Technisch-Designauswahl:

* Schichtenarchitektur
  + Beschreibung / Skizze
  + Trennung
  + Erklärung der Funktion der einzelnen Schichten
  + Abstraktion zur DB-Schicht
  + Domäne
    - Domänenlogik
  + Controller
    - Use-case bezogen (Domäne)
    - Transaktionen
    - State-Chart anfügen
  + GUI-Controller
    - Datentypprüfung
    - Datentypkonvertierung
  + Keine explizite Business-Schicht (für mehrere Domänen)
* Validierung
  + Gui
* Exceptions
  + Weitergabe bzw. neue pro Ebene
* Interfaces (Subpackages)
* Mapping (Objektpersistenz)
  + Namenskonventionen
  + Dynamische Erzeugung von Objekten
  + Austausch der Persistenz-Schicht relativ einfach (bzw. ohne Änderung der Domänenschicht möglich)
  + Einsatz von Introspektion/Reflexion
  + Begriffe
    - Mapping
    - Objektidentität
    - Broker
    - Cache (ev. Probleme bei gleichzeitiger Manipulation…; oder auch nicht)
  + Wie (indriket vs. Direkt)
  + Probleme mit rekursiver Objekterzeugung
    - Tiefenbeschränkung
    - Markieren
* Controller
  + State-Pattern
* Patterns allgemein
  + Eventuell zu den einzelnen Schichten/Controller hinzufügen
  + Observer
  + Mapper
  + State
  + Proxy
  + Fassade
    - View: Logik kann geändert werden – Fassade und Rest von Programm bleibt gleich
* „Zugriffsschutz“(Information-Hiding): Interfaces, Schichtentrennung
* Transaktionen
  + Steuerung (wer, wie)
  + Verwendung
* Hibernate-Framework
  + Abfragen im …-Stil (nicht native-SQL), weil
    - Abstrakter
    - Optimierter
    - Änderungen ersichtlich
    - Teilweise Überprüfung zur compile-zeit
  + Abstraktion zur DB
    - Austausch der DB relativ leicht möglich (!kein native-SQL)
  + Eindeutige Identifikation der Objekte (ID muss mit hochgezogen werden – Domäne)
  + Vererbung
  + „inverse“ 🡪 verhindert rekursives Laden
* SWING-Framework
  + War Vorgabe (wegen späterer Integration von SW anderer Teams)
* Packagestruktur
  + Begründung
    - Übersichtlichkeit
      * Logisch (pro Schicht)
      * Schichten
      * Information-Hiding
* „Lazy Programming“
  + Hibernate
  + Alter-Berechnung erst on-demand (gehalten wird geb.-Datum)
* Client/Server
  + Fat-Client
    - Trennung bei „Datenbankschicht“ (nur DB auf Server)
      * „Kopplung“ bereits bei Client (Hibernate)
      * Vorteile/Nachteile
        + Skalierung der Applikation auf Rechner der Clients (automatisch)
        + Datenbankabfragen „begrenzt“
    - Blick in die Zukunft: Webinterface (Thin-Client)...

Vollbrecht:

- Designdokumentation:

- Packagediagramm zur Übersicht der Schichtenarchitektur, Packagediagramme einzelner Schichten

- Klassendiagramme

- Sequenzdiagramme, mindestens einen vollen Durchlauf durch alle Schichten.

- Textdokument mit der Beschreibung und Begründung wichtiger Designentscheidungen

# Struktur - Designentscheidung

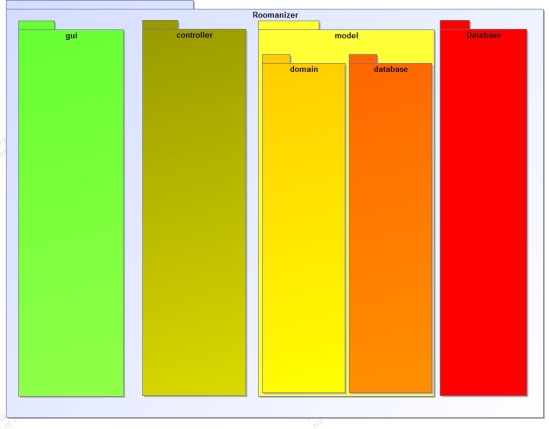
## Schichtenarchitektur

### Überblick

Erklärung des Model-View-Controller Prinzips (ev. Ohne domain/database im Bild)

Sinnhaftigkeit aufgrund…

* Abstraktion zur DB-Schicht
  + Trennung von Funktionalitäten
  + Austauschbarkeit der Schichten
    - Integration anderer Entwicklungen
* „Zugriffsschutz“(Information-Hiding)
* Keine explizite Business-Schicht (für mehrere Domänen)
* Übersichtlichkeit



Klassendiagramme (Teile zu „Packages“)

### GUI

* View/Presentation /*GUI-Controller*
  + Datentypprüfung (Validierung)
  + Datentypkonvertierung
* Desing-Patterns: Observer

### Controller

* Controller
  + Use-case bezogen (Domäne)
  + Transaktionen
  + State-Chart anfügen
* Transaktionen
  + Steuerung (wer, wie)
  + Verwendung
* *State-Pattern (eventuell eigener Punkt: Patterns*

### Model

* Domäne
  + Domänenlogik
  + Domänenfunktionalität
* Mapping
  + Schnittstelle zwischen Domäne und Datenbank
  + Unabhängig von Hibernate und deren Klassen
* Datenbank (Hibernate-Klassen)

### Datenbank

Mysql

Physisch?

*Datenbankmodell ev. Einfügen*

## Mapping

* Namenskonventionen
* Dynamische Erzeugung von Objekten
* Austausch der Persistenz-Schicht relativ einfach (bzw. ohne Änderung der Domänenschicht möglich)
* Einsatz von Introspektion/Reflexion
* Begriffe
  + Mapping
  + Objektidentität
  + Broker
  + Cache (ev. Probleme bei gleichzeitiger Manipulation…; oder auch nicht)
* Wie (indriket vs. Direkt)
* Probleme mit rekursiver Objekterzeugung
  + Tiefenbeschränkung
  + Markieren
  + Überarbeitung in TB2, da keine Zeit und funktioniert
* Desing-Patterns: Mapper, Fassade

## Skalierung

* Client/Server-Architektur
  + Fat-Client
    - Trennung bei „Datenbankschicht“ (nur DB auf Server)
      * „Kopplung“ bereits bei Client (Hibernate)
      * Vorteile/Nachteile
        + Skalierung der Applikation auf Rechner der Clients (automatisch)
        + Datenbankabfragen „begrenzt“
    - Blick in die Zukunft: Webinterface (Thin-Client)...

## Hibernate-Framework

* Hibernate-Framework
  + Abfragen im Criteria-Stil (nicht native-SQL), weil
    - Abstrakter
    - Optimierter
    - Änderungen ersichtlich
    - Teilweise Überprüfung zur compile-zeit
  + Abstraktion zur DB
    - Austausch der DB relativ leicht möglich (!kein native-SQL)
  + Eindeutige Identifikation der Objekte (ID muss mit hochgezogen werden – Domäne)
  + Vererbung relativ leicht auf relationaleDB abbildbar
  + „inverse“ 🡪 verhindert rekursives Laden
* Design-Patterns: Proxy (Funktionalität wird nicht intensiv genützt 🡪 internes Mapping)
  + Lazy Programming“
    - Hibernate
  + Alter-Berechnung erst on-demand (gehalten wird geb.-Datum)

## SWING-Framework

* War Vorgabe (wegen späterer Integration von SW anderer Teams)