Designentscheidung

Roomanizer

# Einleitung

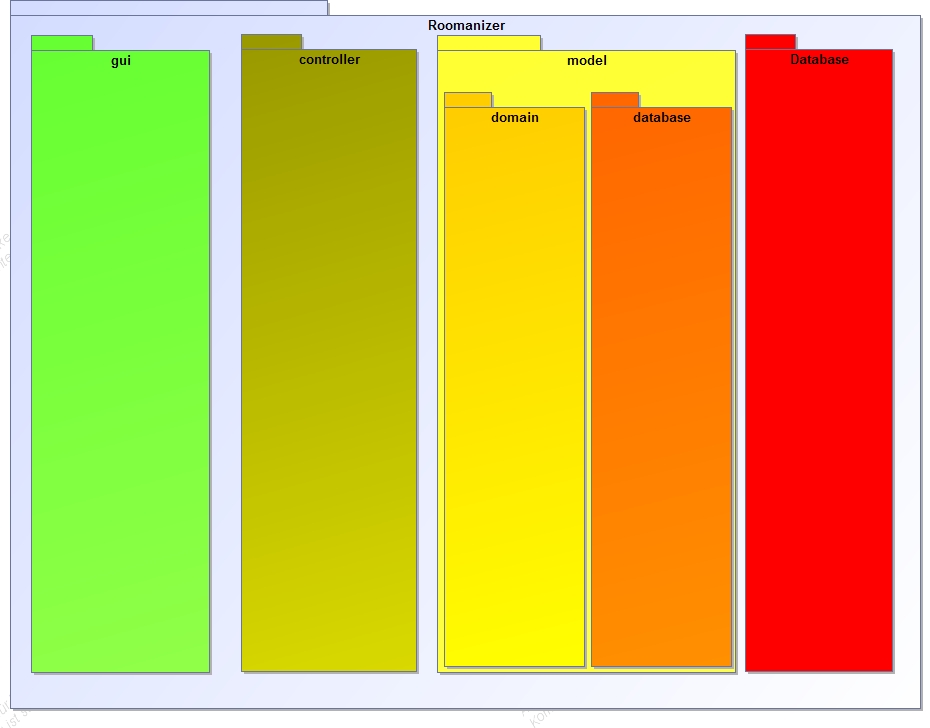
Für das Projekt Roomanizer haben wir uns viele Gedanken über die grundsätzliche Programmstruktur gemacht. Für die endgültige Architektur war das Domänenmodell maßgebend, da dieses die realen Anforderungen sehr gut abbildet. Diese Erkenntnis gewannen wir aus dem Requirements-Workshop mit unserem Kunden 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.

<<eventuell Skizze der Hauptkomponenten: Userstation, Server, Datenbank>>

# 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.

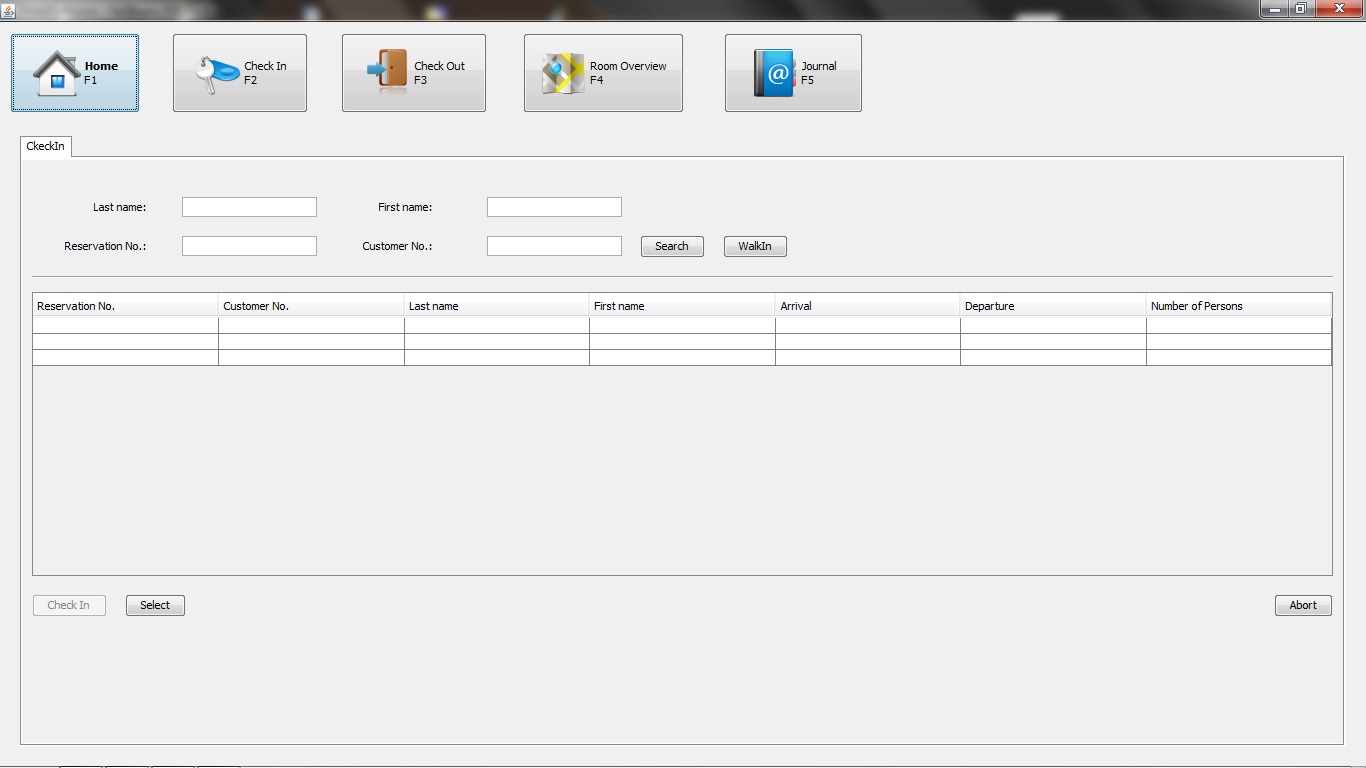


<<aktualisieren>>

Das Paketdiagramm zeigt uns neben der Präsentationsschicht (View), den Controller, welcher für die Koordination und den Domänenspezifischen Ablauf zuständig ist, und das Model, welches die Domänenklassen beinhaltet. Des Weiteren existiert für die Umwandlung von Datenbank- bzw. Hibernate-Objekten in Domänenobjekte eine Mapping-Schicht. Diese garantiert eine vollständige Entkopplung von einerseits notwendigen fachspezifischen Programmteilen und vom Objekt-Relationalen Mapping Framework teilweise vorgenerierten und abhängigen Hibernate-Klassen.  
Durch diese sehr wichtige Trennung erreichen wir volle Unabhängigkeit von einer Bibliothek, die wir möglicherweise in Zukunft adaptieren, erneuern oder sogar ersetzen wollen.  
Die Datenbank ist wie bereits angedeutet nicht objektorientiert (OO), sondern entspricht der klassischen relationalen Architektur. Obwohl einige Zwischenschritte durch eine OO-Datenbank nicht angefallen wären, haben wir uns im Team für die diese Variante und das Produkt MySQL entschieden. Es gibt in diesem Markt eine größere Auswahl und durch die Abstraktion von Hibernate berührt uns das Problem der Materialisierung und der Dematerialisierung nur am Rande.

## Graphical User Interface – GUI

Die grafische Benutzeroberfläche entspricht der View- oder auch der Präsentations-Schicht und hat die Aufgabe, mit dem Anwender zu kommunizieren. Für diesen Programmteil gab es die Vorgabe, das Framework SWING zu verwenden, welches uns schon sehr viele Standardkomponenten zur Verfügung stellt und das somit eine große Hilfestellung ist. Mit dieser Einschränkung fiel für uns die Auswahl weg und wir konnten uns voll und ganz auf eine optimale Usability konzentrieren. In dieser Angelegenheit bekamen wir große Unterstützung von Experten wie Karl-Heinz Weidmann und Philipp von Hellberg. Anhand von Prototypen und deren Diskussion konnten wir die Software immer wieder verbessern und kamen schlussendlich auf eine besonders nutzerzentrierte Lösung. Wichtige Features für diesen Erfolg sind etwa die gute Übersichtlichkeit trotz des recht komplexen Use case „Check-in“, welcher in zwei Schritten durchgeführt werden kann, aber auch die Shortcuts für häufig verwendete Funktionen unterstützen den Anwender. Bei der Analyse stellte sich heraus, dass die Unterstützung von Tastenkürzel ein Muss für Rezeptionisten ist, da Sie parallel zur Programminteraktion die primäre Aufgabe des Kundenkontakt inne haben und diesen auch so gut wie möglich aufrecht erhalten sollten.  
Programmintern ist die Präsentationsschicht, genauer gesagt der GUI-Controller, für simple Datentypvalidierungen sowie Datentypkonvertierungen verantwortlich. Die eigentliche Logik liegt jedoch voll und ganz in der Controller-Ebene und obliegt den verschieden Use case Steuerelementen, sodass auch diese Schicht problemlos adaptiert werden kann. Der Einsatz eines des doch oft gebrauchten Design-Patterns – dem Observer – war für den Check-in nicht notwendig, weshalb wir auch darauf verzichten konnten. Die Kommunikation findet durch einfache Methodenaufrufe von der Seite des GUI beim Controller statt, der dann implizit eine Rückmeldung gibt. Anders wäre es beispielsweise bei sogenannten Nicht-Ereignissen, wenn etwa ein Gast nicht zum abgemachten Zeitpunkt auftaucht und dadurch eine Meldung an den Rezeptionist gegeben werden müsste. Dieser ereignisorientierte Datenaustausch reduziert gegenüber einem ständigen Nachfragen beim Controller den Datenverkehr und ist darüber hinaus auch übersichtlicher.



<<aktualisieren>>

<<eventuell noch Ergänzungen zum Swing-Framework

* Läuft auf jedem Rechner mit Java
* Gleiches Aussehen 🡪 unabhängig von OS
  + Im Gegensatz zu AWT
  + Gleiche Usability
* Guter Support
* Laufend Erneuerungen
  + Bsp. Date-Time-Picker>>

## Controller

Die Steuerung der Domänenlogik übernimmt der Controller; das heißt, er steuert den Ablauf der Use cases. In dieser Timebox haben wir den Controller für den Check-in implementiert, welcher wie alle anderen Use case Controller für die verschiedensten Aufgaben verantwortlich ist:  
Erstens muss er die verschiedenen Zustände, die ein sogenanntes Szenario beinhaltet, koordinieren und gewährleisten, dass keine Übertretungen o.Ä. auftreten. Zweitens führt er die Arbeitsschritte, welche durch den Benutzer angestoßen werden, aus oder delegiert diese weiter an die Model-Objekte. Für die Präsentationsschicht kann der Controller also als eine Fassade für das restliche Programm gesehen werden, da das der einzige reguläre Weg für den Datenaustausch ist. Drittens muss er bei Fehlern in der Verarbeitung oder sonstigen Abweichungen die Fehlerbehandlung durchführen – dazu gehört das Suchen von Alternativen oder die Ausgabe von Benutzergerechten Fehlermeldungen. Die vierte Aufgabe ist Halten von neuen, temporären Daten wie es etwa beim Anlegen eines Gastes passiert. Nicht zuletzt ist es dem Controller auch gestattet auf andere seiner Art zu verweisen, wodurch er indirekt für die Abfolge der Use cases untereinander mitverantwortlich ist. Zum Abschluss ist noch die Transaktionskontrolle als eine seiner Hauptaufgaben zu nennen. Wie wir wissen gibt es mehrere Zustände, die aber nicht immer konsistent sind. Am Ende eines kompletten Vorgangs wird alles dematerialisierte commitet – also dauerhaft persistent gemacht. Falls es dort Probleme geben sollte oder der Benutzer inzwischen den Vorgang abbricht, kann alles rückgängig gemacht werde.

<<Details Stefan>>

<<Klassendiagramm – Controller-Schicht>>

## 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“

## Model

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

## 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)...