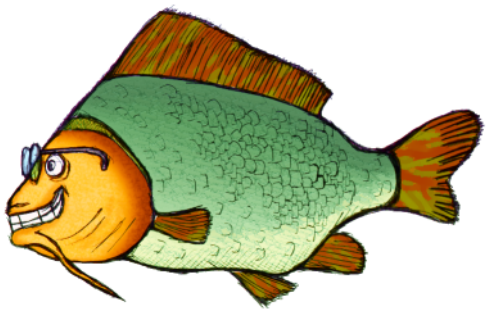
Computer Aided Room Planning

Implementierungsbericht

Johannes Pirkl

9. Juli 2004



Software Engineering Praktikum SS 04
Fakultät für Mathematik und Informatik
Universität Passau

Phase	Phasenverantwortliche(r)	E-Mail
Pflichtenheft	Würdinger Markus	wuerding@fmi.uni-passau.de
Entwurf	Leebmann Philipp	leebmann@fmi.uni-passau.de
Spezifikation	Renner Kerstin	renner@fmi.uni-passau.de
Implementierung	Pirkl Johannes	pirklj@fmi.uni-passau.de
Test	Pfnür Wolfgang	pfnuer@fmi.uni-passau.de
Präsentation	Teichtweier Stefan	teichtwe@fmi.uni-passau.de

Inhaltsverzeichnis

Zusammenfassung

Dieses Dokument soll einen Überblick über den Ablauf der Implementierungsphase des Projekts C.A.R.P. geben. Insbesondere sollen die Änderungen und Erweiterungen, die im Vergleich zum während der Entwurfs- und Spezifikationsphase vorgeschlagenen Aufbau notwendig wurden hier zusammengefasst und erläutert werden. Allerdings wird dabei nur auf größere bzw. strukturelle Abweichungen vom Entwurf eingegangen, um den Rahmen dieses Dokuments nicht zu sprengen.

1 Das package "controller"

Das package "controller" beherbergt wesentlich mehr Klassen , als ursprünglich im Entwurf vorgesehen war (Siehe Abbildung 1). Da es sich aber bei allen zusätzlichen Klassen um Action-Klassen handelt, die mit den vom Benutzer auslösbaren Funktionen übereinstimmen sei hier zur Erklärung nur angeführt, dass die mit der Präfix "**Prepare**" beginnenden Actionklassen i.A. dazu dienen die Anzeige durch eine JSP insoweit vorzubereiten, als dazu beispielsweise ein Auslesen des zu editierenden Objekts aus der Datenbank notwendig ist. Ferner wurden an einigen Stellen ähnliche Klassen zu Vererbungshierarchien zusammengefasst, um doppelten Code zu vermeiden. Im Einzelnen sind noch die folgenden abstrakten Basisklassen erwähnenswert, die das Rollenkonzept um die bereits in Entwurf und Spezifikation vorgesehene Klasse "PrivilegedAction" herum erweitern und abrunden.

1.0.1 OwnedObjectAction

Actions können von dieser Action ableiten, um Benutzeraktionen auf Objekten zu implementieren, deren Ausführung nur dem Eigentümer des Objekts und Benutzern mit einem gewissen Privileg erlaubt sein soll. Dazu wird an die Superimplementierung neben dem entsprechenden Privileg ein String-Wert übergeben, der angibt, welcher Parameter des Requests den abstrakten Namen des zu manipulierenden Objekts darstellt. Die konkrete Implementierung muss dann durch passende Implementierung von **objectForName** eine entsprechende Instanz einer Klasse, die das Interface **OwnedObject** implementiert aus dem Wert dieses Parameters generieren (in unserem Fall ist dies das Auslesen aus der Datenbank), woraufhin die Superimplementierung überprüft, ob der aktuell eingeloggte Benutzer Eigentümer dieses Objekts ist. Hat der Benutzer hingegen das erforderlich Privileg, wird zwar auch **objectForName** aufgerufen, aber die Aktion wird unabhängig davon, ob er Besitzer des Objekts ist oder nicht ausgeführt.

1.0.2 RoleDependentUserAction

Klassen leiten von dieser Action ab, um Benutzeraktionen auf Benutzerobjekten zu implementieren, deren Ausführung nur dem jeweiligen Benutzer selbst, oder denjenigen Benutzern mit einem gewissen Privileg und einer der Rolle des zu bearbeitenden Benutzers übergeordneten Rolle, erlaubt sein soll. Prinzipiell ist also der Vorgang der Überprüfung dem ähnlich der in OwnedObjectAction angewandt wird, allerdings ist die Ausführung der Benutzeraktion auch noch von der Rolle des zu bearbeitenden BenutzerObjekts abhängig, und nicht nur von seinem

Eigentümer (jeder Benutzer gehört sich selbst). Dadurch wird beispielsweise verhindert, dass Administratoren den Superadministrator löschen. Leider konnte diese Relation zwischen auf den Rollen und damit den Benutzerklassen aus zeitlichen Erwägungen nicht mehr in den Datenbankentwurf übernommen werden. Stattdessen entscheidet die konkrete Implementierung der Methode **matchRoleNames** darüber, ob eine Rolle einer anderen übergeordnet ist, oder nicht.

1.1 Das package "controller.prepareviewaction"

Um die Fülle von Actions zumindest etwas übersichtlich aufzuteilen, wurden in diesem Paket Actions zusammen gefasst, die die Anzeige einer stundenplanähnlichen zeitlichen Übersicht über eine gewisse Menge von Einzelbelegungen (SingleOccupancies) insoweit vorbereiten, als sie die dafür benötigten Einzelbelegungen aus der Datenbank auslesen. Da die Einzelbelegungen sowohl nach verschiedenen zeitlichen Kriterien als auch nach verschiedenen datenbankspezifischen Kriterien ausgewählt werden können, und Java keine Mehrfachvererbung unterstützt, wurde ein Delegationsmuster angewandt, um die verschiedenen Kriterien in beliebiger Weise kombinieren zu können, ohne dass sich größere Mengen doppelten Codes ergeben. (Siehe Abbildung2)

1.1.1 PrepareSingleOccupancyViewAction

Diese abstrakte Klasse stellt die Methode **prepareRequest** zur Verfügung, die von den konkreten Implementierungen benutzt werden kann, um Einzelbelegungen aus der Datenbank auszulesen, und im Request zu platzieren.

1.1.2 PrepareSingleOccupancyViewAction.TimeSpanDelegate

Dieses Interface stellt Methoden zur Verfügung, um ausgehend von einem gegebenen Zeitpunkt Länder-spezifisch das Ende und den Anfang einer gewissen den Zeitpunkt enthaltenden Zeitspanne zu berechnen. So beginnt die Woche den USA beispielsweise mit dem Sonntag während in den meisten Ländern Europas der Montag der erste Tag der Woche ist.

1.1.3 PrepareSingleOccupancyViewAction.SingleOccupancyDelegate

Dieses Interface wird benutzt, um anhand anwendungsspezifischer Kriterien SingleOccupancies auszuwählen und aus der Datenbank auszulesen. Die dazu notwendigen Informationen sind dem Request zu entnehmen.

1.1.4 Konkrete Implementierungen

Durch die verschiedenen Implementierungen der internen Interfaces von PrepareSingleOccupancyViewAction und deren Kombination bilden sich restlichen Action-Klassen, die für verschiedene Zeiträume die Anzeige aller Einzelbelegungen eines Raumes, eines Stundenplans, bzw. einer Belegung vorbereiten, und den Request an eine entsprechende View weiterleiten.

2 Das package “db”

In diesem Paket befinden sich lediglich der ConnectionPool sowie ein Hilfsklasse, die in dem Bestreben entstanden ist, den immer wiederkehrenden redundanten Code der **PersistentDatabaseObject** implementierenden Klassen zu eliminieren. Ein aus dem selben Bestreben entsprungener eleganter Ansatz mittels Delegation und ohne Verwendung statischer Methoden konnte mangels zeitlicher Ressourcen leider nicht mehr über das experimentelle Stadium hinaus entwickelt werden und ist hier nicht weiter aufgeführt.

2.0.5 ConnectionPool

Diese Klasse wurde um Methoden zum threadbasierten Transaktionsmanagement erweitert. Die Methoden **beginThreadTransaction**, **endThreadTransaction** und **abortThreadTransaction** dienen zum beginnen, beenden und abbrechen (rollback) einer solchen threadbasierten Transaktion. Alle Operationen die während einer solchen threadbasierten Transaktion innerhalb desselben Threads auf vom ConnectionPool verwalteten Connections ausgeführt werden, sind aus Sicht der Datenbank derselben Transaktion zugehörig. Befindet sich der ausführende nicht in einer threadbasierten Transaktion, verhalten sich die Connections wie unter normalen Bedingungen, d.h. jede Operation wird auf eine einzelne Transaktion abgebildet. Die Tatsache, dass Transaktionen vom ConnectionPool verwaltet werden, hat den Vorteil, dass die Benutzung der Connections von Seiten der persistenten Klassen aus völlig transparent erfolgt, ohne dass Transaktionen beachtet werden müssen. Vielmehr werden vom **ActionHandlerServlet** alle Benutzeraktionen innerhalb eines eigenen Transaktionskontext ausgeführt.

2.0.6 SqlExecuter

Diese Hilfsklasse stellt statische Methoden zur Verfügung, um den oft auftretenden Zugriff auf den ConnectionPool soweit wie möglich zu vereinfachen.

3 Das package “model”

Im Zusammenhang des Pakets “models” sind sowohl einige (kleinere) Klassen hinzugefügt worden, als auch einige konzeptionelle Veränderungen vorgenommen worden.(Siehe Abbildung 3)

3.0.7 Registration

Diese Klasse implementiert **PersistentDatabaseObject** um registrierte, aber noch nicht freigeschaltete Benutzeraccounts zu markieren.

3.0.8 Title

Diese Klasse implementiert **PersistentDatabaseObject** um die verschiedenen den Benutzern zuzuordnenden Titel in der Datenbank zu speichern.

3.0.9 Function

Diese Klasse implementiert **PersistentDatabaseObject** um die verschiedenen den Benutzern zuzuordnenden Funktionen in der Datenbank zu speichern.

3.0.10 OwnedObject

Diese Interface wird im Zusammenhang des Rollen- und Privilegiensystems verwendet, um für Instanzen der implementierenden Klassen den Benutzernamen des Eigentümers nachzuschlagen und ihm ggf. Sonderrechte einzuräumen.

3.0.11 Picture

Diese Klasse implementiert **PersistentDatabaseObject** um die den Räumen zugeordneten Bilder in der Datenbank zu speichern. Dieses Wunschkriterium konnte aus zeitlichen Gründen leider nicht mehr vollständig implementiert werden.

3.0.12 Role

Um den Polymorphismus des Rollen- und Privilegiensystems effektiver zu Gestalten, wurde Role statt als Objekt als Interface verwirklicht. Es wird von **NilPotentRole** und **PersistentRole** implementiert.

3.0.13 NilPotentRole

Der Laufzeit-Typ einer einer Instanz von **AnonymousUser** zugeordneten **Role**. Wie der Name andeutet, besitzt diese Implementierung von **Role** nie Privilegien, sodass dem anonymen Benutzer jegliche privilegierten Aktionen verweigert werden. Somit ist für das Erstellen einer neuen Session und dem damit einhergehenden Anlegen einer Instanz von **AnymousUser** kein Zugriff auf die Datenbank notwendig.

3.0.14 PersistentRole

Der Laufzeit-Typ einer einer Instanz von **User** zugeordneten **Role** Diese Klasse entspricht der Klasse **Role** im ursprünglichen Entwurf. Bei dieser Implementierung wird anhand der in der Datenbank enthaltenen Werte über das Vorhandensein eines Privilegs entschieden.

3.0.15 Timetable

Um den Inklusionspolymorphismus zwischen User und AnonymousUser intelligenter auszunutzen wurde Timetable als Interface statt als Klasse implementiert. Es wird von **SimpleTimetable** und **PersistentTimetable** implementiert.

3.0.16 SimpleTimetable

Der Laufzeit-Typ eines einer Instanz von **AnonymousUser** zugeordneten **Timetable**. Die dem Stundenplan zugeordneten Belegungen werden nicht persistent gespeichert, sondern mit dem Verfallen der Session verworfen

3.0.17 PersistentTimetable

Der Laufzeit-Typ eines einer Instanz von **User** zugeordneten **Timetable** Diese Klasse entspricht der Klasse **Timetable** im ursprünglichen Entwurf. Die dem Stundenplan hinzugefügten Belegungen werden persistent in der Datenbank gespeichert.

3.1 Das package "model.search"

Es sind einige Search-Klassen hinzugekommen. Insbesondere dann wenn dem Benutzer ein Auswahlfeld oder ähnliches mit allen in der Datenbank vorhandenen Objekten eines Typs präsentiert werden soll, muss eine Möglichkeit bestehen, alle Alternativen aufzuzählen. (Siehe Abbildung 4)

3.1.1 SimpleSearch

Diese abstrakte Klasse stellt **protected** Methoden bereit, die von konkreten Implementierungen benutzt werden können, um besonders einfach Kriterien in einfacher Weise auf SQL-Strings abzubilden.

3.1.2 UserSearch

Eine Suche nach Benutzern, die gewissen Kriterien genügen.

3.1.3 RoomGroupSearch

Eine Suche nach Raumgruppen, die gewissen Kriterien genügen.

3.1.4 SimpleSearchWithRoomGroups

Eine abstrakte Basisklasse, die das rekursive Modell der Raumgruppen berücksichtigt, und entsprechend auf Wunsch zunächst die transitive Hülle der Inklusionsbeziehung zwischen Raumgruppen bildet, bevor die eigentliche Suchanfrage gestellt wird.

3.1.5 EquipmentItemSearch

Eine Suche nach Ausstattungsmerkmalen.

3.1.6 OccupancyGroupSearch

Eine Suche nach Belegungsgruppen.

3.1.7 ConflictingSingleOccupancySearch

Eine Suche nach Konflikten in Form zeitlicher Überschneidungen zwischen Einzelbelegungen, die einer gewissen Menge von Belegungen angehören. Sie wird beispielsweise beim Einfügen in den Stundenplan und beim Anlegen von Belegungen verwendet.

3.2 Das package “model.pattern”

In diesem Paket haben sich im Wesentlichen keine Veränderung bis auf die Hinzunahme einer gemeinsamen Basisklasse und der Verwendung des “Factory”-Design-Patterns keine grundlegenden Veränderungen ergeben. (Siehe Abbildung 5)

3.2.1 OccupancyPattern

Im Zuge der Anwendung des “Factory”-Design-Patterns wurde OccupancyPattern statt als Interface als abstrakte Klasse verwirklicht. Die Applikation kann sich über die statische Methode **getPatternNames** zunächst alle verfügbaren Implementierungen von **OccupancyPattern** anhand abstrakter Namen benennen lassen. Über die ebenfalls statische Methode **getPattern** kann schließlich eine der Implementierungen angefordert werden, um als Factory für **SingleOccupancyIterator** zu dienen.

3.2.2 SimpleSingleOccupancyIterator

Eine abstrakte Basisklasse, die es ermöglicht, auf besonders einfache Art und Weise das Interface SingleOccupancyIterator zu implementieren, um von OccupancyPattern abzuleiten. Dazu muss lediglich die Methode **atomicNext** implementiert werden.

4 Das package “view”

Um bei der relative komplexen Darstellungslogik, die für eine stundenplanähnliche zeitliche Ansicht von SingleOccupancies notwendig ist, nicht auf die bescheidenen Methoden zur Fehlerkontrolle, wie sie in JSPs vorhanden sind, beschränkt zu sein wurden mit diesem Paket Actions aufgenommen, die innerhalb des MVC-Modells insofern nicht als controller, sondern als views auftreten, als sie lediglich schon aufbereitete Daten darstellen. Dies hat den weiteren Vorteil, dass für verschiedene Ausgabeformate Code-Sharing auf einer höheren Ebene betrieben werden kann, als dies für gewöhnlich in JSPs möglich ist.(Siehe Abbildung 6)

4.0.3 TimetableView

Eine abstrakte, gemeinsame Basisklasse für stundenplanähnliche zeitliche Ansichten. Die Darstellung benutzt die iText-Bibliothek. Diese Klasse stellt die Methode **createTable** zur Verfügung, die von Subklassen benutzt werden kann, um eine Menge von Einzelbelegungen in einer **Table** (iText-Klasse) darzustellen. Die Art und Weise wie Einzelbelegungen dabei innerhalb der Zellen dargestellt werden kann durch Overriding der Methode **fillCell** verändert werden.

4.0.4 HTMLView

Diese Klasse stellt eine stundenplanähnliche zeitliche Ansicht im HTML-Format dar. Sie stellt dabei nur die entsprechende Tabelle selbst, nicht aber HTML-Header oder ähnliches dar, da sie dazu gedacht ist, vom Standard JSP-Layout aus eingebunden zu werden.

4.0.5 PDFView

Diese Klasse ersetzt zusammen mit den entsprechenden Klassen aus dem Paket controller.prepareviewaction die im ursprünglichen Entwurf aufgeführte PDF-ExportAction. Sie stellt eine stundenplanähnliche zeitliche Ansicht im PDF-Format dar.

5 Das package "tags"

Analog zu den Veränderungen im Paket controller wurde auch hier eine Klasse hinzugefügt. (Siehe Abbildung 7)

5.0.6 OwnedTag

Diese Klasse implementiert ein Tag, das auf JSP-Seite das Darstellungs-Analogon zur OwnedObjectAction ist. Ein in diesem Tag eingeschlossener Bereich einer JSP ist nur dann sichtbar, wenn der eingeloggte Benutzer ein gewisses Privileg hat, oder Eigentümer eines gegebenen Objekts ist.

6 Das package "mail"

Das package mail stellt die Anbindung zur JavaMail-API zur Verfügung.

6.0.7 CarpMail

Die Klasse CarpMail ist nach dem "Singleton"-Design-Pattern verwirklicht, um es einerseits dem ActionHandlerServlet zu ermöglichen, die zum Aufsetzen der Mail-Session notwendigen Parameter bei der Initialisierung festzulegen, und andererseits den Mail-Dienst global erreichbar zu halten.

6.0.8 MailMessages

Diese Klasse dient lediglich zur Internationalisierung der von CarpMail verwendeten Fehlermeldungen.

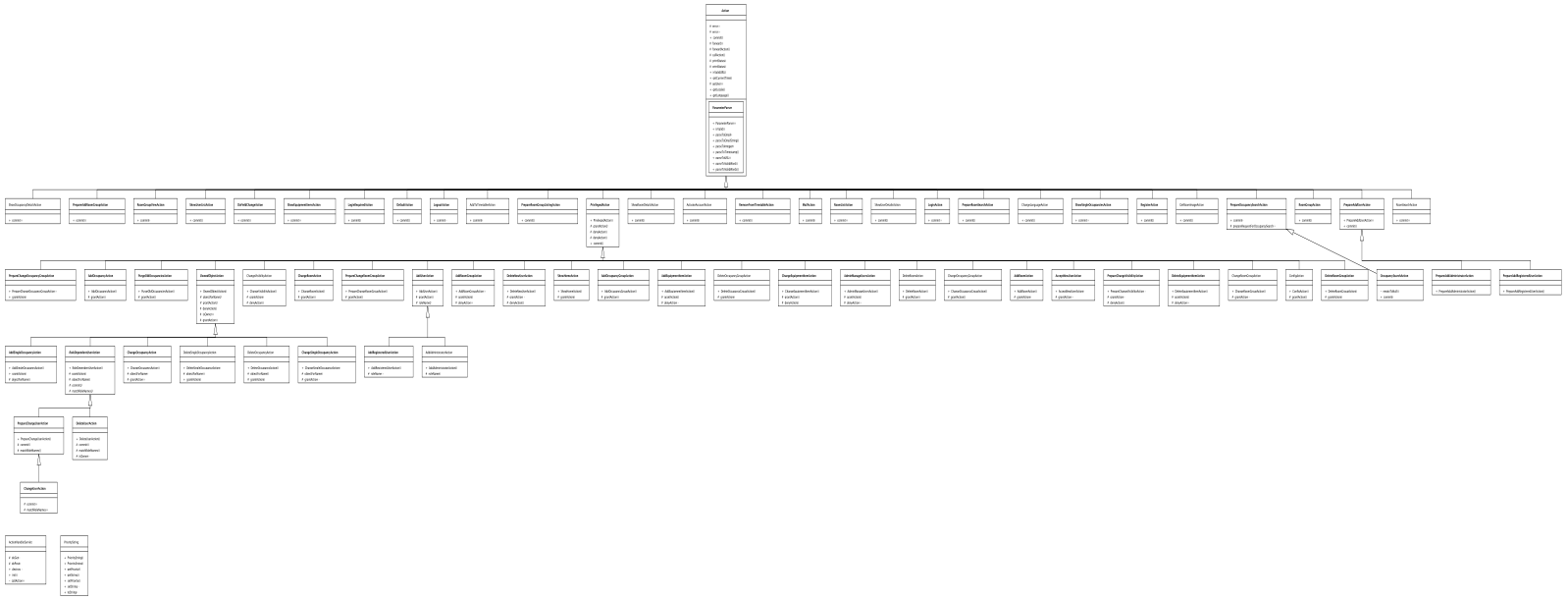


Abbildung 1: Das package controller

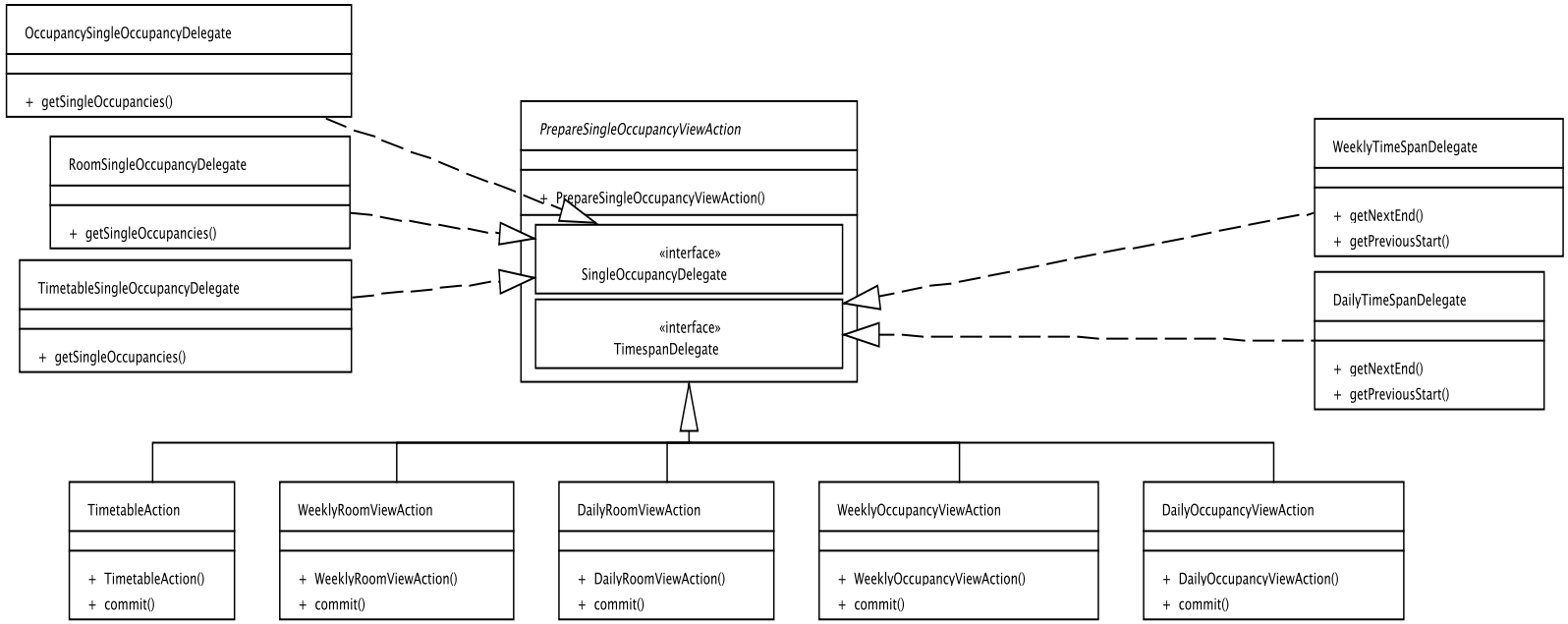


Abbildung 2: Das package controller.prepareviewaction

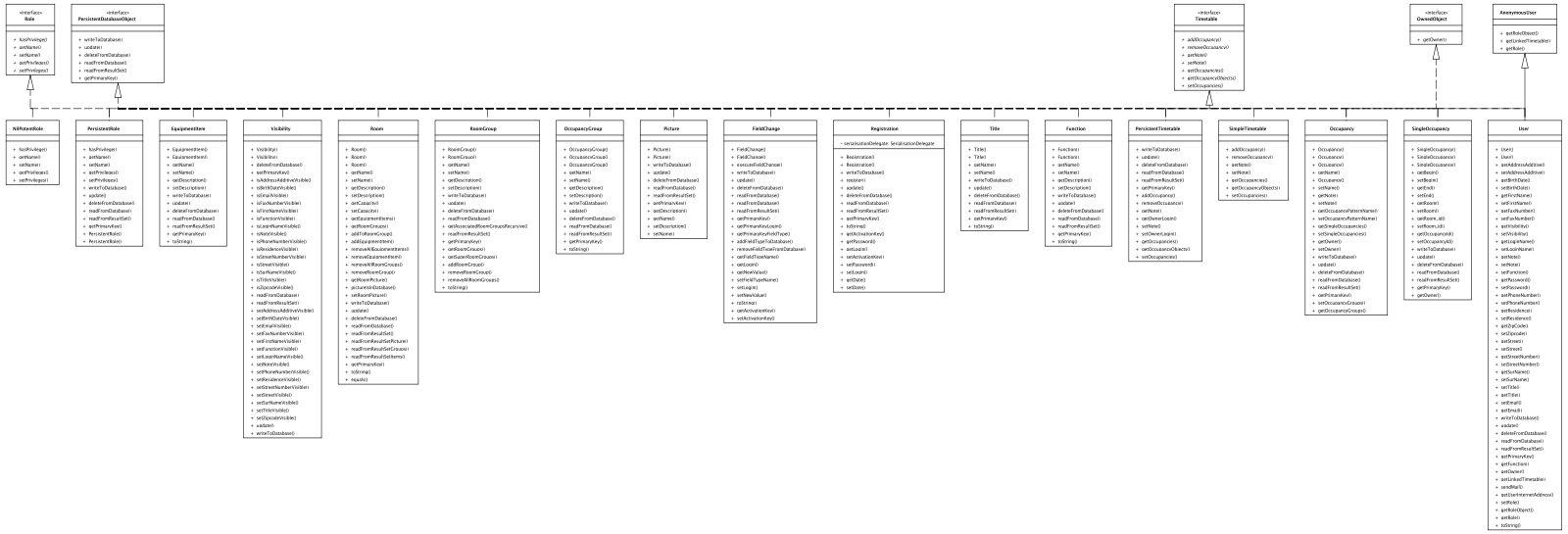


Abbildung 3: Das package model

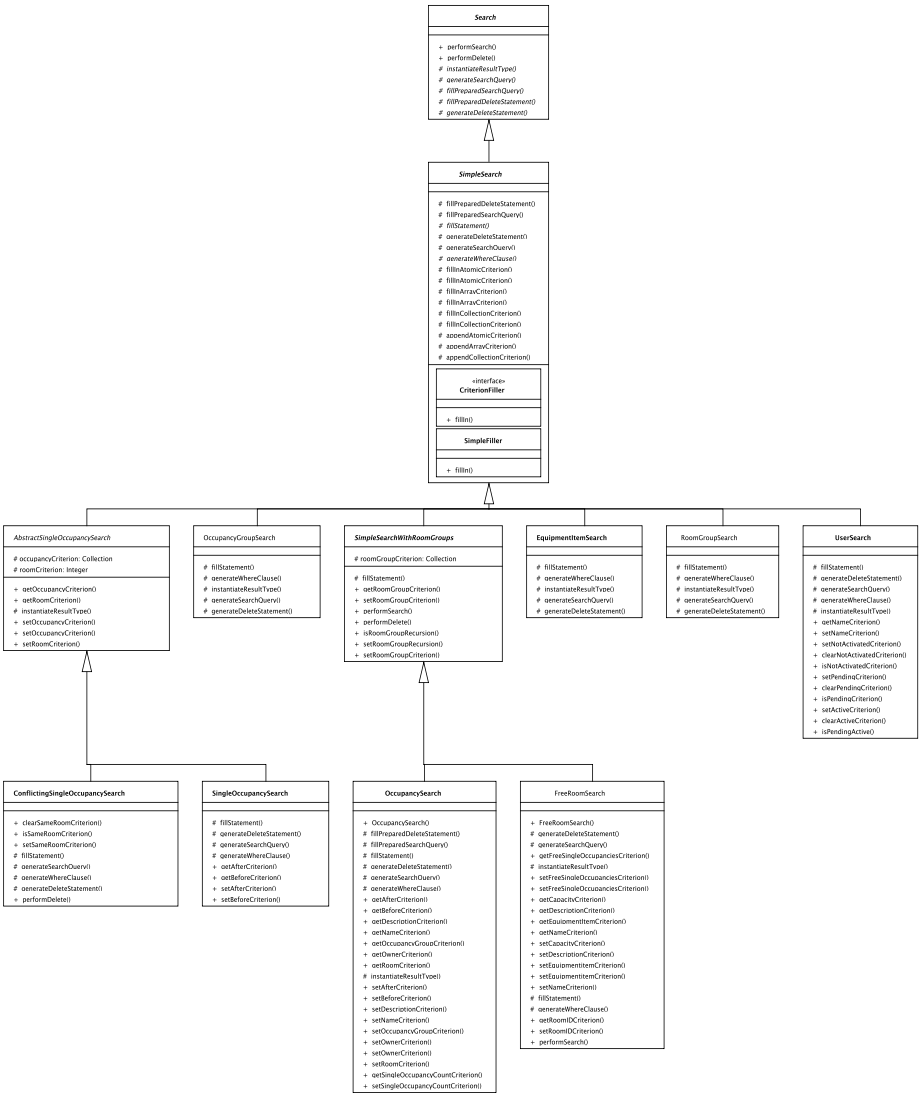


Abbildung 4: Das package model.search

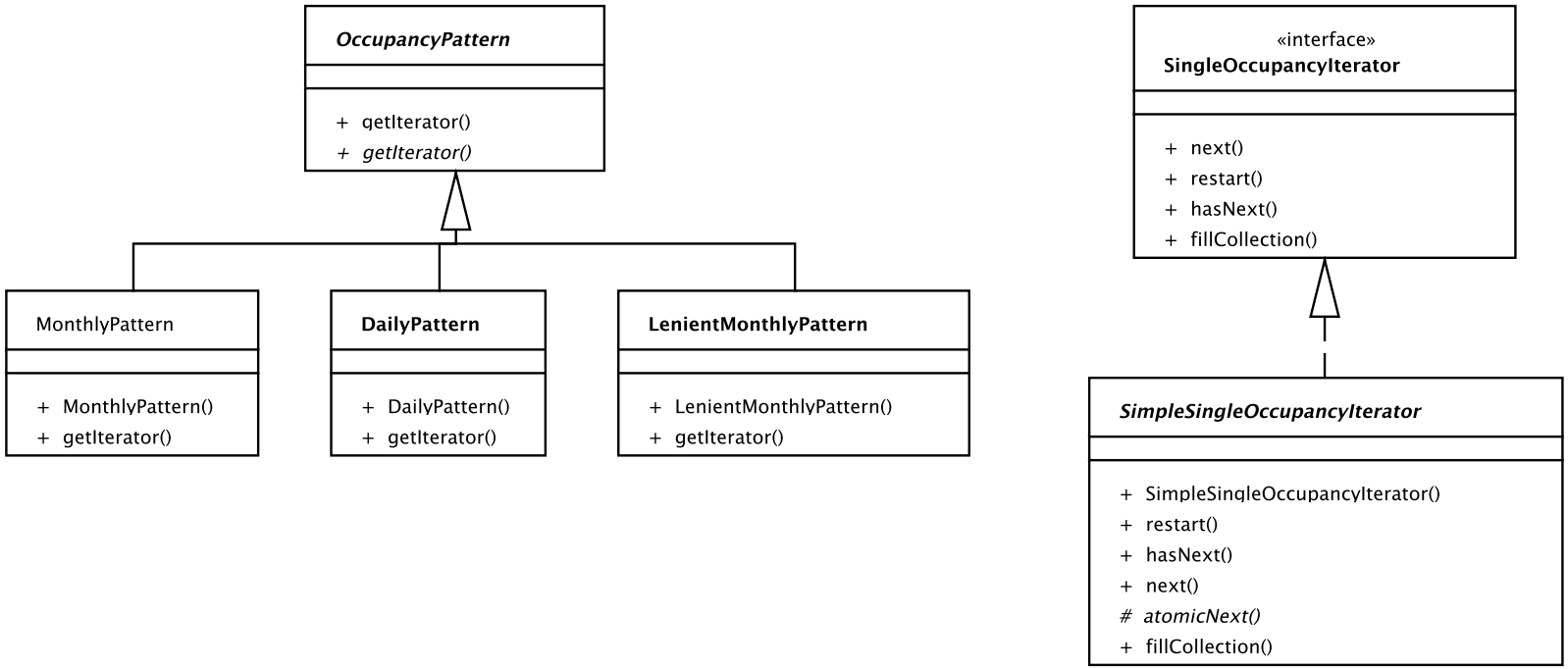


Abbildung 5: Das package model.pattern

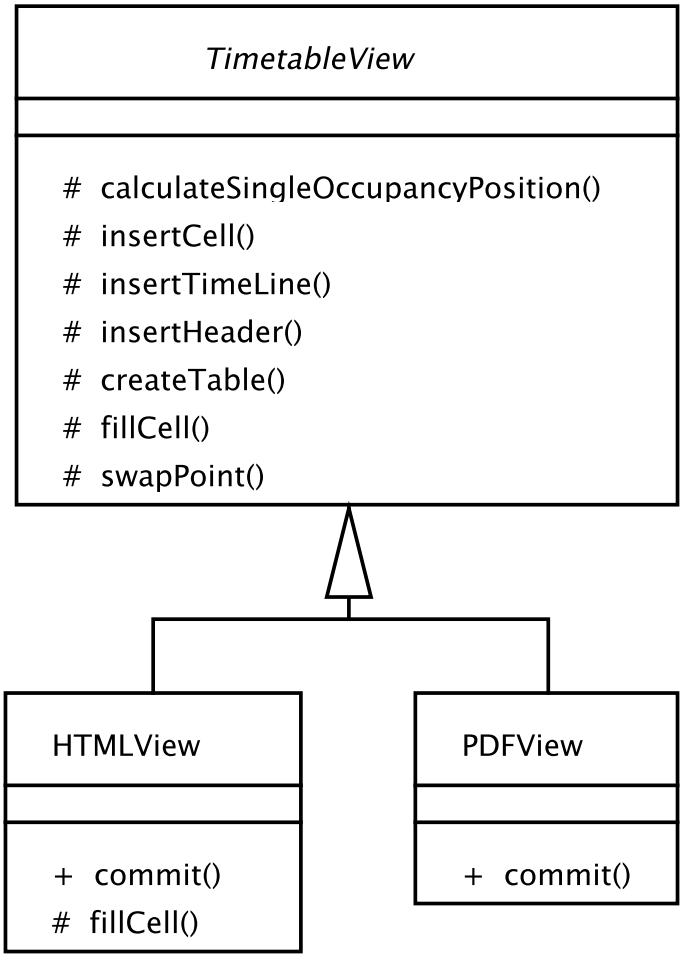


Abbildung 6: Das package view

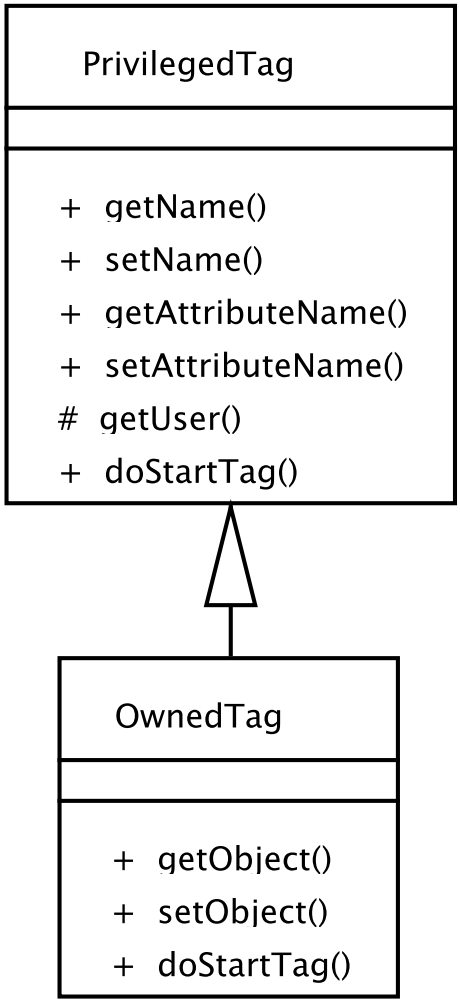


Abbildung 7: Das package tags