
2 setBit7OnRegister3() -> turnOnLoopbackMode()
```

Gute Namen sind simple, accurate, consistent.

5.5. Duplicate Code

Man unterscheidet zwischen zwei Varianten von Code Duplizierung:

Blatant Der genau gleiche Code kommt an mehreren Stellen vor

Subtle Scheinbar unterschiedlicher Code führt die selbe Funktionalität aus (gleiches Boilerplate, einzelne unterschiedliche Methoden werden aufgerufen)

6. Error Handling

6.1. Vermeidung

Es gibt mehrere Möglichkeiten, Fehler zu vermeiden:

- Testing / Simulation schreiben
- Statische Analyse
- Architektur (Review)
- Deliberate Programing
- Reviews

Werkzeuge zur Fehlerbehandlung sind:

- Exceptions (machen alles einfacher)
- Assertions
- Logging

6.2. Defensive Programmierung

Defensive Programmierung beinhaltet eine systematische Fehlerprüfung von allen Inputs, sowie eine **systematische Fehlerbehandlung**. Defensive Programming macht nur in bestimmten Anwedungsbereichen Sinn, da es mit einem grösseren Aufwand verbunden ist.

Dies umfasst:

- Behandeln und Filtern von Benutzereingaben. Anzeigen von Fehlermeldungen und Aktion unterbinden.
- Prüfen von allen Daten aus externen Quellen
- Prüfen von Routinen-Preconditions
- Ungültige Fälle systematisch abfangen (z.B. Exception werfen als default in switch-case → Nicht unterstützte Zustände systematisch abfangen. Somit werden Fehler durch spätere Erweiterungen besser erkannt)

6.2.1. Fehler-Barrikaden

Eine Fehler-Barrikade ist ein geschützter Rahmen mit sichergestellt validen Daten. So könnte man z.B eine Facade erstellen, die einen String auf seine Korrektheit überprüft und den String dann in ein URL Objekt packt, wobei die URL immutable sein sollte. Damit bleibt der String innerhalb der Barrikade konsistent.

- Barrikade im Programm gegen Fehler definieren
  - Hinter den Barrikaden sind Daten gültig
  - Daten bei Grenzübertritt überprüfen
- Barrikade auf Klassenniveau

  - Public Methoden überprüfen Daten
    * Per Exception (externer Input)
  - Private Methoden gehen von gültigen Daten aus
    * Per Asserts (interner Input)

6.3. Fehlerhandling

Konservative / Pessimistische Behandlung:

- Error handling prozedur
- Fehlermeldung anzeigen
- Shutdown

Optimistische Behandlung:

- Neutrales Resultat
- Nächstes plausibles Resultat
- Warnung in Stream loggen
- Optimistische Fehlerbehandlung kann in einem kritischen System gefährlich sein, wenn die getroffenen Annahmen/default Werte falsch gewählt wurden. Bei optimitischer Fehlerbehandlung sollte deshalb immer ein Log erstellt werden.

6.3.1. Korrektheit und Robustheit

Korrektheit Niemals ein ungenaues Resultat liefern (Oft bei Sicherheistkritischen Systemen: z.B Bank Trading)

Robustheit Die Software wenn möglich am Laufen zu halten. (Meist bei unkritischen Systemen: z.B Betriebsystem)

6.3.2. Lokale vs. globale Behandlung

- Lokale fälle sind erwartete Fälle, die nicht höher relevant sind und lokal abschliessend entscheidbar sind. Checked Exceptions sollten lokal behandelt werden.
- Ansonsten sollte die Exception weitergegeben werden.
- Wichtig: Immer gültige Zwischenzustände mit `finally` hinterlassen
- Eine Top-Level Behandlung soll alle Fälle abfangen, behandeln, protokollieren und dem Benutzer eine Meldung abgeben. Falls möglich kann das Programm darauf in einen konsistenten Zustand versetzt oder kontrolliert terminiert werden.
- Error Handling sollte auch getestet werden.

6.3.3. Exceptions vs. Assertions

- Exceptions für produktive Fälle und sicherheitsrelevante Fehler.
- Assertions sind mehr für Debugging, Fehler welche nie auftreten sollten sowie Post/Preconditions. Sie sollten eher zwecks Dokumentation genutzt werden.
- Achtung: Assertions können Nebeneffekte haben, und können ein Programm in einem inkonsistenten Zustand belassen. z.B wenn der `assert` etwas ausführt, bevor ein boolscher Wert zurückgegeben wird.

6.4. Logging

Logging dient der Postmortum-Analyse. Es empfiehlt sich ein Plan zu erstellen, wann und auf welchem Level geloggt wird.

- Logging existiert zu diagnostischen Zwecken. Üblicherweise gibt es verschiedene Log-Levels.
- Für das Logging empfiehlt es sich, ein passendes Framework einzusetzen (z.B. log4j oder log4net)

6.5. Policies

Error Handling Policy
Welche Eingaben und Interaktionen sind erlaubt und wie soll sich das System bei Fehleingaben verhalten?

Exceptions Policy
Werden Exceptions benutzt und wie soll dies gemacht werden?

Assertion Policy
Werden und für was werden Assertions benutzt und wann werden sie eingeschaltet?

Logging Policy
Was und wie detailiert wird in das Log geschrieben? Welche Log Levels werden verwendet?

7. Design by Contract

- Contracts (Verträge) legen Benefits (Rechte) und Obligations (Pflichten) zweier Parteien, meist Client und Supplier, fest.
- Design by Contract legt die Beziehung zwischen Komponenten fest.
- Erfunden wurde es mit dem Hoare Trippel: $\{P\}\, c\, \{Q\}$ (Precondition P, Programm c, Postcondition Q).
- Bertrand Meyer entwickelte dieses Konzept zur Designmethode und Sprache Eiffel weiter.

7.1. Preconditions, Postconditions, Class Invariants

Für jede Methode gibt es:

7.1.1. Preconditions

Dies sind Bedingungen, welche vor dem Aufruf erfüllt sein müssen. Der Aufrufer ist für dessen Einhaltung verantwortlich.

7.1.2. Postcondition

Dies sind Bedingungen, welche nach dem Aufruf erfüllt sein müssen. Dafür ist die Implementation der Methode zuständig.

Bei Query-Only Methoden , welche den internen Zustand nicht ändern, sind keine Postconditions auf interne Zustände möglich. Es gibt höchstens Postconditions für Rückgabewerte.

7.1.3. Class Invariants

Die Klasseninvarianten stellt für eine Klasse gewisse Garantien sicher, welche vor und nach dem Methodenaufruf gelten. Dafür ist die Implementation der Klasse verantwortlich.

7.1.4. Eiffel: Beispiele

```
put (x: ELEMENT; key: STRING) is
-- Insert x so that it will be retrievable through key.
require
   count < capacity
   not key.empty
do
   -- ...Some insertion algorithm ...
ensure
   has (x)
   item (key) = x
   count = old count + 1
end
```

```
class STACK
-- ...
invariant
   count_non_negative: 0 <= count
   count_bounded: count <= capacity
```

```
6     consistent_with_array_size: capacity = representation lcapacity
7     empty_if_no_elements: empty = (count = 0)
8     item_at_top: (count > 0) implies
9        (representationl item (count) = item)
10 end
```

7.1.5. Einschränkungen

Interne Konsistenzen wird nicht überprüft, es werden nur die Interfaces überprüft.

7.2. Gute Verträge, Anwendung in agilem Umfeld

Ein Vorteil guter Verträge ist ein besseres Verständnis, wer wofür verantwortlich ist. Preconditions: Client, Postconditions: Supplier

7.3. Exceptions

Exceptions werden unterschieden in normale und exceptional Exceptions. Bei Exceptional können andere Postconditions greifen als bei Normalen.

7.4. Vererbung

Erbende Klassen müssen immer *mindestens* die Conditions der Superklasse oder des Interfaces erfüllen. D.h. Preconditions dürfen gleich bleiben oder gelockert werden, Postconditions dürfen verschärft, aber nicht gelockert werden. Die Invariante darf gleich bleiben oder verschärft werden. (Liskov Substitution Prinzip oder Behavioural Subtyping → Subtyp muss mindestens den Vertrag der Basisklasse erfüllen.)

7.5. TDD

Contracts sind eine optimale Ergänzung zu Unit Tests bei der TDD-Entwicklung.
TDD Prinzipien für einen Clean Contract:

1. Trenne Abfragen durch @Pure von Kommandos
2. Trenne elementare Abfragen von abgeleiteten Abfragen
3. Erstelle für jede abgeleitete Abfrage eine Nachbedingung
4. Erstelle für jedes Kommando eine Nachbedingung
5. Prüfe für jede Abfrage und jedes Kommando, ob eine Vorbedingung erforderich ist
6. Formuliere Invariante, um unveränderliche Eigenschaften des Objekts beschreiben.

DbC Konstrukte sind deklarativ, Unit Tests sind imperativ. DbC kann als Ergänzung zu Unit Tests angesehen werden. Unit Tests testen in der Regel auch Postconditions.

7.6. Unterstützung von DbC in Programmiersprachen

Es ist grundsätzlich empfehlenswert, in einer Sprache, welche keine Native Unterstützung für Contracts hat, ein passendes Framework zu verwenden.

7.7. Java

Z.B. mit icontract:

```
/**
* @inv !isEmpty() implies top() != null
*/
public interface Stack {
   /**
   * @pre o != null
   * @post !isEmpty()
   * @post top() == o
   */
   void push(Object o);

   /**
   * @pre !isEmpty()
   * @post @return == top()@pre
   */
   Object pop();

   /**
   * @pre !isEmpty()
   */
   Object top();

   boolean isEmpty();
}
```

Achtung! Gilt nicht allgemein, ist Annahme hier!

Abbildung 5: Contract in Java mit icontract

Frameworks : Contract4J, JML, Oval, Cofoja

7.8. Concurrency

Alle bisherigen Aussagen gelten nur wenn entweder die Programme sequentiell ausgeführt werden oder es sich um fully synchronized objects handelt.

8. Code Smells

Die sechs häufigsten Code Smells sind:

- Duplicated Code
- Long Method
- Large Class
- Conditional Complexity
- Comment Smell
- Switch/Case (insbesonders wenn instanceOf() im Spiel ist)

8.1. Prüfungsrelevanz

In der folgenden Liste sollten Sie an einer Prüfung:

- Einen Smell erkennen, wenn er als Code daherkommt
- Wenn der Smell-Name gegeben ist, beschreiben was das das Problem mit dem Smell ist
- Wenn der Name vorgegeben ist, ein gutes Beispiel hinschreiben können (Code-Fragment skizzieren)

Liste der Smells, die Sie kennen sollten:

- Speculative Generality
- Comment
- Long Method
- Long ParameterList
- Magic Numbers
- Duplicated Code
- Large Class
- Conditional Complexity
- Switch/Case (ibs. wenn instanceOf() im Spiel ist)
- Primitive Obsession
- Oddball Solution
- Refused Bequest
- Inapporpriate Intimacy
- Feature Envy
- Lazy Class
- Middle Man
- Shotgun Surgery/Solution Sprawl

9. Refactoring

Unter Refactoring versteht man eine Änderung der interenen Struktur von Software, sodass diese einfacher zu verstehen und günstiger abzuändern ist, jedoch ohne dass das Verhalten der Software verändert wird. Typischerweise wird bem Refactoring folgende Dinge unternommen:

- Removing duplicated or dead code
- Simplifying complex code
- Clarifying unclear code

9.1. Pattern

Die sechs häufigsten refactoring patterns sind:

- Extract Method
- Rename
- Move Method (inklusive pull up/push down)
- Change Method Signature
- Replace Inheritance with Delegation
- Replace Magic Number with Symbolic Constant

9.2. Refactor Zyklus

Es lohnt sich ein Refactoring in vielen kleinen Schritten (Baby Steps) durchzuführen.

1. Verify that all automated tests (microtests) pass
2. Decide what code to change
3. Implement one or more refactorings carefully
4. Run the microtests whenever you wish to confirm that changes have not altered system behavior
5. Repeat until the refactoring is complete or revert to an earlier state

9.3. Prüfungsrelevanz

Die folgende Liste definiert, welche Refactorings Sie gut kennen sollten, insbesonders:

1. erkennen und benennen können, wenn die Refactorings als Vorher/Nachher-Bilder (UML) oder Code daherkommen
2. wenn Refactoring-Name erwähnt, beschreiben, was passiert
3. wissen, welchen Smell das Refactoring behebt, und wann Sie es anwenden
4. Antworten können: gegeben Smell X, welches Refactoring ist hier zielführend? (mehrere möglich, nur 1 nötig für korrekte Antwort)

Liste der Refactorings:

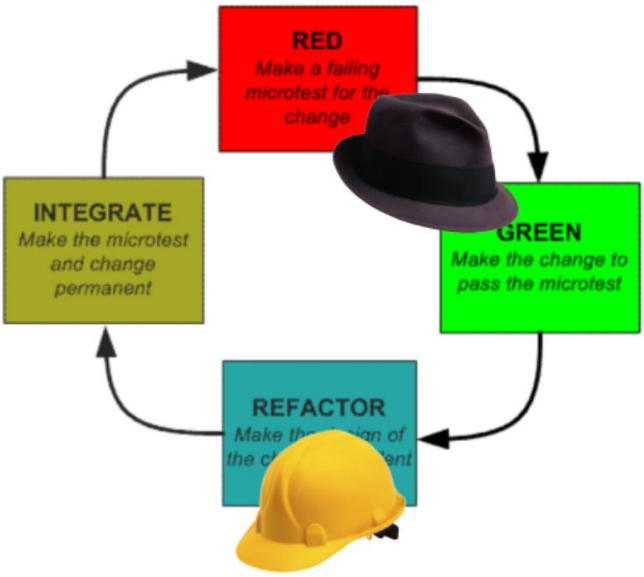

Abbildung 6: Refactoring Cycle

- Extract Method, Inline Method
- Extract Variable, Inline Temp
- Replace Temp with Query
- Rename Method/field/parameter
- Move Method (inkl. pull up/push down)
- Move Field (inkl. pull up/push down)
- Change Value to Reference, Change Reference to Value
- Replace Magic Number with Symbolic Constant
- Change Method Signature
- Encapsulate Field
- Replace Type Code with Class/Subclass
- Replace Type Code with State/-Strategy
- Decompose Conditional
- Replace Nested Conditional with Guard Clauses
- Introduce Null Object
- Replace Constructor with Factory Method
- Replace Exception with Test
- Extract Subclass/Superclass
- Extract Interface
- Collapse Hierarchy
- Replace Inheritance with Delegation
- Oddball Solution: Substitute Algorithm [Fowler], Unify Interfaces with Adapter (GOF)

9.3.1. Nutzen

- Pull Up Method → Duplication Free
- Pull Down Method → Highly Cohesive

- Rename Method → Clarity
- Replace Inheritance With Delegation → Highly Cohesive
- Split Temporary Variable → Clarity

9.4. Eclipse Integration

Alle grösseren IDE bieten heutzutage automatische Refactoring Funktionalität an.

- Move..
- Change Method Signature
- Extract Interface
- Extract Superclass
- Pull Up
- Push Down
- User Supertype where possible
- Introduce Parameter Object
- User Supertype where possible
- Infer Generic Type Arguments

10. Aufteilung Projekte / Story splitting

Grosse Projekte schlagen oft fehl, darum möglichst nie Projekte mit > 9 Monate, > 1 Million Budget oder Team > 10 Personen! Falls Projekte grösser werden, müssen sie aufgesplitted werden.

10.1. Aufteilung im Grossen

10.1.1. Aufteilungs- und Priorisierungskriterien

- Nach Kundendomänen
- Nach Geschäftsprozessen (ungefähr nach User Cases)
- Nach Benutzerrollen (z.B. Alle Endbenutzerprozesse, Administratorfunktionen)
- In Kern Funktionen, Wichtige Funktionen, 08/15 Funktionen, CRUD
- Nach Bereiche im Domain-Model
- Geografisch (z.B. zuerst Kanton GR, dann erst ZH)

10.2. Aufteilung im Kleinen

Zuerst minimale Grundfunktionen bauen, später Features inkrementell nachliefern. Man spricht von dem Minimum Viable Product (MVP) → A product with just enough features to gather validated learning about the product. Der Kostensprung von der Basis Version zum Voll-Ausbau darf nicht vernachlässigt werden. (1:10)

10.3. Arbeitspakete / User Stories

User Stories sind umsetzungsorientiert und auf den Arbeitsfluss/Sprintgrösse angepasst. (Im Gegensatz zu den Use Cases, diese sind kundenorientiert).

Durchschnitt: was 1 Person in 1/4 eines Sprints schafft

Maximum: was 1 Person in 50-70% eines Sprints schafft

10.4. Story Mapping

Zusammenfassen mehrerer User Stories zu einer grösseren Einheit (meist Epics), für mehr Übersicht.

Winziges Projekt *Ohne Hierarchie*	*Kleines Projekt* *2 Hierarchiestufen*	*Mittleres Projekt* *3 Hierarchiestufen*
50 User Stories	8 Epics (≈Use Cases) 120 User Stories oder: 8 User Stories (≈Use Cases) 120 Tasks	12 Epics (≈Use Cases) 200 User Stories 1000 Tasks

Abbildung 7: Projektübersicht / Aufteilung Projekte

11. Testing: Unit Tests and More

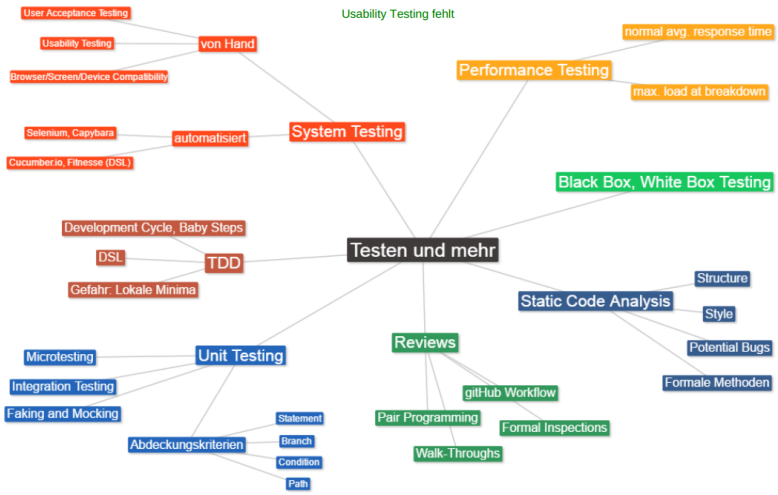

Abbildung 8: Übersicht von Tests

11.1. Unit Tests / Microtests

Isoliertes Testen einer Klasse bzw. meist einzelner Methoden. Sollte schnell gehen (wichtig für CI). Wird oft in Kombination mit Fakes & Mocks eingesetzt.

Nachteile von Unittests sind oft die fehlende Aussagekraft, durch welche sich so mancher in falsche Sicherheit wiegen lässt. Sie funktioneren gut auf tiefen Layers sowie in Libraries, aber nur mit Einschränkungen auf höheren Layers.

11.1.1. Faking & Mocking

Fakes, manchmal Stubs genannt sind ein simpler Ersatz, meist mit fixen Daten.

Mocks geben intelligente Rückgabewerte und schaut/kontrolliert, ob und die Funktionen aufgerufen wurden.

11.1.2. Fallbeispiele Externe Dienste:

FTP-Server FTP-Server: Nicht wegmocken, lieber lokalen Testserver aufsetzten (Docker o.ä.)

Börsenkurse Wegmocken, da auch (noch) nicht reale Szenarien getestet werden sollen.

11.1.3. Coverage

Es gibt verschiedene Arten von Coverage:

Anweisungsabdeckung (Statement Coverage) ist die Meistverwendete Variante, sie zählt Zeilen, welche durch Tests abgedeckt sind.

Zweigabdeckung (Branch Coverage) bedeutet, jeder mögliche Programmzweig (if/else, switch etc.) wird gezählt.

Bedingungsabdeckung (Decision Coverage) Das Verhältnis von ausgewerteten atomaren Werten (Term, Bedingung, ...) innerhalb von Ausdrücken zu allen vorhandenen atomaren Werten in einem Modul.

Pfadabdeckung (Path Coverage) Das Verhältnis der getesteten Pfade (Wege im Kontrollflussgraph), zu allen möglichen Pfaden in einem Modul. Die Pfadabdeckung ist kaum realistisch, da alle möglichen Kombinationen von Zweigen getested werden müsten.

Eine Coverage von 100% ist so gut wie ausgeschlossen, da Spezialfälle wie ungewöhnliche Exceptions aufgrund von Nebeneffekten nur schwer simuliert werden können. Der Coverage-Prozent Indikator ist daher eigentlich nur als Negativaussage nutzbar (z.B. 20% Coverage ist zu wenig).

Merke 11.1: Testabdeckung

Eine hohe Testabdeckung ist eine notwendige aber nicht hinreichende Bedingung. Wichtig ist die Qualität und nicht die Abdeckung. Integrationstests sind wertvoller wie Microtests.

11.2. Integration Tests

Integration Tests sind eigentlich Unit Tests auf höheren Etagen. Sie testen das System als Black Box, und sind daher langlebiger als Microtests. Drittsysteme werden manchmal gefakted oder weggemockt, oft aber auch direkt angesprochen (z.B. lokaler Datenbankserver). Integrationstests entdecken mehr Fehler, weils sie realistischere Szenarious testen.

Ein grosser Nachteil von Integration tests ist, dass diese oft sehr langsam sind, und sich daher nicht für CI eigenen.

11.3. Konsistenz-Tests für Datenbanken

Konsistenz-Tests für Datenbanken sind SQL Statements, welche wie Unittests die Konsistenz der Datenbank prüfen können. z.B. Gibt es Kunden, bei denen kein richtiger Namen eingetragen ist?

11.4. Build Server

Ein Build Server hat üblicherweise folgende Aufgaben:

- fetch from git
- compile, build EXE
- run Unit Tests
- run metrics
- run findbugs, checkstyle
- log everything
- dashboard (z.B. sonarQube)

Da es bei vielen Entwicklern häufig zu Build fehlern kommt, ist es ratsam, grosse Projekte im CI zu unterteilen.

11.4.1. Build Strategien

Daily Build, Nightly Build
Einmal alle 24 Stunden (weil der Build und die Tests sehr lange dauern, z.B. 6 Stunden)

Daily Build
Alle drei Stunden (weil der Build jedesmal ca. eine Stunde dauert)

Continuous Integration
Jedesmal, wenn ein commit auf den Haupt-Zweig gemacht wird (weil ein Build weniger als 10 Minuten dauert)

CI mixed
Tagsüber Continuous Integration mit allen schnellen/billigen Tests, nachts (oder alle drei Stunden) die vollen Tests

11.4.2. Deployment systeme

Oft gibt es folgende deployment (meist teilautomatisierte) Systeme:

Development
Entwicklersystem (relativ stabil für alle Entwickler), Daily Build / CI Machine

Test
Stabile Test-Umgebung für die Entwickler Alle automatisierten Tests Erste Performance-Tests

Acceptance / Staging ($\approx$ Staging):
Manuelles Testen durch Kunde und externe Tester; HW & Testdaten möglichst gleich wie PROD (ausser bei Deployment-Änderungen); hier auch Performance-Tests

Productive
Produktiv-Umgebung, muss stabil sein, keine Experimente! Neue Releases nur alle paar Monate.

Testdaten werden üblicherweise vom Productive richtung Development-System übernommen und z.T. anonymisiert.

Das Testsystem sollte etwa täglich, Acceptancesystem bei Sprint Ende und das Produktivsystem etwa alle zwei Monate propagiert werden.

12. Aufwandschätzung

Rund 70% der Informatik Projekte sind nicht erfolgreich oder über dem Budget. Dabei kommt es öfter zu "Scope Creep", d.h., der Umfang des Projekts wurden laufend erhöht und geändert.

Merke 12.1: Faustregel

Grosse Projekte schlagen oft fehl, darum möglichst nie Projekte mit $>$ 9 Monate, $>$ 1 Million Budget oder Team $>$ 10 Personen!

Die Kommunikation zwischen Teammitglieder verhält sich mit $\frac{n \cdot (n-1)}{2} \rightarrow \mathcal{O}(n^2)$, was einer der grössten Einflussfaktoren eines Projektes ist. Dazu kommen weitere nicht-lineare Faktoren:

- Communication
- Planning
- Management
- Requirements development
- System functional design
- Interface design and specification
- Architecture
- Integration
- Defect removal
- System testing
- Document production
- Unterschiedliche Auffassung in der Vision
- Unterschiedliches technisches Verständnis
- Unterschiedliche Muttersprache

12.1. Warum gibt es Fehlschätzungen

- Politische Vorgaben dominieren, nicht realistische
- Zuviel Optimismus
- Fehlende Sorgfalt, mangelnde Qualität (quick & dirty stays)
- komplexität und unerwartete Nebeneffekte zerstören die Pläne
- **Fehlende Erfahrung/Übung**

12.2. Einflussfaktoren

Nach ISBSG.org:

1. Project size
2. Size of customer base (how many installations)
3. Stability of requirements
4. Cooperation with customer (customer in team)
5. Team fitness/quality
6. Complexity of system (real-time, distributed, multi-platform,edge of technology)
7. Security and safety requirements (banking, medical)

12.3. Vorgehensweisen

12.3.1. Top-Down

Zuerst das Gesamte, dann Teilbereiche schätzen. Wird meistens zu Begin der Elaboration-Phase gemacht.

COCOM-Schätzung

12.4. Bottom-Up

Bei vorliegen aller Requirements, Arbeitspakete sowie einem Entwurf; die einzelnen Elemente werden geschätzt in der Grössenordnung von 0.5 bis 3 Tagen.

Wird meistens gegen das Ende der Elaboration-Phase gemacht.

12.5. Arbeitsfortschritt

Meistens nimmt der Arbeitsfortschritt keine lineare Form, sondern eher die Form eines Sigmoids an. Dies hat folgende Gründe:

- Schwieriges wird (unbewusst) hinausgeschoben, bzw. nicht genau genug abgeschätzt.
- Umgesetze Funktionen funktionieren zwar, verursachen aber an anderen Orten Probleme.
- Kunde sieht SW funktionieren, jetzt kommen Korrekturen und Wünsche.
- Refactoring verbessert den Code ohne dass mehr Funktionalität sichtbar ist. Der Kunde sieht den Fortschritt nicht
- Verbesserungen von Robustheit oder Security kosten viel Arbeit, führen aber nicht zu sichtbarer Funktionalität.

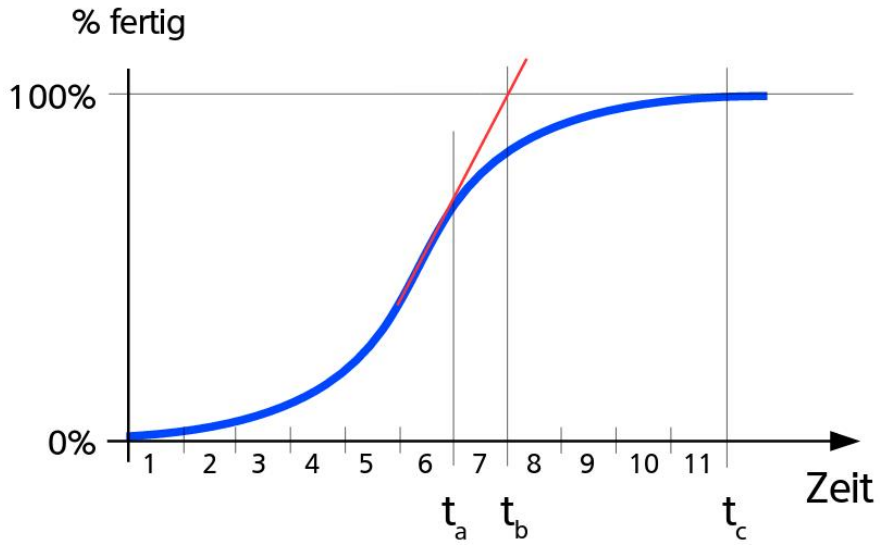

Abbildung 9: Arbeitsfortschritt (Sigmoid)

- t_a: Alle läuft gut, das Team ist optimistisch
- t_b: Der Projektabschluss Zeitpunkt wird von dem optimistischen Team extrapoliert.
- t_c: Tatsächlich get das Projekt aber noch 5 mal so lang, denn die letzten Reste aufzuräumen kostet überproportional viel (80/20 Regel).

Dauert ein Projekt mit 10 Entwickler 2 Jahre, macht es keinen Sinn, das Projekt in einem Jahr mit doppelt so vielen Entwickler umzusetzen. Der Aufwand und somit die Kosten steigen stark an.

12.5.1. LOC pro Monat

Gemessen am gesamten Team inkl. Administration und ohne Tests.

80-150 LOC bei schwierigen Echtzeit-Projekten (Zahl v. IBM, Space Shuttle)

300-500 LOC im Schnitt mit einem guten, nicht zu grossen Team

bis ca. 1000 LOC nur mit kleinen Spitzenteams

über 1500 LOC ist unglaubwürdig (Pfusch, viel Copy/Paste, generiert)

13. Software-Metriken

- Software-Metriken sind z.B. beim Einstieg in ein bestehendes, fremdes Projekt nützlich.
- Metriken sind immer Interpretationssache: Es ist schwierig, vernünftige harte Grenzen festzulegen.
- Entwickler sollten allerdings nicht nach Metriken belohnt werden: Kann unerwartete Nebeneffekte haben!
- Metriken die im Buildprozess integriert sind, geben nützliche Hinweise für ein allfälliges Refactoring.

13.1. Kennmerkmale

Es wird unterschieden in Produktmetriken (Kennzahlen der Software, Zyklomatische Komplexität) und Projektmetriken (Issues, Number of Commits, etc.). Der Fokus liegt in erster Linie auf Produktmetriken.

- Suche nach *.* => was liegt überhaupt vor? welche Sprachen?
- Dokumentation lesen (und verifizieren ?=? tatsächliche Architektur)
- Eingesetzte Tools? (CVS, findbugs/checkstyle/ReSharper, cobertura...)
- Eingesetzte Libraries, Frameworks (zu wenig? veraltet?)
- Unit Tests? (genügende Abdeckung, sinnvolle Tests?)
- (Un)aufgeräumte Baustelle? (uneinheitlich, TODOs, dead code, ...)
- Welche Dateien werden am häufigsten ein- und ausgecheckt?
- Grösste Klassen finden (countlines, notfalls Dateigrösse in KB)
- Komplexität ermitteln (metrics, notfalls Suche nach '&&')
- Lesbarkeit?
- Erweiterbarkeit?
- Sicherheit, Robustheit, ...?

13.2. Die Zyklomatische Zahl (McCabe Metrik)

Misst Komplexität der logischen Struktur auf Basis des Kontrollflussgraph G eines Programmes.
Die Zyklomatische Zahl kann mit folgender Formel berechnet werden:

$$V(G) = e - n + 2p$$

wobei

e Anzahl Kanten (edges)

n Anzahl Knoten (nodes)

p Anzahl Komponenten (unabhängige Teil-Graphen]

Für eine einzelne Methode kann auch die vereinfachte Form verwendet werden:

$V(G) =$ Anzahl Verzweigungen (while, if, case) $+1$

Merke 13.1: Interpretation

Ist die zyklomatische Zahl V(G) einer Funktion/Methode > 10, müss geprüft werden, ob sie eventuell zu vereinfachen ist.

13.3. Lack of Cohesion of Methods (LCOM*)

* Variante nach [Henderson96]

$$LCOM* = (m - Mittelwert(m(A)))/(m\breve{}1)$$

m Anzahl Methoden

$m(A)$ Anzahl Methoden, die auf ein Attribut zugreifen

Interpretation: LCOM* = 1 bedeutet schlechte Kohäsion, LCOM* = 0 bedeutet maximale Kohäsion

13.4. Afferent und Efferent Coupling

Afferent Coupling Eingehende Kopplung: Wie viele Klassen sind von mir abhängig?

Effferent Coupling Ausgehende Kopplung: Von wie vielen Klassen bin ich abhängig?

13.5. Instability

$$I = \frac{C_e}{C_a + C_e}$$

C_a Afferent Coupling: Anzahl Klassen in einem Package, welche von Klassen innerhalb des Packages abhängen

C_e Efferent Coupling: Anzahl Klassen eines Packages, welche von Klassen ausserhalb des Packages abhängen

Interpretation: Umso kleiner I ist, umso "stabiler" ist der Code (d.h., er hat wenig externe Abhängigkeiten). Stabile Klassen sind oft weit unten, da sie von vielen anderen Klassen verwendet werden. (I=0)

13.6. Abstractness

A = Number of abstract classes

13.6.1. Normalized Distance from Main Sequence

$$D_n = |A + I - 1|$$

I Instabilität (0-1)

A Anzahl (abstrakter und normaler) Klassen eine Packages (0-1)

Interpretation: D_n sollte möglichst gering sein bei guten Package Design.

Zonen

Zone of Uselessness In dieser Zone gibt es viele abstrakte Klassen, die nur von wenigen anderen Klassne gebraucht werden.

Zone of Pain In dieser Zone gibt es viele konkrete KLassen mit vielen Abhängikeiten.

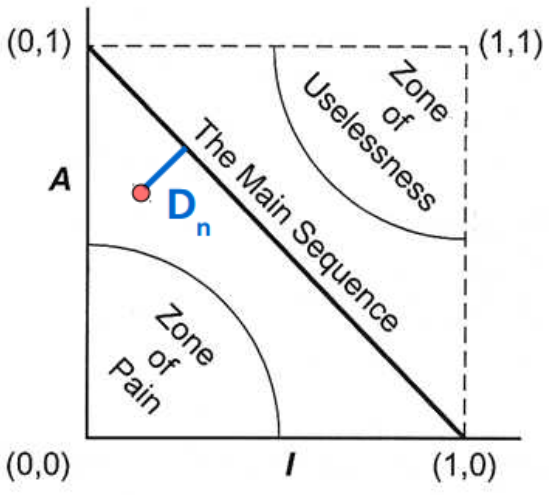

Abbildung 10: Zone of Exclusions

13.7. Tooling

Für Software Metriken gibt es viele nützliche Tools in Java.

13.7.1. Codeanalyse

Codeanalysen helfen, die Codequalität in Projekten zu erhöhen.
Für die Visualisierung und Bearbeitung der nachfolgenden Metriken eignet sich SonarQube.

Für Java:

Structure101 Visualisiert Abhängikeiten von Code bzw. verschiedene Strukturelemente

Checkstyle (Java) Prüft Quellcode nach Konventionen und einfachen Metriken

findbugs (Java) Analysiert Bytecode nach logischen Fehlern

Für .NET: NDepend, FXCop, ReSharper

13.8. Zusammenfassung Metriken

- Oft: je grösser ein Projekt, desto mehr Fehler sind vorhanden.
- Metriken können schnell die neuralgischen Punkte aufzeigen. (keine Steigerung der Qualität)
- Metriken können zeigen, ob die Verbesserungen in die richtige Richtung gehen.
- Richtig gute Software-Qualitätsmetriken gibt es nicht; Abhilfe: Reviews.
- Metriken sind Interpretationssache (also Erfahrungssache und Sache von Spezialisten, daher auch nichts für Manager).
- Achtung: Manager lieben KPIs (Key Performance Indicators) und hätten gerne SW-Metriken dazu. Fast nie eine gute Idee.
- Metriken & Belohnung = Desaster.

14. Code Reviews

Reviews sind eine sehr kosteneffiziente Art, Fehler zu entdecken. Zudem wird das Team insgesamt besser und es gibt einen Know-How transfer.

Es gibt verschiedene Formalitätsgrade von Reviews:

formlos: Pair Programming

- Github Pull Requests
- Gruppen-Reviews

formal: Formal Inspection (IBM)

14.1. Gruppen-Reviews

Meist mehrere Entwickler mit definierten Rollen (moderator, author/presenter, peer, note taker), welche in einer begrenzter Zeit (meist 2-4h) Code anschauen. Dabei wird ein Protokoll geschrieben.

Code Reviews sollten möglichst Zeitnah gemacht werden, da die Behebung mit der Zeit teurer wird. Dies gilt speziell für Regulierte Umgebungen.

1. Stück Code oder Dokumentation auswählen (ca. 500 LOC pro 2h)
2. Personen auswählen (peers) und Rollen verteilen (3-5)
3. Termin finden und Personen Einladen. Links zu Requirements, Architecture Document und Code anhängen.
4. Personen sollten sich die Dokumente im Voraus kurz anschauen, statische Code Tests (14.2)
5. Review durchführen
   - Code walk-trough, geführt vom Autor
   - Es sollte jeder in der Gruppe mal als Author im Review an die Reihe kommen.
   - Kommentare der Reviewer im Protokoll notieren
   - Erstes Ziel ist das finden von Fehlern.
   - abschliessen der Bewertung mit *Gut*, *OK mit Nacharbeiten* oder *nicht OK*
6. Nacharbeiten begleiten und abschliessen. Protokoll ablegen.

14.1.1. Wichtigste Fragen beim Code-Review

- Verständlichkeit (Warnsignal, wenn nicht alle den Code verstehen)
- Namenswahl (Methoden, Variablen, packages; das kann kein Tool)
- "Code Smells" (die üblichen Verdächtigen)
- Übereinstimmung mit Architektur-Ideen und Diagrammen

14.1.2. Format des Protokolls

- Teilnehmende und deren Rollen, Zeit, Ort
- Identifikation des Code-Stücks und der relevanten Unterlagen
- Tabelle mit den offenen Punkten: ID, Beschreibung, Schweregrad, Datum & Kürzel wenn behoben, Req.Ref. Bem.
- Aufgabenverteilung: Wer macht was bis wann (am besten gleich Issues erstellen)
- Verdikt („akzeptiert/akzeptiert mit Nacharbeiten/zurückgewiesen“

14.1.3. Nacharbeiten

Für jeden gefundenen Fehler:

- mindestens einen Unit Test schreiben, der den Fehler provoziert
- Fehler beheben (und/oder refactoring) und dokumentieren
- Unit Test muss jetzt OK sein

14.2. Code Analysis Tools

Code Metriken sind eine gute Unterstützung beim Code Review. Mit Tools können folgende Teilgebiete abgedeckt werden:

- Code Metriken
- Überprüfung des Programmier-Stils
- Prüfung auf Anomalien und mögliche Fehler
- Auswertung der Testabdeckung

(metrics, checkstyle, ESLint, ReSharper, findbugs, EMMA, structure101 etc.)
Diese Tools sollten bei jedem Build sowie auf der Entwickler-Maschine laufen.

14.3. Requirements Reviews

Requirements Reviews helfen, von wolkigen Kundenwünschen zu formalen Requirements Specs zu kommen.

14.4. Architektur-Reviews

- Nicht funktionale Anforderungen (u.a. Performance, Security)
- Kern-Charakteristiken des Systems
- Architektur-Dokumentation stimmt mit Implementation überein
- Szenarien durchspielen (z.B. ATAM Szenarien)

15. Performance Messungen

Voreilige Optimierungen für die Performance sind kontraproduktiv. Immer zuerst fragen: haben wir ein Problem?

Ein häufiger Fehler ist eine voreilige Parallelisierung.

Faustregel:

1. Get it to run
2. Get it right
3. Make it fast

15.1. Java Profiling

Tools:

- NetBeans
- JProfiler
- AppDynamics
- NewRelic

15.2. Last-Messungen (Black Box)

Wann geht der Server in die Knie?

15.2.1. Tools

- gatling.io, JMeter

15.3. Performance-Messungen

Sind die typischen Antwortzeiten immer noch etwas gleich?

15.3.1. Tools

- Selenium (evtl. JMeter/gatling, diese können aber kein JS)
- Headless Browser
- Endlos viele Cloud Dienste

15.3.2. Probleme mit Testdaten

Mit testprozeduren gibt es ein Problem: Sie widerspiegeln oft nicht das Verhalten von Benutzern. Dies hat speziell Einfluss auf Caching und Memory Paging bei Datenbanken.

15.4. Dashboards

Dashbords helfen, einen schnellen Überblick über einen aktuellen Projektstand zu erhalten. Zielgruppe kann Business (z.B. Umsatzzahlen, Marketing-Effekt), Application (z.B. sizes Grösse von Tabellen), Server Infrastruktur

15.4.1. Tools

- Grafana / Graphite
- Diverse Monitoring Lösungen

16. Usability Testing

16.1. Zutaten

- Die neue, zu testende Software
- Mehrere nicht vorbelastete Nutzer (User)
- Ein Usability Lab
- Eine(n) Moderator(in) (facilitator, assistant)
- Mehrere Szenarien zum Testen (Use Cases)

16.2. Durchführen

Etwa in der Mitte des Projektes mit mindestens 3 geeigneten/normalen Testusern

1. Kurze Einführung
   - "Helfen Sie uns" "Denken Sie laut" " Sie können nichts falsch machen" "Die Software wird getestet, nicht Sie"
2. Proband wird in ein Szenario eingeleitet
3. Der Benutzer probiert ohne Hilfe, das Szenario zu "lösen".
   - Der Beobachter macht Notizen, aber greift nicht ein
4. Erkenntnisse werden niedergeschrieben
5. Bedanken beim Benutzer für die Wertvolle Hilfe

17. Architektur

Architektur ist die Summe der Design-Entscheide, die von grosser Tragweite sind, und die länger leben.

Architektur ist die Aufteilung im Grossen.

IEEE STD 1472 Definition: "An architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment and the principles guiding its design and evolution."

Darunter fallen z.B: Interfaces, Datenstrukturen, Grosse Charakteristiken, Cross-cutting concerns (security, logging, exception handling etc.)

17.0.1. Erfolgskriterien für SW-Architektur

Einfach verständlich - und damit auch gut dokumentierbar und kommunizierbar

Stabil d.h. vorausschauend, langlebig

Skalierbar z.B. durch verschiedene Deployment-Varianten

Gut aufgeteilt und somit gut parallel entwickelbar

Gut testbar z.B. durch gut sichtbare Schnittstellen

Erweiterbar (mit zusätzlicher Funktionalität)

Adäquat (kein Over-Engineering) für: Performace, Security, Stability

17.1. Architektur-Präsentation

- Übersicht: Kontext-Diagramm, Demo oder 'Standbilder' (UI und wichtigste UCs)
- Orientierung, Strukturierung:
  - Schichten (UI - Logik - Libraries)
  - Deployment
  - Interfaces (besonders zwischen UI und Rest des SW)
  - Datenstrukturen (Domain-, Datenmodell)
  - Datenspeicherung
- Einordnung, Aufzählungen:
  - Statistiken: Anzahl Klassen, LOC, Sprachen, Teamgrösse, Aufwand
  - Geschichte, Jahreszahlen
  - verwendete Libraries (react, node, flux)
  - Nicht-funktionale Anforderungen und Design-Entscheide
- Besonderheiten
  - Parallele Prozesse und MQ, verteilte NoSQL DB, High Security, Plug-In System
- Rundgang (1:1 erleben):
  - UI - Models & Logic - DB - Models & Logic - Anzeige (Demo und Sequenzdiagramm)

Abbildung 11: Architektur-Präsentation

17.2. Architektur-Überlegungen

System-übergreifende Überlegungen (Cross-cutting concerns):

- Fehlerbehandlung, Exception handling
- Logging (spannend wenn über mehrere Maschinen hinweg)
- Transaction handling
- Internationalisierung
- Backup, Löschen von Überflüssigem, z.B. Log-Entries in DB
- Deployment

Qualität:

- Performance-Überlegungen (Mengen, Zeit, Geschwindigkeit), eventuelle Real Time and Space constraints
- Security/Safety-Überlegungen
- andere Qualitäts-Überlegungen (Erweiterbarkeit, Plattform-Portabilität, Verfügbarkeit...)

Analyse zu den Requirements:

- Beschreibung der umliegenden Systeme und Aktoren, Schnittstellen
- Datenmodellierung (statisch, "was merken wir uns?")
- Prozessbeschreibungen (dynamisch, "was läuft ab?")
- Randbedingungen (IT Landschaft, politisch, juristisch)
- Nutzer, Rollen & Rechte
- Vorgaben zu Security, Performance, Usability, Portability...
- UX-Vorgaben (oft Browser; besonders wichtig bei iOS/Android)

17.2.1. Bausteine zur Beschreibung

- Context Diagramm (informelle und ausführlichere Darstellung von Aktoren und deren Interaktion)
- Schnittstellen zu Umsystemen
- Use Case Diagramm
- Prozessbeschreibung: Aktivitätsdiagramm (Detaillierte Beschreibung von Prozessen)
- Deployment-Diagramm
- Schichten-Architektur (packages)

17.2.2. Technische Architektur-Entwurfsmuster

- Single User Desktop Program (Einfache, lokale Programme)
- Multi-Tier Architectures (Skalierbarkeit, Performance. Sehr komplex)
- Fat Client + Server (grosse Desktop-Programme und Mobile Apps)
- Thin Client (Browser Based + Server)
- DB-zentrierte Architektur (Skalierbar, einfacher Aufbau, aber stark von DB abhängig)
- Produzent → Konsument (Konsument muss schneller sein als Produzent)
- Message Basierte Systeme (Skalierbarkeit, Performence, aber komplex)

Bei Multi-Prozess-Architektur müssen folgende Punkte sorgfältig definiert werden:

- stateful/stateless
- singlethreaded/multithreaded
- Anfrage & Antwort: asynchron/synchron (pro Anfrage)
- first-in/first-out oder priority queue
- point-to-point oder multipoint/broadcast communication

17.2.3. Enterprise Integration Patterns

Diagramm-Details sind nicht Prüfungsrelevant

17.3. Software Architektur Dokumentation

Beschreiben, wie das System aufgebaut wurde und warum das so gemacht wurde.
Die Doku hat zwei Hauptteile:

- Umfeld und Randbedingungen
- Technische Struktur

Struktur:

1. Dokumentation Umfeld
   - Context Diagram: Übersichtsbild, wichtig: die umliegenden Systeme und Aktoren
   - IT-Landschaft mit Schnittstellen zu anderen Systemen
   - Beschreibung der Datenhoheit in anderen Systemen: wo werden die Daten 'geboren', wo ist der 'single point of truth'
   - Beschreibung der Aktoren, Rollen und Rechte
   - Business Process Modelling
   - Randbedingungen (juristisch, historisch ...)
   - (Referenz zu Use Cases mit Use Case Diagramm)

- (Referenz zu Domainmodell)
- (Referenz zu nicht-funktionalen Anforderungen)

2. Dokumentation Technische Struktur
   - Deployment diagram (was läuft wo), oft verschiedene Deployment-Szenarien, (meist sehr high-level)
   - Schichten-Architektur, UML top level package/class diagram (was auf eine A3-Seite passt), zeigt, wie die Klassen/Pakete strukturiert sind.
   - Datenmodell: UML class diagram oder E-R Diagramm.
   - Bei den Beschreibungen v.a. auch auf die Assoziationen achten.
   - Zustandsdiagramme/Lebenszyklen der wichtigsten Entitäten
   - Prozesse, Threads mit ihren Charakteristiken; Message Queues
   - UX Skizzen, Personas & Szenarien, Screen shots, Erklärungen
3. Zoom: Technische Details mit Kommentaren, warum.
   - UML Klassendiagramm als Übersicht: nicht zu detailliert, oder dann 1:1 aus Code generiert (aber: "Welchen Sinn hat eine Landkarte im Masstab 1:1 ?SZitat Umberto Eco)
   - UML Klassendiagramm als Detailbeschreibung v. etwas Kompliziertem
   - UML Zustandsdiagramme
   - UML Sequenzdiagramme
   - parallele Prozesse (UML?)
   - Gut auch: wieder verworfene andere Lösungen, warum verworfen; wo ist welche Erweiterbarkeit/Anpassbarkeit eingebaut (config files, scripting extensions, strategy pattern) und warum
4. Zoom: Dynamik
   - Prozesse, Threads mit ihren Charakteristiken; Prozess-Interaktion über Message Queues (z.B. JMS)
   - UX Architektur (welche Screens folgen auf welche)
   - Ablauf-Szenarien, z.B. Click - call - message - call - message, etc. (Sequenz-Diagramme)
   - Data flows, Control flows (Aktivitäts-Diagramme)

17.4. Architektur-Refactoring

Bei schlechter Architektur, kann folgendermassen vorgegangen werden:

1. Layering einführen/verbssern, so lange, bis es nur Zugriffe von oben nach unten gibt.
2. Mittlere Schicht entwirren: Aufteilung von Business-Services und evtl. Domainklassen.
3. Partitionieren und aufteilen von Klassen.

18. Scrum II

Bei vorgehen nach Scrum gibt es nur einen gröberen Architekturplan Scrum funktioniert nur, wenn man sich an die zwingenden Voraussetzungen hält:

- Teamgrösse max. 10
- kurzes Projekt max. 9 Monate
- Kunde weiss noch nicht so genau, was er will
- Ganzes Team an einem Ort
- Team deckt alle Fähigkeiten ab
- Häufige, mündliche Kommunikation
- Kunde im Team, ständig verfügbar
- Kunde will agiles Vorgehen

Bei gewissen Projekten bieten sich folgende Ergänzungen an:

- Checkpoint End of Elaboration (wie im Unified Process)
- Zweite, höhere Abstraktionsstufe zu User Stories im Backlog
  - Use Cases
  - Szenarien
  - Personas
  - Domain Model
  - Nicht funktionale Anforderungen
- Zusätzliche Rolle des Projektleiters, für all das, was der Product Owner und das Team nicht abdecken

18.1. Product Owner vs. Projektleiter

Der Product Owner vertritt die Kundenseite.
Der Projektleiter ist z.a. für folgende Punkte verantwortlich:

- Qualität und Testing
- Planung von z.B. Datenmigration, Datenverwaltung
- Umsetzungsentscheide bei Unstimmigkeiten
- Organisation
- Stunden und Rechnungen

18.1.1. Projektleiter ohne technisches Verständnis

Ist ein Projektleiter kein Entwickler, braucht er einen "ersten Offizier", welchem er blind vertrauen kann, und der sich sehr gut auskennt.

18.2. Daily Standup

Am Daily Standup dürfen nur aktiv am Projekt involvierte und arbeitende etwas sagen ("Pigs"). Am Rande Beteiligte dürfen zwar teilnehmen, aber nichts sagen ("Chicken").

18.3. Backlog

Der Backlog braucht Pflege, sogenanntes "Backlog Refinement":

- Schätzen
- Priorisieren
- Aufteilen
- Löschen

18.3.1. Impediments

Impediments (Hindernis) Tracking ist das äquivalent zum traditionellen Bug Tracking. Es dient in erster Linie dem aufzeigen von externen Abhängigkeiten / Blockaden

18.4. MVP: Minimum Viable Product

Bei Kunden, welche keine Entwickler sind, sollte bei jedem Sprint nach Möglichkeit eine sichtlich bessere Software herausfallen.

Bei Projekten, deren MVP sehr gross ist, funktioniert Scrum nicht. Bei Projekten, wo die Funktionen von vornherein 100% klar sind, macht Scrum nur mässig Sinn.

18.4.1. Fortschritt sichtbar machen

Für den Kunden

- Burn-Down Chart
- Story Map mit Farben
- Trends (Bug Reports, build failures, Metriken, ...)

Für die Entwickler

- Backlog
- Metriken
- Testabdeckung
- SonarQube

Für die IT-Infrastruktur

- Dashboards mit Ressourcen-Verbrauch (Disk, CPU...)
- Log-Auswertungen visualisiert

19. Proving Programs

Mittels Unit Tests lässt sich nur die Anwesenheit von Bugs beweisen, aber nicht deren Abwesenheit. Deshalb ist ein interessanter Ansatz, Programme formal zu beweisen.

19.1. Hoare Triples

$$\{P\}\, C\, \{Q\}$$

Wenn das Programm C von einem Zustand ausgeführt wird, welche der Vorbedingung P genügt, ist nach der Ausführung die Nachbedingung Q erfüllt.

P, Q und das ganze Hoare Triple sind dabei Prädikate. P und Q zeigen den Zustand des Programms mittels den Variablen in C als freie Variablen.

19.2. Automatisierung von Tests

19.2.1. if

```
{T}
{(x < 0 => -x >= 0) AND (not x < 0 => x >= 0)}
if x < 0 then
   {x >= 0}
   y := -x
else
   {x >= 0}
   y:=x
{y >= 0}
```

Resultiert in der verification condition: $\vdash (x < 0 \implies -x \leq 0) \wedge (\neg(x < 0) \implies x \leq 0)$

19.2.2. while

Ganz beweisen lassen sich while loops nicht (aufgrund des Halting Problems.)

Loop invariant ist eine Eigenschaft, welche immer am Ende einer loop wahr sein muss (d.h. sie bleiben gleich, bzw. ändern sich vorhersagbar).

Ein Loop invariant zu finden, kann nicht automatisch gelöst werden.

```
{x >= 0}
{0 <= x}
i := 0;
{i<=x}
while NOT(i = x) do {Inv : i <= x}
   {i<=x ^ NOT(x=i)}
   {i+1 <= x}
   i := i + 1
   {i<=x}
{i <= x AND NOT (NOT(i = x))}
{i = x}
```

Wenn wir folgendes Beweisen können, ist die Loop invariant richtig:

$$\vdash i \leq x \wedge x \neq i \implies i + 1 \leq x$$
$$\vdash i < x \implies i + 1 \leq x$$
$$\vdash x - 1$$

Wir müssen dabei drei Beweisbedingungen beachten:

- Ist es eine loop-variant
  - Muss eine Bedingung sein, welche zwingend vorher und nachher (allgemein) gültig ist
- kann ich mit der loop-invariant zur post-condition kommen
- kann ich von allen vorbedingung zur loop-invariant implizieren

Danach betrachten wir die Schlaufe als Blackbox; wir müssen nun implizieren, dass wir die letzte postcondition aus der Invariant herleiten können
Das finden von loop variant ist nicht prüfungsrelevant.

A. Listings

B. Abbildungsverzeichnis

C. Tabellenverzeichnis