

Bachelor of Science (BSc) in Informatik
Modul Software-Entwicklung 1 (SWEN1)

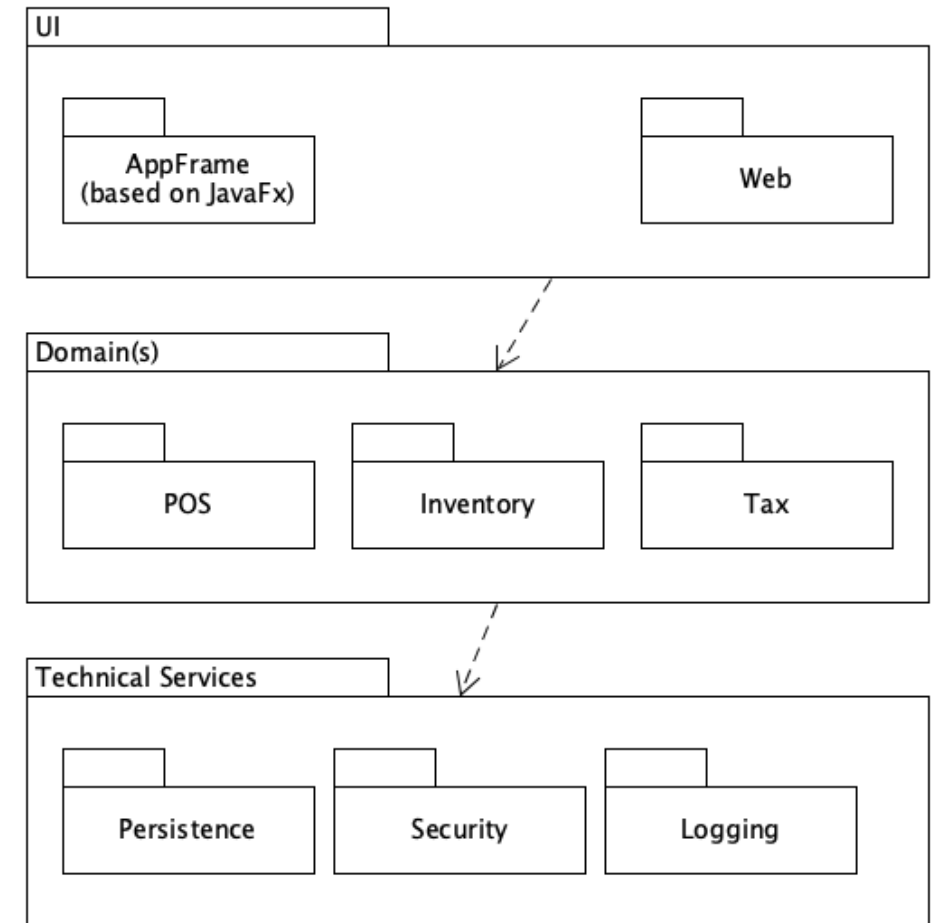
LE 10 – Implementation, Refactoring und Testing

SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

Ausgabe: HS24

Um was geht es?

- Wie kann ich aus den Design Artefakten einen Quellcode erstellen?
- Wie kann ich den Quellcode an neue Anforderungen anpassen bzw. die Qualität des Quellcodes kontinuierlich verbessern?
- Wie kann ich mit Hilfe von Tests die Voraussetzung für Refactoring schaffen?



Lernziele LE 10 – Implementation, Refactoring und Testing

- Sie sind in der Lage:
 - den **Quellcode** aus den Design Artefakten abzuleiten,
 - **Codier-Richtlinien** anzuwenden und eine zusätzliche Code Dokumentation wie Javadoc zu erzeugen,
 - eine **Umsetzungsstrategie** wie Test-Driven-Development (TDD) oder Behavior-Driven Development (BDD) einzusetzen,
 - den Quellcode mit Hilfe von **Refactoring** zu verbessern,
 - im Entwicklungsprozess **Tests** durchzuführen und kennen die grundlegenden **Testarten** und weitere **Teststufen**.

Agenda

1. Design to Code
2. Implementation
3. Refactoring
4. Testing
5. Wrap-up und Ausblick

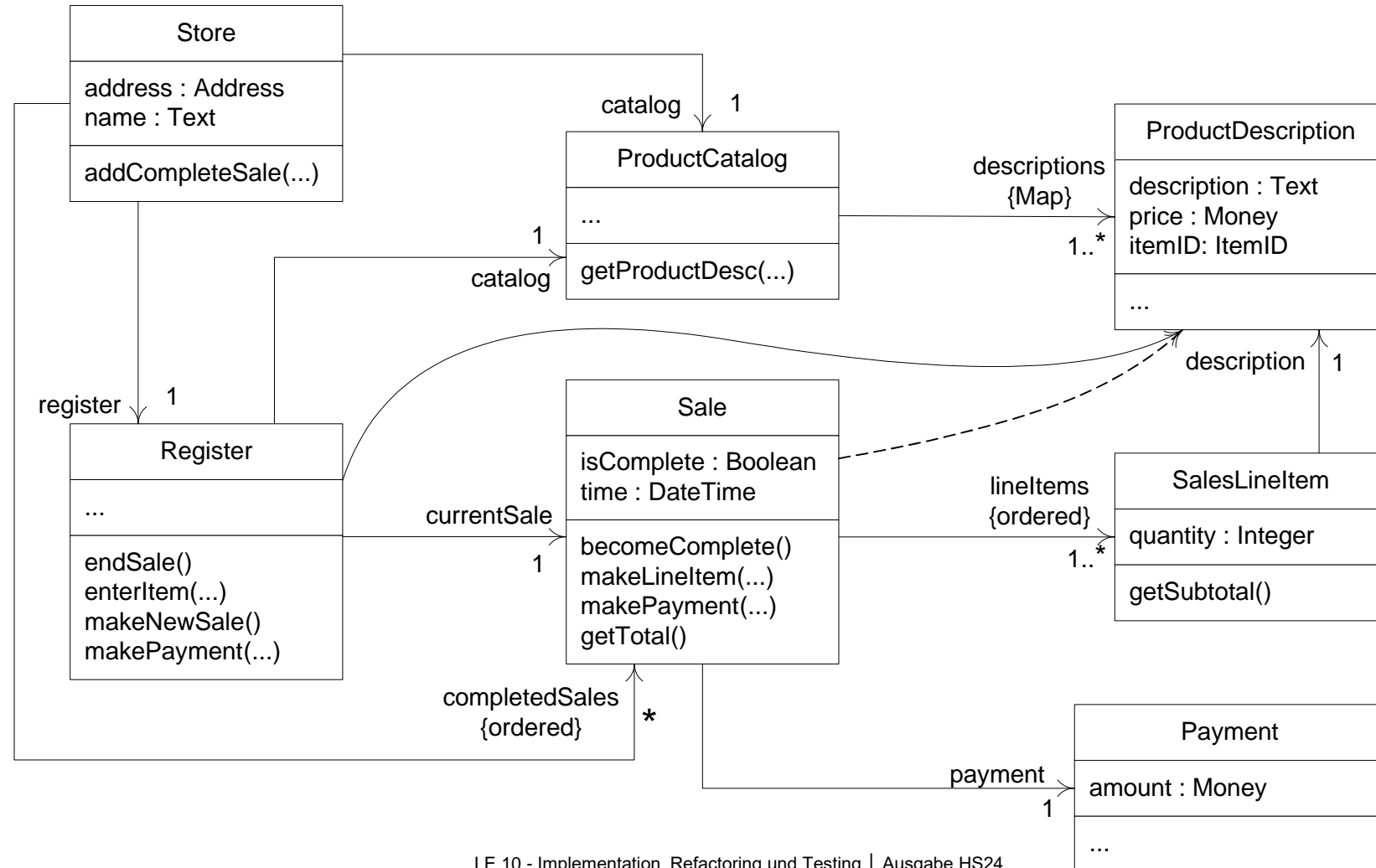
Design To Code (recap)

- Aus den vorhandenen **Design Artefakten** soll der **Quellcode** abgeleitet werden.
- In der Praxis sind **nur Teile des gesamten Quellcodes** zusätzlich als **Design Artefakte** abgebildet.

Beispiel Fallstudie: NextGenPos (recap)

DCD - Design Class Diagramm

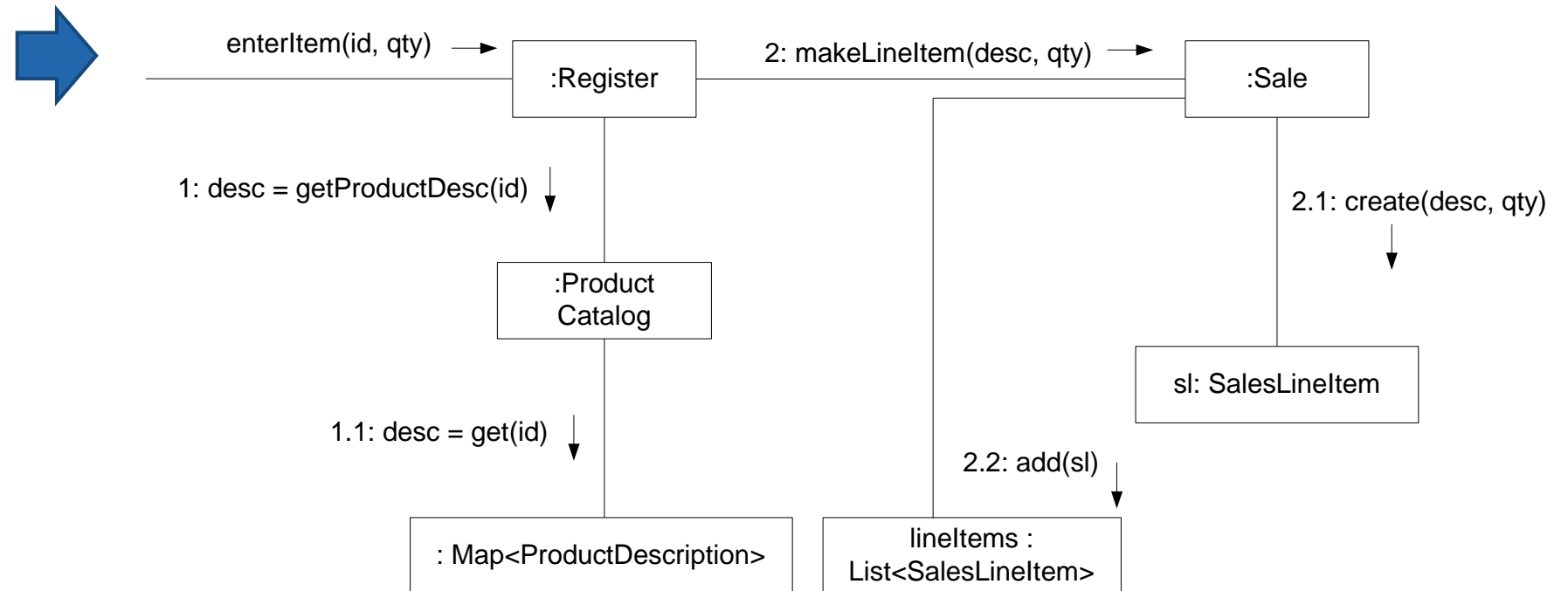
- Klassen
- Attribute
- Methoden
- Assoziation



Beispiel Fallstudie: NextGenPos (recap)

Methoden aus Interaktionsdiagrammen

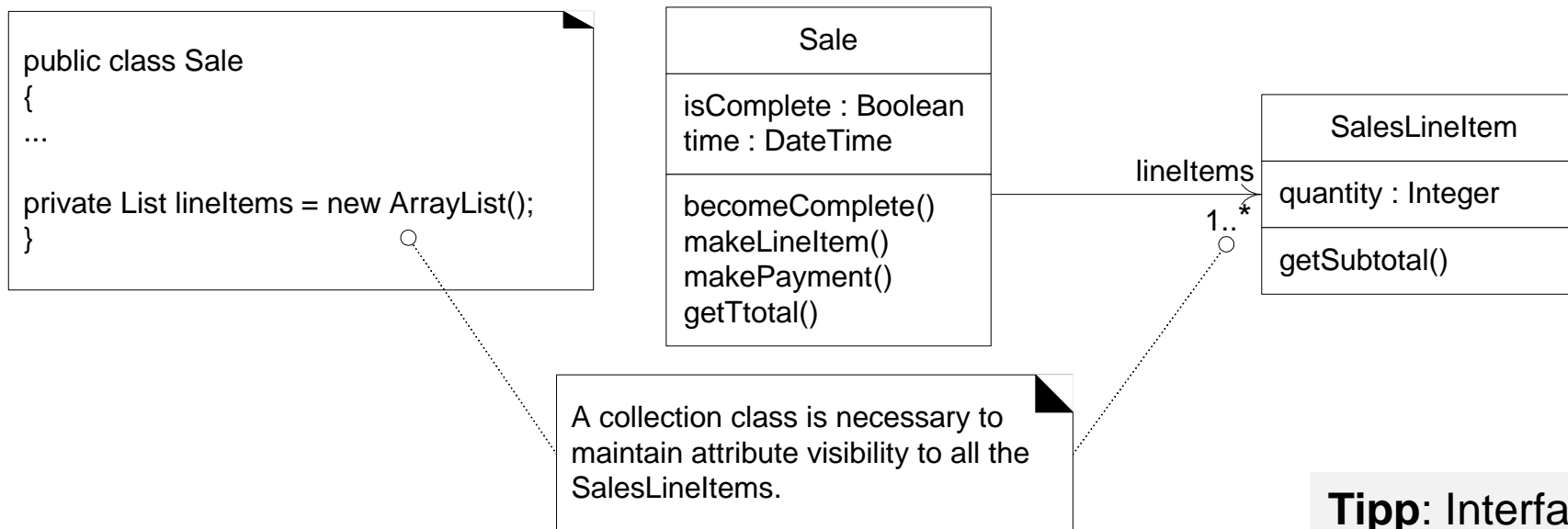
- Methoden mit Signaturen



```
Register.enterItem(int itemId, int qty);  
// Zwei Ereignisse werden an sichtbare Klassen gesendet  
ProductDescription desc = catalog.getProductDescription(itemId);  
currentSale.makeLineItem(desc, qty);
```

Einsatz von Collection Klassen (recap)

- 1 : n Beziehungen erfordern den Einsatz von Collection Klassen
- Die heutigen Programmiersprachen stellen ein reichhaltiges Sortiment an solchen Klassen zur Verfügung



Tipp: Interface und nicht Klasse deklarieren
(Protected Variation!)
Einsatz von Generics

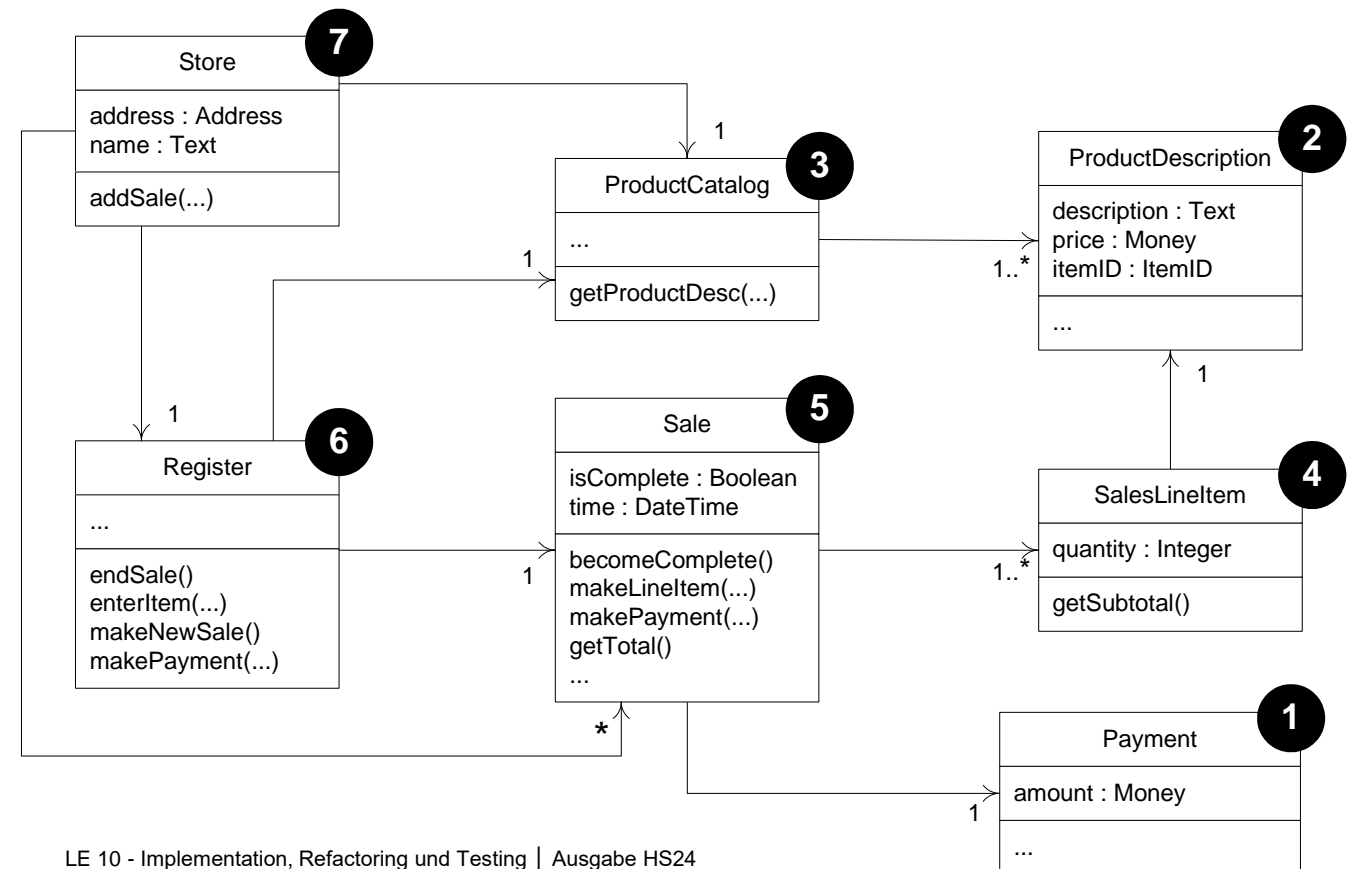
Fehlerbehandlung

- Exceptions verwenden.
 - Nicht C-Style mit Errorcode als Rückgabewert
- Exceptions wirklich nur für Fehlersituationen verwenden, nicht für reguläre Rückgabe-Werte.
- Standard Exceptions verwenden.
- Wo sinnvoll eigene Klassen definieren.
- Jede Schicht kapselt Exception Handling ab und reicht diese weiter.
- Welche Fehlermeldungen sollen dem Benutzer angezeigt werden?

Umsetzungs-Reihenfolge: Variante Bottom-Up Strategie

- Falls alle umzusetzenden Klassen als Design Artefakte vorhanden sind, kann eine Bottom-Up Strategie gewählt werden.

Beispiel Fallstudie: NextGenPos



Umsetzungs-Reihenfolge: Variante Agile

- Im agilen Umfeld werden Funktionen Schritt für Schritt umgesetzt. Es sind nur die für die Iteration notwendigen Klassen bekannt.
- Vorhandene Klassen müssen angepasst (refaktoriert) werden.
- Die Umsetzung wird über die verschiedenen Schichten der Architektur vollzogen wie Model, Controller, Services, Repository.
- Ausgangspunkt ist oft eine Schnittstellenbeschreibung:
 - Benutzerschnittstelle (von UX-Designer)
 - Systemschnittstelle (z.B. OpenApi Swagger)

Codierrichtlinien

- Legt verbindlich fest:
 - Gross/Kleinschreibung
 - Einrücken
 - Klammernsetzung { }
- Erleichtert Zurechtfinden in fremdem Code.
- Prüfprogramme für die Einhaltung der Codierrichtlinien:
 - SonarLint
 - Checkstyle
 - Lint / ESLint
 - ...

Namensgebung für Klassen, Attribute, Methoden, Variablen

- Die Namensgebung ist ausserordentlich wichtig für das Codeverständnis.
- Unbedingt die Namensgebung der Fachdomäne im Code abbilden.
- Falls notwendig die deutschen Begriffe durch englische Begriffe ersetzen und in einem für alle zugänglichen Glossar beschreiben.
- Englische Begriffe sind zentral für den Einsatz von internationalen Entwicklern.

Methoden

- High Cohesion auch auf Methoden anwendbar!
- Eher kleine Methoden mit starkem innerem Zusammenhang.
- CQS – Command Query Separation anwenden.
- Viele if's: Polymorphismus einsetzen?

Agenda

1. Design to Code
- 2. Implementation**
3. Refactoring
4. Testing
5. Wrap-up und Ausblick

Implementierungsstrategie festlegen

- Code-Driven Development
 - Zuerst die Klasse implementieren
- TDD: Test-Driven Development
 - Zuerst Tests für Klassen/Komponenten schreiben, dann den Code entwickeln
- BDD: Behavior-Driven Development
 - Tests aus Benutzersicht beschreiben
 - Zum Beispiel durch die Business Analysten mit Hilfe von Gherkin

Wichtig: Unabhängig von der gewählten Implementierungsstrategie muss jedes Stück Code nach der Fertigstellung auch entsprechende Tests haben!

Denkpause

Aufgabe 7.2 (3')

Frage:

Welche Vorteile können sich aus TDD (Test-Driven Development) ergeben?

Aufgabe 7.2 – Musterlösung

- Tests werden tatsächlich geschrieben.
- Programmierer ist dann motiviert, guten Code zu schreiben.
- Interface und Verhalten werden vorher geklärt.
- Wiederholbare Verifikation
 - Voraussetzung für Refactoring
- Vertrauen steigt, etwas zu ändern.

Erfolgsfaktoren

- Verständnis für die Aufgaben und Verantwortlichkeiten der Komponente
- Schnittstelle so einfach wie möglich
- Sinnvolle Namensgebung
- Code-Review Prozess durch Pull-Request's respektieren, Feedback betreffend Code-Qualität ernst nehmen
- Jede Programmzeile ist eine Entscheidung

Laufzeit Optimierung

- Oft wird versucht, während der Implementation, die Laufzeit zu optimieren.
- Sollte kritisch beobachtet werden.
- 3 Regeln der Optimierung:
 - Optimierte nicht
 - Optimierte NOCH nicht
 - Vor der Optimierung analysieren, wo wirklich Zeit verbraucht wird

Optimierungsregeln

- **Performance Monitor** einsetzen.
 - wo wird wirklich viel Zeit verbraucht !?
- Zeitfresser: Datenbankzugriffe pro Objekt über eine Liste
- Die Algorithmen optimieren.
 - Collections.sort(...) in Java 7 ist doppelt so schnell wie der von Java 6 (!)
- Erst in zweiter Linie den Code anschauen
 - Heutige Compiler optimieren bereits viel
 - Berechnungen aus der for Schleife herausnehmen
 - Java VM optimiert selber, und geht über «Just In Time Compilation» hinaus.

Agenda

1. Design to Code
2. Implementation
- 3. Refactoring**
4. Testing
5. Wrap-up und Ausblick

Refactoring

- Strukturierte, disziplinierte Methode, vorhandenen Code umzuschreiben
- Externes Verhalten bleibt gleich!
- Viele kleine Schritte (Codeänderungen)
- Interne Struktur wird verbessert.
 - Um Erweiterungen einzuleiten
- Trennen von der eigentlichen Weiterentwicklung!
- Low Level Design - Programmieretechnik

Code verbessern ...

- DRY: Keinen duplizierten Code
- Namensgebung: Klarheit erhöhen, Aussagekräftige Namen
- Lange Methoden verkürzen (kein Spaghetti-Code -> neue Methoden)
- Algorithmen strukturieren in
 - Initialisierung
 - Berechnung
 - Aufbereiten des Resultats
- Sichtbarkeit verbessern
- Testbarkeit verbessern

Was sind Code Smells?

- Duplizierter Code
- Lange Methoden
- Klassen mit vielen Instanzvariablen
- Klassen mit sehr viel Code
- Auffällig ähnliche Unterklassen
- Keine Interfaces, nur Klassen
- Hohe Kopplung zwischen Klassen

Unterstützung durch ...

- Automatisierte Tests
 - Stellen sicher, dass nach dem Refactoring der Code immer noch funktioniert
- Moderne Entwicklungsumgebungen
 - Automatisieren alle abhängigen Arbeitsschritte
 - Beispiel: Nach einer Umbenennung einer Variablen werden alle Verwendungen dieser Variablen ebenfalls geändert

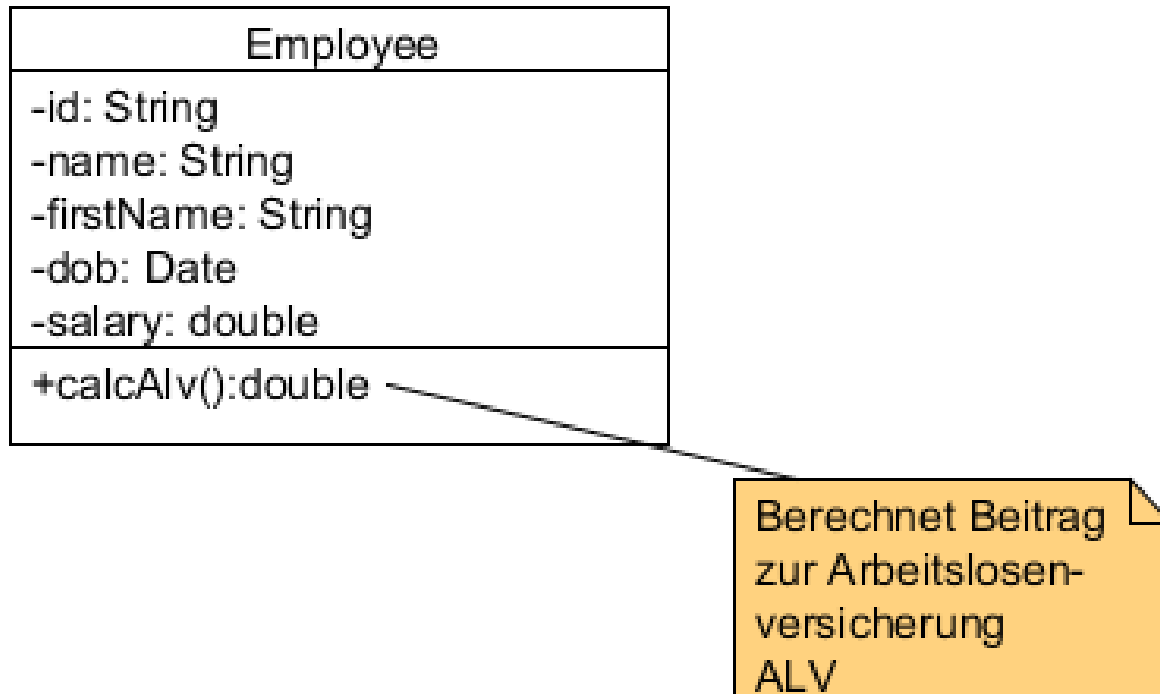
Refactoring Patterns (1/2)

- www.refactoring.com
- Rename Method / Class / Variable
 - Eine Methode/Klasse/Variable wird so umbenannt, dass sie einen aussagekräftigen Namen erhält.
- Pull Up / Push Down
 - Eine Methode wird in eine Superklasse / Subklasse verschoben.
- Extract Interface / Superclass
 - Ein Teil eines bestehenden Interfaces / Klasse wird in eine Superinterface / Superklasse extrahiert.

Refactoring Patterns (2/2)

- Extract Method
 - Teil einer Methode in eine private Methode auslagern.
- Extract Constant
 - Symbolische Konstante verwenden.
- Introduce Explaining Variable
 - Grossen Ausdruck aufteilen, erklärende Zwischenvariablen einfügen.

Beispiel Klasse Employee (1/3)

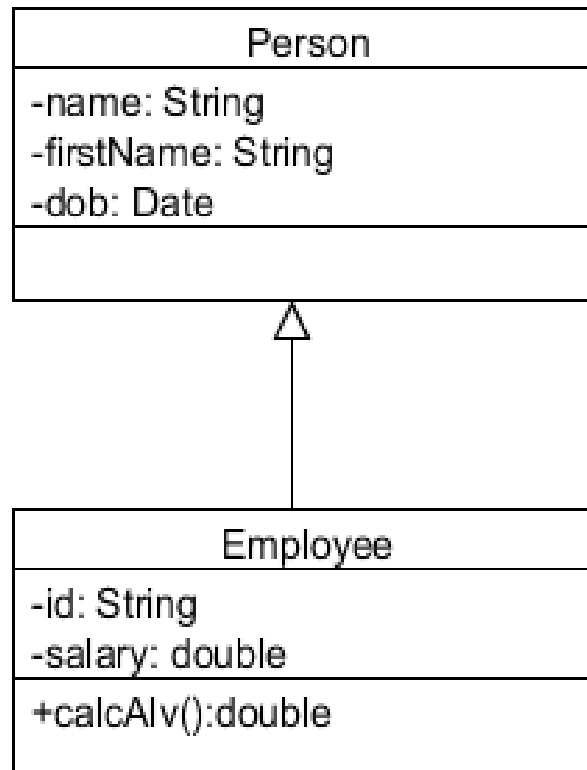


Neu soll eine Klasse
Customer hinzukommen

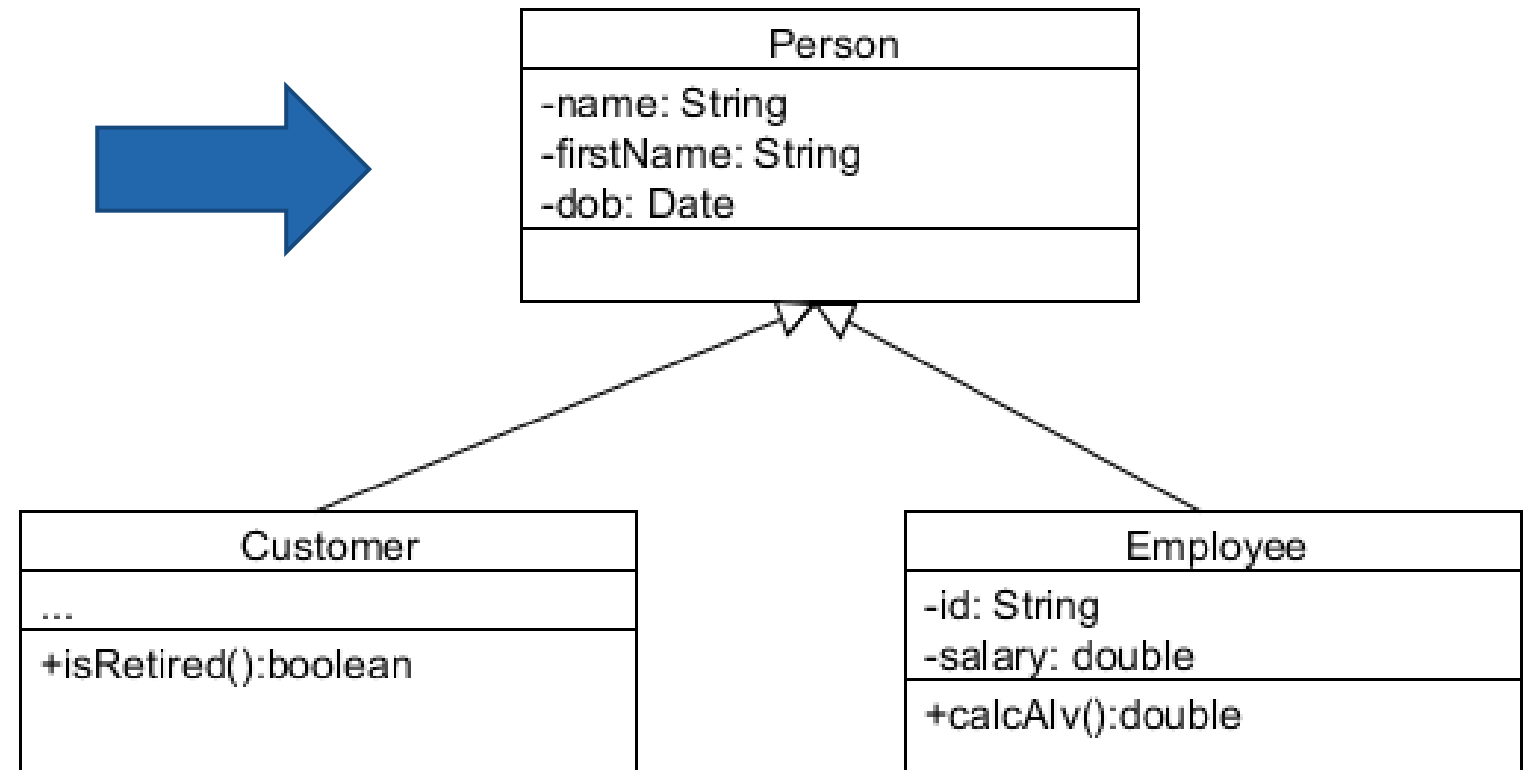
Gibt es allenfalls
Überschneidungen
dieser neuen Klasse
mit Employee?

Beispiel Klasse Employee (2/3)

Klasse Person extrahieren

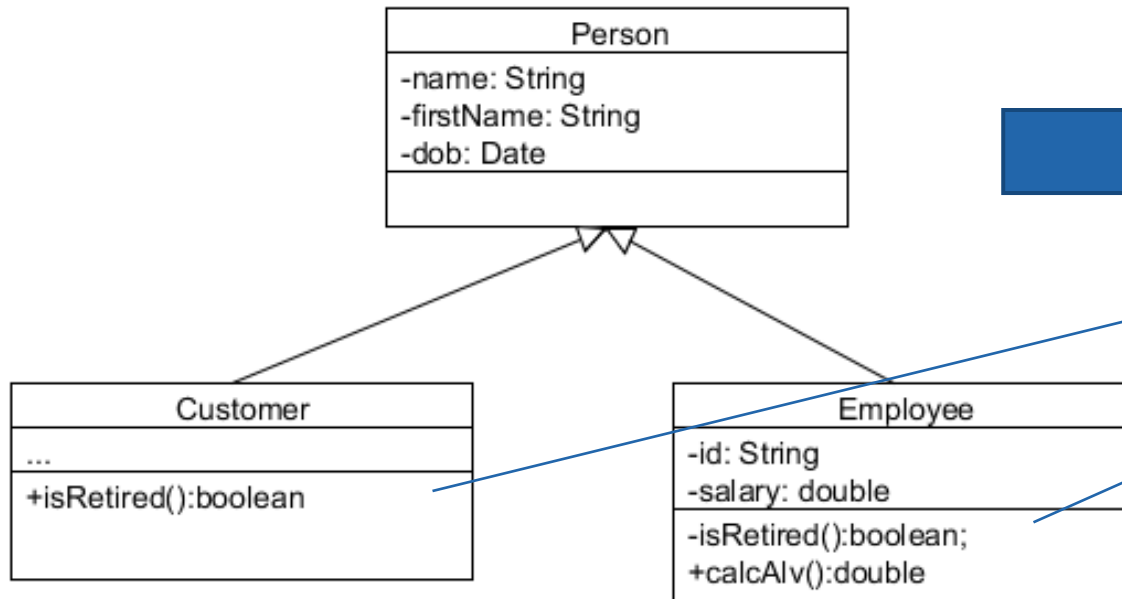


Klasse Customer hinzufügen

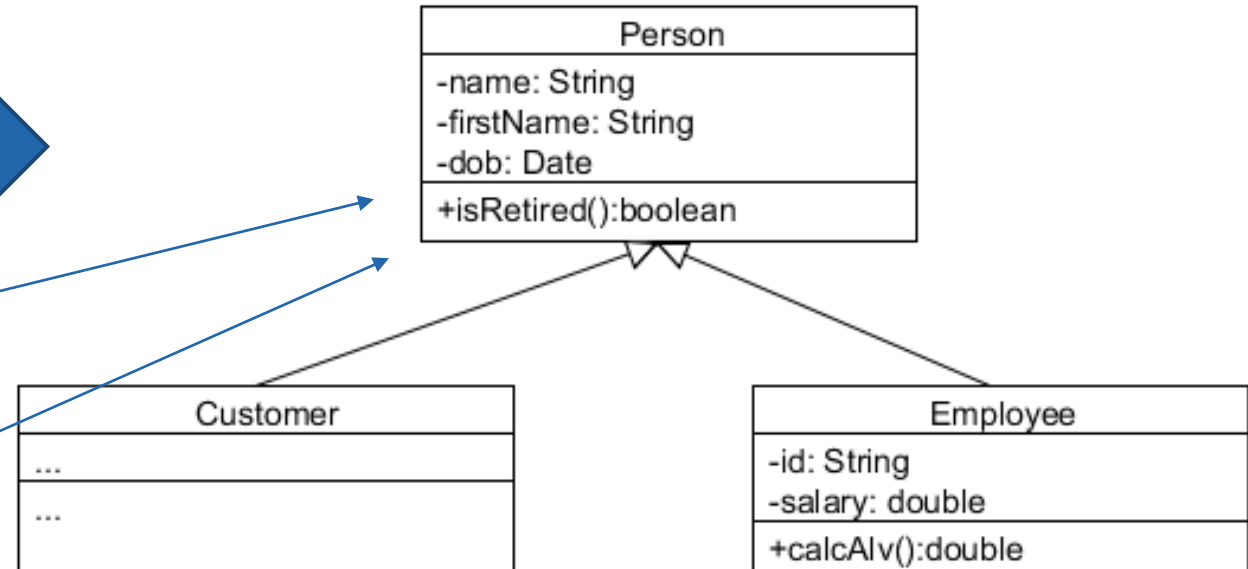


Beispiel Klasse Employee (3/3)

Employee.isRetired extrahieren



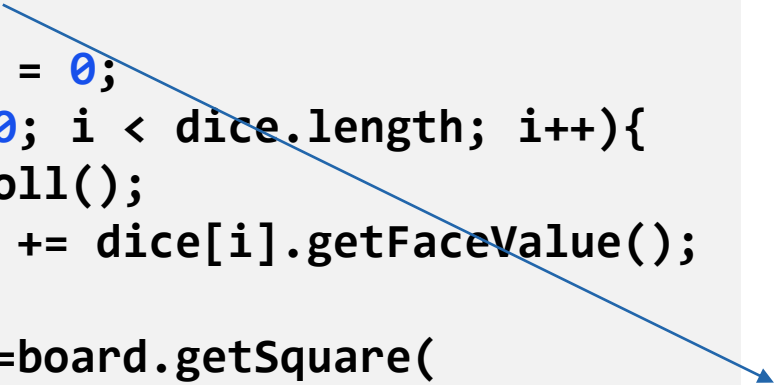
Pull up Employee.isRetired



Sie müssen berechnen, ob ein Mitarbeiter bereits pensioniert ist.
Haben Sie den Code dafür nicht schon an einem anderen Ort programmiert?

Methode extrahieren – kein Spaghetti-Code

```
public void takeTurn(){  
    // roll dice  
    int rollTotal = 0;  
    for (int i = 0; i < dice.length; i++){  
        dice[i].roll();  
        rollTotal += dice[i].getFaceValue();  
    }  
    Square newLoc=board.getSquare(  
        piece.getLocation(), rollTotal);  
    piece.setLocation(newLoc);  
}
```



```
public void takeTurn(){  
    // the refactored helper method  
    int rollTotal = rollDice();  
    Square newLoc = board.getSquare(  
        piece.getLocation(), rollTotal);  
    piece.setLocation(newLoc);  
}  
  
private int rollDice(){  
    int rollTotal = 0;  
    for (int i = 0; i < dice.length; i++){  
        dice[i].roll();  
        rollTotal += dice[i].getFaceValue();  
    }  
    return rollTotal;  
}
```

Lange Codefragmente mit Kommentaren wie
//roll dice
mit verständlichen Methoden verkleinern

Agenda

1. Design to Code
2. Implementation
3. Refactoring
4. **Testing**
5. Wrap-up und Ausblick

Denkpause

Aufgabe 7.4 (2')

Frage:

Was ist das Ziel des (Software) Testings ?

Aufgabe 7.4 – Musterlösung

- Globales **Ziel** des Softwaretestens ist das Messen der Qualität des Softwaresystems. Dabei dienen definierte Anforderungen als Prüfreferenz, mittels derer ggf. vorhandene Fehler aufgedeckt werden. ISTQB: Der Wirkung von Fehlern (im produktiven Betrieb) wird damit vorgebeugt.

Denkpause

Aufgabe 7.5 (3')

Fragen:

- Was ist ein Unit-Test?
- Wer entwickelt ihn?
- Wer führt ihn aus?
- Wie ist er (ganz grob) aufgebaut?

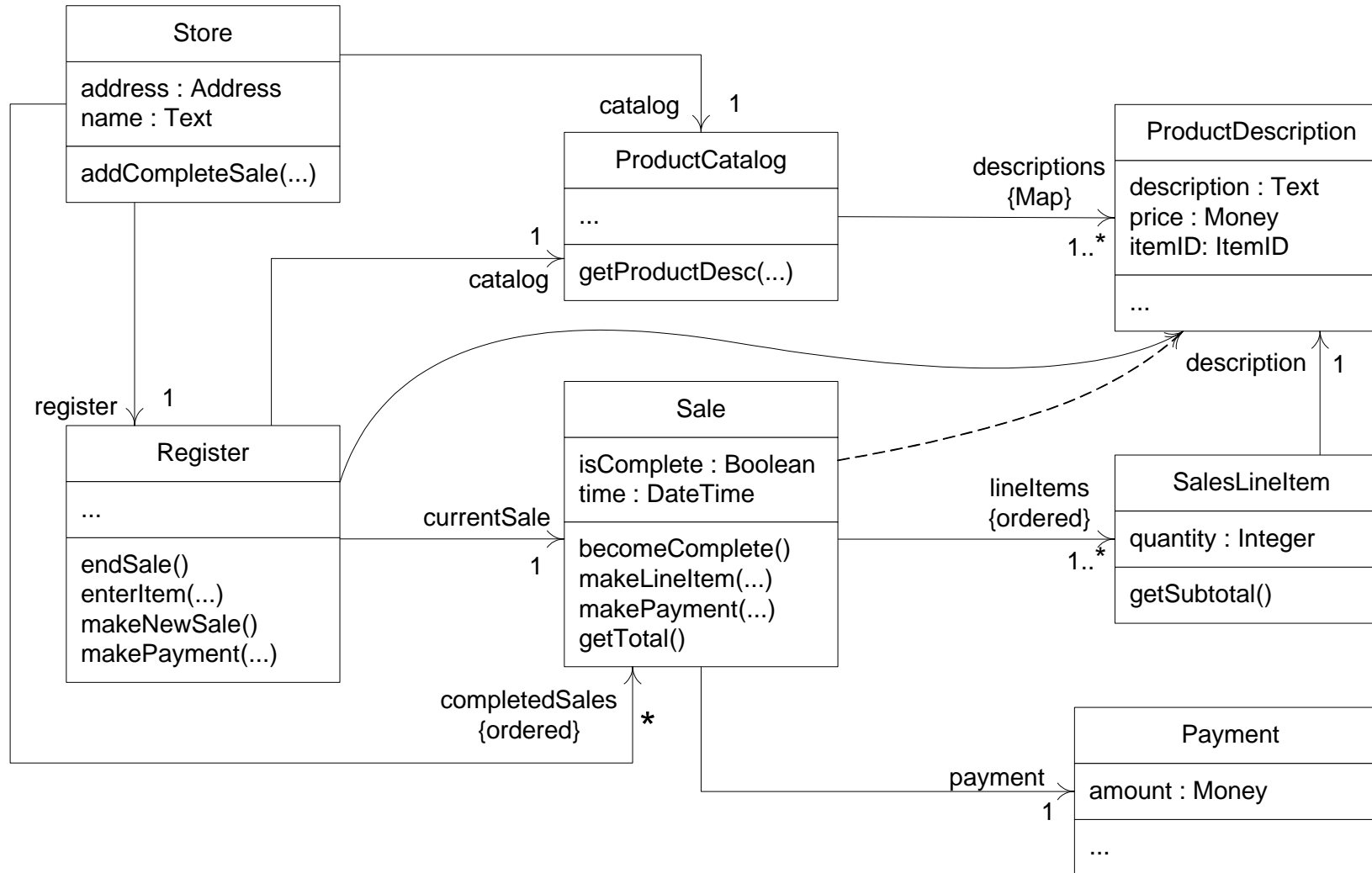
Aufgabe 7.5 – Musterlösung

- Was ist ein Unit-Test:
 - Unit-Tests (=Komponententests) überprüfen, ob die von den Entwicklern geschriebenen Komponenten so arbeiten, wie diese es beabsichtigen.
- Wer entwickelt ihn?
 - Entwickler
- Wer führt ihn aus?
 - Test Runner: in der IDE, Build Automatisierung (maven, gradle), Continuous Integration Server
- Wie ist er (ganz grob) aufgebaut?.
 - Setup, Tests, TearDown

Aufgabe 7.6 (3')

Fragen:

- Sie sollen für die Klasse Sale einen Unit-Test schreiben (Design-Klassendiagramm folgt auf der nächsten Folie).
- Was ist dabei das Problem für einen reinen Unit-Test?
- Wie kann es gelöst werden?



Aufgabe 7.6 – Musterlösung

- Der Unit Test soll nur die Unit und nicht das ganze Umfeld mittesten
- Es muss ein Mocking des Umfeldes durchgeführt werden
- Mocking kann aufwändig sein

Repetition Grundlegende Testarten

- Funktionaler Test (Black-Box Verfahren)
- Nicht funktionaler Test (Lasttest etc.)
- Strukturbezogener Test (White-Box Verfahren)
- Änderungsbezogener Test (Regressionstest etc.)

Denkpause

Aufgabe 7.7 (3')

Fragen:

- Was sind Äquivalenzklassen?
- Können Sie ein Beispiel dafür geben?

Aufgabe 7.7 – Musterlösung

- Ziel: hohe Fehlerentdeckungsrate mit einer möglichst geringen Anzahl von Testfällen zu erreichen.
- Die Äquivalenzklassen sind also bezüglich Ein- und Ausgabedaten ähnliche Klassen bzw. Objekte,
- So sind beispielsweise in einem Programm zur Verwaltung eines Fuhrparks Fahrzeuge äquivalente Klassen („Ferrari“ und „BMW“ sind vergleichbar, „Ferrari“ und „Mitarbeiter“ nicht).
- Quelle: Wikipedia

Denkpause

Aufgabe 7.8 (3')

Fragen:

- Wie wird ein Test mittels Grenzwertanalyse ermittelt?
- Wie lassen sich Äquivalenzklassen und Grenzwertanalyse kombinieren?

Aufgabe 7.8 – Musterlösung

- Die Grenzwertanalyse ist ein Spezialfall der Äquivalenzklassenanalyse. Sie ist aus der Beobachtung entstanden, dass Fehler besonders häufig an den „Rändern“ der Äquivalenzklassen auftreten.
- Quelle: Wikipedia

Weitere Teststufen und Testarten

- Integrationstest
- Systemtest
- Abnahmetest
- Regressionstest

Integrationstest

- Eine Klasse wird im Anwendungskontext eingesetzt
- Es werden nun keine Mockups, sondern die richtigen referenzierten Klassen eingesetzt
- Typischerweise wird dann ein **ganzes Subsystem**, vielleicht auch das ganze System, getestet
- Black-Box-Test, mit zusätzlichem Wissen über Internas

- Das **ganze System** oder die gesamte Anwendungslogik wird getestet
- Typischerweise ein Black-Box-Test
- Wird nicht nur während der Entwicklung, sondern auch vor einer Auslieferung an den Kunden durchgeführt
- Anwendungsfälle beiziehen

Abnahmetest

- Nach der Auslieferung wird die gesamte Software vom Kunden getestet
- Meist ein Systemtest über das UI
- Reiner Black-Box-Test
- Orientiert sich an den Anforderungen des Kunden (was er für wichtig hält)
- Oft relevant für die Bezahlung

Regressionstest

- Automatische Wiederholung von Tests nach Veränderungen am Quelltext
 - Von Vorteil, wenn diese Tests ebenfalls als ausführbarer Code vorliegen
- Nach Refactoring Tätigkeiten
- Nach Weiterentwicklung für die Funktionen, die nicht geändert haben

Reproduktion von Fehlern

- Nachdem ein Fehler der Entwicklungsabteilung gemeldet wurde, wird als Erstes ein Testfall geschrieben.
- Reproduziert den Fehler möglichst exakt.
- Systemtest über UI (nur wenn nicht anders möglich) oder Anwendungslogik (besser).
- Hat eher White-Box-Test Charakter.

Einbindung in den Prozess

- Testfall vor der Implementation schreiben
 - Black-Box Test, den der Entwickler selber schreibt
- Testfall nach der Implementation schreiben
 - Black-Box Test, mit White-Box Test Bereicherungen
 - Unit-, Integration- und/oder Systemtests, Entwickler
- Qualitätssicherung
 - Black-Box System Test, eigene Organisationseinheit
- Abnahmetest
 - Black-Box System Test, Kunde
- Reproduktion von Fehlern

Wichtige Begriffe

- Testling, Testobjekt
 - Objekt, das getestet wird
- Fehler
 - Der Entwickler macht einen Fehler
- Fehlerwirkung, Bug
 - Jedes zu den Spezifikationen abweichende Verhalten
- Testfall
 - Satz von Testdaten zur vollständigen Ausführung eines Tests
- Testtreiber
 - Rahmenprogramm, das den Test startet und ausführt

Merkmale (1/2)

- Was wird getestet?
 - Eine Einheit / Klasse (Unit-Test)
 - Zusammenarbeit mehrerer Klassen
 - Die gesamte Applikationslogik (ohne UI)
 - Die gesamte Anwendung (über UI)
- Wie wird getestet?
 - Dynamisch: Testfall wird ausgeführt
 - Black-Box Test
 - White-Box Test
 - Statisch: Quelltext wird analysiert
 - Walkthrough, Review, Inspektion

Merkmale (2/2)

- Wann wird der Test geschrieben?
 - Vor dem Implementieren (Test-Driven Development, TDD)
 - Nach dem Implementieren
- Wer testet?
 - Entwickler
 - Tester, Qualitätssicherungsabteilung
 - Kunde, Endbenutzer

Schlussbemerkungen Testing

- Testen ist sehr wichtig zur Steigerung der Softwarequalität!
- Automatische Tests für die Domänenlogik programmieren
 - Weitere Motivation, UI von der Domäne zu trennen
- Einsatz von Mocks für echte Unit-Tests
- Gute Mischung aus Unit- und Systemtests
- Schlusstest über UI ist aber immer notwendig

Agenda

1. Design to Code
2. Implementation
3. Refactoring
4. Testing
- 5. Wrap-up und Ausblick**

Wrap-up

- Die **Umsetzung des Entwurfs in Code** ist eine **anspruchsvolle Aufgabe** und braucht **Disziplin**.
- Es muss von **Anfang an eine Umsetzungsstrategie** wie z.B. TDD gewählt werden.
- **Refactoring** ist im agilen Umfeld **eine zentrale Tätigkeit**. Sie dient der kontinuierlichen Qualitätsverbesserung.
- **Testing** ist die **Voraussetzung für Refactoring**.
- Ohne eine **genügende Testabdeckung** ist Refactoring sehr risikobehaftet und bewirkt ein fehleranfälliges Software-Produkt.

Ausblick

- In den nächsten drei Lerneinheiten werden wir:
 - je ein Thema vertiefen (3 aus 4).
 - Verteilte System
 - GUI-Architekturen
 - Persistenz
 - Framework-Design

Quellenverzeichnis

- [1] Spillner, A. und Linz, T.: Basiswissen Softwaretest, dpunkt-Verlag, 2019
- [2] Fowler, M.: Refactoring, Addison-Wesley, 2018
- [3] Fowler, M.: Test Driven Design, Addison-Wesley, 2005