Konzeption und Realisierung eines Terminplaners für ein radiologisches Informationssystem (RIS) unter Berücksichtigung softwareergonomischer Aspekte

Diplomarbeit im Studiengang Informatik an der Universität Ulm

Vorgelegt von

Hariolf Häfele

28. Mai 2002

Inhaltsverzeichnis

1 Einführung **1**

2 Bisherige und geforderte Funktionalität **3**
2.1 Funktionalität des bisherigen Terminplaners . . . 3
2.2 Anforderungen an den neuen Terminplaner . . . 3
2.2.1 Mehrfachtermine . . . 4
2.2.2 Geschwindigkeit . . . 4

3 Theorie der Ablaufplanung **7**
3.1 Planungsprobleme . . . 7
3.1.1 Auftragsmodelle . . . 8
3.1.2 Prozessormodelle . . . 9
3.1.3 Optimierungskriterien . . . 11
3.1.4 Beispiele . . . 12
3.1.5 Erweiterungen . . . 13
3.2 Lösung von Planungsproblemen . . . 13
3.3 Mehrere Prozessoren pro Operation . . . 13
3.4 Terminplanung in der Radiologie . . . 14
3.4.1 Das Modell für Serientermine . . . 16
3.4.2 Das Modell für Profiltermine . . . 16

4 Algorithmen zur Terminvergabe **19**
4.1 Der Algorithmus zur Profilterminvergabe . . . 19
4.1.1 Komplexität . . . 22
4.2 Der Algorithmus zur Serienterminvergabe . . . 23

5 Implementierung **25**
5.1 Schichten bei Anwendungsarchitekturen . . . 25
5.1.1 2-Schicht-Modell . . . 25
5.1.2 3-Schicht-Modell . . . 26
5.2 Verwendete Softwareprodukte . . . 27
5.2.1 Java . . . 27
5.2.2 J2EE und EJB . . . 28
5.2.3 Orion . . . 30
5.2.4 Forte . . . 30
5.2.5 SourceSafe . . . 30
5.2.6 Oracle 8.06 . . . 30
5.2.7 JFCSuite . . . 31

5.3 Benutzeroberfläche . . . 31
5.3.1 Steuerungsleiste . . . 32
5.3.2 Monatsübersicht . . . 32
5.3.3 Multifunktionsleiste . . . 32
5.3.4 Weitere Termine / Abwesenheitsliste . . . 33
5.3.5 Terminübersicht . . . 33
5.3.6 Termindialog . . . 34
5.4 Architektur der Benutzerschicht . . . 34
5.5 Architektur der Vorgangsbearbeitung . . . 36
5.5.1 Session-Beans . . . 38
5.5.2 Entity-Beans . . . 46
5.6 Stand des Projektes . . . 48

6 Fazit und Ausblick **49**

Stichwortverzeichnis **51**

Kapitel 1

Einführung

Die Firma *GE Medical Systems Information Technologies*[1] in Dornstadt bei Ulm ist einer der führenden Hersteller von Radiologie-Informationssystemen (*RIS*) in Europa. Das Hauptprodukt der Firma ist das Programmpaket *Medora*, welches eine vollständige RIS-Lösung für radiologische Kliniken und Praxen darstellt.
Ein RIS dient dazu, Radiologiepatienten und Patientenakten, Materiallager, Befundungen sowie Belegungen von Geräten, Arbeitsplätzen und Mitarbeitern zu verwalten.
Eng verwandt mit dem Begriff des RIS ist der des Krankenhaus-Informationssystems (*KIS*). Ein KIS ist eine Software, die zentral alle Patienten des Krankenhauses erfaßt und verwaltet. RIS und KIS kommunizieren über den sogenannten *HL7*-Standard[2], welcher ein Protokoll zur elektronischen Kommunikation im Gesundheitswesen darstellt. Die Kommunikation zwi-

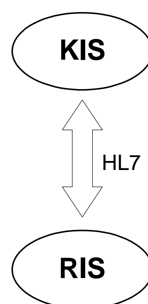

Abbildung 1.1: KIS und RIS

schen KIS und RIS verläuft dabei in beiden Richtungen. Eine typische HL7-Nachricht vom KIS an das RIS wäre etwa die Anforderung einer CT-Aufnahme für einen Patienten an einem bestimmten Tag. Das RIS sendet daraufhin z. B. eine Terminbestätigung an das KIS zurück. HL7-Nachrichten können auch von anderen Krankenhausabteilungen an das RIS gesandt werden.

Das Radiologie-Informationssystem Medora wurde bisher vollständig mit der Programmierumgebung *Oracle Forms* entwickelt[3][4], welches auf die Erstellung von Datenbankanwendungen mit graphischer Oberfläche zugeschnitten ist. Forms ermöglicht es auf relativ einfache

[1] `http://www.gemedical.de/`
[2] `http://www.hl7.org/`
[3] `http://otn.oracle.com/products/forms/content.html`
[4] Zum Zeitpunkt der Drucklegung wurde die Version 4.5 verwendet; es wird im Folgenden, soweit nicht anders angegeben, auf diese Version Bezug genommen

Weise, dialogbasierte Anwendungen zu erstellen, die an eine Datenbank gekoppelt sind. Ein großes Manko von Forms ist allerdings, daß der Entwickler die Oberfläche nur aus einem sehr beschränkten Satz von Komponenten zusammenbauen kann. Daher lassen die damit erzeugten Applikationen in puncto Benutzerfreundlichkeit eher zu wünschen übrig und sind bezüglich Benutzeroberfläche nicht auf dem aktuellen Stand der Technik. So ist es in einer Forms-Applikation für den Benutzer nicht möglich, die Größe von Fenstern oder von Unterkomponenten wie Tabellenspalten zu verändern oder Objekte zu verschieben (*„Drag and Drop“*). Auch gibt es für viele der Elemente einer modernen Benutzeroberfläche wie Kontextmenüs, Baumansichten usw. in Forms keine Entsprechung.

Eine weitere Problematik resultiert aus der Tatsache, daß, um in Forms eine bestimmte Reaktion auf ein *semantisches* Ereignis (Mausklick, Tastatureingabe, Nachricht über Änderung eines Datensatzes, usw.) zu programmieren, im Regelfall eine ganze Reihe von Behandlungsroutinen für sog. *Trigger* bemüht werden müssen. Die Mengen von Triggern, die für semantische Ereignisse benötigt werden, überschneiden sich untereinander, so daß mit steigender Größe des Projekts die Abhängigkeiten zwischen Behandlungsroutinen untereinander zunehmen und es immer schwieriger wird, Änderungen am Code vorzunehmen und dabei unbeabsichtigte Seiteneffekte im Griff zu behalten.
Zu der genannten Unübersichtlichkeit steuert auch die fehlende Objektorientierung von Forms bei, die ebenfalls mit zunehmender Projektgroße immer stärker ins Gewicht fällt.

Aus diesen Gründen beschloß man Mitte 2001, als Technologiestudie – mit Option auf Fortführung als Produkt – ein einzelnes Modul von Medora auf eine Plattform umzustellen, die die erwähnten Mängel nicht aufweisen, aber gleichzeitig eine effektive Datenbankanbindung erlauben würde. Die Plattformwahl fiel auf *Java*[5], da es zum einen auf einfache, aber dennoch flexible Weise gestattet, graphische Benutzeroberflächen zu implementieren und zum anderen durch die *J2EE*-Erweiterung 3-Schicht-Architekturen unterstützt werden, somit also eine Anbindung der objektorientierten Programmiersprache an eine relationale Datenbank gegeben ist.
Als Objekt der Technologiestudie war das Terminplanungsmodul am geeignetsten, da bei ihm erstens die geringste Abhängigkeit zu anderen Medora-Komponenten besteht, es also das am einfachsten austauschbare Modul ist, und zweitens bei der Terminplanung der Bedarf nach einer verbesserten Version am größten war. Außerdem hätte bei einer Forms-Implementierung die erwähnte Zunahme der Unübersichtlichkeit und der Seiteneffekte bei den für die neue Version geplanten Funktionen „Serientermine“ und „Profiltermine“ entsprechenden Mehraufwand bereitet.
Die vorliegende Diplomarbeit beschäftigt sich mit der Konzeption und Realisierung der Technologiestudie „Neue Terminplanung“.

[5] `http://java.sun.com/`

Kapitel 2

Bisherige und geforderte Funktionalität

2.1 Funktionalität des bisherigen Terminplaners

Der bisherige Terminplaner verwaltet Termine auf der Basis von *Terminbüchern*. Ein Terminbuch ist eine Übersicht über die Termine einer *Ressource* (Mitarbeiter oder Gerät). Jeder Ressource ist ein *Terminraster* zugeordnet, welches je nach Tag verschieden sein kann.
Es gibt drei Hauptdialoge, von denen mehrere gleichzeitig geöffnet werden können:

- Tagesübersicht (Stellt ein Terminbuch für eine Ressource an einem Tag dar; siehe Abb. 2.1)
- Jahresübersicht (Stellt einen Jahreskalender für eine Ressource dar; Mausklick auf einen Tag öffnet die Tagesübersicht für diese Ressource; siehe Abb. 2.1)
- Terminauskunft (Suche nach existierenden Terminen)

Zur Anzeige einer Tages- und Jahresübersicht einer Ressource muß der Benutzer diese erst im Ressourcenauswahl-Dialog auswählen (siehe Abb. 2.2). Um einen neuen Termin anzulegen, klickt man in der Tagesübersicht auf eine freie Rasterzeile, worauf sich ein Patientenauswahl-Dialog öffnet (siehe Abb. 2.3). Nachdem der Patient, für den der Termin reserviert werden soll, ausgewählt ist, öffnet sich der eigentliche Terminvergabe-Dialog (Siehe Abb. 2.4).

2.2 Anforderungen an den neuen Terminplaner

Unter Berücksichtigung von Prinzipien modernen GUI-Designs sowie Kommentaren und Wünschen seitens von Kunden wurde ein Pflichtenheft für die neue Version des Terminplaners erarbeitet. Es spezifiziert u. a. folgende Anforderungen:

- Die neue Terminplanung soll die gesamte Funktionalität der alten Version beherrschen
- Die Größe jedes Fensters und abgetrennten Fensterbereichs soll veränderbar sein
- Termine sollen sowohl innerhalb eines Terminbuchs als auch zwischen Terminbüchern verschoben werden können („Drag and Drop“)
- Es sollen *Mehrfachtermine* unterstützt werden (siehe Abschn. 2.2.1)

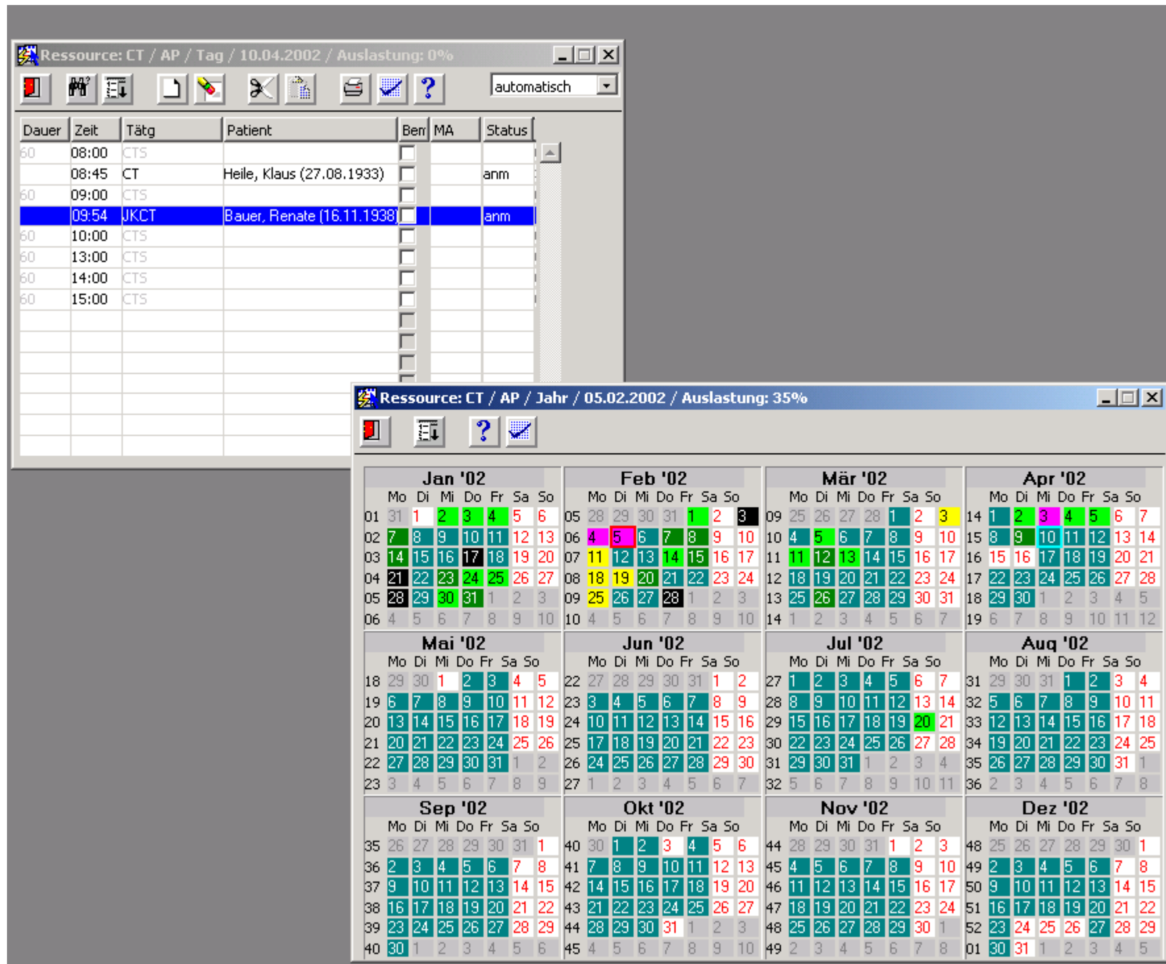

Abbildung 2.1: Jahresübersicht (links oben) und Tagesübersicht (rechts unten)

- Mitarbeiter und Geräte sollen in Mitarbeitergruppen bzw. Gerätegruppen eingeteilt werden können
- Es soll eine Zwischenablage für Termine geben
- Die Zeitdauer für die Ausführung der wichtigsten Operationen soll ein bestimmtes Maß nicht überschreiten (siehe Abschn. 2.2.2)

2.2.1 Mehrfachtermine

Mit der neuen Version des Terminplaners wurden *Mehrfachtermine* eingeführt. Es werden zwei Arten von Mehrfachterminen unterstützt: *Serientermine* und *Profiltermine*. Mehr dazu findet sich in Abschnitt 3.4.

2.2.2 Geschwindigkeit

Ein vorrangiges Kriterium beim neuen Terminkalender war, daß das System für das Eintragen eines neuen Termines und das Ändern eines bereits existierenden Termines nicht länger als eine Sekunde benötigen sollte.

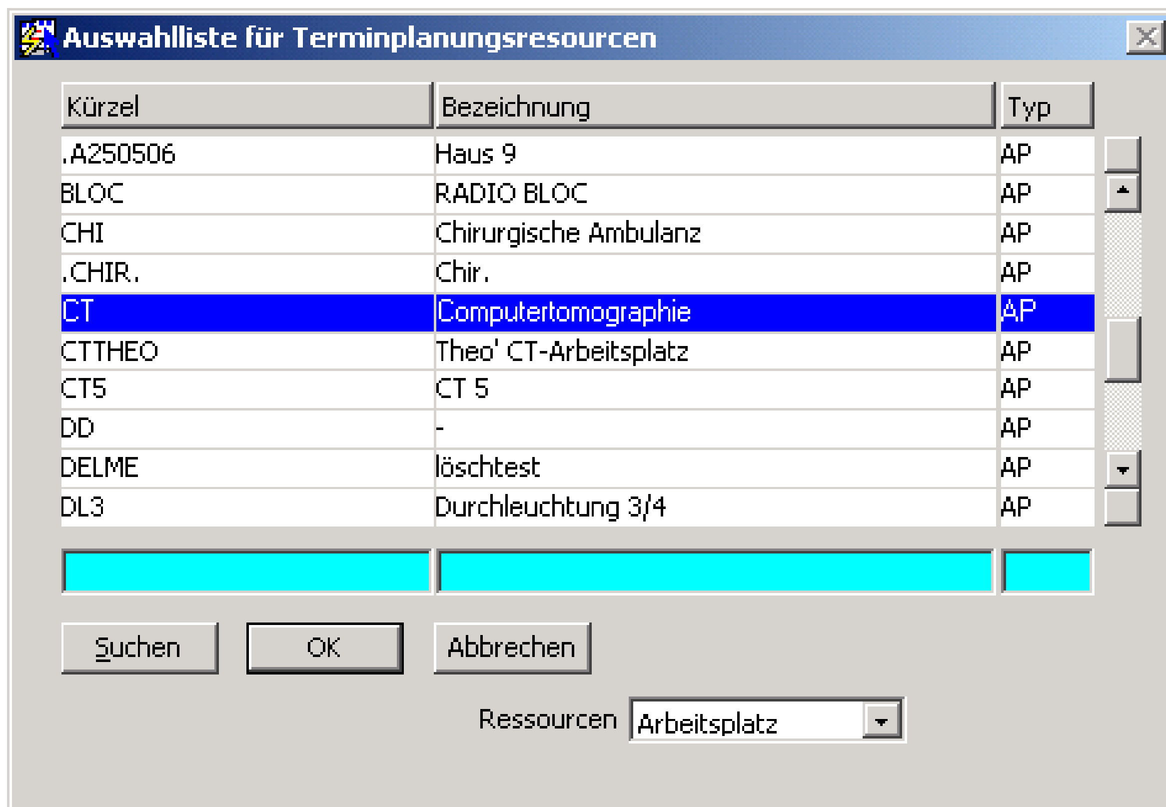

Abbildung 2.2: Der Ressourcenauswahl-Dialog

Aufgrund früherer Betrachtungen [Hieber 96] wurde bisher davon ausgegangen, daß zum Erstellen von Profilterminen ein Backtracking-Algorithmus notwendig ist, der den Zeitaufwand pro einzelnem Termin stark erhöhen würde. Daher wurde die Zeitspanne, die bei Profilterminen pro einzelnem Termin nicht überschritten werden sollte, auf zehn Sekunden festgesetzt. In Kap. 4 wird ein neuer Ansatz vorgestellt werden, der ohne den Einsatz von Backtracking-Algorithmen auskommt.

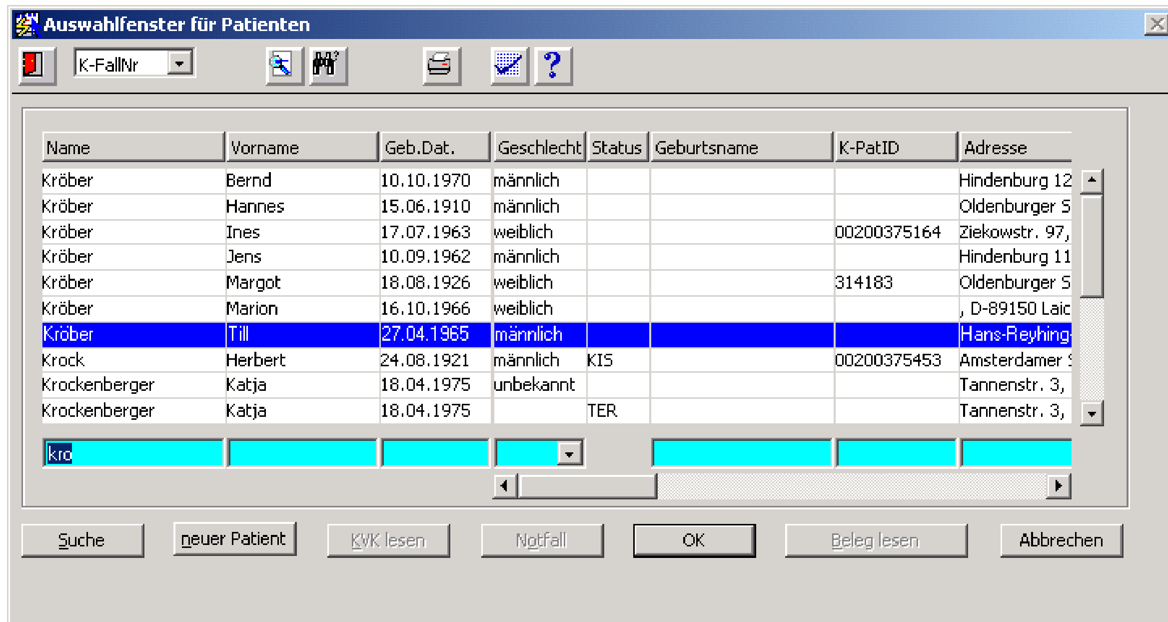

Abbildung 2.3: Der Patientenauswahl-Dialog

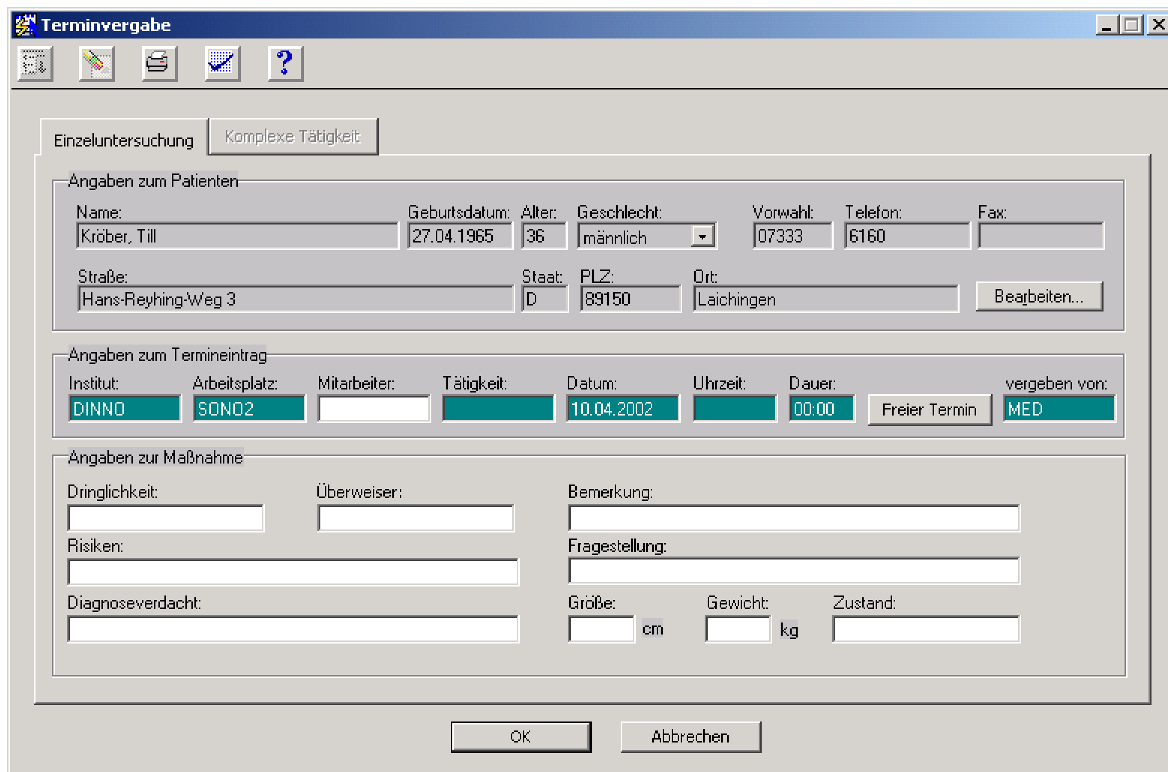

Abbildung 2.4: Der Terminvergabe-Dialog

Kapitel 3

Theorie der Ablaufplanung

Die *Ablaufplanung* ist ein Untergebiet der *Unternehmensforschung* (auch unter der englischen Bezeichnung *Operations Research* bekannt). Die Anfänge der Planungstheorie datieren um die Jahrhundertwende, als man begann, sich wissenschaftlich mit Problemen wie der Maschinenbelegung in Fertigungsanlagen zu befassen (Siehe [Pinedo 95]). Der wichtigste Pionier auf diesem Gebiet war Henry Gantt (1861–1919). Die ersten Resultate in Form von wissenschaftlichen Veröffentlichungen wurden jedoch erst in den 50er Jahren erarbeitet.
Die Ablaufplanung beschäftigt sich mit der

> „Vergabe von Ressourcen über einen Zeitraum hinweg mit dem Ziel, eine Anzahl von Aufträgen zu erledigen“ (Übersetzt nach [Baker 74]).

Es geht also darum, zu entscheiden, in welcher Reihenfolge eine vorhandene Anzahl Ressourcen (im terminus technicus *Prozessoren* genannt) eingesetzt werden soll, um eine gegebene Anzahl von *Aufträgen* gemäß einem *Optimaierungskriterium* zu bearbeiten. Die Aufträge (engl. *Jobs*) setzen sich dabei aus elementaren *Operationen* (engl. *Operations* oder *Tasks*) zusammen, von denen jede von einem anderen Prozessor bearbeitet werden kann. Es gibt unterschiedliche Optimalitätskriterien, die dabei zum Einsatz kommen können, oft ist beispielsweise die Minimierung der mittleren oder maximalen Bearbeitungszeit von Bedeutung, oder die Minimierung von Kosten.
Zur graphischen Darstellung einer Belegung verwendet man *Gantt-Diagramme*. Es gibt prozessororientierte (Abb. 3.1) und aufgabenorientierte (Abb. 3.2) Gantt-Diagramme. Auf der waagrechten Achse ist immer die Zeit abgetragen, auf der senkrechten die Prozessoren bzw. Aufgaben.

3.1 Planungsprobleme

Je nachdem, welche Arten von Prozessoren und Aufgaben vorliegen, d. h., welcher Prozessor welche Operationen in welcher Zeit bearbeiten kann und unter welchen Bedingungen die zu einem Auftrag gehörenden Operationen abgearbeitet werden müssen, entstehen mehrere grundlegend verschiedene Kategorien von Planungsproblemen. Zur Beschreibung von Planungsproblemen gibt es in der Ablaufplanung verschiedene Modelle. Ein solches Modell setzt sich aus drei Teilen zusammen (vgl. [Brucker 95], [Blazewicz 96]):

- Dem Auftragsmodell
- Dem Prozessormodell und

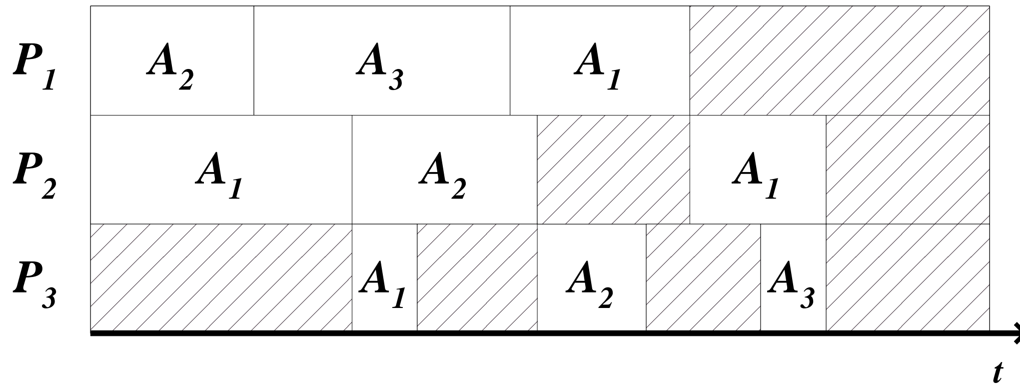

Abbildung 3.1: Prozessororientiertes Gantt-Diagramm

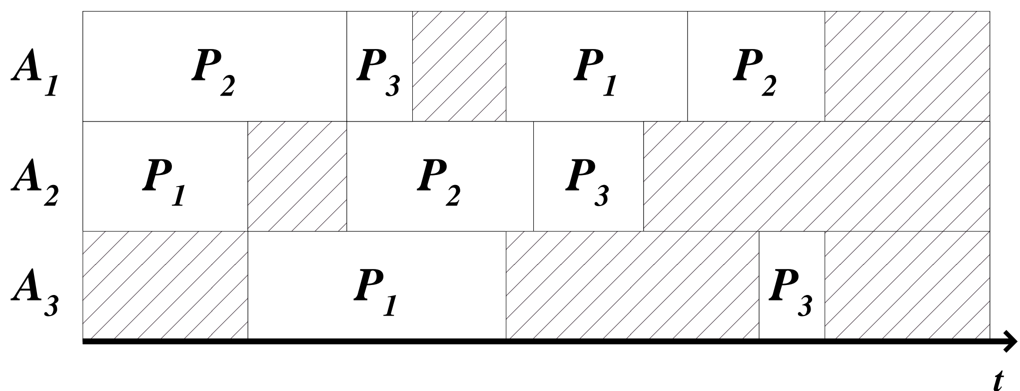

Abbildung 3.2: Gleiche Belegung wie in Abb. 3.1, jedoch als aufgabenorientiertes Diagramm

- Dem Optimierungskriterium.

Zur kompakten Beschreibung von Planungsproblemen wird eine mathematische Notation benutzt, die auf [Graham 79] zurückgeht. Sie hat die Form $\alpha \mid \beta \mid \gamma$, wobei α die Beschreibung des Prozessormodells, β die Beschreibung des Auftragsmodells und γ die des Optimierungskriteriums ist. Eine genauere Erläuterung der drei Teile dieser Notation und deren Bedeutung findet sich in den folgenden Abschnitten. Soweit nicht anders angegeben, liegen diesen Abschnitten die Bücher [Brucker 95] und [Blazewicz 96] zugrunde.

3.1.1 Auftragsmodelle

Bei jedem Planungsproblem ist eine Menge $\{A_1, \ldots, A_n\}$ von Aufträgen zu bearbeiten. Jeder Auftrag A_i besteht aus einer Menge von Operationen $O_{i1}, \ldots, O_{in_i}$. Besteht der Auftrag nur aus einer Operation, so schreibt man A_i statt A_{i1}.
Der Parameter β eines Planungsproblems beschreibt dessen Auftragsmodell und hat die Form $\beta_1\beta_2\beta_3\beta_4\beta_5$. Jeder der fünf Einzelparameter kann entweder leer sein oder einen der Werte, die im folgenden aufgeführt sind, haben: listii

- β_1 gibt an, ob die Operationen *teilbar* (*präemptiv*) sind.

β_1 leer: Die Operationen sind unteilbar, können also nur als Ganzes ausgeführt werden.
$\beta_1 =$ pmtn: Die Operationen sind teilbar, können also in mehreren Etappen auf verschiedenen Prozessoren bearbeitet werden.

- β_2 spezifiziert *Abhängigkeitsbeziehungen* unter den Aufträgen. Ein Satz von Abhängigkeitsbeziehungen eines Planungsproblems wird durch einen gerichteten Graphen G dargestellt.
  β_2 leer: Die Aufträge können in beliebiger Reihenfolge ausgeführt werden.
  $\beta_2 =$ prec: G ist ein beliebiger gerichteter Graph.
  $\beta_2 =$ uan: G ist ein *einfach verbundenes Aktivitätsnetzwerk*, d. h., ein gerichteter Graph, bei dem für jedes Paar von Knoten (u, v) genau ein Pfad p existiert, so daß p u und v verbindet oder umgekehrt.
  $\beta_2 =$ intree: G ist ein *Einwärtsbaum*, also ein Baum, bei dem jeder Knoten höchstens eine wegführende Kante besitzt.
  $\beta_2 =$ outtree: G ist ein *Auswärtsbaum*, also ein Baum, bei dem jeder Knoten höchstens eine herführende Kante besitzt.
  $\beta_2 =$ tree: G ist entweder ein Einwärtsbaum oder ein Auswärtsbaum.
  $\beta_2 =$ chains: G ist eine *Vereinigung von Kettengraphen*, d. h. ein Baum, bei dem jeder Knoten höchstens eine herführende und höchstens eine wegführende Kante besitzt.

- β_3 gibt Auskunft über *Bereitstellungszeiten* (engl. *ready times* oder *release dates*) von Aufträgen.
  β_3 leer: Alle Aufträge müssen zum Zeitpunkt 0 zur Bearbeitung bereitstehen.
  $\beta_3 = r_j$: Für jeden Auftrag ist ein Zeitpunkt angegeben, ab dem er zur Bearbeitung bereitstehen muß.

- β_4 steht für die Bearbeitungszeiten der einzelnen Operationen.
  β_4 leer: Für jede Operation ist eine Bearbeitungszeit angegeben.
  $\beta_4 = (p_j = 1)$ bzw.
  $\beta_4 = (p_{ij} = 1)$: Alle Aufträge haben die gleiche Bearbeitungszeit.

  Daneben gibt es noch eine Reihe anderer Charakteristika, auf die man in der Literatur gelegentlich trifft, wie z. B. $\beta_4 = (p_{ij} \in \{1, 2\})$ oder $\beta_4 = (\underline{p} \leq p_{ij} \leq \bar{p})$.

- β_5 schließlich enthält Informationen darüber, ob bei der Auftragsbearbeitung *Stichtermine* beachtet werden müssen.
  β_5 leer: Es sind keine Stichtermine festgelegt.
  $\beta_5 = d_i$: Jeder Auftrag hat einen Stichtermin, bis zu dem der Auftrag spätestens fertig sein muß.

Es gibt darüberhinaus eine Vielzahl von Spezialfällen von Planungsproblemen; bei diesen können zusätzlich zu $\beta_1, \ldots, \beta_5$ auch noch weitere Bedingungen in β enthalten sein.

3.1.2 Prozessormodelle

Bei einem konkreten Planungsproblem ist stets eine Menge $P = \{P_1, \ldots, P_m\}$ von Prozessoren gegeben. s_k ist die *Geschwindigkeit* des Prozessors P_k. Jedem Auftrag ist eine *Bearbeitungszeit* p_{ik} zugeordnet. p_{ik} ist die Bearbeitungszeit des Auftrags i auf dem Prozessor k unter der

Annahme, daß der Auftrag ausschließlich von diesem einen Prozessor bearbeitet wird, d. h., keine Präemption auftritt.
Jeder Operation O_{ij} eines Auftrags A_i ist eine Menge $\mu_{ij} \subseteq P$ von Prozessoren, die diese Operation bearbeiten können, zugeordnet.
Je nach Art der Prozessoren und der Aufträge teilt man Prozessormodelle in drei Kategorien ein:

- Einprozessorsysteme
- Parallele Prozessoren
- Spezialisierte Prozessoren (In der engl. Literatur *Dedicated Processors* genannt)

Diese Klassen enthalten jeweils Prozessormodelle, von denen jedes durch einen Parameter α bezeichnet wird. Dieser Parameter ist gemäß $\alpha = \alpha_1\alpha_2$ zusammengesetzt, wobei α_1 den Prozessortyp angibt und α_2 die Anzahl der Prozessoren.
Im folgenden werden die verschiedenen Prozessormodelle genauer beschrieben. Dabei gelten für alle Modelle folgende allgemeine Regeln:

- Operationen, die zu verschiedenen Aufträgen gehören, können in beliebiger Reihenfolge ausgeführt werden.
- Jeder Prozessor kann höchstens einen Auftrag gleichzeitig bearbeiten.
- Jeder Auftrag kann nur auf einem Prozessor gleichzeitig laufen.

Einprozessorsysteme

Wenn der Parameter α leer ist, liegt ein Einprozessorsystem vor, d. h. alle Aufträge werden von einem einzelnen Prozessor bearbeitet.

Parallele Prozessoren

Der Begriff „parallel“ ist nach Ansicht des Autors etwas unglücklich gewählt, denn er bezeichnet nicht etwa die Eigenschaft einer Gruppe von Prozessoren, mehrere Aufträge gleichzeitig bearbeiten zu können (welche selbstverständlich auch bei spezialisierten Prozessoren gegeben ist), sondern bedeutet vielmehr, daß die Menge von Operationen, die ein Prozessor ausführen kann, für alle Prozessoren gleich ist. „Parallel“ bezeichnet also den Umstand, daß alle Prozessoren bezüglich Auftragsbearbeitung die gleichen Fähigkeiten haben.
Beim Modell mit parallelen Prozessoren enthält grundsätzlich jeder Auftrag genau eine Operation.
Parallele Prozessoren können sich in ihren Bearbeitungsgeschwindigkeiten unterscheiden. Man unterscheidet daher 3 Unterklassen von parallelen Prozessoren:

1. Wenn $\alpha_1 = P$ ist, handelt es sich um *identische* Prozessoren. Das bedeutet, daß die Bearbeitungszeit für alle Operationen eines Auftrags gleich ist, unabhängig vom Prozessor. Die Bearbeitungszeit hängt also nur vom Auftrag ab. Mathematisch ausgedrückt bedeutet dies $p_{ik} = p_i, 1 \leq k \leq n$.
2. Bei $\alpha_1 = Q$ spricht man von *gleichartigen* Prozessoren. Hier unterscheiden sich die Bearbeitungszeiten für einen Auftrag um einen konstanten Geschwindigkeitsfaktor, der von Prozessor zu Prozessor verschieden sein kann. Es gilt also $p_{ik} = p_i/s_k$, wobei p_i die Bearbeitungszeit des Auftrags auf einem Prozessor mit Geschwindigkeit 1 ist.

3. Ist $\alpha_1 = R$, so sind die Prozessoren *unverwandt*, d. h., die Bearbeitungszeit hängt sowohl vom Prozessor, als auch von der auszuführenden Operation ab. Im Gegensatz zu gleichartigen Prozessoren gibt es hier also keinen konstanten Faktor, durch den sich die Bearbeitungszeiten für eine gegebene Operation von Prozessor zu Prozessor unterscheiden, sondern für jedes Paar $(Prozessor, Operation)$ ist eine eigene Bearbeitungsgeschwindigkeit definiert. Die Bearbeitungszeit ergibt sich also gemäß $p_{ij} = p_i/s_{ij}$, wobei p_i die Bearbeitungszeit auf dem Standardprozessor ist.

Spezialisierte Prozessoren

Im Gegensatz zu Szenarien mit parallelen Prozessoren, wo alle Operationen, die ein Prozessor ausführen kann, auch von jedem anderen Prozessor ausgeführt werden können, ist dies bei spezialisierten Prozessoren nicht gegeben. Vielmehr ist jedem Prozessor eine Menge von Operationen zugeordnet, die er durchführen kann.
Jeder Operation O_{ij} ist eindeutig ein Prozessor zugeordnet. In anderen Worten, $|\mu_{ij}| = 1 (1 \leq i \leq n, 1 \leq j \leq n_i)$. Prozessormodelle mit spezialisierten Prozessoren werden in 3 Kategorien eingeteilt:

1. Das *Job-Shop-Modell* ($\alpha_1 = J$). Hier liegen n Aufträge und m Prozessoren vor. Jedem Auftrag ist eine Abarbeitungsreihenfolge der Operationen zugeordnet.

2. Beim *Flow-Shop-Modell* ($\alpha_1 = F$) hat jeder Auftrag eine Anzahl n von Operationen, die gleich der Zahl der Prozessoren ist. Die Abarbeitungsreihenfolge der Operationen innerhalb eines Auftrags ist festgelegt. Die i-te Operation muß vom Prozessor i bearbeitet werden. Der Flow-Shop ist ein Spezialfall des Job-Shops mit $n_i = m$ und $\mu_{ij} = \{P_j\} (1 \leq i \leq n, 1 \leq j \leq n_i)$.

3. Das *Open-Shop-Modell* ($\alpha_1 = O$) ist mit dem Flow-Shop-Modell identisch, bis auf den Unterschied, daß die Operationen eines Auftrags in beliebiger Reihenfolge bearbeitet werden können.

Anzahl der Prozessoren

Wie schon erwähnt, gibt der Parameter α_2 die Prozessorzahl im System an. Wenn α_2 leer ist, ist die Anzahl variabel. Ist nur ein Prozessor vorhanden, so wird α_2 weggelassen.

3.1.3 Optimierungskriterien

Bei einem Planungsproblem gilt es, eine optimale Planung zu finden, d. h., eine Planung, die eine *Kostenfunktion* minimiert.
Eine Kostenfunktion F für ein Optimierungsproblem besteht aus zwei Funktionen, die miteinander verkettet sind:

$$f_i : \mathbb{R} \mapsto \mathbb{R}, 1 \leq i \leq n$$

und

$$f_{\text{ges}} : \mathbb{R}^n \mapsto \mathbb{R}.$$

Die f_i geben die Kosten für den Auftrag A_i an, abhängig von dem Zeit Zeitpunkt C_i der Fertigstellung des Auftrages. Die Funktion f_{ges} bildet die n Werte für die Kosten der Aufträge

auf einen einzigen Wert ab. Dieser Wert ist ein Maß für die Gesamtkosten der Planung. Die Kostenfunktion F ergibt sich wie folgt:

$$F(\vec{C}) = f_{\text{ges}}(f_1(C_1), f_2(C_2), \ldots, f_n(C_n)).$$

Von f_{ges} gibt es im Wesentlichen vier Varianten:

- $f_{\text{ges}}(\vec{x}) = \max(x_1, \ldots, x_n)$ (Maximum)
- $f_{\text{ges}}(\vec{x}) = \sum_{i=1}^{n} x_i$ (Summe)
- $f_{\text{ges}}(\vec{x}) = \max(w_1 x_1, \ldots, w_n x_n)$ (Gewichtetes Maximum)
- $f_{\text{ges}}(\vec{x}) = \sum_{i=1}^{n} w_i x_i$ (Gewichtete Summe)

Beispiele für f_i sind:

- $f_i = C_i$ (Zeitpunkt der Fertigstellung des Auftrags)
- $f_i = C_i - d_i$, wobei d_i der Stichtermin für den Auftrag A_i ist
- $f_i = \max\{0, C_i - d_i\}$
- $f_i = \begin{cases} 0 & \text{wenn } C_i \leq d_i \\ 1 & \text{sonst} \end{cases}$
- $f_i = |C_i - d_i|$ (Abweichung vom Stichtermin).

Der Parameter γ gibt das Optimalitätskriterium des Planungsproblems an, also die Kostenfunktion. Die am häufigsten verwendeten Kostenfunktionen sind die folgenden:

- $\gamma = C_{\max} : F(\vec{C}) = \max\{C_i | i = 1, \ldots, n\}$
- $\gamma = \sum C_i : F(\vec{C}) = \sum_{i=1}^{n} C_i$
- $\gamma = L_{\max} : F(\vec{C}) = max_{i=1}^{n}(C_i - d_i)$ mit Stichterminen d_i
- $\gamma = \sum T_i : F(\vec{C}) = \sum_{i=1}^{n} \max\{0, C_i - d_i\}$
- $\gamma = \sum U_i : F(\vec{C}) = |\{i | C_i \leq d_i\}|$

3.1.4 Beispiele

Hier nun einige ausgewählte Klassen von Planungsproblemen:

$\boldsymbol{Jk|p_{ij} = 1|\sum T_i}$ bezeichnet das Problem, die durchschnittliche Fristüberschreitung in einer Anlage mit k Prozessoren und einheitlichen Bearbeitungszeiten zu minimieren.

$\boldsymbol{P||C_{\max}}$ ist die Klasse von Planungsproblemen, bei denen Aufträge, deren Operationen unteilbar sind und beliebige Bearbeitungszeiten haben können, auf identische Prozessoren verteilt werden sollen. Die Aufgaben sind alle vom Zeitpunkt 0 an zur Bearbeitung verfügbar, und es soll die Dauer bis zur Fertigstellung des letzten Auftrags minimiert werden.

$\boldsymbol{O3|\text{pmtn}, r_j|\sum C_i}$ steht für ein Open-Shop-Modell mit 3 Prozessoren, bei dem Aufträge zu verschiedenen Zeiten beim System eingehen und Operationen beliebige Bearbeitungszeiten haben. Ziel ist es, die mittlere Bearbeitungszeit zu minimieren.

3.1.5 Erweiterungen

Zusätzlich zum hier vorgestellten Grundmodell der Planungsprobleme sind im Laufe der Zeit auch zahlreiche Erweiterungen untersucht worden. Dazu zählen die Berücksichtigung von Transportzeiten und -wegen, Signallaufzeiten, Materialverbrauch, stochastische Modelle für Auftragsankunftszeiten und Prozessorausfallzeiten, sog. „flexible" Prozessoren, die für verschiedene Aufgaben umgerüstet werden können, usw. ([Brucker 95], [Pinedo 95], [Blazewicz 96]). Diese Modelle sollen hier nicht näher erläutert werden, jedoch wird in Abschnitt 3.3 das Modell dahingehend erweitert werden, daß eine Operation mehrere Prozessoren gleichzeitig benötigen kann.

3.2 Lösung von Planungsproblemen

Für viele Klassen von Planungsproblemen existieren Algorithmen, die das Problem effizient, d. h. in polynomieller Zeit lösen. Bei anderen Klassen hingegen ist bekannt, daß sie NP-hart sind, also aller Wahrscheinlichkeit nach nicht effizient zu lösen sind. Viele Problemklassen sind auch noch offen und es ist nicht bekannt, ob das Problem effizient lösbar ist.

Effizient zu lösen sind im allgemeinen die spezielleren Planungsprobleme, während ab einem bestimmten Grad von Allgemeinheit der Problemstellung das Problem NP-hart wird. So ist das Problem $O2||C_{\max}$ in linearer Zeit lösbar [Gonzales et al. 76], doch wenn man beliebige Auftrags-Ankunftszeiten zuläßt ($O2|r_i|C_{\max}$), entsteht ein NP-hartes Problem [Lawler 81]. Wenn die Bearbeitungszeit aller Operationenen gleich ist, liegt das Problem wieder in P, selbst mit beliebiger Prozessorzahl und mit beliebigen Ankunftszeiten ($O|p_{ij} = 1, r_i|C_{\max} \in P$ [Horn 74].
Mit steigender Prozessorzahl werden Planungsprobleme ebenfalls schwieriger: Das schon erwähnte Problem $O2||C_{\max} \in P$ ist bei 3 Prozessoren ($O3||C_{\max}$) NP-hart [Gonzales et al. 76]. Beim Job-Shop-Modell ist bereits der überwiegende Teil der Probleme mit fester Prozessorzahl NP-hart. So sind $J3|n = 3|\sum C_i$ und $J3|n = 3|C_{\max}$ NP-hart [Sotskov 95], ebenso wie $J2|p_{ij} = 1|\sum T_i$ [Lenstra 79]. Bei in der Praxis auftretenden Typen von Job-Shop-Problemen ist es nach dem heutigen Stand bereits bei Problemgrößen, die 20 Aufträge und 10 Maschinen übersteigen, praktisch nicht mehr möglich, eine optimale Lösung zu finden [Jain 98]. Eine Übersicht der Komplexitätsklassen findet sich unter [Univ. Osnabrück 02].

Der Augenmerk gilt daher seit längerer Zeit *Approximationsalgorithmen*, unter denen sich Tabusuche, Engpaßverschiebung (Shifting Bottleneck Procedure), Lokale Genetische Suche und Simulated Annealing sowie Kombinationen dieser Methoden als die erfolgreichsten erwiesen haben [Jain 98].

3.3 Mehrere Prozessoren pro Operation

Wir wollen nun das Modell dahingehend erweitern, daß zur Bearbeitung einer Operation mehrere Prozessoren gleichzeitig benötigt werden können. Dabei beziehen wir uns im Wesentlichen auf [Brucker 95], Kap. 10. Dieses Modell ist nicht mit *parallelen Systemen* zu verwechseln, bei denen *parallelisierbare Operationen* (engl. *divisible tasks*) auf mehrere Prozessoren aufgeteilt werden können ([Blazewicz 96], Kap. 6).
Bei unserem Modell geht es vielmehr darum, daß jeder Operation O_{ij} eines Auftrags A_i eine Menge $\mu_{ij} \subseteq P$ zugeordnet ist. Während der gesamten Ausführung der Operation werden alle Prozessoren aus μ_{ij} für die Operation benötigt. Damit können nun Situationen, in denen mehrere Ressourcen zur Durchführung eines Arbeitsganges benötigt werden, modelliert

werden. Beispielsweise wäre es in einer Fertigungshalle denkbar, daß eine bestimmte, kompliziert zu handhabende Maschine nur von einer einzigen, speziell ausgebildeten Fachkraft zu bedienen ist. Es muß also sowohl die Maschine selbst, als auch die Fachkraft während der betreffenden Zeitspanne reserviert werden.

Zur Beschreibung von Systemen mit Mehrprozessoroperationen verwendet man eine erweiterte Fassung der Notation aus Kapitel 3.1. Das Feld α kann beim Mehrprozessormodell für α_1 zusätzlich folgende Werte annehmen:

- $\alpha_1 =$ PMPT (Identische, parallele Prozessoren)
- $\alpha_1 =$ JMPT (Job-Shop-Modell)
- $\alpha_1 =$ FMPT (Flow-Shop-Modell)
- $\alpha_1 =$ OMPT (Open-Shop-Modell)

α_2 gibt weiterhin die Prozessorzahl an bzw. bedeutet, falls es leer ist, daß die Prozessorzahl nicht vorgegeben ist.
Ansonsten gelten alle bisherigen Regeln und Definitionen des Grundmodells auch hier, mit Ausnahme der letzten der drei allgemeinen Regeln für Prozessormodelle (siehe Seite 10, Abschnitt 3.1.2), die jetzt wegfällt.

Da Planungsprobleme mit einem Prozessor pro Operation ein Sonderfall der Planungsprobleme mit Mehrprozessoroperationen sind, sind letztere in puncto Komplexität mindestens so schwierig zu lösen wie erstere und teilweise noch schwieriger, wie im Falle von JMPT$2|p_{ij} = 1|C_{\max}$, welches im Gegensatz zu seinem Einprozessor-Gegenstück $J2|p_{ij} = 1|C_{\max}$ NP-hart ist. Es sei hier noch einmal auf [Univ. Osnabrück 02] verwiesen, welches eine Einteilung sowohl der Probleme mit Einprozessor- als auch der mit Mehrprozessoroperationen in Komplexitätsklassen bietet.

3.4 Terminplanung in der Radiologie

In radiologischen und strahlenmedizinischen Einrichtungen müssen – neben „normalen“ Einzelterminen – für einen Patienten häufig sogenannte *Serientermine* vereinbart werden. Damit ist eine Anzahl von Terminen gemeint, bei denen in möglichst gleichem Abstand jeweils die gleiche Untersuchung oder Therapie durchgeführt werden muß. Beispielsweise werden bei der Tumortherapie häufig Bestrahlungsserien angewandt, bei denen z. B. 10 Bestrahlungen im Abstand von 6 Wochen durchgeführt werden.
Eine weiterer Typus von Terminanforderungen ist das Anlegen mehrerer Termine gemäß einem *Terminprofil*. Darin sind mehrere Untersuchungen oder Eingriffe definiert, deren zeitlicher Abstand zwischen einem Mindest- und einen Höchstwert liegen muß. Beispielsweise ist es bei einer Untersuchung mit Kontrastmitteln wichtig, daß zwischen der Injektion des Kontrastmittels und der Aufnahme des Bildes eine gewisse Zeit vergeht, damit sich das Mittel im Körper des Patienten verteilt; andererseits darf auch nicht zuviel Zeit vergehen, weil sich sonst das radioaktive Präparat zu stark abbaut und die Strahlungsintensität nicht für eine Aufnahme ausreicht. Hingegen dürfen bei bestimmten Untersuchungen keine Strahlenquellen im Patientenkörper vorhanden sein, so daß solch eine Untersuchung also zeitlich vor einer Kontrastmittelinjektion liegen muß. Diese Reihenfolge- und Zeitkriterien werden durch Terminprofile beschrieben. Termine, die gemäß einem Profil angelegt werden, heißen *Profiltermine*.

Wir wollen nun untersuchen, welche Art von Planungsproblemen diese zwei Modelle – Profiltermine und Serientermine – aufwerfen. Zunächst einmal ist klar, daß eine Terminanforderung an eine radiologische Klinik unmittelbar bestätigt oder abgelehnt werden muß. Eine Terminanforderung kann von verschiedenen Seiten bei einem Terminplanungssystem einer radiologischen Klinik eintreffen: von einer anderen Krankenhausabteilung, einem anderen Krankenhaus oder einem niedergelassenen Arzt. Die Anfrage kann dabei von der Außenwelt elektronisch (mittels HL7) oder telefonisch übermittelt werden. In jedem Falle möchte die Gegenseite eine prompte Information darüber erhalten, ob die Terminanfrage erfolgreich war und falls ja, wann der Termin oder die Termine eingetragen wurden. Somit müssen Terminanfragen vom Terminplanungssystem immer sofort bearbeitet werden, das System kann also nicht Anfragen über einen Zeitraum sammeln – etwa einen Tag oder eine Woche – und dann eine Tages- oder Wochenplanung erstellen.
Aus demselben Grund können Termine, wenn sie einmal geplant sind, auch nicht beliebig im Nachhinein verschoben werden (wie das z. B. bei einem Maschinenpark der Fall wäre), denn ein externer Arzt wäre sicherlich nicht sehr glücklich damit, seine Terminplanung ständig verwerfen und neu erstellen zu müssen, weil die Radiologieabteilung des Krankenhauses X den Termin seines Patienten verschoben hat.

Das Terminplanungssystem ist also in seiner Freiheit dahingehend eingeschränkt, daß es jede Terminanfrage einzeln bearbeiten muß, sobald sie eintrifft, und für neue Termine nur nichtbelegte Zeiten in Frage kommen, ein alter Termin also nicht zugunsten eines neuen verschoben werden darf.
Da für eine medizinische Maßnahme stets nur eines oder einige wenige Geräte in Frage kommen, mit denen die Maßnahme durchgeführt werden kann, sind medizinische Geräte aus planungstheoretischer Sicht also spezialisierte Prozessoren. Bei Ärzten, MTA und anderem medizinischen Personal ist dies im allgemeinen ebenfalls der Fall (allerdings ist bei Menschen der Spezialisierungsgrad niedriger als bei medizinischen Geräten).
Zur Bearbeitung einer medizinischen Maßnahme müssen mehrere Prozessoren gleichzeitig eingesetzt werden: Ein Patient, ein oder mehrere Mitarbeiter, und ggf. ein oder mehrere Geräte. Es werden also mehrere Prozessoren (≥ 2) pro Operation benötigt. Man könnte das Modell noch weiter dahingehend verallgemeinern, daß zu einer Operation statt einer Menge von Prozessoren eine Menge von Prozessorengruppen angegeben werden können und aus jeder Gruppe ein Prozessor verfügbar sein muß. Solche Situationen treten in Kliniken nicht selten auf, z. B. wenn eine Computertomographie durchgeführt werden soll und mehrere CT-Arbeitsplätze zur Verfügung stehen. Wir wollen uns hier jedoch auf das Standard-Mehrprozessormodell beschränken, d. h. in jedem Profilelement ist eine feste Menge benötigter Prozessoren definiert. Die Anzahl der Prozessoren ist im allgemeinen ungleich der Anzahl der Aufträge. Dadurch scheiden die spezielleren Fälle Open-Shop und Flow-Shop aus, wir haben also ein Job-Shop-Modell vorliegen.
Krankenhausangestellte haben bestimmte Arbeitszeiten, außerhalb derer sie nicht verfügbar sind; Patienten haben Ruhezeiten, während derer sie nicht verplant werden können und medizinische Geräte werden von Zeit zu Zeit gewartet und sind solange nicht einsatzbereit. Um diesem Umstand im Modell Rechnung zu tragen, belegen wir den betreffenden Prozessor während dieser Totzeiten mit einer beliebigen Aufgabe. Dann kann der Prozessor während dieser Zeit keine andere Aufgabe bearbeiten.
Wir wollen uns nun der mathematischen Modellierung von Planungsproblemen in der Radiologie zuwenden. Ein Auftrag umfasse im Folgenden als Operationen alle Einzeltermine eines Profiltermins bzw. einer Terminserie.

3.4.1 Das Modell für Serientermine

Bei Serienterminen ist der Abstand zwischen zwei Terminen festgelegt (mit einer gewissen Toleranz von z. B. ±1 Tag). Planungstheoretisch kann man diese Zeitfenster modellieren, indem man für die einzelnen Operationen eines Auftrags (sprich: einer Terminserie) Bereitstellungszeiten und Stichtermine festlegt. Die Bereitstellungszeit gibt dann den frühesten und der Stichtermin den spätesten Zeitpunkt an, zu dem der Termin eingetragen werden kann. Die Parameter des Planungsproblems sind folglich:

α_1	=	JMPT	(Job-Shop, mehrere Prozessoren pro Operation)
α_2	=	[leer]	(Anzahl Prozessoren ist nicht fest)
β_1	=	[leer]	(Operationen sind unteilbar)
β_2	=	[leer]	(Aufträge sind voneinander unabhängig)
β_3	=	r_j	(Es sind Bereitstellungszeiten festgelegt)
β_4	=	[leer]	(Aufträge haben verschiedene Bearbeitungszeiten)
β_5	=	d_i	(Es sind Stichtermine festgelegt)
$\tilde{\beta}$	=	fest	

Der Parameter $\tilde{\beta}$ = fest steht für unsere Zusatzbedingung, daß Termine einzeln vergeben werden müssen und bereits belegte Zeiten berücksichtigt werden müssen.
Als Kostenfunktion γ nehmen wir

$$F_{\text{ser}}(\vec{C}) = \sum_{A_i \in P_{\text{pat}}} C_i \text{ mit } P_{\text{pat}} = \{p \in P | p \text{ ist Patient}\}, \tag{3.1}$$

dadurch wird die mittlere Wartezeit der Patienten minimiert.
Das Planungsproblem bei Serienterminen lautet also:

$$JMPT|r_j, d_i, \text{fest}|F_{\text{ser}} \tag{3.2}$$

In Kapitel 4 wird darauf eingegangen, wie das Problem im Terminplanungssystem von Medora gelöst wird.

3.4.2 Das Modell für Profiltermine

Bei Serienterminen war für jeden Einzeltermin ein Zeitpunkt festgelegt, um den herum der Termin liegen mußte. Bei Profilterminen dagegen ist für zwei aufeinanderfolgende Operationen ein Mindest- und ein Höchstabstand definiert. Jedes Zeitfenster hängt also vom jeweils vorigen ab. Dadurch ist es nun nicht mehr möglich, feste Bereitstellungs- und Stichtermine anzugeben.
Statt dessen greifen wir zu einem anderen Mittel: Um die Maximalabstände zwischen zwei Operationen O_{ij} und $O_{i,j+1}$ einzuhalten, definieren wir eine Funktion $\tilde{F}_{\text{prf}}$, die 0 wird, wenn das Maximalabstandskriterium erfüllt ist. $\tilde{F}$ sei wie folgt definiert:

$$\tilde{F}_{\text{prf}}(\vec{C}) = \sum_{i=1}^{n} \sum_{j=2}^{n_i} \max\{0, C_{ij} - C_{i,j-1} - p_{ij} - w_{ij}^{\max}\},$$

wobei C_{ij} den Zeitpunkt der Fertigstellung der Operation O_{ij} bezeichne und $w_{ij}^{\max}$ den Maximalabstand zwischen zwei Operationen O_{ij} und $O_{i,j+1}$. $\tilde{F}$ ist stets ≥ 0 und es gilt $F = 0$ genau dann, wenn alle Maximalabstände eingehalten werden.

Um zusätzlich zur Erfüllung des Maximalabstandskriteriums auch die mittlere Wartezeit der Patienten zu minimieren, setzen wir als Kostenfunktion des Planungsproblems

$$F_{\text{prf}} = F_{\text{ser}} + \tilde{F}_{\text{prf}}. \tag{3.3}$$

Um die Minimalabstände $w_{ij}^{\min}$ zwischen zwei Operationen O_{ij} und $O_{i,j+1}$ sicherzustellen, fügen wir für $j = 2, \ldots, n_i$ nach O_{ij} eine „Pufferoperation" $O'_{i\tau}$ in den Auftrag A_i ein, die den Patienten entsprechend der Dauer des Mindestabstandes w_{ij}^{min} reserviert hält. Die Bearbeitungszeit dieser Operation sei $p'_{i\tau} = w_{ij}^{\min}$; die Menge $\mu_{i\tau}$ der benötigten Prozessoren habe als einziges Element den Patienten, für den der Profiltermin angefordert wurde. Die Parameter des Problems lauten somit:

α_1	=	JMPT	(Job-Shop, mehrere Prozessoren pro Operation)
α_2	=	[leer]	(Anzahl Prozessoren ist nicht fest)
β_1	=	[leer]	(Operationen sind unteilbar)
β_2	=	[leer]	(Aufträge sind voneinander unabhängig)
β_3	=	[leer]	(Keine Bereitstellungszeiten)
β_4	=	[leer]	(Aufträge haben verschiedene Bearbeitungszeiten)
β_5	=	[leer]	(Keine Stichtermine)
$\tilde{\beta}$	=	fest	
γ	=	F_{prf}	

In mathematischer Notation lautet das Planungsproblem bei Profilterminen also

$$JMPT|\text{fest}|F_{\text{prf}} \tag{3.4}$$

Eine gegebene Lösung dieses Planungsproblems ist allerdings nur als gültig zu betrachten, wenn das Maximalabstandskriterium erfüllt ist. Nach der Lösung des Planungsproblems muß deshalb als zusätzlichen Schritt noch überprüft werden, ob $\tilde{F}_{\text{prf}} = 0$ gilt. Wenn wir also einen Algorithmus hätten, der das Planungsproblem (3.4) löst, könnten wir diesen in folgenden Algorithmus einbauen, der ausschließlich gültige Lösungen ausgibt:

```
1  Löse das Planungsproblem (3.4)
2  Wenn F̃prf = 0:
3      Ausgabe Lösung
4  Sonst
5      Ausgabe 'Keine Lösung gefunden'
```

Da der fiktive Algorithmus, der in Zeile 1 des Algorithmus verwendet wird, $F_{\text{prf}} = F_{\text{ser}} + \tilde{F}_{\text{prf}}$ minimiert und bei einer gültigen Lösung $\tilde{F}_{\text{prf}} = 0$ gilt, ist F_{ser} und damit die mittlere Wartezeit[1] der Patienten minimal.

Das Hauptproblem ist indes immer noch ungelöst: Wie soll der Algorithmus zur Lösung des eigentlichen Planungsproblems aussehen? Wie wir gesehen haben, sind Job-Shop-Probleme mit Mehrprozessoroperationen bis auf Spezialfälle NP-hart; dies deutet daraufhin, daß auch unser Problem sehr hohe Komplexität hat. Andererseits haben wir einen eben dieser Spezialfälle des Job-Shop-Problems vorliegen, so daß es genausogut denkbar wäre, daß unser Profiltermin-Planungsproblem effizient lösbar ist. Im folgenden Kapitel wollen wir das Problem unter Berücksichtigung dieser Randbedingungen analysieren und versuchen, einen effizienten Lösungsalgorithmus zu finden.

[1]Diese ist probabilistisch zu verstehen, da nur immer nur ein Auftrag im Voraus bekannt ist

Kapitel 4

Algorithmen zur Terminvergabe

4.1 Der Algorithmus zur Profilterminvergabe

Ein Terminprofil besteht aus einer geordneten Liste von *Profilelementen*. Ein Profilelement enthält folgende Informationen:

- Die durchzuführende medizinische Maßnahme
- Den zeitlichen Mindestabstand zur vorigen Maßnahme
- Den Höchstabstand zur vorigen Maßnahme

Durch die Angabe der Maßnahme ist auch die Menge der Prozessoren (Mitarbeiter und Geräte), die zur Durchführung der Maßnahme benötigt werden, festgelegt (Mehrprozessormodell). Ein Terminprofil, also eine Folge von Profilelementen, stellt selber noch kein Planungsproblem dar, da es keinen Zeitpunkt enthält, zu dem die Termine angelegt werden sollen. Erst wenn der Benutzer der Terminplanungssoftware oder eine externe Institution die Anweisung gibt, einen Profiltermin anzulegen, entsteht ein Profiltermin-Planungsproblem. Ein solches Planungsproblem umfaßt folgende Informationen:

- Einer endliche Vereinigung von Zeitintervallen $V_{i1} \subset \mathbb{R}$, in denen die Anfangszeit des ersten Termins des Profils liegen soll
- Der Menge $P = P_1, \ldots, P_m$ aller Prozessoren, bestehend aus dem Patienten, allen Mitarbeitern und allen Geräten
- Der Menge $A_i = O_{i1}, \ldots, O_{in_i}$ von Operationen (medizinischen Maßnahmen) und die Länge p_{ij} jeder Operation
- Für jede Operation O_{ij} eine Menge von Prozessoren $\mu_{ij} \subseteq P$, die für die Operation O_{ij} benötigt werden
- Minimalabstände $\Delta_{ij}^{\min}$ und Maximalabstände $\Delta_{ij}^{\max}$ zwischen dem Ende von $O_{i,j-1}$ und dem Beginn von O_{ij}. $\Delta_{i1}^{\min}$ und $\Delta_{i1}^{\max}$ ist nicht definiert.
- Den Belegungsplänen der Prozessoren aus P. Diese enthalten für jeden Prozessor die Zeiten, zu denen der Prozessor verfügbar ist, d. h. nicht bereits einen der geplanten Aufträge $A_1, \ldots, A_{i-1}$ bearbeitet. Es sei $v_{ik} \subset \mathbb{R}$ eine endliche Vereinigung abgeschlossener Intervalle und bezeichne die Zeiten, zu denen der Prozessor k für den Auftrag A_i verfügbar ist.

Zunächst einmal ist klar, daß bei jeder Einzeloperation O_{ij}, die geplant werden soll, alle für O_{ij} benötigten Prozessoren $P_k \in \mu_{ij}$ während der Dauer der Operation verfügbar sein müssen. Das bedeutet, daß die Menge V_{ij} der für den Startzeitpunkt t_{ij} der Operation O_{ij} möglichen Zeitpunkte eine Untermenge des Schnitts

$$\tilde{V}_{ij} = \bigcap_{k|P_k \in \mu_{ij}} \{[a, b - p_{ij}] | [a, b] \in v_{ik}\} \tag{4.1}$$

ist.
Als zweite Bedingung muß V_{ij} das Mindest- und Höchstabstandskriterium erfüllen. Es bezeichne $[\min_{ij}, \max_{ij}]$ das Intervall, das aufgrund des Mindest- und Höchstabstandes für die Wahl von t_{ij} in Frage kommt. Dieses Intervall hängt von der Wahl von $t_{i,j-1}$ ab und es gilt:

$$\begin{aligned} \min_{ij} &= t_{i,j-1} + p_{i,j-1} + \Delta_{ij}^{\min} \\ \max_{ij} &= t_{i,j-1} + p_{i,j-1} + \Delta_{ij}^{\max} \end{aligned}$$

Anders ausgedrückt muß eine Zeitspanne von mindestens $\Delta_{ij}^{\min}$ und höchstens $\Delta_{ij}^{\max}$ zwischen den Operationen $O_{i,j-1}$ und Operation O_{ij} liegen. Für den Startzeitpunkt t_{ij} muß also

$$\min_{ij} \leq t_{ij} \leq \max_{ij} \tag{4.2}$$

gelten.
Wir erhalten somit für die Menge der für O_{ij} möglichen Zeitpunkte:

$$V_{ij} \subseteq \tilde{V}_{ij} \cap [\min_{ij}, \max_{ij}], \qquad 2 \leq j \leq n_i. \tag{4.3}$$

Dies ist eine Rekursionsformel, da zur Berechnung von t_{ij} und damit $[\min_{ij}, \max_{ij}]$ der Wert $t_{i,j-1}$ benötigt wird.
Wenn wir statt dem $\subseteq$ auch ein Gleichheitszeichen schreiben könnten, könnten wir das Planungsproblem lösen, indem wir, ausgehend von V_{i1}, p_{ij}, μ_{ij} und v_{ik}, sukzessive $V_{i1}, V_{i2}, \ldots, V_{in_i}$ berechneten und dann für $j = n_i, n_i - 1, \ldots, 1$ Zeitpunkte $t_{ij} \in V_{ij}$ wählten.
Diese Gleichheit gilt jedoch nicht, denn nach dem Berechnungsschritt $V_{ij} \to V_{i,j+1}$ sind unter Umständen in V_{ij} Zeitpunkte t_{ij} „ungültige“ Bereiche enthalten, die für kein $t_{i,j+1} \in V_{i,j+1}$ das Minimal-/Maximalkriterium erfüllen würden, wie anhand Abbildung 4.1 ersichtlich wird. Dieses Problem entsteht dadurch, daß die Wahl eines Bereiches $V_{i,j+1}$ den vorhergehenden Bereich V_{ij} „rückwirkend“ einschränken kann. Um das Auftreten ungültiger Bereiche zu verhindern, führen wir nach jedem Rekursionsschritt $V_{i,j-1} \to V_{ij}$ einen *Rückwärtsschritt* aus. Dabei wird für $r = j - 1, \ldots, 1$ die Menge V_{ir} durch

$$\{t \in V_{ir} | \exists\, \delta \in \mathbb{R} : \Delta_{i,r+1}^{\min} \leq \delta \leq \Delta_{i,r+1}^{\max} \text{ und } t + p_{ir} + \delta \in V_{i,r+1}\}$$

ersetzt. Es werden also diejenigen Bereiche von V_{ir} ausgesondert, die zu Kollisionen mit dem Belegungsplan $V_{i,r+1}$ führen würden. Dadurch ist garantiert, daß für ein beliebiges $t_{ij} \in V_{ij}$ stets ein $t_{i,j-1} \in V_{i,j-1}$ existiert, so daß das Minimal-/Maximalkriterium eingehalten wird und kein t_{ij} in einen belegten Bereich fällt.
Wenn bei einem Vorwärts- oder Rückwärtsschritt einem V_{ij} die leere Menge zugewiesen wird, bedeutet das, daß keine Lösung des Planungsproblems existiert.
Abb 4.1 zeigt den Algorithmus zur Lösung des Profiltermin-Planungsproblems. In Zeile 3–5 wird die Rekursionsformel (4.3) auf das aktuelle V_{ij} angewandt, also der Vorwärtsschritt ausgeführt. Danach werden für $r = j - 1, \ldots, 1$ in einer `for`-Schleife die Rückwärtsschritte

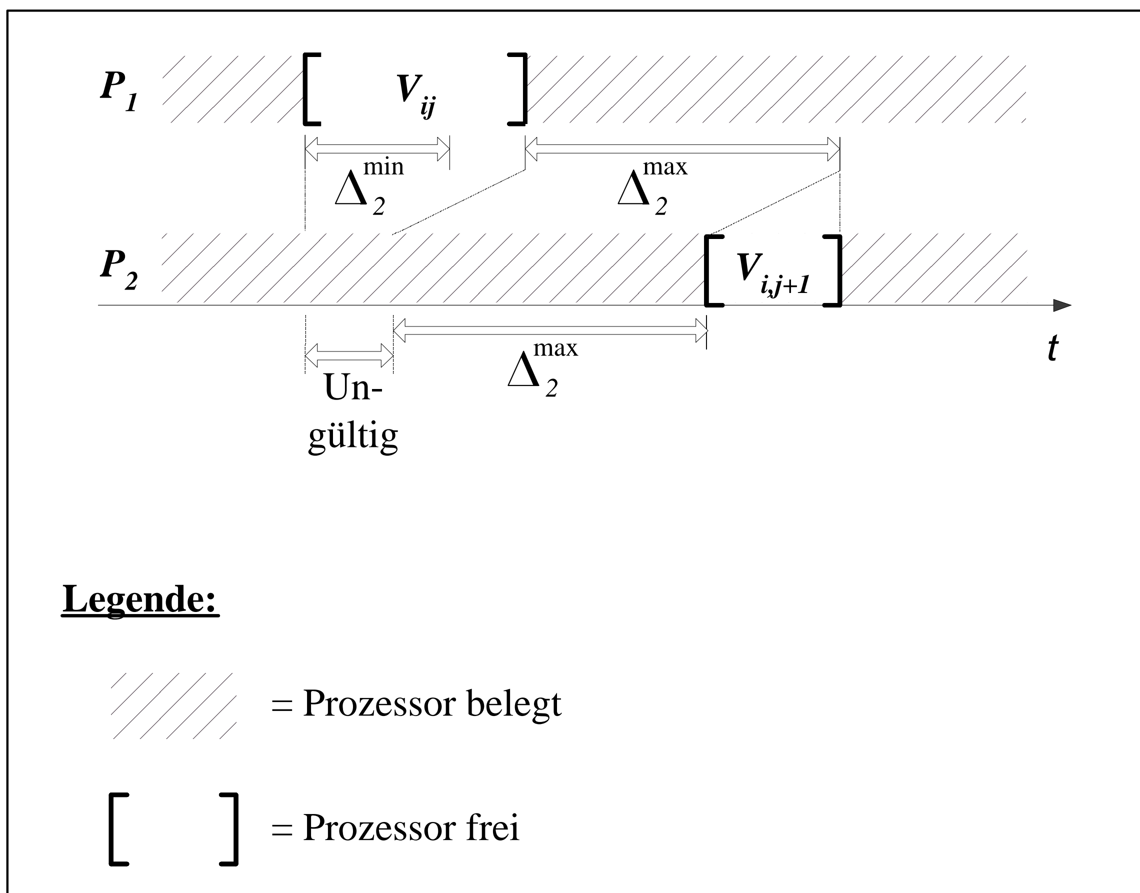

Abbildung 4.1: Ungültige Bereiche bei einfacher Rekursion

```
1  Eingabe: V_i1, μ_i1,...,μ_in_i, v_i1,...,v_im
2  for j = 1 to n_i
3    min_ij := t_i,j−1 + p_i,j−1 + Δ^min_ij
4    max_ij := t_i,j−1 + p_i,j−1 + Δ^max_ij
5    V_ij := Ṽ_ij ∩ [min_ij, max_ij]
6    for r = j − 1 down_to 1
7      V̂_i,r := ∅
8      for [a, b] maximales Teilintervall von V_i,r+1
9        V̂_ir := V̂_ir ⋃ ([a − p_ir − Δ^max_i,r+1, b − p_ir − Δ^min_i,r+1] ⋂ V_ir)
10     end
11     V_i,r := V̂_ir
12     if V_ir leer
13       Ausgabe 'Keine Lösung möglich'
14       Programmende
15     end
16   end
17 end
18 for j = n_i down_to 1
19   Wähle t_ij ∈ V_ij
20 end
21 Ausgabe 'Startzeiten: ', t_i1,...,t_in_i
```

Abbildung 4.2: Algorithmus für das Profiltermin-Planungsproblem

ausgeführt. Diese engen die Intervallmengen $V_{ir-1}, \ldots, V_{i1}$ wie oben beschrieben ein. Dabei wird in Zeile 7–11 jedes $V_{i,r-1}$ auf diejenigen Bereiche eingeschränkt, die aufgrund des Minimal- und Maximalabstands von einem Punkt in V_{ir} erreichbar sind.
Nachdem alle V_{ij} berechnet sind, wird in Zeile 19–21 ein t_{ij} aus jedem V_{ij} ausgewählt und die t_{ij} als Lösung des Planungsproblems ausgegeben. Bei der Wahl der Zeitpunkte aus den Intervallen empfiehlt es sich, immer den frühestmöglichen Zeitpunkt zu nehmen, um den Fertigstellungszeitpunkt des Auftrags zu minimieren.

4.1.1 Komplexität

Die zwei äußeren Schleifen werden jeweils $O(n_i)$-mal ausgeführt. Die Komplexität der inneren Schleife (Zeile 8–10) ist gleich der Anzahl der auftretenden maximalen Teilintervalle von V_{ir}. Diese Anzahl ist durch die Anzahl maximaler Teilintervalle von V_{ir} vor der Ausführung des Algorithmus, also der Anzahl maximaler Teilintervalle im ursprünglichen Belegungsplan des Prozessors beschränkt, da der Algorithmus die Teilintervalle der Mengen V_{ir} immer nur jeweils von unten oder von oben einschränkt, aber nie ein Intervall in zwei Intervalle teilt[1]. Wenn also

$$\xi = \max_{1 \leq j \leq n_i} \{\text{Anzahl max. Teilintervalle von } \tilde{V}_{ij}\}$$

gesetzt wird, kann die Anzahl max. Teilintervalle von V_{ij} nach oben durch ξ abgeschätzt werden und für die Laufzeit des Algorithmus ergibt sich $O(n_i^2 \xi)$, was polynomielle Komplexität

[1] Ein formaler Beweis wird hier aus Zeit- und Platzgründen nicht geführt

bedeutet.

4.2 Der Algorithmus zur Serienterminvergabe

Ein Serientermin (oder Terminserie) ist eine Anzahl Termine, die in ungefähr gleichem Abstand zueinander liegen und bei denen jeweils dieselbe Kombination aus Patient, Gerät(en) und Mitarbeiter(n) beteiligt ist und dieselbe Maßnahme ausgeführt wird. Die Anzahl der Einzeltermine wird dabei entweder direkt vom Benutzer festgelegt oder durch Angabe eines Enddatums. „Ungefähr gleicher Abstand“ heißt, daß die Abstände innerhalb festgelegter Toleranzgrenzen liegen.
Es sind also Startzeitpunkte t_{ij} und eine Toleranz δ gegeben sowie die Aufgabe, für jedes $j \in \{1, \ldots, n_i\}$ einen Zeitpunkt für die Operation O_{ij} innerhalb des Intervalls $[\hat{t}_{ij} - \delta, \hat{t}_{ij} + \delta]$ festzulegen, wobei die Prozessoren $\mu_{ij} \subseteq P$ während der Dauer der Operation verfügbar sein müssen. Diese Problemstellung ist wesentlich einfacher zu lösen als die der Profilterminvergabe, denn es bestehen keine gegenseitigen zeitlichen Abhängigkeiten zwischen den einzelnen Operationen.
Der Algorithmus zur Lösung des Serientermin-Planungsproblems fällt daher deutlich weniger komplex aus, wie anhand Abb. 4.2 erkennbar ist. Wie leicht zu sehen ist, hat dieser Algorith-

```
 1  Eingabe: $\hat{t}_{i1}, \hat{t}_{i2}, \ldots, \hat{t}_{in_i}, \delta$
 2  for j = 1 to $n_i$
 3    if $\exists t : \hat{t}_{ij} - \delta \leq t \leq \hat{t}_{ij} + \delta$
      und alle $\mu \in \mu_{ij}$ während $[t, t + p_{ij}]$ verfügbar
 4      $t_{ij} = t$
 5    else
 6      $t_{ij} = \infty$
 7    end
 8  end
 9  for j = 1 to $n_i$
10    if $t_{ij} \neq \infty$
11      Ausgabe ‘Operation’, j, ‘geplant zum Zeitpunkt’, $t_{ij}$
12    else
13      Ausgabe ‘Prozessor(en) für Operation’, j, ‘nicht verfügbar’
14  end
```

Abbildung 4.3: Algorithmus für das Serientermin-Planungsproblem

mus die Laufzeit $O(n_i)$, also lineare Komplexität bezüglich der Anzahl der Operationen.

Kapitel 5

Implementierung

Wie eingangs erwähnt, entstand ihm Rahmen der vorliegenden Arbeit ein Prototyp der neuen Medora-Terminplanung. In diesem Kapitel wird der Prototyp und seine Implementierung näher erläutert. Die Informationen über 2- und 3-schichtige Applikationsarchitekturen und EJB sind [NETg 00] und [Monson 01] entnommen.

5.1 Schichten bei Anwendungsarchitekturen

Software-Applikationen, die mit einer Datenbank arbeiten, werden in 2- und 3-Schicht-Modelle eingeteilt[1]. Jede Schicht repräsentiert dabei eine Ebene der Informationsverarbeitung und ist zumindest logisch, meistens auch physikalisch von den anderen getrennt, d. h. läuft im Allgemeinen auf verschiedenen Rechnern.
Die bisherigen Medora-Module basieren auf dem 2-Schicht-Modell, die neue Terminplanung auf dem 3-Schicht-Modell.

5.1.1 2-Schicht-Modell

Beim Zweischichtmodell) (5.1 gibt es eine Anzahl von Klientenrechnern, die zusammen die *Benutzerschicht* ausmachen, und einen Datenbankrechner, der die *Datenbankschicht* realisiert. Jeder Klientenrechner kommuniziert direkt mit dem Datenbankrechner, es baut also jeder Klient eine eigene Datenbankverbindung auf.
Da das Herstellen von Datenbankverbindungen zeit- und ressourcenintensiv ist und die maxi-

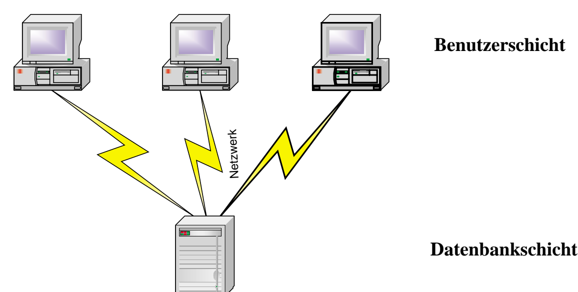

Abbildung 5.1: Das 2-Schicht-Modell

[1]Es gibt auch Modelle mit mehr als 3 Schichten, auf die hier jedoch nicht eingegangen werden soll

male Anzahl offener Datenbankverbindungen beschränkt ist, hat man beim 2-Schicht-Modell die Wahl zwischen ständigem Öffnen und Schließen der Datenbankverbindung, was zeitintensiv ist, und dauerhaften Datenbankverbindungen, was zur Überlastung des Datenbankservers wegen zu hoher Anzahl von Verbindungen führen kann. Das Zweischichtmodell eignet sich daher vor allem für Anwendungen, bei denen die Anzahl der Benutzer auf eine niedrige Zahl beschränkt ist.

5.1.2 3-Schicht-Modell

Beim Dreischichtmodell (5.1) gibt es zwischen der Benutzerschicht und der Datenbankschicht noch die *Vorgangsbearbeitungs-Schicht*. Diese bietet Klienten, also den Rechnern, die die Benutzerschicht ausmachen, die Ausführung von *Vorgängen* an, ohne daß der Klient mit der Datenbank in Kontakt treten muß oder wissen muß, um welche Datenbank-Serversoftware es sich handelt. Die Vorgangsbearbeitung wird von einem *Applikationsserver* erledigt. Dieser ist physikalisch von der ersten und der dritten Schicht getrennt, kommuniziert also mit diesen über Netzwerkverbindungen.
Da alle Operationen, die auf der Datenbank stattfinden, ausschließlich vom Applikationsser-

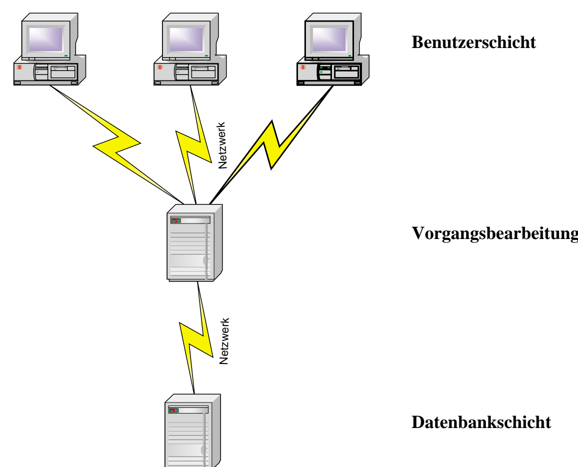

Abbildung 5.2: Das 3-Schicht-Modell

ver getätigt werden, kann dieser Datenbankverbindungen wiederverwenden und auf Vorrat halten („Connection Pooling“). Es muß daher nicht für jeden Klienten eine eigene Datenbankverbindung reserviert werden, sondern kann dem Klienten bei Bedarf zur Verfügung gestellt und anschließend einem anderen Klienten zugeteilt werden. Die zeitliche Verzögerung auf Seiten des Klienten durch das Verbinden zur Datenbank fällt ebenfalls weg, da Klienten eine bereits bestehende Verbindung beim Applikationsserver anfordern können.
Beim 3-Schicht-Modell kann der Datenbankserver kleiner dimensioniert werden als beim 2-Schicht-Modell, da ein Applikationsserver auch einen großen Teil der Last des Datenbankservers übernimmt, indem er als Cache für Datensätze fungiert.
Die genannten Eigenschaften sind der Hauptvorteil des Dreischichtmodells, nämlich der *Skalierbarkeit*. Damit ist gemeint, daß die Zahl der Benutzer prinzipiell unbegrenzt ist, da sie nicht mehr vom Datenbankserver alleine abhängt. Die Schicht der Vorgangsbearbeitung ist ebenfalls skalierbar, indem die Last auf mehrere Applikationsserver verteilt wird (sog. Clustering).

Das dreischichtige Applikationsmodell hat außerdem den Vorzug, daß es einfach ist, völlig verschiedenartige Klienten an das System anzubinden. Die Schnittstelle zum Applikationsserver kann z.B. von einem Klienten über einen Web-Browser, von einem anderen über eine Windows-Anwendung und von einem dritten über ein Programm unter Linux angesprochen werden.

5.2 Verwendete Softwareprodukte

Es sollen hier zunächst die Entwicklungswerkzeuge vorgestellt werden, die für die neue Terminplanung verwendet wurden. Danach wird auf bereits bestehende Softwarekomponenten eigegangen werden, die in die Terminplanung integriert wurden.

5.2.1 Java

Die objektorientierte Programmiersprache Java wurde Anfang der 90er Jahre von der Firma Sun Microsystems entwickelt. Eine der wichtigsten Eigenschaften von Java ist die hohe Abstraktion von rechner- und betriebssystemspezifischen Details. Beispielsweise muß der Programmierer zum Öffnen eines Fensters auf dem Bildschirm nicht über die zahllosen Betriebssystemaufrufe Bescheid wissen, die in Programmiersprachen wie C++, die diesen Grad der Abstraktion nicht besitzen, für eine solche Einzelaktion nötig wären. In Java wird aus der Programmierersicht lediglich ein Fensterobjekt erzeugt und angezeigt, um das eigentliche Anlegen des Fensters kümmert sich die sogenannte *Virtuelle Maschine* (JVM) von Java.
Solche Abstrahierungen haben immer eine Tendenz zum Verlust von Flexibilität, wie z. B. bei Oracle Forms ersichtlich ist (siehe Kap. 1). Beim Entwurf der Architektur von Java wurde daher ein Augenmerk darauf gelegt, daß alle Funktionalität, die im kleinsten gemeinsamen Nenner der auf dem Markt befindlichen Betriebssysteme wie Windows, Linux, MacOS etc. von Java unterstützt wird oder zumindest vom Programmierer ohne allzu großen Aufwand realisierbar ist.
Kompilierte Java-Programme liegen in Form eines *Bytecodes* vor. Dieser kann ohne Neukompilierung auf jedem Rechnersystem ausgeführt werden, auf dem eine JVM läuft. Die ersten JVM waren lediglich Interpreter, die jeden Bytecode-Befehl vor der Ausführung einzeln in eine Folge von Maschinenbefehlen des jeweiligen Rechners umwandelten; moderne JVM führen vor der Ausführung eines Java-Programmes zunächst eine Vorkompilierung durch, die rechnerspezifischen Code erzeugt und führen diesen dann aus, wodurch sich die Programmlaufzeit deutlich verbessert.
Als Basispaket für den Java-Entwickler vertreibt Sun das *Java Development Kit* (JDK). Darin ist eine JVM, ein Java-Compiler sowie alle Standardbibliotheken (u. a. Graphische Komponenten, Datenbankschnittstelle, Netzwerkkommunikation) enthalten. Zum Zeitpunkt der Drucklegung war JDK 1.4 die aktuelle Version.
Die Vorzüge von Java liegen in der weitgehenden Betriebssystem- und Rechnerunabhängigkeit der damit erzeugten Anwendungen, der Eleganz der Sprache sowie der breiten Unterstützung, die es mittlerweile erfährt.
Das größte Problem bei Java-Anwendungen ist deren Ausführungsgeschwindigkeit, besonders die Startzeit, d. h. die Zeitspanne zwischen dem Aufruf des Programms und dessen Erscheinen auf dem Bildschirm. Das liegt daran, daß vor der eigentlichen Java-Anwendung zuerst die JVM gestartet werden muß, da diese normalerweise nicht zusammen mit dem Betriebssystem gestartet wird. Dazu kommt noch der Zeitbedarf für das Vorkompilieren des Bytecodes.

5.2.2 J2EE und EJB

J2EE steht für Java 2 Enterprise Edition[2] und stammt ebenfalls von Sun. J2EE enthält mehrere Technologien, von denen EJB (Enterprise Java Beans) die bedeutendste ist. EJB ist ein Standard für 3-Schicht-Architekturen unter Java, der eine Brücke zwischen der objektorientierten Java-Welt und der tabellenbasierten Datenbankwelt legt. Im EJB-Standard sind Anforderungen an den Applikationserver festgelegt sowie die Schnittstelle zwischen dem Applikationsserver und dem Klienten. Der Standard wurde erstmals 1998 veröffentlicht und liegt zur Zeit in der Version 2.0 vor ([Sun 02]). J2EE ist derzeit noch die am weitesten verbreitete Plattform für Mehrschichtanwendungen, wird aber in absehbarer Zeit durch das neu erschienene *.NET*-Konzept von Microsoft[3] unter starken Druck geraten.

Bei EJB wird die Vorgangsbearbeitung von *Enterprise-Beans* durchgeführt. Ein Enterprise-Bean ist aus der Sicht des Klienten ein Java-Objekt, das bestimmte Methoden anbietet, um Vorgänge durchzuführen. Die Bean-Methode führt dann Code aus, der die Datenbankschnittstelle in entsprechender Weise bedient, um den Vorgang durchzuführen. Ein solcher Vorgang könnte z. B. eine Abhebung von einem Geldautomaten sein, oder die Neuaufnahme eines Patienten in einem Krankenhaus. Der Klient kommt also nie direkt mit der Datenbank in Kontakt, sondern gibt nur dem Bean den Auftrag und überläßt es diesem, die entsprechenden Datenbankoperationen durchzuführen.
Der EJB-Standard legt verschiedene Einschränkungen für Enterprise-Beans fest, u. a. daß eine Enterprise-Bean keine Dateien manipulieren, Ausführungsstränge (Threads) erzeugen und nicht auf Peripheriegeräte des Servers (Bildschirm, Tastatur) zugreifen dürfen ([Sun 02], Abschn. 24.1.2). Diesen Einschränkungen liegt zugrunde, daß der vorrangige Zweck einer Enterprise-Bean ist, Aufträge des Klienten zu bearbeiten und diese Fähigkeit in bestmöglichem Maße gewährleistet werden soll.

Es gibt drei Arten von Enterprise-Beans: *Entity-Beans*, *Session-Beans* und *nachrichtengesteuerte* (engl. message-driven beans).
Ein Entity-Bean ist ein Java-Objekt, das einen Datensatz der Datenbank repräsentiert. Die Spalten der Datenbank entsprechen je einer Variable des Bean-Objektes, auf die seitens des Klienten schreibend und/oder lesend zugegriffen werden kann. Zusätzlich können auch noch weitere Methoden definiert werden, die komplexere Vorgänge ausführen. Als Beispiel für eine Entity-Bean wäre die Bean `Patient` denkbar, oder auch `Termin`.
Entity-Beans gliedern sich in Beans mit *container-verwalteter* und solche mit *bean-verwalteter* Persistenz. Bei ersteren wird lediglich eine Datenbanktabelle und deren Spalten angegeben, und der Applikationsserver (auch Container genannt) übernimmt die Implementierung der Lese- und Schreiboperationen auf der Datenbank. Bei bean-verwalteter Persistenz hingegen muß der Bean-Entwickler selber alle Methoden implementieren, die das Bean zur Verfügung stellt. Das bedeutet in der Praxis, die entsprechenden SQL-Anweisungen über JDBC[4] an die Datenbank abzuschicken.
Session-Beans sind – im Gegensatz zu Entity-Beans, die dauerhaft durch Datensätze in der Datenbank repräsentiert werden – von eher kurzlebiger Natur. Eine Session-Bean modelliert einen Vorgang, der nicht in der Datenbank niedergelegt wird. Die Auswirkungen des Vorganges können natürlich sehr wohl in der Datenbank sichtbar sein und sind es im Regelfall auch.

[2] Java trägt seit der Version 1.2 den offiziellen Namen „Java 2“
[3] `http://www.microsoft.com/net/`
[4] JDBC ist die Datenbankschnittstelle von Java

Zum Beispiel könnte ein Reisebüro eine Session-Bean `Buchung` benutzen, die etwa die Methoden `setzeFlugziel`, `setzeHotel`, `setzeZahlungsmethode`, `buchungAbschliessen` und `buchungStornieren` hat.
Von Session-Beans gibt es zwei Arten, zustandsbehaftete und zustandslose Session-Beans. Eine zustandsbehaftete Session-Bean ist während ihrer gesamten Lebensdauer genau einem Klienten zugeordnet. Nur für diesen Klienten ist die Bean sichtbar, und nur er kann die Bean benutzen. Eine zustandsbehaftete Session-Bean kann Daten in Form von Instanzvariablen zwischen Methodenaufrufen speichern, so daß nachfolgende Methodenaufrufe von diesen Variablen abhängig gemacht werden können. Daher die Bezeichnung „zustandsbehaftet“.
Zustandslose Session-Beans können zwar genauso Instanzvariablen besitzen, jedoch werden Beans dieser Art zwischen Klienten ausgetauscht, was bewirkt, daß eine Bean zwischen zwei Methodenaufrufen den Klienten wechseln kann und umgekehrt ein Klient es bei zwei Methodenaufrufen an die gleiche Bean mit zwei verschiedenen Instanzen der Bean zu tun haben kann. Methodenaufrufe bei zustandslosen Session-Beans sind also aus Klientensicht völlig voneinander isoliert und unabhängig.
Nachrichtengesteuerte Beans gibt es seit der EJB-Version 2.0. Diese definieren keine Methoden, die Klienten aufrufen können, sondern empfangen Nachrichten und führen dann Transaktionen durch. Beispielsweise könnte eine Bean namens `TerminAnfordVerarbeiter` als Teil eines RIS Terminanforderungen von anderen Krankenhausabteilungen empfangen und automatisch entsprechende Termine reservieren.

Bevor ein Klient eine Enterprise-Bean benutzen kann, muß zunächst eine Bean erzeugt oder eine existierende aufgefunden werden.
Nachrichtengesteuerte Beans werden automatisch vom Applikationsserver erzeugt, wenn eine entsprechende Nachricht eintrifft; Entity- und Session-Beans definieren eine Schnittstelle namens *Home-Interface*, mittels derer ein Klient oder eine andere Bean eine Referenz zu der Bean beziehen können. Dies kann durch Erzeugung einer neuen Bean geschehen; bei Entity-Beans kann auch eine vorhandene Bean in der Datenbank gesucht werden.
Entity- und Session-Beans stellen außerdem ein *Remote-Interface* zur Verfügung. Darin sind die Methoden festgelegt, die eine Instanz einer Bean dem Klienten zur Vorgangsbearbeitung anbietet. Die Implementierung einer Enterprise-Bean steckt in der *Bean-Klasse*. Diese implementiert bei Session- und Entity-Beans das Remote-Interface und bei nachrichtengesteuerten Beans das Interface `javax.jms.MessageListener`[5].

Wie man sieht, sind bei EJB die Schnittstellen zwischen den drei Schichten standardisiert: zwischen der Benutzerschicht und der Vorgangsbearbeitung liegen die Home- und Remote-Interfaces bzw. bei nachrichtengesteuerten Beans JMS, der Standard zum Nachrichtenaustausch unter Java. Die Schnittstelle von der Vorgangsbearbeitung zur Datenbankschicht ist durch die Java-Datenbankschnittstelle JDBC festgelegt. Dadurch ist bei einem EJB-basierten System der Applikationsserver beliebig gegen einen anderen austauschbar[6].

EJB in der Praxis

Idealerweise würde man alle Entitäten der wirklichen Welt – von diesen ist normalerweise jede als eine eigene Datenbanktabelle realisiert – auf Entity-Beans abbilden und könnte somit das relationale Datenbankmodell durchgängig objektorientiert modellieren. In der Praxis ist

[5]Dies gilt für Beans, die JMS-Nachrichten empfangen. Zur Zeit ist JMS der einzige unterstützte Standard zur Nachrichtenübermittlung, in Zukunft könnten aber weitere wie SMTP oder JAXM hinzukommen (siehe auch [Monson 01]

[6]Wie immer gilt das natürlich nur, solange nicht ein Hersteller über eigene Erweiterungen den Standard an sich reißt.

dieser Weg jedoch ungangbar, da mit steigender Zahl von Instanzen von Entity-Beans der Applikationsserver leistungsmäßig sehr schnell überfordert wird. Man ist deshalb als Entwickler gezwungen, zur Verabeitung großer Mengen an Datensätzen Session-Beans zu benutzen und Entity-Beans nur zur Manipulation einzelner Datensätze einzusetzen. Beim Terminplanungssystem werden beispielsweise Termine von einer Session-Bean eingelesen und zur Erstellung eines neuen Termins oder Veränderung eines bestehenden Termins eine Entity-Bean benutzt. Es bleibt abzuwarten, ob diese Problematik mit zunehmendem Reifegrad der Applikationsserver irgendwann beseitigt wird.

5.2.3 Orion

Orion ist eine EJB-konforme Applikationsserver-Software der schwedischen Firma Ironflare[7]. Es gibt zahlreiche Konkurrenzprodukte auf dem EJB-Servermarkt, z.B. BEA WebLogic[8], IBM WebSphere[9], Sybase Enterprise Application Server[10] oder JBoss[11]. Der Orion-Server wurde für die Terminplanung gewählt, weil es einer der leistungsfähigsten Server ist, aber trotzdem ein gutes Preis/Leistungsverhältnis hat.

5.2.4 Forte

Als Entwicklungsumgebung für den Terminplaner kam die graphische Java-Entwicklungsumgebung Forte zum Einsatz[12]. Forte ist aus dem Open-Source-Projekt NetBeans[13] hervorgegangen und ist ein Produkt von Sun Microsystems.

5.2.5 SourceSafe

Zur Versionsverwaltung wurde Visual SourceSafe von Microsoft verwendet. SourceSafe verwaltet die Dateien eines Projektes und sorgt dafür, daß Änderungen, die verschiedene Benutzer an derselben Datei vornehmen, miteinander abgeglichen werden. Dadurch wird verhindert, daß ein Entwickler die Änderungen eines anderen Entwicklers überschreibt. SourceSafe speichert auch die Versionen jeder Datei eines Projektes, so daß auf alle Vorgängerversionen einer Datei zurückgegriffen werden kann.
SourceSafe ist ein relativ einfaches System, das jedoch für die Entwicklung des Terminplaners ausreichend war. Wenn es nötig ist, daß mehrere Entwickler gleichzeitig eine Quelldatei editieren, sind mächtigere Produkte wie CVS oder ClearCase anzuraten.

5.2.6 Oracle 8.06

Medora ist eng an Oracle als Datenbank gebunden, da ein Großteil der Funktionalität in PL/SQL geschrieben ist. PL/SQL ist eine Oracle-eigene Programmiersprache, die auf SQL basiert und es ermöglicht, auf dem Datenbankserver Programmbibliotheken zu plazieren, deren Code dann vom Klient ausgeführt werden kann.
Darüberhinaus werden in Medora noch andere Oracle-spezifische Funktionen wie Datenbank-Jobs, Sequences, Volltextrecherche usw. verwendet.

[7] http://www.orionserver.com/
[8] http://www.beasys.com/products/servers_application.shtml
[9] http://www.ibm.com/software/webservers/appserv/
[10] http://www.sybase.com/products/easerver/
[11] http://www.jboss.org/
[12] Es wurde Forte CE 3.0 verwendet. Mittlerweile wird Forte unter dem Namen „Sun One Studio“ vermarktet.
[13] http://www.netbeans.org/

Die Gebundenheit an Oracle wirft beim Terminplanungssystem jedoch keine Probleme auf, da der Applikationsserver mittels SQL auf die Datenbank zugreift und alle Datenbankoperationen, die für den Terminplaner notwendig sind, über SQL abgewickelt werden können.

5.2.7 JFCSuite

JFCSuite[14] ist eine Sammlung von graphischen Komponenten, die bei der Entwicklung von Java-Anwendungen verwendet werden können. Darunter sind verschiedene Arten von Eingabefeldern, Tabellen und Listen; ferner ist eine Datumsauswahl- und Monatskalenderkomponente und eine Terminplanerkomponente enthalten. Die Terminbücher des Terminplaners basieren auf der Terminplanerkomponente von JFCSuite, außerdem werden im Terminplaner die Monatskalenderkomponente und mehrere Arten von Eingabefeldern (numerische Werte, Datum, String) verwendet.
Der Einsatz der JFCSuite-Komponenten im Terminplaner war allerdings nur nach umfangreichen Erweiterungen und Ergänzungen möglich, da sich vor allem die Terminplanerkomponente in der ursprünglichen Form als unzureichend für den Einsatz in der Medora-Terminplanung herausstellte. Beispielsweise unterstützt die Original-Terminplanerkomponente kein Drag-and-Drop und keine Kontextmenüs und ist allgemein sehr unflexibel, da es intern nicht aus Java-Standardkomponenten aufgebaut ist.

5.3 Benutzeroberfläche

Dieser Abschnitt geht auf den äußerlichen Aufbau und die Funktionsweise der Terminplaneroberfläche ein. Es sei an dieser Stelle noch einmal darauf hingewiesen, daß die neue Medora-Terminplanung ein Prototyp und keineswegs ein ausgereiftes Produkt ist. Deshalb sind Teile der Funktionalität noch nicht implementiert und hat dessen Aussehen noch nicht die Qualität, die man von einem fertigen Produkt erwarten würde. Auch ist die Benutzeroberfläche nicht einheitlich in einer Sprache gehalten; diese muß in der Endversion an das jeweilige Land, in dem die Software zum Einsatz kommt, angepaßt sein.
Die gravierendste Unzulänglichkeit der Oberfläche des bisherigen Terminplaners war deren Unübersichtlichkeit und mangelnde intuitive Bedienbarkeit. Das lag zum einen daran, daß häufig benutzte Aktionen wie das Anlegen eines neuen Termines auf viele hintereinander erscheinende Einzeldialoge verteilt waren und zum anderen, daß viele heutzutage selbstverständliche und vom Durchschnittsanwender erwartete Funktionen wie Kontextmenüs, verschiebbare Trennlinien und das Ziehen von Objekten fehlen. Im neuen Terminplaner sind daher in einem Hauptfenster alle wichtigen Informationen und Funktionalitäten integriert (Abb. 5.3).
Das Hauptfenster ist in fünf Funktionsbereiche aufgeteilt (vgl. Abb. 5.4):

- die Steuerungsleiste
- die Monatsübersicht
- die Multifunktionsleiste
- die Komponente „Weitere Termine / Abwesenheitsliste“
- nd schließlich den wichtigsten Teil, die Terminübersicht.

[14] `http://www.infragistics.com/products/java/jfcsuite.asp`

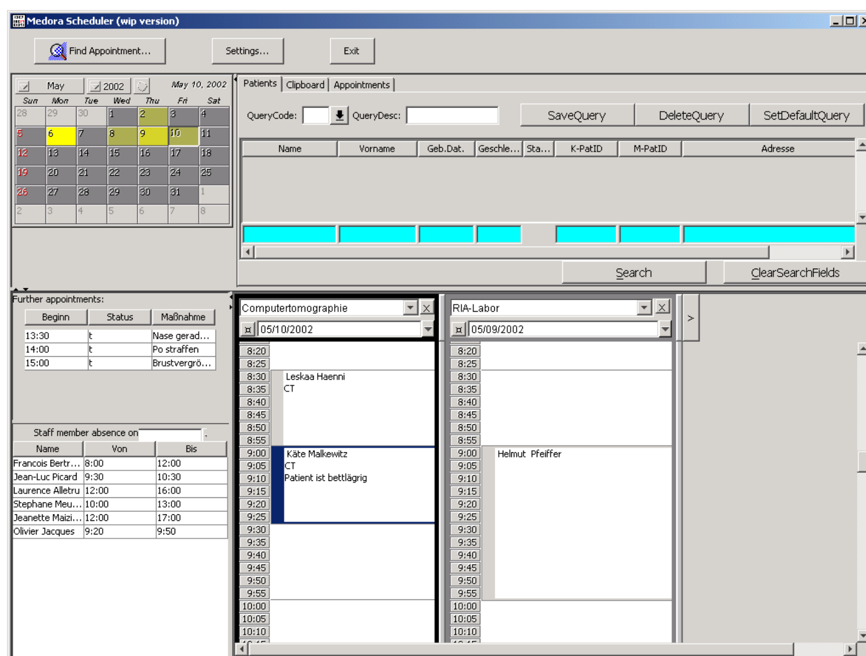

Abbildung 5.3: Hauptfenster des Terminplaners

Die fünf Funktionsbereiche sind durch verschiebbare Trennlinien voneinander getrennt. Dadurch kann der Benutzer mehr Platz für Terminbücher oder Patientenlisten schaffen, wenn dies nötig ist. Alle Bereiche außer der Terminübersicht können auch zusammengeklappt werden, wodurch der freiwerdende Platz dem angrenzenden Bereich zur Verfügung gestellt wird.

5.3.1 Steuerungsleiste

Dieser Bereich enthält die globalen Schaltflächen des Programms, welche als Ersatz für ein Hauptmenü dienen. Der Entscheidung gegen ein Hauptmenü lagen diesbezügliche Wünsche von Kundenseite zugrunde.

5.3.2 Monatsübersicht

Dies ist ein Monatskalender, der den Monat des aktuell in der Terminübersicht angezeigten Terminbuchs darstellt. Jeder Tag ist mit einer Farbe hinterlegt, die dem Belegungszustand der jeweiligen Ressource entspricht. So kann sich der Benutzer schnell einen Überblick über den Belegungszustand der Ressource verschaffen.

5.3.3 Multifunktionsleiste

Die Multifunktionsleiste vereinigt in Form eines Registers (engl. tab control) drei Funktionalitäten, von denen jede in einem separaten Reiter untergebracht ist:

- In der *Patientenauswahl* können Patienten nach Suchkriterien angezeigt werden sowie Termine für einen Patient angelegt werden, indem der Patient auf die Terminübersicht gezogen wird. Der Terminplaner erstellt dann einen neuen Termin für diesen Patienten zur der Zeit, die dem Ort des Loslassens entspricht.

Abbildung 5.4: Aufteilung des Hauptfensters

- Die *Zwischenablage* speichert eine beliebige Anzahl Termine. Diese können von einem Terminbuch auf die Zwischenablage gezogen werden und umgekehrt. Der Termin wird dabei standardmäßig verschoben; durch Drücken der Strg-Taste wird der Termin stattdessen kopiert, also ein zweites Exemplar angelegt.
- Die *Terminauskunft* ermöglicht es, anhand von Suchkriterien nach bestehenden Terminen zu suchen und diese als Liste anzuzeigen.

5.3.4 Weitere Termine / Abwesenheitsliste

Hierin werden zu dem in der Terminübersicht ausgewählten Termin alle folgenden Termine desselben Patienten angezeigt sowie die zum ausgewählten Tag abwesenden Mitarbeiter aufgelistet. Die Funktionalität dieser Komponente ist zur Zeit noch nicht implementiert.

5.3.5 Terminübersicht

Die Terminübersicht besteht aus einem oder mehreren Terminbüchern, die nebeneinander dargestellt werden. Ein Terminbuch stellt die Termine einer *Ressource* an einem bestimmten Tag dar. Eine Ressource kann ein Mitarbeiter, ein Gerät oder ein Arbeitsplatz sein; für ein späteres Entwicklungsstadium des Terminplaners ist vorgesehen, daß auch Mitarbeiter- und Gerätegruppen als Mitarbeiter bzw. Geräte benutzt werden können.
Für jede Ressource ist ein sog. Ressourcenraster festgelegt. Wenn ein neuer Termin angelegt wird, richtet der Terminplaner den Beginn und das Ende des Termins an den Rasterzeilen aus. Natürlich kann der Benutzer den Beginn und die Länge des Termins auch frei setzen. Es kann für jede Ressource ein Standardraster sowie zusätzliche Raster für beliebige Wochentage und Feiertage definiert werden.
Durch einen Doppelklick auf ein leeres Rasterfeld oder einen Termin öffnet sich der Termindialog (siehe nächsten Abschnitt).

Abbildung 5.5: Der Termindialog

5.3.6 Termindialog

Beim Anlegen eines Termins über den Termindialog (Abb. 5.5) ist die Angabe eines Patienten Pflicht. Wenn der Benutzer Patienteninformationen einträgt, die auf mehrere Patienten zutreffen, öffnet sich das Patientenauswahlfenster (Abb. 5.6). Wenn es keinen Patienten gibt, auf den die eingegebenen Informationen zutreffen, hat der Benutzer die Möglichkeit, einen neuen Patienten mit diesen Informationen anzulegen. Im Termindialog kann der Benutzer auch Serientermine anlegen. Dazu schaltet man vom Einzeltermin- in den Serienterminmodus um (Abb. 5.7) und kann dann mittels des Serientermindialogs eine Terminserie anlegen (Abb. 5.8). Es können durch mehrmaliges Aufrufen des Serientermindialogs auch mehrere Serien zu einer kombiniert werden. Zum Anlegen der Termine wird der Algorithmus aus Abschnitt 4.2 verwendet.

5.4 Architektur der Benutzerschicht

Abb. 5.9 zeigt einen groben Überblick über die Klassenstruktur der Benutzeroberfläche. Man sieht, daß die Klassenstruktur über weite Strecken analog mit dem optischem Aufbau der Oberfläche ist. So implementiert die Klasse `FrmMain` das Hauptfenster, und die fünf darunterliegenden Funktionsbereiche werden durch Instanzen der Klassen `JPanel` (Steuerungsleiste), `JPVCalendar` (Monatsübersicht), `JTabbedPane` (Multifunktionsleiste), `JPanel` (Weitere Termine / Abwesenheitsliste) und `ResourceView` (Terminansicht) realisiert. Jede dieser fünf Komponenten hat mehrere Unterkomponenten, z. B. enthält `ResourceView` ein oder mehrere Terminbücher (`JPVDay`), über denen sich jeweils eine Komponente zur Ressourcen- und Datumsauswahl (`ResourceHeaderCell`) befindet.
Jede Java-Oberfläche verwendet eine große Zahl von Komponenten, die für den Benutzer unsichtbar sind und nur als Behälter für andere Komponenten dienen. Dazu zählen `JPanel`, das eine beliebige Anzahl von Komponenten aufnehmen kann und diese mittels einem definierbaren `LayoutManager` anordnet, und `JSplitPane`, die zwei Komponenten durch eine verschiebbare Linie voneinander trennt. Diese Komponenten sind hier der Übersichtlichkeit

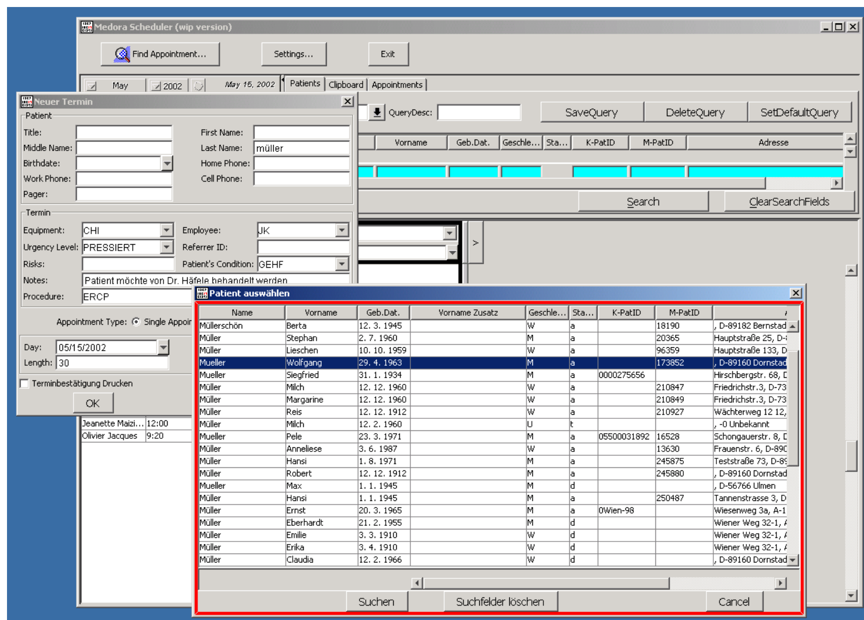

Abbildung 5.6: Patientenauswahl

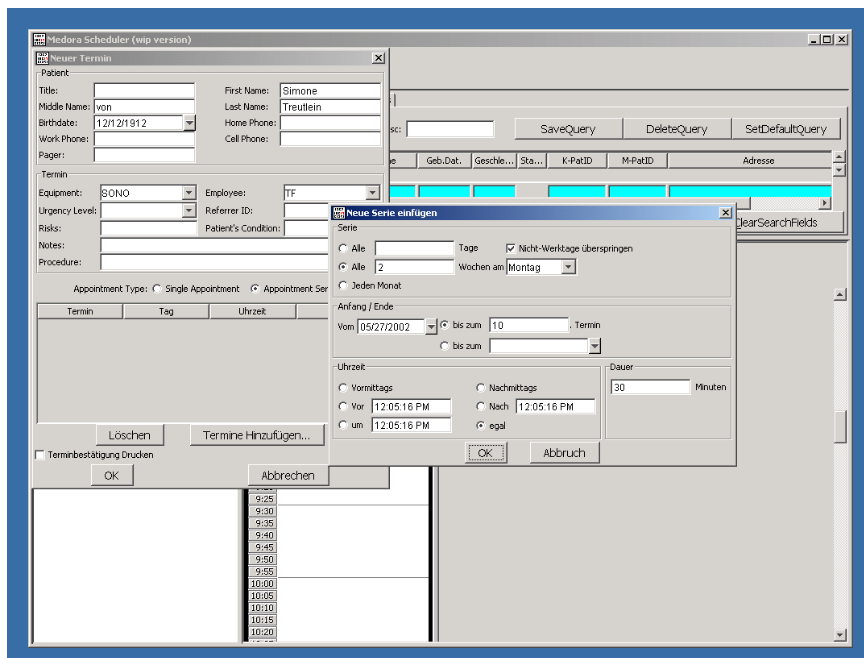

Abbildung 5.7: Der Serientermindialog

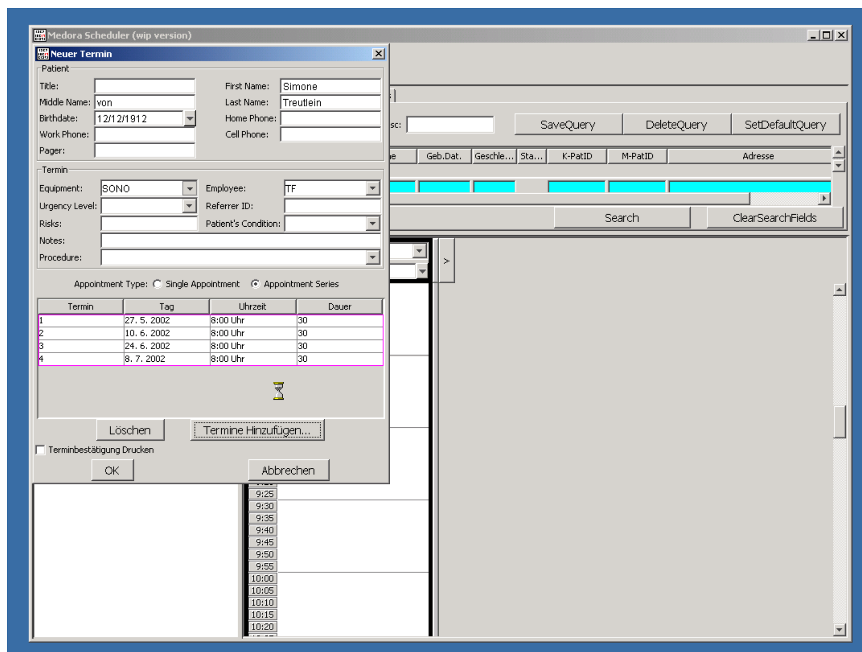

Abbildung 5.8: Serientermin wird angelegt (4 von 10 Terminen erzeugt)

halber nicht aufgeführt.

5.5 Architektur der Vorgangsbearbeitung

Die Vorgangsbearbeitung (mittlere Schicht) des Terminplaners besteht aus Entity- und Session-Beans, die gewissermaßen eine Infrastruktur für die Applikation bereitstellen. Es wurde beim Entwurf dieser Bibliotheken darauf geachtet, daß diese auch für eventuelle spätere Java-Module von Medora geeignet sind.
Im folgenden werden nun die Schnittstellen der einzelnen Beans erläutert.
Generell muß jede Session- und Entity-Bean eine parameterlose Methode namens `create` definieren, mit der eine neue Bean erzeugt werden kann. Darüberhinaus können auch noch weitere `create`-Methoden mit beliebigen Parametern hinzukommen. Bei einem Entity-Bean muß darüberhinaus noch eine Methode `findByPrimaryKey` zum Auffinden existierender Beans über den Primärschlüssel vorhanden sein. Nach dem Muster `findByXXX` können noch weitere Methoden zur Suche über andere Kriterien dazukommen.
Die Home-Interfaces der Beans, die die Terminplanung benutzt, kommen bis auf wenige Ausnahmen mit den Pflichtmethoden `create` und ggf. `findByPrimaryKey` aus, weswegen sie uns nicht im Detail interessieren sollen. Die Remote-Interfaces der Beans hingegen sind für das Verständnis des Systems von wesentlich größerer Bedeutung und werden im folgenden zu jeder Bean angegeben. Zur beseren Lesbarkeit wurde der Code umformatiert und die Javadoc-Kommentare entfernt.
Die Quelltexte und Kommentare sind bei GE Medical Systems grundsätzlich in englischer Sprache gehalten, da mit zahlreichen europäischen und amerikanischen Tochter- und Fremdfirmen Zusammenarbeiten bestehen und eine einheitliche Sprache den Daten- und Informationsaustausch vereinfacht.

JButton

JPanel
Steuerungsleiste

JButton

JPVCalendar
Monatsübersicht

HotBox.Pane
Patientenauswahl

FrmMain
Hauptfenster des Terminplaners

JTabbedPane
Multifunktionsleiste

HotBox.Pane
Terminauskunft

AppointmentClipboard
Zwischenablage

JPanel
Weitere Term. / Abwesenheit

JTable
Abwesenheitsliste

JTable
Weitere Termine

ResourceView
Terminansicht

JPVDay
Terminbuch

ResourceHeaderCell
Ressourcen- und Datumsauswahl

JPVDay
Terminbuch

ResourceHeaderCell
Ressourcen- und Datumsauswahl

Legende:

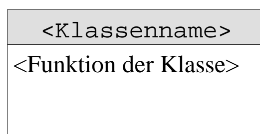

Abbildung 5.9: Vereinfachte Klassenstruktur der Benutzeroberfläche

5.5.1 Session-Beans

ChangeMonitor

Das `ChangeMonitor`-Bean hat die Aufgabe, Änderungen an Terminbüchern festzustellen. Sie hat eine Methode namens `waitForScheduleChange`, welche eine Medora-Ressource, ein Datum und einen Wert für die maximale Wartezeit als Parameter hat. Die Methode verharrt solange im Wartezustand, bis beim Terminbuch der angegebenen Ressource zum angegebenen Tag eine Änderung erfolgt bzw. bis die maximale Wartezeit überschritten ist. Abhängig davon gibt die Methode einen booleschen Wert an den Klienten zurück.
Da der Aufruf der `waitForScheduleChange`-Methode blockierend ist, wird sie im Terminplaner nebenläufig ausgeführt. Jedesmal, wenn die `ChangeMonitor`-Bean eine Änderung an einem auf dem Bildschirm angezeigten Terminbuch registriert, wird das betreffende Terminbuch neu eingelesen. Dadurch werden Terminbücher, die einem Benutzer angezeigt werden, stets mit Änderungen anderer Benutzer konsistent gehalten.

ChangeMonitor.java

```
package com.ge.med.euro.medora.scheduler.bl.changemonitor;

import java.rmi.*;
import javax.ejb.*;
import com.ge.med.euro.medora.scheduler.bl.Resource;
import com.ge.med.euro.medora.util.Date;

/**
 * ChangeMonitor is a stateless session bean which
 * provides methods that detect changes to the database.
 */
public interface ChangeMonitor extends EJBObject
{
    // Waits for <interval> seconds or until an appointment with the
    // specified resource is added, changed, or deleted on the day specified by
    // <date>. Returns true if a change has occured, or false if a timeout
    // occured.
    public boolean waitForScheduleChange(Resource resource,
                                         Date date,
                                         int interval)
                   throws RemoteException;
}
```

DataReader

Dieses Bean enthält die Methoden `readStaff` und `readEquipment` zum Einlesen der Mitarbeiter bzw. Geräte. Diese werden für die Auswahllisten im oberen Teil der Terminbücher benötigt (`ResourceHeaderCell`, siehe auch Abschn. 5.4).
Außerdem gibt es die Methode `readInternationalizedString`, die eine für mehrere Sprachen definierte Zeichenkette aus der Datenbank liest. Da Medora in mehreren Ländern und damit Sprachregionen zum Einsatz kommt, werden Bezeichnungen, Erklärungen, Menüs und dergleichen nicht direkt im Programmcode verwendet, sondern über einen Index angesprochen. Je nach Spracheinstellung der Medora-Installation ergibt sich für den Index dann das

jeweilige sog. *Sprachelement*. Beispielsweise sind zum Index 930 folgende Sprachelemente definiert:

Index	Land	Sprachelement
930	E	¡ No hay lector de tarjetas conectado !
930	F	Un lecteur de carte n'est pas connecté!
930	GER	Es ist kein Kartenlesegerät angeschlossen!
930	I	Non è collegato nessun lettore di tessere!
930	US	No card reading-device has been connected

Die Indizes der Sprachelemente sind in der Java-Implementierung transparent als Elemente einer sog. typensicheren Aufzählungsklasse modelliert. Dieser Begriff bezeichnet eine Klasse, von der eine festgelegte Menge von Instanzen existiert und von der keine zusätzlichen Instanzen angelegt werden können.

DataReader.java

```
package com.ge.med.euro.medora.bl.datareader;

import java.rmi.*;
import javax.ejb.*;
import java.util.List;
import com.ge.med.euro.medora.bl.InternationalizedStringType;
import com.ge.med.euro.medora.bl.EJBServerTools;

/**
 * Provides methods for retrieving miscellaneous types of data from
 * the database
 */
public interface DataReader extends EJBObject
{
    // Retrieves a list of staff from the database. The return
    // value is a List which contains Employee objects.
    public List readStaff() throws RemoteException;

    // Retrieves a list of equipment from the database. The return
    // value is a List which contains PieceOfEquipment objects.
    public List readEquipment() throws RemoteException;

    // Returns a predefined internationalized string from the database
    public String readInternationalizedString(InternationalizedStringType type)
                  throws RemoteException;
}
```

LoggingInitializer

Dieses Bean deklariert die Methode `initializeLogging`. Diese wird einmal beim Programmstart aufgerufen und initialisiert die Protokollierung von Fehler-, Warn- und Überwachungsmeldungen, die von Bean-Methoden erzeugt werden. Da ein Enterprise-Bean nicht in Dateien schreiben und keine Netzwerkverbindungen öffnen darf, werden die Meldungen an den Applikationsserver weitergereicht, der sie dann in die Applikations-Protokolldatei schreibt.

LoggingInitializer.java

```
package com.ge.med.euro.medora.bl.logginginitializer;

import java.rmi.*;
import javax.ejb.*;

public interface LoggingInitializer extends EJBObject
{
    // Sets up a java.util.Handler that forwards
    // all log messages to the Orion logger
    public void initializeLogging() throws RemoteException;
}
```

MedoraCredentials

Dieses Bean reicht den Benutzernamen und das Kennwort für die Datenbank vom Klienten an die Vorgangsverwaltung weiter. Die Methoden `setDatabaseUser` und `setDatabasePassword` werden vom Terminplaner beim Programmstart aufgerufen und speichern Benutzername und Kennwort in dem Bean. Wenn eine Bean Zugang zur Datenbank benötigt, ruft sie `getDatabaseUser` und `getDatabasePassword` auf und meldet sich mit diesen Daten beim Datenbankserver an.
Die Methode `getMedoraUserID` wird von der `Appointment`-Bean beim Anlegen von Terminen benutzt, um den Medora-Benutzer, der den Termin anlegt, zu speichern.

MedoraCredentials.java

```
package com.ge.med.euro.medora.bl.medoracredentials;

import java.rmi.*;
import javax.ejb.*;

/**
 * Provides methods that pass the database user name
 * and password from the application to the middle tier
 */
public interface MedoraCredentials extends EJBObject
{
    public void setDatabaseUser(String user) throws RemoteException;
    public String getDatabaseUser() throws RemoteException;

    public void setDatabasePassword(String user)
                throws RemoteException;
    public String getDatabasePassword() throws RemoteException;

    public long getMedoraUserID(String username) throws RemoteException;
}
```

AppointmentReader

Das `AppointmentReader`-Bean stellt drei Methoden zur Verfügung. `getAppointments` liefert alle Termine zurück, die die angegebene Ressource am angegebenen Tag hat. `getAppointment` gibt den Termin mit dem angegebenen Primärschlüssel zurück. `getFirstAvailableTime` schließlich sucht innerhalb der angegebenen Zeitspanne den ersten möglichen Termin der angegebenen Länge, bei dem der angegebene Mitarbeiter und das angegebene Gerät verfügbar sind.

AppointmentReader.java

```
package com.ge.med.euro.medora.scheduler.bl.appointmentreader;

import java.rmi.*;
import javax.ejb.*;
import java.util.Vector;
import com.ge.med.euro.medora.util.Date;
import com.ge.med.euro.medora.scheduler.bl.*;
import com.ge.med.euro.medora.scheduler.bl.appointment.AppointmentPK;

/**
 * AppointmentReader is a stateless session bean. It provides
 * methods for reading appointments from the database and searching
 * for available times to make appointments.
 */
public interface AppointmentReader extends EJBObject
{
    // Returns all appointments that are scheduled for the specified
    // resource on the specified day. The appointments are instances
    // of pv.util.Appointment.
    public Vector getAppointments(Date day, Resource resource)
                  throws RemoteException;

    // Returns the data for the appointment with the primary key <pk>.
    // If the appointment doesn't exist, RemoteException is thrown.
    public AppointmentData getAppointment(AppointmentPK pk)
                           throws RemoteException;

    // Returns the first available time between <from> and <to> such that an
    // appointment of length <requestedLength> milliseconds can be scheduled
    // with the specified employee and piece of equipment. If no such time can
    // be found, null is returned.
    public java.util.Date getFirstAvailableTime(Employee employee,
                                    PieceOfEquipment equipment,
                                    java.util.Date from,
                                    java.util.Date to,
                                    long requestedLength)
                          throws RemoteException;
}
```

ApptGridReader

Dieses Bean definiert die Methode `getAppointmentGrid`. Diese gibt zu einer Ressource und einem Tag das Terminraster der Ressource an diesem Tag zurück. der Parameter `granularity` gibt die Anzahl Minuten an, auf die die im Raster definierten Zeiten gerundet werden sollen. Die Rundung ist notwendig, da die Terminkalender-Komponente der JFCSuite (siehe Abschn. 5.2.7) als Rasterlängen nur ganzzahlige Vielfache einer Grundlänge unterstützt[15].

ApptGridReader.java

```
package com.ge.med.euro.medora.scheduler.bl.apptgridreader;

import java.rmi.*;
import javax.ejb.*;
import com.ge.med.euro.medora.util.Date;
import com.ge.med.euro.medora.scheduler.bl.Resource;
import com.ge.med.euro.medora.scheduler.bl.AppointmentGrid;

/**
 * ApptGridReader is a session bean that reads
 * appointment grids for resources from the database
 */
public interface ApptGridReader extends EJBObject
{
    // Returns the appointment grid for a resource on a day, or null if none is
    // defined. The grid lines will start and end at multiples of <granularity>.
    // The unit of measurement for the granularity is minutes.
    public AppointmentGrid getAppointmentGrid(Resource resource,
                                              Date date,
                                              int granularity)
                           throws RemoteException;
}
```

UtilizationInfoReader

In der Monatsübersicht wird der Auslastungsgrad (oder auch Belegung) jeder Ressource pro Tag angezeigt. Die Auslastungsgrade der Ressourcen werden dabei in der Datenbank abgelegt und durch die Klientenrechner ausgelesen. Das hat zwei Vorteile: Zum einen muß der Klientenrechner zum Anzeigen eines Terminbuchs nicht erst alle Termine des jeweiligen Monats auslesen und daraus die Auslastung berechnen, sondern liest lediglich die Auslastungsgrade für diesen Monat aus. Dadurch fallen nicht nur wesentlich weniger Daten an (max. 31 Zahlenwerte gegenüber den Termindatensätzen eines ganzen Monats), sondern es wird auch der Berechnungsschritt überflüssig.
Der andere Vorteil ist, daß Änderungen der Auslastung (durch Anlegen, Entfernen oder Längenänderung eines Termins) mittels dem `ChangeMonitor`-Bean automatisch auf allen anderen Terminplanungsklienten aktualisiert werden können.
Das `UtilizationInfoReader`-Bean definiert zum Auslesen der Auslastung die Methode

[15] Genauer gesagt gilt dies nur für die im Rahmen des Terminplaner-Projektes erweiterte Version der Komponente; die Originalkomponente unterstützt nur Einheits-Rasterlängen

`getUtilizationInfo`, die den Auslastungsgrad einer Ressource zu jedem Tag eines gegebenen Monats an den Klienten zurückgibt.
Den erwähnten Vorteilen steht der Zusatzaufwand durch das Aktualisieren der Auslastung in der Datenbank gegenüber, der jedoch durch die Vorteile mehr als aufgewogen wird, da die Aktualisierung nur bei Auslastungsänderungen durchgeführt werden muß und von einem Container-verwalteten Entity-Bean effizient erledigt werden kann. Dieses Bean ist derzeit noch nicht implementiert, ist aber mit geringem Zeitaufwand zu erstellen, da es eine typische 1:1-Abbildung Tabelle-Bean darstellt.

UtilizationInfoReader.java

```
package com.ge.med.euro.medora.scheduler.bl.utilizationinforeader;

import java.rmi.*;
import javax.ejb.*;
import com.ge.med.euro.medora.util.Date;
import com.ge.med.euro.medora.scheduler.bl.Resource;
import com.ge.med.euro.medora.scheduler.bl.MonthUtilization;

public interface UtilizationInfoReader extends EJBObject
{
    // Returns a MonthUtilization object for each
    // day of the month that pertains to <date>.
    public MonthUtilization getUtilizationInfo(Resource resource, Date date)
                            throws RemoteException;
}
```

SelectorModel

Ein Großteil der Daten, für die es in Medora Eingabefelder gibt, hat die Eigenschaft, daß die Menge der möglichen Eingaben durch eine Datenbanktabelle festgelegt ist. Dazu zählen Patienten, Mitarbeiter, Geräte und Maßnahmen, aber auch Dringlichkeitsstufen von Terminen und Zustände von Patienten bei der Einlieferung. Für alle diese Typen von Daten gibt es jeweils eine entsprechende Tabelle, in der die gültigen Datensätze festgelegt sind, z. B. bei Mitarbeitern die Tabelle aller Mitarbeiter.
Zu einem solchen Datensatz gehört ein *Anzeigefeld* und ein *Primärschlüssel* sowie eine Anzahl von weiteren Feldern. Das Anzeigefeld ist dabei dasjenige Feld, das den Datensatz für den Benutzer am intuitivsten repräsentiert. Die anderen Felder werden in Form eines Auswahldialoges angezeigt, wenn der Benutzer eine Auswahl treffen soll. Ansonsten ist nur das Anzeigefeld sichtbar. Beispielsweise ist das Anzeigefeld bei Mitarbeiterdaten der Name des Mitarbeiters, und der Primärschlüssel ist die Personalnummer. Die restlichen Felder helfen dem Benutzer, mit Hilfe des Auswahldialoges einen bestimmten Mitarbeiter zu finden, indem sie sich durch Suchkriterien einschränken lassen. Ein Auswahldialog für Mitarbeiter mit dem zugehörigen Anzeigefeld ist in Abb. 5.10 dargestellt. Auswahlfelder sind also durch eine Datenbanktabelle und eine Anzahl Spalten der Tabelle festgelegt. Es genügt daher in Medora eine einzige graphische Komponente, um die vielen verschiedenen Auswahlfelder zu realisieren. In der Java-Implementierung des Terminplaners heißt diese Komponente `Selector`; diese kann auch für potentielle zukünftige Java-Portierungen anderer Teile von Medora verwendet werden.

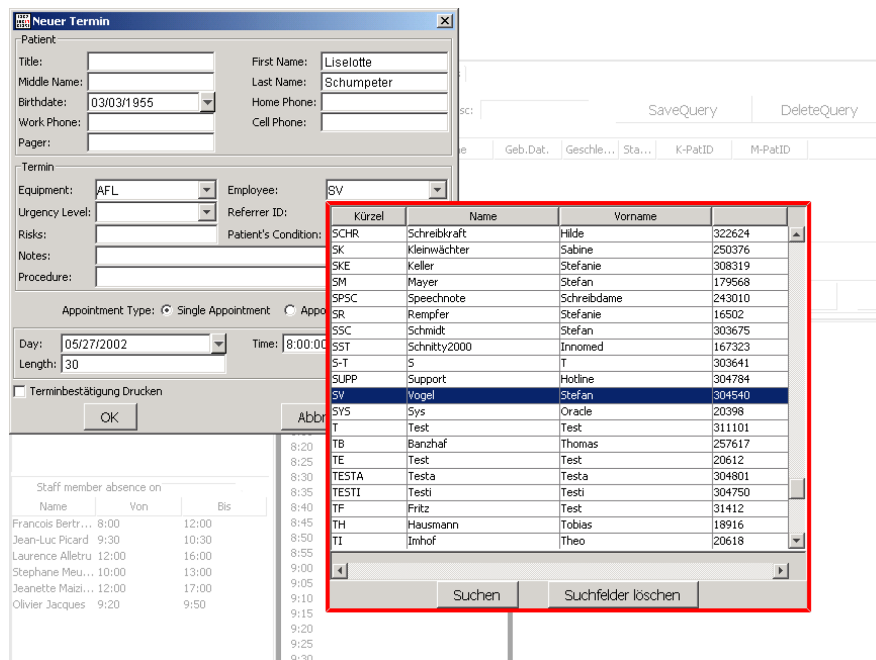

Abbildung 5.10: Mitarbeiter-Auswahldialog und Anzeigefeld (direkt über dem Auswahldialog). Das Anzeigefeld entspricht der Spalte „Kürzel“, und enthält daher den Wert „SV“.

Das Session-Bean `SelectorModel` liefert die Daten, die eine solche `Selector`-Komponente anzeigt. Dazu dient die Methode `getNextResults(int maxRows)`, die in Portionen von `maxRows` Datensätzen diese zurückliefert. Die Datensätze sind dabei durch die Suchkriterien eingeschränkt, die mittels `setSearchTerms` angegeben werden; wenn `setSearchTerms` nicht aufgerufen wird, werden alle Datensätze zurückgeliefert.
Die Methode `getRepresentationColumn` liefert das Anzeigefeld zurück, `getPrimaryKeyColumn` den Primärschlüssel.
Außer diesen Methoden definiert das Bean noch die Methoden `getRepresentationValues`, `getSearchTerms`, `goToFirstResult` und `dispose`, auf die jedoch nicht näher eingegangen werden soll.

SelectorModel.java

```
package com.ge.med.euro.medora.bl.selectormodel;

import java.util.HashMap;
import java.util.List;
import java.rmi.*;
import javax.ejb.*;
import com.ge.med.euro.medora.bl.SelectorColumnDescriptor;

public interface SelectorModel extends EJBObject
{
    public SelectorColumnDescriptor[] getColumnDescriptors()
                                      throws RemoteException;

    // Returns the index of the column whose values represent the
```

```
    // value of the row. The value from the representation column
    // is what the user sees in the text field when they make a
    // selection from the selector combo box. If there is no
    // representation column defined, -1 is returned.
    public int getRepresentationColumn() throws RemoteException;

    // Returns the index of the primary key column.
    // If there is no primary key column defined, -1 is returned.
    public int getPrimaryKeyColumn() throws RemoteException;

    // Returns the representation values of the parameter <keys> in
    // this <SelectorModel>. This method can be used to retrieve
    // display values for a number of selector value id's before
    // the search is performed. This is necessary when one or more
    // selector values need to be displayed of which only the ID is
    // known. <keys> is a list of keys for SelectorValues and may
    // contain null entries.
    // The return value is a HashMap that maps the keys to
    // representation values
    public HashMap getRepresentationValues(List keys) throws RemoteException;

    // Sets the search terms for this SelectorModel. The search
    // terms will be applied to subsequent calls to getNextResults.
    public void setSearchTerms(SelectorSearchTerms searchTerms)
                throws RemoteException;

    public SelectorSearchTerms getSearchTerms() throws RemoteException;

    public SelectorSearchResults getNextResults(int maxRows)
                                 throws RemoteException;
    // Resets the row cursor to the first row. After this method
    // is called, getNextResults will return the first set of
    // search results.
    // Since as of this writing ResultSets in Oracle do not support
    // moving backwards through the results, ResultSet.first()
    // can't be used, so the bean implementation of this method
    // just re-executes the SQL query.
    public void goToFirstResult() throws RemoteException;

    // Disposes of all resources held by the bean. After calling
    // this method, the bean should no longer be used.
    public void dispose() throws RemoteException;
}
```

PatientSelectorModel

Die Felder der Datensätze und Suchkriterien, die von `SelectorModel` verwendet werden, sind „anonym“, da sie nur über Indizes angesprochen werden. Bei den oben erwähnten Auswahlfel-

dern ist dies auch völlig ausreichend, da außerhalb des Auswahldialoges die einzelnen Felder keine semantische Bedeutung tragen und daher nicht unterschieden werden müssen. Patientendaten bilden jedoch einen Sonderfall, denn für einen Patienten gibt es im Termindialog nicht nur ein einzelnes Anzeigefeld, sondern es wird eine ganze Reihe von Feldern wie Name, Geburtsdatum usw. dargestellt (Abb. 5.5), damit der Benutzer die wichtigsten Informationen über einen Patienten beim Anlegen oder Verändern eines Termines stets im Blick hat.
Aus diesem Grunde gibt es als Datenmodell zur Patientenauswahl eigens eine abgeleitete Klasse von `SelectorModel`, und zwar das `PatientSelectorModel`-Bean. Dieses erweitert `SelectorModel` um die Methoden `setSearchTerms` und `getAttributesForColumns`, welche den durchnumerierten Feldern eines Patienten-Datensatzes Patientenattribute zuordnen.

PatientSelectorModel.java

```
package com.ge.med.euro.medora.bl.patientselectormodel;

import java.util.List;
import java.rmi.*;
import javax.ejb.*;
import com.ge.med.euro.medora.bl.PatientData;
import com.ge.med.euro.medora.bl.PatientAttributeType;
import com.ge.med.euro.medora.bl.selectormodel.SelectorModel;

/**
 * This session bean is an extension of the SelectorModel bean. It
 * has methods that map patient attributes to the database columns
 * of the patient selector model. This allows for putting
 * attributes from the Edit Appointment dialog as selector search
 * terms, which in the generic SelectorModel only have indices but
 * no semantic meaning.
 */
public interface PatientSelectorModel extends SelectorModel
{
    // Uses the attributes in <data> as <SelectorSearchTerms>.
    public void setSearchTerms(PatientData data) throws RemoteException;

    // Returns an array that contains the attributes by column
    // index. Some of the array elements may be null since not all
    // columns have to map to an appointment attribute.
    public PatientAttributeType[] getAttributesForColumns()
                                  throws RemoteException;
}
```

5.5.2 Entity-Beans

Appointment

Dieses Bean modelliert die Termine, die von der Terminplanung verwaltet werden. Es hat zwei Methoden, `setDataAppointmentData data)` und `getData(AppointmentData data)`. Die Klasse `AppointmentData` enthält alle Daten, die zu einem Termin gehören.

Appointment.java

```
package com.ge.med.euro.medora.scheduler.bl.appointment;

import java.rmi.*;
import javax.ejb.*;
import java.util.Date;
import com.ge.med.euro.medora.scheduler.bl.AppointmentData;

/**
 * Appointment entity bean remote interface
 */
public interface Appointment extends EJBObject
{
    public void setData(AppointmentData data) throws RemoteException;

    public AppointmentData getData() throws RemoteException;
}
```

ProfileElement

Dieses Bean repräsentiert ein einzelnes Element eines Terminprofils (Siehe Abschnitt 3.4) und ist zur Zeit noch nicht implementiert.

ProfileElement.java

```
package com.ge.med.euro.medora.scheduler.bl.profileelement;

import java.rmi.*;
import javax.ejb.*;
import com.ge.med.euro.medora.bl.Procedure;

public interface ProfileElement extends EJBObject
{
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;

    public long getProfileID() throws RemoteException;
    public void setProfileID(long id) throws RemoteException;

    public int getIndex() throws RemoteException;
    public void setIndex(int index) throws RemoteException;

    public Procedure getProcedure() throws RemoteException;
    public void setProcedure(Procedure procedure)
            throws RemoteException;

    public int getTimePref() throws RemoteException;
    public void setTimePref(int timePref) throws RemoteException;

    public int getTimeMin() throws RemoteException;
```

```
    public void setTimeMin(int timeMin) throws RemoteException;

    public int getTimeMax() throws RemoteException;
    public void setTimeMax(int timeMax) throws RemoteException;
}
```

5.6 Stand des Projektes

Ursprünglich war für den Rahmen dieser Diplomarbeit lediglich eine Implementierung eines Prototyps der Benutzeroberfläche vorgesehen. Wie in diesem Kapitel deutlich geworden ist, geht der tatsächliche Fertigstellungsgrad des Terminplanungssystems weit über das gesteckte Ziel hinaus, da ein Großteil der Vorgangsbearbeitung ebenfalls implementiert wurde. Dennoch bleibt noch etliche Arbeit zu tun, bevor das System reif für den praktischen Einsatz ist. Bezüglich der Funktionalität sind das folgende Punkte:

- Implementierung der Profilterminfunktion
- Implementierung der Funktion „Weitere Termine / Abwesenheitsliste"
- Vollständige Implementierung der Verwaltung der Auslastungsdaten
- Prüfung auf Kollisionen bei manueller Terminvergabe
- Implementierung einer Suche nach freien Einzelterminen
- Datenbanksperren für Termine während deren Bearbeitung

Auch enthält der Terminplaner wie die meisten im Entwicklungsstadium begriffenen Softwareprodukte etliche meist kleinere Fehler, die beseitigt werden müssen. An schwerwiegenden Problemen mit der Bedienbarkeit ist derzeit nur eines bekannt. Dabei handelt es sich um eine Zeitverzögerung, die aus bisher ungeklärter Ursache bei Verschieben von Terminen auftritt. Es liegt allerdings die Vermutung nahe, daß dieses Problem durch eine der Komponenten der JFCSuite verursacht wird.
Schließlich müssen mit dem Terminplanungssystem vor der Auslieferung noch ausgiebige Lasttests und Tests auf Tauglichkeit im Praxisbetrieb durchgeführt werden.

Kapitel 6

Fazit und Ausblick

In der hier vorgestellten Technologiestudie wurde untersucht, ob Java zusammen mit J2EE eine geeignete Plattform für radiologische Informationssysteme ist. Es zeigte sich dabei, daß die Unzulänglichkeiten der alten Terminplanung behoben werden konnten. Die Befürchtung, daß die häufig mit Java in Zusammenhang gebrachte Langsamkeit bei der Programmausführung sich negativ auf die Bedienbarkeit des Terminplaners auswirken könnte, erwies sich als unbegründet; es wurde sogar eine Geschwindigkeitssteigerung gegenüber der alten Version erreicht.
Bei der Entwicklung des Terminplaners wurde darauf geachtet, daß die Schnittstellendefinitionen der Java-Klassen und Enterprise-Beans möglichst allgemein gehalten sind, damit diese auch auch von zukünftigen Medora-Modulen verwendet werden können. Mit der Terminplanung wurde also ein Grundstein für kommende Medora-Versionen gelegt. In Zukunft werden der Terminplanung weitere Java-Module folgen; als nächstes ist die Portierung der sog. Befundmappe geplant.
Der Terminplaner wurde zwar vollständig unter Windows entwickelt; da aber keine Windows-spezifischen Funktionen verwendet wurden, dürfte es ohne großen Aufwand möglich sein, den Terminplaner auch unter anderen Betriebssystemumgebungen wie Linux oder MacOS zum Einsatz zu bringen.
Auf Seiten der theoretischen Grundlagen wurde in Form von zwei Algorithmen der Boden für die automatische Vergabe von Mehrfachterminen bereitet und einer der zwei Algorithmen bereits implementiert. Es wurde nicht untersucht, ob und mit welchem Aufwand sich die Algorithmen auf Prozessorengruppen erweitern lassen; dies ist als Gegenstand zukünftiger Arbeiten geeignet.

Stichwortverzeichnis

α, 10
β, 8
γ, 12
2-Schicht-Modell, 25, 26
3-Schicht-Modell, 25, 26

Abhängigkeitsbeziehungen, 9
Ablaufplanung, 7
Abstand
  zwischen Operationen, 16, 17, 19, 20, 22, 23
Abwesenheitsliste, 31, 33
Aktivitätsnetzwerk
  einfach verbundenes, 9
Ankunftszeit, 13
Anwendungsarchitekturen, 25
Applikationsmodell, *siehe* Anwendungsmodell
Applikationsserver, 26–30
Arzt
  externer, 15
Aufgabe, 7
Auftrag, 7, 15, 16
Auftragsmodell, 7
Auswärtsbaum, 9

Backtracking, 5
Baum, 9
Bearbeitungszeit, 7, 9, 16, 17
Belegungsplan, 19, 20
Benutzeroberfläche
  Neue Terminplanung, 31, 34
Benutzerschicht, 25, 26, 29
  neue Terminplanung, 34
Bereitstellungszeit, 9, 16
Bestrahlungsserien, 14
Bytecode, 27

Clustering, 26
Computertomographie, 15
Connection Pooling, 26

Datenbank, 25, 26, 28, 30
Datenbankschicht, 25, 26
Datenbankserver, 25, 26
Datenbankverbindung, 25, 26

Einprozessorsystem, 10
Einwärtsbaum, 9
Einzeloperation, 20
Einzeltermin, 14–16, 23
EJB, 28, 29
  2.0, 29
  in der Praxis, 29
  Standard, 28
Enddatum, 23
Engpaßverschiebung, 13
Enterprise-Bean, 28, 29, 36
  Entity-, 28
  nachrichtengesteuert, 28, 29
  Session-, 28, 29

Fertigstellung
  Zeitpunkt der, 12, 16, 22
Flow-Shop, 11, 14, 15
Forms, 1, 27
Funktionalität
  bisherige, 3
Funktionsbereich, 31, 32

Gantt, Henry, 7
Gantt-Diagramm, 7
Geräte
  medizinische, 3, 4, 15, 33, 38, 41
Gerätegruppen, 4, 33
Graph
  gerichteter, 9

Hauptfenster, 31
HL7, 1, 15
Home-Interface, 29

Implementierung, 25
Instanzvariable, 29

Interpreter, 27

J2EE, 2, 28
Java, 2, 27, 34
 Ausführungszeit, 27
 Compiler, 27
 Virtuelle Maschine, 27
JDBC, 28, 29
JDK, 27
JFCSuite, 31
JMS, 29
Job-Shop, 11, 13–17

KIS, 1
Klient, 29
Klientenrechner, 25–28
Knoten, 9
Kollision, 20
Komplexität, 14
 Profiltermin-Algorithmus, 22
 Serientermin-Algorithmus, 23
Komplexitätsklassen, 14
Kontrastmittel, 14
 Injektion, 14
Kostenfunktion, 11, 16, 17
Krankenhaus, 28

Lokale Genetische Suche, 13

Maßnahme
 medizinisch, 15, 19
Maschinenbefehl, 27
Medora, 1, 16, 25, 30, 31, 36, 38, 43, 49
 Terminplanung, 3, 4, 31
Mehrfachtermin, 4
Mehrprozessormodell, 15, 19
Mitarbeitergruppen, 4, 33
Monatsübersicht, 31, 32
Multifunktionsleiste, 31, 32

Netzwerkverbindung, 26
NP-hart, 13, 17

Open-Shop, 11, 12, 14, 15
Operation
 Datenbank-, 26
 planungstheoretische, 7, 15–17, 19, 20, 23
Operations Research, 7
Optimierungskriterium, 7, 8, 11, 12
Optimierungsproblem, 11

Parallele Prozessoren, 10
Persistenz
 Bean-verwaltet, 28
 Container-verwaltet, 28
Personal
 medizinisches, 15
Pflichtenheft, 3
Planungsproblem, 15–17, 19, 20
Planungsprobleme
 Beispiele, 12
 Erweiterung von, 13
 Lösung von, 13
Planungstheorie, 7
präemptiv, 8
Präparat
 radioaktiv, 14
Profilelement, 15, 19
Profiltermin, 4, 14–17
 Algorithmus, 19, 20
 Modell, 16
Programmiersprache, 27, 30
Prototyp, 25, 31, 48
Prozessor, 7, 15, 16, 19, 23
 mehrere pro Operation, 13
Prozessorengruppen, 15
Prozessormodell, 7
Prozessorzahl, 14
Pufferoperation, 17

Rückwärtsschritt, 20
Radiologie, 14, 15
Radiologie-Informationssystem, *siehe* RIS
Rekursionsformel, 20
Remote-Interface, 29
Ressource
 Terminplanungs-, 33
Ressourcenraster, 33
RIS, 1, 49

Serientermin, 4, 14, 16, 23
 Algorithmus, 23
 Modell, 16
Simulated Annealing, 13
Skalierbarkeit, 26
Spezialisierte Prozessoren, 10, 11
SQL, 28, 30, 31
Startzeitpunkt, 20, 23

Steuerungsleiste, 31, 32
Stichtermin, 9, 12, 16
Strahlenmedizin, 14
Strahlungsintensität, 14

Teilbarkeit
    von Operationen, 8
Terminübersicht, 31, 33
Terminanforderung, 15
Terminbuch, 3, 32, 33
Termindialog, 33
Terminplanung
    in der Radiologie, 15
    Medora, 16, 25
    neue, 31
Terminprofil, 14, 19
Terminserie, 15, 16, 23
Toleranz, 23
Totzeit, 15

Unternehmensforschung, 7

Vorgangsbearbeitung, 26, 28, 29
    neue Terminplanung, 36
Vorwärtsschritt, 20

Wartezeit
    mittlere, 16, 17
Weitere Termine, 31, 33

Zeit
    lineare, 23
    polynomielle, 13
Zeitfenster, 16
Zeitintervall, 19, 20

Literaturverzeichnis

[Baker 74] Kenneth R. Baker, Introduction to Sequencing and Scheduling. John Wiley & Sons, 1974

[Blazewicz 96] Jacek Blazewicz, K.H. Ecker, E. Pesch, G. Schmidt, J. Weglarz: Scheduling Computer and Manufacturing Processes. Springer, 1996

[Bruckenberger 00] Ernst Bruckenberger, Radiologie im Wandel. <http://www.bruckenberger.de/doc/telemedizin/riw2000.htm>, Stand 12.4.2002

[Brucker 95] Peter Brucker: Scheduling Algorithms. Springer, 1995

[Gonzales et al. 76] T. Gonzales, S. Sahni: Open Shop Scheduling to Minimize Finish Time. Journal of the Association for Computing Machinery 23, Seiten 665–679, 1976

[Graham 79] R.E. Graham, E.I. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan: Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, Ann. Discrete Math. 4, 287-326, 1979

[Hieber 96] Felicitas Hieber: Analyse und Spezifikation einer EDV-Lösung für die Krankenhausprozesse Untersuchungsdokumentation, Termin- und Bestrahlungsplanung mit Schwerpunkt Strahlentherapie (Diplomarbeit). Universität Heidelberg, 1996

[Horn 74] W.A. Horn: Some Simple Scheduling Algorithms. Naval Research and Logistics Quarterly 21, Seiten 177–185, 1974

[Jain 98] A. S. Jain, S. Meeran: Deterministic Job-Shop Scheduling: Past, Present and Future. University of Dundee, 1998

[Kistner 90] Klaus-Peter Kistner, Marion Steven: Produktionsplanung. Physica-Verlag Heidelberg, 1990

[Lawler 81] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan: Minimizing maximum lateness in a two-machine open shop, Mathematics of Operations Research 6, Seiten 153–158, 1981; Erratum, Math. Oper. Res 7, 635.

[Lenstra 79] J.K. Lenstra, A.H.G. Rinnooy Kan: Computational Complexity of Discrete Optimization Problems. Annals of Discrete Mathematics 4, Seiten 121–140, 1979

[Monson 01] Richard Monson-Haefel: Enterprise Java-Beans. Deutsche Aushabe der 3. Auflage, O'Reilly, 2001

[NETg 00] NETg: Enterprise JavaBeans. CD-ROM. Reihe „Skill Builder“, National Education Training Group, 2000

[Pinedo 95] Michael Pinedo: Scheduling. Prentice Hall, 1995

[Sotskov 95] Y.N. Sotskov, N.V. Shakhlevich: NP-hardness of shop scheduling problems with three jobs. Discrete Applied Mathematics 59/3, Seiten 237–266, 1995.

[Sun 02] Sun Microsystems: Enterprise JavaBeans 2.0 Specification. <http://java.sun.com/products/ejb/docs.html>, Stand 17.5.2002

[Univ. Osnabrück 02] Universität Osnabrück, Complexity results for scheduling problems. <http://www.mathematik.uni-osnabrueck.de/research/OR/class/>, Stand 25.4.2002

[Wessel 98] Ivo Wessel: GUI-Design. Hanser, 1998

Ich versichere, daß ich die Arbeit ohne fremde Hilfe angefertigt und noch keinem anderen Prüfungsgremium vorgelegt habe. Wörtlich oder indirekt übernommenes Gedankengut habe ich nach bestem Wissen als solches kenntlich gemacht.

Ulm, Mai 2002