Designentscheidung

Roomanizer

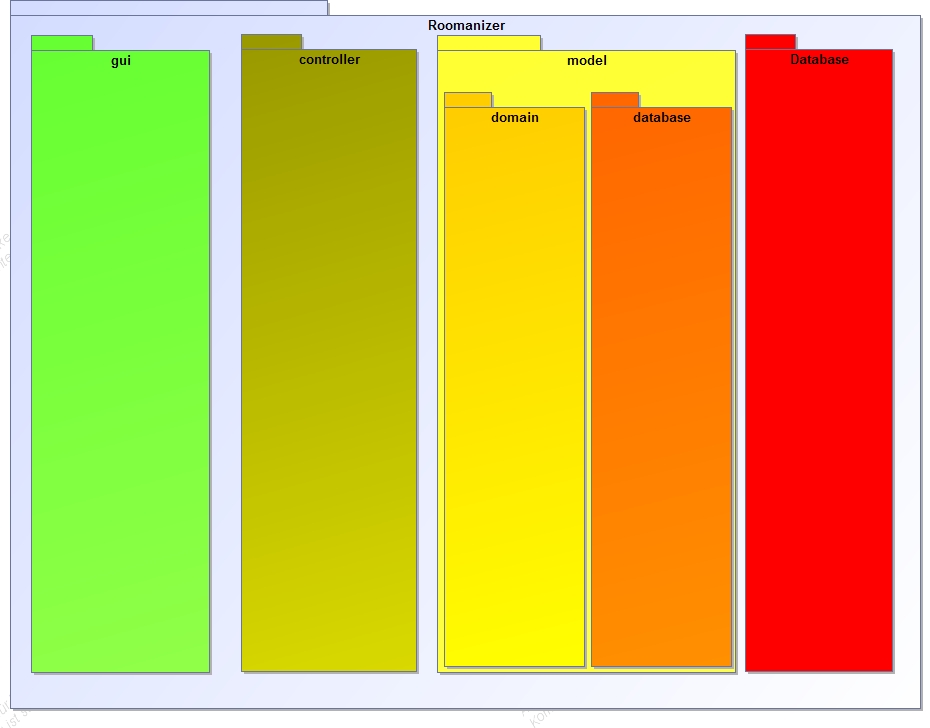
# Einleitung

Für das Projekt Roomanizer haben wir uns viele Gedanken über die grundsätzliche Programmstruktur gemacht. Für die endgültige Architektur war das Domänenmodell maßgebend, da dieses die realen Anforderungen sehr gut abbildet. Diese Erkenntnis gewannen wir aus dem Requirements-Workshop mit unserem Kunden Herrn Paul Tavolato und den fortlaufenden Sitzungen mit externen Coaches.   
Von dieser Basis aus entwickelten wir teamübergreifend ein Datenbankmodell, das uns bei späteren parallelen Entwicklungen die Integration verschiedener Programmteile erleichtern soll. Auf die Datenbank aufbauend haben wir uns beim Mapping der Domänenobjekte, also den abstrakten Komponenten eines Hotels, in die relationale Datenbank für das bereits existierende Framework Hibernate entschieden. Diese Aufgabe ist nicht Domänenspezifisch, weshalb es dort eine relativ große Auswahl an bestehenden Produkten gibt. Hibernate nimmt uns viel Arbeit in diesem Bereich ab, unterstützt uns in komplexen Abläufen und wir erreichen auch eine höhere Unabhängigkeit gegenüber verschiedenen Datenbankprodukten.  
Die Applikation ist nach dem Model-View-Controller-Prinzip aufgebaut, wobei noch verschiedene Zwischenschichten für eine bessere Struktur, Ordnung und Kontrolle mitberücksichtigt worden sind. Die einzelnen Schichten der Client-Server Applikation werden im Anschluss detailliert erläutert.  
Aufgrund einer impliziten Skalierung der Rechenleistung auf den Rechnern der Endbenutzer haben wir uns für einen Fat-Client entschieden. Das heißt, dass der Server nur für die zentrale Datenhaltung verantwortlich ist, wobei bei Client die eigentliche Anwendung abläuft.

# Schichtenarchitektur

## Überblick

Wie bereits einleitend erwähnt ist das Roomanizer – Hotel Software Projekt nach dem Model-View-Controller Prinzips aufgebaut. Dieses Design gibt eine klare Einteilung der Aufgaben vor, wodurch die Übersichtlichkeit gefördert wird, der Zugriffsschutz innerhalb des Programms klar und umsetzbar wird und ein relativ einfache/r Austausch bzw. Erweiterung der einzelnen Schichten möglich wird.



Das Paketdiagramm zeigt uns neben der Präsentationsschicht (View) , dem Controller …

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)
* Reconnect aufgrund mysql-„Überlauf“

## SWING-Framework

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