




Unit Tests



Was ist ein Unit Test?

- Test einer (Teil-)Funktionalität
 - Genauer: Test eines Szenarios für die Ausführung einer Teilfunktionalität
- 



Was ist ein Unit Test?

- Tests von
 - Funktionen
 - Methoden
 - (Klassen)
 - (Einfachen Komponenten)
- Feingranular
 - z.B. Test eines Methodenaufrufs mit gewissen Parametern
- Keine Interaktion zwischen getesteter Einheit und dem Rest des Systems
 - Isolierte Testfälle
- Externe Subsysteme werden oft durch einfache Simulationen ersetzt (Mocks)



Gründe für das Unit Testen

- Frühes Finden von Fehlern
- Einfachere Lokalisierung von Fehlern
- „Sicherheitsnetz“ für Programmierer
 - Fungieren auch als Regressionstests
 - Fehler werden (hoffentlich) gefunden, bevor sie das Gesamtsystem beeinflussen
- Unterstützen Wartbarkeit und Erweiterbarkeit von Code
- Erleichtern Refactoring, da viele dabei auftretende Fehler durch Unit Tests gefunden werden
- Stellen sicher, dass Erweiterungen die existierende Funktionalität nicht beeinträchtigen
- Dienen als zusätzliche Dokumentation
- Können aber Architektordiagramme etc. nicht ersetzen



Eigenschaften von guten Unit Tests

- Unit Tests sollen
 - automatisiert sein
 - selbsttestend sein
 - einzelne Programmelemente isoliert testen
 - zu jedem Zeitpunkt erfolgreich ausführbar sein
 - nicht viel Zeit zur Ausführung benötigen
 - für alle Systembestandteile geschrieben werden
 - alle wichtigen Zustände jedes getesteten Elements abdecken
- Später: detailliertere Richtlinien - FIRST, Right-BICEP, CORRECT



Wie schreibt man gute Unit Tests (Teil 1)

Versuch: Erschöpfendes Testen

- Wir schreiben erschöpfende Tests, d.h. Tests, die alle möglichen Eingaben eines Programms abdecken
- Erschöpfendes Testen ist nicht möglich
- Beispiel Passworteingabe:
 - Angenommen, Passwörter mit maximal 20 Zeichen sind zulässig, 80 Eingabezeichen sind erlaubt (große und kleine Buchstaben, Sonderzeichen)
 - Das ergibt $80^{20} = 115.292.150.460.684.697.600.000.000.000.000.000$ mögliche Eingaben
 - Bei 10ns für einen Test würde man ca. 10^{24} Jahre brauchen, um alle Eingaben zu testen
 - Das Universum ist ungefähr 1.4×10^{10} Jahre alt



Effektivität und Effizienz von Tests

- Unit Tests sollen **effektiv** und **effizient** sein
 - Effektiv: Die Tests sollen so viele Fehler wie möglich finden
 - Effizient: Wir wollen die größte Anzahl an Fehlern mit der geringsten Anzahl an möglichst einfachen Tests finden
- Effizienz ist wichtig, da Tests selbst Code sind, der gewartet werden muss und Fehler enthalten kann



Strategien zum Finden von (effektiven und effizienten) Tests

- Analyse von Randwerten (Boundary Value Analysis, BVA)
- Partitionierung
- Zustandsbasiertes Testen
- Kontrollflussbasiertes Testen
- Richtlinien
- Kenntnis häufiger Fehler in Software
- Kenntnis häufiger Probleme von Tests (Test Smells)

- Werden später besprochen. Zuerst einige „Faustregeln“

Welche Form hat ein Unit Test?

- Arrange
- Act
- Assert
- Given
- When
- Then

```
@Test
void ExampleTest() {
    Screen unit = new Screen(80, 25);
    String output = "Example Output";

    unit.writeText(output);

    assertEquals(output, unit.getText());
}
```



„Faustregeln“ für Unit-Tests

- Teste Funktionalität, nicht Implementierung
- Bevorzuge Tests von Werten gegenüber Tests von Zuständen
- Bevorzuge Tests von Zuständen gegenüber Tests von Verhalten
- Teste kleine Einheiten
- Verwende Test-Doubles (dann, aber auch nur dann) wenn eine Abhängigkeit eine Rakete abfeuert („if it launches a missile“)
 - Zugriff auf Datenbank, Dateisystem
 - Zeit, Zufallswerte
 - Nichtdeterminismus
- (Diese Regeln setzen voraus, dass der Code solche Tests erlaubt)



Teste Funktionalität, nicht Implementierung

- Abstrahiere so weit wie möglich von Implementierungsdetails
 - Auch auf Unit-Test Ebene
 - Dies erfordert oft die Einführung von Methoden, die Invarianten überprüfen
- Warum?
 - Funktionalität ist leichter zu verstehen
 - Funktionalität ist stabiler als Implementierung
 - Funktionalität entspricht eher dem Kundennutzen



Werte > Zustand > Verhalten

- Verständlicher
- Leichter zu testen
- Oft stabiler gegenüber Refactorings
- Ausnahme: Testen von Protokollen



Teste kleine Einheiten (bei Unit-Tests)

- Test von kleinen Einheiten
 - spezifizieren das Verhalten der getesteten Einheit besser
 - erleichtern die Lokalisierung von Fehlern
 - sind leichter zu pflegen
- Tests größerer Einheiten oder des Gesamtsystems sind wichtig als
 - Integrationstests
 - Systemtests
 - Akzeptanztests



Test Doubles

- Test Doubles: Stubs, Fakes, Spies, Mocks
- Ersetzen eine Abhängigkeit im System durch eine vereinfachte Version
 - z.B. Ersetzen einer Datenbankabfrage durch einen fixen Wert
- Test Doubles sind wichtig zum Vereinfachen von Tests
- Aber: zu viele oder komplexe Test Doubles machen Tests unübersichtlich
 - Was wird von einem Test eigentlich getestet?
- Typischer Einsatz von Test Doubles:
 - Zugriff auf Datenbank, Dateisystem
 - Zeit, Zufallswerte
 - Nichtdeterminismus



Wie schreibt man testbaren Code?

- Keine globalen oder statischen Daten
- Techniken aus der funktionalen Programmierung (Streams, Lambdas, etc.)
- Funktionale Datenstrukturen (Immutability)
- Gutes objektorientiertes Design
 - Hohe Kohärenz
 - Geringe Kopplung, Management von Abhängigkeiten
- Etc.
- Hilfsmittel: Test-Driven Development



Test-Driven Development



Idee

Verwende Tests, um das **Design** und die **Feature-Entwicklung** des Programms voranzutreiben

Jeder neue Test beschreibt ein Feature-Inkrement
des Programms

(Gut testbarer Code entsteht dabei quasi als
Nebenprodukt)



Problem

Wie können Tests das Design des Programms
vorantreiben?





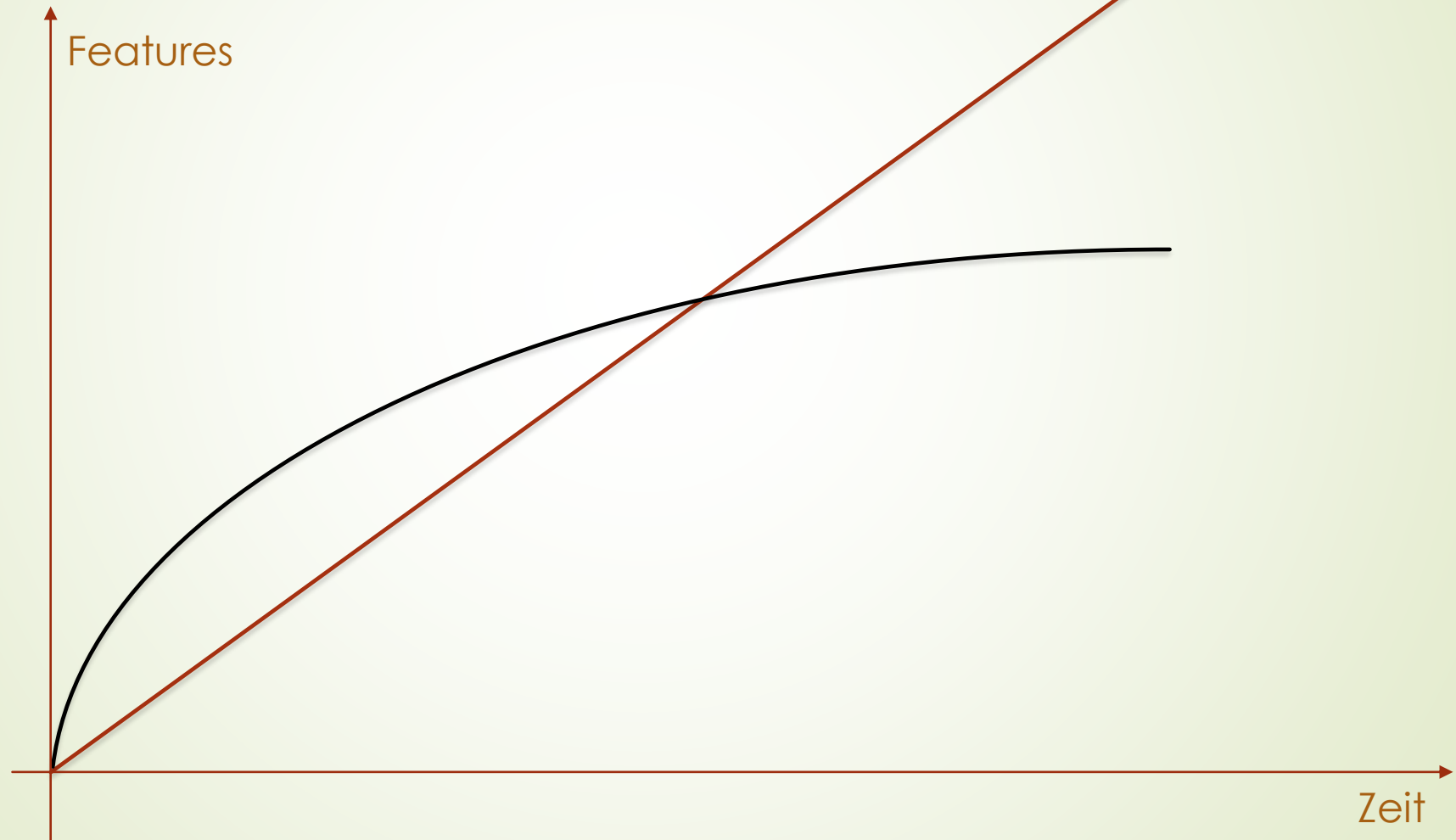
Refactoring

Durch Refactoring wird das Design des Programms
in kleinen Schritten verbessert

Die Korrektheit dieser Schritte wird durch Tests
abgesichert



So what???





Test-Driven Development

- Ziel beim TDD ist nicht in erster Linie, eine hohe Testabdeckung zu erreichen
 - Typischerweise schreibt man keine Tests für Methoden, von denen man überzeugt ist, dass sie nicht fehlschlagen können
- Ziel beim TDD ist es, durch Tests ein gutes Design zu entdecken
 - Beim Schreiben der Tests versucht man, das Interface der Klasse so zu gestalten, dass es leicht zu benutzen ist
 - Dadurch, dass alle wesentlichen Teile des Programms durch Tests abgesichert sind, kann man das Design durch Refactoring permanent an das aktuelle Feature-Set anpassen




Der TDD-Zyklus

- Schreibe einen (minimalen) Test
 - Der Test testet nur ein einziges neues (Teil-)Feature: **Baby Steps**
 - Dieser Test schlägt fehl
- Implementiere die minimale Funktionalität, um den Test zum Laufen zu bringen
 - Dabei muss man nicht auf sauberen Code oder gutes Design achten
 - Aber: **Solve Simply**
- Verbessere den Code
 - Entferne die unsauberen Konstrukte, die im vorhergehenden Schritt eingefügt wurden
 - Generalisiere die Implementierung, wenn zu viel Wiederholung entstanden ist
 - **Dieser Schritt ist nicht optional!!!**



Der TDD-Zyklus

- **Red** (fehlschlagender Test)
 - **Green** (alle Tests sind wieder grün)
 - **Clean/Refactor** (der Code ist wieder sauber)
- 



Noch besser: TDD + Vorbereitungsschritt

- *Refactore den Code, sodass die Änderung einfach wird*
 - Das ist oft nicht so einfach...
 - Wenn beim Refactoring klar wird, dass Tests fehlen, so werden diese hinzugefügt
- Führe die einfache Änderung mit dem TDD-Zyklus durch
- Wiederhole diese Schritte immer wieder




Warum Solve Simply?

- Eine flexible, generische Lösung erhöht oft die Komplexität des Systems
 - Das lohnt sich nur, wenn die Flexibilität auch benötigt wird
- Entwickler können meist schlecht vorhersehen, an welchen Stellen Flexibilität/Erweiterbarkeit benötigt wird
- Eine flexible, generische Lösung ist oft sehr viel schwerer zu implementieren als eine einfache Lösung für einen spezielleren Anwendungsfall
- Die naheliegendste flexible, generische Lösung ist oft nicht der sauberste und wartbarste Code



Annahmen von Solve Simply

- Es ist durch Refactoring möglich, Code in einen sauberen, wartbaren Zustand zu bekommen, ohne dadurch die Funktionalität zu verändern
- Es ist möglich, Code iterativ zu erweitern und flexibler zu machen, wenn das erforderlich ist
- Es ist einfacher, die Refactoring- und Iterations-Schritte durchzuführen, als gleich die endgültige Lösung zu entwickeln
- Diese Annahmen sind nur dann erfüllt, wenn hinreichend viele, gute Unit-Tests vorliegen

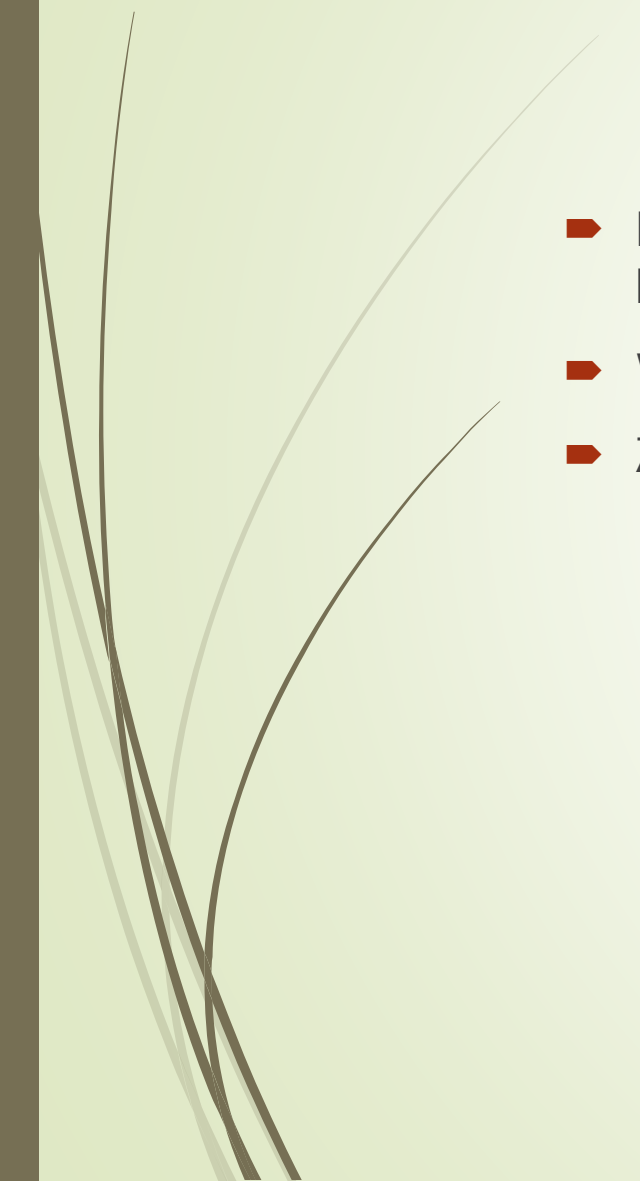


Baby-Steps

- Das System ist nicht stunden- oder tagelang in einem Zustand, in dem es nicht baubar, testbar oder ausführbar ist
- Dadurch bekommt man bei jeder Änderung schnell Feedback vom Code
- Häufiges Mergen und CI wird möglich



Geleitetes Kata: Primfaktorzerlegung

- Eine Übung zu TDD, die zeigt, wie man durch Tests auf eine einfache Implementierung eines Algorithmus geführt werden kann
 - Wichtig ist die Vorgehensweise: Tests sollen das Design treiben
 - Ziel: Lernen inkrementell und iterativ zu arbeiten!
- 

Geleitetes Kata: Primfaktorzerlegung

- Schreiben Sie eine Klasse `PrimeFactors`, die eine statische Methode `generate` hat

- `generate` hat die Signatur

`List<Integer> generate(int n)`

und gibt die Primfaktoren von `n` in aufsteigender Reihenfolge zurück

- Mehrfach vorkommende Primfaktoren sind in der Liste mehrmals enthalten



Workshop

Geleitetes Kata: Primfaktorzerlegung



Kata: FizzBuzz

- Erstellen eines Programms, das ein einfaches Spiel für Kinder implementiert
- Inkrementelles Vorgehen, kleine Schritte!
- Das Programm gibt die Liste der Zahlen von 1 bis 100 aus
- Jede durch 3 teilbare Zahl wird durch „Fizz“ ersetzt
- Jede durch 5 teilbare Zahl wird durch „Buzz“ ersetzt
- Jede durch 3 und 5 teilbare Zahl wird durch „Fizz Buzz“ ersetzt



Workshop

Kata: FizzBuzz



Strategien zum Finden von Tests

- Analyse von Randwerten (Boundary Value Analysis, BVA)
- Partitionierung
- Zustandsbasiertes Testen
- Kontrollflussbasiertes Testen
- Richtlinien
- Kenntnis häufiger Fehler



Boundary Value Analysis

- Viele Fehler treten am „Rand“ des Definitionsbereichs von Funktionen oder Prozeduren auf
- Eine gute Strategie zum effizienten Testen ist es daher, derartige Randwerte zu betrachten
 - Der/die letzten gültigen Werte
 - Werte, die gerade außerhalb des Definitionsbereichs liegen
- Ist z.B. eine Funktion für ganzzahlige Werte zwischen 0 und 5 definiert, so kann sie mit Eingaben -1, 0, 5, 6 getestet werden




Boundary Value Analysis

Vorteil:

- Man konzentriert sich auf empirisch häufige Fehlerquellen

Schwierigkeiten:

- Bei vielen Bereichen ist nicht klar, was „Randwerte“ sind
 - (Allerdings lassen sich oft alternative Kriterien finden, z.B. Länge von Collection-Argumenten)
- Werte außerhalb des Definitionsbereichs können manchmal zu undefiniertem Verhalten führen
- Bei Funktionen mit vielen Parametern gibt es eine kombinatorische Explosion von Randwerten



Merkregel für (erweiterte) BVA: CORRECT

- **Conformance:** Entspricht der Wert einem erwarteten Format?
- **Ordering:** Sind die möglichen Werte geordnet oder ungeordnet?
- **Range:** Hat der Wert einen minimalen und/oder maximalen Wert?
- **Reference:** Hat der Code externe Referenzen, die nicht unter seiner Kontrolle sind?
- **Exist:** Existiert der Wert (ist er nicht null, in einer vorgegebenen Menge enthalten, ...)
- **Cardinality:** Sind genug Werte vorhanden? Sind zu viele Werte vorhanden?
- **Time:** Sind die Werte zum benötigten Zeitpunkt verfügbar? In der erwarteten Reihenfolge?



Partitionierung



- Argumente von Funktionen, Ein/Ausgabe des Programms und Zustände von Klassen können oft in Äquivalenzklassen unterteilt werden, sodass...
 - Das Verhalten für Elemente aus der gleichen Äquivalenzklasse ähnlich ist (z.B. den gleichen Kontrollflusspfad nimmt)
 - Elemente aus unterschiedlichen Klassen verschiedenes Verhalten zeigen
 - Beispiel: Die Argumente der sqrt-Funktion können unterteilt werden in
 - Positive Zahlen und 0
 - Negative Zahlen
 - Eine feinere Unterteilung wäre zusätzlich in Quadratzahlen und Nicht-Quadratzahlen
- Eine derartige Äquivalenzklasse heißt Partition (oder Domäne)



Partitionierung

Finde Partitionen für das getestete Element und teste die folgenden Elemente:

- Einen Wert aus der „Mitte“ der Partition
- Einen Wert auf oder nahe jeder Partitionsgrenze

Häufig findet man Partitionen durch BVA.

Beispiel: Um die Quadratwurzelfunktion zu testen, schreibe Tests für:

- `sqrt(0.0)`
- `sqrt(2.0)`
- `sqrt(-2.0)`



Zustandsbasiertes Testen

Kann man das Verhalten eines Objekts durch ein Zustandsdiagramm beschreiben, so kann man sich beim Testen an den Zuständen und Transitionen orientieren

- Ein zustandsbasierter Test wird durch eine Folge von Events beschrieben, die die Zustandsmaschine steuern
- Die erwarteten Ergebnisse sind
 - die Zustände (falls beobachtbar) und
 - die Aktivitäten bzw. Ausgaben die durch die Eingabe-Events verursacht werden
- Es gibt verschiedene Methoden, um
 - fehlerhafte Aktivitäten bzw. Ausgaben
 - falsche Zustandsübergängezu finden (z.B. Transition Tour, Distinguishing Sequence)



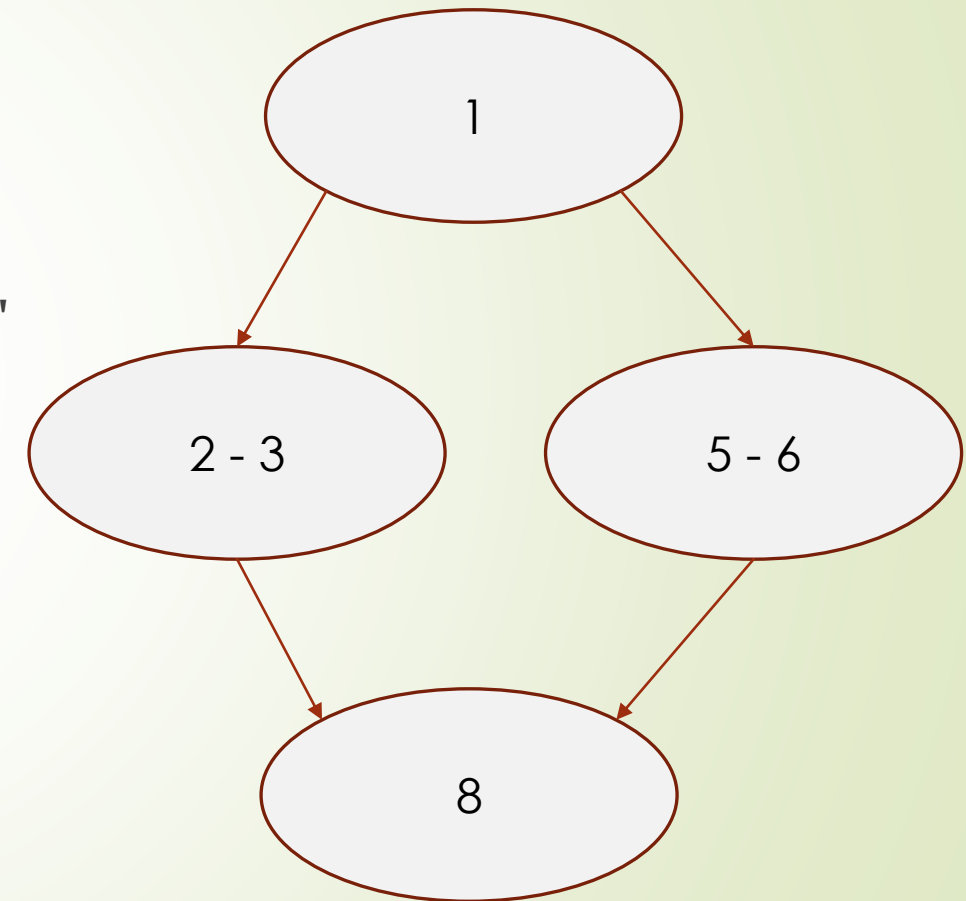
Kontrollflussbasiertes Testen



- Der Kontrollfluss-Graph (CFG) eines Programms ist ein gerichteter Graph, der die Kontrollstruktur eines Programms repräsentiert
 - Knoten sind Basic Blocks (lineare Folgen von Anweisungen)
 - Kanten repräsentieren mögliche Programmmabläufe
- Fallunterscheidungen: Knoten mit mehreren Nachfolgern
- Schleifen im Programm führen zu Schleifen im CFG

Kontrollflussbasiertes Testen

```
1  if (x == 0) {  
2      print "x == 0"  
3      print "This is interesting."  
4  } else {  
5      print "x != 0"  
6      print "The boring case."  
7  }  
8  print "Hello, world!"
```



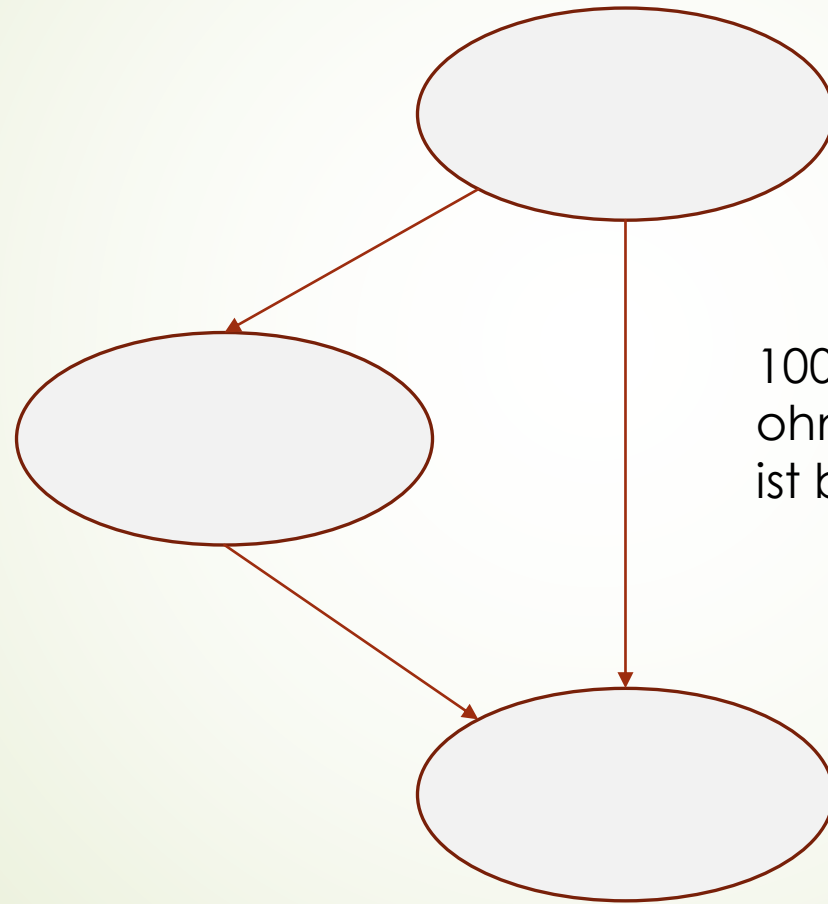


Kontrollflussbasiertes Testen

Mit dem CFG kann man verschiedene Maße für die „Testabdeckung“ des Programms definieren

- Anweisungsüberdeckung (Statement Coverage): Der Prozentsatz an Programmanweisungen (Knoten im CFG), der durch die Tests abgedeckt wird. Eine Test Suite hat 100% Anweisungsüberdeckung wenn jede Programmanweisung durch mindestens einen Test abgedeckt wird
- Entscheidungsüberdeckung (Branch Coverage, Decision Coverage): Der Prozentsatz an Kanten im CFG, der von Tests abgedeckt wird. Eine Test Suite hat 100% Entscheidungsüberdeckung, wenn jede mögliche Verzweigung im Programm von einem Test abgedeckt wird, d.h., wenn jede Kante im CFG von einem Test durchlaufen wird
- 100% Entscheidungsüberdeckung impliziert 100% Anweisungsüberdeckung, aber nicht umgekehrt

Kontrollflussbasiertes Testen



100% Anweisungsüberdeckung
ohne 100% Entscheidungsüberdeckung
ist bei diesem CFG möglich!



Richtlinien

- Repräsentieren projekt- oder domänenspezifisches Wissen
 - Können erweitert werden, wenn Defekte gefunden werden, die nicht von den bisherigen Tests erfasst wurden
- 



Beispiele für Richtlinien

- Schreibe Unit Tests, die jede mögliche Fehlermeldung triggern
- Teste jede Funktion, die einen Buffer verwendet mit einer Eingabe, die größer als die maximale Buffergröße ist
- Teste gecachte Funktionen mehrmals mit der gleichen Eingabe und stelle sicher, dass sich die Ausgabe nicht ändert
- Teste jede Funktion mit Eingaben, die außerhalb des gültigen Definitionsbereichs liegen
- Teste jede Funktion, die eine Collection als Eingabeparameter hat, mit der leeren Collection und mit einelementigen Collections
- Verwende Collections verschiedener Größen in Tests
- Stelle sicher, dass auf Elemente von Anfang, Mitte und Ende einer Collection zugegriffen wird (falls möglich)

Einige „Häufige Fehler“

- Falsche Boole'sche Bedingungen
 - Betrachte Partitionen, die durch Bedingungen generiert werden
 - Betrachte die Randwerte von Bedingungen
- Nichtterminierende Berechnungen
 - Vergessene/unvollständige Basisfälle in rekursiven Funktionen (z.B. Test auf $= 0$ statt ≤ 0)
 - Unvorhergesehenes Inkrement/Dekrement des Zählers in for- oder while-Schleifen
- Falsche Vorbedingungen für Code
 - Schreibe Assertions im Code, die Vorbedingungen überprüfen
 - Schreibe Tests, die diese Assertions triggern
- Falsche Invarianten
 - **Schreibe Funktionen, die Invarianten testen, sofern diese auf nicht-zugreifbarem Zustand beruhen**
 - **Schreibe Tests, die diese Funktionen verwenden**



Einige „Häufige Fehler“

- Nullpointer, nicht-initialisierter Speicher
 - Im Allgemeinen schwer durch Tests zu finden; verwende Tools wie Valgrind und schalte Compilerwarnungen ein, falls verfügbar
 - Versuche Partitionen zu finden, die Werte uninitialized lassen
- Ungewöhnliche Bereiche
 - Leere oder einelementige Collections
 - Sehr kleine Werte (z.B. $1.0e-300$)
 - Sehr große Werte oder Collections
- Off-by-One, Zaunpfahl-Fehler (Fencepost Errors)
 - Teste, dass Schleifen nicht zu oft oder zu selten durchlaufen werden
 - Verwende for-in (falls möglich)



Einige „Häufige Fehler“

- Falsche Operatorpräzedenz
 - Überprüfe, dass Formeln auch die erwartete Bedeutung haben
 - Versuche Partitionen zu finden, die das sicherstellen
- Ungeeignete Algorithmen
 - Schlechte Laufzeiteigenschaften
 - Über-/Unterlauf für bestimmte Eingaben
- Ungeeignete Repräsentation von Daten
 - Gleitkommazahlen für Geldbeträge
 - Darstellung von Werten, die nur aus Ziffern bestehen können (z.B. Bankleitzahlen, Kontonummern, ISBN-Nummern) durch int-Werte (führende Nullen, Länge der Repräsentation)
- Zu wenige Werte für produktiven Einsatz, z.B. 8-bit User-ID



Einige „Häufige Fehler“ bei numerischen Berechnungen

- Verwendung inexakter Repräsentationen, wo eine exakte Repräsentation nötig wäre
 - z.B. Gleitkommazahlen für Brüche
- Verwendung von Gleitkommazahlen mit zu geringer Präzision
- Verwendung numerisch instabiler Algorithmen
- Durchführung numerischer Berechnungen ohne Rücksicht auf Ordnung der Operationen
 - Kann numerischen Fehler drastisch vergrößern



Umfang der Unit Tests

Unit Tests sollen die getestete Einheit möglichst komplett abdecken

- Funktionen/Methoden
 - Aufruf mit Parametern aus jeder (white-Box) Input-Partition
- Klassen/Objekte
 - Test aller relevanten Operationen
 - Mindestens ein Test für jede Zustandspartition
 - Tests für alle möglichen Zustandstransitionen
 - Tests, die sicherstellen, dass geerbte Attribute und Operationen wie gewünscht funktionieren

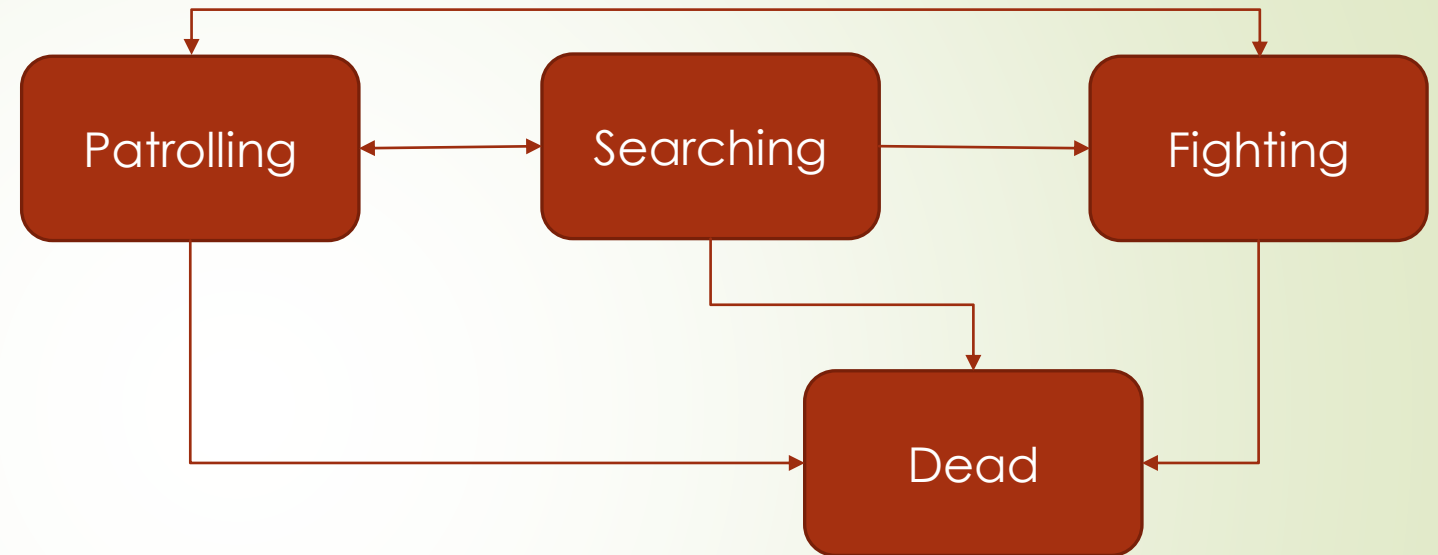
Beispiel: Wächter in einem Spiel

➤ Zustände:

- Patrolling
- Searching
- Fighting
- Dead

➤ Zustandsübergänge

- Patrolling ↔ Searching
- Patrolling ↔ Fighting („→“ nur, wenn er angegriffen wird)
- Searching → Fighting
- (Any) → Dead



Qualitätskriterien für Unit Tests: FIRST

► **Fast**

- Jeder Test benötigt höchstens einige Millisekunden Laufzeit

► **Isolated**

- Jeder Test testet eine geringe Menge an Code und ist vom Rest des Systems isoliert
- Tests sind voneinander isoliert, Reihenfolge der Ausführung spielt keine Rolle

► **Repeatable**

- Das Ergebnis eines Tests sollte immer identisch sein
- Häufige Probleme: Zeit, Datum, Threads, Random

► **Self-Validating**

- Es sollte keine externe Verifikation nötig sein (z.B. manuelle Inspektion der Testausgabe)

► **Timely**

- Unit Tests sollten zusammen mit dem Code, den sie testen, erstellt werden



Was bzw. wie soll getestet werden?

Right-BICEP

- **Right:** Sind die Ergebnisse/Ausgaben des Programms korrekt?
 - „Happy-Path Tests“ - Validierungstesten
- **Boundary Conditions:** Werden Randwerte korrekt behandelt?
- **Inverse Relationships:** Ist es sinnvoll, die inverse Beziehung zu testen? (Beispiel: Quadratwurzel)
- **Cross-Checking:** Können Ergebnisse durch „Gegenprobe“ verifiziert werden? (Beispiel: langsamer aber einfacher Algorithmus)
- **Forcing Error Conditions:** Können Fehlerbedingungen erzwungen werden?
 - „Unhappy-Path Tests“ – Defekttesten
- **Performance Characteristics:** Ist die Performance der Implementierung ausreichend? Skaliert die Implementierung?



Kata: Stack

- Erstellen einer einfachen Stack-Datenstruktur
 - Inkrementelles Vorgehen, kleine Schritte!
- 



Workshop Kata: Stack

Siehe PDF!

(Bitte blättern Sie nach jedem Schritt erst weiter, wenn Sie das jeweilige Feature implementiert haben)



Design:
K&K, HexA, SOLID, Grasp



K&K: Kohäsion und Kopplung

- Kohäsion: Ein Maß dafür, wie gut die verschiedenen Teile eines Moduls (einer Komponente, einer Klasse, etc.) zusammenarbeiten
 - Hohe Kohäsion: enge Zusammenarbeit
- Kopplung: Ein Maß für die Abhängigkeit zwischen verschiedenen Softwarekomponenten (Modulen, Klassen, etc.)
- Hohe Kohäsion und niedrige Kopplung sind erstrebenswert

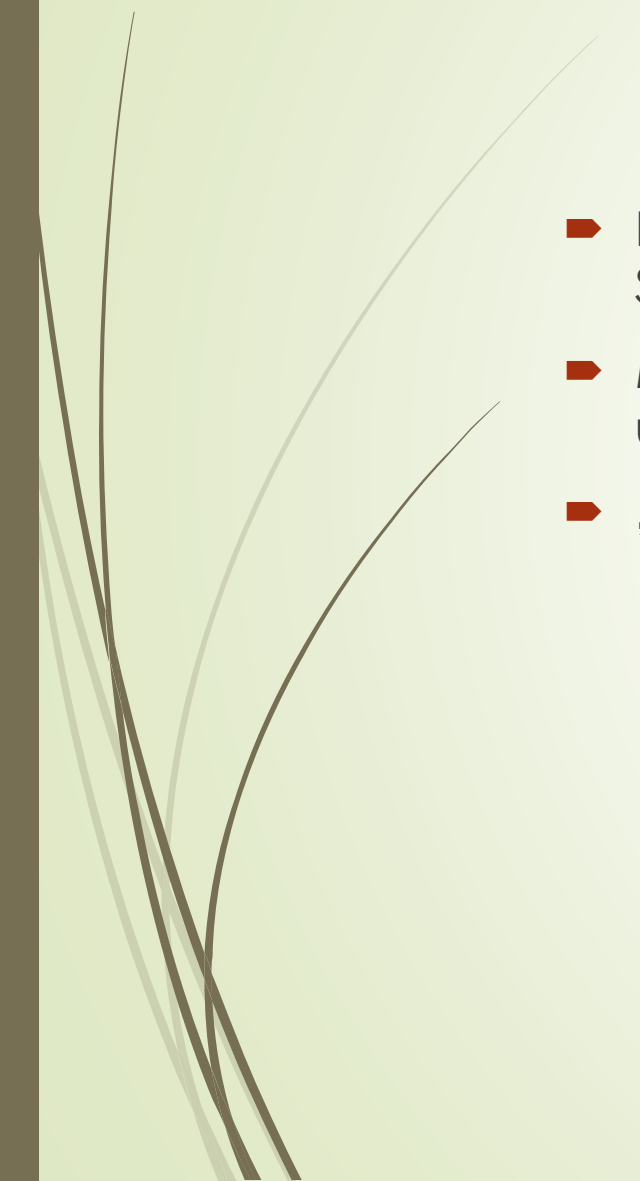


K&K: Kohäsion und Kopplung

- Mangelnde Kohäsion
 - macht es schwer, den Code zu verstehen, da man keine klaren Abstraktionen hat
 - macht es schwer, herauszufinden, wo Änderungen am Code vorgenommen werden müssen
- Hohe Kopplung
 - verhindert, dass man Teile des Systems in Isolation versteht
 - führt dazu, dass jede Änderung weitere Änderungen an großen Teilen der Codebasis nach sich zieht
- Verletzung der Architektur-Schichten führt zu hoher Kopplung und geringer Kohäsion
- Siehe Code Smells: Shotgun Surgery, Divergent Change
- **Mangelnde Kohäsion und enge Kopplung sind zwei der grundlegenden Probleme vieler Software-Systeme**
- **Eines der Hauptmittel dagegen ist bewusste Zuweisung von Verantwortlichkeiten an Klassen und Module (Responsibility-driven Design)**



Hexagonale Architektur

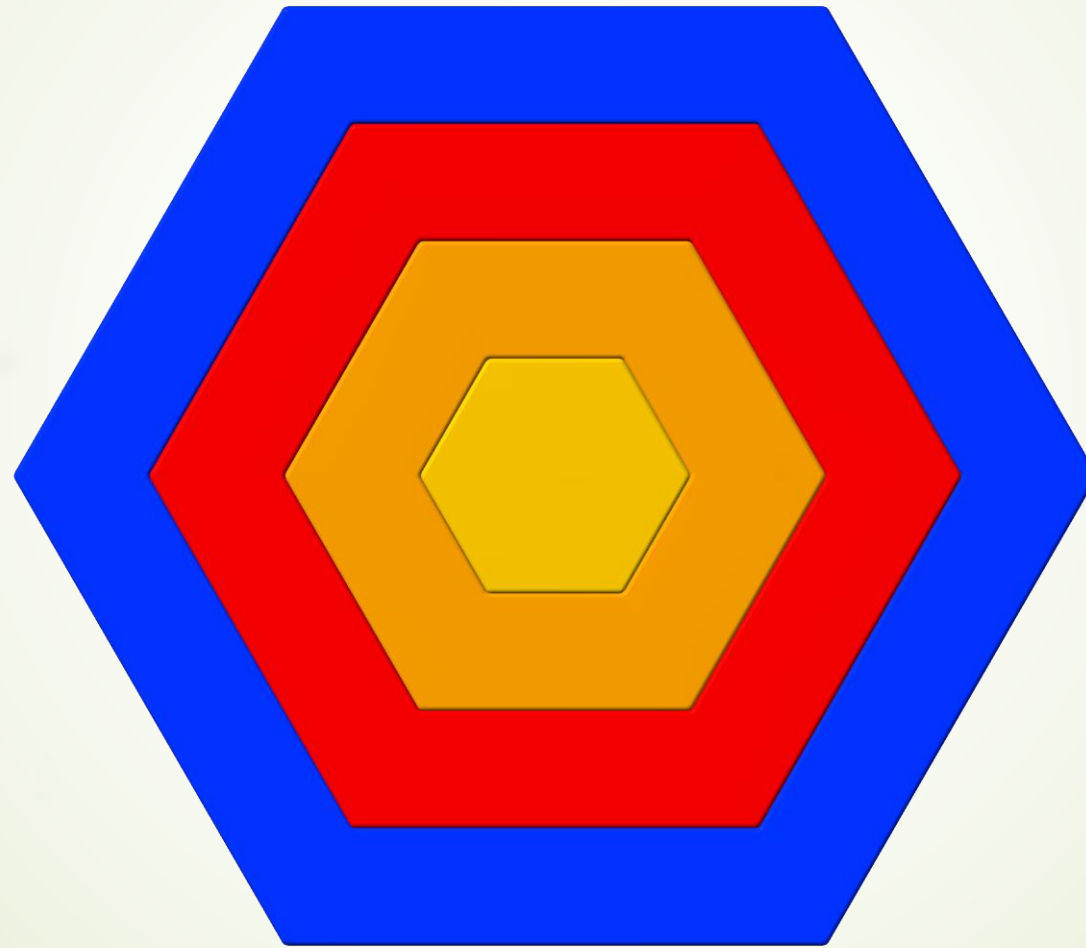
- Konzeptionelle Struktur von Abhängigkeiten und Verantwortlichkeiten in Software
 - Mechanismen, um verschiedene Abstraktionsebenen zu entkoppeln: Ports und Adapter
 - „Best Practices“, keine radikal andere Art, Software zu schreiben
- 



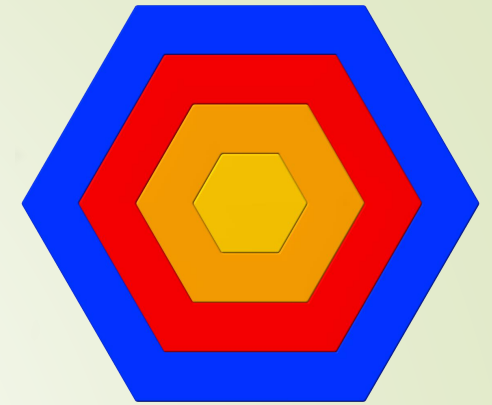
Vergleich mit 4-Schichten Architektur

- Vier-Schichten Architektur
- Bei Business-Anwendungen sind oft vier unterschiedliche „Schichten“ erkennbar:
 - Präsentationsschicht/View Layer
 - Anwendungsschicht/Application Model Layer
 - Domänenschicht/Domain Model Layer (Hier findet sich ein Großteil der Ergebnisse von OO-Design und Analyse)
 - Infrastrukturschicht/Infrastructure Layer
- Vorteile: viele!
- Probleme:
 - Wie werden Abhängigkeiten zwischen den Schichten geregelt?
 - Präsentationsschicht und Infrastrukturschicht sind oft ähnlich oder identisch (z.B. bei SOA)
 - Wo findet man Tests etc.?

Hexagonale Architektur

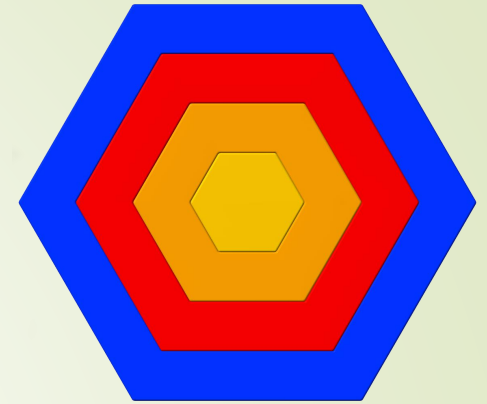


Hexagonale Architektur



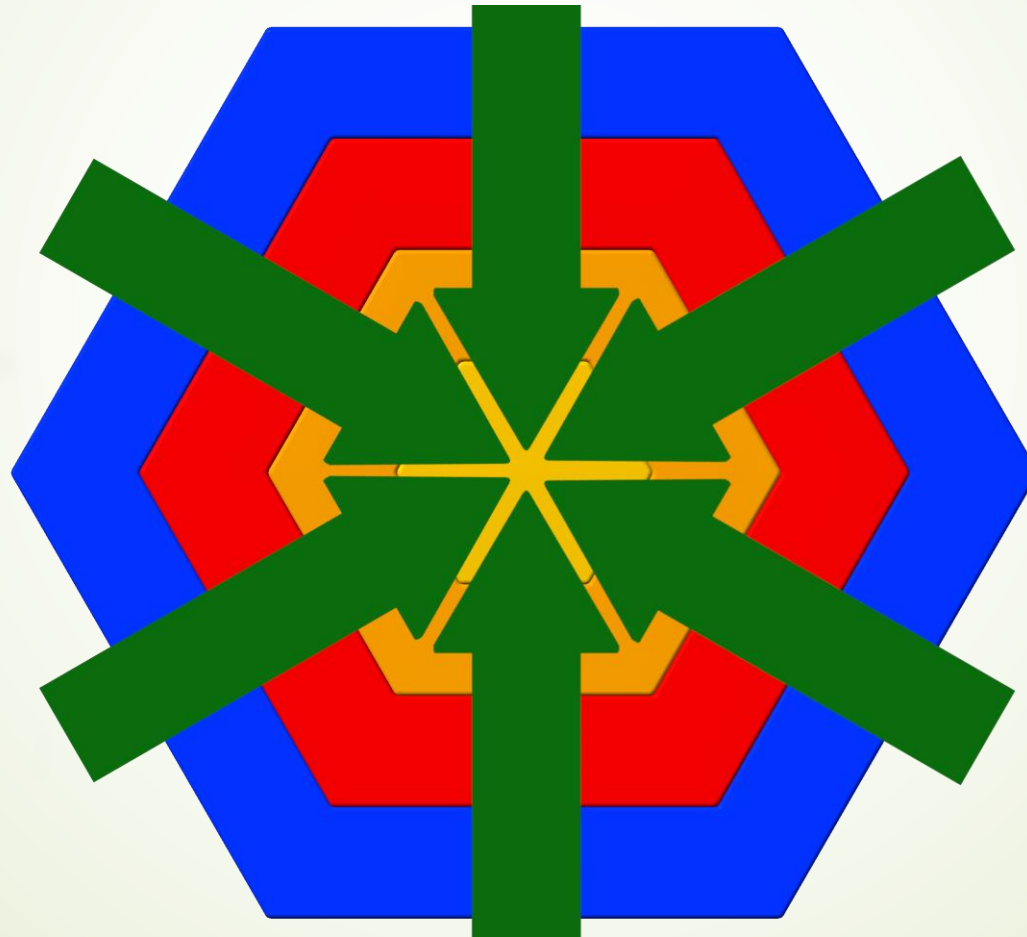
- Die hexagonalen Ringe stellen konzeptionelle Elemente eines Systems dar
- Ringe im Inneren sind abstrakter, nahe am Domänenmodell und zentral für die Funktion der Anwendung
- Die äußeren Ringe stellen die Interaktion des Systems mit der Außenwelt dar
- Die Seiten (Facetten) jedes Hexagons repräsentieren verschiedene „Konversationen,“ die der Ring mit weiter außen liegenden Schichten haben kann (d.h. Ports)
 - Ports sind thematisch ausgerichtet, nicht nach Technologie
- Die Gegenstücke zu Ports (d.h. die Objekte, die an der Grenze zwischen Ringen liegen) sind Adapter, die unterschiedliche Technologien in der äußeren Schicht *an die Bedürfnisse der inneren Schicht anpassen*

Hexagonale Architektur: Konsequenzen



- Der Kern der Anwendung ist von Änderungen der Infrastruktur oder Umgebung entkoppelt
 - Früher: Benachrichtigung von Anwendern durch Telefon, Fax
 - ... dann durch Mail,
 - ... dann per Tweet,
 - ???
- Die äußeren Ringe lassen sich leicht austauschen, z.B. zum Testen oder Automatisieren
- Die Abhängigkeitsstruktur der Anwendung ist sehr klar

Hexagonale Architektur: Abhängigkeiten





Konkrete Ratschläge: Grasp, SOLID

- Die bisher vorgestellten Prinzipien sind sehr allgemein
- Oft ist es hilfreich konkrete Regeln zu haben, die es erleichtern, Software zu schreiben, die...
 - hohe Kohäsion und geringe Kopplung aufweist,
 - dem Principle of Least Surprise folgt,
 - DRY ist,
 - usw.
- Beispiel für derartige Regeln sind Grasp, SOLID, etc.
- Einer der wesentlichsten Aspekte beim Objektdesign ist die Zuweisung von Zuständigkeiten an Klassen (assigning responsibilities to classes)



SOLID

Eine Sammlung von fünf Prinzipien/Patterns/Techniken, die von Robert Martin zusammengestellt wurde:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle



Single Responsibility Principle (SRP)

- Jede Klasse soll nur aus einem Grund geändert werden müssen (Single Reason to Change)
- (Der Name des Prinzips ist strenggenommen falsch; das SRP besagt nicht, dass eine Klasse nur eine einzige Verantwortung oder Funktionalität haben darf)
- Eng verwandt mit hoher Kohäsion und geringer Kopplung
- In den Grasp Regeln wird das Information Expert (IE) Pattern verwendet, um Zuständigkeiten (Responsibilities) an Klassen zuzuweisen
- Information Expert: Gib der Klasse die Zuständigkeit für eine Aufgabe, die die notwendigen Informationen besitzt, die Aufgabe wahrzunehmen
- IE ist oft leichter anzuwenden als SRP, führt aber zu anderen Ergebnissen



Open/Closed Principle

- Softwareartefakte (Klassen, Module, Funktionen, etc.) sollen *offen für Erweiterung aber geschlossen für Modifikationen* sein
- Einfacher ausgedrückt: Es soll möglich sein, das Verhalten von Objekten anzupassen oder neue Arten von Objekten einzuführen, ohne den existierenden Code anzufassen
- In Java bezieht sich das Prinzip hauptsächlich auf
 - das Implementieren gegen Interfaces nicht konkrete Klassen,
 - die Verwendung von Polymorphie statt if- oder switch-Anweisungen und
 - die Vermeidung von checked Exceptions
- Allgemeinere Formulierungen dieses Prinzips sind ‚Geschützte Variationen‘ (Protected Variations) und Information Hiding



Liskov Substitution Principle

- Instanzen von Oberklassen können ohne Probleme durch Instanzen einer Unterklasse ersetzt werden
- Das ist eines der grundlegenden Prinzipien beim Aufbau von Vererbungshierarchien!



Interface Segregation Principle

- Kein Client einer Klasse C soll von Methoden abhängen, die er nicht benutzt
- Wenn das nicht der Fall ist:
 - teile das Interface von C in mehrere unabhängige Interfaces auf
 - ersetze C in jedem Client durch die Interfaces, die der Client wirklich benutzt

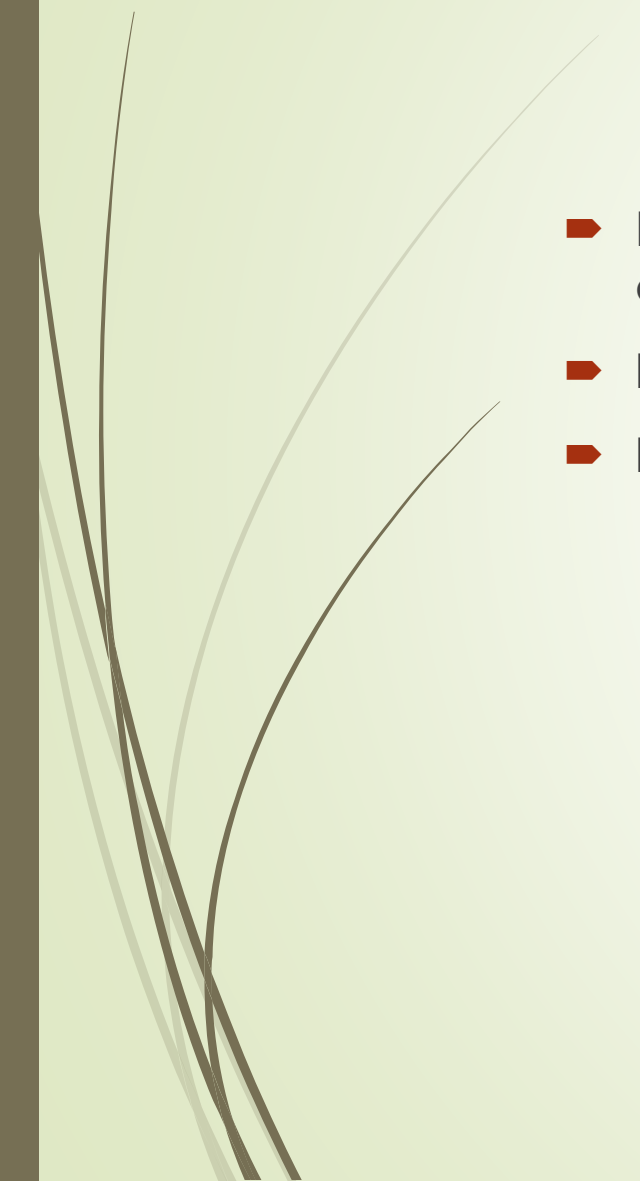


Dependency Inversion Principle

- Die grundlegende Funktionalität des Systems hängt nicht von „Umgebungsfaktoren“ ab (GUI, Datenbank)
- Konkrete Artefakte hängen von Abstraktionen ab, nicht umgekehrt
- Instabile Artefakte hängen von stabilen Artefakten ab, nicht umgekehrt
- Äußere Schichten der Architektur hängen von inneren Schichten ab, nicht umgekehrt
- Klassen/Module hängen nicht von anderen Klassen/Modulen ab, sondern von Abstraktionen (z.B. Interfaces)
- Dependency Inversion erreicht diese Punkte durch Einführen von geeigneten Interfaces



Kata: String Rechner (Roy Osherove)

- Erstellen eines einfachen „Taschenrechners“, der eine Liste von Zahlen addieren kann
 - Inkrementelles Vorgehen, kleine Schritte!
 - In diesem Kata ist es nicht wichtig, Fehlerbehandlung vorzunehmen
- 

Kata: String Rechner (Roy Osheroove)

- Erstellen Sie ein Projekt mit einer Klasse `StringCalculator`, welche die folgende Methode hat:

```
int calculate(String numbers)
```

- `numbers` kann 0, 1 oder 2 durch Kommata getrennte Zahlen enthalten
- Andere Trennzeichen oder Whitespace sind nicht erlaubt
- Schreiben Sie erst einen Test für 0 Zahlen, dann für 1, dann für 2
- Vergessen Sie nicht, zu refactoren, nachdem ein Test funktioniert
- **Solve simply! (Und vergessen Sie nicht, zu refactoren)**



Workshop

Kata: String-Rechner

Siehe PDF!

(Bitte blättern Sie nach jedem Schritt erst weiter, wenn Sie das jeweilige Feature implementiert haben)