Entwicklung einer neuen DV-Methode zur effizienten Personaleinsatzplanung in Pflegeorganisationen und eine zugehörige Softwarerealisierung mit Java-JVC/Swing

Von

Michael Bürmann

Hegelstr. 11 - 40789 Monheim

Mat.-Nr.: 10 233 092

Fachbereich: Wirtschafts-Informatik
Fachhochschule Köln, Abt. Gummersbach

Diplomarbeit zur Erlangung des Grades
Diplom Wirtschafts-Informatiker

1. Prüfer: Prof. Dr. Faeskorn-Woyke
2. Prüfer: Dipl. Math. Dirk Peters

1. Inhaltsverzeichnis

2. Einleitung

Bei moderner Software kommt es auf verschiedene Faktoren an. Der erste für den Anwender zunächst sichtbare Faktor ist die Bedienungsfreundlichkeit des Produktes. Hierbei kommt man nicht mehr um grafische Benutzeroberflächen herum. Außerdem müssen die Schritte der zu erledigenden Arbeiten möglichst intuitiv erlernbar sein.

Der zweite wichtige Faktor ist die Verwendbarkeit des Produktes auf verschiedenen Hardwareplattformen. Dies ist vor allem für Programme wichtig, die sich an einen großen Kundenkreis richten, da man nicht davon ausgehen kann, oder sollte, dass überall das gleiche Betriebssystem eingesetzt wird. Des weiteren wird so auch ein Umstieg auf ein anderes System erleichtert.

Für das Softwarehaus selber ist die Wartbarkeit des Produktes von entscheidender Bedeutung. Objektorientierte Programmierung bietet hier durch Vererbungstechniken, Datenkapselung und ähnlichem die besten Voraussetzungen. Auch der Einsatz von Entwurfsmustern, also lange bewährten Konzepten objektorientierter Entwicklung, ist hier von Vorteil.

Eine Datenbankanbindung ist für fast jede kommerzielle Software notwendig. Hierbei ist es auch von Vorteil sich nicht an eine spezielle Datenbank zu binden. Ein Mittel hierzu ist es die Datenbanksprache SQL und offene Datenbanktreiber wie etwa ODBC zu benutzen.

Bei vielen Programmen ist es auch von Vorteil, wenn sie von Anfang an als verteilte Systeme entworfen werden. Verteilte Systeme haben den Vorteil, dass jedes Modul an einem anderen Ort sein kann und entweder über eine ständige oder aber auch eine temporäre Internetverbindung miteinander synchronisiert werden können.

In dieser Arbeit werde ich anhand eines Praxisbeispieles beschreiben, wie man all diese Faktoren in ein Softwareprodukt unterbringt, und wie die Entwicklung aussieht.

3. Das Swift-Projekt

Damit die Entwicklung eines Systems, welches oben genannte Eigenschaften enthält leicht nachvollziehbar wird, werde ich dies anhand eines Praxisbeispiels demonstrieren. Diese Diplomarbeit ist bei der DLR[1] entstanden. Ein europäisches Projekt, an der die DLR beteiligt ist heißt SWIFT. Im Rahmen dieses Projektes soll sich die Diplomarbeit bewegen. Dazu ist es erst einmal notwendig einige Dinge über Swift zu erklären.

3.1. Was ist Swift?

SWIFT steht als Synonym für „U***S***er oriented and ***W***orkflow ***I***ntegrated ***F***edera***T***ion of service providers for the elderly“. Es ist ein europäisches Projekt mit folgenden Beteiligten:

- Derwentside District Council (GB)
- Health Systems Co-ordination (GB)
- Deutsche Forschungsanstalt für Luft- und Raumfahrt e.V. (D)
- Omega Generation s.r.l. (I)

Die Ergebnisse aus diesem Projekt werden in den Städten Derwentside (GB), Bologna (I), Remscheid (D) und Wansbeck (GB) validiert.

SWIFT soll also ein benutzerorientierter, Workflow nutzender Zusammenschluss von Dienstleistern für ältere Menschen sein.

Ausschlaggebender Punkt um ein solches Projekt zu starten war die alarmierenden Prognosen bezüglich alter Menschen für die Zukunft. Demnach wird schon im Jahre 2005 ein Viertel der europäischen Bevölkerung über 65 Jahre alt sein. Bei der heutigen Struktur der Dienstleister für ältere Menschen ist es abzusehen, dass ohne Reorganisation ein deutlicher Versorgungsengpass dieser Bevölkerungsgruppe entstehen würde. Auch sind die Dienstleistungen häufig durch schlechte Organisation der Arbeitsabläufe zu kostenintensiv, was nicht nur den alten Menschen selbst, sondern auch die Allgemeinheit belastet. Mit diesem Projekt soll nicht nur der Kostenfaktor optimiert werden; ein weiteres Anliegen ist auch die Qualität der Dienstleistungen weiter zu steigern. Die Information über die zu Verfügung stehenden Dienste ist auch lange nicht jedem alten Menschen geläufig, womit auch hier eine Lösung gefunden werden muss. Hierzu wäre beispielsweise ein sehr einfach zu bedienendes Web TV wünschenswert.

Schon heute stehen mit dem Internet und der Workflow-Technologie Möglichkeiten zu Verfügung, die es den Europäern ermöglichen könnte diesem Problem Herr zu werden. Daher wird mit ***SWIFT*** in einem 30-monatigem Projekt untersucht, wie genau diese und andere Technologien helfen können. Im Detail gehören zu den angewendeten Technologien:

- Internet

[1] Deutsche Forschungsanstalt für Luft- und Raumfahrt e.V., Standort Köln-Porz

- Workflow-Engine
- JAVA
- XML (EDI)
- Web TV

Die Konkreten Ziele werden schon annähernd durch den Titel wiedergegeben. Hierbei steht natürlich der ältere Mensch im Vordergrund. Es ist im Sinne des alten Menschen vorzuziehen ihn im Pflegefall an seinem Heimatort zu belassen, da so unnötiger Stress vermieden wird, und dem Menschen ein Teil der Angst vor dem Alter genommen wird. Ein anderes nicht unerhebliches Ziel ist es durch Kostenreduktion eine Vielfalt von Diensten erschwinglich anbieten zu können, ohne den alten Menschen und die Allgemeinheit mehr als nötig zu belasten. Daneben ist es natürlich genau so wichtig, dass über die zu Verfügung stehenden Dienste ausführlich informiert wird, und diese Information jedem schnell und einfach zugänglich ist.

3.2. Spezielle Ausprägung in der DLR (ambulante Altenpflege)

Als spezielle Ausprägung dieser Studie wird an der Deutschen Forschungsanstalt für Luft- und Raumfahrt e.V. in Köln-Porz derzeit untersucht wie eine Computerunterstützung für einen ambulanten Pflegedienst unter Verwendung der genannten Technologien aussehen könnte. Hierzu werden diverse explorative Prototypen erstellt, die mit Hilfe von ambulanten Pflegediensten aus der näheren Umgebung, im speziellen z.B. der Johanniter-Unfall-Hilfe e.V. aus Rodenkirchen, validiert werden.

4. Personaleinsatzplanung in der ambulanten Pflege

4.1. Personaleinsatzplanung in der Praxis

Um ein Personaleinsatzplanungsmodul für einen ambulanten Pflegedienst zu entwickeln müssen wir uns erst einmal ansehen, wie die Vorgehensweise bisher manuell geschah. Dazu besucht man am besten einen ambulanten Pflegedienst und analysiert die Vorgehensweise.

Zunächst einmal betrachten wir uns die Hilfsmittel, die bisher zu Verfügung stehen. Dies ist eine Stecktafel in die kleine, beschriftete Kärtchen in unterschiedlichen Farben gesteckt werden konnten. Als Grundlage für die Personaleinsatzplanung wird auf der einen Seite natürlich Daten über das vorhandene Personal mit entsprechenden Verfügbarkeitsdaten und Qualifikationen benötigt, auf der anderen Seite die Daten über die zu pflegenden Kunden. Dazu gehören Daten wann die Pflegebedürftigen gepflegt werden sollen und wie lange in etwa der jeweilige Besuch dauert. Die Länge des Besuchs berechnet sich aus Erfahrungswerten über die einzelnen zu erbringenden Leistungen bei dem jeweiligen Pflegebedürftigen. Die Adresse des Pflegebedürftigen ist natürlich auch wichtig, damit ggf. eine Routenoptimierung durchgeführt werden kann und der Pfleger nicht unnötig lange Zeit in seinem Einsatzfahrzeug verbringen muss. Dies bringt uns zu den zu Verfügung stehenden Fahrzeugen. Auch darüber müssen natürlich Daten bereitgestellt werden.

Nachdem all diese Daten bereitgestellt wurden kann der Vorgang beginnen. Zunächst werden die Pflegebedürftigen in „Touren“ aufgeteilt. Eine „Tour“ wird von einem Pfleger übernommen. Somit ist jeder Tour auch ein Fahrzeug zugeordnet. In einer Tour werden

nacheinander alle ihr zugeordneten Pflegebedürftigen angefahren und nach Ihren gebuchten Leistungen gepflegt. Die Reihenfolge der Pflegebedürftigen in einer Tour bestimmt die Reihenfolge in der sie abgefahren werden. Zu jeder Tour wird zu Anfang festgelegt wie lange sie dauern soll und wann sie anfängt. Im allgemeinen richten sich diese Daten nach den Arbeitszeiten des Pflegepersonals, so dass man sagen kann dass eine „Tour“ eine Schicht eines Pflegers ist. Somit ist die „Tour“ ein wichtiger Begriff in diesem Zusammenhang. Also sehen wir uns einmal genauer an, wie eine Tour geplant wird.

Nehmen wir einmal an, wir wollen eine Tour für Montag Vormittag planen. Dazu suchen wir uns zunächst einmal heraus, welche Pflegebedürftigen Montag Vormittag Leistungen gebucht haben. Für jeden Pflegebedürftigen werden dann Kärtchen bereitgestellt, auf denen Ihr Name steht. Diese werden nach Uhrzeiten sortiert. Dabei sind dies meist Zeiträume in denen die Leistungen erbracht werden sollen. Weiter ist noch wichtig zu wissen, wie lange der Aufenthalt bei den einzelnen Pflegebedürftigen ungefähr ist. Also wird dies am besten mit auf den Kärtchen vermerkt. Nun kann es losgehen, und die einzelnen Kärtchen werden so zu der Tour gesteckt, dass sich möglichst keine Überschneidungen zweier oder mehrerer Besuche ergeben und möglichst alle zeitlichen Wünsche der Pflegebedürftigen beachtet werden. Sollten nicht alle Pflegebedürftige dieser Tour zugeordnet werden können, so muss eine weitere Tour nach gleichem Vorgehen geplant werden.

Dieses Prinzip der Tourenplanung wird fortgesetzt, bis alle Pflegebedürftigen verplant sind. Danach wird für jede Tour aufgrund der geplanten Leistungen jedes einzelnen Besuchs in dieser Tour eine Mindestqualifikation berechnet. Nach dieser Mindestqualifikation werden dann die zu Verfügung stehenden Mitarbeiter den einzelnen Touren zugeordnet. Hierbei sind die Dienstzeiten des Personals und ggf. Urlaub und Krankheit von Mitarbeitern natürlich zu beachten. Zum Schluss werden noch die zu Verfügung stehenden Fahrzeuge auf die Touren verteilt.

Nach diesem Arbeitsgang haben wir nun alle Touren für Montag Vormittag geplant. Nach gleichem Muster werden dann auch alle andern Tage und Tagesabschnitte geplant, wobei wünschenswert ist, wenn Touren auf andere Zeiträume kopiert werden können, die ähnlich verplant werden. Somit reduziert sich der Aufwand der Planung auf ein Minimum.

4.2. Ausblick auf weitere Planungsmöglichkeiten

Dies war nun eine Planung im herkömmlichen Sinne. Eine Neuerung ist nun die Planung aus Sicht des Pflegebedürftigen. Hierbei gibt es nicht mehr die Übersicht über die Touren, sondern die Planung wird bei jedem einzelnen Pflegebedürftigen separat durchgeführt. Im Detail kann man sich das wie folgt beschrieben vorstellen.

Zunächst wird ein Tag für den gewünschten Klienten ausgewählt. Auf dem Bildschirm sieht man in zeitlich geordneter Reihenfolge welche Besuche bei diesem Pflegebedürftigen für den gewählten Tag geplant wurden, in welchen Zeitfenstern sie ausgeführt werden sollen, wie Lange diese Tätigkeiten dauern werden und welche Qualifikation der pflegende Mitarbeiter haben muss, welcher diesen Besuch übernimmt. In einem andern Bildschirmbereich werden die zu Verfügung stehenden Pfleger dargestellt. Nun kann durch einfaches Verschieben eines Pflegers auf einen entsprechenden ‚Besuch’ festgelegt werden, dass genau dieser Pfleger diesen Besuch bei dem gewähltem Patienten übernimmt. Automatisch wird der Patient in die Tour des jeweiligen Mitarbeiters an der entsprechenden zeitlichen Position übernommen. Sollte der Mitarbeiter noch zu keiner Tour zugeteilt worden sein, so muss dies natürlich jetzt nachgeholt werden, denn sobald ein Mitarbeiter auch nur einen Patienten anfahren muss, so muss er auch eine Tour fahren, auch wenn diese nur aus einem einzigen Patienten besteht.

4.3. Leistungsmerkmale von Personaleinsatzplanungssystemen

Die hier zu entwickelnde Software ist natürlich nicht die erste für den Bereich häusliche Alten- und Krankenpflege. Es gibt bereits einige fertige und auch recht gut durchdachte Programme, die ebenfalls in Zusammenarbeit mit entsprechenden Dienstanbietern für ambulante Altenpflege entworfen wurde. Um festzustellen, in welchen Bereichen diese Software unseren Bedürfnissen entspricht oder welche Bereiche nicht abgedeckt werden, wird im folgenden vorhandene Software auf folgende Gesichtspunkte untersucht:

- Wird eine Personaleinsatzplanung unterstützt? (A)
    - Wird die Einsatzplanung durch einfache Mausoperationen unterstützt? (B)
    - Wird eine automatische Tourenplanung unterstützt? (C)
    - Wird eine Tourenplanung aus verschiedenen Sichten unterstützt? (D)
- Ist das Programm auf einem verteiltem System lauffähig? (E)
- Ist das System multiuserfähig? (F)

Obige für diese Arbeit relevanten Gesichtspunkte werden in einer Tabelle dargestellt. Die oben vermerkten Buchstaben werden aus Platzgründen in der Tabelle verwendet.

Softwareprodukt	A	B	C	D	E	F
IBAS-pflege[2]	Nein	-	-	-	Nein	Ja
CareOffice 5.0[3]	Ja	Ja	Nein	Nein	Nein	Ja
MISLA[4]	Ja	?	?	?	Nein	Ja
Boss-Heim[5]	Nein	-	-	-	Nein	Ja
ProfMed[6]	Ja	Nein	Nein	Nein	Nein	Ja
CareLine[7]	Ja	Nein	Nein	Nein	Nein	?
DM-Office[8]	Ja	Nein	Nein	Nein	Nein	?
SINFONIE[9]	Ja	Nein	Nein	Nein	Nein	Ja
Vivendi[10]	Ja	?	?	?	Nein	Ja

[2] Von Löpertz Software GmbH. Der Punkt F ist aufgrund der Tatsache, dass Oracle als Basis verwendet wird, sicher.

[3] Von C & S Computer und Software GmbH.

[4] Von mikis-Software.

[5] Von BOSS Branchen-Organisation und Software-Systeme GmbH.

[6] IC-SYS Informationssysteme GmbH

[7] Von Microcare Systemhaus GmbH

[8] DM EDV- und Bürosysteme GmbH

[9] Von SIH.

[10] Von MICOS – Mikro Computer Systeme und Anwendungen Vertriebs-GmbH

Die Felder mit „?“ deuten darauf hin, dass ich zu diesen Punkten in den mir vorliegenden Materialien keine Informationen gefunden habe.

Wie wir aus dieser Tabelle erkennen können, gibt es z.B. bei dem Punkt „Tourenplanung aus verschiedenen Sichten“ Defizite in der untersuchten Software. Genau dieser Punkt sollte aber unterstützt werden, wenn man davon ausgeht, dass die Planung auch im Sinne des Patienten erfolgen soll. Natürlich ist es nicht sinnvoll sich ausschließlich auf diese Sicht zu beschränken, jedoch sollte die Möglichkeit bestehen zwischen den unterschiedlichen Sichtweisen zu wählen und auch während der Planung auf beide Möglichkeiten zurückgreifen zu können.

Der Punkt eines verteilten Systems sollte im Zuge der Globalisierung auch nicht vernachlässigt werden. Gerade weil der Anschluss an das Internet der einzelnen Pflegeeinrichtungen immer wahrscheinlicher wird, oder sogar vollzogen ist, sollte es auch die neue Software unterstützen dieses Netz zu nutzen und Arbeit aufzuteilen.

4.4. Zusammenfassung

Wir haben jetzt gesehen, dass die vorhandene Software für die ambulante Altenpflege nicht perfekt ist, und vor allem im Hinblick auf verteiltes Arbeiten und Tourenplanung aus verschiedenen Sichten verbesserungswürdig ist. Daher werden wir im folgenden ein Pflegeeinsatzplanungsmodul mit den Werkzeugen moderner Softwareentwicklung realisieren. Hierzu gehört UML genauso wie die Verwendung einer plattformunabhängigen Sprache wie Java. Ein besonderes Augenmerk soll auch auf die Wiederverwendbarkeit der Software gelegt werden. Denn ein Planungsmodul lässt sich sicherlich in anderen Bereichen auch sinnvoll einsetzen, sofern es nicht zu exakt an die Bedürfnisse von ambulanter Altenpflege gebunden ist. Um diese Wiederverwendbarkeit zu erreichen werden wir auch zum Einsatz von Entwurfsmustern der Softwareentwicklung kommen. Wir werden darauf später genauer eingehen.

5. Theoretische Grundlagen zur Programmentwicklung

5.1. Objektorientierte Analyse/Design (OOA/OOD)

5.1.1. Die Unified Modeling Language (UML)

Beim Design objektorientierter Software ist es unerlässlich so etwas wie Klassendiagramme oder Interaktionsdiagramme zu erstellen. Hier mit einer einheitlichen Notation zu arbeiten ist wünschenswert, damit sich auch andere Entwickler schnell in das Programm eindenken können, falls sie es einmal ändern oder analysieren müssen. Die in dieser Arbeit angewandte Notation nennt sich Unified Modeling Language. Sie wurde entwickelt von Rumbaugh, Booch und Jacobson. Im folgenden werden wir uns die in dieser Arbeit verwendeten Diagrammtypen etwas genauer ansehen.

Szenario

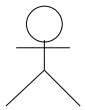
Ein Szenario ist kein Diagramm im herkömmlichen Sinne. Es ist vielmehr ein Text, indem ein Sachverhalt möglicht genau, d.h. mit allen Details beschrieben wird. Es entsteht im Dialog mit den zukünftigen Anwendern des Systems, da diese meist einen besseren Einblick in die Thematik haben als der Entwickler des Softwaresystems. In einem Szenario sollten möglichst viel Sonderfälle enthalten sein, damit diese später beachtet und adäquat implementiert werden können. Ein Szenario ist oft eine Vorstufe zu einem Use Case-Diagramm.

Use Case-Diagramm

Ein Use Case-Diagramm beschreibt das zu erstellende System ähnlich wie ein Szenario, jedoch wird hier schon eine Grafik erstellt, welche das Problem verdeutlichen soll. Die Elemente des Diagramms sind folgende:

Akteur

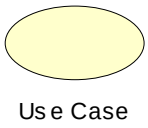
Ein Akteur ist eine Person, die mit dem System in Kontakt tritt. Auch kann das System einen Akteur für bestimmte Aktionen „benutzen“. Ein Akteur steht immer mit mindestens einem Use Case in Beziehung. Die Interaktion mit dem System wird durch einen gerichteten Pfeil gekennzeichnet. Der Pfeil zeigt auf das genutzte Use Case, bzw. den Akteur.



Akteur

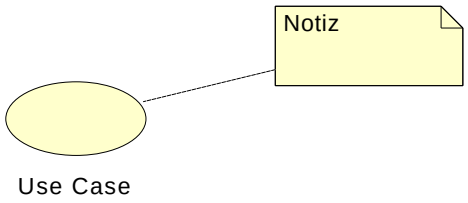
Use Case

Ein Use Case oder auch Anwendungsfall, beschreibt eine Verhaltensweise des Systems. Daher wird ein Use Case immer mit einem Verb beschriftet, da es immer eine „Tätigkeit“ darstellt. Jeder Use Case wird im begleitendem Text genauer beschrieben.



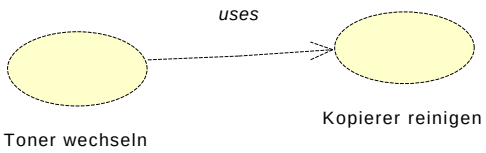
Notiz

Eine Notiz kann mit einer gestrichelten Linie mit jedem Element eines Use Case-Diagramme verbunden werden, wenn es die Verständlichkeit dieses erhöht.



Uses-Beziehung

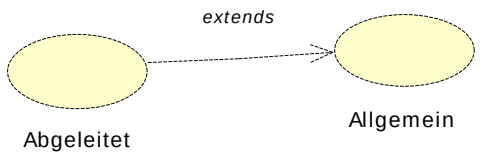
Anwendungsfälle untereinander können ebenfalls Beziehungen eingehen. Mit der Uses-Beziehung wird ausgedrückt, dass jedes Mal, wenn der Ausgangs-Anwendungsfall abgearbeitet wird auch der Ziel-Anwendungsfall der Uses-Beziehung mit abgearbeitet wird. Es ist somit eine Vereinfachung für die spätere textuelle Beschreibung, die somit ggf. kürzer ausfallen kann. Außerdem können hier schon gleiche oder ähnliche Tätigkeiten identifiziert werden.



Extends-Beziehung

Extends ist ähnlich zu definieren wie die Vererbung in der objektorientierten Programmierung. Hier wird ein Anwendungsfall spezialisiert. Man kann sagen, dass ein allgemeinerer Anwendungsfall von spezialisierten „abgeleitet“ wird.

Im Anschluss an das rein grafische Diagramm werden im Text die einzelnen Anwendungsfälle Beschrieben. Aus Übersichtlichkeitsgründen können die Anwendungsfälle auch nummeriert bzw. gegliedert werden.

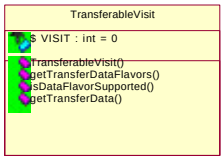


Klassendiagramm

Ein Klassendiagramm beschreibt die Klassenhierarchie und statische Objektstruktur in einem System und die Beziehungen zwischen den Klassen.

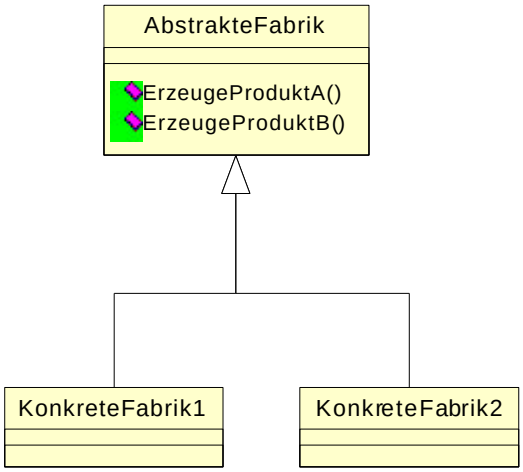
Klasse

Eine Klasse wird grafisch als vertikal dreigeteiltes Rechteck dargestellt. Im oberen Drittel, sozusagen als Überschrift befindet sich die Bezeichnung der Klasse. Im mittleren Drittel finden sich die Attribute der Klasse wieder. Diese werden mit Typ, Name und Sichtbarkeit beschrieben, ggf. werden noch die Werte der Attribute angegeben. Im unteren Drittel werden die Methoden der Klasse verzeichnet. Auch hier werden die wie bei den Attributen die Sichtbarkeit, der Name und der Rückgabetyp beschrieben. Zusätzlich können hier noch die Übergabeparameter aufgeführt werden.



Verallgemeinerungsbeziehung

Mit dieser Beziehungsart wird die Vererbungshierarchie zwischen Klassen dargestellt. Der Pfeil zeigt auf die zu verallgemeinernde Klasse.



Aggregation

Eine Aggregation beschreibt, dass ein Objekt dieser Klasse ein Objekt einer anderen Klasse enthält. Bei dieser Beziehung sollte auch die Kardinalität angegeben werden.

Komposition

Eine Komposition ist im Prinzip das gleiche wie Aggregation, nur hier ist das referenzierte Objekt auch ohne das Ausgangsobjekt existent.

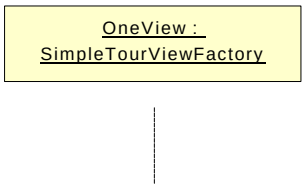
Durch die Beziehungen Aggregation und Komposition wird die Nähe zum ER-Diagramm deutlich, welches von Datenbanksystemen her bekannt ist. Diese werden im folgenden Abschnitt beschrieben.

Sequenz-Diagramme

Ein Sequenz-Diagramm zeigt die Interaktion zwischen Objekten und ist somit Grundlage für die spätere Programmierung. Die Interaktion zwischen Objekten erfolgt bekanntlich durch Nachrichtenaustausch. Dieser Austausch von Nachrichten

Objekt

Ein Objekt wird durch ein Rechteck gekennzeichnet. In diesem Rechteck wird der Objektname und der Typ, also seine Klasse verzeichnet. Das Rechteck kann gleichzeitig die Erzeugung des Objektes darstellen.

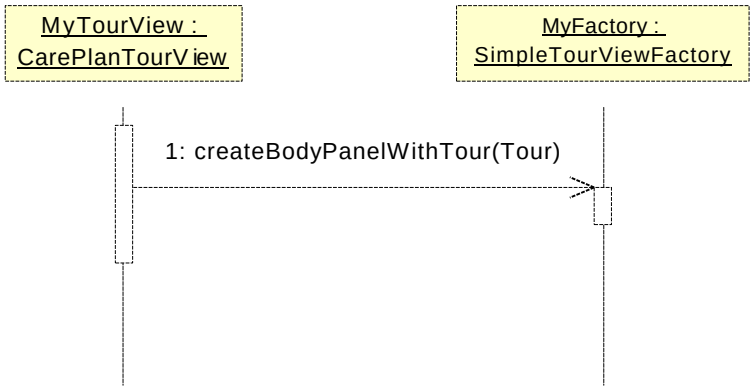


Lebenslinie

Die Lebenslinie des Objektes verläuft vertikal von oben nach unten und beginnt am Objekt-Symbol (eben beschriebenes Rechteck). Sie soll einen chronologischen Zeitverlauf darstellen. Dargestellt wird sie durch eine gestrichelte Linie (s.o.).

Aktion

Tritt ein Objekt in Aktion (z.B. durch erhalten einer Nachricht) wird seine Aktion durch ein Rechteck über der Lebenslinie dargestellt. Nur innerhalb von einer Aktion kann ein Objekt selbst Nachrichten verschicken.



Destruktor

Die Lebenslinie eines Objektes kann innerhalb eines Diagramms mit einem Destruktor enden. Dies bedeutet, dass das Objekt nach diesem Zeitpunkt nicht mehr für andere Objekte verfügbar ist. Natürlich endet eigentlich jede Lebenslinie eines Objektes mit einem Destruktor, nur ist es für das Diagramm nur dann interessant, wenn sich dieser Zeitpunkt innerhalb des betrachteten Bereiches befindet. Der Destruktor wird durch ein Kreuz am Ende der Lebenslinie Symbolisiert.

Nachricht

Eine Nachricht wird mit einem Pfeil gekennzeichnet. An diesen Pfeil wird der Name der Nachricht geschrieben. Das Ende einer Nachricht kann ebenfalls durch einen Pfeil in entgegengesetzter Richtung dargestellt werden, ist aber nur zu empfehlen, wenn es der Aussagekraft des Diagramms deutlich weiterhilft.

Akteur

Auch in einem Sequenzdiagramm gibt es Akteure. Sie haben die gleiche Funktion wie in den Use Case-Diagrammen.

Es gibt noch weiter Diagrammtypen in der UML-Notation, jedoch werde ich mich hier auf diese wenigen beschränken, da ich auch nicht mehr benutzen werde.

Für einen genaueren Überblick verwiese ich an dieser Stelle auf das Buch „Objektorientierte Softwareentwicklung“ von Bernd Oestereich, herausgegeben vom Oldenbourg-Verlag.

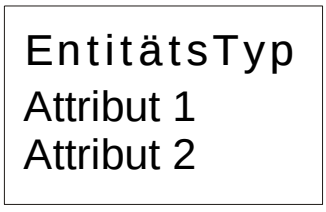
5.1.2. ER-Diagramme

Entity-Relationship (ER) Diagramme werden zur Datenbankmodellierung verwendet. Wie der Name schon sagt werden hier Entitäten und ihre Beziehungen zueinander dargestellt. Eine Menge gleichartiger Entitäten kann man sich vorstellen als eine Tabelle einer Datenbank. Der Entwurf eines solchen Diagramms ist bei der Erstellung einer Datenbank fast unerlässlich, besonders wenn die Datenstrukturen etwas komplexer werden. An einem solchen Diagramm lassen auch die Kardinalitäten der Relationen genau ablesen. Um die Verständlichkeit zu fördern, kann man die Beziehungen unter den einzelnen Entitäten zusätzlich beschriften.

Von der Notation her hat dieser Diagrammtyp viel mit dem Klassendiagramm aus der UML gemeinsam. Hier heißen die Klassen jedoch nur noch Entität und besitzen natürlich auch keine Methoden. Die Sichtbarkeit der Attribute muss natürlich auch nicht mit angegeben werden, da man in einer Relationalen Datenbank keine Datenkapselung kennt.

Die Kardinalität kann auch in der sogenannten Krähenfußnotation angegeben werden. Dabei steht ein „Krähenfuß“ für mehrfach (also n oder m), wobei ein einfacher Strich für 1 steht. Zusätzlich kann auch noch eine Kann-Beziehung ausgedrückt werden. Zu diesem Zweck wird der Teil der in der Beziehung als „Kann“ dargestellt werden soll als gestrichelte Linie dargestellt. Diese kann wiederum auch mit einem Krähenfuß kombiniert werden um eine 0 bis n Beziehung darzustellen. Folgendes Symbol stellt beispielsweise eine cn – 1 – Beziehung dar. Das c steht hier für conditional, also „Kann“.

Eine Entität wird in diesem Diagrammtyp wie folgt dargestellt.



5.2. Objektorientierte Programmierung (OOP)

5.2.1. Datenbanken

Der Zugriff eines Programms auf eine Datenbank sollte nach einem einheitlichen Standard geschehen. Diese Vorgehensweise eröffnet die Möglichkeit die Datenbank unter dem Programm einfach austauschen zu können. Dies ist auch die Grundlage des im folgendem aufgezeichneten Modells eines modernen Computersystems.

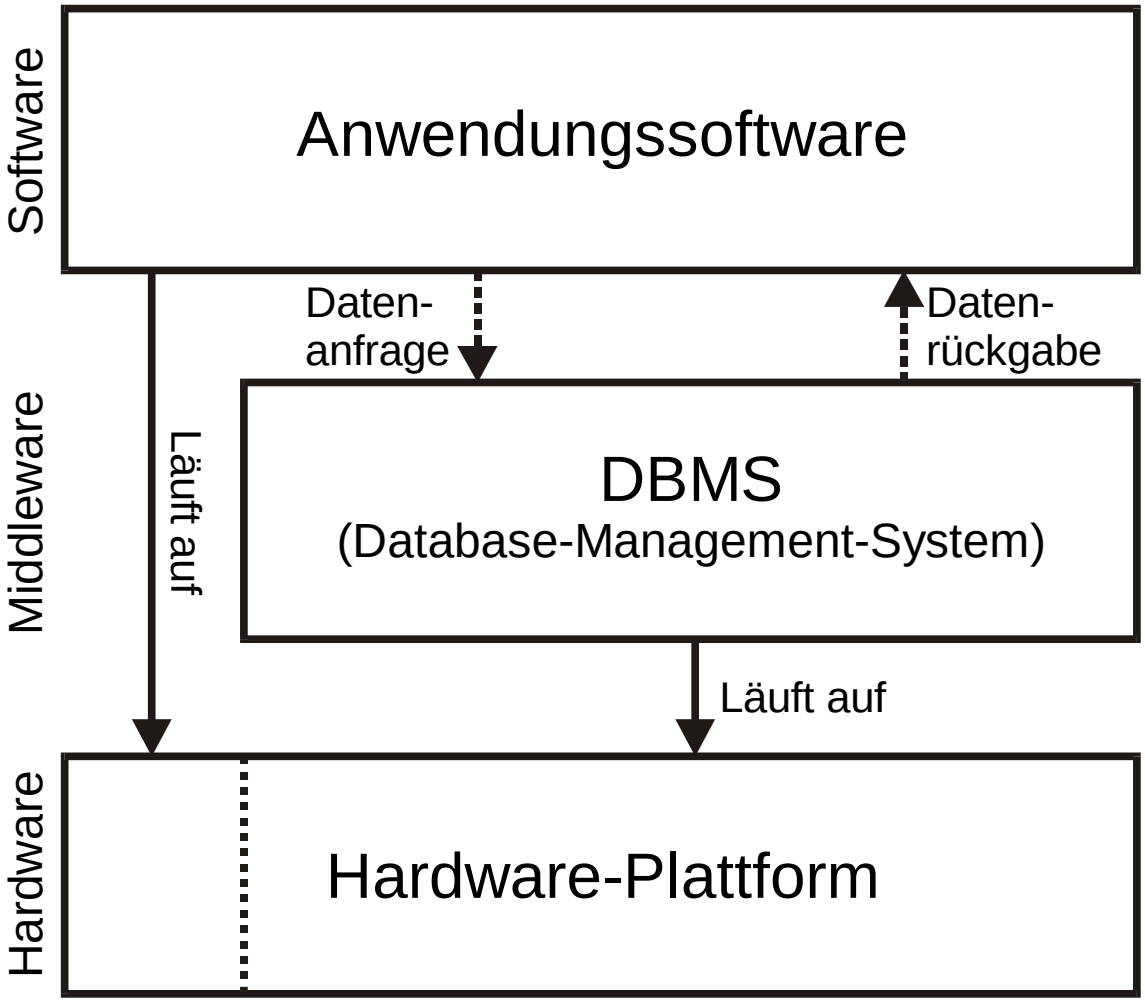


Abbildung 1 Aufteilung in Hardware, Middleware und Software

Die gestrichelte Linie in „Hardware-Plattform“ soll aufzeigen, dass Middleware und Software auch auf verschiedenen Systemen laufen können, sofern sie durch ein Netzwerk verbunden sind.

Mit diesem Schaubild soll verdeutlicht werden, dass diese 3 Komponenten möglichst unabhängig voneinander arbeiten sollten, und sich doch zu einem funktionierenden System ergänzen. Natürlich ist es nicht so zu verstehen, dass Software ohne Hardware funktionieren soll. Dies ist logischerweise nicht möglich. Jedoch sollte die Software auf unterschiedlichen Hardwareplattformen laufen können. Genauso ist es mit der Middleware. Datenbanksysteme gehören in diesem Modell zur Middleware. Also wäre es wünschenswert, wenn eine Software ebenfalls möglichst unabhängig von der Middleware ist. Somit ergibt sich eine größtmögliche

Unabhängigkeit. Um Unabhängigkeit zu erzielen ist es immer notwendig sich auf standardisierte Schnittstellen bzw. Interfaces zu einigen. Das wohl bekannteste und am meisten standardisierte Interface zu Datenbanken ist die „Structured Query Language“, kurz SQL. In dieser „Datenbank-Sprache“ sind alle nötigen Befehle enthalten um Daten aus Datenbanken zu selektieren, zu verändern und zu erzeugen. Auch die Struktur einer Datenbank lässt sich mit dieser Sprache manipulieren. So lassen sich z.B. neue Tabellen oder neue Relationen und Indices erzeugen oder ändern.

5.2.1.1. JDBC

Um in Java auf Datenbanken zugreifen zu können bedient man sich der Funktionalität von JDBC. JDBC heißt Java-DataBase-Connectivity und ist eine Sammlung von Klassen, die den Zugriff auf Datenbanken unterschiedlichsten Typs unterstützen.

Im Programm muss man zunächst einmal den gewünschten Treiber für die Datenbank in den Speicher laden. Da man diesen Treiber nicht mehr direkt ansprechen muss, macht man dies zum Beispiel für die JDBC-ODBC Bridge mit folgendem Java-Code:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Nachdem diese Vorbereitung abgeschlossen ist benötigt man ein Objekt vom Typ Connection, welches das eigentliche Verbindungsobjekt darstellt. Dafür verwendet man folgende Zeile:

```
java.sql.Connection newConnection =
java.sql.DriverManager.getConnection(DatabaseURL, username,
password);
```

DatabaseURL ist vom Typ String und beinhaltet die Lokalisation der Datenbank. Bei einer ODBC-Datenbank kann dies z.B. „jdbc:odbc:swift“ sein. Dabei muss swift auf diesem Computer als Datenbank im ODBC-Treiber registriert sein. Bei „username“ und „password“ handelt es sich auch um Strings die hier zur Authentifizierung im Anmeldevorgang zur Datenbank verwendet werden. Wenn dies nicht benötigt wird, werden hier leere Strings angegeben.

Das so gewonnene Connection Objekt eröffnet die Möglichkeit zum Beispiel Statement Objekte zu erzeugen. Ein Statement Objekt kann SQL-Befehle an die Datenbank senden. Dieses Statement wird wie folgt erzeugt:

```
Java.sql.Statement newstmt = newConnection.createStatement();
```

Nachdem das Statement Objekt erzeugt wurde, kann man es verwenden um entweder eine Abfrage in Form eines einfachen SELECT-Befehls abzusenden oder aber auch die Daten zu verändern, bzw. zu ergänzen mit den SQL Befehlen INSERT und UPDATE. Im Quellcode sieht das wie folgt aus:

```
java.sql.ResultSet AllData= newstmt.executeQuery(SQLSelect);
```

oder für eine Modifikation der Daten:

```
int updateResult = newstmt.executeUpdate(SQLUpdateOrInsert);
```

Im zweiten Fall wird eine Zahl zurückgeliefert, die aussagt wie viele Zeilen der Datenbank von diesem SQL-Befehl betroffen waren.

Der erste Befehl liefert ein ResultSet zurück. Dieses ResultSet beinhaltet alle gewählten Daten die der angegebenen Select-Befehl auswählt. Hier können dann Zeile für Zeile und Spalte für Spalte alle Daten ausgelesen werden. Sollen die gelesenen Daten wieder verändert

in die Datenbank zurückgeschrieben werden, so muss die Methode executeUpdate() für die zu verändernden Daten ausgeführt werden.

5.2.2. Grafische Benutzeroberflächen

Die Benutzeroberfläche ist die Schnittstelle zwischen Mensch und Programm. Daher ist es von enormer Bedeutung, dass diese Schnittstelle möglichst einfach und transparent gehalten wird. Eine gut zu bedienende und schnell erlernbare Oberfläche hat mehrere Vorteile.

- Kurze Einarbeitungszeiten
- Motivation der Benutzer durch rasche Erfolgserlebnisse
- Erhöhte Produktivität durch Vermeidung undurchsichtiger Befehlsfolgen
- Größere Akzeptanz der Benutzer

Einfache Bedienung kann man zum Beispiel durch Annähern von Arbeitsabläufen an die Realität erreichen. Ein einfaches Exempel ist das Löschen eines Dokumentes. Wenn man am Schreibtisch einen Brief nicht mehr benötigt, und auch nicht archivieren möchte, wirft man ihn einfach in den Papierkorb. Diese Aktion kann man am Computer durch eine entsprechende Oberflächenprogrammierung simulieren. Zum Beispiel könnte man mit dem Mauszeiger auf das zu löschende Dokument klicken und es auf ein Papierkorbsymbol ziehen. Diese Verfahrensweise wird bereits in vielen Betriebssystemen mit Erfolg praktiziert. Der Erfolg dieser Methode liegt in dem Bezug zum alltäglichen Leben. Denn was macht man mit einem Dokument das man nicht mehr benötigt? Man wirft es in den Papierkorb. Das einzige was der Benutzer hier lernen muss ist, wie er die Maus zu bedienen hat und dass es wirklich so einfach funktioniert wie in der Realität.

Mit einer zeichenorientierten Darstellung ist es sehr schwierig eine realitätsnahe Benutzeroberfläche zu erschaffen. Dies ist ein Grund für grafische Benutzeroberflächen. Ein anderer Grund ist, dass sich mit Grafik viele Problemstellungen sehr viel einfacher darstellen lassen, als durch textuelle Beschreibungen. Außerdem arbeitet fast jedes moderne Betriebssystem mit einer grafischen Oberfläche, so dass es für den Anwender auch eine gewohntere Umgebung darstellt.

Der Nachteil einer grafischen Oberfläche ist, dass sie sehr viel mehr Systemressourcen für sich beansprucht, sei es nun Arbeitspeicher, wie auch Prozessorleistung. Dies wird schon dadurch klar, dass ein einfacher Buchstabe bei einer grafischen Oberfläche vom System aus vielen einzelnen Punkten aufgebaut werden muss, wobei bei einer zeichenbasierten Oberfläche ein Buchstabe nur aus einem Byte besteht. Die Interpretation dieses Bytes übernimmt die Grafikkarte und entlastet somit den Prozessor. Bei den heutigen Prozessorleistungen und Speicherpreisen ist es jedoch in den meisten Fällen vorzuziehen eine ansprechende Oberfläche zu entwerfen. Die Performanceeinbussen lassen sich meist verschmerzen, da im Normalfall die Anzeige nicht ständig neu dargestellt und berechnet werden muss, und der Prozessor meist nur auf Eingaben vom Benutzer warten muss.

5.2.2.1. Swing

Mit Swing (Beschreibung Swing/JFC Java Foundation Classes) bietet auch JAVA Klassen zum Erzeugen von grafischen Benutzeroberflächen. Die ersten grafischen Benutzeroberflächen von JAVA wurden jedoch mit den AWT (Beschreibung AWT) Klassen erstellt. Diese Klassen benutzten Systemfunktionen, die in allen Betriebssystemen mit grafischer Oberfläche vorhanden waren. Zum Beispiel ist jedes dieser Betriebssysteme in der Lage ein Fenster darzustellen. Außerdem gibt es natürlich auch solche Elemente wie Buttons oder Eingabefelder. Somit bietet AWT eine Schnittmenge der Funktionalitäten aller unterstützter Betriebssysteme.

Mit Swing wird diese Einschränkung durchbrochen, indem jedes Swing-Objekt mit Low-Level Betriebssystemfunktionen dargestellt wird. So wird ein Swing-Objekt mit Punkten, Linien und Rechtecken aufgebaut. Dies benötigt natürlich wiederum mehr Prozessorleistung als AWT, jedoch ist somit ein deutlich größerer Funktionsumfang möglich. Außerdem stellt dieses Verfahren sicher, dass ein einmal geschriebenes Programm auf allen Betriebssystemen (annähernd) gleich aussieht, und der Anwender muss sich nicht an eine neue Optik gewöhnen.

5.2.2.1.1. Layout

Beim Layout eines Programms ist es bei der Fenstertechnik auch wichtig, dass sich die Optik an unterschiedliche Fenstergrößen anpassen kann. So ist es zum Beispiel nicht wünschenswert, dass bei der Verkleinerung eines Fensters wichtige Informationen einfach aus dem Sichtfeld verschwinden. Vielmehr ist es wünschenswert, dass sich in einem solchen Fall der Inhalt des Fensters reorganisiert, so dass auch nach der Verkleinerung alles sichtbar bleibt. Genauso sollen bei einer Vergrößerung des Fensters auch keine großen Lücken entsehen, oder zusammengehörende Informationen auseinanderreißen. Hierzu bietet Swing diverse Layout-Manager. Jedem Container-Objekt, welches andere Objekte beinhalten kann, kann ein Layoutmanager zugewiesen werden. Dieser kümmert sich dann um die Anordnung der Objekte innerhalb dieses Containers. Da auch ein Container in einen andern eingebettet werden kann, lassen sich in dieser Weise verschiedene Layoutmanager miteinander kombinieren.

Im folgenden werden einige Layoutmanager im Detail beschrieben und später an einem Beispielprogramm demonstriert.

BorderLayout

Beim Borderlayout gibt es fünf fest definierte Positionen, an denen ein Objekt platziert werden kann. Zunächst gibt es hier das Zentrum (Center). Dies ist im allgemeinen die Hauptansicht in einem Borderlayout. Die Größe des Zentrums wird bei Größenänderung des Fensters bei Bedarf in alle Richtungen, horizontal und vertikal, geändert. Neben dieser Position gibt es noch vier weitere im Borderlayout, die sich nach den Himmelsrichtungen benennen, nämlich Nord (North), Süd (South), Ost (East) und West (West).Norden liegt über dem Zentrum und erstreckt sich über die gesamte Fensterbreite. Ebenso breit ist Süden, nur dass es unterhalb des Zentrums liegt. Die Höhe von Nord und Süd ergibt sich aus der bevorzugten Höhe des eingebetteten Objektes. Ost und West befinden sich links und rechts vom Zentrum und haben auch die Höhe des Zentrums. Die Breite ist wiederum die bevorzugte Breite der Objekte.

FlowLayout

Beim FlowLayout werden die Objekte wie in einer Textverarbeitung wenn möglich nebeneinander platziert. Die Höhe und Breite der Objekte entspricht den bevorzugten Größen der jeweiligen Objekte. Sollte ein Objekt nicht mehr neben das vorhergehende passen, so wird es in die nächste ‚Zeile’ gesetzt.

GridLayout

Ein GridLayout kann man sich vorstellen wie eine Tabelle mit festgelegter Spalten und Zeilenanzahl. Der Unterschied zu einer normalen Tabelle ist, dass sich nicht einzelne Spaltenbreiten oder Zeilenhöhen verändern lassen. Die Höhe oder Breite einer Spalte wird rechnerisch durch Teilen der Gesamtbreite/Gesamthöhe durch die Anzahl Spalten/Zeilen ermittelt. Somit ist jede Zelle in diesem Grid gleich groß.

Um komplexere Bildschirmszenarien zu entwerfen bedarf es meist der Kombination mehrerer Layouts. So kann ein Panel mit einem FlowLayout beispielsweise als Zentrum in einem Borderlayout agieren. Hiermit ist beliebige Komplexität zu erreichen.

5.2.2.1.2. Designwerkzeuge (z.B. JForge)

Um die Entwicklung einer statischen Oberflächen zu vereinfachen bedient man sich im allgemeinen eines Designwerkzeuges. Mit einem solchen Werkzeug kann man seine Oberfläche fast wie in einem Grafikprogramm zusammenstellen. Wichtig ist hier vor allem bei Java/Swing, dass dieses Werkzeug die unterschiedlichen Layouts unterstützt, und man mit diesen Layouts arbeiten kann. Ein solches Werkzeug erzeugt im Regelfall Java-Code, den man dann in seinem Programm beliebig weiterverwenden kann. Ein kleines Problem bei JForge ist nur, dass man den erzeugten Code nicht viel verändern darf, wenn man ihn ein zweites mal in JForge überarbeiten möchte. Dazu ist es sinnvoll nicht direkt mit den erzeugten Klassen, sondern mit Ableitungen dieser Klassen zu arbeiten. Dadurch bleibt die eigentliche Klasse unberührt, und die Ableitung kann man verändern bzw. erweitern wie man möchte. Ein weiteres wünschenswertes Feature, welches JForge leider nicht unterstützt wäre, dass wenn man den Quelltext verändert man direkt die Änderung auf der Oberfläche sehen könnte.

Es gibt allerdings auch einige Situationen, in denen man mit einem solchen Werkzeug recht wenig anfangen kann. In unserem Falle ist dies z.B. bei den Touren so, da diese dynamisch zur Laufzeit erzeugt werden. Hier ist dieses Werkzeug nur sinnvoll um sich vorher ein Bild darüber zu machen, wie so eine Tour mit Daten gefüllt einmal aussehen könnte.

5.2.2.1.3. Eventhandling

Um bei einer Benutzeroberfläche auf Aktionen des Benutzers reagieren zu können, muss das Programm über diese informiert werden. Diese Aktionen können zum Beispiel Mausklicks, Mausbewegungen, Tastatureingaben aber auch Systemereignisse wie „Fenster schließen“ oder „Fenster minimieren“ sein. Solche Aktionen werden im allgemeinen „Event“ genannt. Im Eventhandling eines Programms wird nun auf solche Ereignisse reagiert. Hier kann beispielsweise die Reaktion auf ein Ereignis wie „Knopf ‚Beenden’ gedrückt“ programmiert sein.

Unter anderem für diese Problematik verwendet Swing das Model-View-Controller Prinzip welches in einem späterem Abschnitt dieser Arbeit genauer erläutert wird. Hier sei nur gesagt, dass dieses Prinzip die Verwendung des Beobachter-Musters impliziert.

Daher wird das Eventhandling wird bei JFC/Swing durch das „Beobachter-Muster“ gelöst. Im Klartext heißt dies, dass sich jedes interessierte Objekt bei einem Objekt, welches Ereignisse produzieren kann anmelden kann. Ein solches Objekt kann, wie in obigem Beispiel, ein Button sein. Damit ein anderes Objekt benachrichtigt werden kann, muss es allerdings auch noch die Vorraussetzungen erfüllen, die das ereigniserzeugende Objekt fordert. Beim Button ist dies, dass es die Methode actionPerformed implementiert. Um dies sicherzustellen muss das Objekt, welches sich anmelden möchte, ein Interface, hier das ActionListener-Interface, implementieren. Somit ist dieses Objekt für den Button ein ActionListener und kann nach der Anmeldung über solche Events mit dem entsprechenden Methodenaufruf informiert werden.

Im folgendem soll diese Funktionsweise des Eventhandlings an einem kleinen Beispiel demonstriert werden. In diesem Programm geht es darum ein Fenster, hier ein JFrame, beim Druck auf einen Button, in diesem Fall vom Typ JButton, zu öffnen. Um das Beispiel nicht unnötig kompliziert zu machen lassen wir einfach einmal alle für dieses Beispiel unnötigen Zeilen Weg. Hier also der Quellcode:

```
import javax.swing.*; //Für Fenster und Button
import java.awt.event.*; //Für ActionListener

public class ButtonBeispiel implements ActionListener{

  //ActionListener Anfang

  //Wird aufgerufen, wenn der Knopf gedrückt wird
  public void actionPerformed(ActionEvent e){
    //Neues Fenster erstellen
    JFrame newFrame=new JFrame("Neues Fenster");
    //Größe des neuen Fensters setzen
    newFrame.setSize(200,100);
    //Neues Fenster anzeigen
    newFrame.show();
  }
  //ActionListener Ende

  //Konstruktor
  public ButtonBeispiel(){
    //Fenster erstellen
    JFrame mainFrame=new JFrame("Startfenster");
    //Button erstellen
    JButton testButton=new JButton("Drück mich!");
    //Beobachter für Button setzen
    testButton.addActionListener(this);
    //Button auf Fenster bringen
    mainFrame.getContentPane().add(testButton);
    //Fenstergröße setzen
    mainFrame.setSize(200,100);
    //Fenster anzeigen
    mainFrame.show();
  }

  //Main Methode zum starten des Programmes
  public static void main(String args[]){
    //Neues "ButtonBeispiel" erzeugen und somit Konstruktor aufrufen
    ButtonBeispiel meinBeispiel=new ButtonBeispiel();
  }
}
```

Der Quelltext ist schon recht ausführlich beschrieben. Ich möchte hier nur noch einmal auf folgende wichtige Zeile etwas genauer eingehen:

```
    //Beobachter für Button setzen
    testButton.addActionListener(this);
```

Diese Zeile fügt dem Button einen ActionListener hinzu. Die Klasse ButtonBeispiel ist durch das implementieren des Interfaces ActionListener zu einem solchen geworden. Dazu gehört, dass die Methode actionPerformed(ActionEvent e) implementiert wird. Wenn nun der Button „testButton“ auf dem Frame „mainFrame“ gedrückt wird, werden alle ActionListener, die sich an dem Button angemeldet haben, also auch ButtonBeispiel informiert, indem die Methode actionPerformed(ActionEvent e) der ActionListener aufgerufen wird. In dieser Methode wird in unserem Beispiel lediglich ein neues Fenster ohne Inhalt geöffnet. Man könnte mit der übergebenen Variable e vom Typ ActionEvent jetzt noch abfragen, welcher Button gedrückt wurde, jedoch ist dies hier nicht nötig, da es nur einen Button gibt.

5.2.2.1.4. Adapter-Klassen

„Adapter“ ist ein Entwurfsmuster, welches einem Objekt die Benutzung einer Klasse ermöglicht, die eigentlich gar nicht für dieses Objekt geschrieben wurde. Dazu ummantelt die Adapterklasse die zu verwendende Klasse und bietet dem benutzendem Objekt eine Schnittstelle, mit der es etwas anfangen kann.

Mit Hilfe dieser Adapterklassen kann man auch anonyme Klassen/Objekte erzeugen, was mit einem Interface natürlich nicht möglich ist. Dies kann bei der Programmierung einiges an Zeit sparen und macht das Programm unter Umständen sogar besser lesbar.

Nun gibt es in den JFC auch Listener-Interfaces die recht ausführlich sind, also recht viele Methoden enthalten. Da bei der Implementierung eines Interfaces alle Methoden programmiert werden müssen, kann es unter Umständen ziemlich umständlich sein, wenn man nur eine einzige Methode benötigt. Andererseits ist es manchmal auch nicht nötig das Objekt, welches die Events verarbeitet, für andere Objekte zugänglich zu machen. Auch für diese Fälle gibt es die Adapter-Klassen in Swing. Diese Adapterklassen sind Klassen, die ein einzelnes Listener-Interface implementieren, allerdings ohne jegliche Funktion hinter den einzelnen Methoden. Der Vorteil hierbei ist, dass man von diesen Klassen einfach mit extends ableiten kann und nur die Methode überschreiben muss, die man wirklich benötigt.

Als Beispiel hierfür können wir das im letzten Kapitel angefangene Programm um eine kleine Funktion erweitern. Wir können z.B. das Programm beenden, wenn das Hauptfenster geschlossen wird. Dazu müssen wir das Programm wie folgt erweitern:

```
...
...
//Konstruktor
public ButtonBeispiel(){
  //Fenster erstellen
  JFrame mainFrame=new JFrame("Startfenster");
  //Button erstellen
  JButton testButton=new JButton("Drück mich!");
  //Beobachter für Button setzen
  testButton.addActionListener(this);
  //Button auf Fenster bringen
  mainFrame.getContentPane().add(testButton);
  //Fenstergröße setzen
  mainFrame.setSize(200,100);

  //Neuer Teil: Programm beenden wenn Fenster geschlossen wird.
  mainFrame.addWindowListener(new WindowAdapter(){ //Annonyme Klasse
       //Überschriebene Methode von "WindowAdapter"
       public void windowClosing(WindowEvent e){
         //Befehl zum Abbruch des Programmes
         System.exit(0);
       }
    }
```

```
    );

    //Fenster anzeigen
    mainFrame.show();
  }
  ...
  ...
```

Wie man hier sieht ist eine anonyme Klasse zur Anwendung gekommen. Diese anonyme Klasse ist abgeleitet von WindowAdapter. Dieser WindowAdapter implementiert alle Methoden eines WindowListeners. Wenn man jetzt nur auf ein bestimmtes Ereignis, wie in unserem Falle beispielsweise auf windowClosing, reagieren möchte, so überschreibt man hier nur die gewünschte Methode und braucht sich um die anderen nicht zu kümmern. Wenn man die Klasse ButtonBeispiel das Interface WindowListener implementieren lassen würde, so müsste man ganze sieben Methoden implementieren, obwohl man doch eigentlich nur in einer Methode den Befehl „System.exit(0)“ schreiben wollte. Dies würde das Programm nur unübersichtlicher machen.

5.3. Verteilte Programmierung

Unter einem verteilten System versteht man ein System, dessen einzelne Module auf getrennten Rechnern lauffähig sind, aber dennoch eine gemeinsame Datenbasis nutzen. Der Unterschied zu einem einfachen mehrbenutzerfähigem Programm besteht in der Modulaufteilung, und darin, dass sich die einzelnen Module bei Bedarf aktuelle Daten auf den Rechner laden, danach im offline Modus mit diesen Daten arbeiten und das Ergebnis wieder in die gemeinsame Datenbasis zurückschreiben können.

Das Hauptproblem bei der Programmierung eines verteilten Systems ist die Koordination der Einzelmodule und der Zugriff auf eine gemeinsame Datenbasis.

Dazu gibt es verschiedene Lösungsansätze, die im folgenden dargestellt werden sollen.

5.3.1. Der Objektmanager

Der Objektmanager ist ein Objekt, welches alle Datenobjekte eines Programms verwalten soll. Jedes Mal, wenn vom Programm beispielsweise Daten aus einer Datenbank angefordert werden, wendet sich die entsprechende Komponente an den Objektmanager und fragt nach diesen Daten. Der Objektmanager selbst durchsucht nun die Datenbank nach den gewünschten Daten, konstruiert aus diesen ein oder mehrere Objekte und sendet sie zurück an die Komponente. Dieser Ansatz hat mehrere Gründe.

Zum Einen können somit Konflikte zwischen konkurrierenden Komponenten, die auf die gleiche Datenbasis zugreifen, einfacher gelöst werden. Wenn beispielsweise sichergestellt werden soll, dass ein bestimmtes Objekt von nur einem Client gleichzeitig bearbeitet werden darf, so kann der Objektmanager dieses Objekt für alle anderen Clienten sperren, bis der Client das Objekt wieder zurückgegeben hat.

Ein anderer Aspekt ist, dass sich der Objektmanager nicht einmal auf dem gleichen System wie die Klienten befinden muss. Er kann genauso gut über RMI (Remote Method Invokation) angesprochen werden. Die Objekte werden dann serialisiert zum Klienten übertragen. Bei Bedarf kann der Klient nach der Transaktion die Verbindung zum Objektmanager trennen und später zur evtl. nötigen Objektrückgabe die Verbindung erst wieder herstellen. Wie RMI funktioniert, wird im folgendem Abschnitt genauer beschrieben.

Wenn man beim Objektmanager RMI mit dem Beobachtermuster kombiniert, welches weiter unten noch genauer erläutert wird, lässt sich sogar eine Kommunikation in beiden Richtungen realisieren. Dazu ist es allerdings notwendig, dass auch der Client als RMI-Server fungiert. Wenn dies der Fall ist, so kann der Objektmanager diesen Client direkt bei Änderung eines Objektes von einem andern Klienten aus informieren. Der Client selbst ist dann dafür verantwortlich, was er mit dieser Benachrichtigung macht, ob er seine Anzeige aktualisiert oder ob er die Benachrichtigung vielleicht doch ignorieren möchte. Ein Nachteil hieran ist, dass eben der Client auch einen RMI-Server haben muss, was wiederum evtl. knappe Ressourcen vergeuden könnte. Darum sollte dieses Verfahren nur angewandt werden, wenn der Client mit sehr kritischen Daten arbeitet, die auf jeden Fall ständig auf dem neuestem Stand gehalten werden sollen.

5.3.2. Remote Method Invokation (RMI)

Remote Method Invokation ist ein Protokoll von Java um Methoden auf einem entfernten Objekt aufrufen zu können. Entfernt heißt in diesem Fall durch ein Netzwerk getrennt. Diese Methode des entfernten Objektes kann dann auch Daten an den Klienten zurückliefern.

Vergleichbar ist diese Technologie in etwa mit der CORBA-Architektur. CORBA hat nur den Vorteil dass es nicht an eine spezielle Sprache gebunden ist, wogegen RMI nur für Java zu Verfügung steht. Da wir aber ausschließlich mit Java programmieren, müssen wir uns nicht eines solchen Schwergewichtes wie CORBA bedienen.

Um RMI zu ermöglichen muss ein Server die Objekte, von denen Methoden ausgeführt werden sollen bereithalten, und Anfragen an diese Objekte weiterleiten, sowie das Ergebnis dieser Operation zurücksenden. Ein solcher Server, der von SUN Microsystems selbst zu Verfügung gestellt wird nennt sich „RMIRegistry“. Dieser Server muss nun auf dem RMI-Host gestartet werden. Von nun an können Java-Programme Objekte an diesem Server anmelden. Methoden dieser Objekte können jetzt auch von anderen Computern, die über ein Netzwerk mit diesem verbunden sind, aufgerufen werden. Dazu verwenden sie die Lookup-Methode des RMI-Servers um nach einem entsprechendem Objekt zu suchen. Sollte es gefunden werden, so kann das entfernte System dieses Objekt instanzieren.

Um die Arbeit mit diesen Remote-Objekten zu ermöglichen werden von der Remote-Klasse mit dem speziellen RMI-Compiler von Java zwei zusätzliche Klassen erzeugt. Zum Einen ist dies das Skeleton, welches vom Server benutzt wird, und so diesem zugänglich sein muss und zum anderen der Stub, der den Klienten unterstützt. Diese beiden Klassen sind nun für den Transport der Daten für diese eine Klasse zuständig. Durch ihre Funktionalität wird dieser Datentransport quasi automatisiert, und der Programmierer muss sich nicht mehr darum kümmern. Wichtig für den Programmierer ist nur zu wissen, dass die Remote-Klasse serialisierbar sein muss um transportiert werden zu können.

5.3.3. Enterprise Java Beans (EJB)

Mit EJB wird nun RMI mit Datenbankfunktionalitäten verbunden. Objekte, die bisher über RMI aufgerufen werden konnten, werden in EJB nun als Beans implementiert. Diese Beans können nun auf Wunsch auch persistent gemacht werden, indem ihr Zustand in einer Datenbank gespeichert wird.

Das Binden von Objekten bzw. EJB an eine Datenbank geschieht erst mit dem Deployment. Erst hier wird festgelegt welche Datenbank zur Speicherung benutzt wird. Die Flexibilität wird noch mehr gesteigert, indem erst hier festgelegt wird mit welchem SQL-Statement die Daten eines bestimmten Objektes gespeichert werden sollen. Somit ist sogar eine Anpassung an bestehende Datenbanksysteme und Tabellen möglich.

An dieser Stelle ist es erst einmal wichtig zu erfahren, dass der Entwicklungsprozess von Software mit der EJB-Technologie sich von herkömmlicher Softwareentwicklung unterscheidet. Wir haben hier nicht nur den Anwendungsentwickler, den Datenbankadministrator und den User. Es kommt noch eine Personengruppe hinzu, und zwar der Deployer. Er ist dafür verantwortlich, das Zusammenspiel zwischen den EJB zu koordinieren und aus diesen eine lauffähige Anwendung zusammenzusetzen. Die EJB selbst werden weiterhin vom Anwendungsentwickler entworfen, wobei dieser Name eigentlich nicht mehr ganz zutrifft. Vielmehr werden hier nicht ganze Anwendungen entwickelt, sondern nur noch einzelne Komponenten.

Der Deployer verwendet für seine Arbeit ein sogenanntes Deploymenttool, mit dem er die EJB auf den Applikationsserver aufspielt und koordiniert.

Eine weiter wichtige Eigenschaft von EJB ist die „Three-Tier-Technologie“. Darunter versteht man eine Aufteilung eines Software-Systems in 3 Schichten. Zunächst einmal wäre hier die Schicht in der sich der Klient befindet. Im Gegensatz zu den herkömmlichen „Two-Tier“ Systemen befindet sich in dieser Schicht jedoch keine Business-Logik. Diese befindet sich erst in der mittleren Schicht (Middle-Tier). Die Klientenschicht zeichnet sich in EJB dadurch aus, dass sie sehr einfach gehalten werden kann, ja sogar als einfache HTML-Seite gestaltet werden kann. Die eigentlichen Enterprise JavaBeans befinden sich in der mittleren Schicht. Sie beinhalten, wie eben schon erwähnt, die komplette Business-Logik. In der letzten Schicht befindet sich nun das Informationssystem des Unternehmens (Enterprise Information Service). Nur die EJBs kommunizieren mit dieser Schicht, so dass der Klient gar nicht direkt mit dieser Schicht in Kontakt tritt.

5.3.4. Zusammenfassung

Nach unseren Tests innerhalb des Programmierteams der Projektgruppe SWIFT sind wir zu dem Entschluss gekommen für unsere Prototypen zunächst den Objektmanager mit RMI-Unterstützung zu implementieren.

Zum Einen hatten wir nur den Application-Server von SUN selbst zu Verfügung, der zudem noch in der Beta-Phase seiner Entwicklung steckte. Das Deployment-Tool, welches hierzu ausgeliefert wurde, war zudem noch recht umständlich zu bedienen, was die Arbeit hiermit vor allem bei Fehlern in der Programmierung, welche bekanntlich beim Prototypisieren häufiger auftreten, enorm erschwerte.

Zum Anderen ist die Fehlersuche bei Verwendung eines solchen Systems, welches viele Dinge selbständig übernimmt, sehr viel schwieriger, als bei einem System, das man von Grund auf kennt, da man es selbst entworfen hat. Wahrscheinlich ist es erst einmal notwendig, mehr Erfahrungen in kleineren Projekten mit EJB zu sammeln, um die daraus gewonnenen Erkenntnisse in größeren Projekten anwenden zu können.

Einen Vorteil, den uns die EJBs im Gegensatz zu dem hier vorgestellten Objektmanager nicht bieten können, ist die Kommunikation in beiden Richtungen. Hierfür sind die EJBs nicht ausgelegt. Hier ist jedoch noch nicht vollständig geklärt, ob wir eine solche Kommunikation überhaupt benötigen. Sicher ist nur, dass eine Lösung mit dem Objektmanager sehr viel weniger Ressourcen benötigt als eine Lösung mit EJB. Und die Entwicklung eines

Objektmanagers ist auch nicht aufwendiger als die Entwicklung von EJBs und der zugehörige Deployment-Prozess.

Dennoch sollte man die Entwicklung der EJBs nicht ganz aus den Augen verlieren, da sich viele Experten einig sind, dass in dieser Entwicklung die Zukunft liegt.

An dieser Stelle sei gesagt, dass im Swift-Projetk zwar vorerst der Objekt-Manager eingesetzt werden soll, jedoch werden wir auf die Anbindung verzichten, da dieser auch noch nicht fertiggestellt ist. Wie wir später sehen werden, geschieht die Datenbankanbindung hier über eine einzige Klasse, so dass diese auch später noch leicht auf die Verwendung eines Objektmanagers umgestellt werden kann.

6. Theoretische Grundkonzepte

6.1. Balanced Scorecard

Bei der Balanced Scorecard (BSC) handelt es sich um ein Management-Instrument. Wie jedes Management-Instrument hat es zum Ziel die Leistung des Unternehmens in verschiedenen Bereichen zu verbessern. Dabei wird von verschiedenen Sichten auf das Unternehmen ausgegangen. Diese Sichten gehen von verschiedenen Perspektiven aus. Die Entwickler des Systems gehen hier im allgemeinen von vier Sichten aus. Diese Perspektiven sind die Sicht des Kunden, der Wirtschaftlichkeit, der Produktion und der Mitarbeiter. Diese Sichten sollen zwar nur einen Anhaltspunkt geben, und sind auch nicht zwingend vorgeschrieben, doch kann man sich an ihnen gut orientieren. Daher hier die genauere Erläuterung der einzelnen Sichten.

Die Kundenperspektive

Die Kundenperspektive repräsentiert die Sicht des Kunden auf das jeweilige Unternehmen. Diese Sicht auf das Unternehmen ist durch folgende Spätindikatoren gekennzeichnet:

„Kundenzufriedenheit, Kundentreue, Neukundenakquisition, Kundenrentabilität und Marktanteil“[11]

Die Geschäftsprozeßperspektive

Bei der Geschäftsprozeßperspektive geht es um die ganzheitliche Erfassung des Hauptgeschäftsprozesses. Es fängt an bei der Identifikation und Umsetzung von Kundenwünschen. Es kann nur sinnvoll gearbeitet werden, wenn man zielgerichtet die Bedürfnisse der Kunden erfüllt, und nicht an ihnen vorbeiproduziert. Direkt anschließend hieran ist die betriebliche Leistungserstellung zu nennen, welche von der Warenbeschaffung über die Produktion bis hin zum Verkauf führt. Nach dem eigentlichem Absatz der Ware beim Kunden ist ein Kundendienst auch von großer Bedeutung. Hierbei geht es um die Nachsorge beim Kunden. Daraus entsteht eine Kommunikation mit dem Kunden, aber auch eine interne Kommunikation, die wiederum Ausgangspunkt für die Identifikation weiterer Kundenwünsche sein kann. Somit schließt sich dieser Prozess zu einem sich verfeinerndem Kreislauf.

Die Mitarbeiterperspektive

Hierbei handelt es sich um die Sicht des Unternehmens auf seine Mitarbeiter. Die Zufriedenheit der Mitarbeiter wirkt sich im allgemeinen positiv auf ihre Produktivität aus. Daher ist dieser Spätindikatoren auch Bestandteil der Mitarbeiterperspektive. Ein weiterer Indikator kann beispielsweise der Fortbildungsstand der Mitarbeiter sein, welcher eher zu den Frühindikatoren zählt.

Die Finanzperspektive

[11] „Balanced Scorecard – Mehr als ein Kennzahlensystem“ von Herwig R. Freidag und Walter Schmidt, Seite 113

In der Finanzperspektive wird das strategische Ziel eines Unternehmens in die Sprache der Anteilseigner übersetzt.[12] Somit fasst diese Sicht auf das Unternehmen die anderen Perspektiven für die finanzielle Seite zusammen. Es gibt bereits diverse Kennzahlen für diesen Bereich. Wichtig ist es hier diejenigen mit strategischer Bedeutung herauszufiltern.

Unter dem Gesichtspunkt, dass man eine BSC für jede Abteilung einführen sollte, lässt sich der Aspekt der unterschiedlichen Sichten sehr gut in unser System einbauen.

Wenn wir, wie in dieser Arbeit beabsichtigt, von der Pflegeeinsatzplanung ausgehen, können wir beispielsweise die Kundenperspektive wie folgt beschreiben:

Dem zu pflegenden Kunden ist es zunächst einmal wichtig, dass er die geplanten Leistungen möglichst zu dem Zeitpunkt erhält, zu der er sie geplant hat. In manchen Fällen kann dies sogar primär wichtig sein, wenn zum Beispiel ein Medikament zu einer bestimmten Uhrzeit gegeben werden muss. Desweiteren ist es aus Sicht des Kunden bestimmt wünschenswert, wenn er zu seinem Pfleger ein persönliches Verhältnis aufbauen kann. Dies ist nur dann möglich, wenn die Pflegekräfte bei einem Kunden nicht allzu häufig wechseln. Dies sind wohl die beiden wichtigsten Faktoren innerhalb dieser Perspektive.

Mit der Geschäftsprozeßperspektive kann man während der Pflegeeinsatzplanung nicht sehr viel anfangen, da dieser Prozess wohl zu klein für diese ganzheitliche Perspektive ist. Hier kann das System lediglich bei der Planung unterstützen, indem es alle Vorbereitungen zur Planung übernimmt (z.B. Patienten- und Mitarbeiterkärtchen erstellen) und bei der Planung gewisse Hilfestellungen leistet.

Die Mitarbeiterperspektive wiederum kann sehr gut unterstützt werden. Für die Mitarbeiter ist es z.B. wünschenswert, dass sie nicht allzu große Lücken zwischen den einzelnen Besuchen haben, dass eine Tour nicht über die eigentliche Arbeitszeit hinausgeht. Auch sollte der Dienst gerecht auf alle Mitarbeiter verteilt werden. Des weiteren sollte die Planung für den Mitarbeiter früh genug zugänglich sein, so dass er sich darauf einstellen kann. Die Beachtung der für eine Tour benötigten Qualifikation ist nicht nur für den Mitarbeiter selbst wichtig.

Innerhalb der Finanzperspektive können dann, wie schon weiter oben erwähnt, die verschiedenen Perspektiven in finanziellen Kennzahlen zusammengefasst werden. Hier würde es sich zum Beispiel anbieten, die Touren aus der Tourenplanung (Mitarbeiterperspektive) auf ihre Wirtschaftlichkeit hin zu überprüfen, oder aber jeden einzelnen Besuch bzw. Besuche eines Kunden auf wirtschaftliche Gesichtspunkte abzuprüfen.

Daraus ergibt sich, dass vor allem die Sicht des Mitarbeiters und die Sicht des Kunden auf die Planung wichtig ist. Die Wirtschaftlichkeitssicht kann auf allen Sichten mit eingeblendet werden.

Für die spätere Implementierung ist es wichtig zu wissen, dass als Kennzahlen für die einzelnen Sichten verschiedene Dinge denkbar sind. Daher ist es von Vorteil eine Schnittstelle für Kennzahlen, die in Zukunft wichtig sein könnten, offen zu implementieren.

[12] „Balanced Scorecard – Mehr als ein Kennzahlensystem“ von Herwig R. Freidag und Walter Schmidt, Seite 183

6.2. Model-View-Controller

Das Model-View-Controller Konzept wird unter anderem in Swing zur Realisierung der grafischen Oberfläche benutzt. Ursprünglich wurde es jedoch in Smalltalk erstmals angewandt. Es beschreibt ein Muster, welches aus den drei Klassentypen Model, View und Controller besteht. Das Model beschreibt hierbei die Daten, oder auch Anwendungsklasse. In diesem Model werden die von der Anwendung benötigten Daten gekapselt zentral verwaltet. Für die Darstellung der Daten eines Models sind die Views zuständig. Dabei kann ein Model beliebig viele Views haben. Ein View greift über die Methoden des Models auf die Daten zu. Auf Benutzereingaben reagiert der Controller.

Werden nun Daten des Models durch Benutzereingaben (oder durch andere Ereignisse) geändert, so benachrichtigt er alle seine Views, um ihnen die Möglichkeit zu geben sich auf einen konsistenten Stand zu bringen. Dazu wird das Beobachtermuster verwendet, welches weiter unten im Kapitel über Entwurfsmuster genauer beschrieben wird.

Der große Vorteil dieses Konzeptes liegt in der Entkopplung dieser drei Klassen. Ein Model muss gar nicht mehr von vornherein wissen, welche Views es aktualisieren muss. Diese Kopplung entsteht erst zur Laufzeit. Die Views selbst sind sozusagen für ihre Aktualisierung verantwortlich und erhalten nur einen Anstoß vom Model.

Um dieses Vorgehen etwas zu verdeutlichen stellt man sich ein Model vor, welches zwei Views besitzt. In einem View ist eine Tabelle mit Werten abgebildet und in der anderen View ein zugehöriges Balkendiagramm. Wenn nun die Werte in der Tabelle geändert werden, geschieht dies natürlich über das Model. Das Model wiederum benachrichtigt alle Views, also die Tabelle und das Balkendiagramm. Beide Views aktualisieren sich nun automatisch mit den eben veränderten Werten.

Ein weiterer Vorteil ist die Erweiterbarkeit eines solchen Systems. Dies ist möglich, da das Model nichts über seine Bildschirmrepräsentation spezifiziert. Aus diesem Grund lassen sich sehr einfach neue Views erstellen und an das System anbinden.

Es ist auch vorstellbar an das Model zu benachrichtigende Komponenten anzubinden, die nicht für die Bildschirmdarstellung verantwortlich sind. Denken wir doch einfach mal an die Datenbankanbindung unseres Systems. Auch das hierfür zuständige Objekt kann auf diese Art und Weise mit dem Model verbunden werden. So ist sichergestellt, dass es alle Änderungen mitbekommt und es ist einfach eine Datenbankverbindung mit einer anderen auszutauschen, oder wenn es sinnvoll sein sollte sogar mehr als einen Datenbankanschluss gleichzeitig zu verwenden.

Wir halten also fest, dass es bei diesem Model im Prinzip darum geht ein Model mit mehreren Views zu versehen und die Verbindungen konsistent zu halten.

6.3. Zusammenfassung

Wenn man sich obige zwei Konzepte ansieht, fällt direkt ins Auge, dass beide von verschiedenen Sichten auf eine Basis ausgehen. Aus diesem Grund ist es wohl auch sinnvoll die Einbindung des Konzeptes der Balanced Score Card durch das Model-View-Controller Konzept zu implementieren. So lassen sich einfach beispielsweise die Kunden- und die Mitarbeitersicht implementieren. Auch ist es später einmal einfach, wenn einem noch zusätzliche Sichten einfallen, bzw. wichtig erscheinen, auch diese in das System einzubinden, ohne dass große Probleme auftreten, die durch eine zu enge Kopplung von Daten und Darstellung entstanden sind. Wie dieses Konzept im Detail implementiert wird, wird später im Abschnitt Softwarerealisierung beschrieben.

7. Softwarerealisierung

7.1. Use Case Diagramme

Module des Programms

Um ein Softwareprojekt auf ein Programmiererteam aufteilen zu können muss es zunächst modularisiert werden, da es keinen Sinn macht, und nur wertvolle Ressourcen verschwendet, wenn viele Softwareentwickler an dem gleichen Programm arbeiten und keine klaren Schnittstellen definiert sind. Dies führt zu einer Verlagerung der Arbeit von produktiver Programmierung zu ständiger Kommunikation und Abstimmung innerhalb des Teams. Es ist viel sinnvoller, wenn die Modularisierung möglichst bald nach gemeinsamer Einarbeitung in das Thema erfolgt. Danach kann die Arbeit am Projekt viel besser und einfacher fokussiert werden.

Im folgendem Diagram sind alle Module des Programms aufgeführt.

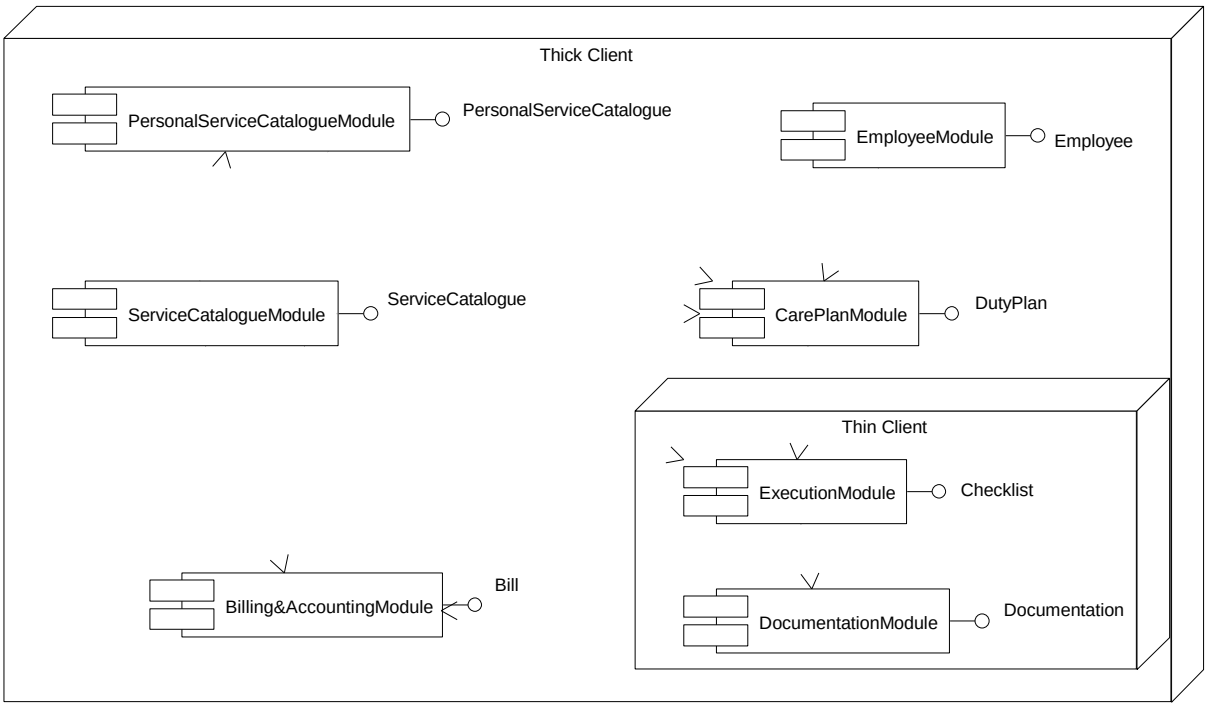


Abbildung 2 Modulaufteilung der Software für ambulante Altenpflegeeinrichtungen

Bei UML werden Szenarios zunächst in Use Case Diagrammen dargestellt. Der Sinn dieser Diagramme ist es neben der Veranschaulichung der Vorgänge im Szenario auszuführen, welche Akteure in den Anwendungsfällen eine Rolle spielen, wie das Szenario in kleinere, in sich geschlossene Anwendungsfälle aufgeteilt werden kann.

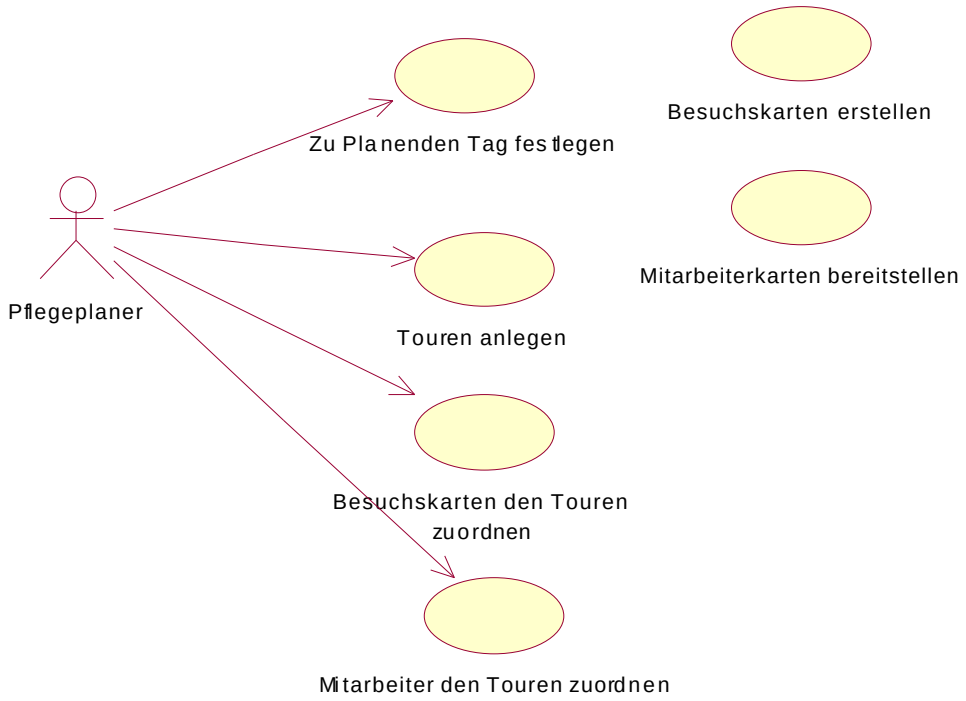


Abbildung 3 Use Case Diagramm Pflegeplanung durchführen (Tourensicht)

Bei diesem Use Case Diagramm handelt es sich um den ersten, den herkömmlichen Fall, nämlich der Planung aus der Sicht der Touren eines Tages.

Hier lässt sich auf einem Blick erkennen, dass bei der Aktivität der Personaleinsatzplanung nur der Akteur „Pflegeplaner“ beteiligt ist. Im folgenden werden nun die einzelnen Use Cases etwas detaillierter betrachtet.

Zu planenden Tag festlegen

Hier ist der Pflegeplaner direkt beteiligt. Er muss zu Beginn der Planung erst einmal festlegen, welchen Tag er planen möchte. Daher kommt dieser Schritt auch ganz zu Anfang. Dieser Schritt startet automatisch die beiden nächsten Use Cases.

Besuchskarten erstellen

Mit Besuchskarten sind in Anlehnung an die herkömmlichen Plantafeln die Kärtchen für einen Besuch bei einem Pflegebedürftigen gemeint. Die hierfür benötigten Daten werden für den gewählten Tag automatisch aus den persönlichen Leistungskatalogen der Pflegebedürftigen gelesen. Ein persönlicher Leistungskatalog stellt die im Pflegevertrag gebuchten Leistungen für einen Pflegebedürftigen dar. Aus den enthaltenen Einzelleistungen wird unter Verwendung des zugrundeliegenden Leistungskataloges die Dauer des Besuches errechnet. Im Leistungskatalog sind alle zur Verfügung stehenden Leistungen des Pflegedienstes verzeichnet. Hier sind auch für jede Dienstleistung Durchschnittswerte für die Dauer der einzelnen Leistungen angegeben. Zu den gebuchten Leistungen im persönlichen Leistungskatalog sind auch die Uhrzeiten angegeben, zu denen sie ausgeführt werden sollten. Dies kann auch ein Zeitraum sein (zum Beispiel „Vormittags“). Aus diesen Daten werden dann die Besuchskarten erzeugt. Auf der Besuchskarte ist dann zu sehen, um welchen

Pflegebedürftigen es sich handelt und wann die Dienstleistungen erbracht werden sollen. Die Dauer der Dienstleistungen spiegelt sich in der vertikalen Ausdehnung der Karte wieder.

Mitarbeiterkarten bereitstellen

Auf der anderen Seite stehen die Mitarbeiter. Auch die Verfügbarkeitsdaten der Mitarbeiter sind unerlässlich für die Pflegeeinsatzplanung. Entsprechend Ihren Dienstzeiten und Abwesenheitszeiträumen werden die Mitarbeiterkarten erstellt und bereitgestellt. Auch bei dieser Tätigkeit ist der Akteur „Pflegeplaner“ nicht direkt beteiligt. Sie wird automatisch nach Datumsauswahl ausgeführt.

Touren anlegen

Ein Pflegebedürftiger kann nur angefahren werden, wenn er in einer Tour aufgeführt ist. Daher müssen jetzt die Touren angelegt werden. Die Tourdaten, die jetzt festgelegt werden müssen bestehen aus Anfangszeit und Länge der Tour. Für diese Tätigkeit ist der Pflegeplaner zuständig.

Besuchskarten den Touren zuordnen

Nachdem diese Vorbereitungen getroffen wurden, kann in diesem Use Case die Zuordnung von Pflegebedürftigen zu Touren erfolgen. Diese Aktion führt wiederum der Pflegeplaner aus. Dabei zieht er die bereitgestellten Besuchskarten auf die angelegten Touren. Mit der Zielposition des „Zieh-Vorganges“ kann die Uhrzeit des Besuches beeinflusst werden. Bei der Zuordnung wird dem Benutzer, also dem Pflegeplaner, die Arbeit dadurch erleichtert, dass ihm angezeigt wird, in welchem zeitlichen Bereich er den Besuch planen kann. Natürlich kann er dies auch ignorieren, wenn es eine Planung nicht anders zulässt.

Mitarbeiter den Touren zuordnen

Damit eine Tour auch ausgeführt werden kann, muss ihr noch ein Mitarbeiter zugewiesen werden. Dies geschieht wie mit den Besuchskarten jetzt mit den Mitarbeiterkarten. Dabei werden die zu Verfügung stehenden Mitarbeiterkarten auf die Kopfbereiche der Touren gezogen. Dabei wird eine Prüfung daraufhin durchgeführt, ob der Mitarbeiter die entsprechende Mindestqualifikation für diese Tour besitzt, und ob der Mitarbeiter zu diesem Datum den entsprechenden Dienst hat. Auch diese Aufgabe übernimmt der Pflegeplaner.

Das folgende Diagramm zeigt die Pflegeeinsatzplanung unter Benutzung der Pflegebedürftigensicht.

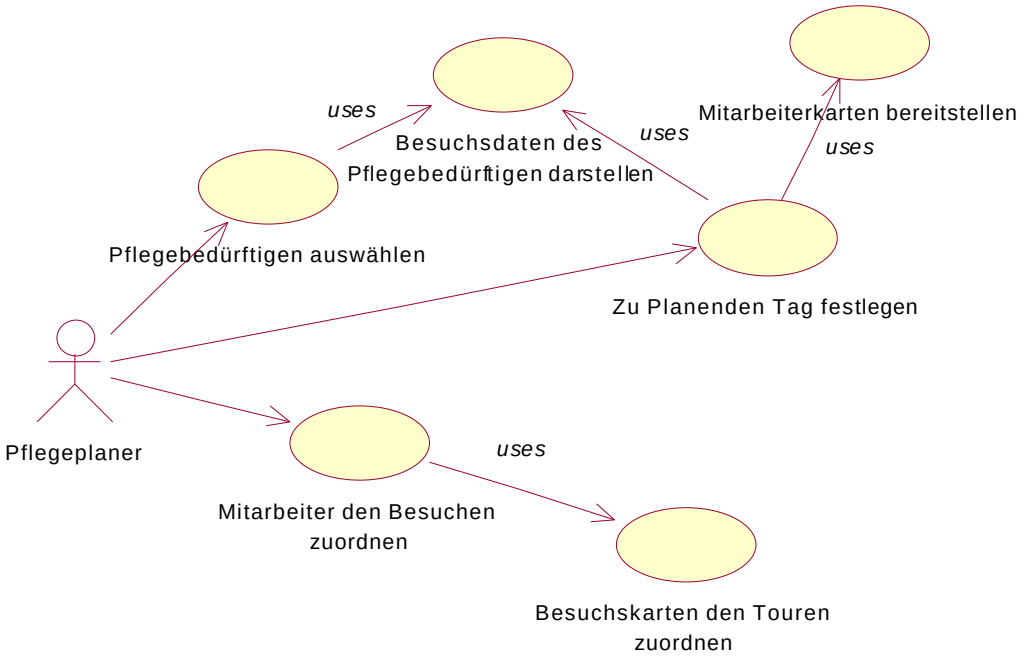


Abbildung 4 Use Case Diagramm Pflegeplanung durchführen (Pflegebedürftigensicht)

Natürlich ist es auch hier bei dem einen Akteur, dem Pflegeplaner geblieben. Wie bei vorhergehendem Diagramm werden hier die noch nicht weiter oben erklärten Use Cases genauer beschrieben.

Pflegebedürftigen auswählen

Anders als bei der Planung aus Tourensicht wird hier als eine der ersten Aktionen der zu planende Pflegebedürftige ausgewählt. Dieser steht dann im Mittelpunkt der Planungsaktivitäten

Zu planenden Tag festlegen

Dieser Punkt ist im letzten Use Case bereits beschrieben.

Mitarbeiterkarten bereitstellen

Dieser Punkt ist im letzten Use Case bereits beschrieben.

Besuchsdaten des Pflegebedürftigen darstellen

Die zu planenden Besuche des gewählten Pflegebedürftigen werden in diesem Use Case dargestellt. Dazu werden wie im Use Case „Besuchskarten erstellen“ beschrieben die Daten der einzelnen geplanten Besuche für den gewählten Tag aus der Datenbank gelesen. Je Besuch wird dann auf einer vertikalen Zeitleiste ein freier Bereich angezeigt, der den Daten des Besuchs entspricht. Auf diese freien Flächen können dann die Mitarbeiter geplant werden, die den Pflegebedürftigen versorgen.

Mitarbeiter den Besuchen zuordnen

Durch einfaches Ziehen eines Mitarbeiters auf einen Besuch, der wie oben beschrieben dargestellt wird, wird der Pflegebedürftige in die Tour des jeweiligen Mitarbeiters übernommen. Ist der Mitarbeiter noch keiner Tour zugewiesen, so muss diese Zuordnung jetzt erfolgen. Für die Zuordnung der Besuche zu den Touren werden die Methoden des oben beschriebenen Use Cases „Besuchskarten den Touren zuordnen“ verwendet.

Besuchskarten den Touren zuordnen

Dieser Punkt ist im letzten Use Case bereits beschrieben.

7.2. Entwurfsmuster

Nachdem die Aufgaben des Moduls durch die Use Cases erkannt wurden, ist es nun an der Zeit das System zu modellieren, welches diese Aufgaben erfüllen kann. Bei der Modellierung eines Systems treten sehr häufig ähnliche Probleme auf, die immer wieder neu zu lösen nicht sinnvoll wäre. Aus diesem Grunde wurden Entwurfsmuster entwickelt. Entwurfsmuster sind objektorientierte Vorschläge zum Design bestimmter Problemstellungen. Sie sind aus langjähriger Erfahrung von Programmierern entstanden, die diese immer wieder erfolgreich einsetzten. In Entwurfsmustern ist nicht nur die Lösung zu bestimmten Problemen beschrieben, sondern auch auf welche Implementierungsdetails man achten sollte, oder an welchen Stellen die Nachteile der einzelnen Muster liegen könnten. Es ist nicht so, dass ein bestimmtes Problem immer nur nach einem Muster gelöst werden kann. Hier muss man sich dann das Entwurfsmuster heraussuchen, welches am besten für diesen ganz bestimmten Fall geeignet ist.

Da auch wir bei diesem Projekt das Rad nicht neu erfinden wollen, ist es ratsam sich nach bereits vorhandenen Entwurfsmustern umzusehen. Hierzu müssen wir zunächst einmal die Designprobleme erkennen, die in unserem Fall auftreten.

Ein Problem, welches direkt offensichtlich ist, ist die Erzeugung der grafischen Benutzeroberfläche. Hier müssen beispielsweise die ‚Kärtchen’ der Patienten oder der Mitarbeiter, aber auch zum Beispiel eine Zeitleiste erzeugt werden, die dann auf dem Bildschirm zusammengesetzt ein einheitliches Bild ergeben müssen. Zu diesem Problemfall gibt es die Objekt-Erzeugungsmuster. In unserm Falle wollen wir eine Familie von Objekten erzeugen, die auch optisch gut zusammenpassen. Wenn man davon ausgeht, dass dieses Planungsmodul auch später einmal evtl. für eine Planung ganz anderer Art wiederverwendet werden soll, so sollten wir die Klassen, welche die benötigten Objekte erzeugen möglichst austauschbar machen. Dazu verwendet man im allgemeinen abstrakte Klassen. Im Programm wird dann nur mit diesen abstrakten Klassen anstatt mit den konkreten, abgeleiteten Klassen gearbeitet. Das gilt sowohl für die Klasse, welche die Objekte erzeugt, als auch für die erzeugten Objekte selber. Im Programm werden die erzeugten Objekte nur über ihre abstrakte Oberklasse verwendet. Dieses Vorgehen nennt man ‚programmieren hin auf eine Schnittstelle, nicht auf die Implementierung’ und bietet höchste Flexibilität und Wiederverwendbarkeit.

7.2.1. Abstrakte Fabrik

Das Entwurfsmuster ‚Abstrakte Fabrik’ ist das am besten geeignetste für diese Aufgabe. Zu dieser Einsicht gelangen wir, wenn wir uns den ‚Anwendbarkeits’-Block dieses Entwurfsmusters genauer ansehen:

„Verwenden Sie das Abstrakte-Fabrik-Muster wenn

- Ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden.“

Diesem Punkt können wir schon zustimmen, denn wenn dieses Planungsmodul für andere Planungsaufgaben verwendet werden sollte, dann wird sich auch sicherlich die Optik der einzelnen zu planenden Elemente ändern.

- „ein System mit einer von mehreren Produktfamilien konfiguriert werden soll“

Dieser Punkt ist sicherlich nicht so wichtig wie der erste, doch ist es auch denkbar, dass die Planung ein und derselben Daten je nach Vorliebe des Benutzers angepasst werden könnte.

- „eine Familie von verwandten Produktobjekten entworfen wurde, zusammen verwendet zu werden, und Sie diese Konsistenzbedingung sicherstellen müssen.“

Hier müssen wir wieder definitiv zustimmen, denn ‚um das Beispiel mit der Planung ganz anderer Elemente wieder aufzugreifen, es ist sicherlich nicht wünschenswert die Planung der Abläufe in einem Labor mit der Planung der Touren durcheinander zu werfen.

- „Sie eine Klassenbibliothek von Produkten anbieten möchten, von denen Sie nur die Schnittstellen, nicht aber ihre Implementierung offen legen möchten.“[13]

Es ist sicherlich auch denkbar, dass man mit diesen einmal entworfenen Elementen einer Planung eine ganz andere Sicht auf die Planung erzeugt. Insofern kann man diese einmal entworfene Fabrik sicherlich auch für weitere Projekte verwenden.

Im folgenden wollen wir uns die Struktur des Entwurfsmusters Abstrakte Fabrik einmal ansehen.

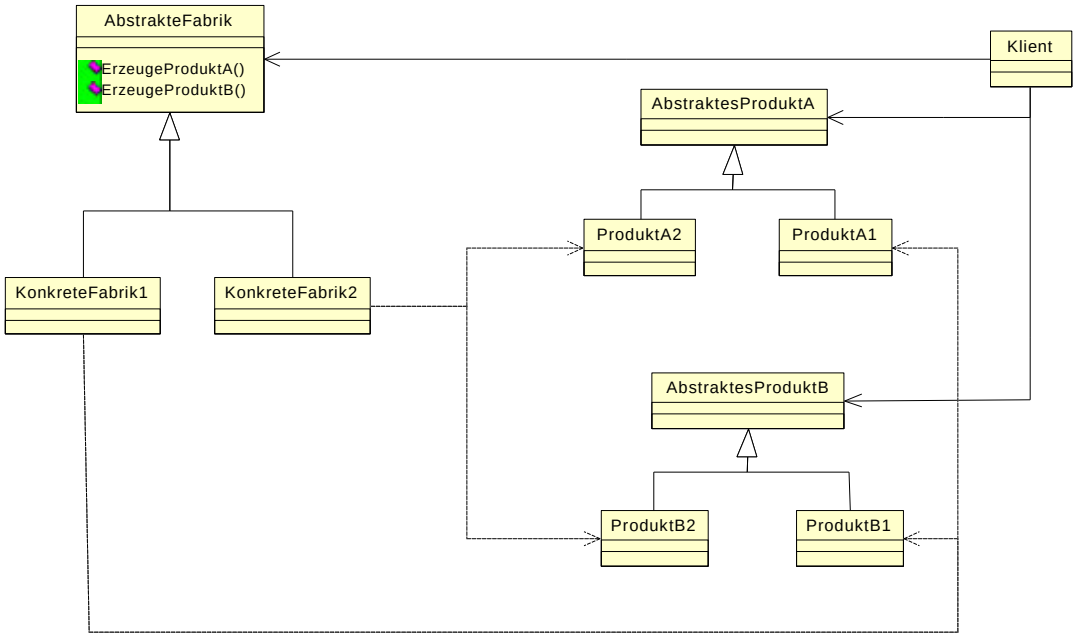


Abbildung 5 Entwurfsmuster Abstrakte Fabrik

Wie hier gut zu erkennen ist, arbeitet der Klient ausschließlich mit den abstrakten Produkten und der abstrakten Fabrik, was die Wiederverwendbarkeit deutlich fördert.

In unserem Falle, dem Fall, dass die Bedienungsoberfläche aus grafischen Elementen mit der abstrakten Fabrik erstellt werden soll, brauchen wir ein abstraktes Produkt nicht einmal gesondert zu definieren. Unser Produkt ist ein „JPanel“, welches beispielsweise direkt in einem „JFrame“ dargestellt werden kann. Die Unterscheidungen der konkreten Fabriken liegen hier im Inhalt des erstellten Panels. Hier können die konkreten Fabriken das Aussehen des Panels beliebig gestalten.

[13] Aus „Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software“ von Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Seite 109

Eine weitere Anpassung des Entwurfsmusters kommt mit den „Drag and Drop“-Funktionalitäten, die noch eingebaut werden sollen einher. Wenn beispielsweise ein Panel aufgebaut wird, welches alle ungeplanten Besuche enthält, dann ist es bei einem „Drag“ wichtig zu wissen, welches „Visit“-Objekt nun transferiert werden soll. Bei einem „Drag“ ist allerdings nur die Bildschirmkomponente bekannt, also z.B. das „JLabel“ über dem sich der Cursor beim Klick auf die Maustaste befand. Um von dieser Komponente zum betroffenen „Visit“-Objekt zu gelangen, befragen wir einfach die entsprechende „Konkrete Fabrik“, welche dieses Panel erzeugt hat. Sie ist dafür verantwortlich die Verbindung von Bildschirmkomponente und Datenobjekten herzustellen. Dies ist zwar eine kleine Abwandlung des Fabrikmusters, aber es ist der Lösung vorzuziehen der Fabrik genau vorzuschreiben wie ein Panel aufgebaut sein muss, damit der Klient die Datenobjektzuordnung selbst übernehmen kann.

7.2.2. Beobachter-Muster

Gerade bei grafischen Benutzeroberflächen muss der Zugriff auf Datenobjekte auch innerhalb eines Programms, welches nur auf einem Computer läuft organisiert und synchronisiert werden. Wenn ein und dasselbe Datenobjekt beispielsweise in zwei unterschiedlichen Fenstern angezeigt wird, muss natürlich das jeweils andere Fenster aktualisiert werden, wenn eine Änderung in einem Fenster ausgeführt wird. Wenn jedes Fenster ständig die Datenobjekte auf Änderung überprüft und gegebenenfalls die Anzeige aktualisiert wäre das Problem gelöst. Allerdings würde dies ständig den Prozessor belasten und eine nicht gerade elegante Methode darstellen. Viel besser ist es, wenn die Fenster benachrichtigt werden, wenn sich ein sie betreffendes Datenobjekt ändert. Genau bei diesen Anforderungen hat sich das Beobachter-Muster am meisten bewährt.

Dieses Konzept baut darauf auf, dass sich jedes Objekt, welches zu bestimmten Ereignissen benachrichtigt werden möchte bei dem Objekt „anmeldet“, welches dieses Ereignis erzeugt. Wenn sich also z.B. ein anderes Objekt dafür interessiert, dass sich ein ganz bestimmtes Datenobjekt ändert, dann meldet es sich bei genau diesem Datenobjekt an. Es kann sich allerdings nur dann anmelden, wenn es bestimmte Vorraussetzungen erfüllt. Diese Vorraussetzungen werden in „Listener-Interfaces“ (oder auch Beobachter-Interfaces) definiert welche das zu benachrichtigende Objekt implementieren muss. In diesen Interfaces stehen alle Methoden, die zur Benachrichtigung über bestimmte Ereignisse aufgerufen werden. Um detaillierte Informationen über das Ereignis zu übermitteln wird jeder Methode ein von `EventObjekt` Abgeleitetes Objekt mitgegeben. In erster Linie enthält dieses Objekt Informationen über das Quellobjekt des Ereignisses. Dies ist nötig, da sich jedes Objekt an beliebig vielen Objekten als „Listener“ anmelden kann. Außerdem werden in einem solchen `EventObject` weiterführende Informationen zu dem Ereignis gespeichert.

Mit einem solchen Konzept lassen sich also Änderungen in einem Datenobjekt von beliebig vielen anderen Objekten verfolgen. Diese Objekte können dann wie gewünscht auf eine Änderung im Datenobjekt reagieren. Ein weiteres Problem tritt dann auf, wenn wir eine Menge von Objekten überwachen wollen. Wenn sich Objekte nur bei den Datenobjekten selbst anmelden, dann erfahren sie nicht, wenn Datenobjekte aus der Menge gelöscht oder hinzugefügt werden. Hier ist es notwendig, dass sich das interessierende Objekt an der Menge, welche die Datenobjekte beinhaltet anmeldet. Denn nur die Menge „weiß“, wann Datenobjekte gelöscht oder hinzugefügt werden. Für das `EventObject` bedeutet dies, dass es zusätzlich Informationen darüber enthalten muss welches Datenobjekt betroffen ist, und welche Aktion auf ihm ausgeführt wurde. Diese Aktionen können entweder „Löschen“ oder

„Hinzufügen“ sein. Damit man Objekte nicht am Mengen- und Datenobjekt anmelden muss, kann man auch noch die Aktion „Ändern“ in das `EventObjekt` der Menge mit aufnehmen.

Im folgenden soll ein Beispielprogramm dieses Zusammenspiel zwischen Mengenobjekt, Datenobjekt und Listener verdeutlichen. In diesem Programm werden die systemspezifischen Daten (`Preferences`) über einen ambulanten Pflegedienst verwaltet. Um es möglichst einfach zu halten, werden hier nur die Pflegestufen (`CareClassifications`) verwaltet. Diese können angelegt, geändert und gelöscht werden. Die benötigten Klassen für dieses Programm werden in Abbildung 6 dargestellt.

Um das Programm zu starten muss die Klasse `PreferencesModuleStarter` aufgerufen werden. Sie ist hier auch die einzige mit einer `main`-Methode. Diese erzeugt nun ein neues `PreferencesModule`. Das `PreferencesModule` ist das Objekt, welches die Verknüpfung zwischen Datenzugriff und der grafischen Benutzeroberfläche herstellt. Folglich füllt es sich zunächst einen `ManagerVector` mit allen, in der zugrundeliegenden Datenbank vorhandenen `CareClassifications`. Der `ManagerVector` ist von Vector abgeleitet und beinhaltet zusätzlich einen `VectorChangeInformant`. Dieser `VectorChangeInformant` kann alle an den `ManagerVector` angemeldeten Objekte mit dem vom `ManagerVector` gewünschten `EvenObject` (hier `VectorChangeEvent`) benachrichtigen. Damit es nicht zu unübersichtlich wird, muss in diesem Beispiel wenn der `ManagerVector` geändert wird die Methode `VectorChanged()` mit dem geändertem Objekt und der Änderungsaktion aufgerufen werden.

`PreferencesModule` implementiert das Interface `VectorChangeListener`, um von einem `ManagerVector` über Änderungen informiert werden zu können. Damit es nun von dem `ManagerVector` welcher die `CareClassifiactions` beinhaltet benachrichtigt wird, meldet es sich bei diesem mit dem Methodenaufruf `addVectorChangeListener(this)` von `ManagerVector` an. Der Zugriff auf die Datenbank erfolgt über die Klasse `JDBC` mit den Methoden `getCareClassifications()`, `updateCareClassifications()` und `deleteCareClassification()`. Diese Klasse wiederum benutzt `JDBCEngine` zum direkten Zugriff auf die Datenbank.

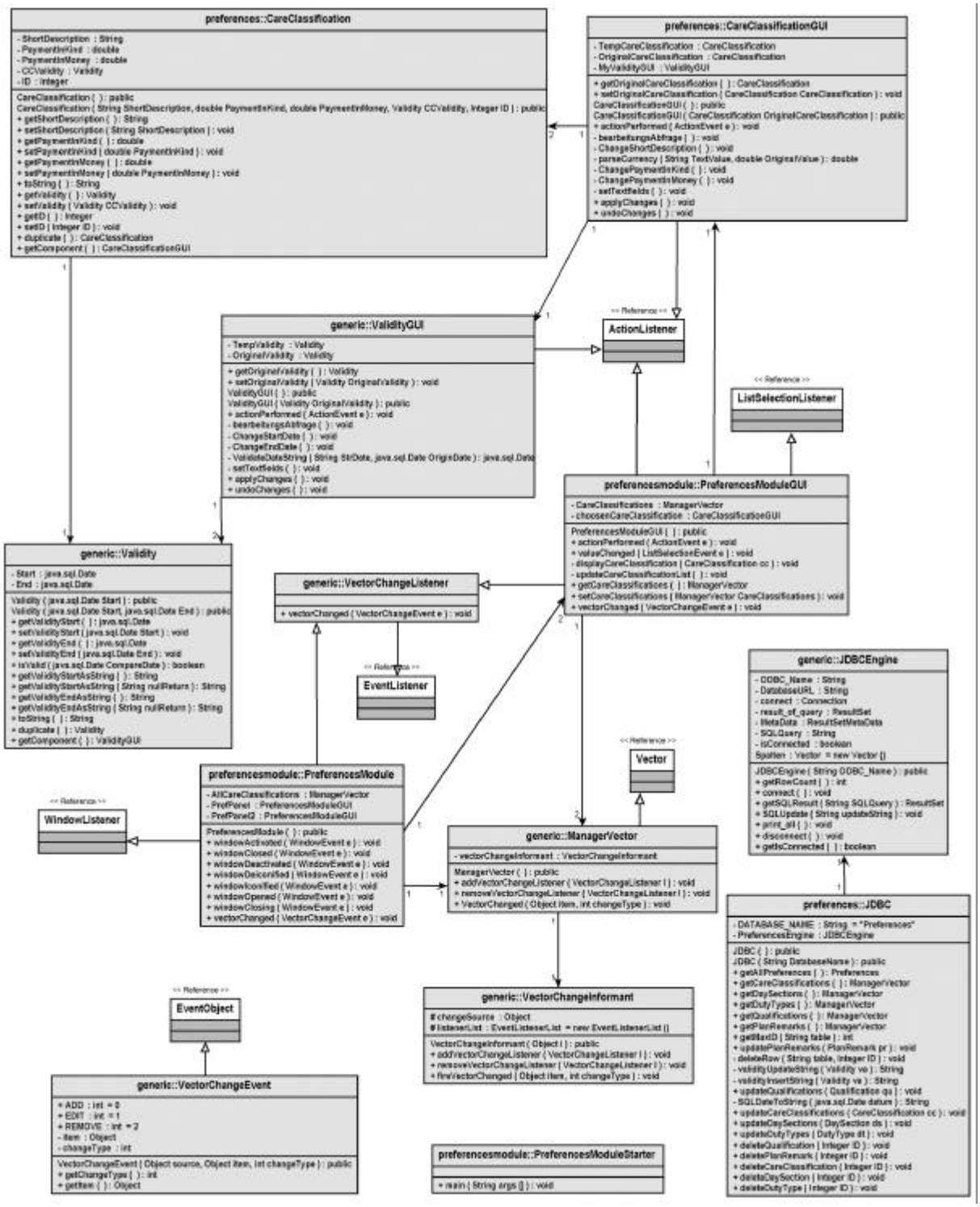


Abbildung 6: Klassendiagram PreferencesModule

`PreferencesModuleGUI` stellt die Oberfläche des Programms dar. Es ist ein `JPanel` und kann deshalb nicht direkt dargestellt werden. Es muss zunächst ein Fenster (`JFrame`) erzeugt werden, in welches es positioniert werden kann. Damit das Programm beim schließen des Fensters beendet wird, registriert sich `PreferencesModule` bei dem Fenster als `WindowListener`. Dieser stellt unter anderem die Methode `windowClosing()` zu Verfügung, mit welcher überwacht wird, wann ein überwachtes Fenster geschlossen wird. Nun werden die `CareClassifications` in Form des `ManageVectors` dem

`PreferencesModuleGUI` übergeben. Nachdem das Fenster angezeigt wird ist die Aufgabe des `PreferencesModule` erst einmal erledigt. Es wartet nun darauf, dass sich entweder der `ManagerVector` verändert, das heißt dass sich Pflegestufen ändern, oder dass das Fenster vom Benutzer geschlossen wird. Im ersten Fall wird die veränderte Pflegestufe direkt in die Datenbank geschrieben, bzw. gelöscht.

Sobald `PreferencesModuleGUI` die `CareClassifications` erhält, meldet es sich ebenfalls als `VectorChangeListener` bei diesem `ManagerVector` an. So erfährt `PreferencesModuleGUI` auch, wenn sich der Vector außerhalb von diesem `PreferencesModuleGUI` ändert und kann entsprechend darauf reagieren.

Im folgenden werden einige Aktionen auf Datenobjekte im Detail betrachtet um die Funktionsweise dieses Programms zu verdeutlichen.

Fall 1: Benutzer fügt eine Pflegestufe hinzu

Wenn der Benutzer den Button „Hinzufügen“ betätigt, werden alle ActionListener dieses Buttons mit der Methode `actionPerformed()` benachrichtigt. `PreferencesModuleGUI` hat sich bei diesem Button, wie auch bei allen anderen Buttons auf diesem Panel als Listener angemeldet. In der Methode `actionPerformed()` wird jetzt erst einmal mit der Methode `getSource()` von ActionEvent geprüft, welcher Button dieses Ereignis ausgelöst hat. Nachdem feststeht, dass es der Button „Hinzufügen“ war, wird eine neue `CareClassification` erzeugt, die Listenauswahl auf die neue `CareClassification` gestellt und diese in dem AnzeigePanel dargestellt. Jetzt wird der `ManagerVector` mit der Methode `VectorChanged()` benachrichtigt, dass er eine neue `CareClassification` enthält. Dieser wiederum benachrichtigt alle angemeldeten `VectorChangeListener` von diesem Ereignis. Also wird `PreferencesModule` und `PreferencesModuleGUI` selbst benachrichtigt. In der Methode `vectorChanged()` dieser Objekte wird nun das `VectorChangeEvent` auf den Typ der Aktion hin überprüft. Wenn es feststeht, dass es sich um ein neues Element in der Menge handelt, speichert `PreferencesModule` dieses einfach mit der Methode `updateCareClassifications()` in der Datenbank. `PreferencesModuleGUI` selbst muss hier nur die Liste, welche alle `CareClassifications` beinhalten neu darstellen.

Fall 2: Benutzer ändert eine Pflegestufe

Wenn der Benutzer den Button „Änderung speichern“ nach einer getätigten Bearbeitung einer Pflegestufe betätigt, werden alle ActionListener dieses Buttons mit der Methode `actionPerformed()` benachrichtigt. `PreferencesModuleGUI` hat sich auch bei diesem als Listener angemeldet. In der Methode `actionPerformed()` wird jetzt erneut mit der Methode `getSource()` geprüft, welcher Button dieses Ereignis ausgelöst hat. Nachdem feststeht, dass es der Button „Änderung speichern“ war, wird die Bearbeitungskopie im Anzeigepanel (CareClassificationGUI) auf das Orginaldatenobjekt mit der Methode applyChanges() übertragen. Somit ist das Object im ManagerVector verändert. Damit der ManagerVector dies erfährt wird er mit der Methode `VectorChanged()` benachrichtigt, dass er eine geänderte `CareClassification` enthält. Dieser wiederum benachrichtigt alle angemeldeten `VectorChangeListener` von diesem Ereignis. Also wird `PreferencesModule` und `PreferencesModuleGUI` selbst benachrichtigt. In der

Methode `vectorChanged()` dieser Objekte wird wiederum das `VectorChangeEvent` auf den Typ der Aktion hin überprüft. Wenn es feststeht, dass es sich um ein geändertes Element in der Menge handelt, speichert `PreferencesModule` dieses mit der Methode `updateCareClassifications()` in der Datenbank. Dabei wird das vorhandene Objekt überschrieben. `PreferencesModuleGUI` selbst muss neben der Aktualisierung der Liste aller `CareClassification` nur auf dieses Ereignis reagieren, wenn sich die geänderte `CareClassification` im Anzeigepanel befindet. Wenn dies der Fall ist, muss auch das Anzeigepanel aktualisiert werden.

Die Darstellung in einem Sequenzdiagram veranschaulicht dieses Beispiel.

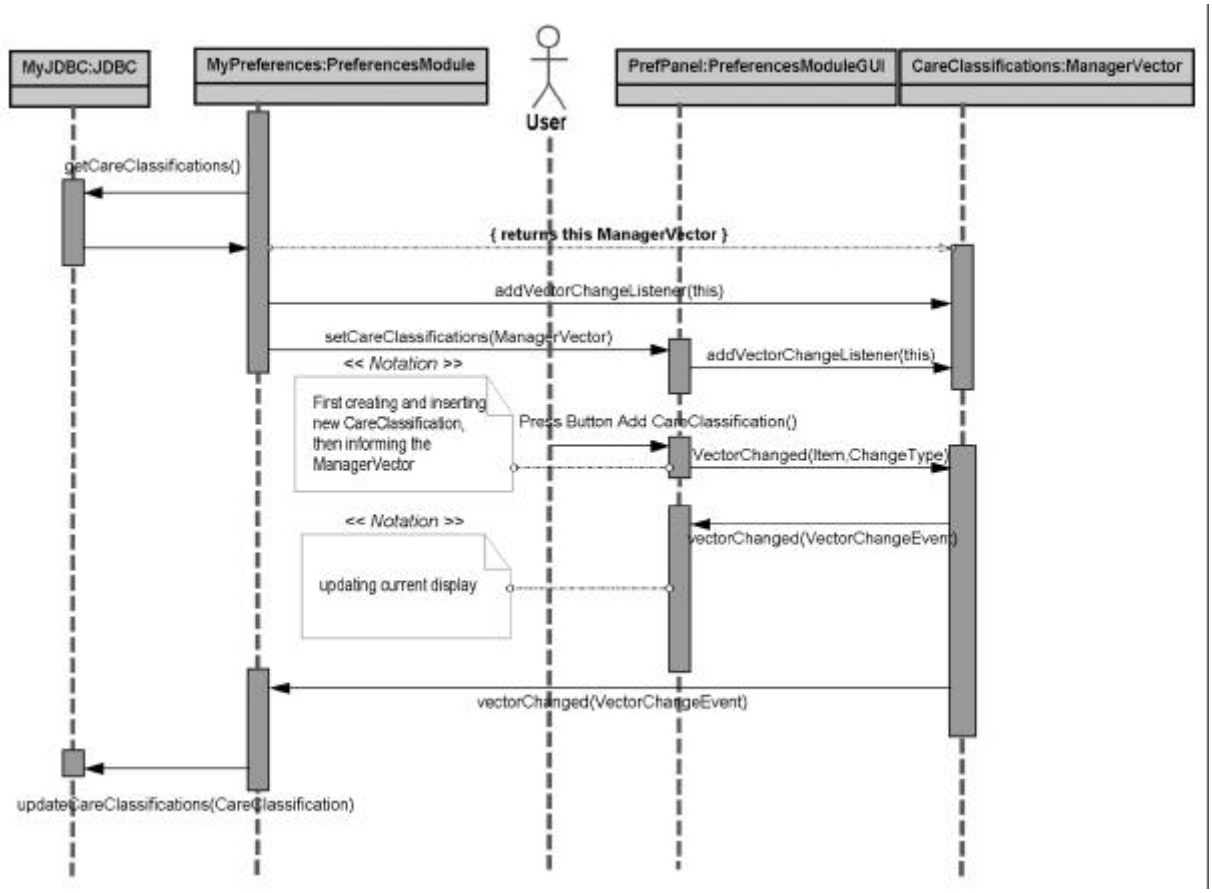


Abbildung 7: Sequenzdiagram Pflegestufe anlegen

7.3. Klassendiagramme

7.3.1. Model-View-Controller

Im folgenden Klassendiagramm ist die Umsetzung des MVC-Paradigams in dieser Arbeit zu sehen.

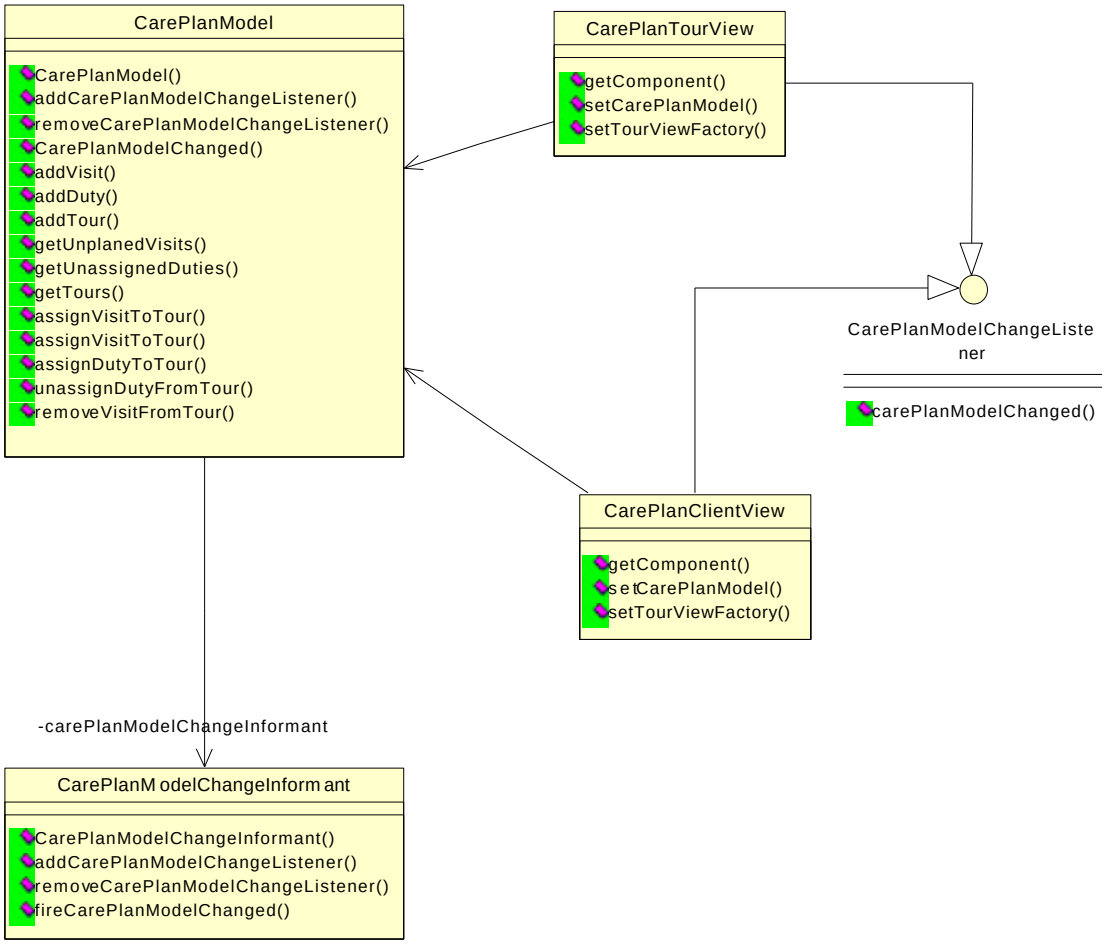


Abbildung 8 Model-View-Controller Implementierung als Klassendiagramm

Das CarePlanModel ist sozusagen der Mittelpunkt dieses Systems. Es speichert alle zur Planung der Touren nötigen Informationen. Nur über die Methoden dieses Datenmodels kann man die Daten abfragen oder auch verändern. An diesem Model sind verschiedene Views als Beobachter angemeldet, so dass das Model diese bei Bedarf über Änderungen informieren kann. Um diese Änderungsinformation weiterzugeben, beinhaltet das CarePlanModel einen CarePlanModelChangeInformant. Dieser wurde zur Übersichtlichkeit eingeführt. Wenn eine Änderungsinformation weitergegeben werden soll, so übergibt das CarePlanModel diese Aufgabe an diesen ChangeInformant. Eine Änderungsinformation wird in unserem Fall jedoch nicht vom Model selbst, sondern von der ändernden View ausgeführt. Dies hat einen einfachen Grund. Wenn eine View beispielsweise mehrere Änderungen unmittelbar hintereinander durchführt, müssen nicht jedes Mal alle anderen Views hiervon informiert

werden. Erst wenn alle Änderungen abgeschlossen sind, sollte dies geschehen. Da das Model aber nicht wissen kann, ob eine Änderung ein Bestandteil einer Reihe von Änderungen ist sollte diese Aufgabe das ändernde Objekt, hier die jeweilige View, übernehmen. Darum existiert die Methode `CarePlanModelChanged` im CarePlanModel.

Eine View meldet sich über die Methode `addCarePlanModelChangeListener` an dem CarePlanModel an. Von hier an wird es von jeder Änderung informiert bis es sich mit der Methode `removeCarePlanModelChangeListener` wieder abmeldet.

Bei einer Änderung wird die View über das Interface `CarePlanModelChangeListener` informiert. Dieses Interface enthält die Methode `carePlanModelChanged` die zur Benachrichtigung aufgerufen wird. Das übergebene Argument `CarePlanModelChangeEvent e` enthält Information über das Model. In diesem Prototypen der Software ist allerdings nur die getSource()-Methode dieses Argumentes von Bedeutung, über die man an das geänderte CarePlanModel gelangt. In einer späteren Version dieser Software könnte man hier weitere Details liefern, die z.B. genau die Teile des CarePlanModel enthalten, die sich auch verändert haben.

An obigem Klassendiagramm fällt auf, dass anscheinend noch ein Controller fehlt. Der Controller ist im MVC-Modell dafür zuständig auf Benutzereingaben zu reagieren. Hierfür sind in unserem Beispiel die Klassen zuständig, welche die Drag and Drop Operationen durchführen. Diese würden hier nur das Diagramm überladen, und werden somit später noch einmal erläutert, wenn es um diese Funktionalität geht.

7.3.2. Datenbankanbindung

Die Datenbankanbindung erfolgt über das Beobachtermuster. Dabei meldet sich das Objekt, welches zur Anbindung an die Datenbank verantwortlich ist an dem CarePlanModel wie eine View an. Das hat den Vorteil, dass natürlich auch dieses Objekt von jeder Änderung informiert wird, und somit die Änderungen direkt in die Datenbank schreiben kann.

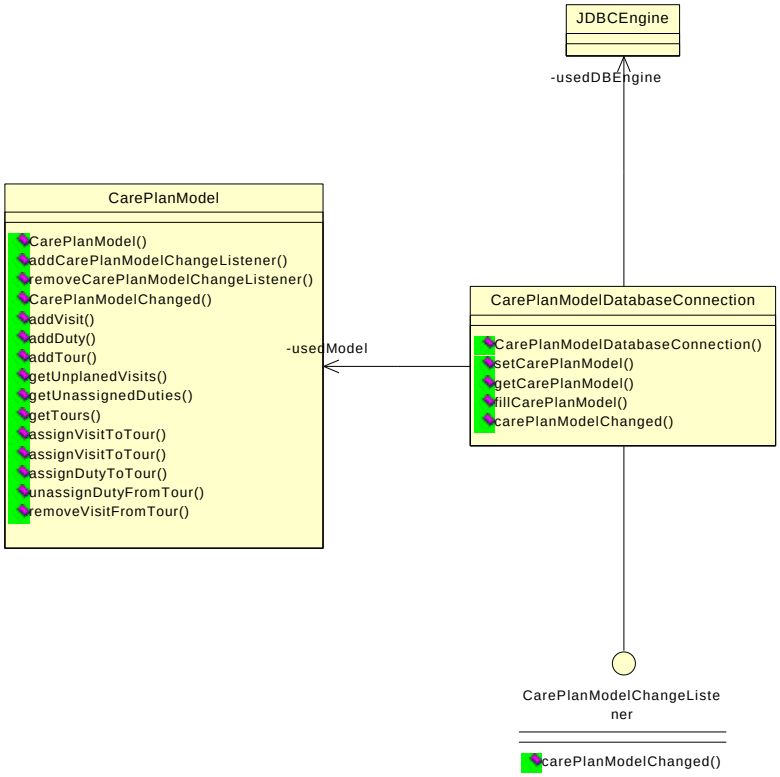


Abbildung 9 Klassenansicht der Datenbankanbindung

In Abbildung 9 kann man sehen, wie diese Datenbankanbindung im Klassendiagramm aussieht. Die Klasse CarePlanModelDatabaseConnection hat sich als Beobachter beim CarePlanModel angemeldet. Wenn nun eine Änderungsinformation vom CarePlanModel ausgeht, so wird dies auch die Datenbankanbindungsklasse erfahren. Daraufhin wird innerhalb von carePlanModelChanged die Änderung in der Datenbank gespeichert. Für den Zugriff auf die Datenbank wird die Klasse JDBCEngine verwendet, die Methoden zu Verfügung stellt direkt auf die Datenbank zuzugreifen. Zum Start des Programms wird das CarePlanModel auch über die Klasse CarePlanModelDatabaseConnection gefüllt. Dazu wird die Methode fillCarePlanModel verwendet. Ihr wird als Argument das Datum angegeben, mit welchem das Model gefüllt werden soll. Hier halten wir also fest, dass das Model immer nur Daten über einen Tag enthält.

Ein weiterer Vorteil dieser Datenbankanbindungstechnik ist, dass das Objekt selbst entscheiden kann, wann es Änderungen auf die Datenbank schreibt; z.B. wäre es möglich erst auf ausdrücklichen Wunsch des Benutzers die Änderungen auf Datenbankebene zu vollziehen. So wäre es möglich eine Undo-Funktion zu implementieren. Das Objekt müsste sich lediglich alle Änderungen in einer geeigneten Datenstruktur merken. Ob diese Funktionalität gewünscht ist, bleibt noch zu klären, aber die Möglichkeit würde bestehen. In diesem Prototyp werden wir jedoch keine Undo-Funktion programmieren.

7.3.3. Auswahl des zu planenden Datums

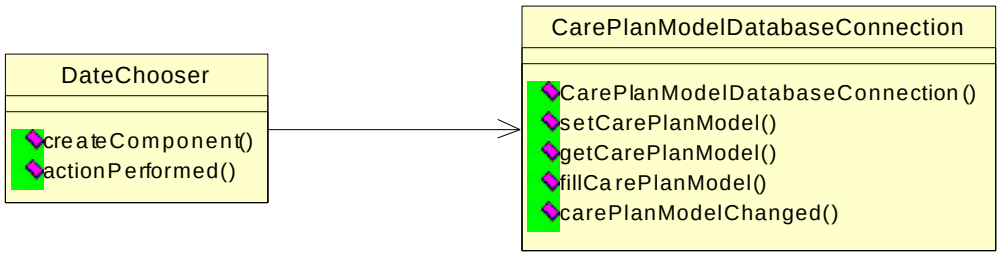


Abbildung 10 Klassenansicht der Datumsauswahl

Mit der Klasse DateChooser wird eine Bildschirmkomponente erzeugt, die es dem Benutzer ermöglicht, das zu planende Datum festzulegen. Erzeugt wird die Komponente durch die createComponent-Methode. In unserem Prototyp besteht diese Komponente aus einem einfachen Textfeld, in dem man das Datum eingeben kann, und einem Button. Erst wenn man den Button betätigt wird über die Methode actionPerformed des implementierten ActionListeners das CarePlanModel mit den Daten aus dem gewünschten Datum neu gefüllt. Dazu bedient sich diese Klasse der CarePlanModelDatabaseConnection. Diese wird dem DateChooser per Konstruktor übergeben.

7.3.4. Abstrakte Fabrik

Nun wollen wir uns einmal die konkrete Implementierung des Abstrakte Fabrik Entwurfsmusters in unser Projekt ansehen. Nehmen wir hierzu einmal den Teil des Programms, der die Planung aus Sicht der Touren ermöglicht. Das Klassendiagramm in UML-Notation ist in Abbildung 11 dargestellt.

Wie man in Abbildung 11 erkennen kann besitzt die Klasse CarePlanTourView, hier sozusagen die Hauptklasse, ein CarePlanModel und eine TourViewFactory. Diese beiden Objekt werden der TourView bei der Instanzierung oder kurz danach zugewiesen. Über das CarePlanModel kann die TourView die benötigten Daten auslesen und ggf. verändern und mit Hilfe der eines Objektes vom Typ TourViewFactory wird die Oberfläche erstellt, über welche die Planung aus Tourensicht vollzogen werden kann. Für diesen Prototypen der Pflegeeinsatzplanung ist eine konkrete TourViewFactory implementiert worden: SimpleTourViewFactory. Sie ist von der abstrakten Klasse TourViewFactory abgeleitet, so dass sie wie eine solche verwendet werden kann.

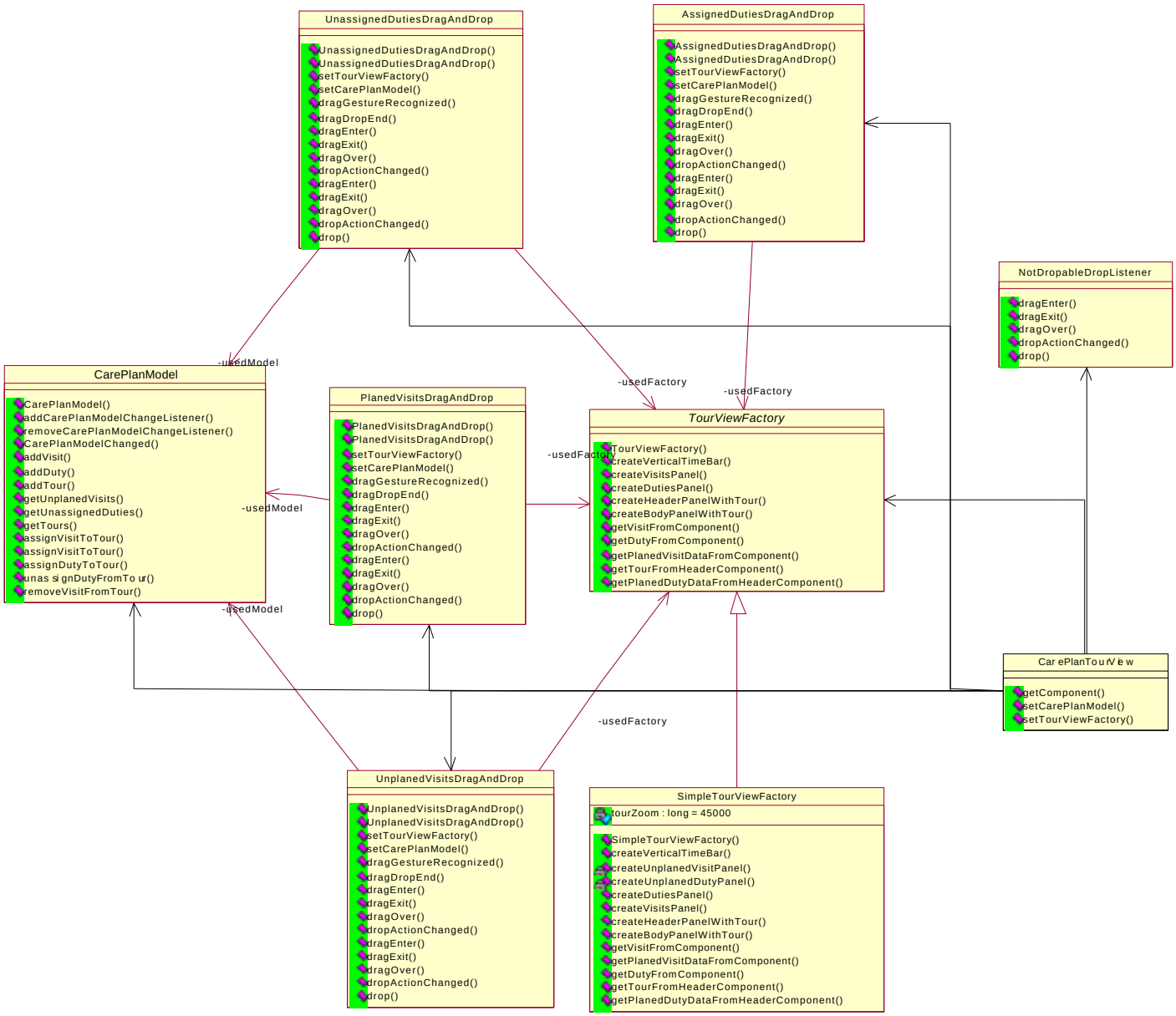


Abbildung 11 Klassendiagramm der Umsetzung des Entwurfsmusters "Abstrakte Fabrik"

Nicht in diesem Klassendiagramm verzeichnet sind die Produkte der TourViewFactory. Diese sind jedoch ausschließlich vom Typ JPanel aus der Klassenbibliothek von Swing. Daher ist es hier nicht unbedingt notwendig, das Klassendiagramm mit Einfügen dieser Klasse und zugehörigen Instanzierungslinien ausgehend von TourViewFactory und Simple TourViewFactory zu überfrachten und unleserlicher zu machen. Hier sei nur gesagt, dass das Layout der TourView somit aus JPanels zusammengesetzt wird.

7.3.5. Datenobjekt-Zuordnungstechnik der SimpleTourViewFactory

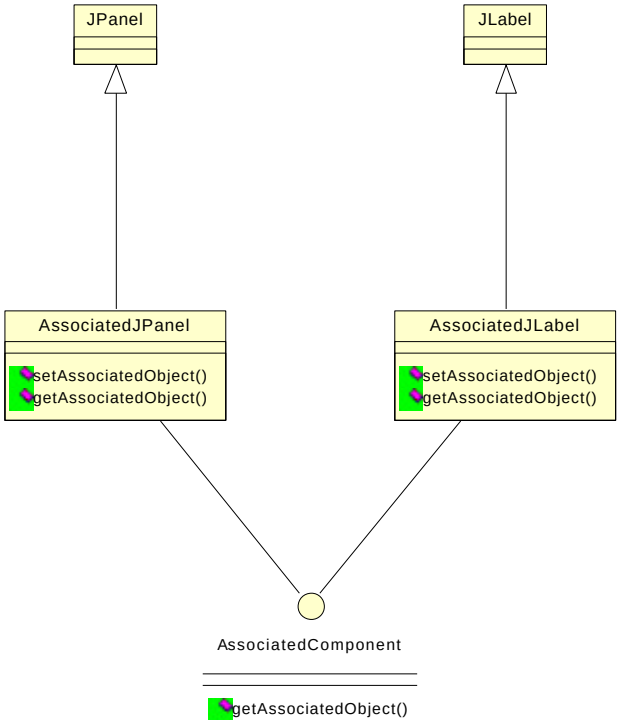


Abbildung 12 Datenobjekt-Zuordnung zu einer Komponente im Klassendiagramm

Wie schon erwähnt, ist eine TourViewFactory selbst dafür verantwortlich, die Zuordnung zwischen Bildschirmkomponente und zugehörigem Datenobjekt herzustellen. Hier gibt es sicherlich einige Möglichkeiten, dies zu realisieren. Beispielsweise könnte man diese Zuordnung in einer Tabelle bei der Erzeugung der einzelnen Komponenten festhalten. Wir werden hier jedoch eine andere Methode anwenden. Wir erzeugen einfach Komponenten, die selbst eine Referenz auf ihr Datenobjekt speichern können. Diese Vorgehensweise macht es nötig, dass wir erst einmal Klassen für diese neuen Komponenten erzeugen. Im obigen Diagramm (Abbildung 12) sehen wir alle benötigten Klassen.

Über die Methoden get- und setAssociatedObject ist ein direkter Zugriff auf das zugehörige Datenobjekt möglich. Ansonsten verhalten sich diese Komponenten wie ganz normale Swing Komponenten, da sie von diesen abgeleitet sind. Wollte man diese TourViewFactory erweitern und die Ansicht aus komplexeren Gebilden zusammensetzen, so könnte man auf oben im Klassendiagramm beschriebene Art auch noch andere AssociatedComponents erzeugen. In der späteren Implementierung werden wir der Einfachheit halber das assoziierte Objekt im Konstruktor mit übergeben. Das spart uns eine Befehlszeile.

7.3.6. Drag and Drop Funktionalität

Desweiteren sind auf dem Klassendiagramm in Abbildung 11 noch einige andere Klassen zu sehen. Sie haben alle mit der „Drag and Drop“-Funktionalität zu tun. Sie implementieren die hierfür notwendigen Interfaces wie DragGestureListener, DragSourceListener und DropTargetListener. Es gibt so viele Klassen, da der Übersichtlichkeit wegen die Aktionen

für verschiedene Komponenten auf verschiedene Klassen aufgeteilt wurden. Hier eine Auflistung dieser Klassen mit den einzelnen Aufgaben:

Klasse mit den implementierten Interfaces	Funktion
`NotDropableDropListener implements DropTargetListener`	Verantwortlich für alle Komponenten, die wegen Autoscroll-Funktion als dropTarget definiert werden mussten, jedoch keine Objekte per „Drag and Drop“ annehmen. Sie lehnt jeden Dropversuch ab.
`AssignedDutiesDragAndDrop implements DragGestureListener, DragSourceListener`	Diese Klasse bearbeitet alle Drag-Methoden die auf geplanten Schichten ausgeführt werden. Geplante Schichten befinden sich im Kopf der einzelnen Touren. Falls keine Schicht im Kopf der gewählten Tour vorhanden ist, wird der „Drag“ nicht gestartet.
`PlanedVisitsDragAndDrop implements DragGestureListener, DragSourceListener`	Hier werden die Methoden implementiert, die für die „Drag“-Operation bereits geplanter Besuche verantwortlich sind. Der Drag wird gestartet, wenn das Objekt unter dem Mauszeiger bei einem Dragversuch ein geplanter Besuch ist.
`UnassignedDutiesDragAndDrop implements DragGestureListener, DropTargetListener`	Diese Klasse ist für zwei Aufgabenbereiche zuständig. Einmal verwaltet sie die Dragversuche auf den ungeplanten Schichten. Der Drag wird gestartet, wenn ein Dragversuch auf einer ungeplanten Schicht ausgeführt wird. Zum anderen wird das Dropverhalten auf den Tour-Köpfen implementiert. Diese nehmen geplante und ungeplante Schichten an, die unterschiedlich verarbeitet werden. Bei geplanten Schichten wird die betroffene Schicht aus der bisherigen Tour gelöscht und der neuen hinzugefügt. Sollte der Ziel-Tour bereits eine Schicht zugewiesen sein, so wird diese Schicht den ungeplanten Schichten zugeführt.
`UnplanedVisitsDragAndDrop implements DragGestureListener, DropTargetListener`	Auch hier werden wieder zwei Aufgaben abgedeckt. Die erste Aufgabe ist die Verwaltung der Dragversuche auf ungeplanten Besuchen. Ein Drag wird gestartet wenn ein Dragversuch auf einem ungeplanten Besuch ausgeführt wird. Der zweite Teil kümmert sich um das Dropverhalten innerhalb einer Tour. Bei einem „Drop“ von ungeplanten Besuchen auf einer freien Zeit wird dieser Besuch möglichst nah an seiner bevorzugten Planzeit eingesetzt ohne jedoch andere Besuche zu verschieben. Bei einem Drop auf einen Bereits geplanten Besuch wird dieser geplante Besuch nach hinten geschoben und der neue an die frei gewordene Stelle eingesetzt. Wenn das Drop-Objekt ein geplanter Besuch ist, so wird nach dem Drop der Besuch aus seiner alten Tour gelöscht.

Abbildung 13 Beschreibung der für "Drag and Drop"-Vorgänge relevanten Klassen

Die Methoden der TourViewFactory, die sich nicht auf das Erzeugen von Bildschirmkomponenten für die TourView beziehen, werden fast ausschließlich von den in obiger Tabelle beschriebenen Klassen verwendet. Hier ist es nämlich wichtig zu wissen, welche Komponente welches Objekt darstellt. Dies kommt daher, da man bei einer „Drag and Drop“-Operation nur die Komponenten erfragen kann, auf der die Operation ausgeführt wurde, nicht aber die Datenobjekte aus denen diese erstellt wurden. Um diese Verbindung herzustellen fragen die obigen Klassen bei der entsprechenden TourViewFabrik nach, welches Objekt mit welcher Komponente verbunden ist. Zu diesem Zweck werden die DragAndDrop Klassen auch von CarePlanTourView mit der benutzten TourViewFactory initialisiert.

Da diese Klassen auch Änderungen an den Datenobjekten vornehmen können, muss an sie auch das benutzte CarePlanModel übergeben werden. Nach einer „Drag and Drop“-Operation wird über dieses Model dann die Daten aktualisiert und die Nachricht an das Model weitergegeben, dass soeben Daten geändert wurden. Um den Rest kümmert sich dann das Model, indem es alle angemeldeten Objekte über die Änderung informiert. Dies wird später bei der Implementierung des Beobachter-Musters genauer beschrieben.

Hier das Klassendiagramm, welches alle für „Drag and Drop“ relevanten Daten beinhaltet. Hier sind die Klassen aller Daten enthalten, die mit der Maus per Drag and Drop innerhalb der Touren- und Patienten-Ansicht geplant werden können.

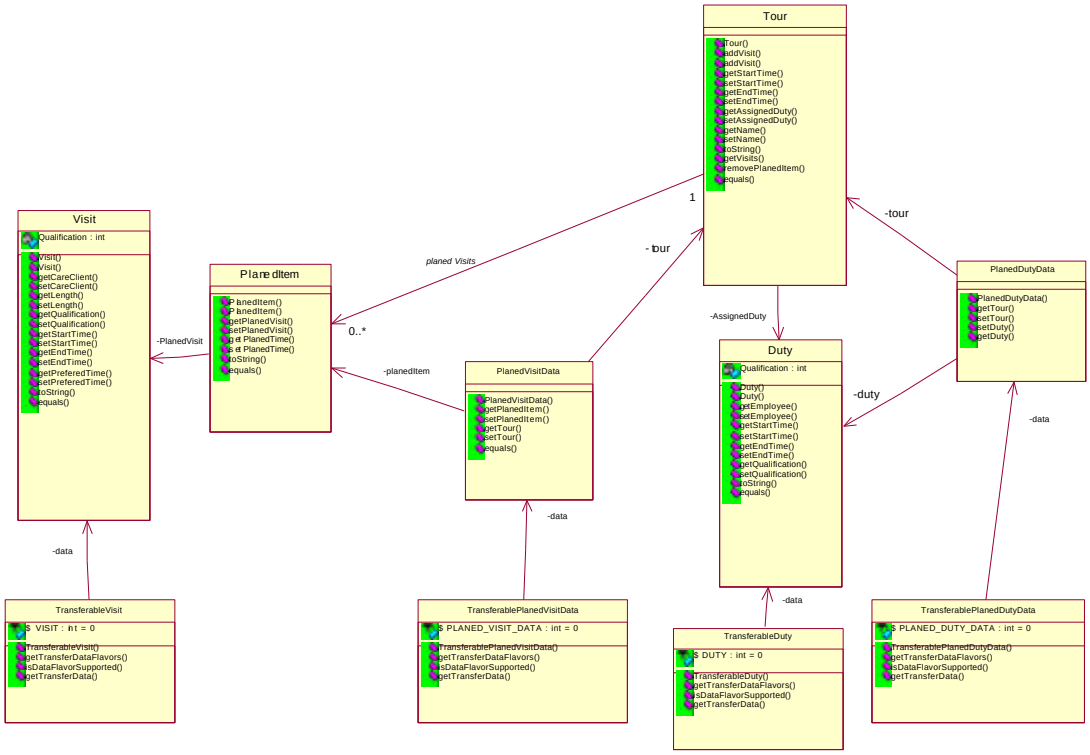


Abbildung 14 Klassendiagramm der für "Drag and Drop" relevanten Datenobjekte

Wie hier gut zu erkennen ist, muss von allen Daten, die mit der Maus „bewegt“ werden sollen eine sogenannte Transferable-Klasse erzeugt werden. Die Schnittstelle dieser Klassen ist im Interface Transferable definiert. Dieses Interface muss von den Klassen implementiert werden. Bei einem Drag and Drop Vorgang wird nun zu Anfang von dem Objekt, welches mit der Maus geplant werden soll ein Transferable-Objekt der entsprechenden Klasse erzeugt. Bei

einem Drop-Ereignis, welches normalerweise einen Drag and Drop Vorgang abschließt, kann über dieses Objekt die Infos zum Ausgangs-Objekt abgerufen werden.

In Java ist es wichtig, dass das zu transferierende Objekt das Interface „Serializable“ implementiert, da es sonst nicht mit dem Transferable-Objekt übertragen werden kann. Wenn diese Schnittstelle fehlt, wird die Drop-Methode zum Abschluss dieses Vorganges nicht aufgerufen.

Um die Arbeitsweise eines Transferable-Objekts zu verstehen sehen wir uns erst einmal das Interface Transferable an:

```
/*
 * @(#)Transferable.java 1.6 98/09/21
 *
 * Copyright 1996-1998 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

package java.awt.datatransfer;

import java.io.IOException;

/**
 * Defines the interface for classes that can be used to provide data
 * for a transfer operation.
 *
 * @version  1.6, 09/21/98
 * @author Amy Fowler
 */

public interface Transferable {

    /**
     * Returns an array of DataFlavor objects indicating the flavors the data
     * can be provided in.  The array should be ordered according to preference
     * for providing the data (from most richly descriptive to least descriptive).
     * @return an array of data flavors in which this data can be transferred
     */
    public DataFlavor[] getTransferDataFlavors();

    /**
     * Returns whether or not the specified data flavor is supported for
     * this object.
     * @param flavor the requested flavor for the data
     * @return boolean indicating wjether or not the data flavor is supported
     */
    public boolean isDataFlavorSupported(DataFlavor flavor);

    /**
     * Returns an object which represents the data to be transferred.  The class
     * of the object returned is defined by the representation class of the flavor.
     *
     * @param flavor the requested flavor for the data
     * @see DataFlavor#getRepresentationClass
     * @exception IOException                if the data is no longer available
     *              in the requested flavor.
     * @exception UnsupportedFlavorException if the requested data flavor is
     *              not supported.
     */
```

```
    public Object getTransferData(DataFlavor flavor) throws
UnsupportedFlavorException, IOException;

}
```

Die Methode, um die Daten aus einem Transferable-Objekt auszulesen, nennt sich hier also getTransferData(DataFlavor flavor). Schon hier kann man eine Besonderheit erkennen. Es wird nicht einfach das Datenobjekt zurückgegeben, sondern man muss einen „flavor“ festlegen, in dem man das Daten-Objekt gerne hätte. Wenn man das Objekt z.B. als String haben möchte, da man das Orginaldatenformat nicht verarbeiten kann, dann muss man dies mit dem „flavor“-Argument übergeben. Die unterstützten Datenformate kann man sich mit getTransferDataFlavors() vom Transferable zurückgeben lassen. Man erhält ein Array von DataFlavor, das man dann nach einem verarbeitbarem Datenformat durchsuchen kann. Eine andere Möglichkeit ist, wenn man nur ein ganz bestimmtes Format lesen kann, dann kann man fragen ob dies geliefert werden kann mit der Methode isDataFlavorSupported(DataFlavor flavor). Wie so eine Transferable-Klasse implementiert aussieht können wir uns im folgendem Quelltext einmal ansehen.

```
import java.util.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;

public class TransferableVisit implements Transferable{
  final static int VISIT = 0;

  final public static DataFlavor VISIT_FLAVOR =
  new DataFlavor(Visit.class, "Visit");

  private Visit data;

  public TransferableVisit(Visit data) {
  this.data = data;
  }

  static DataFlavor flavors[] = {VISIT_FLAVOR};

  /**
   * Returns an array of DataFlavor objects indicating the flavors the data
   * can be provided in.  The array should be ordered according to preference
   * for providing the data (from most richly descriptive to least descriptive).
   * @return an array of data flavors in which this data can be transferred
   */
  public DataFlavor[] getTransferDataFlavors(){
    return flavors;
  }

  /**
   * Returns whether or not the specified data flavor is supported for
   * this object.
   * @param flavor the requested flavor for the data
   * @return boolean indicating wjether or not the data flavor is supported
   */
  public boolean isDataFlavorSupported(DataFlavor flavor){
    return flavor.equals(flavors[VISIT]);
  }

  /**
   * Returns an object which represents the data to be transferred.  The class
   * of the object returned is defined by the representation class of the flavor.
   *
   * @param flavor the requested flavor for the data
   * @see DataFlavor#getRepresentationClass
   * @exception IOException                if the data is no longer available
```

```
     *              in the requested flavor.
     * @exception UnsupportedFlavorException if the requested data flavor is
     *              not supported.
     */
    public Object getTransferData(DataFlavor flavor)
           throws UnsupportedFlavorException, IOException{
      if (flavor.equals(flavors[VISIT])) {
      return data;
    } else {
      UnsupportedFlavorException t=new UnsupportedFlavorException(flavor);
      throw t;
    }
    }
}
```

Ein Objekt dieses Typs kann man nur erstellen, wenn man dem Konstruktor ein Visit-Objekt mit übergibt. Damit wird dem Transferable das Datenobjekt zugewiesen.

Mit den Zeilen

```
  final public static DataFlavor VISIT_FLAVOR =
  new DataFlavor(Visit.class, "Visit");

  …

  static DataFlavor flavors[] = {VISIT_FLAVOR};
```

legen wir fest, dass dieses Transferable nur Daten im Format eines Visit-Objektes liefern kann. Dies kann auch über die oben beschriebenen Methoden abgefragt werden.

Wird nun bei getTransferData(DataFlavor flavor) ein anderes Format angefordert, so wird eine Exception geworfen und keine Daten zurückgegeben. Daher werden wir dies im Programm immer vorher abfragen.

7.3.7. Autoscroll-Support

Bei „drag and drop“-Operationen ist es oft wünschenswert, wenn ein Objekt, welches sich nicht im ViewPort eines Scrollpanes befindet während einer Drag-Operation erreichen kann. Java bietet hierfür auch eine Lösung an. Dazu müssen jedoch mehrere Vorraussetzungen erfüllt sein:

- alle Objekte, über die man „Scrollen“ möchte, müssen DropTargets sein,
- für alle Objekte muss die Methode setAutoscrolls(true) aufgerufen werden und
- alle Objekte müssen das Interface Autoscroll implementieren.

In folgendem Diagramm sehen wir alle Klassen, die das Interface Autoscroll implementieren:

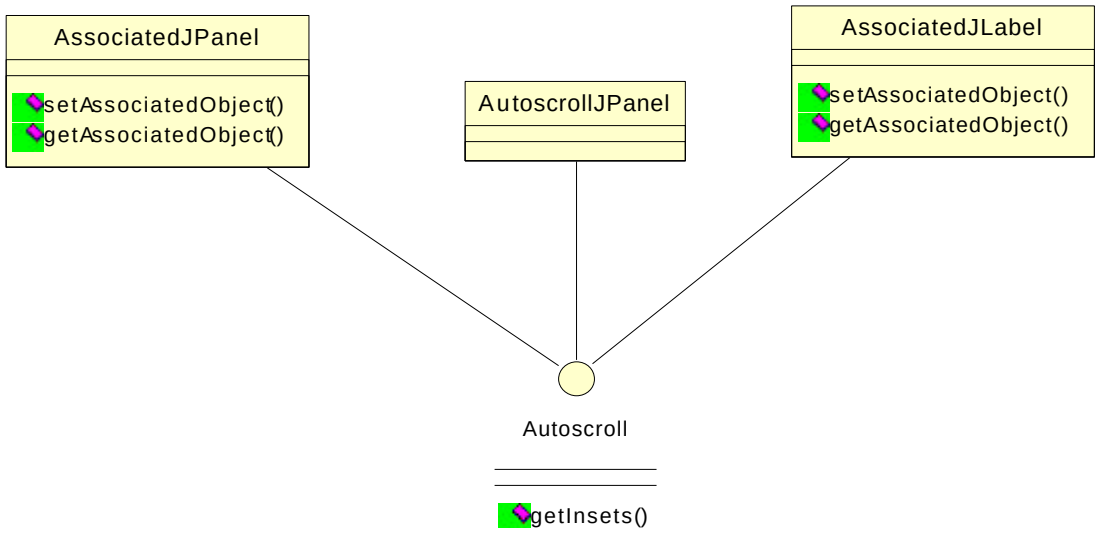


Abbildung 15 Klassen, die Autoscroll implementieren

Das Interface Autoscroll beinhaltet zwei Methoden. Eine Methode heißt getInsets(). Diese Methode wird ständig während eines Drags aufgerufen. Über sie möchte das System erfahren, in welchem Randbereich ein Scroll ausgeführt werden soll. Man muss hier darauf achten, dass die Ränder des DropTargets und nicht des ViewPort gemeint sind. Also muss hier ständig die Umrechnung von den eigentlich gewünschten Rändern des sichtbaren Teils des Viewports auf die Ränder des DropTargets erfolgen. Nachdem sich das System diese Ränder geholt hat, überprüft es, ob sich der Mauszeiger innerhalb dieser Ränder befindet. Ist dies der Fall, so wird die Methode autoscroll() des jeweiligen DropTargets aufgerufen. Innerhalb dieser Methode muss nun der Scroll-Vorgang durchgeführt werden. Dazu muss zunächst einmal der Viewport ermittelt werden, der für diese Komponente verantwortlich ist. Danach muss über die Mausposition ermittelt werden in welche Richtung gescrollt werden soll, und zum Abschluss muss der ermittelte Viewport gescrollt werden.

7.3.8. Datenobjekte

Zum Abschluss sehen wir uns noch einmal alle Datenobjekte an, die in diesem Projekt benötigt werden. Es sind zwar schon fast alle bei der Erklärung der „Darg and Drop“-Funktionalität erklärt worden, doch wollen wir diese aus Übersichtlichkeitsgründen noch einmal genauer ansehen, da sie für das Projekt sehr wichtig sind.

Abbildung 16 Klassendiagramm der benötigten Datenobjekte

Das zentrale Objekt, was es in dieser Software zu planen gilt ist die Tour. Diese enthält in einem Vector 0 bis n geplante Besuche (PlanedItem). Wie wir später sehen werden ist diese Struktur sehr Ähnlich der Struktur in der Datenbank. Daher werden alle weiteren Details der Datenobjekte erst bei Vorstellung des ER-Diagrammes erklärt.

7.4. ER-Diagramm

Die bearbeiteten und erzeugten Daten müssen natürlich auch über das Ende des Programms hinaus gespeichert werden. Hierzu verwenden wir eine Datenbank. In der Datenbank müssen hierzu erst einmal Tabellen für die Daten angelegt werden. Wir wollen uns hierzu einmal das ER-Diagramm dieser für uns relevanten Daten ansehen.

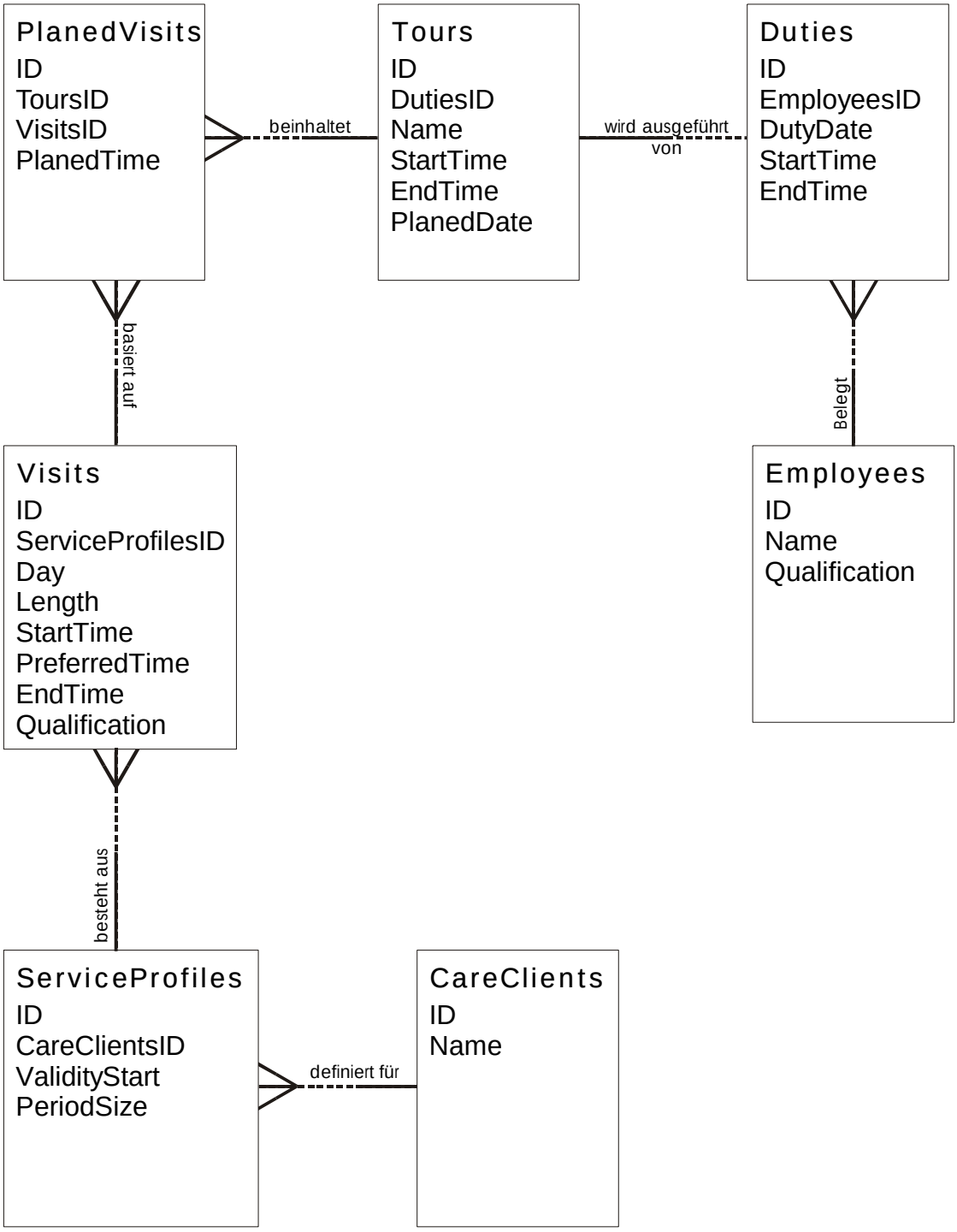


Abbildung 17 ER-Diagramm der relevanten Daten

Dabei sind die Daten, die wir mit diesem Modul verändern oder erzeugen wollen, in den Tabellen Tours und PlanedVisits enthalten. Tours beinhaltet alle Daten über die Touren selbst. Dazu gehören der Name der Tour , die Anfangszeit, die Endzeit und das Datum an dem die Tour geplant ist. Außerdem ist ein Verweis auf die Schicht enthalten, die dieser Tour zugeordnet ist. In PlanedVisits sind zu den Touren alle in ihnen geplanten Besuche enthalten. Dazu muss zu jedem geplanten Besuch ein Verweis auf seine Tour, ein Verweis auf den hier geplanten Besuch (zur Tabelle Visits) und die Zeit an der der Besuch stattfinden soll gespeichert werden.

Alle anderen Tabellen enthalten lediglich Daten, die wir zur Erfüllung dieser Aufgabe benötigen. Im folgenden soll kurz erläutert werden, welche dieser nur zum Lesen bestimmten Tabellen welche für uns wertvolle Informationen enthalten.

Auf der einen Seite wären hier zunächst einmal die Duties. In dieser Tabelle sind die zu Verfügung stehenden Schichten gespeichert. Dazu gehören die Daten zu welchem Datum diese Schicht geplant ist, wann sie anfängt, zu welcher Zeit sie aufhört, und ein Verweis auf den Mitarbeiter, der dieser Schicht zugeordnet ist. Die Mitarbeiter selbst sind in der Tabelle Employees gespeichert. Für unser Projekt ist hier nur der Name und die Qualifikation des Mitarbeiters von Bedeutung.

Auf der anderen Seite haben wir die Besuche. Ein Besuch ist zunächst einmal definiert mit seiner Länge. Darüber hinaus wird noch ein Zeitraum, bzw. Zeitpunkt angegeben, zu dem dieser Besuch ausgeführt werden sollte. Der Zeitraum ist dargestellt durch die Anfangs- und die Endzeit. Wenn ein ganz bestimmter Zeitpunkt bevorzugt ist, wird zusätzlich noch dieser bevorzugte Zeitpunkt angegeben. Jeder Besuch erfordert eine gewisse Qualifikation vom Mitarbeiter. Diese Qualifikation wird hier auch gespeichert. Außerdem ist noch der Tag angegeben, an dem der Besuch durchgeführt wird. Dieser Tag ist eine Zahl zwischen 0 und der Periodenlänge des Service-Profils, auf das wir zur Erklärung hier auch noch eingehen müssen. Ein Service-Profil definiert alle Besuche eines Patienten. Dieses Profil startet an einem festgelegtem Gültigkeitsanfang und wiederholt sich nach einer festgelegten Anzahl von Tagen. Diese Anzahl steht in der Periodengröße (PeriodSize). In dem Profil ist natürlich auch ein Verweis auf den Patienten gespeichert, dem dieses Profil zugeordnet ist. Von dem Patienten interessiert uns hier nur der Name.

7.5. SQL-Statements

Um im Programm mit der Datenbank arbeiten zu können, müssen wir SQL-Statements entwerfen, die uns entweder die gewünschten Daten liefern, oder die geänderten Daten in die Datenbank zurückschreiben. Die hierzu benötigten Befehle werden in diesem Abschnitt vorgestellt.

Folgendes SQL-Statement dient dazu alle ungeplanten Besuche aus der Datenbank zu lesen. Suchdatum gibt hierbei das gewünschte Datum an.

```
SELECT    a.ID AS DutyID,
          a.DutyDate AS DutyDate,
          a.StartTime AS DutyStartTime,
          a.EndTime AS DutyEndTime,
          c.ID AS EmployeeID,
          c.Name AS EmployeeName,
          c.Qualification AS EmployeeQualification
FROM      Duties AS a,
          Employees AS c
WHERE     a.EmployeesID=c.ID and
          a.DutyDate=Suchdatum and
          (a.ID NOT IN (
               SELECT    d.id
               FROM      duties as d,
                         tours as e
               WHERE     e.DutiesID=d.ID and
                         e.PlanedDate=Suchdatum and
                         d.DutyDate= Suchdatum
               )
          );
```

Mit dem nächsten einfachen Statement liest man nacheinander alle CareClinetIDs aus der Datenbank:

```
SELECT a.ID AS CareClientsID
FROM CareClients AS a
```

Um für jeden einzelnen CareClient nun ein gültiges ServiceProfil zu einem bestimmten Datum zu erhalten, benötigen wir folgenden Befehl. Das Datum ist in Suchdatum und die CareClient-ID in SuchID enthalten.

```
SELECT    b.ID AS ServiceProfileID,
          b.ValidityStart AS ServiceProfileValidityStart,
          b.PeriodSize as ServiceProfilePeriodSize
FROM      ServiceProfiles AS b
WHERE     b.CareClientsID=SuchID and
          b.ValidityStart IN (
               SELECT    max(a.ValidityStart)
               FROM      ServiceProfiles AS a
               WHERE     a.ValidityStart<=Suchdatum And
                         a.CareClientsID=SuchID
          );
```

Wenn man nun das folgende Statement anwendet, erhält man alle ungeplanten Besuche für die aus den letzten zwei Statements gewonnenen Daten. Nur den SuchTag muss man sich noch mit dem SuchDatum und dem ServiceProfil ausrechnen.

```
SELECT    d.ID AS VisitsID,
          d.Length AS VisitsLength,
          d.StartTime AS VisitsStartTime,
          d.PreferredTime AS VisitsPreferredTime,
          d.EndTime AS VisitsEndTime,
          d.Qualification AS VisitsQualification,
          f.ID AS CareClientsID,
          f.Name AS CareClientsName
FROM      visits AS d,
          ServiceProfiles AS e,
          careclients AS f
WHERE     d.Day=SuchTag AND
          d.ServiceProfilesID=e.ID and
          e.ID=SuchServiceProfil and
          e.careclientsID=f.ID and
          ((d.ID) Not In (
               SELECT    a.ID
               FROM      visits AS a,
                         planedVisits AS b,
                         tours AS c
               WHERE     b.toursID=c.ID And
                         b.VisitsID=a.ID And
                         c.PlanedDate=SuchDatum
               )
          );
```

Um alle Touren für ein gewünschtes Datum zu erhalten, setzen wir folgenden Befehl ein:

```
SELECT    a.ID AS ToursID,
          a.DutiesID AS ToursDutiesID,
          a.Name AS ToursName,
          a.StartTime AS ToursStartTime,
          a.EndTime AS ToursEndTime
FROM      tours AS a
WHERE     a.PlanedDate=SuchDatum;
```

Wenn man jetzt aufgrund dieser Daten eine bestimmte Schicht aus der Datenbank holen möchte, so erledigt man diese Arbeit unter Angabe der SuchDutyID mit folgendem Befehl:

```
SELECT    a.ID AS DutyID,
          a.DutyDate AS DutyDate,
          a.StartTime AS DutyStartTime,
          a.EndTime AS DutyEndTime,
          c.ID AS EmployeeID,
          c.Name AS EmployeeName,
          c.Qualification AS EmployeeQualification
FROM      Duties AS a,
          Employees AS c
WHERE     a.ID=SuchDutyID and
          a.EmployeesID=c.ID;
```

Alle Besuche zu jeder einzelnen Tour werden mit dem nächsten Statement geliefert. Hier wird die SuchTourID angegeben.

```
SELECT    d.ID AS VisitsID,
          d.Length AS VisitsLength,
          d.StartTime AS VisitsStartTime,
          d.PreferredTime AS VisitsPreferredTime,
          d.EndTime AS VisitsEndTime,
          d.Qualification AS VisitsQualification,
          f.ID AS CareClientsID,
          f.Name AS CareClientsName,
          a.PlanedTime AS PlanedVissitsPlanedTime
FROM      planedVisits AS a,
          visits AS d,
          ServiceProfiles AS e,
          careclients AS f
WHERE     a.ToursID=SuchTourID and
          a.VisitsID=d.ID and
          d.ServiceProfilesID=e.ID and
          e.CareClientsID=f.ID;
```

Alle folgenden SQL-Befehle werden wir benötigen um Daten wieder zurück in die Datenbank zu schreiben.

Fangen wir an mit dem Statement das benötigt wird um vor dem speichern erst einmal alle geplanten Besuche zu einer Tour wieder zu löschen. Gegeben ist hier wieder die entsprechende SuchTourID:

```
DELETE    *
FROM      planedvisits AS a
WHERE     a.ToursID=SuchTourID;
```

Nachdem dieser Befehl auf jeder Tour ausgeführt wurde, können wir mit einem Befehl alle Touren löschen. Gegeben ist diesmal das SuchDatum.

```
DELETE    *
FROM      tours AS a
WHERE     a.PlanedDate=SuchDatum;
```

Nachdem die Datenbank auf diese Weise bereinigt wurde kann man sie wieder füllen. Um für die nächste zu speichernde Tour die ID zu erhalten verwenden wir folgenden Befehl[14]:

```
SELECT max([ID])+1 AS NewToursID
FROM Tours;
```

Gleiches gilt für die neue ID eines geplanten Besuches:

```
SELECT    max([ID])+1 AS NewPlanedVisitsID
FROM PlanedVisits;
```

Um nun eine neue Tour anzulegen verwenden wir diesen Befehl: (Die mit * gekennzeichneten Werte sind zu übergeben)

```
INSERT INTO Tours
     (ID,DutiesID,Name,StartTime,EndTime,PlanedDate)
VALUES
     (newTourID*,
     AssignedDutyID*,
     TourName*,
     StartTime*,
     EndTime*,
     Datum*);
```

[14] Dies wird nur verwendet, wenn in der Datenbank das ID-Feld der Tour nicht als Autowert-Feld definiert wurde.

Geplante Besuche zu einer Tour müssen nun zuletzt auch noch angelegt werden:

```
INSERT INTO PlanedVisits
     (ID,ToursID,VisitsID,PlanedTime)
Values
     (newPlanedVisitID*,
     newTourID*,
     VisitsID*,
     PlanedTime*);
```

Dies waren nun alle SQL-Statements die in diesem Projekt zur Anwendung kommen. Die konkrete Implementierung lässt sich im Quelltext des Programms nachlesen. Alle Datenbankaktivitäten finden in der Klasse CarePlanModelDatabaseConnection statt.

8. Zusammenfassung

Zum Abschluss dieser Arbeit fassen wir noch einmal einige Erfahrungen und Probleme zusammen, die während dieses Projektes aufgetreten sind. Zunächst einmal wären hier einige Anmerkungen zur Programmiersprache Java zu sagen. Obwohl es sich hierbei um eine stilistisch sehr schöne Programmiersprache handelt gibt es auch noch einige Punkte, die den Einsatz dieser Sprache erschweren. Da wäre zunächst einmal die noch nicht vollends ausgereiften virtuellen Maschinen (JVM). So läuft beispielsweise das hier entwickelte Programm auf der Java-Version 1.2.1 tadellos, wogegen es bei Version 1.2.2 regelmäßig zu Abstürzen der virtuellen Maschine kommt. Ein weiteres Manko ist, dass einige Dinge nicht ausreichend dokumentiert wurden. Hier sei nur gesagt, dass ein Transferable serialisierbar sein muss. Diese wichtige Information sucht man in der Java API vergebens, und so gehen einige Stunden an Test- und Debugaktivitäten ins Land, bevor man ein Problem gelöst hat, welches man eigentlich gar nicht selbst zu verantworten hat. Genauso sensibel reagiert die JVM beim Eventhandling. So darf lange nicht alles während eines Event-Aufrufs geschehen.

Die Rationalisierung von ambulanten Pflegediensten wird vor dem Hintergrund diverser Kürzungen im Gesundheitssektor immer wichtiger. Trotzdem sind die Bedürfnisse der gepflegten Personen nicht zu vernachlässigen. In manchen Fällen sollte die Qualität der Pflege bzw. Pflegeplanung noch verbessert werden. Dies war auch ein Grund für die Entstehung dieses Programms. Beispielsweise mit der Einführung einer neuen Sichtweise bei der Personaleinsatzplanung und der damit verbundenen Integration des Management-Werkzeuges „Balanced Scorecard“ ist ein deutlicher Schritt in diese Richtung gemacht worden.

Jedoch darf man nicht aus den Augen verlieren, dass dies nur ein Prototyp ist, den es nun zu validieren und in bestehende Systeme zu integrieren gilt. Im Rahmen dieser Arbeit ist es noch nicht geschehen den fertigen Prototypen zukünftigen Anwendern vorzustellen, und somit noch wichtige Informationen für spätere Weiterentwicklungen dieses Moduls zu sammeln. Dieser Schritt wird jedoch von anderen Teilnehmern des SWIFT-Projektes noch nachgeholt, damit der Kreislauf der Softwareentwicklung, indem das explorative Prototypisieren nur ein Schritt ist, erfolgreich fortgesetzt werden kann.

Für die Einbettung dieses Moduls in ein größeres System sollte sich der Programmierer folgende Klassen genauer ansehen:

- CarePlanModule Starter
- DateChooser
- CarePlanModelDatabaseConnection

Neben den Datenobjekt-Klassen enthalten diese Klassen die wichtigsten Informationen, wie man die Komponenten auf den Bildschirm bekommt, und wie die Datenbankanbindung mit diesem Modul funktioniert. Weitere Details findet man dann im Anhang beim dokumentierten Quelltext.

Die nächste Erweiterung dieses Programms könnte sein, dass man Kennzahlen im Hinblick auf Einbindung der Balanced Scorecard einbaut. Diese können auf einer separaten View angezeigt werden. Beispielsweise könnte man zu einzelnen Touren anzeigen lassen, wie viel Leerlauf in ihnen enthalten ist. Oder man könnte für Patienten festlegen zu wie viel Prozent ihre Wünsche erfüllt sind, oder wie viele verschieden Mitarbeiter sie betreuen. All dies sind Kennzahlen, die den Erfolg der Planung belegen könnten. Und genau diesen Erfolg nachzuvollziehen ist der Sinn einer Balanced Scorecard.

9. Anhang

9.1. Wichtige Informationen zum Start des Prototypen

Es gibt nur ein paar Dinge, die man beim Start des Prototypen beachten muss. Diese werden im folgenden aufgeführt.

1. Die Datenbank muss unter dem Namen „CarePlanModel“ als ODBC-System-Datenbank eingerichtet werden.
2. In der Testdatenbank sind nur Daten zum 01.01.2000 eingetragen. Vor diesem Datum bestehen keine Service-Profile und Schichten, und nach diesem Datum bestehen keine Schichten.
3. Es sollte JDK 1.2.1 verwendet werden. Unter dieser Version wurde der Prototyp bereits getestet. Unter der Version 1.2.2 stürzt die Java-Virtual-Maschine ab. Evtl. läuft das Programm aber wieder unter der Version 1.3. Dies wurde noch nicht verifiziert.
4. Die Start-Klasse heißt CarePlanModuleStarter. An Sie müssen keine Argumente übergeben werden.
5. Nach dem Start werden drei Fenster geöffnet, die alle übereinander liegen: die TourView, die ClientView und die Datumsauswahl. Um vernünftig mit dem Programm arbeiten zu können, sollten diese Fenster wenn möglich nebeneinander, oder zumindest überlappend angeordnet werden. Dann lassen sich die View-übergreifenden Änderungen am besten beobachten.

9.2. Screenshots des Programms

Folgendes Bild zeigt die Planung aus der Tourensicht. Wie hier zu erkennen ist, stehen die Touren im Mittelpunkt der Planung und der zu planenden Besuche befinden sich auf der linken Seite des Fensters. Auf der rechten Seite befinden sich die zu Verfügung stehenden Schichten der Mitarbeiter.

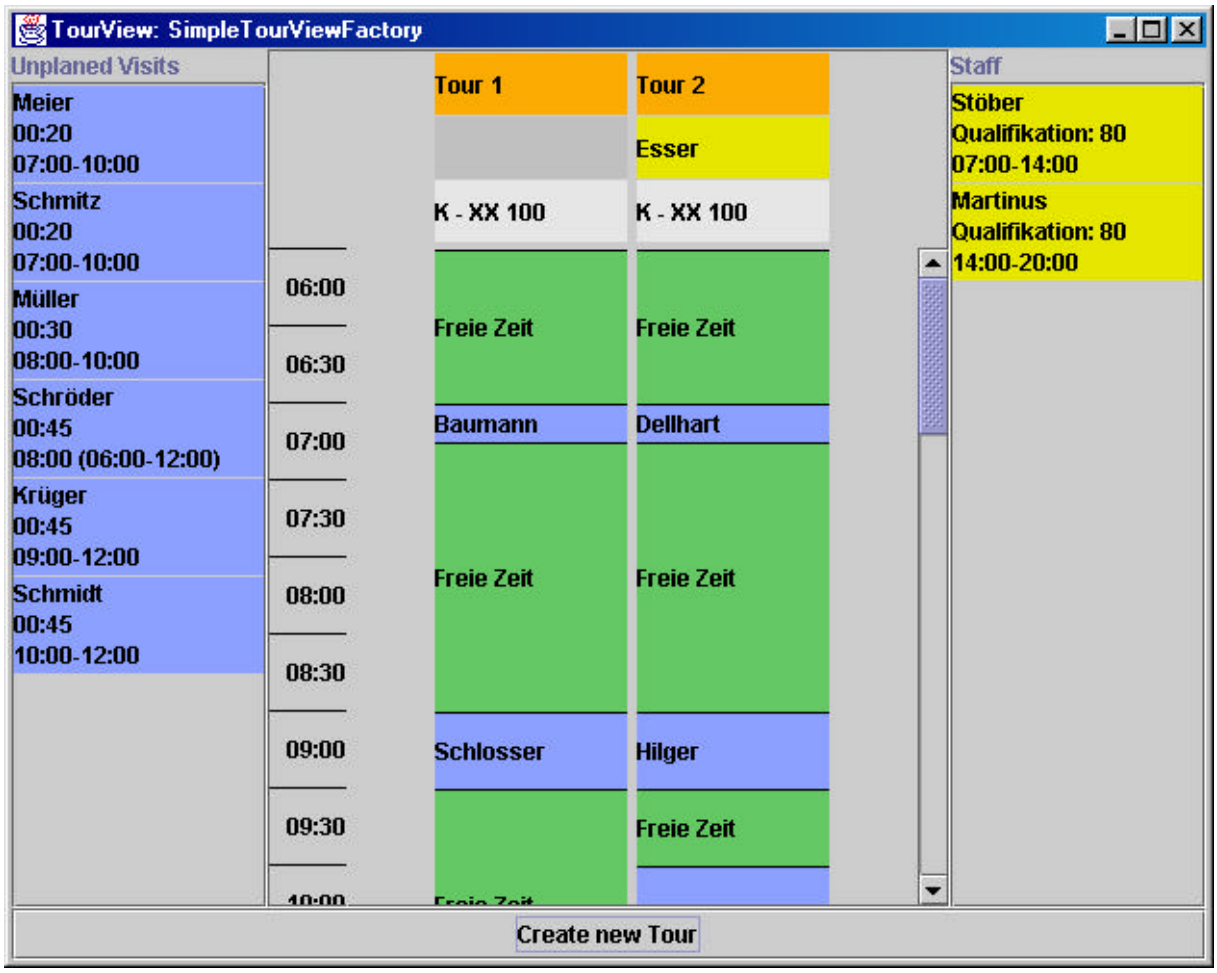


Abbildung 18 Screenshot der Planung aus Tourensicht

Das nächste Bild zeigt im Gegensatz hierzu die Planung aus Sicht eines einzelnen Klienten. Hier steht natürlich der Klient in der Mitte. Zusätzlich ist der einzelne Patient noch in Soll und Ist unterteilt. Im Soll stehen alle Besuche, die er laut Pflegevertrag bekommen sollte, im Ist sieht man die tatsächlich geplanten Besuche. Zu beachten ist hierbei, dass bei den Soll-Besuchen der Zeitraum in dem der Besuch stattfinden sollte, eingefärbt ist, wogegen im Ist nur der Bereich eingefärbt ist, an dem der Besuch tatsächlich stattfindet. Dieser Bereich ist im Normalfall sehr viel kleiner. Auf der rechten Seite kann man die Touren, aber auch die Schichten der Mitarbeiter direkt auswählen, um sie dem Patienten zuzuweisen. Dies geschieht per Drag and Drop auf den Soll- oder auf den Ist-Besuchen. Auf der rechten Seite kann man den angezeigten Klienten auswählen.

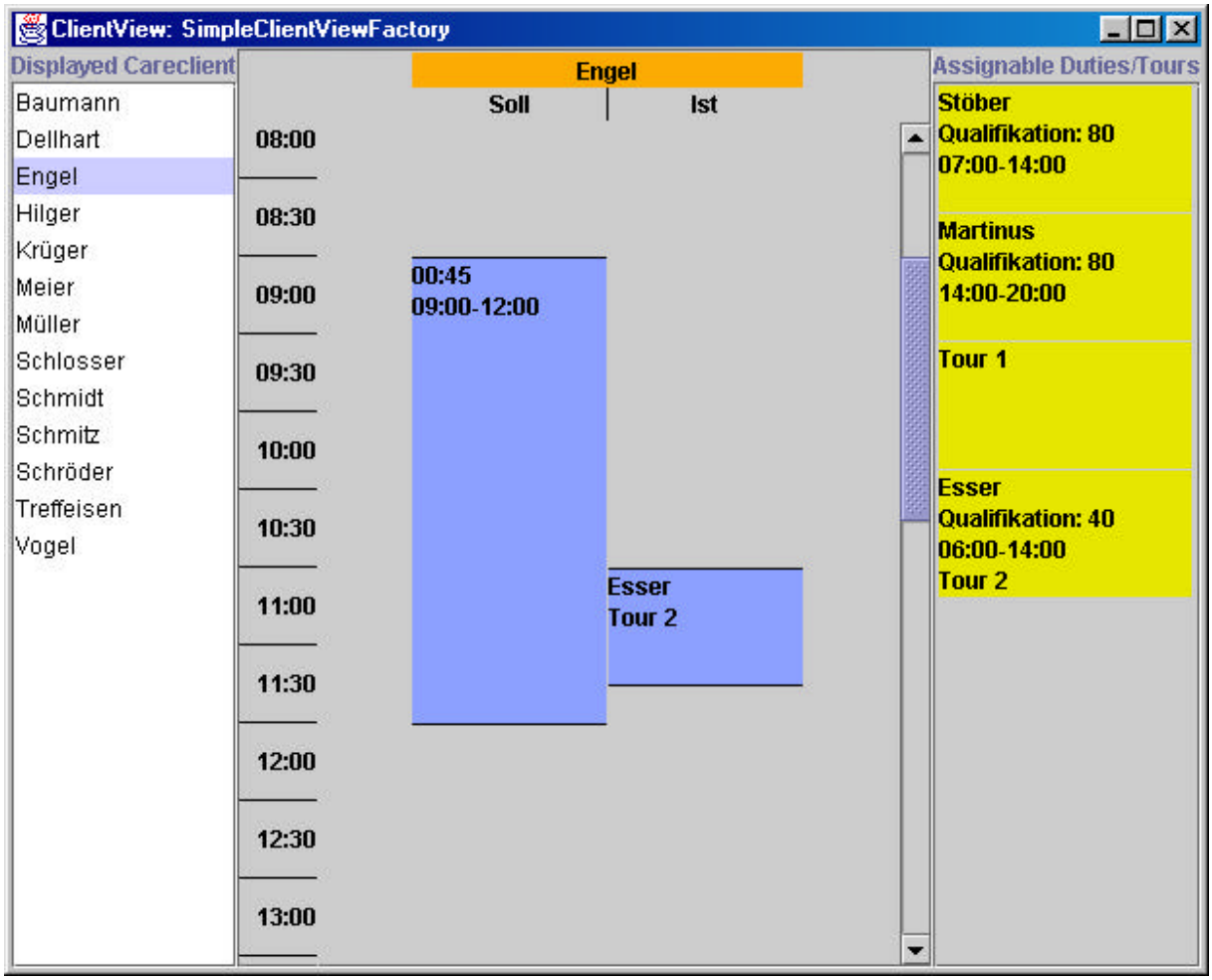


Abbildung 19 Screenshot der Planung aus Patientensicht

Fehlt nur noch das Fenster indem man das Datum wählen kann:

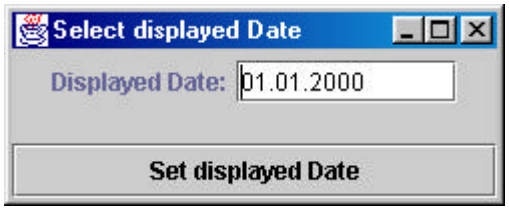


Abbildung 20 Screenshot der Datumsauswahl

9.3. Quelltexte

9.3.1. Klasse „AssignedDutiesDragAndDrop“

```
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

/**
 * Berabeitet alle Drag-Operationen auf geplanten Schichten.
 * Es wird die TourViewFactory benötigt, um von der Komponente auf
 * das Datenobjekt zu schliessen. Desweiteren wird das benutzte
 * CarePlanModel benötigt, um die Daten, die durch die Operationen
 * verändert wurden aktualisieren zu können.
 *
 */

public class AssignedDutiesDragAndDrop
  implements DragGestureListener,DragSourceListener {

  private TourViewFactory usedFactory;
  private CarePlanModel usedModel;

  public AssignedDutiesDragAndDrop(){
    this.usedFactory=new SimpleTourViewFactory();
  }

  /**
   * Konstruktor, der direkt die benutzte TourViewFactory übergibt.
   * @param usedFactory Die TourViewFactory, die von der Klasse verwendet werden
soll.
   *
   */
  public AssignedDutiesDragAndDrop(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  /**
   * Benutzte TourViewFactory kann hier festgelegt werden.
   * @param usedFactory Die TourViewFactory, die von der Klasse verwendet werden
soll.
   *
   */
  public void setTourViewFactory(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  /**
   * Benutztes CarePlanModel kann hier festgelegt werden.
   * @param usedModel Das CarePlanModel, das von der Klasse verwendet werden soll.
   *
   */
  public void setCarePlanModel(CarePlanModel usedModel){
    this.usedModel=usedModel;
  }

  /**
   * Hier wird der Drag gestartet. Diese Methode ist von DragGestureListener
   * geerbt un hier implementiert.
   * @param dge Das Event, welches diese Methode ausgelöst hat.
   *
   */
  //DragGestureListener
```

```
  public void dragGestureRecognized(DragGestureEvent dge){

  if(usedFactory.getTourFromHeaderComponent(dge.getComponent()).getAssignedDuty()!=
null)
      dge.getDragSource().startDrag(dge, DragSource.DefaultMoveDrop, new
TransferablePlanedDutyData(new
PlanedDutyData(usedFactory.getTourFromHeaderComponent(dge.getComponent()).getAssign
edDuty(),usedFactory.getTourFromHeaderComponent(dge.getComponent()))), this);
  }

  /**
   * Nach einem vollendeten Drag wird diese Methode aufgerufen.
   * Von DragSourceListener geerbt.
   * Wenn der Drag-Vorgang nicht erfolgreich war, wird die 'gezogene' Schicht
   * in die Liste der ungeplanten Schichten eingereiht.
   * @param DragSourceDropEvent Das Event, welches diese Methode ausgelöst hat.
   *
   */
  //DragSourceListener
  public void dragDropEnd(DragSourceDropEvent DragSourceDropEvent) {
    if(!DragSourceDropEvent.getDropSuccess()){
      //aus der Tour entfernen und in die ungeplantenListe schreiben.
      Tour
dragTour=usedFactory.getTourFromHeaderComponent(DragSourceDropEvent.getDragSourceCo
ntext().getComponent());
      usedModel.addDuty(dragTour.getAssignedDuty());
      usedModel.unassignDutyFromTour(dragTour.getAssignedDuty(),dragTour);

      //Dieser umständliche Aufruf von 'usedModel.CarePlanModelChanged(null,1)' muß
sein
      //da sich sonst JAVA aufhängt!
      SwingUtilities.invokeLater(new Runnable() {
          public void run() {
             usedModel.CarePlanModelChanged(null,1);
          }
        }
      );
    }
  }

  public void dragEnter(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dragExit(DragSourceEvent DragSourceEvent) {
  }
  public void dragOver(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dropActionChanged(DragSourceDragEvent DragSourceDragEvent) {
  }

}
```

9.3.2. Interface „AssociatedComponent“

```
/**
* Dieses Interface legt eine Methode fest, um auf ein zugeordnetes Objekt
zuzugreifen.
*
*/

public interface AssociatedComponent{
  /**
  * Dieses Methode gibt das zu dem Objekt zugeordnete Datenobjekt zurück.
  * @return zugeordnetes Datenobjekt
  */
  public Object getAssociatedObject();
}
```

9.3.3. Klasse „AssociatedJLabel“

```
import javax.swing.*;
import java.awt.dnd.*;
import java.awt.*;

/**
* Definiert ein JLabel, welchem ein Datenobjekt zugeordnet werden kann.
* Außerdem ist das Autoscroll-Interface Implementiert.
*
*/

public class AssociatedJLabel extends JLabel implements
AssociatedComponent,Autoscroll{

  //Anfang Autoscroll

  /**
  * Legt die Größe des Randbereichs fest, innerhalb dessen gescrollt werden soll.
  */
  final static private int intScrollInsets=20;

  private Insets actInsets=new Insets(0,0,0,0);

  private JViewport getMainViewport(Component source){
    Component actComponent=source;
    boolean viewportNotFound=true;
    while(viewportNotFound){
      if(actComponent.getParent()!=null){
        if(actComponent.getParent().getClass()==(new JScrollPane()).getClass()){
          return (JViewport)actComponent;
        }
        actComponent=actComponent.getParent();
      }
      else {
        return null;
      }
    }
    return null;
  }

  public void autoscroll(Point cursorLocn){
    JViewport scrollableViewport=getMainViewport(this);
    boolean mustScroll=false;
    Dimension viewSize=scrollableViewport.getExtentSize();
    Dimension viewRealSize=scrollableViewport.getViewSize();
    Point viewPoint=scrollableViewport.getViewPosition();
    Point
aktPoint=SwingUtilities.convertPoint(scrollableViewport.getView(),viewPoint,this);
    Point visPoint=aktPoint;
    if(cursorLocn.getY()<actInsets.top){
      mustScroll=true;
      if(viewPoint.getY()-(actInsets.top-cursorLocn.getY())>=0){
        visPoint=new Point((int)viewPoint.getX(),(int)(viewPoint.getY()-
(actInsets.top-cursorLocn.getY())));
      }
      else {
        visPoint=new Point((int)viewPoint.getX(),0);
      }
    }
    else {
      if(cursorLocn.getY()>(this.getHeight()-actInsets.bottom)){
        mustScroll=true;
        if((int)viewPoint.getY()+(int)(cursorLocn.getY()-(this.getHeight()-
actInsets.bottom))<=viewRealSize.getHeight()-viewSize.getHeight())
          visPoint=new
Point((int)viewPoint.getX(),(int)viewPoint.getY()+(int)(cursorLocn.getY()-
(this.getHeight()-actInsets.bottom)));
```

```
          else
            visPoint=new Point((int)viewPoint.getX(),(int)(viewRealSize.getHeight()-
viewSize.getHeight()));
        }
      }
      if(mustScroll){
        visPoint.x=0;
        scrollableViewport.setViewPosition(visPoint);
      }
    }

    public Insets getAutoscrollInsets() {

      JViewport scrollableViewport=getMainViewport(this);
      Dimension viewSize=scrollableViewport.getExtentSize();
      Dimension viewRealSize=scrollableViewport.getViewSize();
      Point aktPoint=scrollableViewport.getViewPosition();

      Point
calculatedPoint=SwingUtilities.convertPoint(scrollableViewport.getView(),aktPoint,t
his);
      double yTop=calculatedPoint.getY()+intScrollInsets;
      double yBottom;

      if((yTop+viewSize.getHeight()-
2*intScrollInsets)>=0||(yTop+viewSize.getHeight()-
2*intScrollInsets)<=this.getHeight()){
        yBottom=this.getHeight()-(yTop+viewSize.getHeight()-2*intScrollInsets);
      }
      else {
        yBottom=0;
      }
      if(yTop<0)
        yTop=0;
      actInsets=new Insets((int)yTop,0,(int)yBottom,0);
      return actInsets;
    }

    //Ende Autoscroll

    private Object AssociatedObject;

    public AssociatedJLabel(){
      this.AssociatedObject=null;
    }

    public AssociatedJLabel(String newLabel,Object AssociatedObject){
      super(newLabel);
      this.AssociatedObject=AssociatedObject;
    }

    public AssociatedJLabel(Object AssociatedObject){
      this.AssociatedObject=AssociatedObject;
    }

    public AssociatedJLabel(String newLabel){
      super(newLabel);
      this.AssociatedObject=null;
    }

    public Object getAssociatedObject(){
      return AssociatedObject;
    }

    public void setAssociatedObject(Object AssociatedObject){
      this.AssociatedObject=AssociatedObject;
    }
```

```
}
```

9.3.4. Klasse „AssociatedJPanel“

```
import javax.swing.*;
import java.awt.dnd.*;
import java.awt.*;

  //Dokumentation wie bei AssociatedJLabel

public class AssociatedJPanel extends JPanel implements
AssociatedComponent,Autoscroll{

  //Anfang Autoscroll

  final static private int intScrollInsets=20;

  private Insets actInsets=new Insets(0,0,0,0);

  private JViewport getMainViewport(Component source){
    Component actComponent=source;
    boolean viewportNotFound=true;
    while(viewportNotFound){
      if(actComponent.getParent()!=null){
        if(actComponent.getParent().getClass()==(new JScrollPane()).getClass()){
          return (JViewport)actComponent;
        }
        actComponent=actComponent.getParent();
      }
      else {
        return null;
      }
    }
    return null;
  }

  public void autoscroll(Point cursorLocn){
    JViewport scrollableViewport=getMainViewport(this);
    boolean mustScroll=false;
    Dimension viewSize=scrollableViewport.getExtentSize();
    Dimension viewRealSize=scrollableViewport.getViewSize();
    Point viewPoint=scrollableViewport.getViewPosition();
    Point
aktPoint=SwingUtilities.convertPoint(scrollableViewport.getView(),viewPoint,this);
    Point visPoint=aktPoint;
    if(cursorLocn.getY()<actInsets.top){
      mustScroll=true;
      if(viewPoint.getY()-(actInsets.top-cursorLocn.getY())>=0){
        visPoint=new Point((int)viewPoint.getX(),(int)(viewPoint.getY()-
(actInsets.top-cursorLocn.getY())));
      }
      else {
        visPoint=new Point((int)viewPoint.getX(),0);
      }
    }
    else {
      if(cursorLocn.getY()>(this.getHeight()-actInsets.bottom)){
        mustScroll=true;
        if((int)viewPoint.getY()+(int)(cursorLocn.getY()-(this.getHeight()-
actInsets.bottom))<=viewRealSize.getHeight()-viewSize.getHeight())
          visPoint=new
Point((int)viewPoint.getX(),(int)viewPoint.getY()+(int)(cursorLocn.getY()-
(this.getHeight()-actInsets.bottom)));
        else
          visPoint=new Point((int)viewPoint.getX(),(int)(viewRealSize.getHeight()-
viewSize.getHeight()));
```

```
      }
    }
    if(mustScroll){
      visPoint.x=0;
      scrollableViewport.setViewPosition(visPoint);
    }
  }

  public Insets getAutoscrollInsets() {

    JViewport scrollableViewport=getMainViewport(this);
    Dimension viewSize=scrollableViewport.getExtentSize();
    Dimension viewRealSize=scrollableViewport.getViewSize();
    Point aktPoint=scrollableViewport.getViewPosition();

    Point
calculatedPoint=SwingUtilities.convertPoint(scrollableViewport.getView(),aktPoint,t
his);
    double yTop=calculatedPoint.getY()+intScrollInsets;
    double yBottom;

    if((yTop+viewSize.getHeight()-
2*intScrollInsets)>=0||(yTop+viewSize.getHeight()-
2*intScrollInsets)<=this.getHeight()){
      yBottom=this.getHeight()-(yTop+viewSize.getHeight()-2*intScrollInsets);
    }
    else {
      yBottom=0;
    }
    if(yTop<0)
      yTop=0;
    actInsets=new Insets((int)yTop,0,(int)yBottom,0);
    return actInsets;
  }

  //Ende Autoscroll

  private Object AssociatedObject;

  public AssociatedJPanel(){
    this.AssociatedObject=null;
  }

  public AssociatedJPanel(Object AssociatedObject){
    this.AssociatedObject=AssociatedObject;
  }

  public Object getAssociatedObject(){
    return AssociatedObject;
  }

  public void setAssociatedObject(Object AssociatedObject){
    this.AssociatedObject=AssociatedObject;
  }

}
```

9.3.5. Klasse „AutoscrollJPanel“

```
import javax.swing.*;
import java.awt.dnd.*;
import java.awt.*;

/** Definiert ein JPanel, welches die Autoscroll-Funktionalität implementiert.
 */
```

```
public class AutoScrollJPanel extends JPanel implements Autoscroll{

  //Anfang Autoscroll

  final static private int intScrollInsets=20;

  private Insets actInsets=new Insets(0,0,0,0);

  private JViewport getMainViewport(Component source){
    Component actComponent=source;
    boolean viewportNotFound=true;
    while(viewportNotFound){
      if(actComponent.getParent()!=null){
        if(actComponent.getParent().getClass()==(new JScrollPane()).getClass()){
          return (JViewport)actComponent;
        }
        actComponent=actComponent.getParent();
      }
      else {
        return null;
      }
    }
    return null;
  }

  public void autoscroll(Point cursorLocn){
    JViewport scrollableViewport=getMainViewport(this);
    boolean mustScroll=false;
    Dimension viewSize=scrollableViewport.getExtentSize();
    Dimension viewRealSize=scrollableViewport.getViewSize();
    Point viewPoint=scrollableViewport.getViewPosition();
    Point
aktPoint=SwingUtilities.convertPoint(scrollableViewport.getView(),viewPoint,this);
    Point visPoint=aktPoint;
    if(cursorLocn.getY()<actInsets.top){
      mustScroll=true;
      if(viewPoint.getY()-(actInsets.top-cursorLocn.getY())>=0){
        visPoint=new Point((int)viewPoint.getX(),(int)(viewPoint.getY()-
(actInsets.top-cursorLocn.getY())));
      }
      else {
        visPoint=new Point((int)viewPoint.getX(),0);
      }
    }
    else {
      if(cursorLocn.getY()>(this.getHeight()-actInsets.bottom)){
        mustScroll=true;
        if((int)viewPoint.getY()+(int)(cursorLocn.getY()-(this.getHeight()-
actInsets.bottom))<=viewRealSize.getHeight()-viewSize.getHeight())
          visPoint=new
Point((int)viewPoint.getX(),(int)viewPoint.getY()+(int)(cursorLocn.getY()-
(this.getHeight()-actInsets.bottom)));
        else
          visPoint=new Point((int)viewPoint.getX(),(int)(viewRealSize.getHeight()-
viewSize.getHeight()));
      }
    }
    if(mustScroll){
      visPoint.x=0;
      scrollableViewport.setViewPosition(visPoint);
    }
  }

  public Insets getAutoscrollInsets() {

    JViewport scrollableViewport=getMainViewport(this);
    Dimension viewSize=scrollableViewport.getExtentSize();
    Dimension viewRealSize=scrollableViewport.getViewSize();
```

```
    Point aktPoint=scrollableViewport.getViewPosition();

    Point
calculatedPoint=SwingUtilities.convertPoint(scrollableViewport.getView(),aktPoint,t
his);
    double yTop=calculatedPoint.getY()+intScrollInsets;
    double yBottom;

    if((yTop+viewSize.getHeight()-
2*intScrollInsets)>=0||(yTop+viewSize.getHeight()-
2*intScrollInsets)<=this.getHeight()){
      yBottom=this.getHeight()-(yTop+viewSize.getHeight()-2*intScrollInsets);
    }
    else {
      yBottom=0;
    }
    if(yTop<0)
      yTop=0;
    actInsets=new Insets((int)yTop,0,(int)yBottom,0);
    return actInsets;
  }

  //Ende Autoscroll

}
```

9.3.6. Klasse „CareClient“

```
import java.io.*;

/** Klasse, die eine CareClient beinhaltet. Die ID wird automatisch gesetzt,
 * sofern sie nicht im Konstruktoir übergeben wird, oder nachträglich
 * verändert wird. Implementiert PlanableCareClient, um vom System
 * verwendet werden zu können.
 */
public class CareClient implements PlanableCareClient,Serializable{

  private Long iD;
  private String name;

  public static Long highestID=new Long(0);

  public CareClient(){
    iD=new Long(highestID.longValue());
    highestID=new Long(highestID.longValue()+1);
  }

  public CareClient(String name){
    iD=new Long(highestID.longValue());
    highestID=new Long(highestID.longValue()+1);
    this.name = name;
  }

  public CareClient(String name, Long iD){
    this.name = name;
    setID(iD);
  }

  public String toString(){
    return name;
  }

  public boolean equals(PlanableCareClient cc){
    return ((CareClient)cc).getID().equals(iD);
  }
```

```
  public String getName(){
    return this.name;
  }

  public void setName(String name){
    this.name = name;
  }

  public Long getID(){
    return this.iD;
  }

  public void setID(Long iD){
    if(iD!=null){
      if(iD.compareTo(highestID)>0)
        highestID=new Long(iD.longValue()+1);
      this.iD = iD;
    }
  }
}
```

9.3.7. Klasse „CarePlanClientView“

```
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.dnd.*;
import java.text.*;

/**Stellt ein Beispiel dar, wie eine Planung aus Klientensicht vollzogen
 * werden könnte. Wird über die Methode des CarePlanModelChangeListener
 * über Änderungen informiert und aktualisiert sich darüber ständig.
 *
 */
public class CarePlanClientView
  implements CarePlanModelChangeListener, ListSelectionListener{

  private CarePlanModel UsedModel;
  private JPanel clientViewPanel;
  private ClientViewFactory usedFactory;

  private Time dayStart=new Time(6,0,0); //6 Uhr
  private Time dayEnd=new Time(22,0,0); //14 Uhr
  private Time dayRaster=new Time(0,30,0); //30 Minuten

  private PlanableCareClient displayedClient=new CareClient("Choose a
Careclient!");

  private JScrollPane mainScrollpane;
  private JList ccList;
  private boolean listIndexSelbstGesetzt;

  private ClientViewUnplanedDutiesDragAndDrop dutiesDnD;
  private NotDropableDropListener dummyDropListener=new NotDropableDropListener();

  /** Liefert alle Komponenten eines Containers zurück.
   * wird z.B. verwendet um alle Componenten eines Containers
   * zu DropTargets zu machen
   */
  private Vector getAllComponentsOfContainer(Container container){
    Container vglClass=new Container();
    Vector returnVector = new Vector();
    Vector subVector = new Vector();
    Component[] AlleKomps=container.getComponents();
    for(int i=0;i<AlleKomps.length;i++){
```

```
      returnVector.add(AlleKomps[i]);
      if(vglClass.getClass().isAssignableFrom(AlleKomps[i].getClass())){
         subVector=getAllComponentsOfContainer((Container)AlleKomps[i]);
         for(int z=0;z<subVector.size();z++){
          returnVector.add(subVector.get(z));
         }
      }
    }
    return returnVector;
  }

  public CarePlanClientView(){
    UsedModel=null;
    clientViewPanel=null;
    usedFactory=new SimpleClientViewFactory();
    dutiesDnD=new ClientViewUnplanedDutiesDragAndDrop(usedFactory);
  }

  /**
   * Erzeugt die eigentliche Komponente dieser Ansicht.
   * Falls schon einmal diese Componente erzeugt wurde, wird
   * nur ihr Inhalt gelöscht und neu gefüllt.
   */
  private void createComponent(){
    int listIndex=-1;
    Point scrollPoint=new Point(0,0);
    if(clientViewPanel!=null){
      scrollPoint=mainScrollpane.getViewport().getViewPosition();
      listIndex=ccList.getSelectedIndex();
      clientViewPanel.removeAll();
    }
    else {
      clientViewPanel=new JPanel();
    }
    clientViewPanel.setLayout(new BorderLayout());

    //Auf der Linken Seite Liste zur Klientenauswahl
    JPanel clientChoicePanel=new JPanel();
    clientChoicePanel.setLayout(new BorderLayout());
    clientChoicePanel.add(new JLabel("Displayed Careclient"),BorderLayout.NORTH);
    ccList=new JList();
    JScrollPane listScrollPane=new JScrollPane();
    listScrollPane.getViewport().add(ccList);
    clientChoicePanel.add(listScrollPane,BorderLayout.CENTER);
    clientViewPanel.add(clientChoicePanel,BorderLayout.WEST);
    if(UsedModel!=null){
      ccList.setListData(UsedModel.getAllCareClients());
      ccList.addListSelectionListener(this);
      if(listIndex>=0){
        listIndexSelbstGesetzt=true;
        ccList.setSelectedIndex(listIndex);
        listIndexSelbstGesetzt=false;
      }
    }

    //In der Mitte wird der gewählte Klient dargestellt
    mainScrollpane=new JScrollPane();
    clientViewPanel.add(mainScrollpane,BorderLayout.CENTER);
    AutoScrollJPanel mainViewportPanel=new AutoScrollJPanel();
    mainViewportPanel.setAutoscrolls(true);
    new DropTarget(mainViewportPanel,dummyDropListener);
    mainScrollpane.getViewport().add(mainViewportPanel);

    //Auf der Linken Seite des Scrollpanes eine Zeitleiste
    mainScrollpane.setRowHeader(new JViewport());
    mainScrollpane.getRowHeader().add(usedFactory.createVerticalTimeBar(new
Time(dayStart),new Time(dayEnd)));
```

```
    JPanel header=new JPanel();
    header.add(usedFactory.createHeaderPanelWithCareClient(displayedClient));
    mainScrollpane.setColumnHeader(new JViewport());
    mainScrollpane.getColumnHeader().add(header);

    //Auf der Rechten Seite des clientViewPanels die geplanten und ungeplanten
Duties
    JPanel rightPanel=new JPanel();
    rightPanel.setLayout(new BorderLayout());
    rightPanel.add(new JLabel("Assignable Duties/Tours"),BorderLayout.NORTH);
    JScrollPane rightScrollPane=new JScrollPane();
    JPanel
dutiesPanel=usedFactory.createDutiesPanelWithDutiesAndTours(UsedModel.getUnassigned
Duties(),UsedModel.getTours());
    rightScrollPane.getViewport().add(dutiesPanel);
    rightPanel.add(rightScrollPane,BorderLayout.CENTER);
    clientViewPanel.add(rightPanel,BorderLayout.EAST);

    //Drag and Drop
    //Alle DragSources definieren:
    Vector allDuties=getAllComponentsOfContainer(dutiesPanel);
    DragSource dutiesDragSource;
    for(int i=0;i<allDuties.size();i++){
      dutiesDragSource=DragSource.getDefaultDragSource();

  dutiesDragSource.createDefaultDragGestureRecognizer((Component)allDuties.get(i),
DnDConstants.ACTION_COPY_OR_MOVE, dutiesDnD);
    }

    //Auf dem MainViewport die geplanten und ungeplanten Besuche
    JPanel
visitsPanel=usedFactory.createBodyPanelWithClientVisits(UsedModel.getUnplanedVisits
WithClient(displayedClient),UsedModel.getPlanedVisitsWithClient(displayedClient));
    mainViewportPanel.add(visitsPanel);

    //Drag and Drop
    //Alle DropTargets definieren:
    Vector allVisits=getAllComponentsOfContainer(visitsPanel);
    DropTarget visitsDropTarget;
    for(int i=0;i<allVisits.size();i++){
      visitsDropTarget=new DropTarget((Component)allVisits.get(i),dutiesDnD);
    }
    mainScrollpane.getViewport().setViewPosition(scrollPoint);
  }

  public void refreshDisplay(){
    if(UsedModel!=null){
    }
    else {
      System.out.println("CarePlanClientView: No Model assigned!");
    }
    createComponent();
    clientViewPanel.updateUI();
  }

  public void setCarePlanModel(CarePlanModel UsedModel){
    this.UsedModel=UsedModel;
    UsedModel.addCarePlanModelChangeListener(this);
    dutiesDnD.setCarePlanModel(UsedModel);
    refreshDisplay();
  }

  //CarePlanModelChangeListener
  public void carePlanModelChanged(CarePlanModelChangeEvent e){
    refreshDisplay();
  }

  public JComponent getComponent(){
```

```
    //createComponent();
    return clientViewPanel;
  }

  public PlanableCareClient getDisplayedClient(){
    return this.displayedClient;
  }

  public void setDisplayedClient(PlanableCareClient displayedClient){
    this.displayedClient=displayedClient;
  }

  //ListSelectionListener
  public void valueChanged(ListSelectionEvent e){
    if(!e.getValueIsAdjusting() && !listIndexSelbstGesetzt){

  displayedClient=(PlanableCareClient)((JList)e.getSource()).getSelectedValue();
      refreshDisplay();
    }
  }

}
```

9.3.8. Klasse „CarePlanModel“

```
import java.util.*;

/** Definiert das Model, welches alle Daten des Systems
 * für einen Tag beinhaltet
 */
public class CarePlanModel{

  private Vector Visits;
  private Vector Duties;
  private Vector Tours;

  private CarePlanModelChangeInformant carePlanModelChangeInformant;

  public CarePlanModel(){
    carePlanModelChangeInformant=new CarePlanModelChangeInformant(this);
    Visits=new Vector();
    Duties=new Vector();
    Tours=new Vector();
  }

  //add- un removeChangeListener an changeInformant weiterleiten
  /**
   * Diese Methode meldet einen Beobachter an diese Klasse an.
   */
  public void addCarePlanModelChangeListener(CarePlanModelChangeListener l){
    carePlanModelChangeInformant.addCarePlanModelChangeListener(l);
  }

  /**
   * Diese Methode entfernt den angegebenen Beobachter aus dieser Klasse.
   */
  public void removeCarePlanModelChangeListener(CarePlanModelChangeListener l){
    carePlanModelChangeInformant.removeCarePlanModelChangeListener(l);
  }

  //Zur Zeit nur benachrichtigen, wenn benachrichtigungswunsch expliziet
  //vom 'Änderer' gewünscht wird.
  /**
   * Diese nur von aussen aufgerufene Methode benachrichtigt alle Beobachter
   * dieser Klasse über eine Änderung. Details zur Änderung können in
   * den Argumenten dieser Klasse mit untergebracht werden. Sie werden vom
```

```
    * aktuellen System jedoch nicht berücksichtigt.
    */
   public void CarePlanModelChanged(Object item,int changeType){
     carePlanModelChangeInformant.fireCarePlanModelChanged(item,changeType);
   }

   /**
    * Mit dieser Methode kann man einen ungeplanten Besuch zur Liste
    * der ungeplanten Besuche hinzufügen. Er wird aufgrund der Planzeiten
    * in die Liste einsortiert. Dabei hat PreferredTime Vorrang vor StartTime.
    */
   public void addVisit(Visit newVisit){
     Time SortTime;
     if(newVisit.getPreferedTime()!=null){
       SortTime=new Time(newVisit.getPreferedTime());
     }
     else {
       SortTime=new Time(newVisit.getStartTime());
     }
     boolean NotAdded=true;
     int InsertPosition=0;
     for(InsertPosition=0;InsertPosition<Visits.size() &&
NotAdded;InsertPosition++){
       Visit vglVisit=(Visit) Visits.get(InsertPosition);
       if(vglVisit.getPreferedTime()!=null){
         if(vglVisit.getPreferedTime().isGreater(SortTime)){
           NotAdded=false;
           InsertPosition--;
         }
       }
       else {
         if(vglVisit.getStartTime().isGreater(SortTime)){
           NotAdded=false;
           InsertPosition--;
         }
       }
     }
     Visits.add(InsertPosition,newVisit);
     System.out.println("Model: Visit added [" + newVisit+"]");
   }

   /**
    * Mit dieser Methode kann man eine ungeplanten Schicht zur Liste
    * der ungeplanten Schichten hinzufügen. Sie wird aufgrund der Schichtzeiten
    * in die Liste einsortiert.
    */
   public void addDuty(Duty newDuty){
     Time SortTime=new Time(newDuty.getStartTime());
     boolean NotAdded=true;
     int InsertPosition=0;
     for(InsertPosition=0;InsertPosition<Duties.size() &&
NotAdded;InsertPosition++){
       Duty vglDuty=(Duty) Duties.get(InsertPosition);
       if(vglDuty.getStartTime().isGreater(SortTime)){
         NotAdded=false;
         InsertPosition--;
       }
     }
     Duties.add(InsertPosition,newDuty);
     System.out.println("Model: Duty added [" + newDuty+"]");
   }

   /**
    * Mit dieser Methode fügt man eine Komplette Tour dem
    * Model hinzu.
    */
   public void addTour(Tour newTour){
     Tours.add(newTour);
```

```
    System.out.println("Model: Tour added [" + newTour.getName() +"]");
  }

  /**
   * Gibt den Vector mit allen ungeplanten Besuchen zurück.
   */
  public Vector getUnplanedVisits(){
    return Visits;
  }

  /**
   * Gibt den Vector mit allen ungeplanten Schichten zurück.
   */
  public Vector getUnassignedDuties(){
    return Duties;
  }

  /**
   * Gibt den Vector mit allen Touren zurück.
   */
  public Vector getTours(){
    return Tours;
  }

  /**
   * Löscht alle Daten aus dem Model
   */
  public void clearModel(){
    Visits=new Vector();
    Duties=new Vector();
    Tours=new Vector();
  }

  /**
   * Ordnet einen ungeplanten Besuch einer Tour zu. Die Tour muss im
   * Model vorhanden sein. Die gewünschte Planzeit wird mit dem Parameter Time
   * übergeben. Anschliessend wird der Besuch ggf. aus der Liste der
   * ungeplanten Besuche gelöscht.
   */
  public void assignVisitToTour(Visit newVisit,Time Time,Tour newTour){
    //Nachsehen, ob newTour im Model ist
    Tour foundTour=null;
    boolean notfound=true;
    for(int i=0;i<Tours.size() && notfound;i++){
      if(Tours.get(i).equals(newTour)){
        foundTour=(Tour)Tours.get(i);
        notfound=false;
      }
    }
    if(notfound){
      System.out.println("Model: Specified Tour not found!");
    }
    else{
      //Suche newVisit in ungeplanten Besuchen
      notfound=true;
      for(int i=0;i<Visits.size() && notfound;i++){
        if(((Visit)Visits.get(i)).equals(newVisit)){
          Visits.remove(i);
          notfound=false;
        }
      }

      //Suche newVisit in geplanten Besuchen
      notfound=true;
      for(int i=0;i<Tours.size();i++){
        ((Tour)Tours.get(i)).removeVisit(newVisit);
      }
```

```
        foundTour.addVisit(newVisit,new Time(Time));
        System.out.println("Model: Added Visit ["+newVisit+"] to Tour
["+foundTour.getName()+"]");
      }
    }

    /**
     * Ordnet einen ungeplanten Besuch einer Tour zu. Die Tour muss im
     * Model vorhanden sein. Die gewünschte Planzeit wird mit dem Parameter
planedItem
     * übergeben. Dieser schon geplante Besuch wird in der Tour gesucht, nach hinten
     * versetzt, un an seine Stelle tritt der neu zu planende Besuch.
     * Anschliessend wird der Besuch ggf. aus der Liste der
     * ungeplanten Besuche gelöscht.
     */
    public void assignVisitToTour(Visit newVisit,PlanedItem planedItem,Tour newTour){
      //Nachsehen, ob newTour im Model ist
      boolean notfound=true;
      Tour foundTour=new Tour();
      for(int i=0;i<Tours.size() && notfound;i++){
        if(Tours.get(i).equals(newTour)){
          foundTour=(Tour)Tours.get(i);
          notfound=false;
        }
      }
      if(notfound){
        System.out.println("Model: Specified Tour not found!");
      }
      else{
        //Suche newVisit in ungeplanten Besuchen
        notfound=true;
        for(int i=0;i<Visits.size() && notfound;i++){
          if(((Visit)Visits.get(i)).equals(newVisit)){
            Visits.remove(i);
            notfound=false;
          }
        }
        foundTour.addVisit(newVisit,planedItem);
        System.out.println("Model: Added Visit ["+newVisit+"] to Tour
["+foundTour.getName()+"]");
      }
    }

    /**
     * Ordnet einen ungeplanten Besuch einer Tour zu. Die Tour muss im
     * Model vorhanden sein. Die Planzeit ist die Start- bzw. PreferredTime
     * des Besuches.
     * Anschliessend wird der Besuch ggf. aus der Liste der
     * ungeplanten Besuche gelöscht.
     */
    public void assignVisitToTour(Visit newVisit,Tour newTour){
      Time planTime;
      if(newVisit.getPreferedTime()!=null){
        planTime=newVisit.getPreferedTime();
      }
      else {
        planTime=newVisit.getStartTime();
      }
      assignVisitToTour(newVisit,planTime,newTour);
    }

    /**
     * Ordnet eine Schicht einer Tour zu. Die Tour muss im
     * Model vorhanden sein. Ggf. wird die Schicht aus der Liste der ungeplanten
     * Schichten gelöscht.
     */
    public void assignDutyToTour(Duty newDuty, Tour newTour){
      //Nachsehen, ob newTour im Model ist
```

```
    Tour foundTour=null;
    boolean notfound=true;
    for(int i=0;i<Tours.size() && notfound;i++){
      if(Tours.get(i).equals(newTour)){
        foundTour=(Tour)Tours.get(i);
        notfound=false;
      }
    }
    if(notfound){
      System.out.println("Model: Specified Tour not found!");
    }
    else{
      //Suche newDuty in ungeplanten Duties
      notfound=true;
      for(int i=0;i<Duties.size() && notfound;i++){
        if(((Duty)Duties.get(i)).equals(newDuty)){
          Duties.remove(i);
          notfound=false;
        }
      }
      if(foundTour.getAssignedDuty()!=null)
        Duties.add(foundTour.getAssignedDuty());
      foundTour.setAssignedDuty(newDuty);
      System.out.println("Model: Assigned Duty ["+newDuty+"] to Tour
["+newTour.getName()+"]");
    }
  }

  /**
   * Löscht eine gegebene Schicht aus einer Tour zu. Die Tour muss im
   * Model vorhanden sein. Die gelöschte Schicht wird zur Liste der ungeplanten
   * Schichten hinzugefügt.
   */
  public void unassignDutyFromTour(Duty newDuty, Tour newTour){
    //Nachsehen, ob newTour im Model ist
    boolean notfound=true;
    Tour foundTour=null;
    for(int i=0;i<Tours.size() && notfound;i++){
      if(Tours.get(i).equals(newTour)){
        notfound=false;
        foundTour=(Tour)Tours.get(i);
      }
    }
    if(notfound){
      System.out.println("Model: Specified Tour not found!");
    }
    else{
      //Suche newDuty in ungeplanten Duties
      if(foundTour.getAssignedDuty().equals(newDuty)){
        foundTour.setAssignedDuty(null);
        //addDuty(newDuty);
      }
    }
  }

  /**
   * Löscht einen gegebenen geplanten Besuch aus einer Tour zu. Die Tour muss im
   * Model vorhanden sein. Der geplante Besuch wird zur Liste der ungeplanten
   * Besuche hinzugefügt.
   */
  public void removeVisitFromTour(PlanedItem delVisit,Tour delTour){
    //Nachsehen, ob newTour im Model ist
    boolean notfound=true;
    Tour foundTour=new Tour();
    for(int i=0;i<Tours.size() && notfound;i++){
      if(Tours.get(i).equals(delTour)){
        notfound=false;
        foundTour=(Tour)Tours.get(i);
```

```
      }
    }
    if(notfound){
      System.out.println("Model: Specified Tour not found!");
    }
    else{
      foundTour.removePlanedItem(delVisit);
    }
  }

  /**
   * Liefert alle für einen gegebenen CareClient geplanten Besuche an
   * diesem Tag in Form eines Vectors.
   * Der Vector besteht aus PlanedVisitData-Objekten.
   */
  public Vector getPlanedVisitsWithClient(PlanableCareClient client){
    Vector tourVisits;
    Vector sortedVisits=new Vector();
    PlanedItem vglVisit;
    boolean inserted;
    for(int i=0;i<Tours.size();i++){
      tourVisits=((Tour)Tours.get(i)).getVisits();
      for(int j=0;j<tourVisits.size();j++){
        vglVisit=(PlanedItem)tourVisits.get(j);
        if(vglVisit.getPlanedVisit().getCareClient().equals(client)){
          //Geplanten Besuch in Vector nach Startzeit einsortieren
          inserted=false;
          for(int k=0;k<sortedVisits.size();k++){

  if(((PlanedItem)sortedVisits.get(k)).getPlanedTime().isGreater(vglVisit.getPlaned
Time())){
              sortedVisits.add(k,new
PlanedVisitData((Tour)Tours.get(i),vglVisit));
              inserted=true;
              k=sortedVisits.size();
            }
          }
          if(!inserted){
            sortedVisits.add(new PlanedVisitData((Tour)Tours.get(i),vglVisit));
          }
        }
      }
    }
    return sortedVisits;
  }

  /**
   * Liefert alle für einen gegebenen CareClient ungeplanten Besuche an
   * diesem Tag in Form eines Vectors.
   * Der Vector besteht aus Visit-Objekten.
   */
  public Vector getUnplanedVisitsWithClient(PlanableCareClient client){
    Vector sortedVisits=new Vector();
    Visit vglVisit;
    boolean inserted;
    Time vglTime;
    for(int j=0;j<Visits.size();j++){
      vglVisit=(Visit)Visits.get(j);
      if(vglVisit.getCareClient().equals(client)){
        //Geplanten Besuch in Vector nach Startzeit einsortieren
        inserted=false;
        if(vglVisit.getPreferedTime()!=null)
          vglTime=vglVisit.getPreferedTime();
        else
          vglTime=vglVisit.getStartTime();
        for(int k=0;k<sortedVisits.size();k++){
          if(((Visit)sortedVisits.get(k)).getPreferedTime()!=null){
            if(((Visit)sortedVisits.get(k)).getPreferedTime().isGreater(vglTime)){
```

```
                sortedVisits.add(k,vglVisit);
                inserted=true;
                k=sortedVisits.size();
              }
            }
            else {
              if(((Visit)sortedVisits.get(k)).getStartTime().isGreater(vglTime)){
                sortedVisits.add(k,vglVisit);
                inserted=true;
                k=sortedVisits.size();
              }
            }
          }
          if(!inserted){
            sortedVisits.add(vglVisit);
          }
        }
      }
      return sortedVisits;
    }

    /**
     * Liefert einen Vector gefüllt mit allen CareClients.
     * Der Vector besteht aus PlanableCareClients.
     */
    public Vector getAllCareClients(){
      Vector allCC=new Vector();
      PlanableCareClient vglCC;
      int insertPos;
      boolean vorhanden;
      for(int j=0;j<Visits.size();j++){
        vglCC=((Visit)Visits.get(j)).getCareClient();
        insertPos=-1;
        vorhanden=false;
        for(int k=0;k<allCC.size() && !vorhanden;k++){
          if(((PlanableCareClient)allCC.get(k)).equals(vglCC))
            vorhanden=true;
          else {
            if(insertPos<0){

    if(vglCC.toString().compareTo(((PlanableCareClient)allCC.get(k)).toString())<0){
                insertPos=k;
              }
            }
          }
        }
        if(!vorhanden){
          if(insertPos>=0)
            allCC.add(insertPos,vglCC);
          else
            allCC.add(vglCC);
        }
      }

      Vector tourVisits;
      for(int i=0;i<Tours.size();i++){
        tourVisits=((Tour)Tours.get(i)).getVisits();
        for(int j=0;j<tourVisits.size();j++){

    vglCC=(PlanableCareClient)((PlanedItem)tourVisits.get(j)).getPlanedVisit().getCar
eClient();
          insertPos=-1;
          vorhanden=false;
          for(int k=0;k<allCC.size() && !vorhanden;k++){
            if(((PlanableCareClient)allCC.get(k)).equals(vglCC))
              vorhanden=true;
            else {
              if(insertPos<0){
```

```
  if(vglCC.toString().compareTo(((PlanableCareClient)allCC.get(k)).toString())<0){
                insertPos=k;
              }
            }
          }
        }
        if(!vorhanden){
          if(insertPos>=0)
            allCC.add(insertPos,vglCC);
          else
            allCC.add(vglCC);
        }
      }
    }
    return allCC;
  }

}
```

9.3.9. Klasse „CarePlanModelChangeEvent“

```
import java.util.EventObject;
/**
 * Definiert die Klasse die Daten über die Änderung an
 * einem CarePlanModel enthält. Alle Methoden und
 * Felder werden bisher vom Programm ignoriert,
 * Da bei Änderung immer das ganze Model gespeichert,
 * bzw. auf dem Bildschirm aktualisiert wird.
 */
public class CarePlanModelChangeEvent extends EventObject {

    public static int ADD=0;
    public static int EDIT=1;
    public static int REMOVE=2;

    private Object item;
    private int changeType;

    public CarePlanModelChangeEvent(Object source,Object item,int changeType) {
        super(source);
        this.item=item;
        this.changeType=changeType;
    }

    public int getChangeType(){
    return changeType;
    }

    public Object getItem(){
    return item;
    }
}
```

9.3.10. Klasse „CarePlanModelChangeInformant“

```
import javax.swing.event.*;

/** Übernimmt die Aufgabe des CarePlanModel
 * Bei Änderung alle beobachter zu informieren.
 * Das CarePlanModel ruft hierzu die Methode fireCarePlanModelChanged()
 * auf.
 */
public class CarePlanModelChangeInformant{
```

```
  protected Object changeSource;

  public CarePlanModelChangeInformant(Object i){
    changeSource=i;
  }

  protected EventListenerList listenerList = new EventListenerList();

  protected ChangeListener changeListener = null;

  public void addCarePlanModelChangeListener(CarePlanModelChangeListener l) {
      listenerList.add(CarePlanModelChangeListener.class, l);
  }

  public void removeCarePlanModelChangeListener(CarePlanModelChangeListener l) {
      listenerList.remove(CarePlanModelChangeListener.class, l);
  }

  public void fireCarePlanModelChanged(Object item,int changeType) {
      Object[] listeners = listenerList.getListenerList();
      for (int i = listeners.length-2; i>=0; i-=2) {
          if (listeners[i]==CarePlanModelChangeListener.class) {
              CarePlanModelChangeEvent changeEvent = new
CarePlanModelChangeEvent(changeSource,item,changeType);

((CarePlanModelChangeListener)listeners[i+1]).carePlanModelChanged(changeEvent);
          }
      }
  }

}
```

9.3.11. Interface „CarePlanModelListener“

```
import java.util.*;

/**
 * Definiert das Interface weclhes Beobachter des CarePlanModel
 * implementieren müssen um von diesem über Änderungen informiert zu werden.
 */
public interface CarePlanModelChangeListener extends EventListener{
  public void carePlanModelChanged(CarePlanModelChangeEvent e);
}
```

9.3.12. Klasse „CarePlanModelDatabaseConnection“

```
import java.util.*;
import java.sql.*;
import java.text.*;

/**
 * Diese Klasse stellt die Verbindung zwischen CarePlanModel und
 * Datenbank her. Sie fungiert praktisch wie eine View des CarePlanModels.
 */
public class CarePlanModelDatabaseConnection implements
CarePlanModelChangeListener{

  private static final String dbURL="CarePlanModel";
  private static DateFormat
dfUS=DateFormat.getDateInstance(DateFormat.SHORT,Locale.US);

  private CarePlanModel usedModel;
  private JDBCEngine usedDBEngine;
```

```
  private java.util.Date fillDate=null;

  public CarePlanModelDatabaseConnection(CarePlanModel usedModel){
    usedDBEngine=new JDBCEngine(dbURL);
    this.usedModel=usedModel;
    usedModel.addCarePlanModelChangeListener(this);
  }

  public void setCarePlanModel(CarePlanModel usedModel){
    if(this.usedModel!=null)
      usedModel.removeCarePlanModelChangeListener(this);
    this.usedModel=usedModel;
  }

  public CarePlanModel getCarePlanModel(){
    return this.usedModel;
  }

  /**
   * Hier wird das CarPlanModel mit allen Daten zu einem gegebenen
   * Datum gefüllt.
   *
   */
  public void fillCarePlanModel(java.util.Date fillDate){
    this.fillDate=fillDate;
    System.out.println("Loading Database Content ...");
    usedModel.clearModel();
    usedDBEngine.connect();

    //Zunächst die ungeplanten Schichten aus der DB holen
    String fillDateString=dfUS.format(fillDate);

    String sqlQueryString="";
    sqlQueryString+="SELECT a.ID AS DutyID, a.DutyDate AS DutyDate, a.StartTime AS
DutyStartTime, a.EndTime AS DutyEndTime, c.ID AS EmployeeID, c.Name AS
EmployeeName, c.Qualification AS EmployeeQualification ";
    sqlQueryString+="FROM Duties AS a, Employees AS c ";
    sqlQueryString+="WHERE a.EmployeesID=c.ID and a.DutyDate=#"+fillDateString+"#
and (a.ID Not In ( ";
      sqlQueryString+="select d.id from duties as d,tours as e ";
      sqlQueryString+="WHERE e.DutiesID=d.ID and e.PlanedDate=#"+fillDateString+"#
and d.DutyDate=#"+fillDateString+"# ";
    sqlQueryString+=") ";
   sqlQueryString+="); ";

    ResultSet myResult=usedDBEngine.getSQLResult(sqlQueryString);
    try{
      System.out.println("Loading unassigned Duties: ");
      Employee newEmpl;
      Duty newDuty;
      while(myResult.next()){
        newEmpl=new
Employee(myResult.getString("EmployeeName"),myResult.getInt("EmployeeQualification"
),new Long(myResult.getLong("EmployeeID")));
        newDuty=new Duty(newEmpl,new Time(myResult.getTime("DutyStartTime")),new
Time(myResult.getTime("DutyEndTime")),new Long(myResult.getLong("DutyID")));
        usedModel.addDuty(newDuty);
      }
      System.out.println(" done!");
    } catch (Exception e){
    }

    //nun alle ungeplanden Besuche hinzufügen

    //dazu zunächst alle CareClients durchlaufen
    sqlQueryString="";
    sqlQueryString+="SELECT a.ID AS CareClientsID FROM CareClients AS a;";
    myResult=usedDBEngine.getSQLResult(sqlQueryString);
```

```
    try{
      System.out.println("Loading unplaned Visits: ");
      ResultSet validityResult;
      String sqlValidityQuery;
      Long serviceProfileID=null;
      java.util.Date serviceProfileValidityStart=null;
      long ccID;
      while(myResult.next()){
        ccID=myResult.getLong("CareClientsID");
        //Von jedem CareClient eine Gültigkeit eines ServiceProfils holen
        sqlValidityQuery="";
        sqlValidityQuery+="SELECT b.ID AS ServiceProfileID, b.ValidityStart AS
ServiceProfileValidityStart, b.PeriodSize as ServiceProfilePeriodSize ";
        sqlValidityQuery+="FROM ServiceProfiles AS b ";
        sqlValidityQuery+="WHERE b.CareClientsID="+ccID+" and b.ValidityStart IN (
";
        sqlValidityQuery+="SELECT max(a.ValidityStart) ";
        sqlValidityQuery+="FROM ServiceProfiles AS a ";
        sqlValidityQuery+="WHERE a.ValidityStart<=#"+fillDateString+"# And
a.CareClientsID="+ccID+" ";
        sqlValidityQuery+="); ";
        validityResult=usedDBEngine.getSQLResult(sqlValidityQuery);
        if(validityResult.next()){
          //Wenn gültiges Profil vorhanden, dann Tag aus Profil berechnen
          serviceProfileID=new Long(validityResult.getLong("ServiceProfileID"));

  serviceProfileValidityStart=validityResult.getDate("ServiceProfileValidityStart")
;

          long dayDiff=0;
          dayDiff=(long)((long)(fillDate.getTime()-
serviceProfileValidityStart.getTime())/(long)(1000*60*60*24));
          dayDiff= dayDiff%validityResult.getLong("ServiceProfilePeriodSize");

          //Jetzt alle Besuche für diesen Tag in diesem Profil holen
          ResultSet upVisitResult;

          String sqlUPVisitsString="";
          sqlUPVisitsString+="SELECT d.ID AS VisitsID, d.Length AS VisitsLength,
d.StartTime AS VisitsStartTime, d.PreferredTime AS VisitsPreferredTime, d.EndTime
AS VisitsEndTime, d.Qualification AS VisitsQualification, f.ID AS CareClientsID,
f.Name AS CareClientsName ";
          sqlUPVisitsString+="FROM visits AS d, ServiceProfiles AS e, careclients
AS f ";
          sqlUPVisitsString+="WHERE (((d.Day)="+dayDiff+") AND
d.ServiceProfilesID=e.ID and e.ID="+serviceProfileID+" and e.careclientsID=f.ID and
((d.ID) Not In (SELECT a.ID ";
          sqlUPVisitsString+="FROM visits AS a, planedVisits AS b, tours AS c ";
          sqlUPVisitsString+="WHERE b.toursID=c.ID And b.VisitsID=a.ID And
c.PlanedDate=#"+fillDateString+"# ";
          sqlUPVisitsString+="))); ";
          upVisitResult=usedDBEngine.getSQLResult(sqlUPVisitsString);
          CareClient newCC;
          java.sql.Time vprefTime;
          Time vPref;
          Visit newVisit;
          while(upVisitResult.next()){
            //Hier ungeplanten Besuch anlegen!
            //Im folgenden die Feldnamen der Abfrage:

  //VisitsID,VisitsLength,VisitsStartTime,VisitsPreferredTime,VisitsEndTime,VisitsQ
ualification
            //CareClientsID,CareClientsName
            newCC=new CareClient(upVisitResult.getString("CareClientsName"),new
Long(upVisitResult.getLong("CareClientsID")));
            vprefTime=upVisitResult.getTime("VisitsPreferredTime");
            if(vprefTime!=null)
              vPref=new Time(vprefTime);
```

```
              else
                vPref=null;
              newVisit=new Visit(newCC,new
Time(upVisitResult.getTime("VisitsLength")),(int)upVisitResult.getLong("VisitsQuali
fication"),new Time(upVisitResult.getTime("VisitsStartTime")),new
Time(upVisitResult.getTime("VisitsEndTime")),vPref,new
Long(upVisitResult.getLong("VisitsID")));
              usedModel.addVisit(newVisit);
            }
          }
        }
        System.out.println(" done!");

        //Jetzt alle Touren erstellen
        System.out.println("Loading unplaned Tours: ");
        String sqlTourString="";
        sqlTourString+="SELECT a.ID AS ToursID, a.DutiesID AS ToursDutiesID, a.Name
AS ToursName, a.StartTime AS ToursStartTime, a.EndTime AS ToursEndTime ";
        sqlTourString+="FROM tours AS a ";
        sqlTourString+="WHERE a.PlanedDate=#"+fillDateString+"#; ";

        ResultSet tourResult=usedDBEngine.getSQLResult(sqlTourString);

        Tour newTour;
        Long toursID;
        Long toursDutiesID;
        String sqlSingleDutyString;
        ResultSet singleDutyResult;
        Duty newSingleDuty;
        Employee newSingleEmployee;
        String sqlPlanedVisitsString;
        ResultSet pVisitsResult;
        Visit newPlanedVisit;
        CareClient newPlanedCC;
        Time prefVTime;
        java.sql.Time sqlVTime;
        //Alle Touren durchlaufen
        while(tourResult.next()){
          newTour=new Tour();
          toursID=new Long(tourResult.getLong("ToursID"));
          toursDutiesID=new Long(tourResult.getLong("ToursDutiesID"));
          if(toursDutiesID.longValue()==0)
            toursDutiesID=null;
          newTour.setStartTime(new Time(tourResult.getTime("ToursStartTime")));
          newTour.setEndTime(new Time(tourResult.getTime("ToursEndTime")));
          newTour.setName(tourResult.getString("ToursName"));
          if(toursDutiesID!=null){
            sqlSingleDutyString="";
            sqlSingleDutyString+="SELECT a.ID AS DutyID, a.DutyDate AS DutyDate,
a.StartTime AS DutyStartTime, a.EndTime AS DutyEndTime, c.ID AS EmployeeID, c.Name
AS EmployeeName, c.Qualification AS EmployeeQualification ";
            sqlSingleDutyString+="FROM Duties AS a, Employees AS c ";
            sqlSingleDutyString+="WHERE a.ID="+toursDutiesID+" and
a.EmployeesID=c.ID; ";

            singleDutyResult=usedDBEngine.getSQLResult(sqlSingleDutyString);
            if(singleDutyResult.next()){
              newSingleEmployee=new
Employee(singleDutyResult.getString("EmployeeName"),singleDutyResult.getInt("Employ
eeQualification"),new Long(singleDutyResult.getLong("EmployeeID")));
              newSingleDuty=new Duty(newSingleEmployee,new
Time(singleDutyResult.getTime("DutyStartTime")),new
Time(singleDutyResult.getTime("DutyEndTime")),new
Long(singleDutyResult.getLong("DutyID")));
              newTour.setAssignedDuty(newSingleDuty);
            }
          }
```

```
          //Jetzt zu jeder Tour die geplanten Besuche holen
          sqlPlanedVisitsString="";
          sqlPlanedVisitsString+="SELECT d.ID AS VisitsID, d.Length AS VisitsLength,
d.StartTime AS VisitsStartTime, d.PreferredTime AS VisitsPreferredTime, d.EndTime
AS VisitsEndTime, d.Qualification AS VisitsQualification, f.ID AS CareClientsID,
f.Name AS CareClientsName, a.PlanedTime AS PlanedVissitsPlanedTime ";
          sqlPlanedVisitsString+="FROM planedVisits AS a, visits AS d,
ServiceProfiles AS e, careclients AS f ";
          sqlPlanedVisitsString+="WHERE a.ToursID="+toursID+" and a.VisitsID=d.ID and
d.ServiceProfilesID=e.ID and e.CareClientsID=f.ID; ";

          pVisitsResult=usedDBEngine.getSQLResult(sqlPlanedVisitsString);
          while(pVisitsResult.next()){
            newPlanedCC=new
CareClient(pVisitsResult.getString("CareClientsName"),new
Long(pVisitsResult.getLong("CareClientsID")));
            sqlVTime=pVisitsResult.getTime("VisitsPreferredTime");
            if(sqlVTime!=null)
              prefVTime=new Time(sqlVTime);
            else
              prefVTime=null;
            newPlanedVisit=new Visit(newPlanedCC,new
Time(pVisitsResult.getTime("VisitsLength")),pVisitsResult.getInt("VisitsQualificati
on"),new Time(pVisitsResult.getTime("VisitsStartTime")),new
Time(pVisitsResult.getTime("VisitsEndTime")),prefVTime,new
Long(pVisitsResult.getLong("VisitsID")));

            newTour.addVisit(newPlanedVisit,new
Time(pVisitsResult.getTime("PlanedVissitsPlanedTime")));
          }

          usedModel.addTour(newTour);
        }
        System.out.println(" done!");

      } catch (Exception e){
        System.out.println("Fehler!!!");
      }

      System.out.println("Database Content Loaded!");
      usedDBEngine.disconnect();
      usedModel.CarePlanModelChanged(null,1);
    }

    /**
     * In diesem Event werden alle Daten des CarePlanModels
     * zurück in die Datenbank geschrieben.
     *
     */
    public void carePlanModelChanged(CarePlanModelChangeEvent e){
      System.out.println("Save changes to database ...");
      if(fillDate!=null){
        String fillDateString=dfUS.format(fillDate);
        usedDBEngine.connect();
        //Zunächst alle Touren löschen.
        //Dazu müssen von jeder Tour erst einmal alle PlanedVisits gelöscht werden.
        //Erst danach werden die Touren gelöscht.
        //Jetzt alle Touren erstellen
        try{
          String sqlTourString="";
          sqlTourString+="SELECT a.ID AS ToursID ";
          sqlTourString+="FROM tours AS a ";
          sqlTourString+="WHERE a.PlanedDate=#"+fillDateString+"#; ";

          ResultSet tourResult=usedDBEngine.getSQLResult(sqlTourString);

          Long toursID;
```

```
          String sqlDeletePlanedVisits;
          String sqlDeleteTours;
          //Alle Touren durchlaufen und geplante Besuche löschen
          while(tourResult.next()){
            toursID=new Long(tourResult.getLong("ToursID"));
            sqlDeletePlanedVisits="";
            sqlDeletePlanedVisits+="DELETE * ";
            sqlDeletePlanedVisits+="FROM planedvisits AS a ";
            sqlDeletePlanedVisits+="WHERE a.ToursID="+toursID+"; ";
            usedDBEngine.SQLUpdate(sqlDeletePlanedVisits);
          }
          //Jetzt alle Touren löschen
          sqlDeleteTours="";
          sqlDeleteTours+="DELETE * ";
          sqlDeleteTours+="FROM tours AS a  ";
          sqlDeleteTours+="WHERE a.PlanedDate=#"+fillDateString+"#;  ";
          usedDBEngine.SQLUpdate(sqlDeleteTours);

          //Nachdem alle alten Daten gelöscht wurden können jetzt neue erstellt
werden

          //Zunächst neue TourID holen.
          String sqlNewTourIDString="SELECT max([ID])+1 AS NewToursID FROM Tours;";
          ResultSet newTourIDResult;
          newTourIDResult=usedDBEngine.getSQLResult(sqlNewTourIDString);
          long newTID=1;
          if(newTourIDResult.next()){
            newTID=newTourIDResult.getLong("NewToursID");
          }
          if(newTID<=0)
            newTID=1;

          //Jetzt neue ID für PlanedVisits holen.
          String sqlNewPVisitsIDString="SELECT max([ID])+1 AS NewPlanedVisitsID FROM
PlanedVisits;";
          ResultSet newPVisitsIDResult;
          newPVisitsIDResult=usedDBEngine.getSQLResult(sqlNewPVisitsIDString);
          long newPVID=1;
          if(newPVisitsIDResult.next()){
            newPVID=newPVisitsIDResult.getLong("NewPlanedVisitsID");
          }
          if(newPVID<=0)
            newPVID=1;

          Vector tourVector=usedModel.getTours();
          Vector planedVisits;
          Tour insertTour;
          String sqlInsertNewTourString;
          PlanedItem insertItem;
          String sqlInsertNewPVisitString;
          for(int i=0;i<tourVector.size();i++){
            insertTour=(Tour)tourVector.get(i);
            sqlInsertNewTourString="";
            sqlInsertNewTourString+="INSERT INTO Tours ";
            sqlInsertNewTourString+="(ID,DutiesID,Name,StartTime,EndTime,PlanedDate)
";
            if(insertTour.getAssignedDuty()!=null){
              sqlInsertNewTourString+="VALUES
("+(newTID)+","+insertTour.getAssignedDuty().getID()+",'"+insertTour.getName()+"','
"+insertTour.getStartTime().toString()+"','"+insertTour.getEndTime().toString()+"',
#"+fillDateString+"#); ";
            }
            else {
              sqlInsertNewTourString+="VALUES
("+(newTID)+",0,'"+insertTour.getName()+"','"+insertTour.getStartTime().toString()+
"','"+insertTour.getEndTime().toString()+"',#"+fillDateString+"#); ";
            }
            usedDBEngine.SQLUpdate(sqlInsertNewTourString);
```

```
            //für diese Tour jetzt nach alle Planed Visits anlegen
            planedVisits=insertTour.getVisits();
            for(int j=0;j<planedVisits.size();j++){
              insertItem=(PlanedItem)planedVisits.get(j);
              sqlInsertNewPVisitString="";
              sqlInsertNewPVisitString+="INSERT INTO PlanedVisits ";
              sqlInsertNewPVisitString+="(ID,ToursID,VisitsID,PlanedTime) ";
              sqlInsertNewPVisitString+="Values
('"+(newPVID++)+"',"+newTID+","+insertItem.getPlanedVisit().getID()+",'"+insertItem
.getPlanedTime().toString()+"'); ";
              usedDBEngine.SQLUpdate(sqlInsertNewPVisitString);
            }
            newTID++;
          }

        } catch (Exception except){
          System.out.println("Fehler!");
        }
        usedDBEngine.disconnect();
      }
      System.out.println("Changes Saved!");
    }

}
```

9.3.13. Start-Klasse „CarePlanModelStarter“

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;


/** Dies ist die Startklasse dieses Prototypen.
 * Sie sollte man sich genauer ansehen, wenn man den Prototypen
 * in ein bestehendes Programm integrieren möchte.
 */
public class CarePlanModuleStarter implements ActionListener{

  private CarePlanModel myModel;
  private JButton changeButton;
  private NewTourPanelGUI newTourGUI;
  private JDialog newTourDialog;
  private JFrame myFrame;

  public CarePlanModuleStarter(){
    System.out.println("Initialize CarePlanModule ...");
    //Zunächst ein CarePlanModel erzeugen
    myModel=new CarePlanModel();
    //Dann die Datenbank an das Model anbinden
    CarePlanModelDatabaseConnection myCarePlanDBConnection=new
CarePlanModelDatabaseConnection(myModel);

    //jetzt die einzelnen Views erzeugen, an das Model binden und in Fenstern
    //darstellen. Später vielleicht auf Panels innerhalb eines Programmes.
    CarePlanTourView myTourView=new CarePlanTourView();
    myTourView.setCarePlanModel(myModel);

    CarePlanClientView myClientView=new CarePlanClientView();
    myClientView.setCarePlanModel(myModel);

    //Nur zum Test
    myModel.CarePlanModelChanged(null,1);

    myFrame=new JFrame("TourView: SimpleTourViewFactory");
    myFrame.getContentPane().setLayout(new BorderLayout());
```

```
    myFrame.getContentPane().add(myTourView.getComponent(),BorderLayout.CENTER);
    //Über diesen Button kann man Touren Manuell erzeugen.
    changeButton=new JButton("Create new Tour");
    changeButton.addActionListener(this);
    myFrame.getContentPane().add(changeButton,BorderLayout.SOUTH);
    myFrame.setSize(620,500);
    myFrame.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
           System.exit(0);
         }
       }
    );
    myFrame.show();

    myFrame=new JFrame("ClientView: SimpleClientViewFactory");
    myFrame.getContentPane().setLayout(new BorderLayout());
    myFrame.getContentPane().add(myClientView.getComponent(),BorderLayout.CENTER);

    myFrame.setSize(620,500);
    myFrame.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
           System.exit(0);
         }
       }
    );
    myFrame.show();

    //Zum Schluss noch die Datumsauswahl an die Datenbankanbindung koppeln.
    DateChooser myDate=new DateChooser(myCarePlanDBConnection);

  }

  public void actionPerformed(ActionEvent e){
    if(e.getSource()==changeButton){
      //"Neue Tour"-Dialog öffnen.
      newTourDialog=new JDialog(myFrame,"Create new Tour",true);
      newTourDialog.getContentPane().setLayout(new BorderLayout());
      newTourGUI=new NewTourPanelGUI();
      newTourDialog.getContentPane().add(newTourGUI,BorderLayout.CENTER);
      JButton OKButton=new JButton("Create");
      newTourDialog.getContentPane().add(OKButton,BorderLayout.SOUTH);
      OKButton.addActionListener(this);
      newTourDialog.setSize(200,150);
      newTourDialog.show();
      myModel.CarePlanModelChanged(null,1);
    }
    else {
      //Wenn Tour erzeugt wurde, dann einfach dem Model hinzufügen,
      //und alle Beobachter infromieren.
      Tour newTour=new Tour();

      newTour.setName(newTourGUI.getTourName());
      newTour.setStartTime(newTourGUI.getTourStart());
      newTour.setEndTime(newTourGUI.getTourEnd());

      newTourDialog.dispose();

      myModel.addTour(newTour);
      myModel.CarePlanModelChanged(null,1);

    }
  }


  public static void main(String args[])
  {
    System.out.println("Starte CarePlanModule ...");
```

```
    CarePlanModuleStarter myCarePlanModuleStarter=new CarePlanModuleStarter();

    System.out.println("CarePlanModule gestartet!");

  }

}
```

9.3.14. Klasse „CarePlanTourView“

```
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.dnd.*;
import java.text.*;

/**Stellt ein Beispiel dar, wie eine Planung aus Tourensicht vollzogen
 * werden könnte. Wird über die Methode des CarePlanModelChangeListener
 * über Änderungen informiert und aktualisiert sich darüber ständig.
 *
 */
public class CarePlanTourView implements CarePlanModelChangeListener{


  private CarePlanModel UsedModel;
  private JPanel tourViewPanel;

  private Time dayStart=new Time(6,0,0); //6 Uhr
  private Time dayEnd=new Time(22,0,0); //14 Uhr
  private Time dayRaster=new Time(0,30,0); //30 Minuten

  private TourViewFactory UsedTourViewFactory;

  private UnplanedVisitsDragAndDrop unplanedVisitsDnD=new
UnplanedVisitsDragAndDrop();
  private UnassignedDutiesDragAndDrop unassignedDutiesDnD=new
UnassignedDutiesDragAndDrop();
  private PlanedVisitsDragAndDrop planedVisitsDnD=new PlanedVisitsDragAndDrop();
  private AssignedDutiesDragAndDrop assignedDutiesDnD=new
AssignedDutiesDragAndDrop();
  private NotDropableDropListener notDropableListener=new
NotDropableDropListener();
  private JScrollPane MainScrollPane=null;

  //Drag and Drop
  DragSource[] upVisitsDragSource;
  DragSource[] plVisitsDragSource;
  DragSource[] uaDutiesDragSource;
  DragSource[] asDutiesDragSource;
  DropTarget[] pVisitsDropTarget;
  DropTarget[] pToursDropTarget;

  private Vector getAllComponentsOfContainer(Container container){
    Container vglClass=new Container();
    Vector returnVector = new Vector();
    Vector subVector = new Vector();
    Component[] AlleKomps=container.getComponents();
    for(int i=0;i<AlleKomps.length;i++){
      returnVector.add(AlleKomps[i]);
      if(vglClass.getClass().isAssignableFrom(AlleKomps[i].getClass())){
         subVector=getAllComponentsOfContainer((Container)AlleKomps[i]);
         for(int z=0;z<subVector.size();z++){
          returnVector.add(subVector.get(z));
         }
      }
    }
```

```
    return returnVector;
  }

  public CarePlanTourView(){
    UsedTourViewFactory=new SimpleTourViewFactory();
    unplanedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
    unassignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
    planedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
    assignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
    //createComponent();
    UsedModel=null;
    //refreshDisplay();
  }

  public CarePlanTourView(TourViewFactory usedFactory){
    if(usedFactory!=null){
      UsedTourViewFactory=usedFactory;
      unplanedVisitsDnD.setTourViewFactory(usedFactory);
      unassignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
      planedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
      assignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
    }
    else{
      UsedTourViewFactory=new SimpleTourViewFactory();
      unplanedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
      unassignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
      planedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
      assignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
    }
    createComponent();
    UsedModel=null;
    refreshDisplay();
  }

  public CarePlanTourView(CarePlanModel UsedModel){
    UsedTourViewFactory=new SimpleTourViewFactory();
    unplanedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
    unassignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
    planedVisitsDnD.setTourViewFactory(UsedTourViewFactory);
    assignedDutiesDnD.setTourViewFactory(UsedTourViewFactory);
    createComponent();
    this.UsedModel=UsedModel;
    UsedModel.addCarePlanModelChangeListener(this);
    unplanedVisitsDnD.setCarePlanModel(UsedModel);
    unassignedDutiesDnD.setCarePlanModel(UsedModel);
    assignedDutiesDnD.setCarePlanModel(UsedModel);
    planedVisitsDnD.setCarePlanModel(UsedModel);
    refreshDisplay();
  }

  private void createComponent(){
    //create a new Panel with a scrollpane on it
    Point scrollPoint=new Point(0,0);
    if(tourViewPanel==null){
      tourViewPanel=new JPanel(new BorderLayout());
    }
    else {
      scrollPoint=MainScrollPane.getViewport().getViewPosition();
      tourViewPanel.removeAll();
    }
    JScrollPane newScrollPane=new JScrollPane();
    tourViewPanel.add(newScrollPane,BorderLayout.CENTER);
    MainScrollPane=newScrollPane;
    //Create VisitsPanel
    JPanel leftPanel=new JPanel();
    leftPanel.setLayout(new BorderLayout());
    JScrollPane visitsScrollPane=new JScrollPane();
    tourViewPanel.add(leftPanel,BorderLayout.WEST);
```

```
    leftPanel.add(visitsScrollPane,BorderLayout.CENTER);
    leftPanel.add(new JLabel("Unplaned Visits"),BorderLayout.NORTH);
    JPanel newAJPanel;
    if(UsedModel!=null)
      if(UsedModel.getUnplanedVisits()!=null)

  newAJPanel=UsedTourViewFactory.createVisitsPanel(UsedModel.getUnplanedVisits());
      else
        newAJPanel=UsedTourViewFactory.createVisitsPanel(new Vector());
    else
      newAJPanel=UsedTourViewFactory.createVisitsPanel(new Vector());
    visitsScrollPane.getViewport().add(newAJPanel);
    // *** Drag and Drop
    if(UsedModel!=null){

      Vector dragSources = getAllComponentsOfContainer(newAJPanel);
      upVisitsDragSource=new DragSource[dragSources.size()];

      for(int i=1; i<dragSources.size();i++){
        upVisitsDragSource[i]=DragSource.getDefaultDragSource();

  upVisitsDragSource[i].createDefaultDragGestureRecognizer((Component)dragSources.g
et(i), DnDConstants.ACTION_COPY_OR_MOVE, unplanedVisitsDnD);
      }
    }
    // ***

    //Create DutyPanel
    JPanel rightPanel=new JPanel();
    rightPanel.setLayout(new BorderLayout());
    JScrollPane dutyScrollPane=new JScrollPane();
    rightPanel.add(new JLabel("Staff"),BorderLayout.NORTH);
    rightPanel.add(dutyScrollPane,BorderLayout.CENTER);
    tourViewPanel.add(rightPanel,BorderLayout.EAST);
    JPanel newDutyPanel;
    if(UsedModel!=null)
      if(UsedModel.getUnassignedDuties()!=null)

  newDutyPanel=UsedTourViewFactory.createDutiesPanel(UsedModel.getUnassignedDuties(
));
      else
        newDutyPanel=UsedTourViewFactory.createDutiesPanel(new Vector());
    else
      newDutyPanel=UsedTourViewFactory.createDutiesPanel(new Vector());
    dutyScrollPane.getViewport().add(newDutyPanel);
    // *** Drag and Drop
    if(UsedModel!=null){

      Vector uaddragSources = getAllComponentsOfContainer(newDutyPanel);

      uaDutiesDragSource=new DragSource[uaddragSources.size()];

      for(int i=1; i<uaddragSources.size();i++){
        uaDutiesDragSource[i]=DragSource.getDefaultDragSource();

  uaDutiesDragSource[i].createDefaultDragGestureRecognizer((Component)uaddragSource
s.get(i), DnDConstants.ACTION_COPY_OR_MOVE, unassignedDutiesDnD);
      }
    }
    // ***


    //place a panel (Flowlayout) on the viewport of the scrollpane
    JPanel ViewportPanel=new AutoScrollJPanel();

    ViewportPanel.setAutoscrolls(true);
    DropTarget nedt=new DropTarget(ViewportPanel,notDropableListener);
```

```
    newScrollPane.getViewport().add(ViewportPanel);
    ((FlowLayout)ViewportPanel.getLayout()).setVgap(1);
    //((FlowLayout)ViewportPanel.getLayout()).setAlignment(FlowLayout.LEFT);

    //Als ScrollPanel Kopfzeile die Tour-Header
    JPanel HeaderPanel=new JPanel();
    newScrollPane.setColumnHeader(new JViewport());
    newScrollPane.getColumnHeader().add(HeaderPanel);

    //Auf der Linken Seite eine Zeitleiste
    newScrollPane.setRowHeader(new JViewport());
    newScrollPane.getRowHeader().add(UsedTourViewFactory.createVerticalTimeBar(new
Time(dayStart),new Time(dayEnd)));

    //create all Tours
    JPanel tstPanel;
    JPanel tourPanel;
    JPanel noTourPanel;
    if(UsedModel!=null){
      Vector ToursVector=UsedModel.getTours();
      if(ToursVector!=null){
        for(int i=0;i<ToursVector.size();i++){

          JPanel
singleHeaderPanel=UsedTourViewFactory.createHeaderPanelWithTour((Tour)ToursVector.g
et(i));
          // *** Drag and Drop Header
          Vector pddropTargets = getAllComponentsOfContainer(singleHeaderPanel);

          pToursDropTarget=new DropTarget[pddropTargets.size()];
          asDutiesDragSource=new DragSource[pddropTargets.size()];
          for(int j=0; j<pddropTargets.size();j++){
            asDutiesDragSource[j]=DragSource.getDefaultDragSource();

  asDutiesDragSource[j].createDefaultDragGestureRecognizer((Component)pddropTargets
.get(j), DnDConstants.ACTION_COPY_OR_MOVE, assignedDutiesDnD);


            pToursDropTarget[j]=new
DropTarget((Component)pddropTargets.get(j),unassignedDutiesDnD);
          }

          HeaderPanel.add(singleHeaderPanel);


  tstPanel=UsedTourViewFactory.createBodyPanelWithTour((Tour)ToursVector.get(i));
          nedt=new DropTarget(tstPanel,notDropableListener);

          // *** Drag and Drop Body
          Vector dropTargets = getAllComponentsOfContainer(tstPanel);

          pVisitsDropTarget=new DropTarget[dropTargets.size()];
          plVisitsDragSource=new DragSource[dropTargets.size()];
          for(int j=0; j<dropTargets.size();j++){

            pVisitsDropTarget[j]=new
DropTarget((Component)dropTargets.get(j),unplanedVisitsDnD);

            plVisitsDragSource[j]=DragSource.getDefaultDragSource();

  plVisitsDragSource[j].createDefaultDragGestureRecognizer((Component)dropTargets.g
et(j), DnDConstants.ACTION_COPY_OR_MOVE, planedVisitsDnD);
          }


          ViewportPanel.add(tstPanel);
        }
      }
```

```
    }
    MainScrollPane.getViewport().setViewPosition(scrollPoint);
  }

  public void setCarePlanModel(CarePlanModel UsedModel){
    this.UsedModel=UsedModel;
    unplanedVisitsDnD.setCarePlanModel(UsedModel);
    unassignedDutiesDnD.setCarePlanModel(UsedModel);
    planedVisitsDnD.setCarePlanModel(UsedModel);
    assignedDutiesDnD.setCarePlanModel(UsedModel);
    UsedModel.addCarePlanModelChangeListener(this);
    refreshDisplay();
  }

  public void refreshDisplay(){
    if(UsedModel!=null){
    }
    else {
      System.out.println("CarePlanTourView: No Model assigned!");
    }
    createComponent();
    tourViewPanel.updateUI();
  }

  //CarePlanModelChangeListener
  public void carePlanModelChanged(CarePlanModelChangeEvent e){
    refreshDisplay();
  }

  public JComponent getComponent(){
    //createComponent();
    return tourViewPanel;
  }


}
```

9.3.15. Abstrakte Klasse „ClientViewFactory“

```
import java.util.*;
import javax.swing.*;
import java.awt.*;

/**
 * Mit dieser abstrakten Klasse wird die Clientview erzeugt. Sie enthält auch Daten
 * um über Komponenten an Datenobjekte zu gelangen. Für informationen hierüber
 * sieht man sich am besten die Konkrete implementierung an.
 */
public abstract class ClientViewFactory{
  public ClientViewFactory(){
  }

  public JPanel createVerticalTimeBar(Time StartTime,Time EndTime){
    return new JPanel();
  }

  public JPanel createBodyPanelWithClientVisits(Vector unplanedVisits,Vector
planedVisits){
    return new JPanel();
  }

  public JPanel createDutiesPanelWithDutiesAndTours(Vector unplanedDuties,Vector
tours){
    return new JPanel();
  }
```

```
  public JPanel createHeaderPanelWithCareClient(PlanableCareClient careClient){
    return new JPanel();
  }

  public Tour getTourFromComponent(Component comp){
    return null;
  }

  public Duty getDutyFromComponent(Component comp){
    return null;
  }

  public Visit getVisitFromComponent(Component comp){
    return null;
  }

  public PlanedVisitData getPlanedVisitDataFromComponent(Component comp){
    return null;
  }

}
```

9.3.16. Klasse „ClientViewUnplanedDutiesDragAndDrop“

```
import java.awt.*;
import java.awt.event.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

/**
 * Diese Klasse bearbeitet alle Drag and Drop operationen innerhalb der
 * ClientView.
 */
public class ClientViewUnplanedDutiesDragAndDrop
  implements
DragGestureListener,DragSourceListener,DropTargetListener,ActionListener {

  private ClientViewFactory usedFactory;
  private CarePlanModel usedModel;

  private JDialog newTourDialog;
  private Component dropComponent;
  private NewTourPanelGUI newTourGUI;
  private Tour newTourForDuty;
  private Duty transferDuty;
  private Visit dropVisit;
  private PlanedVisitData dropVisitData;

  private ClientViewUnplanedDutiesDragAndDrop thisObject;

  final static private Class vglFrameClass=(new Frame()).getClass();

  private Frame getFrame(Component source){
    Component actComponent=source;
    boolean viewportNotFound=true;
    while(viewportNotFound){
      if(vglFrameClass.isAssignableFrom(actComponent.getClass()))
        return (Frame)actComponent;
      if(actComponent.getParent()!=null)
        actComponent=actComponent.getParent();
      else
        return null;
    }
    return null;
```

```
  }

  public ClientViewUnplanedDutiesDragAndDrop(){
    this.usedFactory=new SimpleClientViewFactory();
    thisObject=this;
  }

  public ClientViewUnplanedDutiesDragAndDrop(ClientViewFactory usedFactory){
    this.usedFactory=usedFactory;
    thisObject=this;
  }

  public void setClientViewFactory(ClientViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  public void setCarePlanModel(CarePlanModel usedModel){
    this.usedModel=usedModel;
  }

  //DragGestureListener
  public void dragGestureRecognized(DragGestureEvent dge){
    if(usedFactory.getTourFromComponent(dge.getComponent())!=null){
      dge.getDragSource().startDrag(dge, DragSource.DefaultMoveDrop, new
TransferablePlanedDutyData(new
PlanedDutyData(usedFactory.getTourFromComponent(dge.getComponent()).getAssignedDuty
(),usedFactory.getTourFromComponent(dge.getComponent()))), this);
    }
    else {
      if(usedFactory.getDutyFromComponent(dge.getComponent())!=null){
        dge.getDragSource().startDrag(dge, DragSource.DefaultMoveDrop, new
TransferableDuty(usedFactory.getDutyFromComponent(dge.getComponent())), this);
      }
    }
  }

  //DragSourceListener
  public void dragDropEnd(DragSourceDropEvent DragSourceDropEvent) {
  }

  public void dragEnter(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dragExit(DragSourceEvent DragSourceEvent) {
  }
  public void dragOver(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dropActionChanged(DragSourceDragEvent DragSourceDragEvent) {
  }


  private boolean isDragAcceptable(DropTargetDragEvent dropTargetDragEvent){
    return (dropTargetDragEvent.isDataFlavorSupported(TransferableDuty.DUTY_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedDutyData.PLANED_DUT
Y_DATA_FLAVOR));
  }
  // interface DropTargetListener
  public void dragEnter (DropTargetDragEvent dropTargetDragEvent) {
    if(isDragAcceptable(dropTargetDragEvent))
    dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void dragExit (DropTargetEvent dropTargetEvent) {
  }

  public void dragOver (DropTargetDragEvent dropTargetDragEvent) {
```

```
    if(isDragAcceptable(dropTargetDragEvent))
    dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void dropActionChanged (DropTargetDragEvent dropTargetDragEvent) {
    if(isDragAcceptable(dropTargetDragEvent))
    dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void drop (DropTargetDropEvent dropTargetDropEvent) {
    dropTargetDropEvent.acceptDrop (DnDConstants.ACTION_MOVE);
    boolean newTour=false;
    boolean newTourWithPlanedVisitData=false;
    try{
      if(dropTargetDropEvent.isDataFlavorSupported(TransferableDuty.DUTY_FLAVOR)){

  if(usedFactory.getVisitFromComponent(dropTargetDropEvent.getDropTargetContext().g
etDropTarget().getComponent())!=null){
          //DropTarget-Component festhalten, falls später ein Dialog aufgerufen
werden soll.
          //Dieser muß dann an den Frame geknüpft werden, in dem diese Componente
liegt.

  dropComponent=dropTargetDropEvent.getDropTargetContext().getDropTarget().getCompo
nent();
          //Daten von DropTarget und DragSource holen
          dropVisit=usedFactory.getVisitFromComponent(dropComponent);

  transferDuty=(Duty)dropTargetDropEvent.getTransferable().getTransferData(Transfer
ableDuty.DUTY_FLAVOR);

          //Hier muss zunächst eine neu Tour erstellt werden!
          newTour=true;//Weiter unten am Ende de Events erst die Tour erstellen!
        }
        else{

  if(usedFactory.getPlanedVisitDataFromComponent(dropTargetDropEvent.getDropTargetC
ontext().getDropTarget().getComponent())!=null){
            //DropTarget-Component festhalten, falls später ein Dialog aufgerufen
werden soll.
            //Dieser muß dann an den Frame geknüpft werden, in dem diese
Componente liegt.

  dropComponent=dropTargetDropEvent.getDropTargetContext().getDropTarget().getCompo
nent();
            //Daten von DropTarget und DragSource holen

  dropVisitData=usedFactory.getPlanedVisitDataFromComponent(dropTargetDropEvent.get
DropTargetContext().getDropTarget().getComponent());

  transferDuty=(Duty)dropTargetDropEvent.getTransferable().getTransferData(Transfer
ableDuty.DUTY_FLAVOR);

            //Falls die geplante Tour noch keiner Schicht zugeordnet ist, dann
diese
            //Schicht der Tour zuordnen. Ansonsten neue Tour erstellen
            if(dropVisitData.getTour().getAssignedDuty()==null){
              usedModel.assignDutyToTour(transferDuty,dropVisitData.getTour());
            }
            else
            //Hier muss zunächst eine neu Tour erstellt werden!
              newTourWithPlanedVisitData=true; //Weiter unten am Ende de Events
erst die Tour erstellen!
          }
```

```
        }

      }
      else {

  if(dropTargetDropEvent.isDataFlavorSupported(TransferablePlanedDutyData.PLANED_DU
TY_DATA_FLAVOR)){

  if(usedFactory.getVisitFromComponent(dropTargetDropEvent.getDropTargetContext().g
etDropTarget().getComponent())!=null){
            //Daten von DropTarget und DragSource holen

  dropVisit=usedFactory.getVisitFromComponent(dropTargetDropEvent.getDropTargetCont
ext().getDropTarget().getComponent());
            PlanedDutyData
transferDutyData=(PlanedDutyData)dropTargetDropEvent.getTransferable().getTransferD
ata(TransferablePlanedDutyData.PLANED_DUTY_DATA_FLAVOR);

            //Besuch der Tour zuweisen
            usedModel.assignVisitToTour(dropVisit,transferDutyData.getTour());

          }
          else {

  if(usedFactory.getPlanedVisitDataFromComponent(dropTargetDropEvent.getDropTargetC
ontext().getDropTarget().getComponent())!=null){
              //Daten von DropTarget und DragSource holen
              PlanedVisitData
dropVisitData=usedFactory.getPlanedVisitDataFromComponent(dropTargetDropEvent.getDr
opTargetContext().getDropTarget().getComponent());
              PlanedDutyData
transferDutyData=(PlanedDutyData)dropTargetDropEvent.getTransferable().getTransferD
ata(TransferablePlanedDutyData.PLANED_DUTY_DATA_FLAVOR);

              //Zunächst Besuch aus der alten Tour nehmen, dann der neuen Tour
zuweisen

  usedModel.removeVisitFromTour(dropVisitData.getPlanedItem(),dropVisitData.getTour
());

usedModel.assignVisitToTour(dropVisitData.getPlanedItem().getPlanedVisit(),dropVisi
tData.getPlanedItem().getPlanedTime(),transferDutyData.getTour());

            }
          }
        }
        else {
          System.out.println("Drop with unknown falvor!");
        }
      }

    } catch (Exception e){
    }

    dropTargetDropEvent.dropComplete(true);

    //Dieser umständliche Aufruf von 'usedModel.CarePlanModelChanged(null,1)' muß
sein
    //da sich sonst JAVA aufhängt!
    if(!newTour)
      if(!newTourWithPlanedVisitData)
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              //Änderungsbenachrichtigung!
              usedModel.CarePlanModelChanged(null,1);
            }
          }
        );
```

```
      else
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              int antw=JOptionPane.showConfirmDialog(null,"If you want to assign
this Duty to a Careclient\nyou must create a new Tour.\nCreate a new Tour now?",
"Unassigned Duty", JOptionPane.YES_NO_OPTION);
              if(antw==JOptionPane.YES_OPTION){

                newTourDialog=new JDialog(getFrame(dropComponent),"New Tour for
Duty",true);
                newTourDialog.getContentPane().setLayout(new BorderLayout());
                newTourGUI=new NewTourPanelGUI();

                //Start und Endzeit der neuen Tour mit Defaulteinstellungen
belegen
                Time endTime=new Time();

  if(dropVisitData.getPlanedItem().getPlanedVisit().getPreferedTime()!=null){
                  endTime=new
Time(dropVisitData.getPlanedItem().getPlanedVisit().getPreferedTime());

  if(dropVisitData.getPlanedItem().getPlanedVisit().getPreferedTime().isGreater(tra
nsferDuty.getStartTime()))
                    newTourGUI.setTourStart(transferDuty.getStartTime());
                  else

  newTourGUI.setTourStart(dropVisitData.getPlanedItem().getPlanedVisit().getPrefere
dTime());
                }
                else {
                  endTime=new
Time(dropVisitData.getPlanedItem().getPlanedVisit().getStartTime());

  if(dropVisitData.getPlanedItem().getPlanedVisit().getStartTime().isGreater(transf
erDuty.getStartTime()))
                    newTourGUI.setTourStart(transferDuty.getStartTime());
                  else

  newTourGUI.setTourStart(dropVisitData.getPlanedItem().getPlanedVisit().getStartTi
me());
                }

  endTime.addTime(dropVisitData.getPlanedItem().getPlanedVisit().getLength());
                if(endTime.isGreater(transferDuty.getEndTime()))
                  newTourGUI.setTourEnd(endTime);
                else
                  newTourGUI.setTourEnd(transferDuty.getEndTime());
                //Defaulteinstellung erledigt


  newTourDialog.getContentPane().add(newTourGUI,BorderLayout.CENTER);
                JButton OKButton=new JButton("Create");
                newTourDialog.getContentPane().add(OKButton,BorderLayout.SOUTH);
                OKButton.addActionListener(thisObject);
                newTourDialog.setSize(200,150);
                newTourForDuty=null;
                newTourDialog.show();

                if(newTourForDuty!=null){
                  //Schicht aus der alten Tour löschen

  usedModel.removeVisitFromTour(dropVisitData.getPlanedItem(),dropVisitData.getTour
());

                  //Schicht der neuen Tour zuweisen, dann Besuch dieser Tour
zuweisen
                  usedModel.assignDutyToTour(transferDuty,newTourForDuty);
```

```
  usedModel.assignVisitToTour(dropVisitData.getPlanedItem().getPlanedVisit(),dropVi
sitData.getPlanedItem().getPlanedTime(),newTourForDuty);

                    //Änderungsbenachrichtigung!
                    usedModel.CarePlanModelChanged(null,1);
                  }
                }
              }
            }
          );
      else
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              int antw=JOptionPane.showConfirmDialog(null,"If you want to assign
this Duty to a Careclient\nyou must create a new Tour.\nCreate a new Tour now?",
"Unassigned Duty", JOptionPane.YES_NO_OPTION);
              if(antw==JOptionPane.YES_OPTION){

                newTourDialog=new JDialog(getFrame(dropComponent),"New Tour for
Duty",true);
                newTourDialog.getContentPane().setLayout(new BorderLayout());
                newTourGUI=new NewTourPanelGUI();

                //Start und Endzeit der neuen Tour mit Defaulteinstellungen belegen
                Time endTime=new Time();
                if(dropVisit.getPreferedTime()!=null){
                  endTime=new Time(dropVisit.getPreferedTime());

  if(dropVisit.getPreferedTime().isGreater(transferDuty.getStartTime()))
                    newTourGUI.setTourStart(transferDuty.getStartTime());
                  else
                    newTourGUI.setTourStart(dropVisit.getPreferedTime());
                }
                else {
                  endTime=new Time(dropVisit.getStartTime());

  if(dropVisit.getStartTime().isGreater(transferDuty.getStartTime()))
                    newTourGUI.setTourStart(transferDuty.getStartTime());
                  else
                    newTourGUI.setTourStart(dropVisit.getStartTime());
                }
                endTime.addTime(dropVisit.getLength());
                if(endTime.isGreater(transferDuty.getEndTime()))
                  newTourGUI.setTourEnd(endTime);
                else
                  newTourGUI.setTourEnd(transferDuty.getEndTime());
                //Defaulteinstellung erledigt

                newTourDialog.getContentPane().add(newTourGUI,BorderLayout.CENTER);
                JButton OKButton=new JButton("Create");
                newTourDialog.getContentPane().add(OKButton,BorderLayout.SOUTH);
                OKButton.addActionListener(thisObject);
                newTourDialog.setSize(200,150);
                newTourForDuty=null;
                newTourDialog.show();

                if(newTourForDuty!=null){

                  //Schicht der neuen Tour zuweisen, dann Besuch dieser Tour
zuweisen
                  usedModel.assignDutyToTour(transferDuty,newTourForDuty);
                  usedModel.assignVisitToTour(dropVisit,newTourForDuty);

                  //Änderungsbenachrichtigung!
                  usedModel.CarePlanModelChanged(null,1);
                }
              }
```

```
            }
          }
        );
  }

  //ActionListener Interface
  public void actionPerformed(ActionEvent e){
    newTourForDuty=new Tour();

    newTourForDuty.setName(newTourGUI.getTourName());
    newTourForDuty.setStartTime(newTourGUI.getTourStart());
    newTourForDuty.setEndTime(newTourGUI.getTourEnd());

    newTourDialog.dispose();

    usedModel.addTour(newTourForDuty);
  }

}
```

9.3.17. Klasse „DateChooser“

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

/**
 * Über diese Klasse lässt sich das dargestellte Datum wählen. Wenn das Datum
geändert wird,
 * wird CarePlanModelDatabaseConnection darüber informiert, so dass das Model neu
 * gefüllt werden kann.
 */
public class DateChooser implements ActionListener{

  private static DateFormat df=DateFormat.getDateInstance();

  private JPanel datePanel;
  private Date displayedDate;
  private JTextField dateInputBox;
  private CarePlanModelDatabaseConnection usedCPMDBConnection;

  public DateChooser(){
    try{
    displayedDate=df.parse("01.01.2000");
    } catch (Exception e){
    }
    createComponent();
  }

  public DateChooser(CarePlanModelDatabaseConnection usedCPMDBConnection){
    try{
    displayedDate=df.parse("01.01.2000");
    } catch (Exception e){
    }
    createComponent();
    this.usedCPMDBConnection=usedCPMDBConnection;
    usedCPMDBConnection.fillCarePlanModel(displayedDate);
  }

  private void createComponent(){
    JFrame dateFrame=new JFrame("Select displayed Date");
    datePanel=new JPanel();
    datePanel.setLayout(new BorderLayout());
    JPanel dateInput=new JPanel();
```

```
    dateInput.setLayout(new FlowLayout());
    dateInput.add(new JLabel("Displayed Date:"));
    dateInputBox=new JTextField();
    dateInputBox.setColumns(10);
    dateInputBox.setText(df.format(displayedDate));
    dateInput.add(dateInputBox);
    datePanel.add(dateInput,BorderLayout.CENTER);
    JButton setDateButton=new JButton("Set displayed Date");
    setDateButton.addActionListener(this);
    datePanel.add(setDateButton,BorderLayout.SOUTH);
    dateFrame.getContentPane().setLayout(new BorderLayout());
    dateFrame.getContentPane().add(datePanel,BorderLayout.CENTER);
    dateFrame.setSize(250,100);
    dateFrame.show();
  }

  public void actionPerformed(ActionEvent e){
    System.out.println("Changing displayed date ...");
    try{
      displayedDate=df.parse(dateInputBox.getText());
      //CarePlanModuleDatabaseConnection informieren, damit er sich neue Daten
holen kann!
      if(usedCPMDBConnection!=null)
        usedCPMDBConnection.fillCarePlanModel(displayedDate);

    }
    catch (Exception exception){
    }
    System.out.println("Date changed!");
  }

}
```

9.3.18. Klasse „Duty“

```
import java.io.*;

/**
 * Diese Klasse repräsentiert eine Schicht.
 */
public class Duty implements Serializable {

  private Employee Employee;
  private Long iD;
  private Time StartTime;
  private Time EndTime;

  public Duty(){
  }

  public Duty(Employee Employee,Time StartTime,Time EndTime,Long iD){
    this.Employee = Employee;
    this.EndTime = EndTime;
    this.StartTime = StartTime;
    this.iD=iD;
  }

  public Employee getEmployee()
  {
    return this.Employee;
  }

  public void setEmployee(Employee Employee)
  {
    this.Employee = Employee;
  }
```

```
  public Time getStartTime()
  {
    return this.StartTime;
  }

  public void setStartTime(Time StartTime)
  {
    this.StartTime = StartTime;
  }

  public Time getEndTime()
  {
    return this.EndTime;
  }

  public void setEndTime(Time EndTime)
  {
    this.EndTime = EndTime;
  }

  public String toString(){
    return Employee.toString();
  }

  public boolean equals(Object obj)
  {
    Duty vglDuty=(Duty) obj;
    return Employee.equals(vglDuty.getEmployee())
      && EndTime.equals(vglDuty.getEndTime())
      && StartTime.equals(vglDuty.getStartTime())
      && iD.equals(vglDuty.getID());
  }


  public Long getID()
{
  return this.iD;
}

  public void setID(Long iD)
{
  this.iD = iD;
}
}
```

9.3.19. Klasse „Employee“

```
import java.io.*;

/**
 * Diese Klasse repräsentiert einen Mitarbeiter.
 */

public class Employee implements Serializable{

  private String name;
  private int qualifikation;
  private Long iD;

  public Employee(String name,int qualifikation, Long iD){
    this.name=name;
    this.qualifikation=qualifikation;
    this.iD=iD;
  }
```

```
  public String getName()
{
  return this.name;
}

  public void setName(String name)
{
  this.name = name;
}

  public int getQualifikation()
{
  return this.qualifikation;
}

  public void setQualifikation(int qualifikation)
{
  this.qualifikation = qualifikation;
}

  public String toString()
  {
    return name;
  }

  public Long getID()
{
  return this.iD;
}

  public void setID(Long iD)
{
  this.iD = iD;
}

  public boolean equals(Object obj)
  {
    return iD.equals(((Employee)obj).getID());
  }
}
```

9.3.20. Klasse „JDBCEngine“

```
import java.sql.*;
import java.util.*;

/**
 * Diese Klasse bietet Methoden zur Datenbankanbindung.
 */

public class JDBCEngine{

  private String ODBC_Name;
  private String DatabaseURL;
  private  Connection connect;
  private ResultSet result_of_query;
  private ResultSetMetaData MetaData;
  private String SQLQuery;
  private boolean isConnected;
  Vector Spalten = new Vector();

  public JDBCEngine(String ODBC_Name){
    try{
      this.ODBC_Name=ODBC_Name;
      System.out.println("New JDBCEngine on jdbc:odbc:" + ODBC_Name + "\n");
```

```
      try {
         System.out.print("Loading JDBC-driver...");
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");   // Laden des JDBC-
Treibers
         System.out.println("done.");
         DatabaseURL = "jdbc:odbc:" + ODBC_Name;
      }
      catch(Exception exception) {
         System.out.println("An Exception has occured");
         System.out.println(exception);
      }
    }
    catch(Exception normalException){
      System.out.println("Exception occured....");
      System.out.println(normalException);
    }
    isConnected=false;
  }

  public int getRowCount(){
    int returnValue=0;
    try{
      while(result_of_query.next()){
        returnValue++;
      }
    }
    catch(SQLException exception){
      System.out.println("SQLException has occured");
      System.out.println(exception);
    }
    return returnValue;
  }

  public void connect(){
    if (isConnected==false)
    {
      System.out.print("Connecting to database...");

      try{
        connect = DriverManager.getConnection(DatabaseURL,"","");  // Strings sind
für %username und %password
        System.out.println("done.");
      }
      catch(SQLException exception){
        System.out.println("SQLException has occured");
        System.out.println(exception);
      }
      isConnected=true;
    }
  }

  public ResultSet getSQLResult(String SQLQuery){
    result_of_query=null;
    this.SQLQuery=SQLQuery;
    try{
      Statement query = connect.createStatement();    // Erzeugen eines Statements
      result_of_query = query.executeQuery(SQLQuery);
      MetaData = result_of_query.getMetaData();
    }
    catch(SQLException exception){
      System.out.println("SQLException has occured");
      System.out.println(exception);
    }
    return result_of_query;
  }

  public void SQLUpdate(String updateString){
    int updateResult;
```

```
    try{
      Statement query = connect.createStatement();    // Erzeugen eines Statements
      updateResult = query.executeUpdate(updateString);
    }
    catch(SQLException exception){
      System.out.println("SQLException has occured");
      System.out.println(exception);
    }
  }

  public void print_all(){
    String Header="";
    String Zeile;
    try{
      Statement query = connect.createStatement();
      ResultSet AllData=query.executeQuery(SQLQuery);
      int columns = MetaData.getColumnCount();
      for(int i=1; i<=columns; i++){
        Spalten.add(new Integer(i));
        Header = Header + MetaData.getColumnName(i) + "\t\t";
      }
      System.out.println(Header);

      while(AllData.next()){
        Zeile="";
        for(int i=0; i<Spalten.size(); i++){
            Zeile = Zeile +
AllData.getString(((Integer)(Spalten.get(i))).intValue()) + "\t\t";
        }
        System.out.println(Zeile);
      }
    }
    catch(SQLException exception){
      System.out.println("SQLException has occured");
      System.out.println(exception);
    }
  }

  public void disconnect(){
    if (isConnected==true)
    {
      System.out.print("Closing connection...");
      try{
        connect.close();
      }
      catch(SQLException exception){
        System.out.println("SQLException has occured");
        System.out.println(exception);
      }
      System.out.println("done.\n");
      isConnected=false;
    }
  }

  public boolean getIsConnected()
  {
    return this.isConnected;
  }

}
```

9.3.21. Klasse „NewTourPanelGUI“

```
/**
 * Diese Klasse beitet lediglich Methoden um auf Felder der von JForge
 * erzeugte Klasse NewTourPanelRSC zuzugreifen.
```

```
 * Diese  wurde erzeugt um neue Touren über einen Dialog anzulegen.
 */

public class NewTourPanelGUI extends NewTourPanelRSC{

  public NewTourPanelGUI(){
  }

  public String getTourName(){
    return JTextFieldTourName.getText();

  }

  public Time getTourStart(){
    return Time.parse(JTextFieldTourStart.getText());
  }

  public Time getTourEnd(){
    return Time.parse(JTextFieldTourEnd.getText());
  }

  public void setTourStart(Time StartTime){
    JTextFieldTourStart.setText(StartTime.toString());
  }

  public void setTourEnd(Time EndTime){
    JTextFieldTourEnd.setText(EndTime.toString());
  }
}
```

9.3.22. Klasse „NewTourPanelRSC“

```
/*-----------------------------------------
 Java GUI generated by JForge version 2.62
 The Java GUI Builder, by Tek-Tools


 File   : C:/Daten/Michael/Diplomarbeit/CarePlanModule/NewTourPanelRSC.java
-----------------------------------------*/

import java.awt.*;
import java.net.*;
import javax.swing.*;

public class NewTourPanelRSC extends JPanel
{


  // Instance variables
  public JPanel jpanel1;
  public JLabel jlabel0;
  public JTextField JTextFieldTourName;
  public JPanel jpanel2;
  public JLabel jlabel1;
  public JTextField JTextFieldTourStart;
  public JPanel jpanel3;
  public JLabel jlabel2;
  public JTextField JTextFieldTourEnd;

  public NewTourPanelRSC()
  {

    // Setup GUI

    this.setBounds(203,101,194,100);
```

```
    this.setSize(new Dimension(194,100));
    this.setPreferredSize(new Dimension(181,93));
    this.setMinimumSize(new Dimension(85,93));
    this.setBounds(new Rectangle(203,101,194,100));
    this.setLocation(new Point(203,101));
    this.setName("this");

    jpanel1 = new JPanel();
    jpanel1.setPreferredSize(new Dimension(181,31));
    jpanel1.setMinimumSize(new Dimension(85,31));
    jpanel1.setName("jpanel1");

    jlabel0 = new JLabel();
    jlabel0.setMaximumSize(new Dimension(66,17));
    jlabel0.setMinimumSize(new Dimension(66,17));
    jlabel0.setText("Tour-Name:");
    jlabel0.setName("jlabel0");
    jlabel0.setLayout(null);

    JTextFieldTourName = new JTextField();
    JTextFieldTourName.setSelectionEnd(8);
    JTextFieldTourName.setPreferredSize(new Dimension(100,21));
    JTextFieldTourName.setText("New Tour");
    JTextFieldTourName.setName("JTextFieldTourName");
    JTextFieldTourName.setScrollOffset(5);
    JTextFieldTourName.setSelectionStart(8);
    JTextFieldTourName.setLayout(null);

    FlowLayout jpanel1Layout = new FlowLayout();
    jpanel1Layout.setHgap(5);
    jpanel1Layout.setVgap(5);
    jpanel1Layout.setAlignment(FlowLayout.LEFT);
  jpanel1.setLayout(jpanel1Layout);
    jpanel1.add(jlabel0);
    jpanel1.add(JTextFieldTourName);

    jpanel2 = new JPanel();
    jpanel2.setPreferredSize(new Dimension(126,31));
    jpanel2.setMinimumSize(new Dimension(80,31));
    jpanel2.setName("jpanel2");

    jlabel1 = new JLabel();
    jlabel1.setMaximumSize(new Dimension(61,17));
    jlabel1.setMinimumSize(new Dimension(61,17));
    jlabel1.setText("Tour-Start:");
    jlabel1.setName("jlabel1");
    jlabel1.setLayout(null);

    JTextFieldTourStart = new JTextField();
    JTextFieldTourStart.setSelectionEnd(4);
    JTextFieldTourStart.setPreferredSize(new Dimension(50,21));
    JTextFieldTourStart.setText("6:00");
    JTextFieldTourStart.setName("JTextFieldTourStart");
    JTextFieldTourStart.setScrollOffset(8);
    JTextFieldTourStart.setSelectionStart(4);
    JTextFieldTourStart.setLayout(null);

    FlowLayout jpanel2Layout = new FlowLayout();
    jpanel2Layout.setHgap(5);
    jpanel2Layout.setVgap(5);
    jpanel2Layout.setAlignment(FlowLayout.LEFT);
  jpanel2.setLayout(jpanel2Layout);
    jpanel2.add(jlabel1);
    jpanel2.add(JTextFieldTourStart);

    jpanel3 = new JPanel();
    jpanel3.setPreferredSize(new Dimension(119,31));
    jpanel3.setMinimumSize(new Dimension(73,31));
```

```
    jpanel3.setName("jpanel3");

    jlabel2 = new JLabel();
    jlabel2.setMaximumSize(new Dimension(54,17));
    jlabel2.setMinimumSize(new Dimension(54,17));
    jlabel2.setText("Tour-End:");
    jlabel2.setName("jlabel2");
    jlabel2.setLayout(null);

    JTextFieldTourEnd = new JTextField();
    JTextFieldTourEnd.setSelectionEnd(5);
    JTextFieldTourEnd.setPreferredSize(new Dimension(50,21));
    JTextFieldTourEnd.setText("14:00");
    JTextFieldTourEnd.setName("JTextFieldTourEnd");
    JTextFieldTourEnd.setScrollOffset(8);
    JTextFieldTourEnd.setSelectionStart(5);
    JTextFieldTourEnd.setLayout(null);

    FlowLayout jpanel3Layout = new FlowLayout();
    jpanel3Layout.setHgap(5);
    jpanel3Layout.setVgap(5);
    jpanel3Layout.setAlignment(FlowLayout.LEFT);
  jpanel3.setLayout(jpanel3Layout);
    jpanel3.add(jlabel2);
    jpanel3.add(JTextFieldTourEnd);

    GridLayout thisLayout = new GridLayout(1,1);
    thisLayout.setHgap(0);
    thisLayout.setVgap(0);
    thisLayout.setRows(3);
    thisLayout.setColumns(1);
  this.setLayout(thisLayout);
    this.add(jpanel1);
    this.add(jpanel2);
    this.add(jpanel3);
  }


  URL getURL(String txtURL)
  {
    // return a URL object given the URL
    try
    {
      if (txtURL == null)
        return new URL("file://");
      else
        return new URL(txtURL);
    }
    catch(MalformedURLException mue)
    {
      return null;
    }
  }


  public String[][] getIconPathForComponent(Component comp, String methodName)
  {
    // return an Icon path(s) for a given Component/Method
    {
      String[][] rval = {{null}};
      return rval;
    }
  }


  private String getReference(String[][] references, String methodName)
  {
    for (int i = 0; i < references.length; i++)
```

```
      if (references[i][0] == methodName)
        return references[i][1];
    return null;
  }

}
```

9.3.23. Klasse „NotDropableDropListener“

```
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

/**
 * Hier wird eine Klasse definiert, die eine Droplistener implementiert, der
 * alle Drop-Versuche zurückweist. Er wird verwendet, da nur eine DropTarget
 * Autoscrollfunktionalität besitzen kann, aber evtl. auch über Komponenten
 * Gescrollt werden soll, die kein DropTarget im eigentlichen Sinne sind.
 * Diese werden dann einfach alks Droptarget mit diesem DropListener konfiguriert.
 */
public class NotDropableDropListener
  implements DropTargetListener {

  // interface DropTargetListener
  public void dragEnter (DropTargetDragEvent dropTargetDragEvent) {
    dropTargetDragEvent.rejectDrag();
  }

  public void dragExit (DropTargetEvent dropTargetEvent) {
  }

  public void dragOver (DropTargetDragEvent dropTargetDragEvent) {
    dropTargetDragEvent.rejectDrag();
  }

  public void dropActionChanged (DropTargetDragEvent dropTargetDragEvent) {
    dropTargetDragEvent.rejectDrag();
  }

  public void drop (DropTargetDropEvent dropTargetDropEvent) {
     dropTargetDropEvent.rejectDrop ();
  }

}
```

9.3.24. Interface „PlanableCareClient“

```
import java.io.*;

/**
 * Definiert eine Schnittstelle für CareClients, die mit diesem
 * Prototypen geplant werden sollen. Dies vereinfacht die Integration
 * in ein bestehendes System.
 */

public interface PlanableCareClient{

  /**
   * Wird immer dann aufgerufen, wenn eine Bildschirm-Repräsentation des
   * CareClients in Stringform benötigt wird.
   */
  public String toString();
```

```
  /**
   * Prüft, ob ein PlanableCareClient dem anderen gleicht.
   */
  public boolean equals(PlanableCareClient cc);

}
```

9.3.25. Klasse „PlanedDutyData“

```
import java.io.*;

/**
 * Diese Klasse fügt der Information über eine Schicht die
 * zugeordnete Tour hinzu. Wird bei Drag and Drop Operationen
 * benötigt.
 */

public class PlanedDutyData implements Serializable{

  private Duty duty;
  private Tour tour;

  public PlanedDutyData(Duty duty,Tour tour){
    this.duty=duty;
    this.tour=tour;
  }

  public Tour getTour(){
    return tour;
  }

  public void setTour(Tour tour){
    this.tour=tour;
  }

  public void setDuty(Duty duty){
    this.duty=duty;
  }

  public Duty getDuty(){
    return duty;
  }
}
```

9.3.26. Klasse „PlanedItem“

```
import java.io.*;

/**
 * Repräsentiert einen geplanten Besuch mit zugehöriger Planzeit.
 */

public class PlanedItem implements Serializable{

  private Visit PlanedVisit;
  private Time PlanedTime;

  public PlanedItem(){
  }

  public PlanedItem(Visit PlanedVisit,Time PlanedTime){
    this.PlanedVisit = PlanedVisit;
    this.PlanedTime = PlanedTime;
```

```
  }

  public Visit getPlanedVisit()
  {
    return this.PlanedVisit;
  }

  public void setPlanedVisit(Visit PlanedVisit)
  {
    this.PlanedVisit = PlanedVisit;
  }

  public Time getPlanedTime()
  {
    return this.PlanedTime;
  }

  public void setPlanedTime(Time PlanedTime)
  {
    this.PlanedTime = PlanedTime;
  }

  public String toString(){
    return PlanedVisit.toString();
  }

  public boolean equals(Object obj)
  {
    PlanedItem pi=(PlanedItem)obj;
    if(PlanedVisit!=null)
      if(pi.getPlanedVisit()!=null)
        return pi.getPlanedVisit().equals(PlanedVisit) &&
pi.getPlanedTime().equals(PlanedTime);
      else
        return false;
    else
      if(pi.getPlanedVisit()!=null)
        return false;
      else
        return pi.getPlanedTime().equals(PlanedTime);
  }
}
```

9.3.27. Klasse „PlanedVisitData“

```
import java.io.*;

/**
 * Fügt einem PlanedItem die Information hinzu in welcher Tour
 * er geplant wurde. Für Drag and Drop Operationen wichtig.
 */
public class PlanedVisitData implements Serializable{

  private Tour tour;
  private PlanedItem planedItem;

  public PlanedVisitData(Tour tour,PlanedItem planedItem){
    this.tour=tour;
    this.planedItem=planedItem;
  }

  public PlanedItem getPlanedItem(){
    return planedItem;
  }

  public void setPlanedItem(PlanedItem planedItem){
```

```
    this.planedItem=planedItem;
  }

  public Tour getTour(){
    return tour;
  }

  public void setTour(Tour tour){
    this.tour=tour;
  }

  public boolean equals(Object obj)
  {
    PlanedVisitData vgldata=(PlanedVisitData) obj;
    return vgldata.getPlanedItem().equals(planedItem)
      && vgldata.getTour().equals(tour);
  }
}
```

9.3.28. Klasse „PlanedVisitsDragAndDrop“

```
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

/**
 * Bearbeitet Alle Drag Operationen auf bereits geplanten Besuchen.
 */
public class PlanedVisitsDragAndDrop
  implements DragGestureListener,DragSourceListener{

  private TourViewFactory usedFactory;
  private CarePlanModel usedModel;

  public PlanedVisitsDragAndDrop(){
    this.usedFactory=new SimpleTourViewFactory();
  }

  public PlanedVisitsDragAndDrop(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  public void setTourViewFactory(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  public void setCarePlanModel(CarePlanModel usedModel){
    this.usedModel=usedModel;
  }

  //DragGestureListener
  public void dragGestureRecognized(DragGestureEvent dge){
    if(usedFactory.getPlanedVisitDataFromComponent(dge.getComponent())!=null)

  if(usedFactory.getPlanedVisitDataFromComponent(dge.getComponent()).getPlanedItem(
).getPlanedVisit()!=null)
        dge.getDragSource().startDrag(dge, DragSource.DefaultMoveDrop, new
TransferablePlanedVisitData(usedFactory.getPlanedVisitDataFromComponent(dge.getComp
onent())), this);
  }

  //DragSourceListener
  public void dragDropEnd(DragSourceDropEvent DragSourceDropEvent) {
    if(!DragSourceDropEvent.getDropSuccess()){
      //aus der Tour entfernen und in die ungeplantenListe schreiben.
```

```
        PlanedVisitData
dragVisit=usedFactory.getPlanedVisitDataFromComponent(DragSourceDropEvent.getDragSo
urceContext().getComponent());
        usedModel.removeVisitFromTour(dragVisit.getPlanedItem(),dragVisit.getTour());
        usedModel.addVisit(dragVisit.getPlanedItem().getPlanedVisit());

        //Dieser umständliche Aufruf von 'usedModel.CarePlanModelChanged(null,1)' muß
sein
        //da sich sonst JAVA aufhängt!!!! Keine Ahnung warum!
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
               usedModel.CarePlanModelChanged(null,1);
            }
          }
        );
      }

    }

    public void dragEnter(DragSourceDragEvent DragSourceDragEvent) {
    }
    public void dragExit(DragSourceEvent DragSourceEvent) {
    }
    public void dragOver(DragSourceDragEvent DragSourceDragEvent) {
    }
    public void dropActionChanged(DragSourceDragEvent DragSourceDragEvent) {
    }

}
```

9.3.29. Klasse „SimpleClientViewFactory“

```
import java.util.*;
import javax.swing.*;
import java.text.*;
import java.awt.*;


/**
 * Eine konkrete ClientViewFactory.
 */
public class SimpleClientViewFactory extends ClientViewFactory{

    /**
     * Dieses Feld legt fest, dass 45 Sekunden auf dem Bildschirm einen
     * Pixel repräsentieren.
     */
    private long tourZoom=45000;

    /**
     * Dieses Feld legt fest, dass das Zeitraster 30 Minuten betragen soll.
     */
    private Time dayRaster=new Time(0,30,0); //30 Minuten

    /**
     * Dieses Feld legt fest, dass Ein Arbeitstag um 6 Uhr beginnen soll
     */
    private Time dayStart=new Time(6,0,0); //6 Uhr
    /**
     * Dieses Feld legt fest, dass Ein Arbeitstag um 22 Uhr enden soll
     */
    private Time dayEnd=new Time(22,0,0); //22 Uhr

    public SimpleClientViewFactory(){
    }
```

```
  public JPanel createVerticalTimeBar(Time StartTime,Time EndTime){
    DateFormat df = DateFormat.getTimeInstance(DateFormat.SHORT);
    JPanel newBar=new JPanel();
    JLabel newLabel;
    for(Time dayAktTime=new
Time(StartTime);EndTime.isGreater(dayAktTime);dayAktTime.addTime(dayRaster)){
      newLabel=new JLabel(dayAktTime.toString());
      newLabel.setPreferredSize(new
Dimension(40,(int)((dayRaster.getTime()/tourZoom)-1)));
      newLabel.setForeground(new Color(0,0,0));
      newLabel.setHorizontalAlignment(SwingConstants.RIGHT);
      newLabel.setOpaque(true);
      newBar.add(newLabel);
    }
    newBar.setBackground(new Color(0,0,0));
    newBar.setPreferredSize(new Dimension(40,(int)((EndTime.getTime()-
StartTime.getTime())/tourZoom)));
    return newBar;
  }

  private AssociatedJPanel createUnplanedVisitPanelWithVisit(Visit upVisit,int
breite){
    AssociatedJPanel newPanel=new AssociatedJPanel(upVisit);
    newPanel.setLayout(new FlowLayout(FlowLayout.LEFT,0,1));
    newPanel.setAutoscrolls(true);

    AssociatedJLabel LengthLabel=new
AssociatedJLabel(upVisit.getLength().toString(),upVisit);
    LengthLabel.setPreferredSize(new Dimension(breite,15));
    LengthLabel.setForeground(new Color(0,0,0));
    LengthLabel.setAutoscrolls(true);
    newPanel.add(LengthLabel);

    String Time;
    Time=upVisit.getStartTime().toString() + '-' + upVisit.getEndTime().toString();
    if(upVisit.getPreferedTime()!=null){
      Time=upVisit.getPreferedTime().toString() + " (" + Time + ')';
    }
    AssociatedJLabel TimeLabel=new AssociatedJLabel(Time,upVisit);
    TimeLabel.setPreferredSize(new Dimension(breite,15));
    TimeLabel.setForeground(new Color(0,0,0));
    TimeLabel.setAutoscrolls(true);
    newPanel.add(TimeLabel);

    newPanel.setBackground(new Color(140,160,255));
    newPanel.setPreferredSize(new Dimension(breite,
(int)(((upVisit.getEndTime().getTime()-upVisit.getStartTime().getTime())/tourZoom)-
1) ));
    return newPanel;
  }

  private AssociatedJPanel
createPlanedVisitPanelWithPlanedVisitData(PlanedVisitData pVisitData,int breite){
    AssociatedJPanel newPanel=new AssociatedJPanel(pVisitData);
    newPanel.setLayout(new FlowLayout(FlowLayout.LEFT,0,1));
    newPanel.setAutoscrolls(true);

    AssociatedJLabel LengthLabel=null;
    if(pVisitData.getTour().getAssignedDuty()!=null){
      LengthLabel=new
AssociatedJLabel(pVisitData.getTour().getAssignedDuty().getEmployee().toString(),pV
isitData);
      LengthLabel.setPreferredSize(new Dimension(breite,15));
      LengthLabel.setForeground(new Color(0,0,0));
      LengthLabel.setAutoscrolls(true);
      newPanel.add(LengthLabel);

      LengthLabel=new AssociatedJLabel(pVisitData.getTour().getName(),pVisitData);
```

```
      LengthLabel.setPreferredSize(new Dimension(breite,15));
      LengthLabel.setForeground(new Color(0,0,0));
      LengthLabel.setAutoscrolls(true);
      newPanel.add(LengthLabel);

    }
    else {
      LengthLabel=new AssociatedJLabel(pVisitData.getTour().getName(),pVisitData);
      LengthLabel.setPreferredSize(new Dimension(breite,15));
      LengthLabel.setForeground(new Color(0,0,0));
      LengthLabel.setAutoscrolls(true);
      newPanel.add(LengthLabel);
    }

    newPanel.setBackground(new Color(140,160,255));
    newPanel.setPreferredSize(new Dimension(breite,
(int)(((pVisitData.getPlanedItem().getPlanedVisit().getLength().getTime())/tourZoom
)-1) ));
    return newPanel;
  }

  public JPanel createBodyPanelWithClientVisits(Vector unplanedVisits,Vector
planedVisits){
    AutoScrollJPanel body=new AutoScrollJPanel();
    body.setAutoscrolls(true);


    body.setLayout(new GridLayout(1,2,1,0));

    //Alle ungeplanten Besuche in einen Vector
    Vector allUnplanedVisits=new Vector();
    for(int i=0;i<unplanedVisits.size();i++)
      allUnplanedVisits.add(unplanedVisits.get(i));
    //Jetzt die geplanten Besuche hier einsortieren
    Visit vglVisit;
    Time vglTime;
    boolean inserted;
    for(int i=0;i<planedVisits.size();i++){

  vglVisit=((PlanedVisitData)planedVisits.get(i)).getPlanedItem().getPlanedVisit();
      if(vglVisit.getPreferedTime()!=null){
        vglTime=vglVisit.getPreferedTime();
      }
      else {
        vglTime=vglVisit.getStartTime();
      }
      inserted=false;
      for(int j=0;j<allUnplanedVisits.size();j++){
        if(((Visit)allUnplanedVisits.get(j)).getPreferedTime()!=null){

  if(((Visit)allUnplanedVisits.get(j)).getPreferedTime().isGreater(vglTime)){
            inserted=true;
            allUnplanedVisits.add(j,vglVisit);
          }
        }
        else {
          if(((Visit)allUnplanedVisits.get(j)).getStartTime().isGreater(vglTime)){
            inserted=true;
            allUnplanedVisits.add(j,vglVisit);
          }
        }
      }
      if(!inserted){
        allUnplanedVisits.add(vglVisit);
      }
    }

    //Jetzt Panel aus allen ungeplanten Besuchen erstellen
```

```
    AutoScrollJPanel upVisitsPanel=new AutoScrollJPanel();
    upVisitsPanel.setAutoscrolls(true);
    upVisitsPanel.setLayout(new FlowLayout(FlowLayout.CENTER,0,1));
    Time aktZeit=new Time(dayStart);
    AutoScrollJPanel fillPanel;
    for(int i=0;i<allUnplanedVisits.size();i++){
      if(((Visit)allUnplanedVisits.get(i)).getStartTime().isGreater(aktZeit)){
        fillPanel=new AutoScrollJPanel();
        fillPanel.setAutoscrolls(true);
        fillPanel.setPreferredSize(new
Dimension(100,((int)((((Visit)allUnplanedVisits.get(i)).getStartTime().getTime()-
aktZeit.getTime())/tourZoom)-1 )));
        upVisitsPanel.add(fillPanel);
      }

  upVisitsPanel.add(createUnplanedVisitPanelWithVisit((Visit)allUnplanedVisits.get(
i),100));
      aktZeit=new Time(((Visit)allUnplanedVisits.get(i)).getEndTime());
    }
    if(dayEnd.isGreater(aktZeit)){
      fillPanel=new AutoScrollJPanel();
      fillPanel.setAutoscrolls(true);
      fillPanel.setPreferredSize(new Dimension(100,((int)((dayEnd.getTime()-
aktZeit.getTime())/tourZoom)-1 )));
      upVisitsPanel.add(fillPanel);
    }
    upVisitsPanel.setBackground(Color.black);
    body.add(upVisitsPanel);

    //Jetzt Panel aus allen geplanten Besuchen erstellen
    AutoScrollJPanel pVisitsPanel=new AutoScrollJPanel();
    pVisitsPanel.setAutoscrolls(true);
    pVisitsPanel.setLayout(new FlowLayout(FlowLayout.CENTER,0,1));
    aktZeit=new Time(dayStart);
    for(int i=0;i<planedVisits.size();i++){

  if(((PlanedVisitData)planedVisits.get(i)).getPlanedItem().getPlanedTime().isGreat
er(aktZeit)){
        fillPanel=new AutoScrollJPanel();
        fillPanel.setAutoscrolls(true);
        fillPanel.setPreferredSize(new
Dimension(100,((int)((((PlanedVisitData)planedVisits.get(i)).getPlanedItem().getPla
nedTime().getTime()-aktZeit.getTime())/tourZoom)-1 )));
        pVisitsPanel.add(fillPanel);
      }

  pVisitsPanel.add(createPlanedVisitPanelWithPlanedVisitData((PlanedVisitData)plane
dVisits.get(i),100));
      aktZeit=new
Time(((PlanedVisitData)planedVisits.get(i)).getPlanedItem().getPlanedTime());

  aktZeit.addTime(((PlanedVisitData)planedVisits.get(i)).getPlanedItem().getPlanedV
isit().getLength());
    }
    if(dayEnd.isGreater(aktZeit)){
      fillPanel=new AutoScrollJPanel();
      fillPanel.setAutoscrolls(true);
      fillPanel.setPreferredSize(new Dimension(100,((int)((dayEnd.getTime()-
aktZeit.getTime())/tourZoom)-1 )));
      pVisitsPanel.add(fillPanel);
    }
    pVisitsPanel.setBackground(Color.black);
    body.add(pVisitsPanel);

    body.setPreferredSize(new Dimension(201,(int)((dayEnd.getTime()-
dayStart.getTime())/tourZoom)));

    return body;
```

```
  }

  private AssociatedJPanel createUnplanedDutyPanel(Duty dutyToDisplay,int breite){
    DateFormat df = DateFormat.getTimeInstance(DateFormat.SHORT);
    AssociatedJPanel returnPanel=new AssociatedJPanel(dutyToDisplay);

    AssociatedJLabel NameLabel=new
AssociatedJLabel(dutyToDisplay.getEmployee().toString(),dutyToDisplay);
    NameLabel.setPreferredSize(new Dimension(breite,15));
    NameLabel.setForeground(Color.black);
    returnPanel.add(NameLabel);

    AssociatedJLabel LengthLabel=new AssociatedJLabel("Qualifikation:
"+dutyToDisplay.getEmployee().getQualifikation(),dutyToDisplay);
    LengthLabel.setPreferredSize(new Dimension(breite,15));
    LengthLabel.setForeground(Color.black);
    returnPanel.add(LengthLabel);

    String Time;
    Time=dutyToDisplay.getStartTime().toString() + '-' +
dutyToDisplay.getEndTime().toString();
    AssociatedJLabel TimeLabel=new AssociatedJLabel(Time,dutyToDisplay);
    TimeLabel.setPreferredSize(new Dimension(breite,15));
    TimeLabel.setForeground(Color.black);
    returnPanel.add(TimeLabel);

    returnPanel.setBackground(new Color(230,230,0));
    returnPanel.setPreferredSize(new Dimension(breite,65));
    return returnPanel;
  }

  private AssociatedJPanel createPlanedDutyPanel(Tour dutyToDisplay,int breite){
    DateFormat df = DateFormat.getTimeInstance(DateFormat.SHORT);
    AssociatedJPanel returnPanel=new AssociatedJPanel(dutyToDisplay);

    if(dutyToDisplay.getAssignedDuty()!=null){
      AssociatedJLabel NameLabel=new
AssociatedJLabel(dutyToDisplay.getAssignedDuty().getEmployee().toString(),dutyToDis
play);
      NameLabel.setPreferredSize(new Dimension(breite,15));
      NameLabel.setForeground(Color.black);
      returnPanel.add(NameLabel);
      AssociatedJLabel LengthLabel=new AssociatedJLabel("Qualifikation:
"+dutyToDisplay.getAssignedDuty().getEmployee().getQualifikation(),dutyToDisplay);
      LengthLabel.setPreferredSize(new Dimension(breite,15));
      LengthLabel.setForeground(Color.black);
      returnPanel.add(LengthLabel);

      String Time;
      Time=dutyToDisplay.getStartTime().toString() + '-' +
dutyToDisplay.getAssignedDuty().getEndTime().toString();
      AssociatedJLabel TimeLabel=new AssociatedJLabel(Time,dutyToDisplay);
      TimeLabel.setPreferredSize(new Dimension(breite,15));
      TimeLabel.setForeground(Color.black);
      returnPanel.add(TimeLabel);
    }


    AssociatedJLabel TourLabel=new
AssociatedJLabel(dutyToDisplay.getName(),dutyToDisplay);
    TourLabel.setPreferredSize(new Dimension(breite,15));
    TourLabel.setForeground(Color.black);
    returnPanel.add(TourLabel);

    returnPanel.setBackground(new Color(230,230,0));
    returnPanel.setPreferredSize(new Dimension(breite,65));
    return returnPanel;
  }
```

```
  public JPanel createDutiesPanelWithDutiesAndTours(Vector unplanedDuties,Vector
tours){
    JPanel dutiesPanel=new JPanel();
    dutiesPanel.setLayout(new FlowLayout(FlowLayout.CENTER,0,1));

    int hoehe=0;
    AssociatedJPanel newPanel;
    if(unplanedDuties!=null){
      for(int i=0;i<unplanedDuties.size();i++){
        newPanel=createUnplanedDutyPanel((Duty)unplanedDuties.get(i),130);
        hoehe+=newPanel.getPreferredSize().getHeight()+1;
        dutiesPanel.add(newPanel);
      }
    }

    if(tours!=null){
      for(int i=0;i<tours.size();i++){
        newPanel=createPlanedDutyPanel((Tour)tours.get(i),130);
        hoehe+=newPanel.getPreferredSize().getHeight()+1;
        dutiesPanel.add(newPanel);
      }
    }

    dutiesPanel.setPreferredSize(new Dimension(130,hoehe));
    return dutiesPanel;
  }

  public JPanel createHeaderPanelWithCareClient(PlanableCareClient careClient){
    JPanel headerPanel=new JPanel();
    headerPanel.setLayout(new BorderLayout());
    JLabel nameLabel=new JLabel(careClient.toString());
    nameLabel.setForeground(Color.black);
    nameLabel.setBackground(new Color(252,171,3));
    nameLabel.setHorizontalAlignment(JLabel.CENTER);
    nameLabel.setOpaque(true);

    headerPanel.add(nameLabel,BorderLayout.CENTER);
    JPanel titelPanel=new JPanel();
    titelPanel.setLayout(new GridLayout(1,2,1,0));
    nameLabel=new JLabel("Soll");
    nameLabel.setForeground(Color.black);
    nameLabel.setHorizontalAlignment(JLabel.CENTER);
    nameLabel.setOpaque(true);
    titelPanel.add(nameLabel);
    nameLabel=new JLabel("Ist");
    nameLabel.setForeground(Color.black);
    nameLabel.setHorizontalAlignment(JLabel.CENTER);
    nameLabel.setOpaque(true);
    titelPanel.add(nameLabel);
    titelPanel.setBackground(Color.black);
    headerPanel.add(titelPanel,BorderLayout.SOUTH);
    headerPanel.setPreferredSize(new Dimension(201,35));
    return headerPanel;
  }

  public Tour getTourFromComponent(Component comp){
    if(((AssociatedComponent)comp).getAssociatedObject().getClass()==(new
Tour()).getClass())
      return (Tour)((AssociatedComponent)comp).getAssociatedObject();
    else
      return null;
  }

  public Duty getDutyFromComponent(Component comp){
    if(((AssociatedComponent)comp).getAssociatedObject().getClass()==(new
Duty()).getClass())
      return (Duty)((AssociatedComponent)comp).getAssociatedObject();
```

```
    else
      return null;
  }

  public Visit getVisitFromComponent(Component comp){
    if(((AssociatedComponent)comp).getAssociatedObject().getClass()==(new
Visit()).getClass())
      return (Visit)((AssociatedComponent)comp).getAssociatedObject();
    else
      return null;
  }

  public PlanedVisitData getPlanedVisitDataFromComponent(Component comp){
    if(((AssociatedComponent)comp).getAssociatedObject().getClass()==(new
PlanedVisitData(new Tour(),new PlanedItem())).getClass())
      return (PlanedVisitData)((AssociatedComponent)comp).getAssociatedObject();
    else
      return null;
  }

}
```

9.3.30. Klasse „SimpleTourViewFactory“

```
import java.util.*;
import javax.swing.*;
import java.text.*;
import java.awt.*;

/**
 * Eine konkrete ClientViewFactory.
 */
public class SimpleTourViewFactory extends TourViewFactory{

  /**
   * Dieses Feld legt fest, dass 45 Sekunden auf dem Bildschirm einen
   * Pixel repräsentieren.
   */
  private long tourZoom=45000;

  /**
   * Dieses Feld legt fest, dass das Zeitraster 30 Minuten betragen soll.
   */
  private Time dayRaster=new Time(0,30,0); //30 Minuten

  /**
   * Dieses Feld legt fest, dass Ein Arbeitstag um 6 Uhr beginnen soll
   */
  private Time dayStart=new Time(6,0,0); //6 Uhr
  /**
   * Dieses Feld legt fest, dass Ein Arbeitstag um 22 Uhr enden soll
   */
  private Time dayEnd=new Time(22,0,0); //22 Uhr

  public SimpleTourViewFactory(){
  }

  public JPanel createVerticalTimeBar(Time StartTime,Time EndTime){
    DateFormat df = DateFormat.getTimeInstance(DateFormat.SHORT);
    JPanel newBar=new JPanel();
    JLabel newLabel;
    for(Time dayAktTime=new
Time(StartTime);EndTime.isGreater(dayAktTime);dayAktTime.addTime(dayRaster)){
      newLabel=new JLabel(dayAktTime.toString());
      newLabel.setPreferredSize(new
Dimension(40,(int)((dayRaster.getTime()/tourZoom)-1)));
```

```
        newLabel.setForeground(new Color(0,0,0));
        newLabel.setHorizontalAlignment(SwingConstants.RIGHT);
        newLabel.setOpaque(true);
        newBar.add(newLabel);
      }
      newBar.setBackground(new Color(0,0,0));
      newBar.setPreferredSize(new Dimension(40,(int)((EndTime.getTime()-
StartTime.getTime())/tourZoom)));
      return newBar;
    }

    private AssociatedJPanel createUnplanedVisitPanel(Visit visitToDisplay,int
breite){
      DateFormat df = DateFormat.getTimeInstance(DateFormat.SHORT);
      AssociatedJPanel returnPanel=new AssociatedJPanel(visitToDisplay);

      AssociatedJLabel NameLabel=new
AssociatedJLabel(visitToDisplay.getCareClient().toString(),visitToDisplay);
      NameLabel.setPreferredSize(new Dimension(breite,15));
      NameLabel.setForeground(new Color(0,0,0));
      returnPanel.add(NameLabel);

      AssociatedJLabel LengthLabel=new
AssociatedJLabel(visitToDisplay.getLength().toString(),visitToDisplay);
      LengthLabel.setPreferredSize(new Dimension(breite,15));
      LengthLabel.setForeground(new Color(0,0,0));
      returnPanel.add(LengthLabel);

      String Time;
      Time=visitToDisplay.getStartTime().toString() + '-' +
visitToDisplay.getEndTime().toString();
      if(visitToDisplay.getPreferedTime()!=null){
        Time=visitToDisplay.getPreferedTime().toString() + " (" + Time + ')';
      }
      AssociatedJLabel TimeLabel=new AssociatedJLabel(Time,visitToDisplay);
      TimeLabel.setPreferredSize(new Dimension(breite,15));
      TimeLabel.setForeground(new Color(0,0,0));
      returnPanel.add(TimeLabel);

      returnPanel.setBackground(new Color(140,160,255));
      returnPanel.setPreferredSize(new Dimension(breite,50));
      return returnPanel;
    }

    private AssociatedJPanel createUnplanedDutyPanel(Duty dutyToDisplay,int breite){
      DateFormat df = DateFormat.getTimeInstance(DateFormat.SHORT);
      AssociatedJPanel returnPanel=new AssociatedJPanel(dutyToDisplay);

      AssociatedJLabel NameLabel=new
AssociatedJLabel(dutyToDisplay.getEmployee().toString(),dutyToDisplay);
      NameLabel.setPreferredSize(new Dimension(breite,15));
      NameLabel.setForeground(Color.black);
      returnPanel.add(NameLabel);

      AssociatedJLabel LengthLabel=new AssociatedJLabel("Qualifikation:
"+dutyToDisplay.getEmployee().getQualifikation(),dutyToDisplay);
      LengthLabel.setPreferredSize(new Dimension(breite,15));
      LengthLabel.setForeground(Color.black);
      returnPanel.add(LengthLabel);

      String Time;
      Time=dutyToDisplay.getStartTime().toString() + '-' +
dutyToDisplay.getEndTime().toString();
      AssociatedJLabel TimeLabel=new AssociatedJLabel(Time,dutyToDisplay);
      TimeLabel.setPreferredSize(new Dimension(breite,15));
      TimeLabel.setForeground(Color.black);
      returnPanel.add(TimeLabel);
```

```
    returnPanel.setBackground(new Color(230,230,0));
    returnPanel.setPreferredSize(new Dimension(breite,50));
    return returnPanel;
  }

  public JPanel createDutiesPanel(Vector Duties){
    AssociatedJPanel returnPanel=new AssociatedJPanel();
    int hoehe=0;
    if(Duties!=null){
      AssociatedJPanel newPanel;
      for(int i=0;i<Duties.size();i++){
        newPanel=createUnplanedDutyPanel((Duty)Duties.get(i),130);
        hoehe+=newPanel.getPreferredSize().getHeight()+1;
        returnPanel.add(newPanel);
      }
    }
    returnPanel.setPreferredSize(new Dimension(130,hoehe));
    return returnPanel;
  }

  public JPanel createVisitsPanel(Vector Visits){
    AssociatedJPanel returnPanel=new AssociatedJPanel();
    int hoehe=0;
    if(Visits!=null){
      AssociatedJPanel newPanel;
      for(int i=0;i<Visits.size();i++){
        newPanel=createUnplanedVisitPanel((Visit)Visits.get(i),130);
        hoehe+=newPanel.getPreferredSize().getHeight()+1;
        returnPanel.add(newPanel);
      }
    }
    returnPanel.setPreferredSize(new Dimension(130,hoehe));
    return returnPanel;
  }

  public JPanel createHeaderPanelWithTour(Tour newTour){
    JPanel HeaderPanel=new JPanel();
    HeaderPanel.setLayout(new GridLayout(3,1,0,1));
    AssociatedJLabel newLabel;
    newLabel=new AssociatedJLabel(newTour.getName(),newTour);
    newLabel.setForeground(Color.black);
    newLabel.setBackground(new Color(252,171,3));
    newLabel.setOpaque(true);
    HeaderPanel.add(newLabel);
    if(newTour.getAssignedDuty()!=null){
      newLabel=new AssociatedJLabel(newTour.getAssignedDuty().toString(),newTour);
      newLabel.setForeground(Color.black);
      newLabel.setBackground(new Color(230,230,0));
    }
    else {
      newLabel=new AssociatedJLabel("",newTour);
      newLabel.setBackground(Color.lightGray);
    }
    newLabel.setOpaque(true);
    HeaderPanel.add(newLabel);
    newLabel=new AssociatedJLabel("K - XX 100",newTour);
    newLabel.setBackground(new Color(230,230,230));
    newLabel.setForeground(Color.black);
    newLabel.setOpaque(true);
    HeaderPanel.add(newLabel);
    HeaderPanel.setPreferredSize(new Dimension(100,100));
    return HeaderPanel;
  }

  public JPanel createBodyPanelWithTour(Tour newTour){
    JPanel tourPanel=new AutoScrollJPanel();

    tourPanel.setAutoscrolls(true);
```

```
    tourPanel.setBackground(Color.black);
    Vector allVisits=newTour.getVisits();


    Time zeit=new Time(newTour.getStartTime());

    if(zeit.isGreater(dayStart)){
      AssociatedJPanel startPanel=new AssociatedJPanel();
      startPanel.setPreferredSize(new Dimension(100,(int)(((zeit.getTime()-
dayStart.getTime())/(int)tourZoom)-1)));
      tourPanel.add(startPanel);
    }

    //Time zeit=new Time(newTour.getStartTime());
    AssociatedJLabel newLabel;
    PlanedItem freeItem;
    String ttText;
    for(int i=0;i<allVisits.size();i++){
      PlanedItem tempItem=(PlanedItem)allVisits.get(i);
      if(tempItem.getPlanedTime().isGreater(zeit)){
        newLabel=new AssociatedJLabel("Freie Zeit",new PlanedVisitData(newTour,new
PlanedItem(null,new Time(zeit))));
        newLabel.setPreferredSize(new
Dimension(100,(int)(((tempItem.getPlanedTime().getTime()-
zeit.getTime())/(int)tourZoom)-1)));
        newLabel.setBackground(new Color(100,200,100));
        newLabel.setForeground(Color.black);
        newLabel.setOpaque(true);
        newLabel.setAutoscrolls(true);
        tourPanel.add(newLabel);
      }
      newLabel=new AssociatedJLabel(tempItem.toString(),new
PlanedVisitData(newTour,tempItem));
      newLabel.setPreferredSize(new
Dimension(100,(int)((tempItem.getPlanedVisit().getLength().getTime()/(int)tourZoom)
-1)));
      newLabel.setBackground(new Color(140,160,255));
      newLabel.setForeground(Color.black);
      newLabel.setOpaque(true);
      newLabel.setAutoscrolls(true);

      ttText=tempItem.getPlanedVisit().getStartTime().toString() + '-' +
tempItem.getPlanedVisit().getEndTime().toString();
      if(tempItem.getPlanedVisit().getPreferedTime()!=null){
        ttText=tempItem.getPlanedVisit().getPreferedTime().toString() + " (" +
ttText + ')';
      }
      ttText="Should be planed at: "+ttText;
      newLabel.setToolTipText(ttText);
      tourPanel.add(newLabel);
      zeit=new Time(tempItem.getPlanedTime());
      zeit.addTime(tempItem.getPlanedVisit().getLength());
    }

    if(newTour.getEndTime().isGreater(zeit)){
        newLabel=new AssociatedJLabel("Freie Zeit",new PlanedVisitData(newTour,new
PlanedItem(null,new Time(zeit))));
        newLabel.setPreferredSize(new
Dimension(100,(int)(((newTour.getEndTime().getTime()-
zeit.getTime())/(int)tourZoom)-1)));
        newLabel.setBackground(new Color(100,200,100));
        newLabel.setForeground(Color.black);
        newLabel.setOpaque(true);
        newLabel.setAutoscrolls(true);
        tourPanel.add(newLabel);
    }
```

```
    if(dayEnd.isGreater(zeit)){
      AssociatedJPanel startPanel=new AssociatedJPanel();
      startPanel.setPreferredSize(new Dimension(100,(int)(((dayEnd.getTime()-
zeit.getTime())/(int)tourZoom)-1)));
      tourPanel.add(startPanel);
    }


    //tourPanel.setPreferredSize(new
Dimension(100,(int)(((newTour.getEndTime().getTime()-
newTour.getStartTime().getTime())/(int)tourZoom)+0)));
    tourPanel.setPreferredSize(new Dimension(100,(int)(((dayEnd.getTime()-
dayStart.getTime())/(int)tourZoom)+0)));
    return tourPanel;
  }

  public Visit getVisitFromComponent(Component newComponent){
    return (Visit)((AssociatedComponent)newComponent).getAssociatedObject();
  }

  public PlanedVisitData getPlanedVisitDataFromComponent(Component newComponent){
    return
(PlanedVisitData)((AssociatedComponent)newComponent).getAssociatedObject();
  }

  public Duty getDutyFromComponent(Component newComponent){
    return (Duty)((AssociatedComponent)newComponent).getAssociatedObject();
  }

  public Tour getTourFromHeaderComponent(Component newComponent){
    return (Tour)((AssociatedComponent)newComponent).getAssociatedObject();
  }

  public PlanedDutyData getPlanedDutyDataFromHeaderComponent(Component
newComponent){
    return
(PlanedDutyData)((AssociatedComponent)newComponent).getAssociatedObject();
  }

}
```

9.3.31. Klasse „Time“

```
import java.io.*;
import java.text.*;
import java.util.*;

/**
 * Diese Klasse stellt einen Zeitpunkt an einem Tag, bzw. eine
 * Dauer dar. Den wert den man mit getTime() erhält kann man
 * direkt einem java.util.Date hinzuadieren.
 */
public class Time implements Serializable{

  final public static long SECOND=1000;
  final public static long MINUTE=SECOND*60;
  final public static long HOUR=MINUTE*60;

  private static long corrTimeFormat=-HOUR;

  private long time=0;
  private long hour=0;
  private long minute=0;
  private long second=0;

  private static DateFormat df=DateFormat.getTimeInstance(DateFormat.SHORT);
```

```
public Time(){
}

public Time(long hour,long minute,long second){
  time=hour*HOUR+minute*MINUTE+second*SECOND;
  this.hour=hour;
  this.minute=minute;
  this.second=second;
}

public Time(Time newTime){
  time=newTime.getTime();
}

public Time(long newTime){
  time=newTime;
}

public Time(java.sql.Time newTime){
  time=newTime.getTime()-corrTimeFormat;
}

public void setTime(long hour,long minute,long second){
  time=hour*HOUR+minute*MINUTE+second*SECOND;
  this.hour=hour;
  this.minute=minute;
  this.second=second;
}

public long getTime(){
  return time;
}

public String toString()
{
  return df.format(new Date(time+corrTimeFormat));
}

public boolean equals(Object obj)
{
  Time vglTime=(Time) obj;
  return vglTime.getTime()==time;
}

public void addTime(Time addedTime){
  time+=addedTime.getTime();
}

public void subtractTime(Time addedTime){
  time-=addedTime.getTime();
}

public void addTime(long addedTime){
  time+=addedTime;
}

public void subtractTime(long addedTime){
  time-=addedTime;
}

public void addTime(long hour,long minute,long second){
  time+=(hour*HOUR+minute*MINUTE+second*SECOND);
}

public void subtractTime(long hour,long minute,long second){
  time-=(hour*HOUR+minute*MINUTE+second*SECOND);
}
```

```
  public boolean isGreater(Time vglTime){
    return time>vglTime.getTime();
  }

  static public Time parse(String parseTime){
    try {
      return new Time(df.parse(parseTime).getTime()-corrTimeFormat);
    } catch(Exception e) {
      return null;
    }
  }

  protected Object clone() throws CloneNotSupportedException
  {
    return new Time(time);
  }
}
```

9.3.32. Klasse „Tour“

```
import java.util.*;
import java.text.*;
import java.io.*;

/**
 * Repräsentiert das Datenobjekt Tour.
 */
public class Tour implements Serializable{

  private Time StartTime;
  private Time EndTime;
  private Duty AssignedDuty;
  private String Name;

  private Vector PlanedItems;

  public Tour(){
    PlanedItems=new Vector();
  }

  /**
   * Fügt einer Tour einen Besuch zu indem er an einen festen Zeitpunkt gesetzt
wird. Daruffolgende
   * Besuche werden nach hinten verschoben falls nötig. Der Bescuh wird weiter
hinten geplant,
   * wenn sich der Zeitpunkt innerhalb eines schon geplanten Besuchs befindet.
   */
  public void addVisit(Visit newVisit,Time Time){
    if(StartTime.isGreater(Time)){
      Time=new Time(StartTime);
    }
    else {
      Time vorraussichtlichEndzeit=new Time(Time);
      vorraussichtlichEndzeit.addTime(newVisit.getLength());
      if(vorraussichtlichEndzeit.isGreater(EndTime)){
        Time=new Time(EndTime);
        Time.subtractTime(newVisit.getLength());
      }
    }
    Time BesuchsLaenge=newVisit.getLength();
    Time vglTime=new Time();
    PlanedItem newItem=new PlanedItem(newVisit,Time);
    int ItemPosition=0;
    boolean NotPlaned=true;
```

```
    for(ItemPosition=0;ItemPosition<PlanedItems.size() &&
NotPlaned;ItemPosition++){

  if(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime().isGreater(Time)){
        NotPlaned=false;
        ItemPosition--;
      }
    }

    if(!NotPlaned){
      vglTime=new
Time(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime());

  vglTime.addTime(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedVisit().getL
ength());
      if(!Time.isGreater(vglTime) &&
Time.isGreater(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime())){
        newItem.setPlanedTime(vglTime);
        Time=new Time(vglTime);
        ++ItemPosition;
      }
      else {
        if(ItemPosition>0){
          vglTime=new Time(((PlanedItem)PlanedItems.get(ItemPosition-
1)).getPlanedTime());
          vglTime.addTime(((PlanedItem)PlanedItems.get(ItemPosition-
1)).getPlanedVisit().getLength());
          if(!Time.isGreater(vglTime)){
            newItem.setPlanedTime(vglTime);
            Time=new Time(vglTime);
            //++ItemPosition;
          }
        }
      }
    }
    else {
      if(PlanedItems.size()>0){
        vglTime=new Time(((PlanedItem)PlanedItems.lastElement()).getPlanedTime());

  vglTime.addTime(((PlanedItem)PlanedItems.lastElement()).getPlanedVisit().getLengt
h());
        if(!Time.isGreater(vglTime)){
          newItem.setPlanedTime(vglTime);
          Time=new Time(vglTime);
          ++ItemPosition;
        }
      }
    }

    if(ItemPosition>PlanedItems.size())
      PlanedItems.add(newItem);
    else
      PlanedItems.add(ItemPosition,newItem);

    Time endzeit=new Time(BesuchsLaenge);
    endzeit.addTime(Time);
    for(ItemPosition++;ItemPosition<PlanedItems.size();ItemPosition++){

  if(endzeit.isGreater(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime())
){
        ((PlanedItem)PlanedItems.get(ItemPosition)).setPlanedTime(new
Time(endzeit));

  endzeit.addTime(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedVisit().getL
ength());
      }
    }
```

```
    System.out.println("Tour: PlanedItem added [" + newItem + "]");
  }


  /**
   * Fügt einer Tour einen Besuch zu indem er an die stelle gesetzt wird, an der
   * Das übergebene PlandItem zur zeit steht. Dieses PlanedItem wird nach hinten
verschoben.
   */
  public void addVisit(Visit newVisit,PlanedItem planedVisit){
    DateFormat df=DateFormat.getTimeInstance(DateFormat.SHORT);
    if(planedVisit.getPlanedVisit()!=null){
      //Planung auf einen vorhandenen Visit

      Time Time;
      Time BesuchsLaenge=newVisit.getLength();
      Time=new Time(planedVisit.getPlanedTime());
      PlanedItem newItem=new PlanedItem(newVisit,Time);
      //jetzt alle nachfolgenden Besuche nach hinten schieben
      int ItemPosition=0;
      int insertPosition=0;
      boolean NotPlaned=true;
      for(ItemPosition=0;ItemPosition<PlanedItems.size() &&
NotPlaned;ItemPosition++){

  if(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime().equals(Time)){
          NotPlaned=false;
          ItemPosition--;
        }
      }
      insertPosition=ItemPosition;

      Time endzeit=new Time(BesuchsLaenge);
      endzeit.addTime(Time);
      Time newEndzeit=new Time(endzeit);

  endzeit.addTime(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedVisit().getL
ength());
      for(ItemPosition++;ItemPosition<PlanedItems.size();ItemPosition++){

  if(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime().isGreater(endzeit)
){
          ((PlanedItem)PlanedItems.get(ItemPosition)).setPlanedTime(new
Time(endzeit));

  endzeit.addTime(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedVisit().getL
ength());
        }
      }
      ((PlanedItem)PlanedItems.get(insertPosition)).setPlanedTime(newEndzeit);
      PlanedItems.add(insertPosition,newItem);
      System.out.println("Tour: PlanedItem added [" + newItem + "] from
PlanedItem");
    }
    else {
      //Planung in freier Zeit
      Time Time=new Time(planedVisit.getPlanedTime());
      if(StartTime.isGreater(Time)){
        Time=new Time(StartTime);
      }
      else {
        Time vorraussichtlichEndzeit=new Time(Time);
        vorraussichtlichEndzeit.addTime(newVisit.getLength());
        if(vorraussichtlichEndzeit.isGreater(EndTime)){
          Time=new Time(EndTime);
          Time.subtractTime(newVisit.getLength());
        }
      }
```

```
      int ItemPosition=0;
      boolean NotPlaned=true;
      Time maxtime=new Time(EndTime);
      for(ItemPosition=0;ItemPosition<PlanedItems.size() &&
NotPlaned;ItemPosition++){

  if(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime().isGreater(Time)){
          maxtime=new
Time(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime());
          NotPlaned=false;
          ItemPosition--;
        }
      }
      Time plantime=new Time(Time);
      if(newVisit.getPreferedTime()!=null){
        if(newVisit.getPreferedTime().isGreater(Time)){

  if(newVisit.getPreferedTime().getTime()+newVisit.getLength().getTime()<maxtime.ge
tTime()){
            plantime=new Time(newVisit.getPreferedTime());
          }
          else {
            plantime=new Time(maxtime);
            plantime.subtractTime(newVisit.getLength());
            if(Time.isGreater(plantime)){
              plantime=new Time(Time);
            }
          }
        }
        else {
          plantime=new Time(Time);
        }
      }
      else {
        if(newVisit.getStartTime().isGreater(Time)){

  if(newVisit.getStartTime().getTime()+newVisit.getLength().getTime()<maxtime.getTi
me()){
            plantime=new Time(newVisit.getStartTime());
          }
          else {
            plantime=new Time(maxtime);
            plantime.subtractTime(newVisit.getLength());
            if(Time.isGreater(plantime)){
              plantime=new Time(Time);
            }
          }
        }
        else {
          plantime=new Time(Time);
        }
      }
      PlanedItem newItem=new PlanedItem(newVisit,plantime);
      PlanedItems.add(ItemPosition,newItem);

      Time endzeit=new Time(plantime);
      endzeit.addTime(newVisit.getLength());
      for(ItemPosition++;ItemPosition<PlanedItems.size();ItemPosition++){

  if(endzeit.isGreater(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedTime())
){
          ((PlanedItem)PlanedItems.get(ItemPosition)).setPlanedTime(new
Time(endzeit));

  endzeit.addTime(((PlanedItem)PlanedItems.get(ItemPosition)).getPlanedVisit().getL
ength());
        }
      }
```

```
      System.out.println("Tour: PlanedItem added [" + newItem + "] from Time");
    }
  }

  public Time getStartTime()
  {
    return this.StartTime;
  }

  public void setStartTime(Time StartTime)
  {
    this.StartTime = StartTime;
  }

  public Time getEndTime()
  {
    return this.EndTime;
  }

  public void setEndTime(Time EndTime)
  {
    this.EndTime = EndTime;
  }

  public Duty getAssignedDuty()
  {
    return this.AssignedDuty;
  }

  public void setAssignedDuty(Duty AssignedDuty)
  {
    this.AssignedDuty = AssignedDuty;
  }

  public String getName()
  {
    return this.Name;
  }

  public void setName(String Name)
  {
    this.Name = Name;
  }

  public String toString(){
    String ergebnis;
    ergebnis="Tour: "+Name+"\n";
    if(AssignedDuty!=null)
      ergebnis+="Duty: "+AssignedDuty+"\n";
    else
      ergebnis+="Duty: null\n";
    ergebnis+="Planed Visits:\n";
    Time zeit=new Time(StartTime);
    for(int i=0;i<PlanedItems.size();i++){
      PlanedItem tempItem=(PlanedItem)PlanedItems.get(i);
      if(tempItem.getPlanedTime().isGreater(zeit)){
        ergebnis+="   Freie Zeit\n";
      }
      ergebnis+=(i+1)+". "+tempItem+"\n";
      zeit=new Time(tempItem.getPlanedTime());
      zeit.addTime(((Visit)tempItem.getPlanedVisit()).getLength());
    }
    if(EndTime.isGreater(zeit)){
      ergebnis+="   Freie Zeit\n";
    }
    return ergebnis;
  }
```

```
  public Vector getVisits(){
    return PlanedItems;
  }

  public void removePlanedItem(PlanedItem delVisit){
    for(int i=0;i<PlanedItems.size();i++){
      if(delVisit.equals(PlanedItems.get(i))){
        PlanedItems.removeElementAt(i);
        i=PlanedItems.size();
      }
    }
  }

  public void removeVisit(Visit delVisit){
    for(int i=0;i<PlanedItems.size();i++){
      if(delVisit.equals(((PlanedItem)PlanedItems.get(i)).getPlanedVisit())){
        PlanedItems.removeElementAt(i);
        i=PlanedItems.size();
      }
    }
  }

  public boolean equals(Object obj)
  {
    Tour vglTour=(Tour) obj;
    if(vglTour.getAssignedDuty()!=null){
      if(AssignedDuty!=null)
        return vglTour.getStartTime().equals(StartTime)
          && vglTour.getEndTime().equals(EndTime)
          && vglTour.getAssignedDuty().equals(AssignedDuty)
          && vglTour.getName().equals(Name)
          && vglTour.getVisits().equals(PlanedItems);
      else
        return false;
    }
    else {
      if(AssignedDuty!=null)
        return false;
      else
        return vglTour.getStartTime().equals(StartTime)
          && vglTour.getEndTime().equals(EndTime)
          && vglTour.getName().equals(Name)
          && vglTour.getVisits().equals(PlanedItems);
    }
  }
}
```

9.3.33. Abstrakte Klasse „TourViewFactory“

```
import java.util.*;
import javax.swing.*;
import java.awt.*;

/**
 * Mit dieser abstrakten Klasse wird die Tourview erzeugt. Sie enthält auch Daten
 * um über Komponenten an Datenobjekte zu gelangen. Für informationen hierüber
 * sieht man sich am besten die Konkrete implementierung an.
 */
public abstract class TourViewFactory{
  public TourViewFactory(){
  }

  public JPanel createVerticalTimeBar(Time StartTime,Time EndTime){
    return new JPanel();
  }
```

```
  public JPanel createVisitsPanel(Vector Visits){
    return new JPanel();
  }

  public JPanel createDutiesPanel(Vector Duties){
    return new JPanel();
  }

  public JPanel createHeaderPanelWithTour(Tour newTour){
    return new JPanel();
  }
  public JPanel createBodyPanelWithTour(Tour newTour){
    return new JPanel();
  }

  public Visit getVisitFromComponent(Component newComponent){
    return null;
  }

  public Duty getDutyFromComponent(Component newComponent){
    return null;
  }

  public PlanedVisitData getPlanedVisitDataFromComponent(Component newComponent){
    return null;
  }

  public Tour getTourFromHeaderComponent(Component newComponent){
    return null;
  }

  public PlanedDutyData getPlanedDutyDataFromHeaderComponent(Component
newComponent){
    return null;
  }

}
```

9.3.34. Klasse „TransferableDuty“

```
import java.util.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;

/**
 * Ein Onjekt dieser Klasse wird bei Drag and Drop Operationen von ungeplanten
 * Schichten erzeugt.
 */
public class TransferableDuty implements Transferable{
  final static int DUTY = 0;

  final public static DataFlavor DUTY_FLAVOR =
  new DataFlavor(Duty.class, "DUTY");

  private Duty data;

  public TransferableDuty(Duty data) {
  this.data = data;
  }

  static DataFlavor flavors[] = {DUTY_FLAVOR};

  /**
   * Returns an array of DataFlavor objects indicating the flavors the data
   * can be provided in.  The array should be ordered according to preference
```

```
    * for providing the data (from most richly descriptive to least descriptive).
    * @return an array of data flavors in which this data can be transferred
    */
   public DataFlavor[] getTransferDataFlavors(){
     return flavors;
   }

   /**
    * Returns whether or not the specified data flavor is supported for
    * this object.
    * @param flavor the requested flavor for the data
    * @return boolean indicating wjether or not the data flavor is supported
    */
   public boolean isDataFlavorSupported(DataFlavor flavor){
     return flavor.equals(flavors[DUTY]);
   }

   /**
    * Returns an object which represents the data to be transferred.  The class
    * of the object returned is defined by the representation class of the flavor.
    *
    * @param flavor the requested flavor for the data
    * @see DataFlavor#getRepresentationClass
    * @exception IOException                 if the data is no longer available
    *              in the requested flavor.
    * @exception UnsupportedFlavorException if the requested data flavor is
    *              not supported.
    */
   public Object getTransferData(DataFlavor flavor)
          throws UnsupportedFlavorException, IOException{
     if (flavor.equals(flavors[DUTY])) {
     return data;
   } else {
     UnsupportedFlavorException t=new UnsupportedFlavorException(flavor);
     throw t;
   }
   }
}
```

9.3.35. Klasse „TransferablePlanedDutyData“

```
import java.util.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;

/**
 * Ein Onjekt dieser Klasse wird bei Drag and Drop Operationen von geplanten
 * Schichten erzeugt.
 */
public class TransferablePlanedDutyData implements Transferable{
  final static int PLANED_DUTY_DATA = 0;

  final public static DataFlavor PLANED_DUTY_DATA_FLAVOR =
  new DataFlavor(PlanedDutyData.class, "PlanedDutyData");

  private PlanedDutyData data;

  public TransferablePlanedDutyData(PlanedDutyData data) {
  this.data = data;
  }

  static DataFlavor flavors[] = {PLANED_DUTY_DATA_FLAVOR};

  /**
   * Returns an array of DataFlavor objects indicating the flavors the data
```

```
    * can be provided in.  The array should be ordered according to preference
    * for providing the data (from most richly descriptive to least descriptive).
    * @return an array of data flavors in which this data can be transferred
    */
   public DataFlavor[] getTransferDataFlavors(){
     return flavors;
   }

   /**
    * Returns whether or not the specified data flavor is supported for
    * this object.
    * @param flavor the requested flavor for the data
    * @return boolean indicating wjether or not the data flavor is supported
    */
   public boolean isDataFlavorSupported(DataFlavor flavor){
     return flavor.equals(flavors[PLANED_DUTY_DATA]);
   }

   /**
    * Returns an object which represents the data to be transferred.  The class
    * of the object returned is defined by the representation class of the flavor.
    *
    * @param flavor the requested flavor for the data
    * @see DataFlavor#getRepresentationClass
    * @exception IOException                if the data is no longer available
    *              in the requested flavor.
    * @exception UnsupportedFlavorException if the requested data flavor is
    *              not supported.
    */
   public Object getTransferData(DataFlavor flavor)
          throws UnsupportedFlavorException, IOException{
     if (flavor.equals(flavors[PLANED_DUTY_DATA])) {
     return data;
   } else {
     UnsupportedFlavorException t=new UnsupportedFlavorException(flavor);
     throw t;
   }
   }
}
```

9.3.36. Klasse „TransferablePlanedVisitData“

```
import java.util.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;

/**
 * Ein Onjekt dieser Klasse wird bei Drag and Drop Operationen von geplanten
 * Besuchen erzeugt.
 */
public class TransferablePlanedVisitData implements Transferable{
  final static int PLANED_VISIT_DATA = 0;

  final public static DataFlavor PLANED_VISIT_DATA_FLAVOR =
  new DataFlavor(PlanedVisitData.class, "PlanedVisitData");

  private PlanedVisitData data;

  public TransferablePlanedVisitData(PlanedVisitData data) {
  this.data = data;
  }

  static DataFlavor flavors[] = {PLANED_VISIT_DATA_FLAVOR};

  /**
```

```
    * Returns an array of DataFlavor objects indicating the flavors the data
    * can be provided in.  The array should be ordered according to preference
    * for providing the data (from most richly descriptive to least descriptive).
    * @return an array of data flavors in which this data can be transferred
    */
   public DataFlavor[] getTransferDataFlavors(){
     return flavors;
   }

   /**
    * Returns whether or not the specified data flavor is supported for
    * this object.
    * @param flavor the requested flavor for the data
    * @return boolean indicating wjether or not the data flavor is supported
    */
   public boolean isDataFlavorSupported(DataFlavor flavor){
     return flavor.equals(flavors[PLANED_VISIT_DATA]);
   }

   /**
    * Returns an object which represents the data to be transferred.  The class
    * of the object returned is defined by the representation class of the flavor.
    *
    * @param flavor the requested flavor for the data
    * @see DataFlavor#getRepresentationClass
    * @exception IOException                if the data is no longer available
    *              in the requested flavor.
    * @exception UnsupportedFlavorException if the requested data flavor is
    *              not supported.
    */
   public Object getTransferData(DataFlavor flavor)
          throws UnsupportedFlavorException, IOException{
     if (flavor.equals(flavors[PLANED_VISIT_DATA])) {
     return data;
   } else {
     UnsupportedFlavorException t=new UnsupportedFlavorException(flavor);
     throw t;
   }
   }
}
```

9.3.37. Klasse „TransferableVisit“

```
import java.util.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;

/**
 * Ein Onjekt dieser Klasse wird bei Drag and Drop Operationen von ungeplanten
 * Besuchen erzeugt.
 */
public class TransferableVisit implements Transferable{
  final static int VISIT = 0;

  final public static DataFlavor VISIT_FLAVOR =
  new DataFlavor(Visit.class, "Visit");

  private Visit data;

  public TransferableVisit(Visit data) {
  this.data = data;
  }

  static DataFlavor flavors[] = {VISIT_FLAVOR};
```

```
  /**
   * Returns an array of DataFlavor objects indicating the flavors the data
   * can be provided in.  The array should be ordered according to preference
   * for providing the data (from most richly descriptive to least descriptive).
   * @return an array of data flavors in which this data can be transferred
   */
  public DataFlavor[] getTransferDataFlavors(){
    return flavors;
  }

  /**
   * Returns whether or not the specified data flavor is supported for
   * this object.
   * @param flavor the requested flavor for the data
   * @return boolean indicating wjether or not the data flavor is supported
   */
  public boolean isDataFlavorSupported(DataFlavor flavor){
    return flavor.equals(flavors[VISIT]);
  }

  /**
   * Returns an object which represents the data to be transferred.  The class
   * of the object returned is defined by the representation class of the flavor.
   *
   * @param flavor the requested flavor for the data
   * @see DataFlavor#getRepresentationClass
   * @exception IOException                if the data is no longer available
   *              in the requested flavor.
   * @exception UnsupportedFlavorException if the requested data flavor is
   *              not supported.
   */
  public Object getTransferData(DataFlavor flavor)
         throws UnsupportedFlavorException, IOException{
    if (flavor.equals(flavors[VISIT])) {
    return data;
  } else {
    UnsupportedFlavorException t=new UnsupportedFlavorException(flavor);
    throw t;
  }
  }
}
```

9.3.38. Klasse „UnassignedDutiesDragAndDrop“

```
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

/** Diese Klasse übernimmt alle Aufgaben die für Dragt and Drop
 * Operationen von ungeplanten Schichten nötig sind.
 */
public class UnassignedDutiesDragAndDrop
  implements DragGestureListener,DragSourceListener,DropTargetListener {

  private TourViewFactory usedFactory;
  private CarePlanModel usedModel;

  public UnassignedDutiesDragAndDrop(){
    this.usedFactory=new SimpleTourViewFactory();
  }

  public UnassignedDutiesDragAndDrop(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }
```

```
  public void setTourViewFactory(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  public void setCarePlanModel(CarePlanModel usedModel){
    this.usedModel=usedModel;
  }

  //DragGestureListener
  public void dragGestureRecognized(DragGestureEvent dge){
    dge.getDragSource().startDrag(dge, DragSource.DefaultMoveDrop, new
TransferableDuty(usedFactory.getDutyFromComponent(dge.getComponent())), this);
  }

  //DragSourceListener
  public void dragDropEnd(DragSourceDropEvent DragSourceDropEvent) {
  }

  public void dragEnter(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dragExit(DragSourceEvent DragSourceEvent) {
  }
  public void dragOver(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dropActionChanged(DragSourceDragEvent DragSourceDragEvent) {
  }

  // interface DropTargetListener
  public void dragEnter (DropTargetDragEvent dropTargetDragEvent) {
    if(dropTargetDragEvent.isDataFlavorSupported(TransferableDuty.DUTY_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedDutyData.PLANED_DUT
Y_DATA_FLAVOR))
    dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void dragExit (DropTargetEvent dropTargetEvent) {
  }

  public void dragOver (DropTargetDragEvent dropTargetDragEvent) {
    if(dropTargetDragEvent.isDataFlavorSupported(TransferableDuty.DUTY_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedDutyData.PLANED_DUT
Y_DATA_FLAVOR))
      dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void dropActionChanged (DropTargetDragEvent dropTargetDragEvent) {
    if(dropTargetDragEvent.isDataFlavorSupported(TransferableDuty.DUTY_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedDutyData.PLANED_DUT
Y_DATA_FLAVOR))
      dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void drop (DropTargetDropEvent dropTargetDropEvent) {
    dropTargetDropEvent.acceptDrop (DnDConstants.ACTION_MOVE);
    try{
      if(dropTargetDropEvent.isDataFlavorSupported(TransferableDuty.DUTY_FLAVOR)){
        Tour
dropTour=usedFactory.getTourFromHeaderComponent(dropTargetDropEvent.getDropTargetCo
ntext().getDropTarget().getComponent());
```

```
          Duty
transferDuty=(Duty)dropTargetDropEvent.getTransferable().getTransferData(Transferab
leDuty.DUTY_FLAVOR);
          usedModel.assignDutyToTour(transferDuty,dropTour);

        }
        else {

  if(dropTargetDropEvent.isDataFlavorSupported(TransferablePlanedDutyData.PLANED_DU
TY_DATA_FLAVOR)){
            Tour
dropTour=usedFactory.getTourFromHeaderComponent(dropTargetDropEvent.getDropTargetCo
ntext().getDropTarget().getComponent());
            PlanedDutyData
transferDuty=(PlanedDutyData)dropTargetDropEvent.getTransferable().getTransferData(
TransferablePlanedDutyData.PLANED_DUTY_DATA_FLAVOR);
            if(!transferDuty.getTour().equals(dropTour)){
              usedModel.assignDutyToTour(transferDuty.getDuty(),dropTour);

usedModel.unassignDutyFromTour(transferDuty.getDuty(),transferDuty.getTour());
            }
            else {
              System.out.println("Sourcetour equals Targettour!");
            }
          }
          else {
            System.out.println("Unknown flavor!");
          }
        }

      } catch (Exception e){
      }

      dropTargetDropEvent.dropComplete(true);

      //Dieser umständliche Aufruf von 'usedModel.CarePlanModelChanged(null,1)' muß
sein
      //da sich sonst JAVA aufhängt!
      SwingUtilities.invokeLater(new Runnable() {
          public void run() {
             usedModel.CarePlanModelChanged(null,1);
          }
        }
      );
    }

}
```

9.3.39. Klasse „UnplanedVisitsDragAndDrop“

```
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

/** Diese Klasse übernimmt alle Aufgaben die für Dragt and Drop
 * Operationen von ungeplanten Besuchen nötig sind.
 */
public class UnplanedVisitsDragAndDrop
  implements DragGestureListener,DragSourceListener,DropTargetListener {

  private TourViewFactory usedFactory;
  private CarePlanModel usedModel;

  public UnplanedVisitsDragAndDrop(){
    this.usedFactory=new SimpleTourViewFactory();
```

```
  }

  public UnplanedVisitsDragAndDrop(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  public void setTourViewFactory(TourViewFactory usedFactory){
    this.usedFactory=usedFactory;
  }

  public void setCarePlanModel(CarePlanModel usedModel){
    this.usedModel=usedModel;
  }

  //DragGestureListener
  public void dragGestureRecognized(DragGestureEvent dge){
    dge.getDragSource().startDrag(dge, DragSource.DefaultMoveDrop, new
TransferableVisit(usedFactory.getVisitFromComponent(dge.getComponent())), this);
  }

  //DragSourceListener
  public void dragDropEnd(DragSourceDropEvent DragSourceDropEvent) {
  }

  public void dragEnter(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dragExit(DragSourceEvent DragSourceEvent) {
  }
  public void dragOver(DragSourceDragEvent DragSourceDragEvent) {
  }
  public void dropActionChanged(DragSourceDragEvent DragSourceDragEvent) {
  }

  // interface DropTargetListener
  public void dragEnter (DropTargetDragEvent dropTargetDragEvent) {

  if(usedFactory.getPlanedVisitDataFromComponent(dropTargetDragEvent.getDropTargetC
ontext().getDropTarget().getComponent())!=null)
       if(dropTargetDragEvent.isDataFlavorSupported(TransferableVisit.VISIT_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedVisitData.PLANED_VI
SIT_DATA_FLAVOR))
      dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
      else
       dropTargetDragEvent.rejectDrag();
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void dragExit (DropTargetEvent dropTargetEvent) {
  }

  public void dragOver (DropTargetDragEvent dropTargetDragEvent) {

  if(usedFactory.getPlanedVisitDataFromComponent(dropTargetDragEvent.getDropTargetC
ontext().getDropTarget().getComponent())!=null)
       if(dropTargetDragEvent.isDataFlavorSupported(TransferableVisit.VISIT_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedVisitData.PLANED_VI
SIT_DATA_FLAVOR))
      dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
      else
       dropTargetDragEvent.rejectDrag();
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void dropActionChanged (DropTargetDragEvent dropTargetDragEvent) {
```

```
  if(usedFactory.getPlanedVisitDataFromComponent(dropTargetDragEvent.getDropTargetC
ontext().getDropTarget().getComponent())!=null)
       if(dropTargetDragEvent.isDataFlavorSupported(TransferableVisit.VISIT_FLAVOR)

  ||dropTargetDragEvent.isDataFlavorSupported(TransferablePlanedVisitData.PLANED_VI
SIT_DATA_FLAVOR))
      dropTargetDragEvent.acceptDrag (DnDConstants.ACTION_MOVE);
      else
       dropTargetDragEvent.rejectDrag();
    else
    dropTargetDragEvent.rejectDrag();
  }

  public void drop (DropTargetDropEvent dropTargetDropEvent) {
    dropTargetDropEvent.acceptDrop (DnDConstants.ACTION_MOVE);
    try{

  if(dropTargetDropEvent.isDataFlavorSupported(TransferableVisit.VISIT_FLAVOR)){
        PlanedVisitData
dropVisit=usedFactory.getPlanedVisitDataFromComponent(dropTargetDropEvent.getDropTa
rgetContext().getDropTarget().getComponent());
        Visit
transferVisit=(Visit)dropTargetDropEvent.getTransferable().getTransferData(Transfer
ableVisit.VISIT_FLAVOR);


usedModel.assignVisitToTour(transferVisit,dropVisit.getPlanedItem(),dropVisit.getTo
ur());

      }
      else {

  if(dropTargetDropEvent.isDataFlavorSupported(TransferablePlanedVisitData.PLANED_V
ISIT_DATA_FLAVOR)){
          PlanedVisitData
dropVisit=usedFactory.getPlanedVisitDataFromComponent(dropTargetDropEvent.getDropTa
rgetContext().getDropTarget().getComponent());
          PlanedVisitData
transferVisit=(PlanedVisitData)dropTargetDropEvent.getTransferable().getTransferDat
a(TransferablePlanedVisitData.PLANED_VISIT_DATA_FLAVOR);
          if(!dropVisit.equals(transferVisit)){

  usedModel.removeVisitFromTour(transferVisit.getPlanedItem(),transferVisit.getTour
());

  usedModel.assignVisitToTour(transferVisit.getPlanedItem().getPlanedVisit(),dropVi
sit.getPlanedItem(),dropVisit.getTour());
          }
          else {
            System.out.println("Source equals Target!");
          }
        }
        else {
          System.out.println("Unknown flavor!");
        }
      }

    } catch (Exception e){
    }

    dropTargetDropEvent.dropComplete(true);

    //Dieser umständliche Aufruf von 'usedModel.CarePlanModelChanged(null,1)' muß
sein
    //da sich sonst JAVA aufhängt!
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
```

```
            usedModel.CarePlanModelChanged(null,1);
          }
        }
      );
  }

}
```

9.3.40. Klasse „Visit“

```
import java.awt.datatransfer.*;
import java.io.IOException;
import java.io.*;

/**
 * Enthält alle Daten die einen Besuch ausmachen.
 */
public class Visit implements Serializable{

  private PlanableCareClient careClient;
  private Time Length;
  private int Qualification;
  private Time StartTime;
  private Time EndTime;
  private Time PreferedTime;
  private Long iD;

  public Visit(){
    this.careClient = null;
    this.Length = null;
    this.Qualification = 0;
    this.StartTime = null;
    this.EndTime = null;
    this.PreferedTime = null;
    this.iD=null;
  }

  public Visit(PlanableCareClient careClient,Time Length,int Qualification,Time
StartTime,Time EndTime,Time PreferedTime, Long iD){
    this.careClient = careClient;
    this.Length = Length;
    this.Qualification = Qualification;
    this.StartTime = StartTime;
    this.EndTime = EndTime;
    this.PreferedTime = PreferedTime;
    this.iD=iD;
  }

  public PlanableCareClient getCareClient()
  {
    return this.careClient;
  }

  public void setCareClient(PlanableCareClient careClient)
  {
    this.careClient = careClient;
  }

  public Time getLength()
  {
    return this.Length;
  }

  public void setLength(Time Length)
  {
    this.Length = Length;
```

```
  }

  public int getQualification()
  {
    return this.Qualification;
  }

  public void setQualification(int Qualification)
  {
    this.Qualification = Qualification;
  }

  public Time getStartTime()
  {
    return this.StartTime;
  }

  public void setStartTime(Time StartTime)
  {
    this.StartTime = StartTime;
  }

  public Time getEndTime()
  {
    return this.EndTime;
  }

  public void setEndTime(Time EndTime)
  {
    this.EndTime = EndTime;
  }

  public Time getPreferedTime()
  {
    return this.PreferedTime;
  }

  public void setPreferedTime(Time PreferedTime)
  {
    this.PreferedTime = PreferedTime;
  }

  public String toString(){
    return careClient.toString();
  }


  public boolean equals(Object obj)
  {
    Visit vis=(Visit)obj;
    if(vis.getPreferedTime()!=null){
      if(PreferedTime!=null)
        return vis.getCareClient().equals(careClient)
          &&vis.getLength().equals(Length)
          &&vis.getQualification()==Qualification
          &&vis.getStartTime().equals(StartTime)
          &&vis.getEndTime().equals(EndTime)
          &&vis.getPreferedTime().equals(PreferedTime);
      else
        return false;
    }
    else {
      if(PreferedTime==null)
        return vis.getCareClient().equals(careClient)
          &&vis.getLength().equals(Length)
          &&vis.getQualification()==Qualification
          &&vis.getStartTime().equals(StartTime)
          &&vis.getEndTime().equals(EndTime);
```

```
      else
        return false;
    }
  }


  public Long getID()
{
  return this.iD;
}

  public void setID(Long iD)
{
  this.iD = iD;
}
}
```

9.4. Abbildungsverzeichnis

9.5. Quellennachweis

1. „Objektorientierte Softwareentwicklung“ von Bernd Oestereich, Oldenbourg, 4. aktualisierte Auflage.
2. „Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software“ von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, Addison-Wesley, 1996.
3. „Balanced Scorecard – Mehr als ein Kennzahlensystem“ von Herwig R. Freidag und Walter Schmidt, Haufe, 1999
4. „Java in a Nutshell“ von David Flanagan, OReilly/VVA, 1997
5. Im Internet: http://java.sun.com/products/jdk/1.2/docs/index.html , vollständige Dokumentation der Java 2 Plattform in der Version 1.2 inkl. API-Dokumentation und Java-Language-Specifikation.

6. Im Internet: http://java.sun.com/j2ee/docs.html , vollständige Dokumentation der Java 2 Enterprise Edition.
7. SWIFT Project Summary.
8. SWIFT Project Description.
9. Vorlesungsscript „Datenbanken und Informationssysteme“ von Prof. Dr. Heide Faeskorn-Woyke.