Designentscheidung

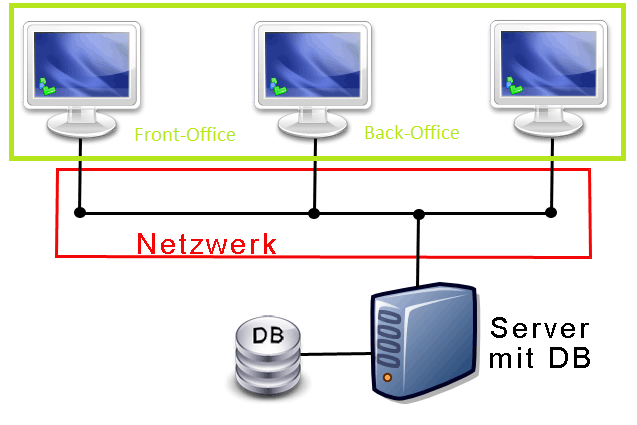
Roomanizer

# 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 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 bei Client die eigentliche Anwendung abläuft.

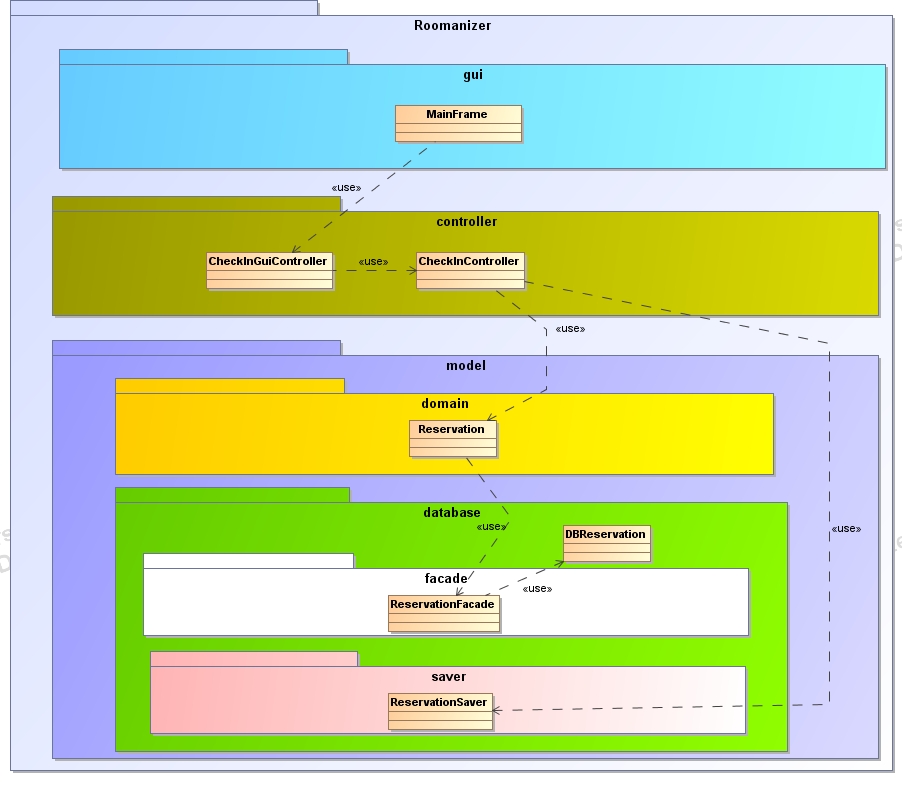
*Schematische Darstellung der Applikation:*



# Schichtenarchitektur

## Überblick

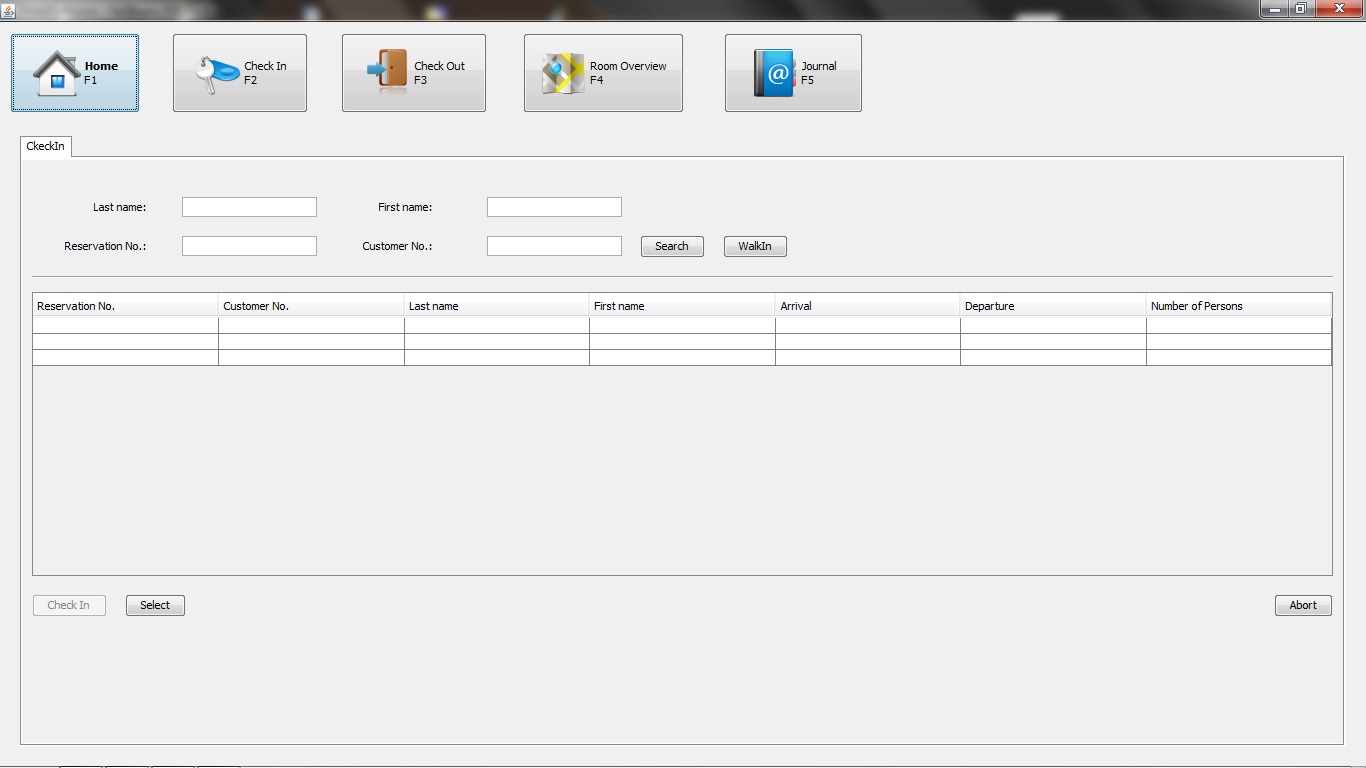
Wie bereits einleitend erwähnt ist das Software-Projekt Hotel (Roomanizer) nach dem Model-View-Controller-Konzept aufgebaut. Dieses Design gibt eine klare Einteilung der Aufgaben vor, wodurch die Übersichtlichkeit gefördert wird, der Zugriffsschutz innerhalb des Programms klar und umsetzbar wird und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



Das Paketdiagramm zeigt uns neben der Präsentationsschicht (gui), 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 externe Mapping-Schicht. Diese garantiert eine vollständige Entkopplung von einerseits notwendigen fachspezifischen Programmteilen (domain) und von teilweise generierten Hibernate-Klassen (database).  
Durch diese sehr wichtige Trennung erreichen wir volle Unabhängigkeit von einer Bibliothek (Hibernate), 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 berühren uns die Probleme der Materialisierung, der Dematerialisierung und des Mappings 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 uns somit eine große Hilfestellung ist. Mit dieser Einschränkung fiel für uns das Auswahlverfahren 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 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 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 komplett in der Controller-Ebene und obliegt den verschieden Use case Steuerelementen, sodass auch diese Schicht problemlos adaptiert werden kann. Der Einsatz des doch oft gebrauchten Design-Patterns, dem Observer, war für diesen Programmteil 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 in diesem Fall 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
* Java native Librarys (überall lauffähig)
  + >>

## 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 unterschiedlichen Zustände, die ein sogenanntes Szenario beinhaltet, koordinieren und gewährleisten, dass keine Übertretungen oder Ähnliches auftreten. 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.   
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 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 commitet – also dauerhaft persistent gemacht. Falls es dort Probleme geben sollte oder der Benutzer den Vorgang abbricht, kann dieser rückgängig („rollback“) gemacht werde.

<<Details Stefan: Desing-Pattern (State) und technische Ergänzungen>>

<<State-Chart einfügen>>

<<Klassendiagramm – Controller-Schicht abstrakt (nicht überladen!!! Oder weglassen)>>

## Hibernate Framework

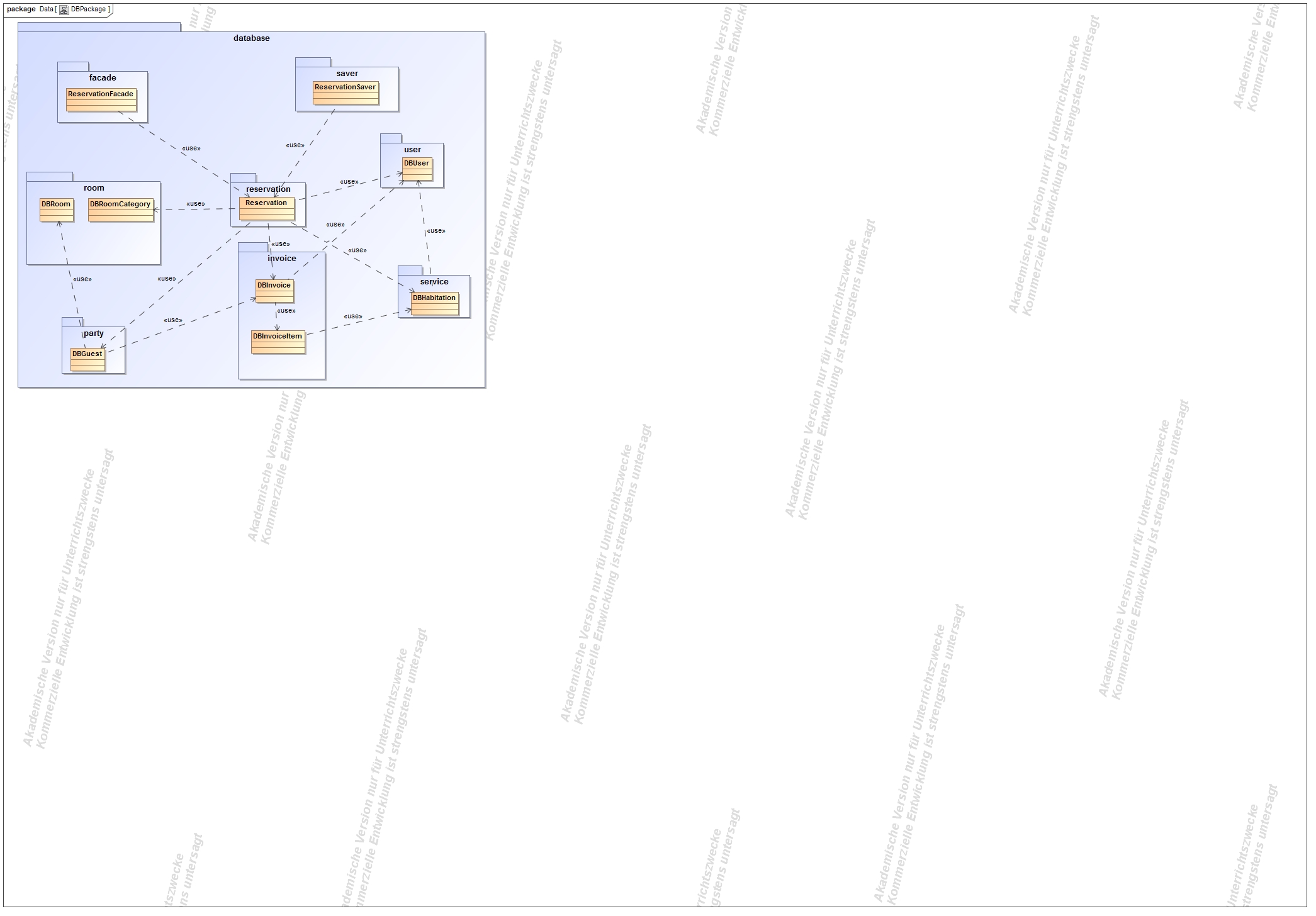
Das Hibernate Framework dient der Persistierung von Domänenobjekten in einer relationalen Datenbank. Die Entscheidung für dieses Produkt geschah im Kollektiv und wurde durch folgende Merkmale herbeigeführt: Hibernate ist Open-Source, wird ständig von einem engagierten Team (JBoss) weiterentwickelt und erfüllt die von uns geforderte Funktionalität hinsichtlich Mapping und Transaktion. Es fallen somit jegliche Lizenzgebühren weg, die beispielsweise für die meisten Objektorientierte Datenbanken anfallen würden, und die Bibliothek wird regelmäßig dem Stand der Entwicklung auf diesem Gebiet angepasst.   
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 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. 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. Das ist aber nicht alles. 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. Diese Option wird, sofern die Ressourcen ausreichen, noch adaptiert, um den Datenfluss von der Datenbank zum Client und die Speicherbelastung zu verringern sowie die Ladezeiten stark zu reduzieren.



## Model

Unser Modell beinhaltet hauptsächlich Domänenklassen, die hauptverantwortlich für die hotelspezifische Funktionalität sind. In diesen Klassen wird beispielsweise jegliche Manipulation von Reservierungen zur Verfügung gestellt, die weiterführend 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) eingeführt haben. Damit schließen wir größere Probleme bei einem modulhaften Framework-Wechsel aus.  
Für das Domänenpaket bietet das Datenbankpaket Fassaden an, über die etwa eine offene Rechnung angefordert werden kann. Diese Klassen sind die letzte Instanz vor dem Hibernate Framework und der physischen Datenbank.  
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.



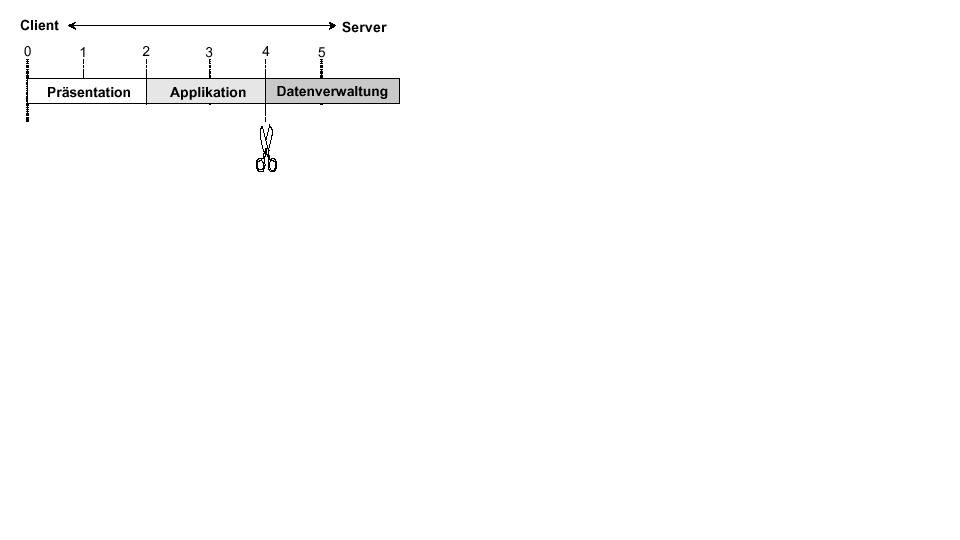
Um zwischen den Objekten des Domänenpakets und den Objekten des Datenbankpakets zu mappen haben wir einen eigenen „DynamicMapper“ geschrieben, der dynamisch zwischen den beiden Paketen abbildet. Diese Programmkomponente ist also eine Schnittstelle und schafft eine völlige Unabhängigkeit für beide Seiten 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 beim Thema Introspektion.  
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. Wie funktioniert unser „DynamicMapper“ im Detail? Diese Frage wird im folgenden Kapitel beantwortet.

### „Dynamic Mapper“

Um den „DynamicMapper“ 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 „DynamicMapper“ 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 „Guest“-Klasse.   
Hat nun also der „DynamicMapper“ 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 tiefenbegrenzt ist, was wir über einen zusätzlichen Methodenparameter „Rekursionstiefe“ erreichen. Die Abbruchbedingung der Rekursion tritt ein, wenn keine „set“-Methoden mehr gefunden werden.

## Skalierung

Zur guten Skalierbarkeit der Anwendung trägt hauptsächlich unsere Client-Server-Architektur bei. Wir setzten einen Fat-Client ein. 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 implizit 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.



<<anpassen>>

## Datenbank

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.

