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

12

**Roomanizer**

**Designentscheidung**

**Roomanizer**

**Version 2.3**

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |
| 15.05.2012 | 2.0 | Dokument für Timebox 2 anpassen | Stefan Dunst |
| 20.05.2012 | 2.1 | Änderungen Persistenzschicht (Manager anstelle von Saver/Fassaden) | Christian Lins |
| 23.05.2012 | 2.2 | Kapitel Mapper überarbeiten | Johannes Schwendinger |
| 24.05.2012 | 2.3 | Bilder aktualisiert und Inhalte korrigiert | Team E |

Inhaltsverzeichnis

[Einleitung 3](#_Toc325656861)

[Schichtenarchitektur 4](#_Toc325656862)

[Überblick 4](#_Toc325656863)

[Allgemein 6](#_Toc325656864)

[Usability 6](#_Toc325656865)

[Kommunikation 7](#_Toc325656866)

[Layout 7](#_Toc325656867)

[Controller 8](#_Toc325656868)

[Allgemein 8](#_Toc325656869)

[Aufgaben 8](#_Toc325656870)

[„State-Pattern“ 9](#_Toc325656871)

[„Singleton-Pattern“ 9](#_Toc325656872)

[Hibernate Bibliothek 10](#_Toc325656873)

[Allgemein 10](#_Toc325656874)

[Mapping 10](#_Toc325656875)

[Transaktionen 10](#_Toc325656876)

[Vorteile 10](#_Toc325656877)

[Modell 12](#_Toc325656878)

[Allgemein 12](#_Toc325656879)

[Persistierung 13](#_Toc325656880)

[„Dynamic Mapper“ 13](#_Toc325656881)

[Skalierung 15](#_Toc325656882)

[Datenbank 16](#_Toc325656883)

[Allgemein 16](#_Toc325656884)

[Datenbankmodell 16](#_Toc325656885)

[Adaptierung 17](#_Toc325656886)

[Adapter für Programmteile von Team F 17](#_Toc325656887)

[GUI Adaptierung 18](#_Toc325656888)

Designentscheidung

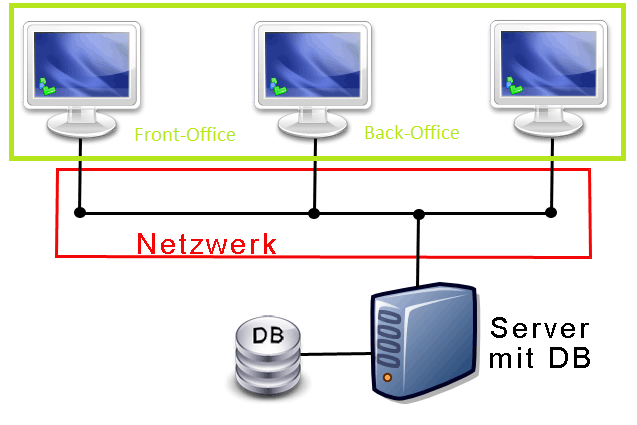
Software-Projekt Hotel (Roomanizer)

## 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 object-relational mapping (ORM) 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.

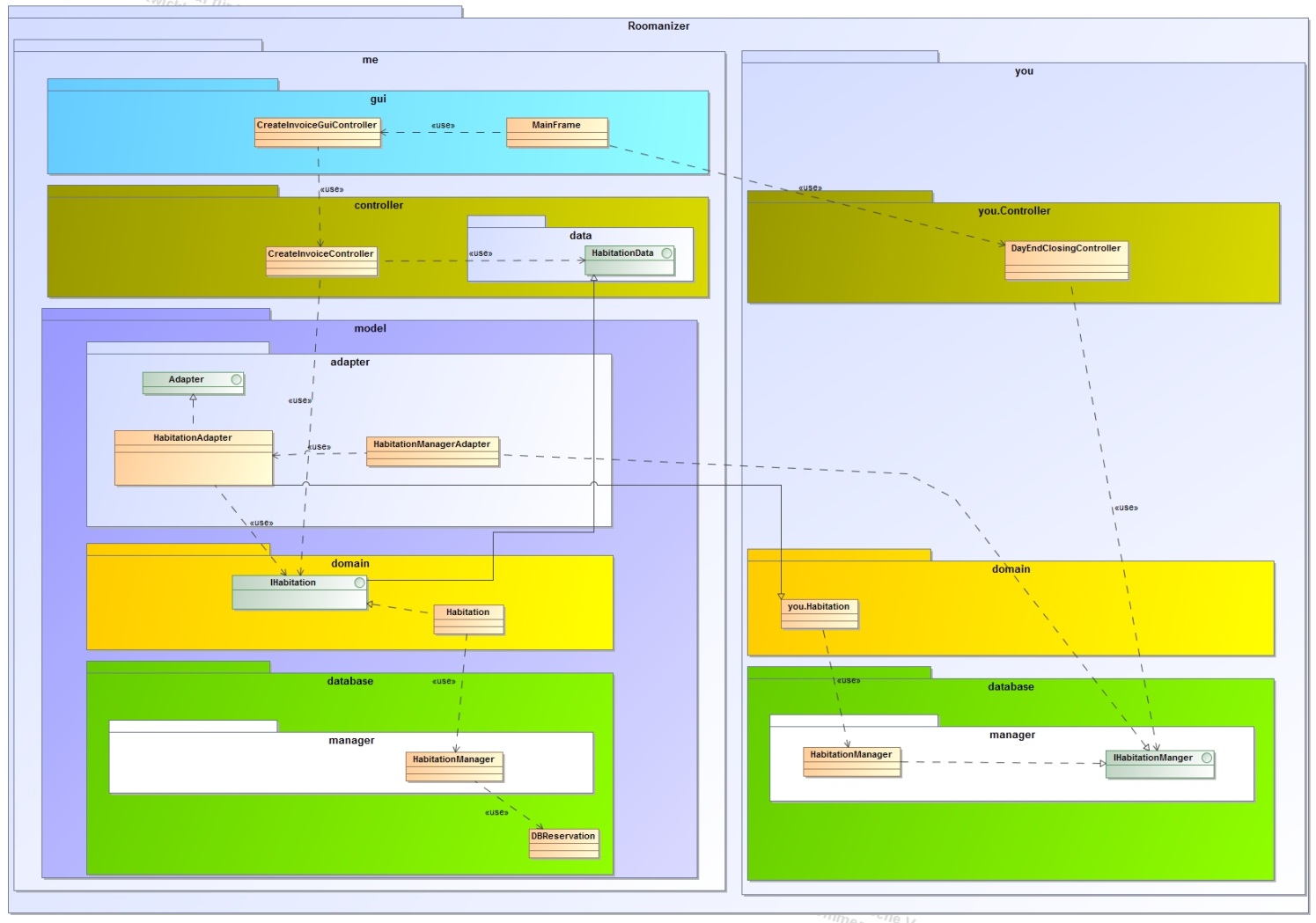
*Schematische Darstellung der Applikation:*



# 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.



*Hochauflösendes Bild beiliegend*

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. Ein positiver Nebeneffekt dieser Trennung ist das wir somit mit Klassen ohne Annotations arbeiten können. Dies erhöht die Übersichtlichkeit des Codes erheblich und es können keine zufälligen Fehler an den Annotations gemacht werden. Zu guter Letzt haben wir mit unserer Domänenschicht den Vorteil, dass wir nicht auf unseren Datenbankobjekten arbeiten müssen und somit vollkommen unabhängig von der Datenbankschicht sind die nur als Datenhalter verwendet wird.

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 entschieden und aus dem großen Angebot am Markt das Produkt MySQL, das populärste Open-Source-Datenbankverwaltungssystem der Welt, ausgewählt. Durch die Abstraktion von Hibernate berühren uns die Probleme der Materialisierung, der Dematerialisierung und des Mappings nur am Rande.

Benutzer-Schnittstelle (GUI)

### Allgemein

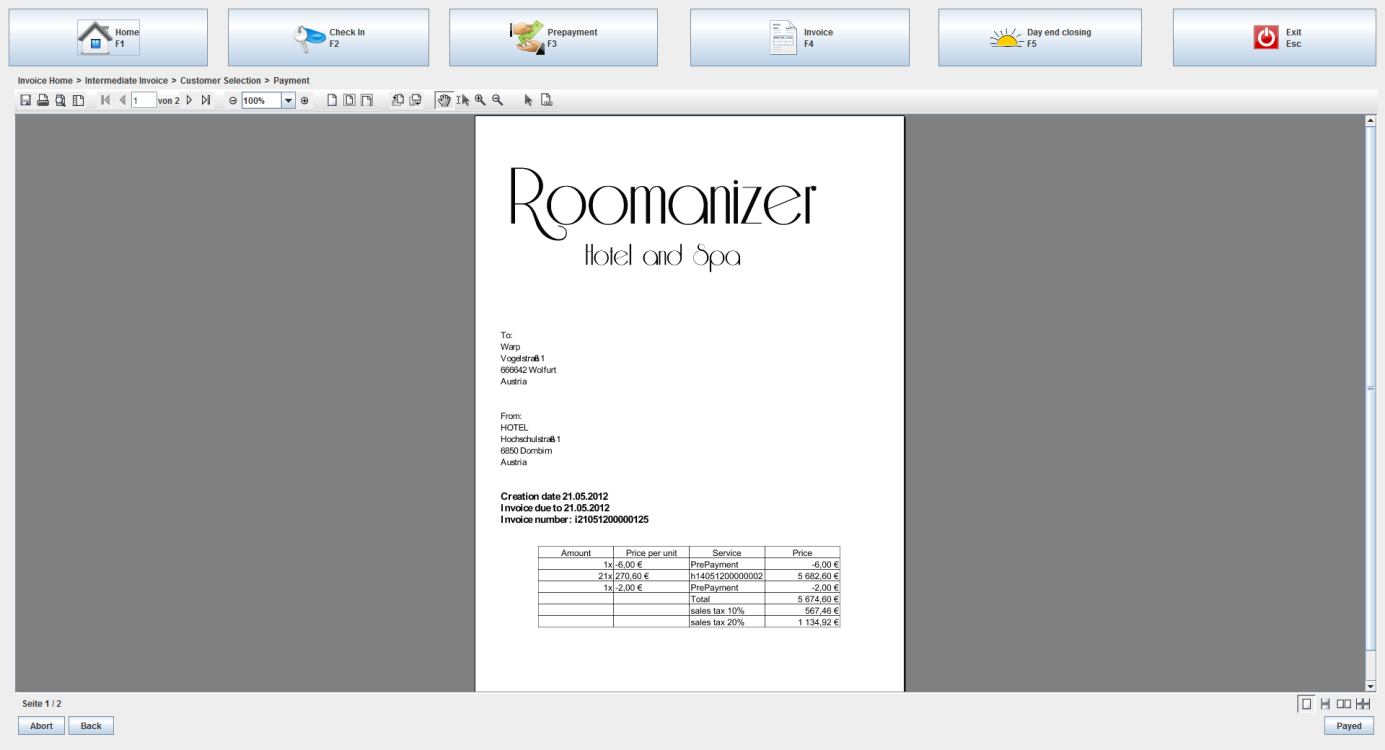
Die grafische Benutzeroberfläche entspricht der View- oder auch der Präsentations-Schicht und hat die Aufgabe, mit dem Anwender zu kommunizieren. Für diesen Programmteil gab es die Vorgabe, das Framework SWING zu verwenden, welches uns schon sehr viele Standardkomponenten zur Verfügung stellt und uns 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.

Durch die Integration des Programmteils von Team F, welches das SWT-GUI-Framework verwendet, findet sich in unserer Applikation auch ein einzelnes SWT Fenster welches separat aufgerufen wird.



### 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, Datentypkonvertierungen sowie die Verwaltung von darstellungsspezifischen Operationen die nichts mit der eigentlichen Programmlogik zu tun haben 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.

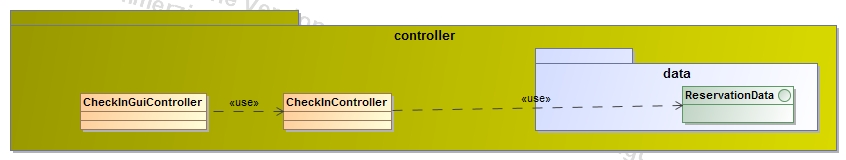
### Layout

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:

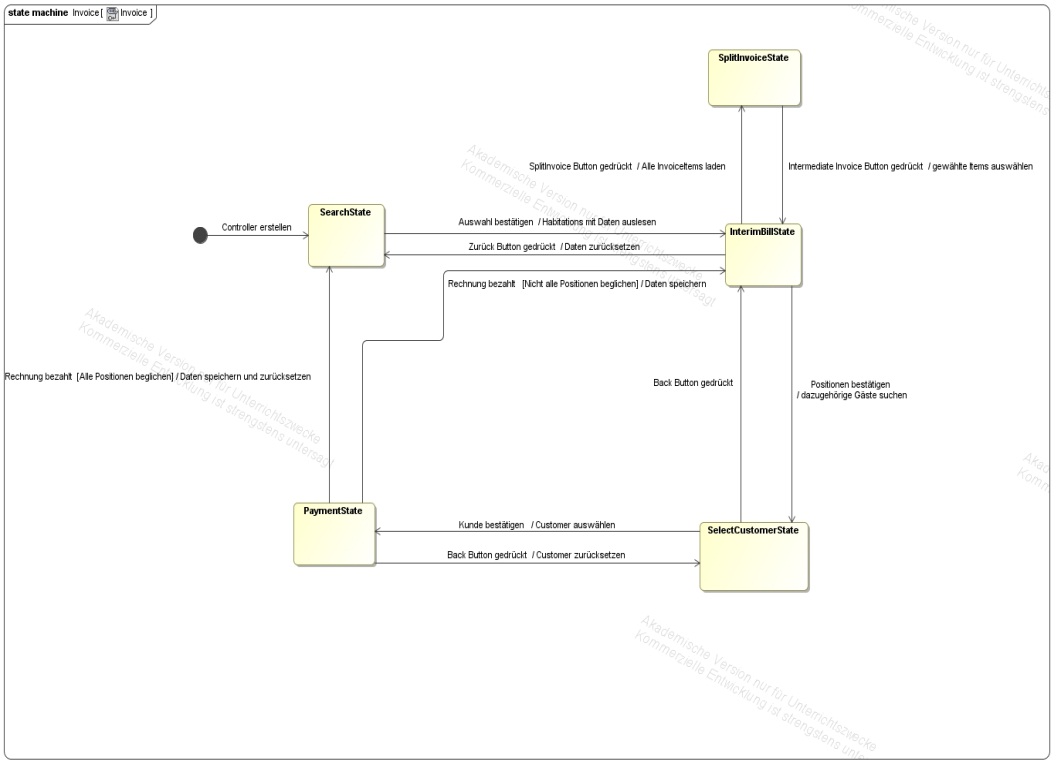


In dem Paket-Ausschnitt sehen wir, wie der GUI-Controller seine Anfragen an den Check-in-Controller stellt. Dieser bearbeitet seine Anfragen, gibt dabei nur Daten mit Lesezugriff zurück (spezielle Interfaces: z.B. „ReservationData“) und steuert den internen Ablauf wie im Anschluss beschrieben.

### 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“



*Hochauflösendes Bild beiliegend*

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 einzelner Vorgang bei einer Installation des Programms erfolgen kann, haben wir ein „Singleton-Pattern“ für die Use case 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.

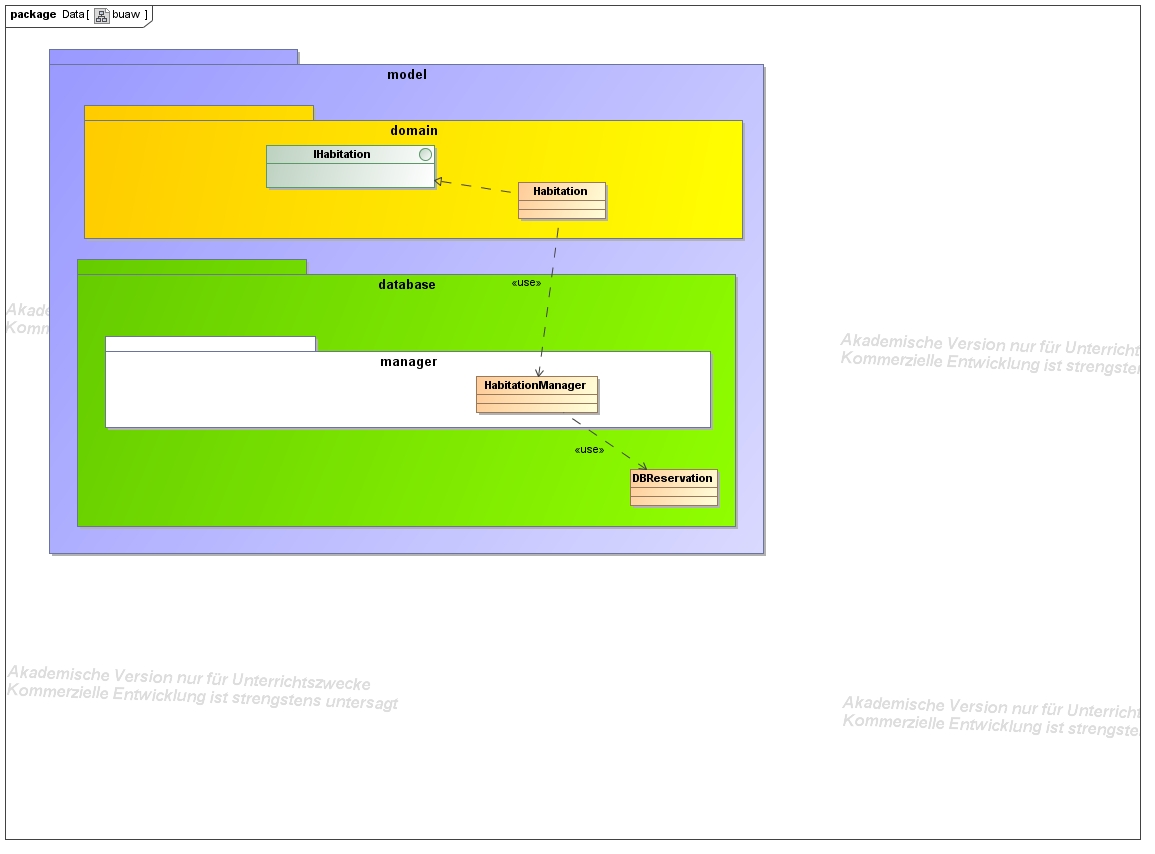
### Vorteile

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 Proxies 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.

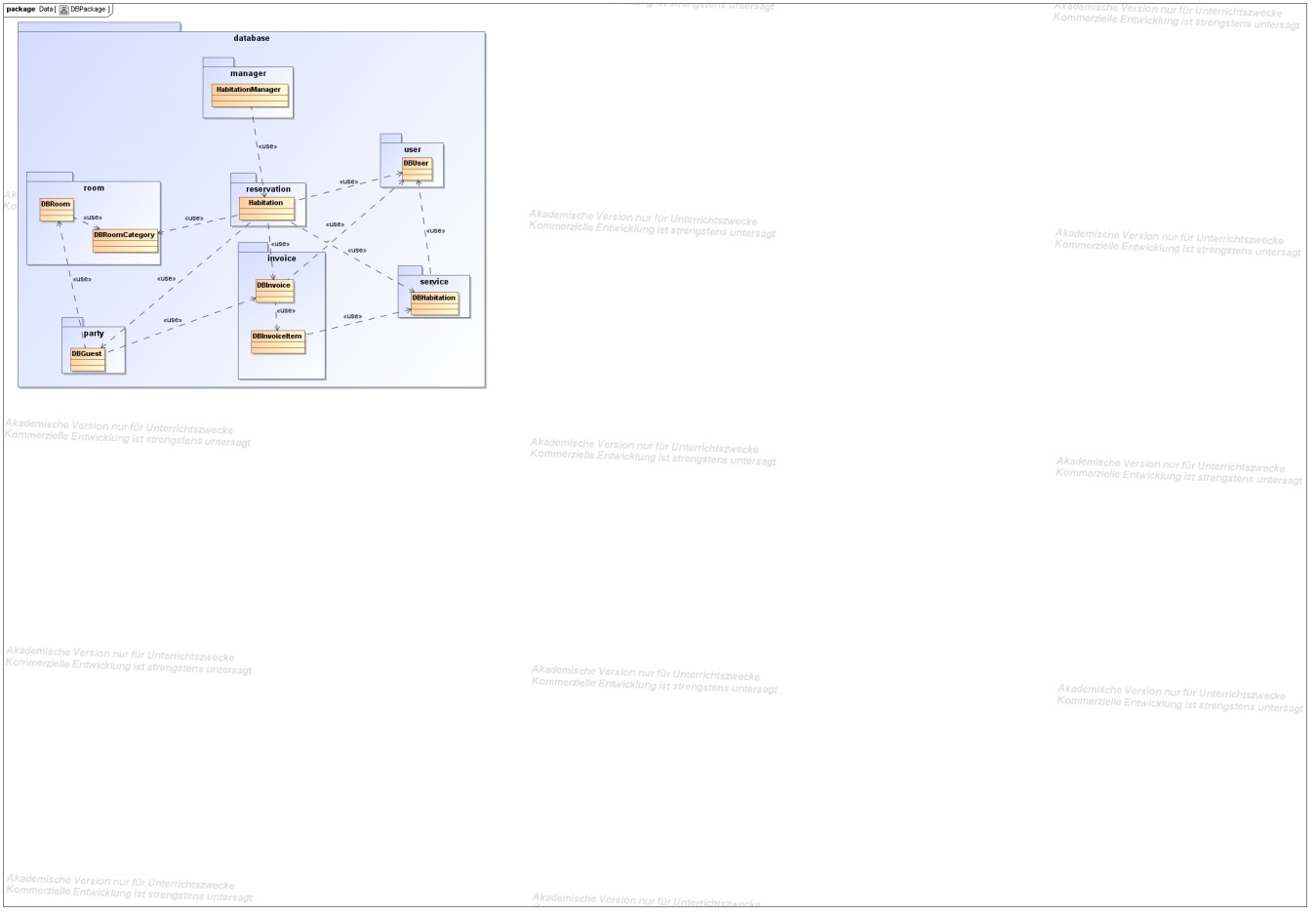
## Modell

### 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.



### Persistierung

Das Datenbankpaket bietet sogenannte Manager an, über die Daten von der Datenbank geholt werden können und auch die Speicherung bzw. Dematerialisierung findet dort statt. Dies hat den Vorteil, dass nach außen für jedes Paket nur eine Anlaufstelle vorhanden ist um etwas von der Datenbank zu bekommen oder zu speichern.

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

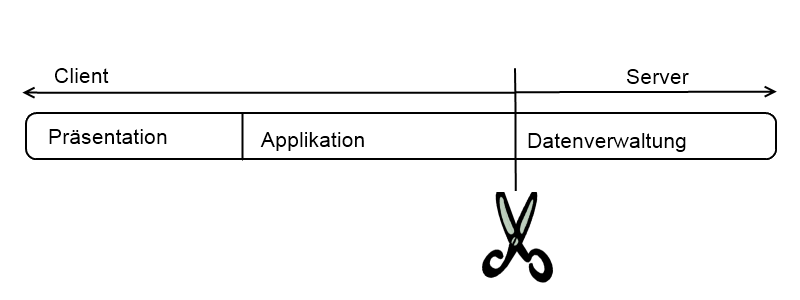
### „Dynamic Mapper“

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. Diese Unterbrechung bewerkstelligen wir mithilfe einer HashMap die wir bei jedem rekursiven Aufruf der Funktion mitgeben. In dieser HashMap werden alle gemappten Paare gespeichert und somit kann überprüft werden ob ein Objekt bereits konvertiert wurde.

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 mithelfe der oben bereits erwähnten HashMap umgangen wird. Diese Vorgehensweise mit der HashMap hat eine Geschwindigkeitsverbesserung des Mappers im Vergleich zur vorherigen Variante mithilfe von Tiefenberschränkung um den Faktor von ca. 300 ergeben.

## 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.

Im Laufe von Timebox 2 wurden ein paar kleinere Datenbankverbesserungen durchgeführt (aktuelles ERM-Modell beiliegend).

# Adaptierung

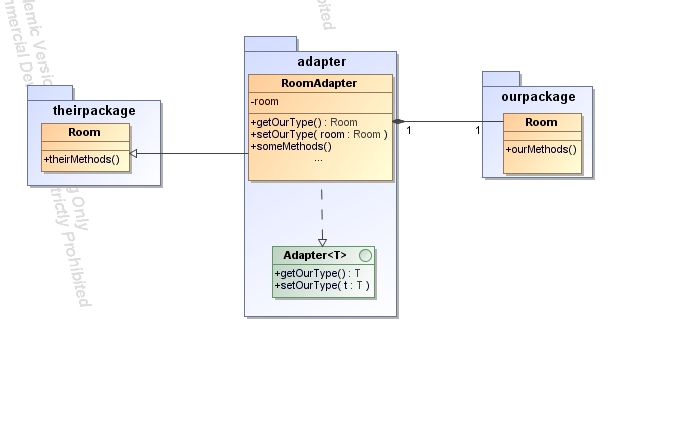
## Adapter für Programmteile von Team F

Eine Integration der anderen Programmteile durch einfache Implementierung von Interfaces in den Adapter-Klassen war in unserem Fall nicht möglich da uns keine Interfaces zur Verfügung gestellt wurden. Eine Änderung der Methoden-Schnittstellen von Team F, um diese Vorgehensweise umzusetzen, wurde von uns abgelehnt, da dafür umfangreiche Änderungen am fremden Code nötig gewesen wären. Diese Änderungen hätten wir dann bei eventuellen neueren Versionen der Programmteile wieder von neuem vornehmen müssen.

Das Problem wurde damit umgangen, dass nun unsere Adapter Klassen direkt von den Domänenobjekten des anderen Teams erben und ansonsten wie ganz normale Adapter-Klassen unser eigenes Interface beinhalten und die Methoden an ebendieses delegieren. Weiters implementiert jeder Adapter ein generisches „Adapter“-Interface welches Methoden bereitstellt um den inneren (also den von uns implementierten) Typ zu setzen und auszulesen. Dies ermöglichte uns dynamischere Programmierung, da alle unsere Adapter dieses Interface unterstützen und so ganz einfach von anderen Klassen erzeugt oder ausgelesen werden können.

Problematisch bei dieser Umsetzung ist, dass die Adapter auch die Methoden der Superklassen der adaptierten Klasse überschreiben müssen, da Java ja keine Mehrfachvererbung erlaubt und wir deshalb nicht von anderen Adaptern erben können.

Das folgende Diagramm visualisiert unser Adapter Design:



Als Einstiegspunkt für unser Programm wählten wir die „ManagerFactory“ von Team F, anstelle von konkreten Managern liefert unsere angepasste Factory nun Interfaces der Manager welche von uns konkret mit „ManagerAdapter“-Klassen umgesetzt werden. Die Schnittstellen der „ManagerAdapter“ verwenden die Klassen von Team F, intern wird aber auf unsere eigene Persistenzschicht zugegriffen indem mittels dem zuvor erwähnten Adapter-Interface unsere Typen ausgelesen und gespeichert werden oder im umgekehrten Fall eine neue Adapter-Klasse erzeugt und mit unserem inneren Typ ausgestattet wird. Die Übergabe kann dank der eingesetzten Vererbung trotzdem über die von Team F definierten Schnittstellen erfolgen.

## GUI Adaptierung

Team F setzt auf ein SWT-GUI, welches sich nicht ohne weiteres in unser Swing GUI integrieren lässt.

Jedoch besitzen die von uns zu adaptierenden Use cases von Team F keine umfangreichen User Interfaces, der Tagesabschluss selbst wird einfach durch einer MessageBox bestätigt. Hier wurde anstelle der SWT MessageBox einfach das entsprechende Swing Pendant verwendet.

Die Akonto-Buchung selbst wurde von Team F in ihrem Habitation-Panel umgesetzt, da sich ein SWT-Panel nicht in eine Swing Anwendung integrieren lässt und wir das User Interface nicht umschreiben wollten, da Team F noch Änderungen daran vornahm, wird das entsprechende Panel nun in einem eigenen Fenster aus unserer Applikation heraus geöffnet.