Software Engineering 2

Zusammenfassung SE2 FS14

Emanuel Duss

2014-08-19 13:38

Inhaltsverzeichnis

1 Continuous Integration **2**

2 Projektplanung **3**
2.1 Agil, iterativ 3
2.1.1 End of Elaboration 3

3 Software Engineering Practices **3**
3.1 Requirements Practices 3
3.2 Design Practices 3
3.3 Implementation Practices 3
3.4 Verification Practices 3

4 Advanced Code Design **3**
4.1 Error Handling Design 3
4.2 Concurrency Design & Testing (Parallelisierung) 4

5 Test Driven Development **5**
5.1 Der TDD Zyklus 5
5.1.1 Red, Green, Refactor & Integrate 5
5.1.2 Übersicht 5
5.2 Warum TDD? 6
5.3 TDD Patterns 6
5.3.1 Specify It 6
5.3.2 Frame It 7
5.3.3 Evolve It 7
5.4 Beispiel 7
5.4.1 Klasse `BonusCalculator` 7
5.4.2 JUnit Testklasse `BonusCalculatorTest`: 7

6 Refactoring **8**
6.1 Der Refactoring-Prozess 8
6.2 Refactorings 8
6.3 Code Smells 9
6.4 Refactoring to Patterns 9
6.5 Fazit 9

7 Code Review **9**
7.1 Vorgedingung 9

8 Code Smells (Industrial Logic Album) **9**
8.1 Typen 9

9 Common Code Smells 9

10 Software Design Patterns 10
10.1 Builder . . . 10
10.2 Prototype . . . 10
10.3 Flyweight . . . 10
10.4 Proxy . . . 10
10.5 Chain of Responsibility . . . 11
10.6 Interpreter . . . 11
10.7 Mediator . . . 11
10.8 Memento . . . 11
10.9 Visitor . . . 11

11 Produktmetriken 11
11.1 Metriken für Objektorientierte Software . . . 12
11.2 Software . . . 12
11.3 Dynamische Analyse . . . 12

12 Design By Contract (DbC) 12

13 Refactoring 13
13.1 Understanding Refactoring . . . 13
13.2 Refactorings . . . 13

14 Faking und Mocking 13
14.1 Testing unter heiklen Situationen . . . 13
14.2 Faking . . . 13
14.3 ## Begriffe . . . 13
14.4 ## Slip a Fake . . . 14
14.5 ## Faking Techniken . . . 14
14.6 ## Extract and Override . . . 14
14.7 . Heikle Teile in separate `protected` Methoden packen . . . 15
14.8 . Diese Methode in einer abgeleiteten Test-Klasse überschreiben . . . 15
14.9 . Diese abgeleitete Klasse testen . . . 15
14.10Mocking . . . 15
14.11## Begriffe . . . 15
14.12## Mocks . . . 15
14.13## Auto-Mocks . . . 16
14.14. Implementiere ein Interface oder erbe von einer Klasse. . . . 16
14.15. Schreibe für jede interessante Funktion ein Body. . . . 16
14.16. Packe alle übergebenen Argumente in eine Collection. . . . 16
14.17. Die Methode gibt Resultate von einer weiteren Collection zurück. . . . 16
14.18## Mocking-Tools . . . 16
14.19Fake Simulators . . . 17

15 Prüfungsvorbereitung 17

1 Continuous Integration

- Gerrit: "Firewall" vor Git Branches → "Master branch is always stable"

2 Projektplanung

2.1 Agil, iterativ

Inception Vorgaben werden vorgegeben, Abschluss mit Kickoff
Elaboration Ausarbeitung, eher Kostengünstig
Construction x
Transition Schluss mit Ablieferung

2.1.1 End of Elaboration

- Anforderungen definiert
  - UseCases helfen den Scope zu definieren
- User Interface Prototyp erstellt
- Objektorientierte Analyse gemacht
- Entwicklungs-Werkzeuge (IDE, GIT, Buildserver)
- Liste der Arbeitspakete

3 Software Engineering Practices

3.1 Requirements Practices

- Dig for Requirements (mit dem Benutzer Zusammenarbeiten)
- Make Quality a Requirement
  - Möglichst testbare Qualitätsanforderungen
  - Basierend auf echten Anforderungen
- Deal with Changes
  - Requirement Änderungen antizipieren
  - Design for Change
  - Kurze Iterationen
  - Change Assessment

3.2 Design Practices

- Don't Repeat Yourself (DRY)
  - Z. B. finale Konstanten
- Achieve Orthogonality
  - Low Coupling und hight Cohesion

3.3 Implementation Practices

3.4 Verification Practices

4 Advanced Code Design

4.1 Error Handling Design

- Defensive Programming: Systematische Fehlerprüfung aller Werte und Behandlung Alle Werte

- Ungültige Eingaben verhindern und Fehlermeldung (Exception, Keine Resultate bei ungültigen Input)
- Design by Contract
- Precondition: Abfangen ungültier Fälle
- Exceptions: Für mögliche produktive und sicherheitsrelevante Fehler
- Assertions: Ein- und ausschaltbar (`-ea`)
  - für Programmierfehler, die nie auftreten sollen;
  - Post- und Preconditions
  - Kann Seiteneffekte haben
- Fehlerbehandlungstechniken
  - Konservative Behandlung: Fehlermeldung anzeigen, beenden
  - Optimistische Behandlung: Neutrales Resultat mit einer Warnung
- Korrektheit (für sicherkeitskritische Systeme): Niemals ein ungenaues Resultat liefern
- Robustheit (für unkritische Systeme): Versuch die Sofware am Laufen zu halten
- Lokaler Exceptions Handler: Wenn nicht höher relevante Fehler; Nie ungültige Zustände hinterlassen!
- Globaler Exceptions Handler: An Aufrufer delegierenj; Protokollieren und Programm kontrollieren oder terminieren

Global Exception Handler:

```
Thread.setDefaultUncaughtExceptionHandler(new UncaughtExceptionHandler() {
  @Override
  public void uncaughtException(Thread t, Throwable e) {
    System.out.println("Fehler in Thread " + t + " mit Fehler " + e.getMessage());
    e.printStackTrace();
  }
});
```

4.2 Concurrency Design & Testing (Parallelisierung)

- Modellierung im Voraus
- Code testen
- Code für Nebenläufigkeit vorbereiten
- Aktive Instanzen festlegen
  - Prozesse, Threads und Tasks selche selbstaktiv sind erkennen
- Implizite Threads berücksichtigen (Thread Pools, Finalizers, asynchrone Calls, externe parallele Aufrufe)
- Klassendiagramm hilft dabei
  - Confined: Eine Instanz gehört gleichzeitig nur einem Thread
- Thread Save: Ich darf es gleichzeitig von mehreren Threads verwenden; Synchronisation ist intern implementiert
- Nicht Thread Save: Synchronisation muss von ausserhalb sichergestellt werden.
- Concurrency-Fehler
  - Race Conditions: Synchronisation definieren, Critical Sections definieren und schützen
  - Deadlocks: Nested Locks, Hierarcie für Nested Locks (`Timeout` bei `@Test` Annotation)
  - Starvation: Faire Synchronisationsprimitiven benutzen, keine Priority Inversion
- Testing
  - Es kann sein, dass der JUnit Test fertig ist, bevor die Exception im Thread geworfen wird. Mindestens ein Join, damit der Fehler in der JUnit Testmethode noch abgefangen werden kann.
  - Abhilfe: Global Exception Handler

Abhilfe zur Exception-Erkennung (als JUnit Basisklasse offerierbar)

```
private ConcurrentLinkedQueue<Throwable> uncaughtExceptions =
  new ConcurrentLinkedQueue<>();
@Before
public void before() {
  Thread.setDefaultUncaughtExceptionHandler(
    (thread, exception) -> uncaughtExceptions.add(exception));
}
@After
public void after() {
  for (Throwable throwable : uncaughtExceptions)
    throwable.printStackTrace();
  if (uncaughtExceptions.size() > 0)
    throw new RuntimeException("Errors in other threads");
}
```

5 Test Driven Development

5.1 Der TDD Zyklus

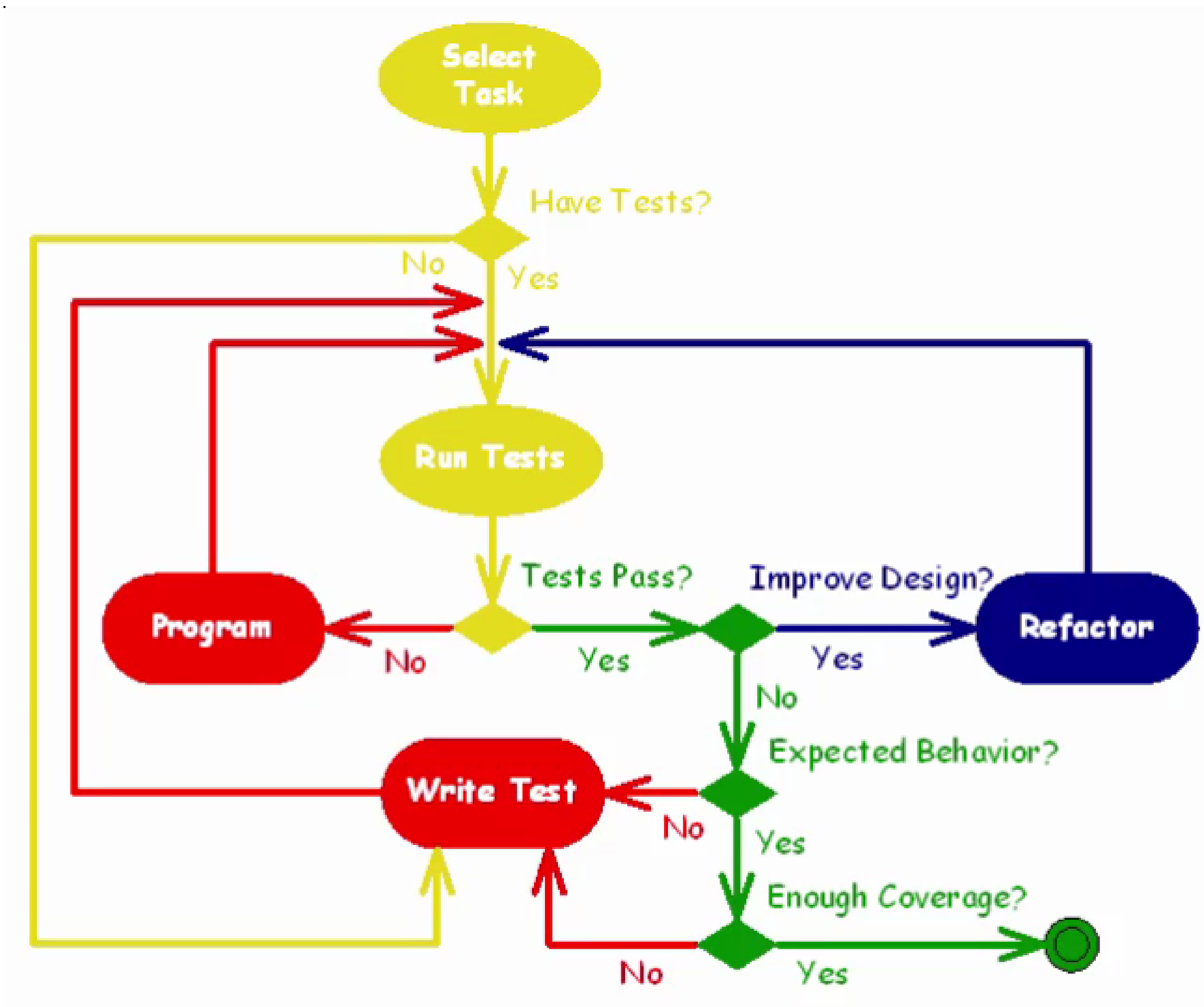
5.1.1 Red, Green, Refactor & Integrate

1. Red: Einen effektiven Mikrotest machen, der fehlschlägt
   a) Fördert Denkweise "was" statt "wie"
2. Green: Code ändern, damit Mikrotest durchläuft
   a) Code muss noch nicht schön sein
   b) Neuer Code geht nur soweit bis der Test durchläuft
   c) Existierende Tests sollen weiterhin durchlaufen
3. Refactor: Design der Änderung perfektionieren
   a) Selbe Funktionalität muss bestehen bleiben
   b) Methoden oder Klassen extrahieren, umbenennen, verschieben
   c) Lässt man das Refactoring weg, kann es unübersichtlich werden
4. Integrate: Änderung und Tests permanent machen
   a) In Versionsverwaltung einpflegen
   b) Wer oft committet, verhindert merge Konflikte
   c) Jeder Commit ist eine fallback Position
   d) Mach jetzt eine Pause! ;-)

5.1.2 Übersicht

Übersicht über den TDD Zyklus [1]

[1] Quelle: http://industriallogic.com



5.2 Warum TDD?

- Der TDD Zyklus zwingt einem kleine übersichtliche Schritte zu machen.
- Steigerung der Codequalität.
  - Achtung: Die Funktionalität des ganzen Systems kann nicht getestet werden.
- Die Mehrheit der Software hat "blöde" Fehler.
- Mit TDD steigert man die Produktivität, man kann aber nicht die Funktionalität eines ganzen Systems testen
- Weniger den Debugger nutzen

5.3 TDD Patterns

Beim TDD soll man so vorgehen:

5.3.1 Specify It

1. Essence First: Was ist das wesentliche?
2. Test First: Guten Methodennamen für den Test definieren
3. Assert First: `assert` formulieren: Was wird getestet und was ist das erwartete Ergebnis? Die zu testende Methode muss noch nicht implementiert sein.

5.3.2 Frame It

1. Frame First: Grundgerüst vorbereiten (Klassen, Konstruktoren, Methoden, ...)

5.3.3 Evolve It

1. Do The Simplest Thing That Could Possibly Work: Möglichst einfacher Code schreiben, damit der Test durchläuft.
2. Break It To Make It: Neues Assert schreiben, um mehr Funktionalität zu erreichen.
3. Refactor Mercilessly: Design des Codes verbessern
4. Test Driving: So weitermachen

Wichtig Pro Assert sollte man eine eigene Testmethode machen und nicht pro Methode mehrere Asserts verwenden.

5.4 Beispiel

Folgendes sind die Lösungen vom Test[2]:

5.4.1 Klasse `BonusCalculator`

```
public class BonusCalculator {
   public double individualBonus(int sales, int quota,
      double commission, double tax) {
      return baseBonus(sales, quota, commission) * taxAdjustmentFactor(tax);
   }
   private double taxAdjustmentFactor(double tax) {
      return 1.0 - tax / 100.0;
   }
   public double teamBonus(int sales, int quota, double commission,
      int numberOfTeamMembers) {
      if (numberOfTeamMembers == 0)
         return 0.0;
      return baseBonus(sales, quota, commission) / numberOfTeamMembers ;
   }
   private double baseBonus(int sales, int quota, double commission) {
      return  (sales - quota) > 0 ? (sales - quota) * commission / 100.0 : 0.0;
   }
}
```

5.4.2 JUnit Testklasse `BonusCalculatorTest`:

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;
public class BonusCalculatorTest {
   private static final double precision = 0.001;
   private BonusCalculator bonusCalculator;
   @Before
   public void setUp() {
      bonusCalculator = new BonusCalculator();
   }
```

[2]Quelle: http://industriallogic.com

```
    @Test
    public void individualBonusWhenSalesAboveQuota() {
        assertEquals(90.0, bonusCalculator.individualBonus(12000, 11000, 10.0, 10.0),
          precision);
    }
    @Test
    public void individualBonusWhenSalesBelowQuota() {
        assertEquals(0.0, bonusCalculator.individualBonus(12000, 15000, 10.0, 10.0),
          precision);
    }
    @Test
    public void individualBonusWhenSalesEqualQuota() {
        assertEquals(0.0, bonusCalculator.individualBonus(12000, 12000, 10.0, 10.0),
          precision);
    }
    @Test
    public void teamBonusWhenSalesAboveQuota() {
        assertEquals(25.0, bonusCalculator.teamBonus(12000, 11000, 10.0, 4),
          precision);
    }
    @Test
    public void teamBonusWhenSalesBelowQuota() {
        assertEquals(0.0, bonusCalculator.teamBonus(12000, 15000, 10.0, 4),
          precision);
    }
    @Test
    public void teamBonusWhenSalesEqualQuota() {
        assertEquals(0.0, bonusCalculator.teamBonus(12000, 12000, 10.0, 4),
          precision);
    }
    @Test
    public void teamBonusWhenNoTeamMembersExist() {
        assertEquals(0.0, bonusCalculator.teamBonus(12000, 11000, 10.0, 0),
          precision);
    }
}
```

6 Refactoring

6.1 Der Refactoring-Prozess

1. Problem erkennen
2. Refactoring wählen
3. Refactoring in kleinen Schritten durchführen und fortlaufend testen (kleine Änderungen können besser rückgängig gemacht werden).

6.2 Refactorings

Vgl. Vorlesungsfolien.

6.3 Code Smells

6.4 Refactoring to Patterns

6.5 Fazit

7 Code Review

- Eclipse Plugin `metrics`
- Testabdeckung: Welche Codezeilen werden mit Tests abgedeckt.

7.1 Vorgedingung

- Static Code Analysis, Code übersetzt richtig, keine Warnungen und Fehler, Dokumentation aktuell
- Durchführung: Fehler finden (nicht gleich fixen)

8 Code Smells (Industrial Logic Album)

- Code Smells sagt, wie man Code nicht designen soll und zeigt klassiche Fehler auf.

8.1 Typen

- Feature Envy
- High Technical Debt: Nach Jahren ist der Code unlesbar und unwartbar (lange Methoden, niemand blickt mehr durch...)

9 Common Code Smells

- Duplicate Code :Wenn zwei Codestücke gleiche oder sehr ähnliche Sachen machen. (→ Extract Methode)
- Comment: Der Code sollte genug Dokumentieren
- Long Method: Nicht lesbar, nicht teilbar, nicht testbar (→ Extract Method) ausgelagert
- Large Class: Zuviele Methoden, Code, Variabeln und Verantwortungen in einer Klasse (→ Extract Class, Move Method)
- Dead Code: Löschen!
- Lazy Class: Klasse macht zuwenig, das kann in eine andere Klasse ausgelagert werden (→ collapse Hierarchy, Inline Class)
- Oddball/Inconsistent Solution: Zwei Probleme werden auf verschieden Weisen gelöst, andere Methoden müssen evtl. refactored werden. Z. B. Magic Numbers statt Konstanten (→ Substitute Algorithm)
- Primitive Obsession: Ein Problem wird mit zu einfachen Mitteln gelöst und ist somit zu kompliziert (Primitiven, Low-Level-Methoden, String-Konkaternation), es gibt bessere und einfachere Lösungen. (→ Replace Data Value With Class)
- Switch Statement: Man sollte Polymorphismus in Betracht ziehen (→ Replace Conditional with Polymorphism)
- Speculative Generality: Entsteht beim Gedanken, dass ein Feature vielleicht später einmal kommen wird. (→ Inline Class, Collapse Hierarchy)
- Long Parameter List: Zeichen von Primitive Obsession; Auch Methoden mit Default-Argumenten können unleserlich werden. Beheben: Parameter Object (start, end = Range); Daten von sich nehmen, statt übergeben; Methode aufsplitten

- Conditional Complexity: Ein System wird mit vielen verschachtelten if-Abfragen und Bedingungen immer komplexer (→ Code in mehrere Klassen/Subklassen auslagern, Explaining Variable einfügen, separate Methoden)
- Combinatorial Explosion: Wenn Teile von Methoden und Klassen mehrmals verwendet werden (findPriceAndColor, findPricee) (→ Protection Proxy Pattern)
- Alternative Classes With Different Interfaces: Wenn Klassen dasselbe tun, aber unterschiedliche Interfaces nutzen (→ Rename)
- Inappropriate Intimacy: Wenn eine Klasse zu viel private Sachen nach Aussen gibt (Z.B. setWorhshops statt add/removeWorkshop)
- Indecent Exposure: Manche Klassen dürfen nicht sichtbar sein.
- Refused Bequest: Wenn Klassen nicht alle Methoden eines Interfaces implementieren wollen (implementieren einfach eine do-nothing Methode). Wird manchmal dazu verwendet um Combinatorial Explosion: zu verhindern. (→ Vererbung mit Delegation ersetzen)
- Black Sheep: Eine Unterklasse passt nicht zu den Superklassen.
- Data Class: Falls eine Klasse nur Daten enthält (kann aber auch gut sein: Kommunikation zwischen Tiers, verteilte Systeme) (→ Nützliche Methoden einführen)
- Solution Sprawl: Methoden und/oder Daten sind über zuviele Klassen verteilt. (→ Factory Class)
- Feature Envy: Methoden/Daten für ein Job sind am falschen Ort (Anzeichen: viele Aufrufe in eine andere Klasse) (→ Code in andere Klassen verschieben)
- Temporary Field: Eine Instanzvariable sollte während der gesamten Lebenszeit eines Objekts eine Bedeutung für das ganze Objekt haben und nicht nur für eine einzige Methode (→ Eine Methode soll die temporären Felder berechnen)
- Side Effect: Wenn eine Methode mehr macht, als als der eigentliche Zweck.

10 Software Design Patterns

- Patterns werden immer wieder neu Programmiert (kein Reuse)

10.1 Builder

- Konstruktionsschnittstelle
- Für verschiedene Repräsentationen von verschiedenen Typen
- Jeder Typ hat ein Konstruktionsbefehl

10.2 Prototype

- Neue Objekte erzeugen durch kopieren des Prototyps, welche angepasst werden können
- Essenziell ist die `clone()` Methode

10.3 Flyweight

- Eher eine Notlösung
- Wenn zu viele kleine Objektinstanzen vorhanden (Speicherprobleme)
- Lösung mittels Sharing: Objekt `a` wird dann aber mehrmals verwendet
- Wird aber komplexer

10.4 Proxy

- Klasse, welche Anfragen bei Bedarf weiterleitet

10.5 Chain of Responsibility

10.6 Interpreter

- Grammatik abbilden
- Rekursion zur Auswertung (zB zuerst linken ausdrucke und danach rechten ausdruck auswerten, was rekursiv geschechen kann)

10.7 Mediator

- Vermittler: Kapselt Kommunikation mit Widgets
- Vorteil: Entkopplung er Kollegen, Einfacheres Protokoll 1-zu-N statt N-zu-N
- Nachteil: Vermittler hat zu grosse Verantwortung und Logik

10.8 Memento

- Mommentaufnahme merken (internen Zustand des Urhebers)
- `createMemento()` Backup
- `setMemento(Memento m)` Zurücksetzen
- Prüfungsfrage wer welche Rolle übernimmt

10.9 Visitor

11 Produktmetriken

- Metrik: Messwerte/Kennzahlen
- McCabe Metrik/Zahl: Misst Komplexität der logischen Struktur
  - Zählt Anzahl Wege durch Methode
  - Graphen zeichnen
  - Zu einem If ein Else hinzufügen ändert die Zahl nicht
  - V(G) = e - n + 2 // e = edges, n = knoten
  - → Alle möglichen Wege einmal nach unten
  - Einfacher: Anzahl if (ohne else!!!) + while + 1
  - Unsicher ob do..while welches zwingend durchlaufen wird die Zahl erhöht
  - If-Else: kein unterschied ob geschachtelt oder hintereinander
  - Falls Grösser 10: Methode genauer anschauen ob zu vereinfachen
- Metrics Plugin für Eclipse
- Zones of Exclusion
  - Y-Achse: Abstrakt/Konkret; Abstrakt = Abhängigkeiten nach Aussen
    - Oben: Reines Interface
  - X-Achse:
  - Oben Links: Interface welches keine Abhängigkeiten nach aussen hat, sondern oft implementiert wird.
  - Oben Rechts: Interface, welche nicht implementiert werden
  - Zone of Pain: Wenn was geändert wird, hat man schnell ein Problem
  - Zone of Uselessnes: Interface welches nicht gebraucht wird
  - Am besten aber schwiering zum Refactoren: links unten

11.1 Metriken für Objektorientierte Software

- Umfangsmetriken, Logische Strukturmetriken, Metriken für Kohäsion und Kopplung
- NOC - Number of Classes:
- NSC - Number of Children: Anzahl direkter Unterlassen und Interfacec
- NOI - Number of Interfaces:
- DIT - Depth of Inheritance Tree
- NORM - Number of Overridden Methods
- NOM - Number of Methods
- NOF - Number of Fields
- TLOC - Total Lines of Code
- MLOC - Method Lines of Code
- Specialization Index:
  - Mittelwert von NORM * DIT / NOM
  - Metrik auf Klassenebene

-_ LCOM* - Lack of Cohesion of Methods (LCOM*): -_ LCOM* = (m - Mittelwert (m(A))) / (m – 1)

11.2 Software

- Metrics
- STAN
- Checkstyle
- Findbugs

11.3 Dynamische Analyse

12 Design By Contract (DbC)

- Rechte und Pflichten
- Kunden/Lieferanten-Beziehung zwischen Komponenten
- Contracts für Systemoperationen
  - Preconditions: Was gilt vor Ausführung der Systemoperation?
  - Postconditions: Was gilt nach Ausführung der Systemoperation?
- Contract für Klasse
  - Preconditions:_ Bedingungen, die vor dem Aufruf erfüllt sein müssen, _ Verantwortlich ist der Aufrufer
  - Postconditions:_ Bedingungen, die nach dem Aufruf gültig sind_ Verantwortlich ist die Implementation der Methode
- Class Invariant
- In Java mit `icontract`: Spezielle Kommentare für die Bedingungen des Contract
- Die Verantwortung muss im Vertrag geretelt sein
- Contracts mit `C4J` schreiben. Wird zur Laufzeit geprüft.
- Inputvalidierung: Eingabe vom User =! Contractbedingungen
- An der Systemgrenze Testen
- Query-Methoden verändert den Zustand nicht
- Postconditions bei Query-Methoden machen sinn, wenn sie eine Aussage über den Rückgabewert der Query-Methode machen.
- Invarianten: Gelten immer

- Contracts einer Subklasse müssen den Vertrag einer Superklasse erfüllen
  - Preconditions dürfen gelockert werden,a ber nicht verschärft
  - Postconditions genau umgekehrt!
- C4J: @Pure für Query-Only Methoden

13 Refactoring

13.1 Understanding Refactoring

- A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- Rewriting: Von Grund auf neu schreiben
- Keine Logik verändern
- Ziele

13.2 Refactorings

```
 - Clarity
- Douplication Free

- Hight Cohesion
```

- Hide Method: public -> private
- Inline Method: Code von einer Methode zum Aufrufer verschieben
- Extract Method: Codeteile in eigene Methode auslagern
-

14 Faking und Mocking

14.1 Testing unter heiklen Situationen

- Ruft ein Objekt eine Methode von einem anderen Objekt auf, kann es zu ungewöhnlicher oder zeitintensiver Kommunikation kommen.
- Zeitabhängigkeiten, Zufälligkeit und kaputte Mitspieler führen zu heiklen Situationen.
- Zuviele Abhängigkeiten (auch transitive) machen das Testen unübersichtlich und langsam, worauf Entwicker das Testen meiden.
- Beispiele: Datenbankzugriff, spezielle Hardware, Druckjobs, Filesystemzugriffe, Systeme von Drittanbietern, Logging-Systeme, Auszahlung am Geldautomaten ;-), ...

14.2 Faking

14.3 ## Begriffe

- *Inversion of Control (IOC)*: Statt Klasse A die Klasse B kontroliert, geht es umgekehrt (CLI Tool fragt User vs. GUI Tool wird von User kontrolliert)
- *Dependency Injection (DI)*: DI ist eine Form von IOC. Einer Klasse wird ein Objekt übergeben, statt selber eines zu erzeugen. (`provider =  Provider.getProvider();`)

- *Slipping a Fake (SF)*: Slipping a Fake ist eine form von Dependency Injection, mit dem einzigen Zweck Tests zu ermöglichen (Methode in ein Interface extrahieren und überschreiben um ein Fake zurückzugeben). Dies ist eine Art von Dependency Injection (`provider = getProvider();`)

14.4 ## Slip a Fake

- Grundprinzip: Die zu testende Klasse redet mit einem Fake Objekt um den Test eiinfacher zu machen (Slipping heist das Einfügen der Fake-Klasse in die Testklasse).
- Typen: Listening Fake, Talking Fake, Mock, Auto-Mock und Simulatoren; Egal wie man sagt, man soll immer gleich sagen.
- Stub: Leichtgewichtiges Fake um durch einen Test zu kommen (kann sogar eine private Klasse in der Testklasse sein).
- Listening Fake: Nimmt Daten entgegen und macht sie für den Test verfügbar (`public` Variabeln).
- Talking Fake: Falls der Test Daten braucht, stellt der talking Fake präparierte Daten zur Verfügung, damit z. B. kein externes System abgefragt werden muss.
- Können Fakes nicht einfach in Tests eingefügt werden, muss man ein Refactoring durchführen.
- Singletons und globale Variabeln können schlecht gefaked und getestet werden.

14.5 ## Faking Techniken

- Nimmt die zu testende Methode den Mitspieler als Argument, kann man einfach das Fake übergeben (oder auch direkt beim Instanzieren eines Objekts im Konstruktor).
- Instanziert die Testmethode den Mitspieler, extrahiert man die Methode und überschreibt diese, damit man dieser Methode den Fake unterschieben kann.
- Man erstellt eine Factory Klasse, welche eine Instanz eines Fake-Mitspielers zurückgibt.
  - `FakeOrderItem` erbt von `OrderItem`
  - `RealItem` und `FakeItem` implementieren Interface `Item`
- Hat man kein Zugriff auf den Code, macht man ein `Introduce Adapter`: Interface (Adapter) erstellen mit einer Implementierung die den richtigen Code (`ProductionAdapter`) ausführt und eine Implementierung mit einem Fake (`FakeAdapter`), welche jeweils unterschiedliche Methoden aufrufen.

Beispiel: Constructor Argument Slip

```
public class Foo {
  private Bar bar;
  Foo(){ // This is the productive bar
    bar = new Bar();
  }
  Foo(Bar bar){ // This is the fake bar
    this.bar = bar;
  }
}
```

14.6 ## Extract and Override

Typischer Ablauf:

14.7 . Heikle Teile in separate `protected` Methoden packen

14.8 . Diese Methode in einer abgeleiteten Test-Klasse überschreiben

14.9 . Diese abgeleitete Klasse testen

Pros/Cons:

- Nachteil: Vermeiden wenn: Testklasse nicht effektiv wiederverwendet werden kann; Tests an den Code gebunden sind
- Vorteil: Kann immer verwendet werden

Beispiel:

```java
public class Foo {
  public void Bar(){
    int blah = fnord();
  }
  protected int fnord(String s){
    doStuff(); // Do Stuff...
    return doOtherStuff();
  }
}
public class FooTest {
  class FakeFoo extends Foo {
    protected int fnord(String s){
      return 23; // Now no complex operations
    }
  }
  @Test
  public void testFnord(){
    Foo f = new FakeFoo();
    // Do Stuff
    assertEquals(23, foo.fnord()); // Much faster than before!
  }
}
```

14.10 Mocking

14.11 ## Begriffe

- Man soll keinen Code duplizieren, damit man Tests schreiben kann.
- Ein Mock ist ein Fake, mit eingebauten Tests.
- Mit Mocks kann man die Interaktion zwischen Objekten testen.
- Man testet jedoch nicht die richtigen Sachen

14.12 ## Mocks

```java
class FakeFoo extends Foo {
  public void assertFnord(){ // ...
    assertEquals(...);
  }
}
class FooTest {
```

```
  Foo foo = new FakeFoo();
  foo.assertFnord(); // Tests werden innerhalb von FakeFoo ausgeführt
}
```

14.13 ## Auto-Mocks

Ablauf:

14.14 . Implementiere ein Interface oder erbe von einer Klasse.

14.15 . Schreibe für jede interessante Funktion ein Body.

14.16 . Packe alle übergebenen Argumente in eine Collection.

14.17 . Die Methode gibt Resultate von einer weiteren Collection zurück.

Beispiel:

```
class Foo {
  public String bar(Parameter p){
    return baz;
  }
}

class FakeFoo extends Foo {
  ArrayListet barParameters = new ArrayList();
  ArrayListet barResults = new ArrayList();
  int barIndex = 0;
  public String bar(Parameter p){
    barParameters.add(p);
    return barResults.get(barIndex++);
  }
}

class FooTest {
  FakeFoo ff = new FakeFoo();
  @Test
  public void snafu(){
    ff.fooResults.add(x);
    otherFunction(ff); // ff nutzen
    assertEquals(, ff.fooParameters.size());
    assertEquals(new Parameter("right one"), ff.fooParameters.get(0));
  }
}
```

14.18 ## Mocking-Tools

- Braucht etwas, bis man sich an die Tools gewöhnt hat.
- Für Anfänger nicht empfohlen.
- Einfache Fakes reichen meist aus.

14.19 Fake Simulators

- Simulatoren brauchen viel Zeit um sie zu entwickeln.
- Beispiel: Die aktuelle Software nutzt nur einen Simulator, während andere Entwickler die nötige Hardware zur Software entwickelt.

15 Prüfungsvorbereitung

- Letzte Prüfung Titelblatt
  - Memento, Proxy, Architektur, aufwandschätzung, dbc, review, fitnesse, faking and mocking, ppractices, advanced code design, tdd, code smells, refactoring, metrics,