0.0.1 Design to Code

- Quellcode aus Design Artefakten ableiten
- Praxis: Nur Teile des gesamten Quellcodes zusätzlich als Design Artefakte abgebildet.

Einsatz von Collection Klassen: Erforderlich bei 1:n Beziehungen

Fehlerbehandlung:

- Exceptions verwenden
- Exceptions nur für Fehlersituationen, nicht für reguläre Rückgabewerte
- Standard Exceptions verwenden
- Wo sinnvoll eigene Klassen definieren
- Jede Schicht kapselt Exception Handling ab

Umsetzungs-Reihenfolge

Bottom-Up Strategie: Wenn alle umzusetzende Klassen als Design Artefakte vorhanden sind

Variante Agile:

- Nur für die Iteration notwendigen Klassen bekannt. Funktionen Schritt für Schritt umgesetzt.
- Vorhandene Klassen müssen angepasst werden (Refaktoriert)
- Umsetzung über verschiedene Schichten der Architektur vollzogen (Model, Controller, Services, Repository)
- Ausgangspunkt oft Schnittstellenbeschreibung:
  - Benutzerschnittstelle (UI-Designer)
  - Systemschnittstelle (OpenApi Swagger)

Methoden:

- High Cohesion
- Eher kleine Methoden mit startkem inneren Zusammenhang
- CQS - Command Query Separation anwenden (setter und getter?)
- Wenn viele if's: Polymorphismus einsetzen

0.1 Implementation

3 Verschiedene Implementierungsstrategien:

1. Code-Driven Development (Zuerst die Klasse implementieren)
2. Test-Driven Development (TDD) (Zuerst Tests für Klassen/Komponenten dann Code entwickeln)
3. Behaviour-Driven Development(BDD:
   - Test aus Benutzerschicht beschreiben
   - Z.B durch die Business Analystem mit Hilfe von Gherking

Unabhängig der gewählten Strategie, jedes stück Code muss am Schluss Tests haben.

0.2 Refactoring

Definition:

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

Code verbessern:

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

Code Smells:

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

Nach Refactorn wieder testen ob Code noch funktioniert.

0.2.1 Refactoring Patterns

- Rename Method/Class/Variable
- Pull Up / Push Down
  - Methode in Superklasse / Subklasse verschoben
- Extract Interface / Superclass
  - Teil bestehendes Interfaces/Klasse wird in eine Superinterface / Superklasse extrahiert.
- Extract Method

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

0.3 Testing

Testarten:

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

Weitere Teststufen und Testarten

- Integrationstest
- Systemtest
- Abnahmetest
- Regressionstest

0.3.1 Integrationstest

- Eine Klasse wird im Anwendungskontext eingesetzt
- Keine Mockups sondern die richtigen referenzierten Klassen eingesetzt
- Ganzes Subsystem oder ganzes System getestet
- Black-Box-Test mit zusätzlichem Wissen über Internas

0.3.2 Systemtest

- Ganzes System oder gesamte Anwenderlogik wird getestet
- Typischerweise Black-Box-Test
- Nicht nur während Entwicklung sondern auch vor Auslieferung an Kunden
- Anwendungsfälle beiziehen

0.3.3 Abnahmetest

- Nach der Auslieferung wird die gesamte Software vom Kunden getestet
- Meist Systemtest über das UI
- Reiner Black-Box-test
- Orientiert sich an Anforderungen des Kunden
- Oft relevant für die Bezahlung

0.3.4 Regressionstest

- Automatische Wiederholung von Tests nach Veränderungen am Quelltext
- Nach Refactoring
- Nach Weiterentwicklung für Funktionen, die nicht geändert haben.

0.3.5 Reproduktion von Fehlern

- Tesfall schreiben nach Meldung Fehler
- Reproduziert Fehler möglichst exakt
- Am besten Systemtest Anwendungslogik sonst über UI
- Eher White-Box-Test

0.3.6 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

0.3.7 Wichtige Begriffe

- Testling, Testobjekt
  - Objekt, das getestet wird
- Fehler
  - 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

0.3.8 Merkmale

- Was wird getestet
  - Einheit/Klasse (Unit-Test)
  - Zusammenarbeit mehrerer Klassen
  - Gesamte Applikationslogik (ohne UI)
  - gesamte Anwendung (über UI)
- Wie wird getestet
  - Dynamisch: Testling wird ausgeführt
    - Black-Box Test
    - White-Box Test
  - Statisch: Qelltext wird analysiert
    - Walkthrough, Review, Inspektion
- Wann wird der Test geschrieben?
  - Vor dem Implementieren (TDD)
  - Nach dem Implementieren
- Wer testet?
  - Entwickler
  - Tester, Qualitätssicherung
  - Kunde, Endbenutzer