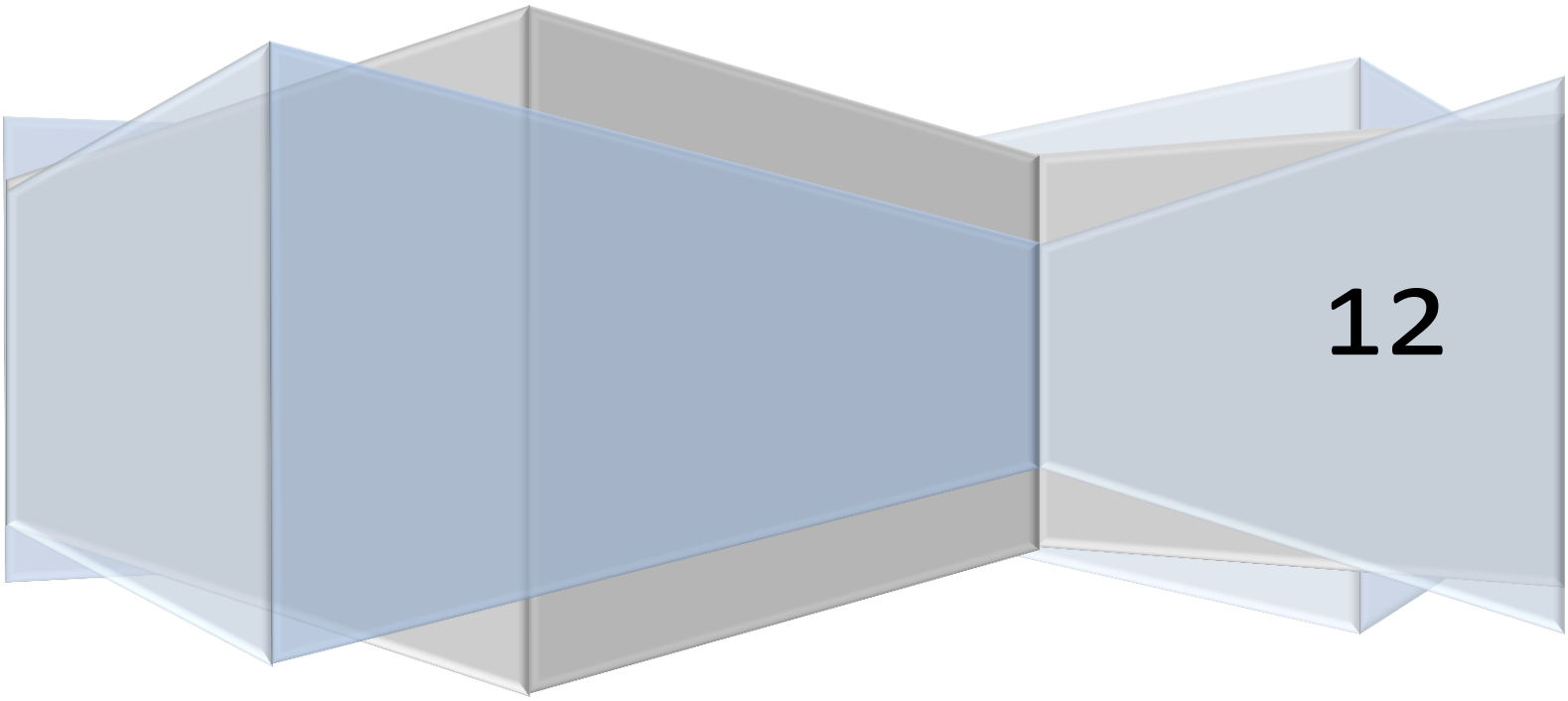
Stefan Dunst, Christian Lins, Tobias Meusburger, Markus Mohanty, Hubert Rall, Johannes Schwendinger

Roomanizer

Designentscheidung



12

Roomanizer

Version 1.4

Datum	Version	Beschreibung	Autor
21.04.2012	1.0	Datei angelegt und Grundgerüst erstellt	Christian Lins
27.04.2012	1.1	Allgemeine Inhalte verfasst	Christian Lins
29.04.2012	1.2	Controller-Details beschrieben	Stefan Dunst
29.04.2012	1.3	Benutzer-Schnittstelle Details ergänzt	Johannes Schwendinger
30.04.2012	1.4	Inhalte korrigiert und angepasst	Team E

Inhaltsverzeichnis

Designentscheidung

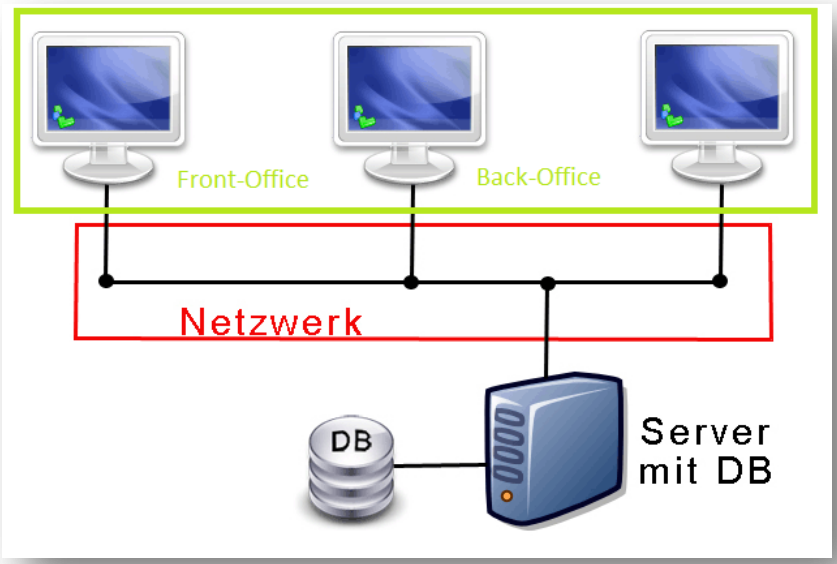
Software-Projekt Hotel (Roomanizer)

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 Klassenobjekte (abstrakte Komponenten eines Hotels) in die relationale Datenbank für das 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-Konzept 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 beim Client die eigentliche Anwendung abläuft.

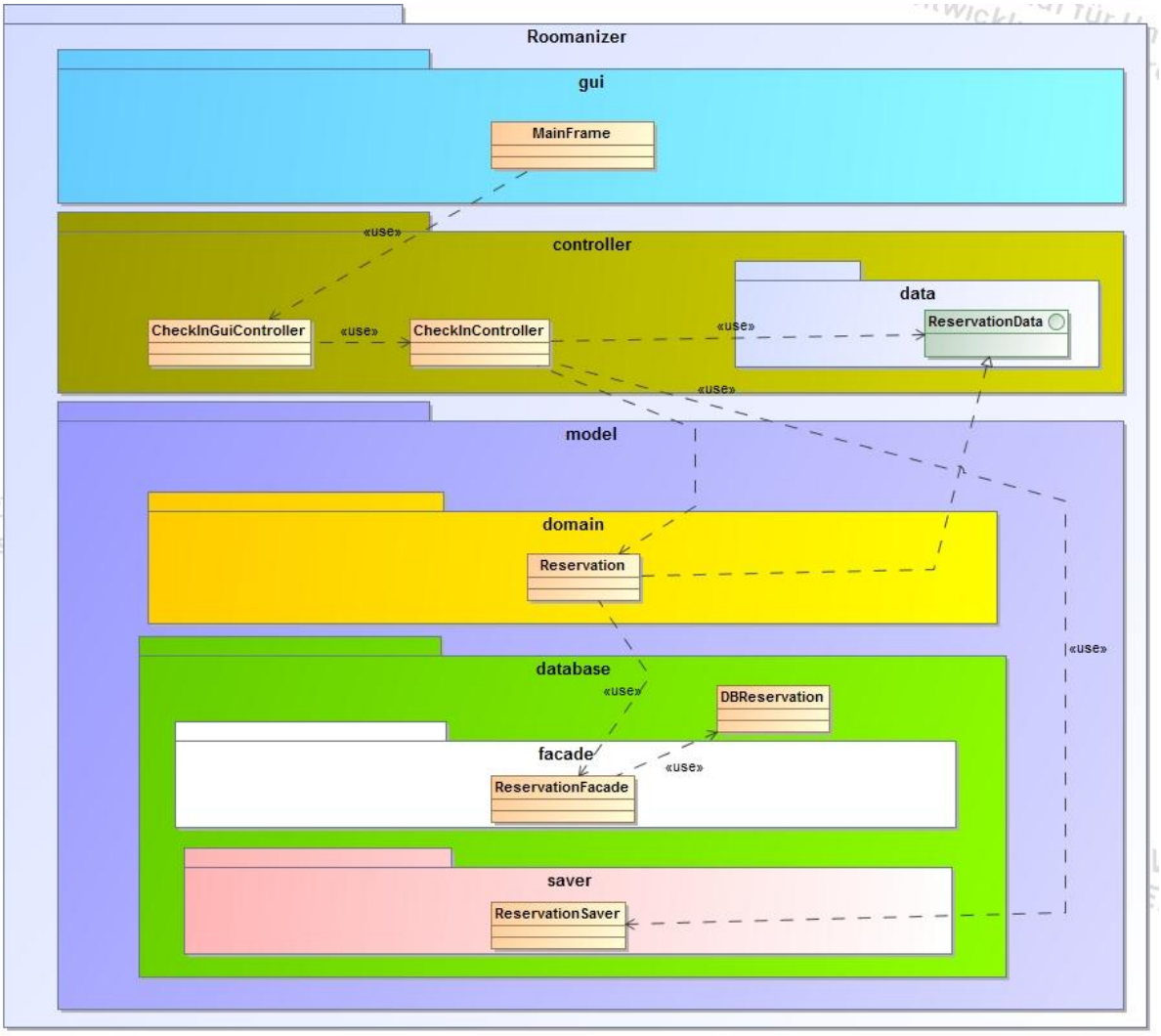
Schematische Darstellung der Applikation:



Schichtenarchitektur

Überblick

Wie bereits einleitend erwähnt ist das Software-Projekt Hotel (Roomanizer) nach dem Model-View-Controller-Konzept 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 einfacher Austausch bzw. eine simple Erweiterung der einzelnen Schichten möglich wird.

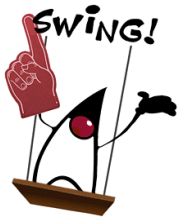


Das Paketdiagramm zeigt uns neben der Präsentationsschicht (Package: „gui“), den Controller, welcher für die Koordination und den Domänenspezifischen Ablauf zuständig ist, und das Modell (Package: „model“), welches die Domänenklassen beinhaltet. Des Weiteren existiert für die Umwandlung von Datenbank- bzw. Hibernate-Objekten in Domänenobjekte eine externe Mapping-Schicht. Diese garantiert eine vollständige Entkopplung von einerseits notwendigen fachspezifischen Programmteilen (Package: „domain“) und von teilweise generierten Hibernate-Klassen (Package: „database“).
Durch diese sehr wichtige Trennung erreichen wir volle Unabhängigkeit vom Mapping-Framework (Hibernate), welches wir dadurch in Zukunft adaptieren, erneuern oder sogar ersetzen könnten.
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 berühren uns die Probleme der Materialisierung, der Dematerialisierung und des Mappings nur am Rande.

Benutzer-Schnittstelle (GUI)

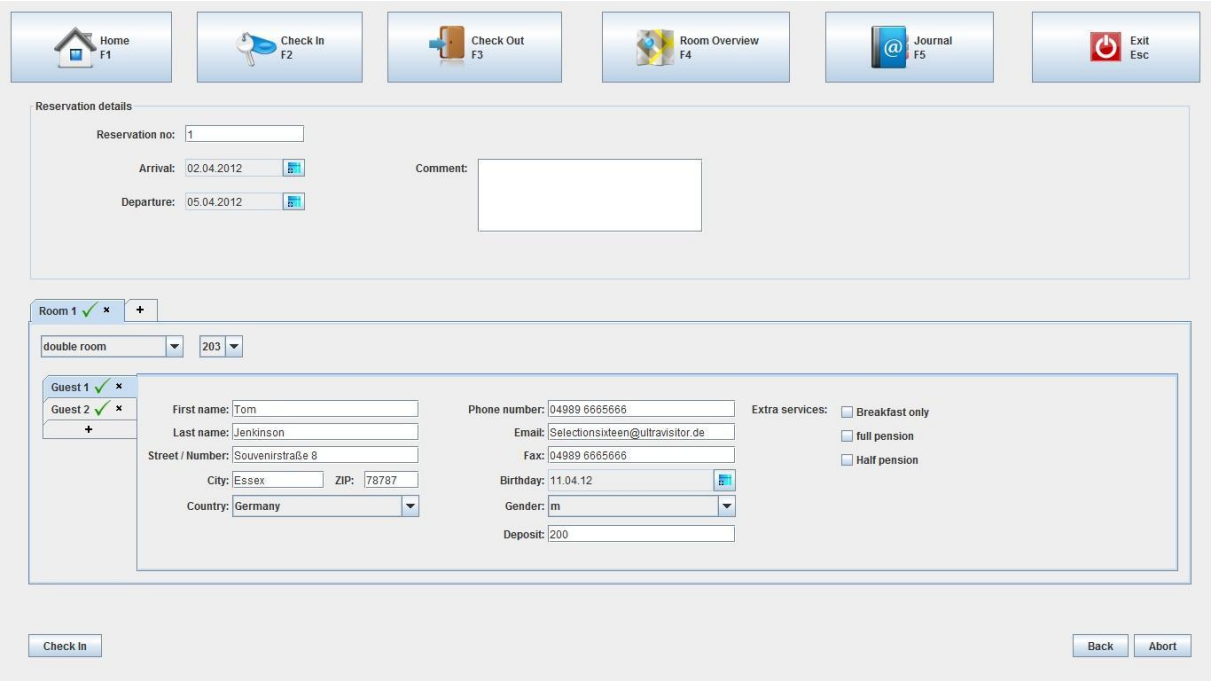
Allgemein

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 uns somit eine große Hilfestellung ist. Des Weiteren gibt es gegenüber AWT keine Probleme bei der Verwendung von unterschiedlichen Betriebssystemen mit der einheitlichen Darstellung. Mit der Einschränkung „SWING“ fiel für uns das Auswahlverfahren weg und wir konnten uns voll und ganz auf optimierte Usability konzentrieren.



Usability

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 wenigen Schritten durchgeführt werden kann, aber auch die Shortcuts für häufig verwendete Funktionen unterstützen den Anwender. Schon 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 Kundenkontakts inne haben und diesen auch so gut wie möglich aufrecht erhalten sollten. Besonderen Fokus legten wir auch auf sinnvolle Warn- und Fehlermeldungen: Pop-ups mit Warnmeldungen verwendeten wir beispielsweise nur, wenn inmitten eines komplexen Use cases, abgebrochen wird, sodass sich der Benutzer seine Eingabe nochmals überdenken kann und seine Arbeit nicht verloren geht.



Kommunikation

Programmintern ist die Präsentationsschicht, genauer gesagt der GUI-Controller, für simple Datentypvalidierungen sowie Datentypkonvertierungen verantwortlich. Die eigentliche Logik liegt jedoch komplett im Use case-Controller und obliegt den verschieden Use case Steuerelementen, sodass auch diese Schicht problemlos adaptiert werden kann. Der Einsatz des doch oft verwendeten „Observer-Pattern“ war für diesen Programmteil nicht notwendig, weshalb wir auch darauf verzichten konnten. Die Kommunikation findet durch einfache Methodenaufrufe von der Seite des GUIs 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.

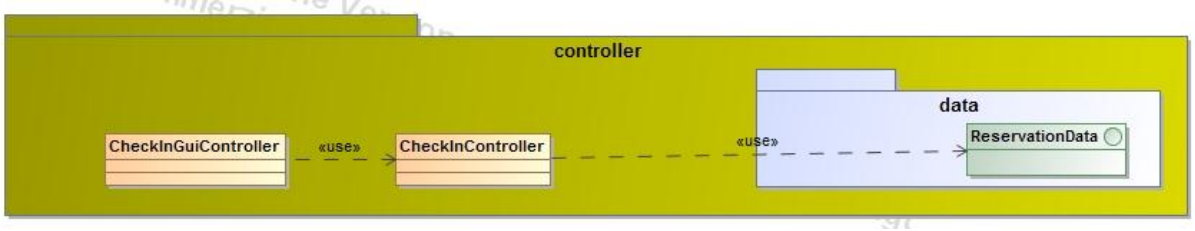
Layout

In unserem GUI kommen verschiedene Standard Layout Manager von Swing zum Einsatz. Neben dem simplen Flow-Layout für die Tab-Komponenten und dem standardmäßigen Group-Layout, das vom Netbeans GUI-Designer verwendet wird, benutzen wir für das Hauptpanel das Card-Layout. Dieses erlaubt uns ein einfaches Vor- und Zurücknavigieren innerhalb eines Use cases. Das Hauptpanel ist in unserem sogenannten „MainFrame“ (Rahmen), welcher das Hauptmenü mit den verschiedenen Links zu den interessantesten Use cases beinhaltet, untergebracht. In dieses Panel kommen nun alle anderen GUI Elemente, je nachdem, was zur Laufzeit benötigt wird.

Controller

Allgemein

Die Steuerung der Domänenlogik übernimmt der Controller; das heißt, er steuert den Ablauf der Use cases. In der ersten Timebox haben wir den Check-in implementiert. Dieser schaut wie folgt aus:

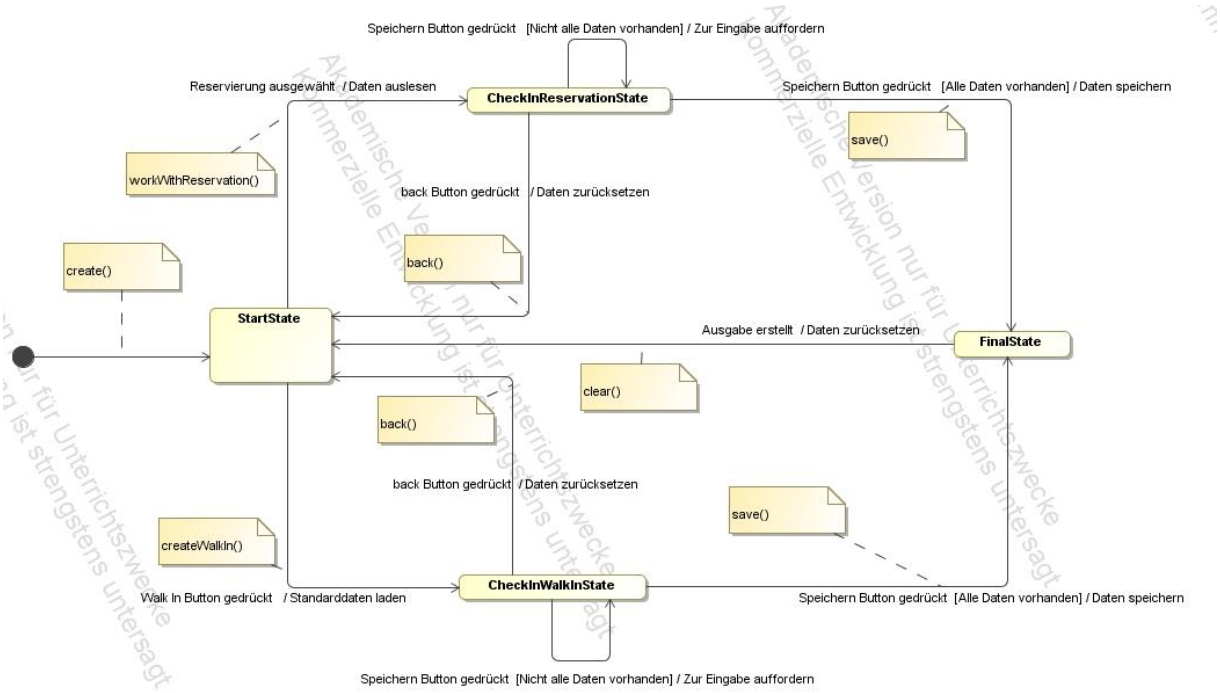


In dem Paket-Ausschnitt sehen wir, wie der GUI-Controller seine Anfragen an den Check-in-Controller stellt. Dieser bearbeitet seine Anfragen, gibt dabei nur Daten mit Lesezugriff zurück (spezielle Interfaces: z.B. „ReservationData“) und steuert den internen Ablauf wie im Anschluss beschrieben.

Aufgaben

Die Aufgaben eines Use case-Controller sind, wie im Anschluss beschrieben, sehr umfangreich:
Erstens muss er die unterschiedlichen Zustände, die ein sogenanntes Szenario beinhaltet, koordinieren und gewährleisten, damit keine Übertretungen oder Ähnliches auftreten („State-Pattern“).
Zweitens führt er die Arbeitsschritte, welche durch den Benutzer angestoßen werden, aus oder delegiert diese weiter an die Model-Komponenten. Von der Perspektive der Präsentationsschicht kann der Controller somit als eine Fassade für das restliche Programm gesehen werden, da er der einzige reguläre Vermittler für den Datenaustausch ist. Wichtig ist es auch zu erwähnen, dass er nur Daten-Interfaces zum GUI, welche nur „Getter“ beinhalten, weiter gibt, da wir dadurch die Manipulation der Daten unterbinden. Des Weiteren haben wir im Zuge dessen keine expliziten Modell-Objekte auf der GUI-Ebene.
Drittens muss der Controller 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 das 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 und deren Verschachtelung 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 „committed“ – also dauerhaft persistent gemacht. Falls es dort Probleme geben sollte oder der Benutzer den Vorgang abbricht, kann dieser rückgängig („Rollback“) gemacht werden.

„State-Pattern“



Um die verschiedenen Zustände des Controllers abzubilden haben wir das „State-Pattern“ eingesetzt. Der Check-in-Controller hält dabei ein Status-Objekt. Die abstrakte Implementierung hält alle möglichen Methoden, wirft dabei aber nur eine „IllegalStateException“. Die konkreten Status-Klassen überschreiben die Methoden, die für den aktuellen Status relevant sind, indem sie korrekt implementieren werden. Die „IllegalStateExceptions“ sind dabei Hilfen bei der Entwicklung und sollten bei korrekter Verwendung des Controllers zur Laufzeit nicht mehr ausgelöst werden.
Das Status Objekt hält immer eine Referenz auf den Controller, wodurch es ermöglicht wird, dass die einzelnen Status‘ sich selbst austauschen können und zwischengespeicherte Daten auch beim Übergang von einem Zustand in den anderen erhalten werden können, wenn sie im Controller gespeichert werden.

„Singleton-Pattern“

Da jederzeit nur ein Check-in-Vorgang bei einer Installation des Programms erfolgen kann, haben wir ein „Singleton-Pattern“ für den Check-in-Controller eingesetzt. Dadurch wird gewährleistet, dass nur ein Controller zu einem bestimmten Zeitpunkt verfügbar sein kann und gleichzeitig können wir, anders als bei statischer Implementierung der Klassen, objektorientiert programmieren.
Man kann also mit dem Vererbungskonzept arbeiten und die Methoden auf Objekten und nicht auf Klassen aufrufen, was eine eventuelle Umstellung auf mehrere Instanzen merklich vereinfachen würde.

Hibernate Bibliothek

Allgemein

Das Hibernate Framework dient der Persistierung von Domänenobjekten in einer relationalen Datenbank. Die Entscheidung für dieses Produkt geschah im Kollektiv und wurde durch folgende Merkmale herbeigeführt: Hibernate ist Open-Source, wird ständig von einem engagierten Team (JBoss) weiterentwickelt und erfüllt die von uns geforderte Funktionalität hinsichtlich Mapping und Transaktion. Es fallen somit keine Lizenzgebühren an, die beispielsweise für die meisten objektorientierten Datenbanken anfallen würden, und die Bibliothek wird regelmäßig dem Stand der Entwicklung auf diesem Gebiet angepasst.

Mapping

Die Konfiguration des Mappings wird von unserer Seite über Annotations durchgeführt, wobei alternativ auch spezielle XML-Files eingesetzt werden könnten. Externe Files sind erfahrungsgemäß langsamer und da wir nicht direkt mit den Hibernate Klassen arbeiten und uns die Annotations dadurch in keiner Weise behindern, haben wir uns für diese Variante entschieden.
Der eigentliche Ablauf des Mappings wird von Hibernate im Hintergrund erledigt und ist nicht Teil unseres Aufgabenbereichs.

Transaktionen

Transaktionen werden ebenfalls von der Bibliothek verwaltet. Das bedeutet konkret, dass unser Check-in-Controller eine Hibernate-Session eröffnet, um einen vollständigen Ablauf durchzuführen und diesen zu persistieren. Aber besonders wichtig ist es, dass bei Fehlschlägen oder einem Abbruch ein sogenannter „Rollback“ durchgeführt werden kann, um in den letzten konsistenten Zustand zu gelangen.

Vorteile

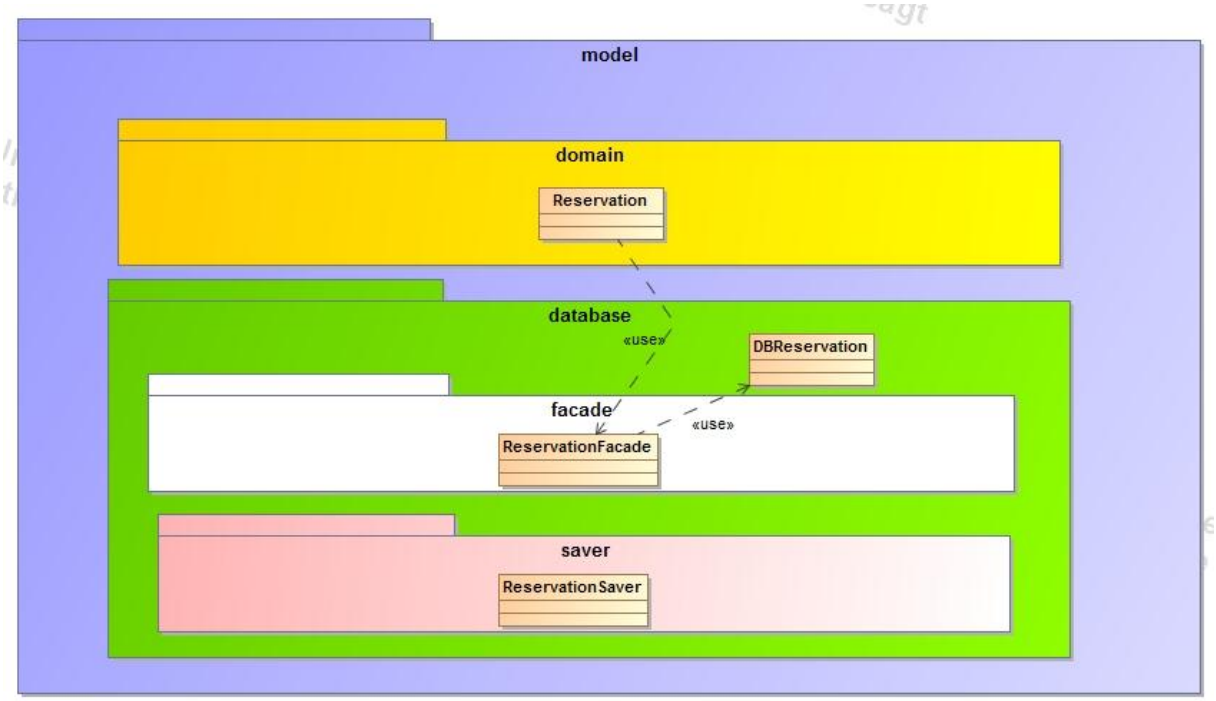
Verschiedene funktionale Eigenschaften wie Materialisieren, Dematerialisieren und Caching werden im Hintergrund von Hibernate erledigt und benötigen von unserer Seite keine Aufmerksamkeit. Des Weiteren ist das Konzept der Vererbung von Java besonders gut in das Framework integriert und es müssen auch dort keine Erweiterungen von uns implementiert werden.
Was für Vorteile bietet das Framework noch? Wir erreichen eine große Abstraktion zu der Datenbank wodurch ein Austausch jeglicher relationalen Datenbankprodukte problemlos möglich ist. Das einzige was zu machen ist, das ist der Austausch des Treibers, der im Normalfall vom Hersteller der Datenbank zur Verfügung gestellt wird. Dabei ist es auch wichtig, kein Nativ-SQL zu verwenden, da die Dialekte vom Standard immer wieder abweichen. Die HQL (Hibernate Query Language) oder auch der „Criteria“-Stil passen die Abfragen im Hintergrund auf die Dialekte an. Mit den Kriterien wird der Code kompakt und der Ablauf ist im Sinne der Objektorientiertheit eleganter. Des Weiteren werden die „Queries“ optimiert und einige Fehler können bereits zur Kompilierzeit erkannt werden, was beim reinen SQL nicht möglich ist.
Das „lazy-fetching“ – Laden, wenn etwas wirklich benötigt wird (Proxy-Pattern) - wird von uns nicht genützt. Wir materialisieren die Objekte sofort und vollständig („Eager-fetching“), wodurch wir einen großen Teil der Datenbank im Speicher haben. Das ist auf unser Mapping von der Hibernate- auf die

Domänenschicht zurückzuführen, da wir dort immer ganze Objekte instanziieren und keine Proxys verwenden.
Durch diese Vorgehensweise benötigt der Programmstart zwar etwas mehr Zeit. Die Daten werden dafür zu Beginn in den Cache geladen, womit in weiterer Folge ein schnellerer Datenzugriff ermöglicht wird. Dieser zusätzliche Aufwand ist für uns kein Problem, da das Programm normalerweise im Dauereinsatz ist und deshalb nicht oft neu gestartet werden muss.
Externe Änderungen in der Datenbank werden von Hibernate trotzdem erkannt und bei Bedarf in den Cache geladen.

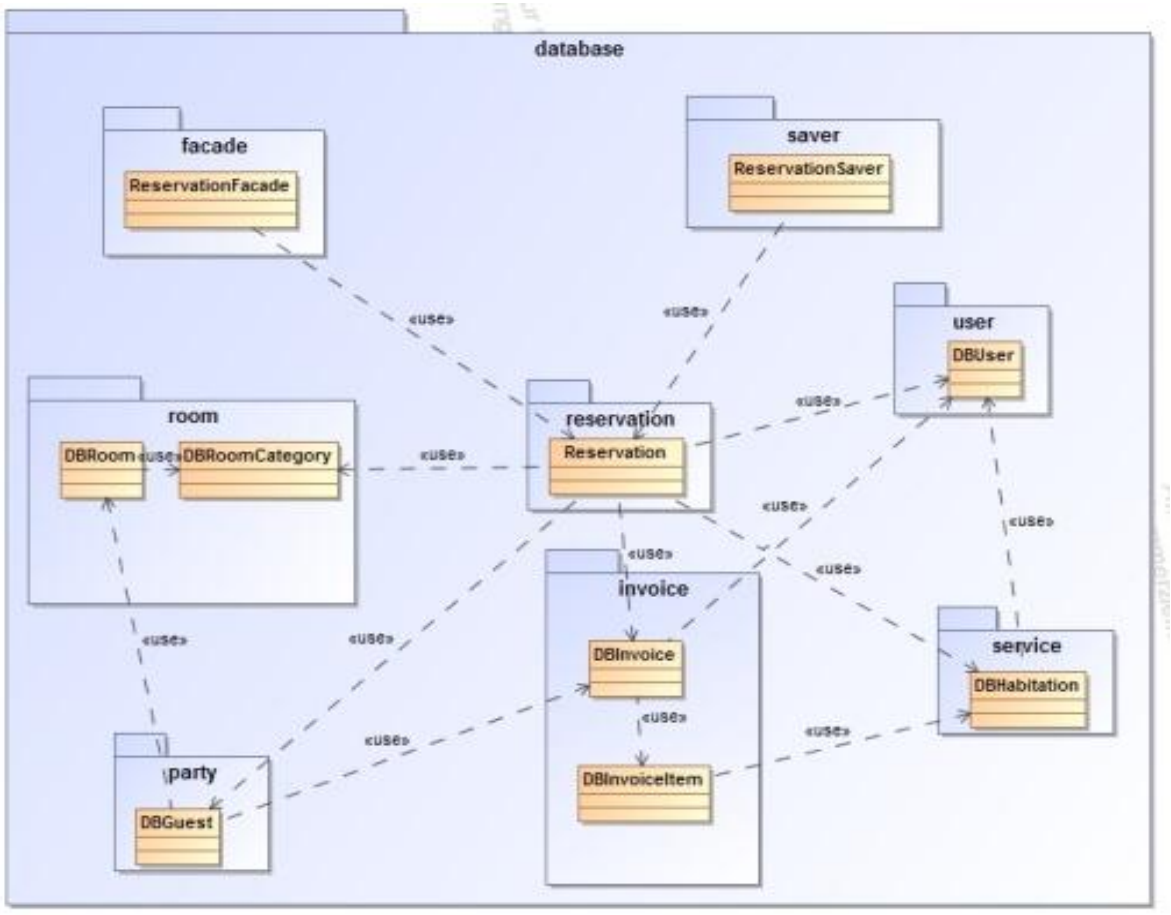
Modell

Allgemein

Unser Modell beinhaltet hauptsächlich Domänenklassen, die für die hotelspezifische Funktionalität verantwortlich sind. In diesen Klassen wird beispielsweise jegliche Manipulation von Reservierungen zur Verfügung gestellt, die weiterführend etwa das Erstellen von Optionen erledigt. Wenn der Controller eine ganze Reihe von Reservierungen oder eine neue Instanz benötigt, dann läuft diese Abfrage ebenfalls über die Klasse „Reservation“. Somit gibt es in diesem Fall keine Sprünge über die einzelnen Schichten hinweg und wir können problemlos eine Schicht austauschen, ohne im ganzen Programm Änderungen vornehmen zu müssen.



Wie bereits einleitend erwähnt liegen im Paket „database“ die Klassen, die direkt von Hibernate verarbeitet werden. Diese müssen den Framework-Anforderungen genügen und weisen einige Einschränkungen und Abhängigkeiten auf, weshalb wir, wie bereits erwähnt, eigene Arbeiterklassen („domain-Package“) eingeführt haben. Damit hält sich der Aufwand bei einem Framework-Wechsel in Grenzen.



Fassade

Das Datenbankpaket bietet Fassaden an, über die etwa eine offene Rechnung angefordert werden kann. Diese Klassen im Hintergrund (z.B. „DBReservation“) sind dabei die letzte Instanz vor dem Hibernate Framework und der physischen Datenbank. Die Vorteile dieses Design-Pattern („Facade-Pattern“) liegen in einer noch besseren Strukturierung: Es gibt nach außen zu den anderen Packages nur eine Anlaufstelle, die die Komplexität versteckt und auch die Kopplung weiter löst. Mit der Fassade wird also das eigentliche System versteckt und der Kommunikationskanal ist klar definiert. Ein daraus resultierender Nutzen ist, dass das Subsystem sehr leicht erweitert werden kann, ohne dass in darüber liegenden Klassen die Aufrufe etc. angepasst werden müssen.

Persistierung

Die Speicherung bzw. Dematerialisierung wird über das Subpaket „saver“ durchgeführt. Der Check-in-Controller kann beispielsweise die Aufenthalte über den „ServiceSaver“ in die Datenbank schreiben. Die einzelnen „Saver“ stellen somit die Umkehrung der Fassaden-Klassen dar.

Fassaden und Saver sind jeweils als Singletons realisiert, da sie beide keine verschiedenen Zustände besitzen, sondern nur Schnittstellen in Form von Methoden zwischen den Schichten zur Verfügung stellen.

„Dynamic Mapper“

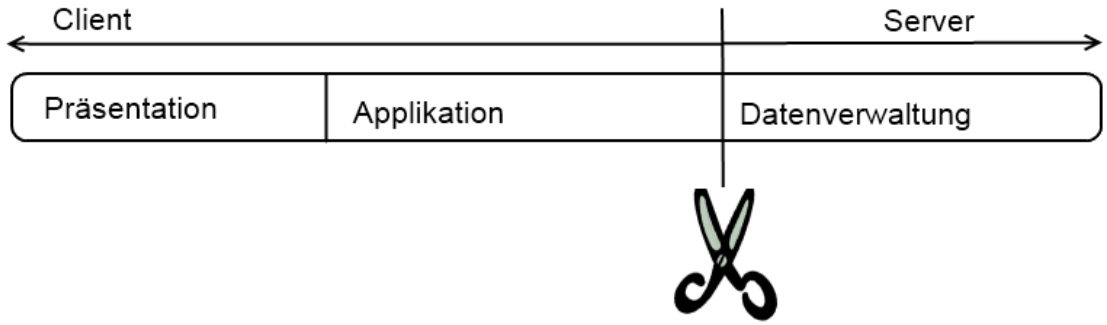
Um zwischen den Objekten des Domänenpakets und den Objekten des Datenbankpakets zu mappen haben wir einen eigenen „Dynamic Mapper“ geschrieben, der dynamisch zwischen den beiden Schichten vermittelt. Diese Programmkomponente ist also eine Schnittstelle und schafft eine völlige Unabhängigkeit für beide Seiten und die Persistenzschicht kann problemlos ausgetauscht werden.

Die Idee für dieses Vorhaben entstand in einer Besprechung über die Schichtentrennung und das Klassendesign der Software, aber auch unser technische Coach, Herr Wolfgang Auer, unterstützte diese strikte Entkopplung und gab uns entsprechenden Input – etwa bei den Themen Introspektion und Reflexion.
Besondere Maßnahmen beim Programmieren, etwa die strenge Namenskonvention, mussten über die ganze Entwicklungszeit rigoros eingehalten werden, um eine dynamisches Mapping für alle möglichen Objekte der Model-Schicht zu ermöglichen. Dies ist notwendig, da wir über die Metadaten wie etwa Klassennamen, Methodennamen und Attribute, die Struktur analysieren und dementsprechend weiter vorgehen. Dadurch erfahren wir alle nötigen Informationen, um in beiden Richtungen zwischen Domäne- und Datenbank-Schicht abzubilden. Mapping bedeutet in diesem Fall also, dass wir verwandte Objekte erzeugen. Diese sind teilweise sehr komplex, da sie verschiedene andere Objekte und Kollektionen halten, die in sich wieder Referenzen aufweisen. Kleine Probleme gab es, wenn wir genau durch die gerade genannte Thematik auf das ursprüngliche Objekt gelangen: Es entsteht ein graphentheoretischer Kreis, den wir explizit unterbrechen müssen. Wir haben uns für eine Behelfslösung entschieden, die die rekursive Tiefe beschränkt. Wenn die Ressourcen ausreichen, dann werden wir an diese Stelle mit unserem Team und den technischen Coaches eine optimale Lösung suchen und implementieren.

Um den „Dynamic Mapper“ zu implementieren verwenden wir, wie bereits erwähnt, Introspektion oder auch Reflektion. Das bedeutet, dass ein Programm seine eigene Struktur kennt und diese, wenn nötig, modifizieren kann.
In unserem Fall bekommt der „Dynamic Mapper“ die Information, was für ein Objekt zu mappen ist, und sucht sich damit laut unseren Namenskonventionen die zum Objekt verwandte Klasse. Ein Beispiel für unsere Namenskonvention wäre, falls das Objekt vom Typ „DBGuest“ ist, die dazu verwandte Klasse „Guest“.
Hat nun also der „Dynamic Mapper“ das „Urobjekt“ und die dazugehörige verwandte Klasse, erzeugt er sich ein neues Objekt des Typs, in den er mappen soll. Anschließend sucht er sich zu dem eben erzeugten Objekt alle Methoden, die mit „set“ beginnen. Für jede dieser Methode wird überprüft, ob es eine identische „get“-Methode auf der anderen Seite (Klasse des ursprünglichen Objektes) gibt und führt diese aus. Sollte das so erhaltene Sub-Objekt (Attribut des zum „Urobjekt“ verwandten Objekts) einer Instanz einer Klasse der abzubildenden Seite entsprechen, so beginnt die ganze Prozedur von vorne. Hier befinden wir uns nun an der Stelle, an der die bereits erwähnte Problematik mit dem graphentheoretischen Kreis auftritt. Und zwar ist das der Fall, wenn das neu erhaltene Objekt wieder ein Objekt hält das vom ersten Typ ist. So entsteht eine indirekte Rekursion, die durch eine Tiefenbegrenzung umgangen wird. Die Abbruchbedingung der Rekursion tritt ein, wenn keine „set“-Methoden mehr gefunden werden.

Skalierung

Wir setzten einen Fat-Client ein, der maßgeblich zu der guten Skalierbarkeit der Anwendung beiträgt. Das heißt, dass beim Computer des Anwenders der Großteil der Berechnungen ausgeführt wird. Somit kann das System auch bei mehreren Rezeptionisten, die parallel an verschiedenen Schaltern tätig sind, die Performance halten, da die Rechenleistung durch die Geräte vor Ort erhöht wird. Das hat einen großen Vorteil, da das Serversystem relativ einfach gehalten werden kann – also keine komplexe Lastverteilung implementiert werden muss. Der Server dient bei unserer derzeitigen Applikationen der reinen globalen Datenhaltung, und hat keine Objekt-Mapping- oder Domänenfunktionalität implementiert. Die Datenbank muss, da sie auf dem Server liegt, alle Anfragen bearbeiten. Sie ist für diesen Zweck schnell genug; wichtig ist jedoch, dass die Netzwerkverbindung schnell ist, um die Daten in adäquater Zeit zu transportieren.
Für verschiedene Weiterentwicklungen, etwa einem Webinterface (Thin-Client) muss der Server hardwaretechnisch aber auch softwaretechnisch angepasst werden. Die Domänenlogik muss dabei auf den Server adaptiert werden; sie kann aber wiederverwendet werden.



Datenbank

Allgemein

Beim Hotelsoftwareprojekt haben wir uns bezüglich der Datenbank für das Open-Source-Produkt MySQl entschieden. Gegenüber proprietären Produkten wie Oracle fallen bei uns also keine Lizenzkosten für die Datenbank an. Des Weiteren ist die Installation und Wartung sehr einfach, wodurch wir uns viel Zeit und Geld sparen. Hinter MySQL steht eine große Entwicklergemeinschaft, die das Datenbanksystem ständig weiterentwickelt. Da dieses relationale Datenbanksystem am weitesten verbreitet ist, ist es auch relativ leicht geeignete Mitarbeiter zu finden, um die Hotel-Datenbank zu pflegen.

Datenbankmodell

Das Datenbankmodell wurde von den Teamleitern der drei kooperierenden Teams gemeinsam entworfen, um eventuellen Kompatibilitätsproblemen bei der Integration der Programmteile der anderen Teams vorzubeugen.
Vererbungshierarchien wurden mittels Joins zwischen den Primary Keys der Klassen der Vererbungshierarchie umgesetzt, da diese Lösung am nächsten an der tatsächlichen Umsetzung im Domänenmodell liegt und gleichzeitig von Hibernate unterstützt wird.
Andere Methoden hätten die Nachteile von redundanten Spalten, was in einer schlechten Wartbarkeit enden würde, oder Einträgen mit vielen leeren Feldern, was bei Datenbanken mit einem schlechten „NULL-Handling“ einen enormen Performanceverlust zur Folge hätte, mit sich gebracht.

Grundlegend unterteilt werden kann das Datenbankmodell – genauso wie in der Domänenebene - in die Teile Service, Invoice, User, Party, Room und Reservation. Teilweise existieren reine Auflösungstabellen. Diese werden in Hibernate aber direkt über eine „Many-to-Many-Relation“ gelöst und sind deshalb nicht im Programm aufgeführt.