CT Präsentationsprüfung
Zusammenfassung

Marvin Borner

13. Juli 2021

1 Präsentation

1.1 Grober Aufbau

1. Deckblatt
2. Inhalt (Gliederung - Beschreiben)
3. Problemstellung
4. Grundlagen
5. Hauptteil
6. Schluss/Fazit
7. IDE/Demonstration

1.2 Hinweise

- Keine dunkle IDE nutzen (sieht man nicht gut auf dem Beamer - 2NP Abzug!)
- Nicht monoton reden (=> Motiviert sein!)
- max. 12min Vortrag; 8min Kolloquium/Fragen

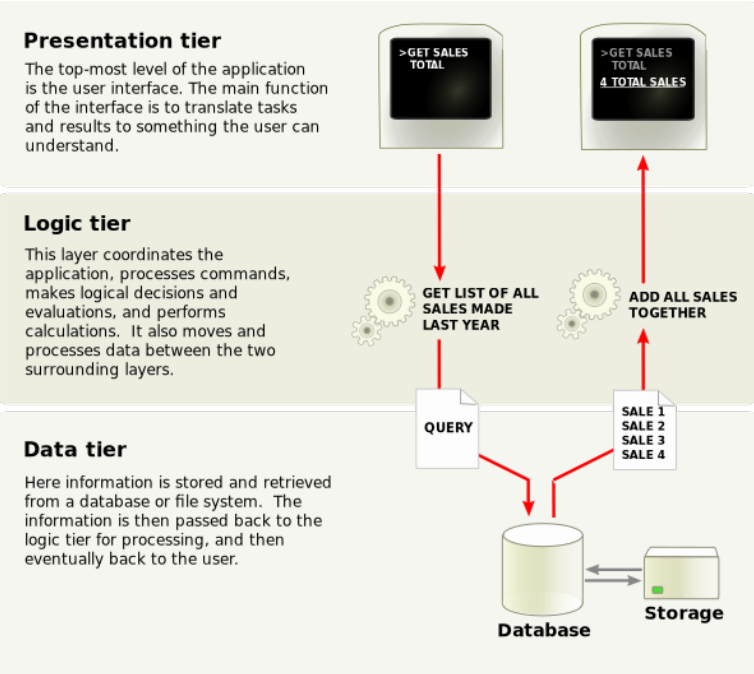
2 Kolloquium Themen

2.1 Patterns (Entwurfsmuster)

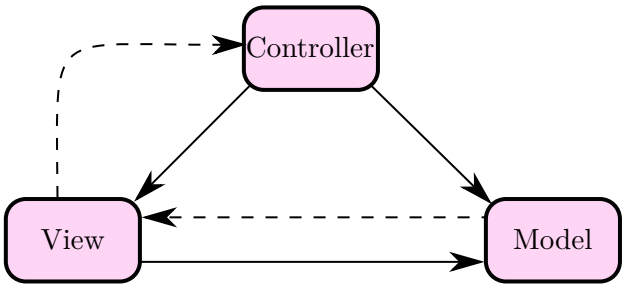
Muster für wiederkehrende Entwurfsprobleme in der Softwarearchitektur und -entwicklung => Vorlagen zur Problemlösung.

Relevante Muster:

- **Drei-Schichten-Architektur**
  - Überbegriff für viele verschiedene Architekturen
  - Reduktion von Komplexität (=> bessere Wartung)
  - Teilweise langsamer, da Daten häufig zwischen Schichten transportiert werden müssen
  - Typischerweise Präsentationsschicht (GUI; Presentation), Logikschicht (Steuerung; Logic) und Datenhaltungsschicht (Daten; Data)

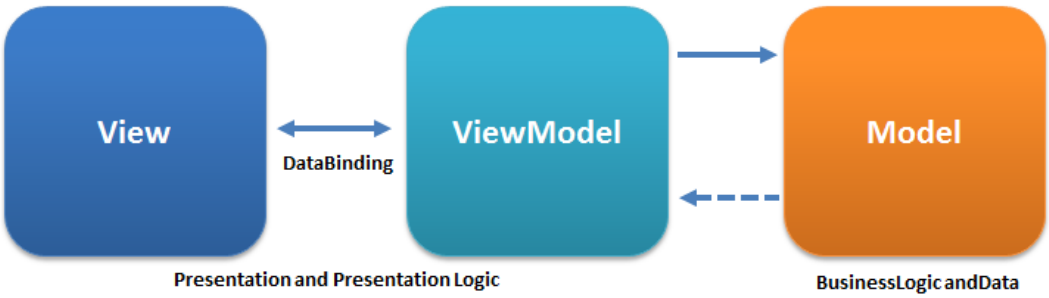


- **MVC**
  - Steht für *Model-View-Controller*
  - Ziel: Flexibler Programmentwurf; offen für spätere Erweiterungen und Änderungen; Wiederverwendbarkeit einzelner Komponenten
  - Besitzt viele Abwandlungen
  - Aufbau
    * Model: enthält Daten, die von der View dargestellt werden; benachrichtigt die View über Änderungen; unabhängig von View und Controller
    * View: Darstellung aller GUI-Elemente; unabhängig von Controller; Bekanntgabe an Controller mittels Listener; aktualisiert UI mittels Listener zu Controller
    * Controller: Verwaltet View und Model; bekommt UI-Interaktionen von View über Listener; „schickt“ UI-Änderungen an Listener der View; „schickt“ Änderungen von Daten an Listener des Models



- **MVVM**
  - Steht für *Model-View-ViewModel*
  - Variante des MVC; nutzt Datenbindungsmechanismen
  - Beispiel: C# mit WPF (Window Presentation Framework) - *Databindings und Interfaces S.83*
  - Trennt Darstellung und Logik

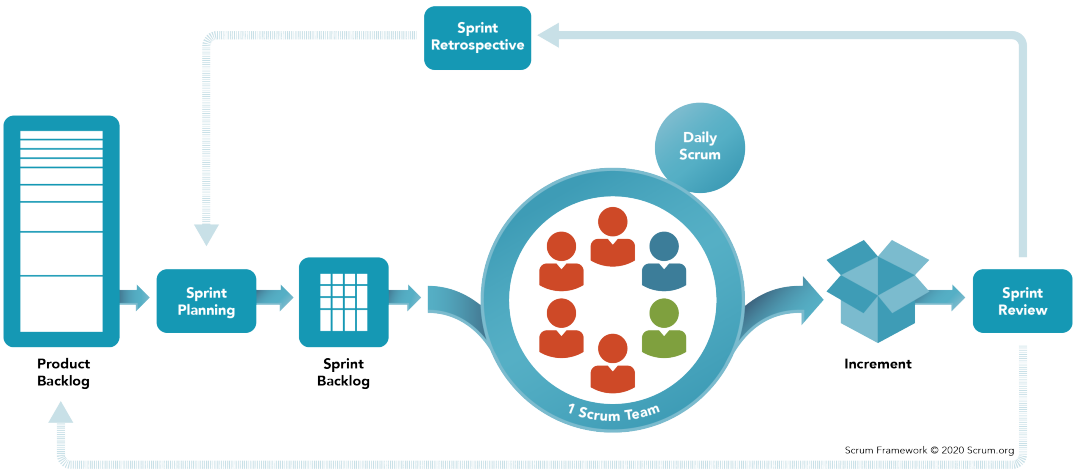
- Im Gegensatz zu MVC muss man nicht für alles Controller implementieren => geringerer Implementierungsaufwand => bessere Testbarkeit
- Durch drei einzelne Instanzen besser testbar (keine extra UI-Tests nötig)
- Rollentrennung von UI-Designern und Entwicklern möglich (z.B. in Firmen)
- Aufbau
  * Model: *Datenzugriffsschicht*; benachricht über Datenänderungen; führt Validierungen von Benutzereingaben durch; enthält gesamte „Geschäftslogik“; ist als einzelnes Element mit Unit-Tests testbar
  * View: Darstellung aller GUI-Elemente; Bindung zu Viewmodel; einfach austauschbar/modifizierbar/erweiterbar ohne alles andere zu zerstören
  * ViewModel: UI-Logik (Model der View); verbindet View und Model; tauscht Informationen mit Model aus; stellt der View Eigenschaften und Befehle zur Verfügung welche von der View and Elemente gebunden werden; hat keine Ahnung von Elementen der View



2.2 Projektmanagement

- **Scrum**
  - Strikte Rollen: Product Owner (priorisiert Anforderungen), Entwicklungsteam (verantwortlich für die Erreichung der Ziele), Scrum Master (kontrolliert Scrum-Ablauf und Kooperation)
  - User Stories: Anforderungen an das Produkt
  - Entwicklungsprozess ist in Iterationen (=> *Sprints*) organisiert (brauchen max. 4 Wochen)
    1. **Product Backlog**: Team bekommt priorisierte Liste von Anforderungen
    2. **Sprint Planning**: Team wählt Teil der Anforderungen als Ziel für diesen Sprint aus (=> nutzbare Zwischen-Version nach Sprint: *Inkrement*); Planung der Zeit, die der Sprint benötigen soll; Organisation: Wer macht was?
    3. **Sprint Backlog**: Erstellung des für alle sichtbaren Scrum-Boards (To-Do, In-Progress, Done); Aufteilung der Anforderungen in kleinere Aufgaben (inklusive Tests, Dokumentationen, etc)
    4. **Sprint**: Entwickler des Teams arbeiten an ihren jeweiligen Aufgaben
    5. **Daily Scrum Meeting**: Max. 15min; täglicher Abgleich des Fortschrittes; Kontrolle und Hilfestellungen einzelner Entwickler; wird wiederholt, bis Sprint zuende ist
    6. **Sprint Review**: Am Ende des Sprints; Überprüfung des Inkrements; ggf. Anpassung des Product Backlogs; zurück zum Produkt Backlog wenn alle Anforderungen erfüllt sind

7. **Sprint Retrospektive**: Am Ende des Sprints; Evaluation von Kritik/Verbesserungsvorschlägen am Ablauf
8. Zurück zum Sprint Planning für den nächsten Sprint - Sprints werden so lange wiederholt, bis alle Anforderungen erfüllt sind



- **Wasserfall**
  - Lineares Modell (im Gegensatz zum iterativen Scrum)
  - Sequenziell: Jede Phase muss beendet sein, bevor die nächste anfängt
  - Vorteile
    * Einfach und verständlich
    * Klare Abgrenzung der Phasen
    * Einfache Möglichkeiten der Planung und Kontrolle
    * Bei stabilen Anforderungen: Klare Abschätzung von Kosten und Aufwand
  - Nachteile
    * Klar abgegrenzte Phasen in der Praxis unrealistisch - häufig fließender Übergang
    * In der Praxis sind Rückschritte oft unvermeidlich - in Wasserfall nicht erlaubt
    * Unflexibel gegenüber Änderungen
    * Frühes Festschreiben der Anforderungen kann problematisch sein, wenn es Änderungen gibt (alles muss erneut durchgeführt werden)
    * Kein durchgehendes Testen (wie bei Scrum nach jedem Sprint), sondern erst wenn alles fertig ist (=> potentielle Katastrophe bei finaler Implementation)
  - Typische Phasen
    1. Anforderungsanalyse (=> Lastenheft)
    2. Systemdesign (=> Softwarearchitektur)
    3. Programmierung/Unit-Tests (=> Software)
    4. Integrations-/Systemtests
    5. Auslieferung, Einsatz und Wartung
- **Test-Driven-Development** (TDD)
  - Phasen bei Implementation neuer Features
    1. Test schreiben, welcher die Spezifikation des Features voll erfüllt

2. Alle Tests durchlaufen lassen: Neuer Test muss fehlschlagen (Prävention eines fehlerhaften Tests, welcher immer besteht)
3. Einfachste Implementation, die den Test bestehen lässt (muss nicht schön sein, soll einzig und allein den Test bestehen lassen)
4. Alle Tests durchlaufen lassen: Alle Tests müssen jetzt bestanden sein
5. Refactor: Die neue Implementation lesbarer und wartbarer überarbeiten mit durchgehender Test-Kontrolle
6. Für jedes neue Feature wiederholen

- Vorteile
  * Spezifikationen werden schon im Vorraus beachtet (=> weniger Fehleranfällig)
  * Implementations-Code wird automatisch möglichst klein, da dieser nur geschrieben wird, um die Tests zu bestehen
  * Weniger debugging, da mithilfe der Tests und guten VCS genau verfolgt werden kann, was schiefläuft
  * Besser wartbar, weil modularisiert und flexibel
- Nachteile
  * Vollständige Tests können vernachlässigt werden, weil Unit-Tests ein falsches Gefühl der Sicherheit verursachen können
  * Schlecht geschriebene Tests führen zu schwieriger Wartbarkeit und Implementation neuer Features
  * Zeitaufwändig da sehr viele Tests geschrieben werden
  * Großer Code-Overhead
- Beispiele
  * Tic-Tac-Toe: Test zu Korrektheits-Algorithmen (Diagonal, Horizontal, ...)
  * Taschenrechner: Tests für einzelne Funktionalitäten (z.B. Addieren, auch Dinge wie Overflows, etc)

2.3 Objektorientierte Programmierung

- Klassen sind *Baupläne* (vgl. Haus-Bauplan)
- Objekte sind *Instanzen* der Klassen (vgl. fertiges Haus)
- **Sichtbarkeiten**
  - **Public (+)**: Jede Klasse/Methode/... kann darauf zugreifen
  - **Private (-)**: Nur dieselbe Klasse hat Zugriff
  - **Protected (#)**: Dieselbe Klasse und abgeleitete Klassen (mittels Vererbung) haben Zugriff
- **Erzeugung**
  - **Deklaration**: Primitiv `int a;` - Komplex `TolleKlasse b;`
  - **Initialisierung**: Primitiv `a = 42;` - Komplex `b = new TolleKlasse();`
- **Generalisierung**: Enthält die Attribute, die alle Entitäten gemeinsam haben (z.B. abstrakte Klasse `Tier`)
- **Spezialisierung**: Enthält nur die speziell zutreffenden Attribute und erbt von generalisierter Klasse (z.B. `Nashorn : Tier`)

- **Konstruktor**: Wird bei Initialisierung der Klasse aufgerufen. Mit `: base(...)` kann der Konstruktor der Basisklasse (von der geerbt wird) aufgerufen werden
- **Kapselung**: Nutzung der Sichtbarkeiten, um nur bestimmte Daten zugänglich zu machen. Häufig: Alle Attribute private, Zugriff über public getter/setter Methoden
- **Static**
  - Member: Statische Member (Attribute, Methoden) bleiben in jeder Instanz gleich
  - Klasse: Statische Klassen können nicht instanziiert werden (z.B. Console, Math). Alle Member sind ebenfalls static
- **Virtual**: Virtuelle Methoden haben eine Implementation, welche von abgeleiteten Klassen (Vererbung) mit `override` überschrieben werden können
- **Abstract**
  - Klassen: Können nicht instanziiert werden, nur zum Vererben und für Polymorphie (z.B. `new List<Tier>();` mit `Tier` als abstrakte Klasse); kann (muss nicht) abstrakte Methoden enthalten
  - Methoden: Müssen in einer abstrakten Klasse deklariert sein und haben keine Implementation; muss von abgeleiteten Klassen (Vererbung) mit `override` implementiert werden.
- **Interface**
  - Enthält nur Signaturen, keine Implementationen
  - Anwendung häufig ähnlich wie abstrakte Klassen
  - Abgeleitete Klassen müssen **alle** Methoden implementieren (ohne `override`), die im Interface deklariert sind
  - Unterschied zu abstrakten Klassen: Abgeleitete Klassen von abstrakten Klassen müssen nur diejenigen Methoden implementieren (mit `override`), die in der Basisklasse abstract sind
- **Polymorphie** (= „Mehrere Varianten einer Methode“)
  - Overloading: Methoden können überladen werden, indem eine Methode mit gleichem Namen aber unterschiedlicher Signatur geschrieben wird. Beim Aufruf wird dann automatisch diejenige Implementation gewählt, dessen Signatur mit dem Aufruf übereinstimmt
  - Overriding: Geschieht z.B. bei abgeleiteten `abstract` Klassen oder überschriebenen `virtual` Methoden
  - Statisch: Es steht schon bei Compile-Time fest, welche Operation verwendet werden soll (**Early Binding**) - beispielsweise mit Overloading
  - Dynamisch: Es steht erst bei Run-Time fest, welche Operation verwendet werden soll (**Late Binding**) - beispielsweise mit Overriding
- **Properties**
  - `public Typ TolleProperty { get; set; }`
  - `public Typ TolleProperty { get { <getter code> } set { <setter code> }}`
  - <getter code> bzw. <setter code> könnte bspw. auf private Attribute zugreifen
  - get/set optional => Zugriff beschränkbar

  - Vorteil zu standard getter/setter: Weniger Code, übersichtlicher, einheitlich

- **Variablen-Swap**

```
int a = 42, b = 69;

// Langweilig (Dreieckstausch)
int temp = a;
a = b;
b = temp;

// Besser
(a, b) = (b, a);

// 10head
a ^= b ^= a ^= b;
```

2.4 Datenbanken

- **Primärschlüssel** (PS): Eindeutige Identifizierung (z.B. id)
- **Fremdschlüssel** (FS): Zeigt auf einen PS einer anderen Tabelle; verbindet Tabellen
- 1. Normalform: Jedes Attribut ist atomar (unteilbar)
- 2. Normalform: 1. NF && jedes Attribut muss vom ganzen PS abhängig sein
- 3. Normalform: 2. NF && kein Attribut vom Primärschlüssel transitiv abhängig
- **SQL** (Structured Query Language)
  - Mögliche Abfrage: `SELECT DISTINCT {<spalten> (AS) <name>} FROM <tabelle1> INNER JOIN <tabelle2> ON <tabelle1.PS> = <tabelle2.FS> WHERE <bedingung> AND/OR/NOT <bedingung> LIKE '%teil%' GORUP BY <spalte> HAVING <bedingung> ORDER BY <spalte> ASC/DESC`
  - `Funktionen: AVG(), COUNT(), SUM(), MAX(), MIN()`
- Indizierung von Spalten kann Performance-Probleme bei großen Joins beheben