Prüfer: Prof. K. Rothermel
Betreuer: Dipl.-Inform. Markus Straßer
Beginn am: 01.05.99
Beendet am: 31.10.99
CR-Nummer: C 2.4, D 1.5

Studienarbeit Nr. 1754

Reiserouten-Konzepte für Mobile Agenten

Jürgen Buschle

Kurzfassung

Mobile Agenten erledigen ihnen gestellte Aufgaben selbständig. Dazu wandern sie in einem Netzwerk von Rechner zu Rechner und nehmen dort angebotene Dienste in Anspruch. Der „Weg“, den ein Agent zurückzulegen hat sollte sehr flexibel gestaltbar sein. Dadurch wird ein Agent nicht blockiert, wenn irgend ein Rechner nicht erreichbar ist. Er könnte dann entweder einen alternativen Rechner auswählen oder aber diesen Rechner erst später besuchen.

In dieser Studienarbeit wird ein gegebenes Konzept zur flexiblen Gestaltung der Reiseroute eines Agenten auf Erweiterungsmöglichkeiten überprüft und daraufhin implementiert. Weiterhin wird das implementierte Konzept in das Mobile-Agent-System Mole integriert.

Inhaltsverzeichnis

1. Mobile-Agent-System 1
   1.1 Mobile Agenten 1
   1.2 Reiseroute 1
   1.3 Mobile-Agent-System 2
2. Mole 3
   2.1 Einführung 3
   2.2 Agenten 3
   2.3 Reiseroute 3
   2.4 Location 4
   2.5 Engine 4
3. Reiseroutenkonzept 5
4. Entwurf 11
   4.1 Die Unified Modeling Language 11
   4.2 Entries-Klassen 12
      4.2.1 Die Klasse Entry 12
      4.2.2 Die Klasse B_Entry 16
      4.2.3 Die Klasse I_Entry 18
      4.2.4 Die Klassen Closed/Open_I_Entry 28
      4.2.5 Die Klasse Itinerary 28
   4.3 Precondition-Klassen 30
      4.3.1 Die Klasse Precond 30
      4.3.2 Die Klasse Predicate 35
      4.3.3 Die Klasse Func 36
      4.3.4 Die Klasse Operator 37
      4.3.5 Die Klasse Equation 38
      4.3.6 Weitere Klassen 39
   4.4 Erstellen einer kompletten Reiseroute 41
5. Implementierung 45
   5.1 Prioritäten 45
   5.2 Vorbedingungen 49
      5.2.1 Erstellen und Verändern von Vorbedingungen 49
      5.2.2 Auswerten von Vorbedingungen 49
   5.3 Entries 54
      5.3.1 Ausführbare Entries holen 54
      5.3.2 Nicht ausführbare Entries ermitteln 57

6. Moleintegration 59
7. Fazit 62

Literaturverzeichnis 63

Anhang 64
   Projektplan 64

1. Mobile-Agent-System

In diesem Kapitel werden die wichtigsten Grundlagen von Mobilen Agentensystemen erläutert.

1.1 Mobile Agenten

Mobile Agenten sind aktive Objekte, die ein Verhalten und einen Zustand haben. Weiterhin besitzt ein Agent einen Ort auf dem er gerade ausgeführt wird. Mobile Agenten lösen ihnen gestellte Aufgaben in dem Sie in einem Netzwerk von Knoten zu Knoten wandern und dort bestimmte Funktionen ausführen. Das ändern des Aufenthaltsortes eines Agenten wird als Migration bezeichnet. Dazu wird der Agent auf dem Knoten auf dem er sich gerade befindet beendet. Dann wandert der Agent auf einen anderen Knoten. Dort wird dann der aktuelle Zustand wiederhergestellt und der Agent wird daraufhin auf dem Knoten, auf dem er sich jetzt befindet wieder gestartet. Ein Agent hat noch die Möglichkeit mit anderen Agenten über Nachrichten zu kommunizieren. Dabei kann sich der Agent dem eine Nachricht geschickt werden soll an einem anderen Ort im Netzwerk befinden. Dann wird über einen Remote Procedure Call der Austausch von Daten ermöglicht. Dies entspricht aber dem Vorgehen bei Client/Server Modellen und bringt somit nichts Neues. Dieses Vorgehen hat folgende Nachteile. Zum ersten müssen alle Daten über das Netzwerk transportiert werden. Weiterhin wird dem Rechner auf dem der Agent ausgeführt wird, dies ist bei diesem Vorgehen immer der gleiche, ständig Rechenleistung entzogen.
Um diese Nachteile zu umgehen hat ein mobiler Agent nun die Möglichkeit zu dem Knoten zu migrieren auf dem sich der andere Agent befindet. Somit muß nur der Agent über das Netzwerk transportiert werden. Ist der Agent dort angekommen schickt er dem anderen Agenten eine Nachricht, daß er einen Dienst für ihn erledigen soll. Das Ergebnis des Dienstes bekommt er dann wiederum als Nachricht zurück. Bei diesem Prinzip kann der Rechner auf dem der Agent gestartet wurde vorübergehend abgeschaltet werden. Der Agent wandert dann selbständig durch das Netzwerk und erledigt seine Aufgaben. Um die Ergebnisse zurück liefern zu können, muß natürlich wieder eine Verbindung zum Startrechner bestehen.

1.2 Reiseroute

Bevor ein mobiler Agent gestartet wird muß er wissen welche Rechner er besuchen und welche Aufgaben er auf dem jeweiligen Rechner erledigen muß. Diese Informationen werden im folgenden als Reiseroute bezeichnet. Man kann sich verschiedene Ansätze beim Erstellen und Erledigen einer Reiseroute vorstellen. Einmal kann man eine feste Reiseroute vorgeben von der auf keinen Fall abgewichen werden kann. Dabei entsteht das Problem, daß beim Nichterreichen eines Rechner solange gewartet werden muß, bis der Rechner wieder erreichbar ist. Deshalb kann man sich ein flexibleres Prinzip vorstellen. Dabei werden Alternativen für bestimmte Rechner angegeben. Bei der Ausführung kann der Agent dann entscheiden welchen Rechner er besucht. Wenn der eine Rechner ausgefallen ist kann er einfach den anderen besuchen.

1.3 Mobile-Agent-System

Ein Agentensystem ist eine Art Laufzeitumgebung der Agenten. Ohne ein vorhandenes Agentensystem können auf einem Rechner keine Agenten ausgeführt werden. Es stellt Orte zur Verfügung, an denen sich Agenten befinden können um dort ihre Aufgaben zu erledigen. Auch der Nachrichtenaustausch sowie die Migration werden erst durch das Agentensystem ermöglicht.

2. Mole – Ein mobiles Agentensystem

2.1 Einführung

Mole ist ein Java basiertes Agentensystem. Es wurde 1995 in einer Diplomarbeit entworfen und als Prototyp implemeniert. Das System liegt mittlerweile in der Version 3.0 vor, welche auch Grundlage für diese Studienarbeit ist.

2.2 Agenten

Agenten sind aktive Objekte die auf Locations ausgeführt werden. Dabei muß man zwischen zwei Arten von Agenten unterscheiden:

- Systemagenten:
  Systemagenten haben Zugriff auf Systemressourcen und können Dienste bereitstellen um anderen Agenten unter anderem Informationen über das System zu liefern. Diese Agenten dürfen nicht auf andere Locations migrieren. Dies ist aus Sicherheitsgründen so realisiert.
- Benutzeragenten:
  Ein Agent dieser Art hat keinen Zugriff auf Systemressourcen, hat aber dafür die Möglichkeit sich von Location zu Location zu bewegen um dort die Dienste anderer Agenten in Anspruch zu nehmen.

Es gibt zwei verschiedene Kommunikationsarten zwischen Agenten. Zum einen besteht die Möglichkeit mit einem Remote Procedure Call eine Methode eines anderen Agenten aufzurufen. Diese Kommunikation ist immer synchron, das heißt der Agent, der die Methode aufruft wird solange blockiert bis er das Ergebnis erhalten hat. Die zweite Möglichkeit zu kommunizieren besteht im Versenden von Nachrichten. Dabei wird diese an die Location geschickt auf der sich der Agent, der die Nachricht erhalten soll befindet. Falls der gesuchte Agent sich auf der Location befindet wird eine bestimmte Methode des Agenten aufgerufen, die speziell zum Empfangen von Nachrichten gedacht ist. In dieser Methode werden dann abhängig von der empfangenen Nachricht unterschiedliche Aktionen angestoßen. Diese datenorientierte Kommunikation kann sowohl synchron als auch asynchron erfolgen. Beiden Kommunikationsmechanismen ist gemein, daß die beteiligten Agenten sich entweder auf der selben oder auf verschiedenen Locations befinden können.

2.3 Reiseroute

Wenn ein Agent auf eine andere Location migrieren will muß er die Methode migrateTo der Location aufrufen auf der er sich gerade befindet. Als Parameter wird dabei der Name der Ziellocation übergeben. Durch den Code des Agenten ist die Reiseroute also fest vorgegeben. Zur Laufzeit können durch den Agenten keine Veränderungen mehr vorgenommen werden. Der Agent kann also nicht auf das Ausfallen oder Nichterreichen eines Rechners reagieren.
Dieses Konzept soll in dieser Arbeit flexibler gestaltet werden. Dadurch wird dem Agenten eine Reiseroute mitgegeben, die nicht komplett festgelegt ist. Der Agent kann selbständig zwischen gleichwertigen Einträgen aussuchen. Dabei werden zwischen den Einträgen Reihenfolgebeziehungen und Prioritäten definiert, die den Rahmen für die Ausführung bilden. Das genaue Konzept wird später dargestellt.

2.4 Location

Locations sind die Orte des Agenten-Systems. Die Location stellt die Funktionalität für die Agenten zur Verfügung und verwaltet diese. Sie ist für das Starten, Beenden und Migrieren von Agenten zuständig. Außerdem stellt sie Methoden zur Kommunikation zur Verfügung.

2.5 Engine

Die Engine ist die Laufzeitumgebung von Mole. Auf jedem Rechner, der Agenten aufnehmen möchte, muß eine Engine laufen. Beim Start der Engine wird mindestens eine Location erstellt, welche wiederum jeweils ihre Agenten startet. Eine Engine kann mehrere Locations verwalten.

3. Reiseroutenkonzept

Aufgabe dieser Arbeit ist es, ein gegebenes Konzept zur expliziten, sehr flexiblen Definition einer Reiseroute auf Erweiterungsmöglichkeiten zu evaluieren und dieses darauf hin zu implementieren. Im Folgenden wird das gegebene Konzept genauer betrachtet.

Eine Reiseroute, ab hier auch als Itinerary $I = (S, R_p)$ bezeichnet soll dargestellt werden durch eine Menge $S = \{e_1, e_2, ..., e_n\}$. S ist dabei eine Menge von Einträgen e_i in dieser Reiseroute. Dabei darf jeder Eintrag nur einmal in der Reiseroute vorkommen. $R_p \subset S \times S$ stellt eine Prioritätsrelation zwischen den Einträgen der Itinerary dar. Ein Eintrag oder auch Entry genannt ist ein Tripel aus Vorbedingung, Knoten und Methode und beschreibt welche Methode der Agent bei erfüllter Vorbedingung auf dem angegebenen Knoten ausführen soll. Für diese drei Begriffe werden auch die englischen Entsprechungen verwendet werden. Somit sieht so ein Entry wie folgt aus: (precondition, node, method). Dabei stellt die Vorbedingung einen booleschen Ausdruck der Form $p(e_1, e_2, ..., e_n)$ dar. Ein Entry (bedingung1, knoten1, methode1) wird wie folgt abgearbeitet. Zuerst wird die Vorbedingung (bedingung1) ausgewertet. Ist diese erfüllt so kann der Entry ausgeführt werden. Dies bedeutet, daß der Agent auf den Knoten knoten1 migriert und dort die Methode mehtode1 ausführt. Wenn ein Entry zu jeder Zeit ausgeführt werden kann, so ist seine Precondition folglich auf TRUE gesetzt. Precond(e) ist dabei die Funktion, die die Precondition des Entries e auswertet. Um Reihenfolgebeziehungen zwischen den Entries zu definieren kann das Prädikat D(e) in Vorbedingungen verwendet werden. D(e) wird genau dann wahr wenn das Entry e bereits ausgeführt wurde. Prioritäten zwischen Entries werden mit der Prioritätsrelation R_p definiert. Dabei hat der Entry e_i eine höhere Priorität als e_j, wenn $(e_i, e_j) \in R_p$ ist. Wenn jetzt die Preconditions, die von e_i und die von e_j beide wahr sind wird zuerst e_i ausgeführt werden. Wenn nur die Vorbedingung von einem der beiden Entries e_i oder e_j wahr ist, wird die Priorität nicht beachtet. In diesem Fall wird der mit der wahren Vorbedingung ausgeführt. R_p darf keine Zyklen enthalten wie dies im folgenden Beispiel der Fall wäre: $(e_i, e_j), (e_j, e_k), ..., (e_l, e_i)$

Das bisher Erläuterte soll an zwei einfachen Beispielen gezeigt werden:

Beispiel 1: Itinerary $I = (\{e_1, e_2\}, \{\})$ mit zwei Entries e_1 und e_2, wobei e_1 vor e_2 ausgeführt werden soll.

$$e_1 = (TRUE, n_1, m_1)$$
$$e_2 = (D(e_1), n_2, m_2)$$

Beispiel 2: Itinerary $I = (\{e_1, e_2\}, R_p)$ mit zwei Entries e_1 und e_2 , wobei entweder e_1 oder e_2 ausgeführt werden sollen. Dabei ist e_1 der bevorzugte Entry.

$$e_1 = (\neg D(e_2), n_1, m_1)$$
$$e_2 = (\neg D(e_1), n_2, m_2)$$
$$R_p = \{(e_1, e_2)\}$$

Um sehr einfach ausdrücken zu können, daß genau i Entries aus einer Menge von j Entries $e_1,...,e_j$ $(i \leq j)$ ausgeführt werden sollen, werden Ausdrücke in Form von (number > expression) in die Preconditions aufgenommen. Expression ist dabei eine Summe $d(e_1) + d(e_2) + ... + d(e_j)$. Die Funktion d(e) ist wie folgt definiert: d(e) = 1, wenn D(e) = TRUE und d(e) = 0, wenn D(e) = FALSE.

Beispiel 3: Führe genau i von j Entries $e_1, e_2, ..., e_j$ $(i \leq j)$ aus.

$$e_k = ((i > d(e_1) + d(e_2) + ... + d(e_j)), n_k, m_k) \text{ für } k = 1...j$$

Um mehrere Entries gruppieren und somit zu einer logischen Einheit zusammenfassen zu können, wird das Prinzip der geschachtelten Reiserouten eingeführt. Dieses erlaubt neben den Tripeln (precondition, node, method) ganze Reiserouten als Eintrag in einer Reiseroute. Um diese verschiedenen Einträge namentlich zu unterscheiden werden die (precondition, node, method) Tripel als B-Entries und die Einträge, die selbst wieder eine Reiseroute darstellen als I-Entries bezeichnet. B-Entry und I-Entry sind Abkürzungen für base entry und itinerary entry. Ein I-Entry e sieht wie folgt aus: $e = (\text{precondition}, \{e_1, e_2, ..., e_n\}, R_p, \text{type})$. Dabei sind $e_1, e_2, ..., e_n$ die Entries der I-Entry e. Die Menge der Entries $\{e_1, e_2, ..., e_n\}$ bezeichnet man als Entries(e). Die Precondition von e muß erfüllt sein um das Ausführen von Entries aus der I-Entry e starten zu können. Um also ein B-Entry das sich direkt oder indirekt in Entries(e) befindet, ausführen zu dürfen, muß sich die Vorbedingung von e zu TRUE auswerten. Wie man leicht sieht ist eine Itinerary $I = (\{e_1, e_2, ..., e_n\}, R_p)$ ein Spezialfall einer I-Entry $e = (\text{precondition}, \{e_1, e_2, ..., e_n\}, R_p, \text{type})$. Dabei ist die Precondition TRUE und der type der Itinerary ist beliebig wählbar. Der Typ bzw. type einer I-Entry wird erst später erklärt. Die Preconditions der Entries einer I-Entry e dürfen sich nur auf Entries beziehen, die ebenfalls direkt in e enthalten sind ($\text{Precond}(e_i) = p(e_1, e_2, ..., e_n) \ \forall \ e_i \in \text{Entries}(e)$). Eine Precondition setzt sich aus Beziehungen zu Entries zusammen. Durch diese Beziehungen ist die Reihenfolge der Ausführung festgelegt. Solche Beziehungen können nur zwischen Entries, die sich direkt im gleichen I-Entry befinden definiert werden.
R_p definiert die Prioritäten zwischen den Entries $e_1, e_2, ..., e_n$. Zunächst wird an einem Beispiel die neue Struktur gezeigt und erklärt was mit direkt und indirekt enthalten gemeint ist.

Beispiel 4: Itinerary $I = (\text{TRUE}, \{e_1, e_2, ..., e_m, e_{n+1}\}, \{\}, \text{type})$, wobei die Entries $e_{m+1},...,e_n$, die im I-Entry e_{n+1} liegen, nach den Entries $e_1,...,e_m$ ausgeführt werden sollen.

$$e_i = (\text{TRUE}, n_i, m_i) \text{ für } i = 1...m$$

$$e_{n+1} = (D(e_1) \wedge D(e_2) \wedge ... \wedge D(e_m), \{e_{m+1}, e_{m+2}, ..., e_n\}, \{\}, \text{type})$$

Dabei sind die Entries $e_1, e_2, ..., e_m, e_{n+1}$ direkt im Itinerary Objekt I, die Entries $e_{m+1}, e_{m+2}, ..., e_n$ dagegen nur indirekt enthalten.

Beispiel 5: Dieses Beispiel soll die Struktur einer Itinerary mit B-Entries und I-Entries verdeutlichen.

$$I = (TRUE, \{e_1, e_2, e_3, e_{11}, e_{13}\}, \{\}, type)$$
$$e_{11} = (TRUE, \{e_4, e_5, e_{12}\}, \{\}, type)$$
$$e_{12} = (TRUE, \{e_6, e_7\}, \{\}, type)$$
$$e_{13} = (TRUE, \{e_8, e_9, e_{10}\}, \{\}, type)$$

In Abbildung 3.1 ist diese Struktur bildlich dargestellt.

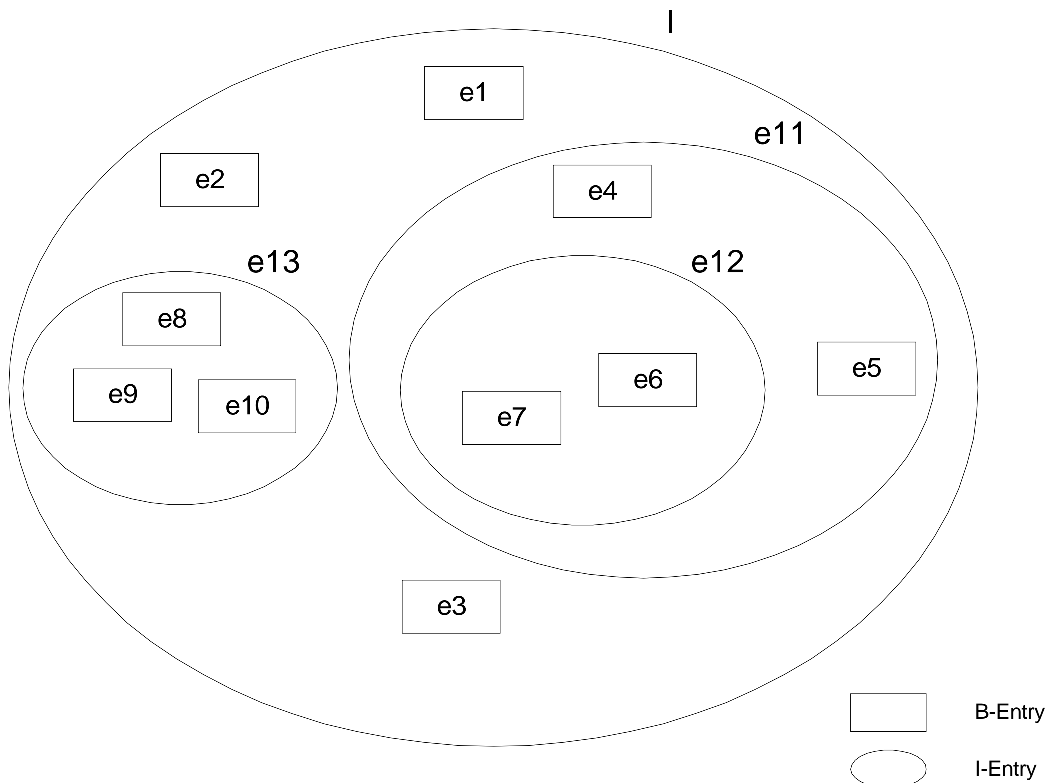


Abbildung 3.1: Struktur der Reiseroute

Es gibt zwei verschiedene Typen von I-Entries:

- Open-I-Entries $e = (precondition, \{e_1, e_2, ..., e_n\}, R_p, open)$

  Die Ausführung einer Open-I-Entry kann sich mit der Ausführung von B-Entries, die nicht direkt oder indirekt in der I-Entry enthalten sind überschneiden. Daher kann es vorkommen, daß die Vorbedingung der Open-I-Entry FALSE wird, nämlich durch Ausführen von nicht enthaltenen B-Entries. Also wird festgelegt, daß die Precondition einer Open-I-Entry nur beim Ausführen der ersten B-Entry daraus erfüllt sein muß. Die anderen B-Entries werden auch ausgeführt wenn die Vorbedingung des Open-I_Entries dann nicht mehr erfüllt ist.

- Closed-I-Entries $e = (precondition, \{e_1, e_2, ..., e_n\}, R_p, closed)$

  Die Ausführung einer Closed-I-Entry darf sich nicht mit der Ausführung von B-Entries, die nicht direkt oder indirekt in der I-Entry enthalten sind überschneiden. Sobald also ein B-Entry einer Closed-I-Entry e ausgeführt wurde, dürfen solange

nur noch B-Entries aus e gestartet werden bis die Preconditions der noch nicht ausgeführten B-Entries von e alle FALSE werden. Dies beschreibt das Prädikat D(e) dessen genaue Definition weiter unten folgt.
Closed-I-Entries können z.B. benutzt werden, wenn mehrere Aktionen in einer Transaktion ausgeführt werden sollen. Dadurch können die benutzen Ressourcen schneller wieder frei gegeben werden, da die Ausführung des Closed-I-Entry nicht durch außerhalb liegende B-Entries unterbrochen wird.

Um Vorbedingungen zwischen I-Entries besser in den Griff zu bekommen wird noch das Prädikat S(e) benutzt. S(e) liefert TRUE, wenn bereits Entries der I-Entry e ausgeführt worden sind. Dadurch läßt sich das sequentielle Ausführen zweier Open-I-Entries realisieren. Beide Open-I-Entries starten nur dann wenn der andere bereits vollständig abgearbeitet ist, also D(e) gilt, oder wenn der andere noch nicht gestartet wurde und somit S(e) wahr ist. S(e) ist wie folgt rekursiv definiert:

$$\begin{aligned}
&S(e) = TRUE \Leftrightarrow \\
&\quad ((e \text{ ist } B-Entry) \wedge e \text{ wurde schon ausgeführt}) \vee \\
&\quad ((e \text{ ist } I-Entry) \wedge (\exists\, e_i \in Entries(e) : S(e_i)))
\end{aligned}$$

Analog zu d(e) wird hier s(e) definiert durch s(e) = 1 wenn S(e) = TRUE und s(e) = 0 wenn S(e) = FALSE.
Mit diesem neuen Prädikat S(e) läßt sich nun auch die rekursive Definition für D(e) wiedergeben:

$$\begin{aligned}
&D(e) = TRUE \Leftrightarrow \\
&\quad ((e \text{ ist } B-Entry) \wedge e \text{ wurde schon ausgeführt}) \vee \\
&\quad ((e \text{ ist } I-Entry) \wedge (\forall\, e_i \in Entries(e) : (D(e_i) \vee (\neg S(e_i) \wedge (Precond(e_i) = FALSE)))))
\end{aligned}$$

Das folgende Beispiel zeigt die Benutzung der oben definierten Prädikate.

Beispiel 6: Zwei Mengen von B-Entries $e_1,...e_m$ und $e_{m+1},...e_n$ sollen in zufälliger Reihenfolge ausgeführt werden. Dabei dürfen sich die Ausführungen der beiden Mengen nicht überschneiden. Es gibt noch weitere B-Entries $e_{n+3},...e_{n+k}$ deren Ausführung sich mit der, der Entries $e_1,...e_n$ überschneiden darf.

$$\begin{aligned}
&e_i = (TRUE, n_i, m_i) \text{ für } i = 1...n \\
&e_{n+1} = (D(e_{n+2}) \vee \neg S(e_{n+2}), \{e_1,...,e_m\}, \{\}, open) \\
&e_{n+2} = (D(e_{n+1}) \vee \neg S(e_{n+1}), \{e_{m+1},...,e_n\}, \{\}, open) \\
&I = (TRUE, \{e_{n+1}, e_{n+2}, e_{n+3},...,e_{n+k}\}, \{\}, open)
\end{aligned}$$

Somit wurden alle Elemente, die in Vorbedingungen benutzt werden können beschrieben und wir können die komplette Syntax einer Precondition wiedergeben.

Dies wird im Folgenden mit Hilfe der Backus-Naur-Form gemacht:

Precond=	("(" Precond ")" | "¬" Precond | Precond "∨" Precond | Precond "∧" Precond | Lit).
Lit=	(Predicate | Equation | "TRUE" | "FALSE").
Predicate =	("D(" Entry ")" | "S(" Entry ")").
Equation=	"(" Number Op Expr ")".
Op=	("<" | "≤" | "=" | "≥" | ">").
Expr=	(Func "+" Func | Func).
Func=	("d(" Entry ")" | "s(" Entry ")").

Ausführbares B_Entry:

Nach dem das Konzept vorgestellt wurde wird hier noch einmal klar gemacht, welche Kriterien erfüllt sein müssen, damit ein B-Entry der Reiseroute ausführbar ist.
Ein B_Entry e_1 ist ausführbar wenn die folgenden Kriterien 1-4 erfüllt sind:

1. Open-I-Entries, in denen sich e_1 befindet müssen entweder schon im Zustand STARTED sein oder ihre Precondition muß erfüllt sein.

Die Precondition eines Open-I-Entry muß nur beim Ausführen des ersten B-Entries erfüllt sein. Danach können die anderen B-Entries daraus auch bei nicht erfüllter Precondition des Open-I-Entry ausgeführt werden. Aus dieser Festlegung folgt direkt Kriterium 1.

2. Der B-Entry e_1 darf sich nicht außerhalb eines Closed-I-Entry, der den Status STARTED hat, befinden.

Wenn bereits ein B-Entry eines Closed-I-Entries ausgeführt wurde, der Closed-I-Entry also bereits gestartet wurde, dürfen solange nur noch B_Entries aus dem Closed-I-Entry ausgeführt werden bis dieser vollständig abgearbeitet ist.

3. Die Precondition von e_1 muß erfüllt sein, d.h. sie muß zu TRUE ausgewertet werden.

4. Es darf keinen anderen B-Entry e_2 mit höherer Priorität geben, für den die Kriterien 1-3 ebenfalls erfüllt sind. In diesem Fall wäre e_2 ausführbar falls er Kriterium 4 erfüllt.

Wenn die Preconditions zweier B-Entries (e_1, e_2) beide wahr sind, und zwischen diesen beiden Entries eine Prioritätsbeziehung $((e_1,e_2) \in R_p)$ existiert, so wird der B-Entry mit der höheren Priorität (e_1) zuerst ausgeführt.

Hiermit ist das Konzept der zu implementierenden Reiseroute vollständig beschrieben. Weiterhin werden Anforderungen an die Schnittstelle gestellt. Diese werden im Folgenden erläutert.

Anforderungen an die Schnittstelle

Es sollen Methoden zum Hinzufügen und Entfernen von Entries definiert werden. Weiterhin muß es die Möglichkeit geben die Entries in der Ausführreihenfolge aus der Reiseroute geliefert zu bekommen.
Um das Definieren von Vorbedingungen zu vereinfachen müssen hier einige Methoden bereitgestellt werden. Es muß die Möglichkeit bestehen aus Strukturen, die boolesche Werte liefern, mit Methoden wie AND, OR und NOT neue Strukturen zu erzeugen. Die Argumente und das Resultat dieser Methoden sind jeweils Objekte welche boolesche Ausdrücke repräsentieren.
Die Methoden $\text{before}(e_1, e_2)$ und $\text{after}(e_1, e_2)$ sollen ebenfalls definiert werden. Dadurch wird ausgedrückt, daß Entry e_1 vor (nach) Entry e_2 ausgeführt wird. Die Funktion $\text{before}(e_1, e_2)$ erweitert die Vorbedingung von e_2 zu $\text{Precond}(e_2) = \text{Precond}(e_2) \wedge D(e_1)$. Die Funktion $\text{after}(e_1, e_2)$ hat den gleichen Effekt wie $\text{before}(e_2, e_1)$.
Die Funktion $\text{atMost}(i, e_1, e_2, ..., e_j)$ gibt an, daß maximal i Entries der Menge $\{e_1, e_2, ..., e_j\}$ ausgeführt werden sollen. Dadurch ändern sich die Preconditions wie folgt: $\text{Precond}(e_k) = \text{Precond}(e_k) \wedge (i > s(e_1) + s(e_2) + ... + s(e_j)$ für $1 \leq k \leq j$.
Weiterhin soll eine Funktion $\text{afterAtLeast}(e_1, ..., e_m, i, e_{m+1}, ..., e_{m+j})$ implementiert werden. Sie legt fest, daß vor dem Ausführen von $e_1, ..., e_m$ erst mindestens i Entries aus $e_{m+1}, ..., e_{m+j}$ ausgeführt werden. Diese Funktion erweitert die Precondition der Entries $e_x \in \{e_1, ..., e_m\}$ wie folgt:
$\text{Precond}(e_x) = \text{Precond}(e_x) \wedge (i \leq d(e_{m+1}) + ... + d(e_{m+j}))$.
Des weiteren sollen Funktionen bereitgestellt werden, die die Konsistenz der Reiseroute überprüfen. So könnte ein Test durchgeführt werden, der Entries findet, die auf Grund von zyklischen Beziehungen nicht ausgeführt werden können.

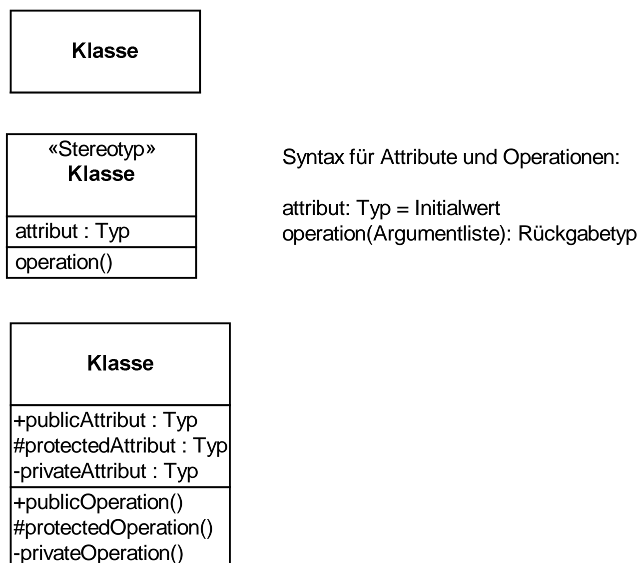
4. Entwurf

In diesem Kapitel werden alle Klassen mit ihren Attributen und Methoden erklärt. Weiterhin wird mit Beispielen gezeigt wie sie beim Erzeugen einer Reiseroute benutzt werden können.

4.1 Die Unified Modeling Language (UML)

UML wird bei der Analyse und beim Design objektorientierter Software eingesetzt. Dabei wird die Software durch verschiedene Typen von Diagrammen genau beschrieben. In dieser Studienarbeit werden nur die Klassendiagramme von UML benutzt. Die in diesem Kapitel beschriebenen Klassen sind in UML Notation dargestellt. Eine Notationsübersicht für die in der Ausarbeitung verwendeten Diagramme ist in der folgenden Abbildung zu sehen.

Klassenbeschreibungsmöglichkeiten



Vererbung

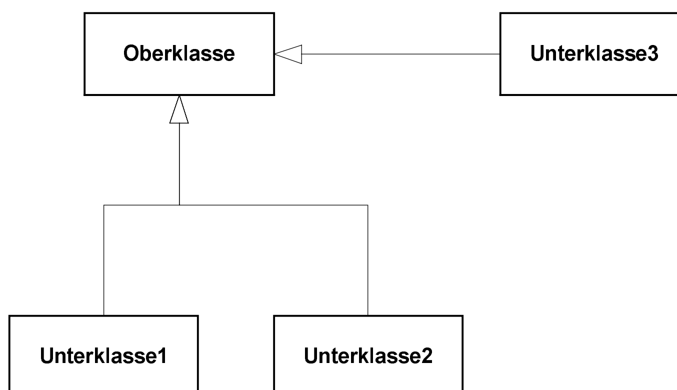


Abbildung 4.0: UML

Die komplette Notationsübersicht von UML ist in [Oester 98] zu finden.

4.2 Entries-Klassen

In der folgenden Abbildung werden die Vererbungshierarchien aller Entry-Klassen dargestellt.

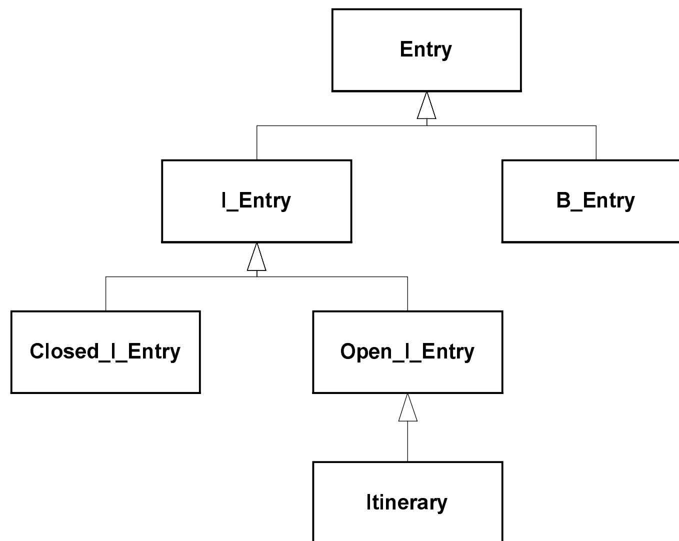


Abbildung 4.1: Entries-Klassen

4.2.1 Die Klasse Entry

Diese Klasse ist die abstrakte Basisklasse aller weiteren Entry-Klassen. In Abbildung 4.2 sind alle Attribute und Methoden dieser Klasse dargestellt.
Die Attribute `prioritiesAfter` und `prioritiesBef` sind zum Speichern der Prioritäten. Im Vector `prioritiesAfter` werden die Entries eingetragen, die eine niedrigere Priorität haben und in `prioritiesBef` werden die Entries eingetragen, die eine höhere Priorität haben. Um sich dies besser merken zu können, gibt es eine Eselsbrücke. In `priorities` werden die Entries eingetragen, die bei erfüllter Vorbedingung auf jeden Fall **nach** dem Entry ausgeführt werden.
In der Eigenschaft `precondition` wird, wie der Name schon sagt, die Vorbedingung gespeichert. Diese muß sich zu TRUE auswerten, daß der Entry ausgeführt werden darf. Die Struktur einer Vorbedingung wird unter 4.3 genauer beschrieben.
Weiterhin hat jeder Entry einen Status, hier mit `state` bezeichnet. Diese Eigenschaft kann die Werte `NOTHING`, `ACTIVE`, `STARTED` und `DONE` annehmen. Beim Erzeugen erhält der Entry den Status `NOTHING`. Ein B-Entry kann durch aktivieren in den Zustand `ACTIVE` und nach erfolgreichem Ausführen in den Zustand `DONE` versetzt werden. Ein I-Entry kann außer dem Zustand `NOTHING` noch die Zustände `STARTED` und `DONE` annehmen. Wenn einer der Entries der I-Entry ausgeführt wurde so wird der Status der I-Entry auf `STARTED` gesetzt. Sobald dann alle Entries der I-Entry abgearbeitet wurden oder keine der Vorbedingungen, der noch nicht ausgeführten Entries erfüllt sind so wird der Status der I-Entry auf `DONE` gesetzt.
Im Attribut `belongsTo` wird festgehalten in welcher I-Entry sich der Entry befindet. Diese Eigenschaft wird unter anderem benötigt um das Reiseroutenobjekt zu finden.

«abstract» **Entry**
#prioritiesAfter : Vector = null #prioritiesBef : Vector = null #precondition : Precond = null #state : int = NOTHING #belongsTo : I_Entry = null +NOTHING : int = 0 +ACTIVE : int = 1 +STARTED : int = 2 +DONE : int = 3
+Entry() #D() : boolean #S() : boolean #d_int() : int #s_int() : int #setState(state : int) : void #getNext(evalList : Vector, doneList : Vector) : B_Entry #getAllNext(evalList : Vector, doneList : Vector, allEntries : Vector) : boolean #getNextInVector(priorities : Vector, evalList : Vector, doneList : Vector) : Entry #getAllNextInVector(priorities : Vector, evalList : Vector, doneList : Vector, allEntries : Vector) : boolean #getAllPrecondTrue(allEntries : Vector) : void #getAbsAllEntries(allEntries : Hashtable) : void #cyclesInPrior(search : Entry, triedYet : Vector) : boolean #deleteAllPriorities() : void #getItinerary() : Itinerary +setPrecond(precond : Precond) : void +getPrecond() : Precond +precondNot(not : Precond) : void +precondAnd(first : Precond, second : Precond) : void +precondOr(first : Precond, second : Precond) : void +precondConst(precond : boolean) : void +delPrecond() : void +getState() : int +getBelongsTo() : I_Entry +getPrioritiesBef() : Vector +getPrioritiesAfter() : Vector

Abbildung 4.2: Klasse Entry

Protected Methoden:

Die Methode `D()` überprüft den Zustand des Entry. Sie liefert TRUE wenn der Zustand `DONE` ist, ansonsten wird FALSE zurückgeliefert.
Auch die Methode `S()` trifft Aussagen über den Zustand des Entry. Sie gibt TRUE zurück wenn der Entry im Zustand `STARTED` ist.
Zu den beiden eben erwähnten Methoden `D()` und `S()` gibt es die entsprechenden Methoden `d_int()` und `s_int()`. Statt einem booleschen Wert liefern Sie einen Integerwert (0 statt FALSE und 1 statt TRUE) zurück.
Die vier zuletzt genannten Methoden werden beim Auswerten der Vorbedingungen benötigt.
Die Methode `setState(state: int)` setzt den übergebenen Status in der Entry.

Um ein B_Entry zu erhalten, daß als nächstes ausgeführt werden kann, wird die Methode `getNext(evalList: Vector, doneList: Vector)` benutzt. Diese Methode wird rekursiv für alle Entries aufgerufen bis ein entsprechendes B_Entry gefunden wird. Wird ein solcher B_Entry gefunden wird er zurückgeliefert, ansonsten wird null zurückgegeben. Im Vector `evalList` werden alle bereits untersuchten Entries gehalten. Dadurch wird gewährleistet, daß nicht die gleichen Entries

mehrfach untersucht werden. Im Vector `doneList` werden alle bereits fertiggestellten Entries gespeichert. So wird sichergestellt, daß abgearbeitete Entries nicht noch mal angefaßt werden. Dieser Mechanismus wird in Kapitel 5.3.1 genauer beschrieben.
Wenn alle ausführbaren B_Entries gesucht werden so wird die Methode `getAllNext(evalList: Vector, doneList: Vector, allEntries: Vector)` benutzt. Sie funktioniert im Prinzip gleich wie `getNext(...)`, bricht aber beim Finden eines ausführbaren B_Entries nicht ab sondern trägt den gefundenen B_Enry im Vector `allEntries` ein. Wenn die Methode ausführbare Entries gefunden hat, gibt sie den Wert TRUE sonst FALSE zurück.
Die beiden Methoden `getNextInVector(priorities: Vector,...) und getAllNextInVector(priorities: Vector,...)` werden in den Funktionen `getNext(...)` und `getAllNext(...)` benutzt. Um ein ausführbares B_Entry zu finden wird sowohl im Vector `prioritiesBef` als auch im Vector `prioritiesAfter` gesucht. Dazu werden die gerade genannten Methoden verwendet. Ihnen wird einer der Vectoren `prioritiesBef` oder `prioritiesAfter` im Parameter `priorities` übergeben. Sie checken dann für alle Entries im übergebenen Vector ob sie ausführbar sind.
Die Methode `getAllPrecondTrue(allEntries: Vector)` wird rekursiv für alle Entries aufgerufen. Sie gibt im Vector `allEntries` alle B_Entries zurück, die die ersten drei Kriterien für ein ausführbares B_Entry erfüllen. Diese Kriterien sind im vorigen Kapitel genauer erläutert. Prioritätsbeziehungen werden also im Gegensatz zur Methode `getAllNext(...)` nicht beachtet.

Als nächstes wird die Methode `getAbsAllEntries(allEntries: Hashtable)` beschrieben. Sie durchläuft rekursiv alle Entries und sammelt diese in der Hashtable `allEntries`. Diese Methode wird beim Einfügen neuer Entries in die Reiseroute benutzt. Da ausgeschlossen werden soll, daß ein Entry mehrmals in der Itinerary vorkommt werden alle existierenden Entries mit dieser Methode gelesen und dann wird geschaut ob der einzufügende Entry in der Hashtable `allEntries` bereits existiert. Wann diese Methode aufgerufen werden muß und wann nicht wird im Abschnitt 4.2.5 beschrieben.

Sehr wichtig ist die Methode `cyclesInPrior(search: Entry, triedYet: Vector)`. Sie wird beim Hinzufügen von Prioritäten verwendet. Es wird überprüft ob beim Hinzufügen einer Prioritätsbeziehung Zyklen in den Prioritäten entstehen würden. Wenn dies der Fall ist wird TRUE ansonsten FALSE zurückgegeben. Wie der Mechanismus zum Verhindern von Zyklen genau funktioniert wird in 5.1 erklärt.

Mit der Methode `deleteAllPriorities()` werden alle Prioritätsbeziehungen eines Entries gelöscht. Diese Methode wird beim Löschen eines Entries benutzt.

Zuletzt wird die Methode `getItinerary()` kurz erklärt. Jeder Entry hat mit dieser Methode die Möglichkeit sich das Reiseroutenobjekt zu holen. Dabei wird in einer Schleife solange das oben erwähnte Attribut `belongsTo` gelesen, bis entweder ein Itinerary Objekt gefunden oder `belongsTo` gleich null ist. Somit liefert die Methode entweder ein Itinerary Objekt oder null zurück.

Public Methoden:

Zum einfachen Setzen der Vorbedingung dient die Methode `setPrecond(precond: Precond)`. Zuerst wird getestet ob alle Entries, die in der übergebenen Precondition enthalten sind, im selben I_Entry liegen. Ist dies nicht der Fall, wird die Precondition nicht gesetzt und es wird eine NotInThisEntryException ausgelöst. Wenn der Entry noch nicht zu einem I_Entry hinzugefügt wurde kann dieser Test nicht durchgeführt werden weil kein I_Entry Objekt verfügbar ist. Dann wird ebenfalls eine Exception vom oben erwähnten Typ ausgelöst. Bevor also einem Entry eine Precondition zugeordnet werden kann muß dieser erst zur Reiseroute hinzugefügt werden.
Mit der Methode `getPrecond()` wird die Precondition eines Entries gelesen.

Um aus vorhandenen Vorbedingungen eine neue zusammenzubauen und direkt ins Entry zu setzen werden die folgenden Methoden bereitgestellt:

- `precondNot(not: Precond): void`
  Die Methode erweitert die übergebene Precondition in dem sie sie mit dem booleschen Operator „not“ verknüpft. Die so erhaltene Precondition liefert folglich immer das negierte Ergebnis der übergebenen Precondition. Die neue Vorbedingung wird dann mit der oben erwähnten Methode `setPrecond(...)` gesetzt. Somit wird auch in dieser Methode eine NotInThisEntryException ausgelöst, wenn die übergebene Precondition Fehler enthält.
- `precondAnd(first: Precond, second: Precond): void`
  Diese Methode verknüpft die Preconditions `first` und `second` mit dem booleschen Operator „and“ und setzt die so erhaltene Vorbedingung im Entry. Dabei kann wie bei `precondNot(...)` beschrieben eine NotInThisEntryException ausgelöst werden.
- `precondOr(first: Precond, second: Precond): void`
  Hier werden die Preconditions `first` und `second` mit dem booleschen Operator „or“ hintereinander gesetzt. Sonst verhält sich die Methode wie `precondAnd(...)`.

Um die Precondition des Entry direkt auf wahr oder falsch zu setzen dient die Methode `precondConst(precond: boolean): void`. Die Methode erzeugt dann eine Precondition, die sich je nach übergebenem Wert immer zu wahr oder zu falsch auswertet.

Weiterhin existieren einige „get-Funktionen“ um die Attribute zu lesen.
Bei den Methoden `getPrioritiesBef()` und `getPrioritiesAfter()` wird nur eine Kopie des Vektors zurückgegeben wobei aber die Entries Objekte im Vector nicht kopiert werden. Es handelt sich dabei um eine sogenannte „flache“ Kopie des Vectors.

4.2.2 Die Klasse B_Entry

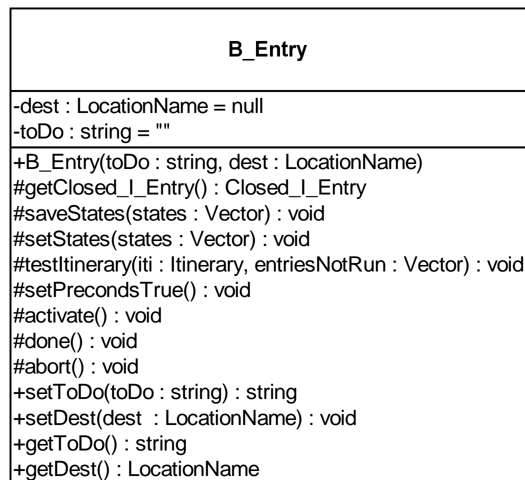


Abbildung 4.3: Klasse B_Entry

Diese Klasse stellt einen Tripel (precondition, node, method) dar. Das Attribut `precondition` wird von der Klasse Entry geerbt. Der Knoten, also der Ort auf dem der B_Entry ablaufen soll, wird im Attribut `dest` gespeichert. Den Namen der Methode, die dort vom Agenten ausgeführt werden soll wird als String in `toDo` abgelegt.

Beim Erzeugen eines B_Entry mit dem Konstruktor wird kontrolliert ob die Methode, die durch den Parameter `toDo` übergeben wurde, im Agenten existiert. Dazu wurde die Klasse Agent um die Methode `methodExecutable(name: string)` erweitert. Diese Methode liefert true zurück wenn es im Agenten eine Methode mit dem Namen `name` gibt die als public deklariert wurde und keine Parameter verlangt. Das Agenten Objekt muß erreichbar sein um dessen Methode `methodExecutable(...)` aufrufen zu können. Als Lösung dafür wurde die Itinerary Klasse (s. Kap. 4.2.5) um das Attribut `agent` erweitert. Der Agent wird somit über die Methode `Entry.getItinerary()` und anschließendem Lesen des Attributs `agent` des I_Entry geholt. Existiert die auszuführende Methode des B_Entry im Agenten nicht, so wird eine MethodNotCorrectException ausgelöst.

Protected Methoden:

Die Methode `getClosed_I_Entry()` sucht nach einem sich nicht im Zustand `DONE` befindlichen Closed_I_Entry in dem sich der B_Entry befindet. Dazu wird in einer Schleife das Attribut `Entry.belongsTo` gelesen bis entweder ein Closed_I_Entry mit den eben angesprochenen Kriterien gefunden wird oder bis `belongsTo` gleich null ist. Diese Methode wird zum Durchsetzen der korrekten Reihenfolge bei der Ausführung der B_Entries benötigt (s. Kap. 5.3.1).

Die Reiseroute kann auf niemals ausführbare B_Entries untersucht werden. Dabei werden die folgenden drei Methoden benötigt:

- `saveStates(states: Vector): void`
  Diese Methode sichert alle Zustände der umgebenden I_Entries im Vector `states` ab. Während dem Test ob das B_Entry ausführbar ist können sich diese Zustände ändern deshalb werden sie gesichert.
- `setStates(states: Vector): void`
  Mit dieser Methode werden die mit `saveStates(...)` gesicherten Zustände der I_Entries wiederhergestellt. Dadurch befindet sich die Reiseroute nach dem gesamten Test wieder im gleichen Zustand wie davor.
- `testItinerary(iti: Itinerary, entriesNotRun: Vector): void`
  Diese Methode simuliert das Ausführen des B_Entries. Um nach dieser Simulation den ursprünglichen Zustand wiederherstellen zu können, merkt sie sich diesen. Die Zustände der umgebenden I_Entries werden mit der Methode `saveStates(...)` ebenfalls gesichert. Danach wird der Zustand des Entry auf `DONE` gesetzt. Es wird also so getan als wäre der B_Entry ausgeführt worden. Dieser B_Entry trägt sich aus dem Vector `entriesNotRun` aus. In diesem Vector werden alle Entries gehalten für die die Methode `testItinerary(...)` noch nicht aufgerufen wurde, die also während der gesamten Simulation der Reiseroute noch nicht ausgeführt werden konnten. Danach wird geschaut welche B_Entries jetzt ausführbar sind. Dies geschieht mit der Methode `Itinerary.getAllPrecondTrue()`. Für alle B_Entries, die jetzt abgearbeitet werden können wird rekursiv die Methode `testItinerary(...)` aufgerufen. Danach wird der ursprüngliche Zustand des B_Entries sowie die Zustände der I_Entries, in denen sich der B_Entry befindet, zurückgesetzt.

Weitere Informationen dazu sind im Abschnitt 4.2.5 unter `Itinerary.testItinerary()` sowie im Kapitel 5.3.2 zu finden.

Die Precondition eines Open_I_Entry muß nur beim Ausführen seines ersten B_Entries erfüllt sein. Danach können alle enthaltenen B_Entries ohne Berücksichtigung der Precondition der Open_I_Entry abgearbeitet werden. Das aber nur dann, wenn ihre eigene Precondition wahr ist. Dies wird in der Reiseroute dadurch gelöst, daß nach dem Ausführen eines B_Entries die Preconditions sämtlicher Open_I_Entries, in denen sich der B_Entry befindet, auf TRUE gesetzt werden. Dazu wird die Methode `setPrecondsTrue()` benutzt.

Mit den Methoden `activate()`, `done()` und `abort()` wird der Status des B_Entry verändert. Bevor ein B_Entry ausgeführt werden darf muß er mit `activate()` in den Zustand `ACTIVE` versetzt werden. Dabei wird dafür gesorgt, daß sich immer nur ein B_Entry im Zustand `ACTIVE` befindet. Wenn also bereits ein anderes B_Entry aktiviert ist wird eine AnotherEntryActiveException ausgelöst. Ebenfalls vermieden werden muß das Aktivieren eines Entries der bereits abgearbeitet wurde, und sich somit bereits im Zustand `DONE` befindet. Dies wird durch Auslösen einer EntryDoneException unterbunden. Aus dem aktivierten Zustand kann der B_Entry entweder nach erfolgreicher Abarbeitung von `toDo` mittels `done()` auf den Status `DONE` oder bei Nichterreichen der Location sowie bei Fehlschlagen von `toDo` mit der Methode `abort()` in den Zustand `NOTHING` gesetzt werden. Wenn beim Aufruf von `done()` kein B_Entry aktiviert ist wird eine NoEntryActiveException ausgelöst. Ist der B_Entry für den `done()` aufgerufen wird nicht aktiviert so wird eine AnotherEntryActiveException erzeugt. Die Methode `abort()` verhält sich gleich wie `done()` nur setzt sie den Zustand wieder auf `NOTHING` zurück. Der Zustand `NOTHING` bedeutet, daß der B_Entry noch ausgeführt werden muß.

Public Methoden:

Zum Setzen der Attribute gibt es die Methoden `setToDo(toDo: string)`, `setDest(dest: LocationName)`.
Die Methode `setToDo(...)` prüft ob die Methode mit dem Namen `toDo` im Agenten existiert. Ist dies nicht der Fall wird eine MethodNotCorrectException ausgelöst. Genaueres zu diesem Vorgang wurde weiter oben in der Beschreibung des Konstruktors der Klasse B_Entry erwähnt.
Beim Setzen des Attributs `dest` mit der Methode `setDest(...)` werden keine Kontrollen durchgeführt, der übergebene Parameter wird einfach ins Attribut `dest` gesetzt.

4.2.3 Die Klasse I_Entry

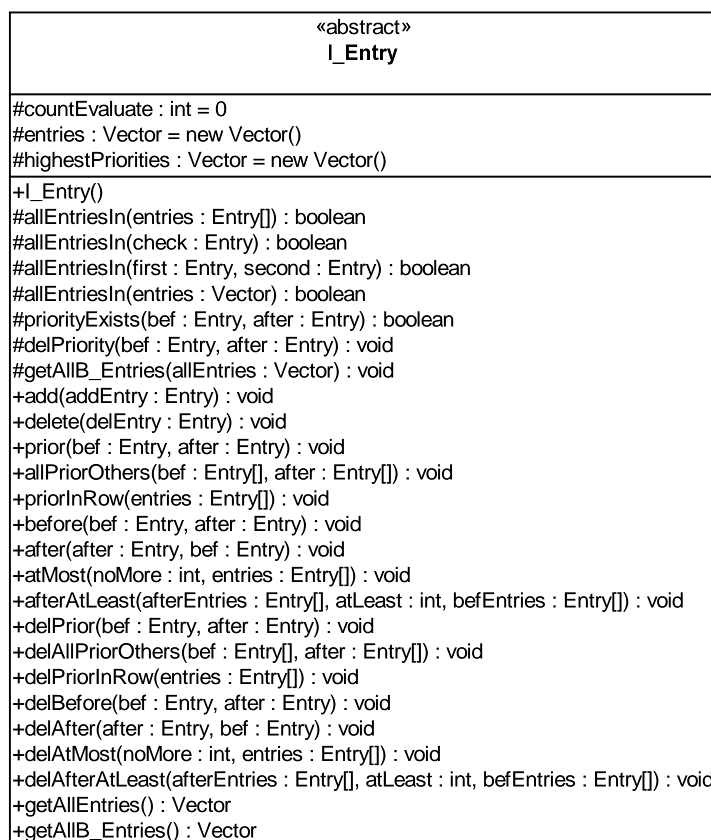


Abbildung 4.4: Klasse I_Entry

Die Klasse I_Entry stellt die Grundlage zum Schachteln und Gruppieren von B_Entries dar. Weiterhin enthält sie Funktionen zum Erzeugen von Vorbedingungen und zum Definieren von Prioritäten.

Sie beinhaltet das Attribut `countEvaluate`, mit dem gewährleistet wird, daß Vorbedingungen, die von mehreren Entries benutzt werden nicht mehrfach ausgewertet werden. Dies wird im Kapitel 5.2.2 genauer beschrieben werden.
Das Attribut `entries` wird benötigt um die Entries zu halten, die direkt zum I_Entry hinzugefügt werden.

Im Vector `highestPriorities` sind alle Entries enthalten für die keine Prioritätsbeziehung definiert wurde, durch welche ein anderes Entry eine höhere Priorität erhält als die Entries in `highestPriorities`.

Beispiel: Es existiert ein I_Entry e_6, der wie folgt aussieht:

$$e_6 = (\text{TRUE}, \{e_1, e_2, e_3, e_4, e_5\}, R_p, \text{type})$$

$$R_p = \{(e_1, e_2), (e_2, e_3), (e_3, e_4), (e_5, e_4)\}$$

$$\text{highestPriorities} = \{e_1, e_5\}$$

Der Entry e_2 ist nicht in `highestPriorities` enthalten, da durch die Prioritätsbeziehung (e_1, e_2) dem Entry e_1 eine höhere Priorität als dem Entry e_2 zugewiesen wird. Es wurde aber keine Prioritätsbeziehung definiert, so daß einem Entry aus $\{e_1, e_2, e_3, e_4, e_5\}$ eine höhere Priorität als den Entries e_1, e_5 zugewiesen wurde.

Protected Methoden:

Prioritäten dürfen nur zwischen Entries, die in den selben I_Entry eingefügt wurden definiert werden. Somit muß die Möglichkeit bestehen für zwei Entries zwischen denen eine Priorität definiert werden soll, festzustellen ob sie im selben I_Entry liegen. Diese Funktionalität wird von der Methode `allEntriesIn(...)` bereitgestellt. Sie existiert in vier verschiedenen Varianten:
- `allEntriesIn(entries: Entry[]): boolean`
- `allEntriesIn(check: Entry): boolean`
- `allEntriesIn(first: Entry, second: Entry): boolean`
- `allEntriesIn(entries: Vector): boolean`

Diese Methode überprüft ob alle durch die Parameter übergebenen Entries im aufrufenden I_Entry liegen. Ist dies der Fall wird TRUE ansonsten FALSE zurückgegeben.

Die nächsten beiden Methoden werden beim Löschen von Prioritäten benötigt. Dies ist zum einen die Methode `priorityExists(bef: Entry, after: Entry)`. Sie überprüft ob eine Prioritätsbeziehung zwischen Entry `bef` und `after` besteht. Wenn die gesuchte Beziehung gefunden wird liefert die Methode TRUE ansonsten FALSE zurück.
Die zweite Methode heißt `delPriority(bef: Entry, after: Entry)` und löscht die Priorität zwischen `bef` und `after`.

Die Methode `getAllB_Entries(allEntries: Vector)` sammelt alle B_Entries die sich direkt oder indirekt im I_Entry befinden. Sie durchläuft den `entries` Vector in einer Schleife. Stößt sie dabei auf ein B_Entry wird er in den Parameter `allEntries` eingetragen. Handelt es sich bei dem untersuchten Entry um ein I_Entry wird für diesen wieder rekursiv die Methode `getAllB_Entries(...)` aufgerufen.

Public Methoden:

Um Entries in ein I_Entry einzutragen wird die Methode `add(addEntry: Entry)` benutzt. Dabei werden folgende Überprüfungen durchgeführt:

- In ein I_Entry das bereits vollständig abgearbeitet wurde, sich also im Zustand `DONE` befindet dürfen keine Entries mehr eingetragen werden. Also wird zuerst überprüft ob der Status der I_Entry ungleich `DONE` ist. Ansonsten wird eine EntryDoneException ausgelöst.
- In ein I_Entry dürfen nur dann Entries eingefügt werden wenn der I_Entry bereits zur Reiseroute hinzugefügt wurde. Somit wird mit der Methode `Entry.getItinerary()` (s. Kap. 4.2.1) nach einem Itinerary Objekt gesucht. Findet diese Methode kein Reiseroutenobjekt so wird eine I_EntryNotAddedException erstellt.
- Als letztes wird kontrolliert, daß jedes Entry sich nur einmal in der Itinerary befindet. Dazu wird im gefundenen Itinerary Objekt in der Hashtable `allEntries` (s. Kap. 4.2.5) geschaut ob der einzufügende Entry bereits existiert. Ist dies der Fall so wird eine EntryExistsException ausgelöst. Ansonsten wird der Entry sowohl zum Vector `entries` der I_Entry als auch zum Vector `Itinerary.allEntries` hinzugefügt, so daß sich beim nächsten Aufruf von `add(...)` wieder alle Entries der Itinerary in der Hashtable befinden.

In Abbildung 4.5 ist die Verwendung der Methode `add(...)` an einem Beispiel erläutert. Rechts wird gezeigt wie die Reiseroute verändert wird wenn der links daneben stehende Code ausgeführt wird.

Um einen Entry wieder aus einem I_Entry zu löschen kann die Methode `delete(delEntry: Entry)` benutzt werden. Bevor der `delEntry` gelöscht wird werden folgende Tests durchgeführt:

- Zuerst wird überprüft ob sich der Entry `delEntry` im I_Entry befindet. Ist dies nicht der Fall wird eine NotInThisEntryException erzeugt.
- Ist entweder der zu löschende Entry oder der I_Entry im Zustand `DONE` darf nicht gelöscht werden. Also werden die Zustände überprüft und wenn einer davon `DONE` ist wird eine EntryDoneException geworfen.

Jetzt kann der Entry aus der I_Entry entfernt werden. Zuerst werden sämtliche Prioritätsbeziehungen des zu löschenden Entries entfernt. Danach wird die Verbindung zur Precondition gelöscht und das Attribut `precondition` auf null gesetzt. Als letztes wird der Entry `delEntry` aus dem Vector `entries` der I_Entry und aus der Hashtable `allEntries` der Itinerary ausgetragen.

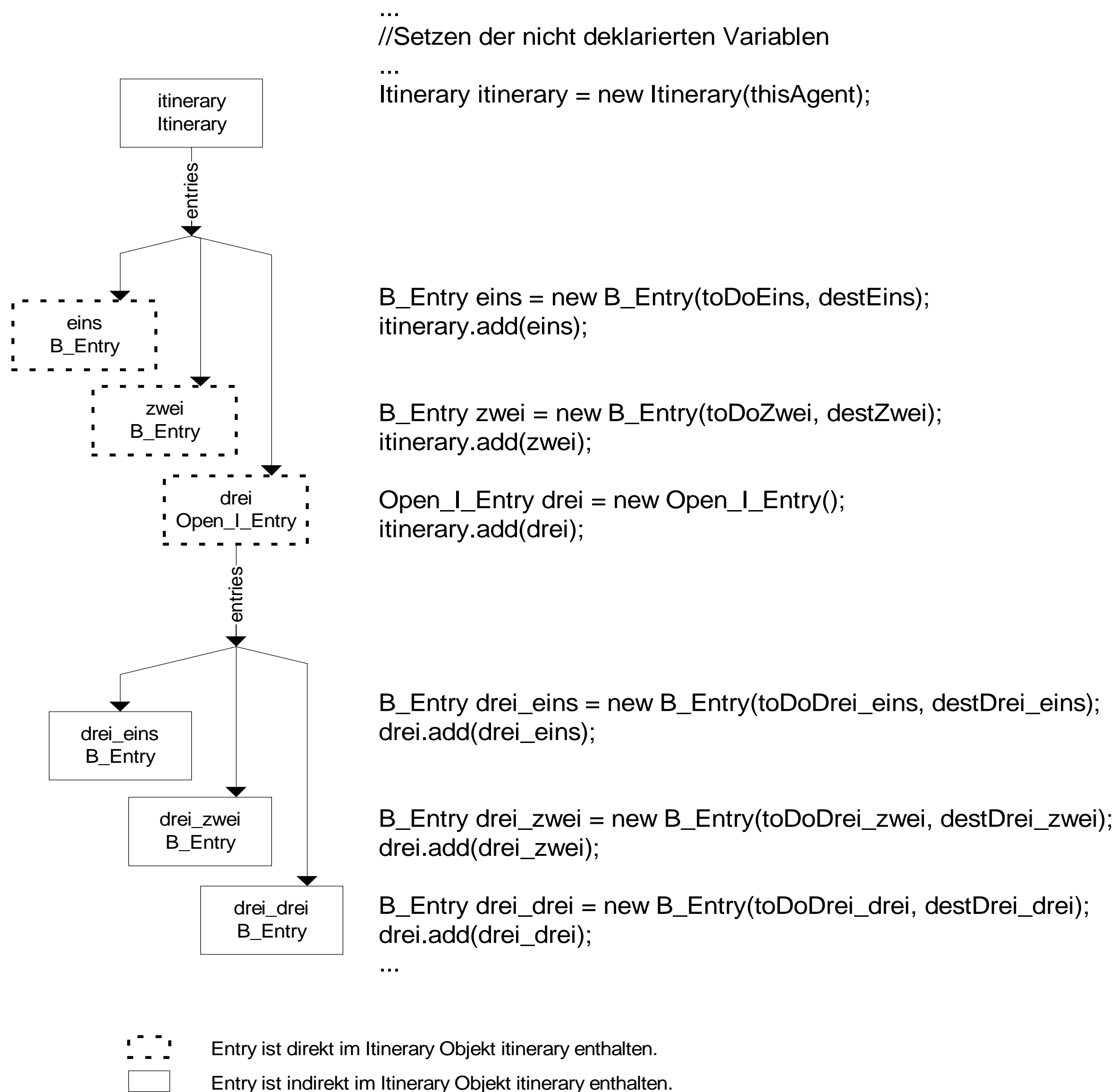


Abbildung 4.5: Aufbau einer einfachen Reiseroute

Prioritäten:

Um Prioritäten zwischen Entries definieren zu können wurden die folgenden Methoden eingeführt:

- `prior(bef: Entry, after: Entry): void`
- `priorInRow(entries: Entry[]): void`
- `allPriorOthers(bef: Entry[], after: Entry[]): void`

Die Entries, die über die Parameter übergeben werden, müssen sich alle direkt im I_Entry befinden. Dies wird mit der weiter oben beschriebenen Methode `allEntriesIn(...)` überprüft. Sind nicht alle im I_Entry enthalten wird eine NotInThisEntryException ausgelöst. Bevor dann die Beziehungen in den Attributen `prioritiesBef` und `prioritiesAfter` eingetragen werden, wird zuerst überprüft ob dabei Zyklen entstehen würden. Dies wird in Kapitel 5.1 genauer erklärt. Wenn bei der Überprüfung herauskommt, daß ein Zyklus in den Prioritäten entstehen würde, wird eine CyclesInPriorException ausgelöst. Im anderen Fall werden die Beziehungen in der Reiseroute eingefügt. Für jede der Methoden wird im Folgenden

kurz beschrieben, welche Prioritäten eingetragen werden müssen. Wie dann die Struktur der Entries nach Ausführen der Methoden aussieht wird in Abbildung 4.6 verdeutlicht.
Beim Aufruf von `prior(bef, after)` muß die Prioritätsbeziehung (bef, after) in der Reiseroute eingetragen werden.

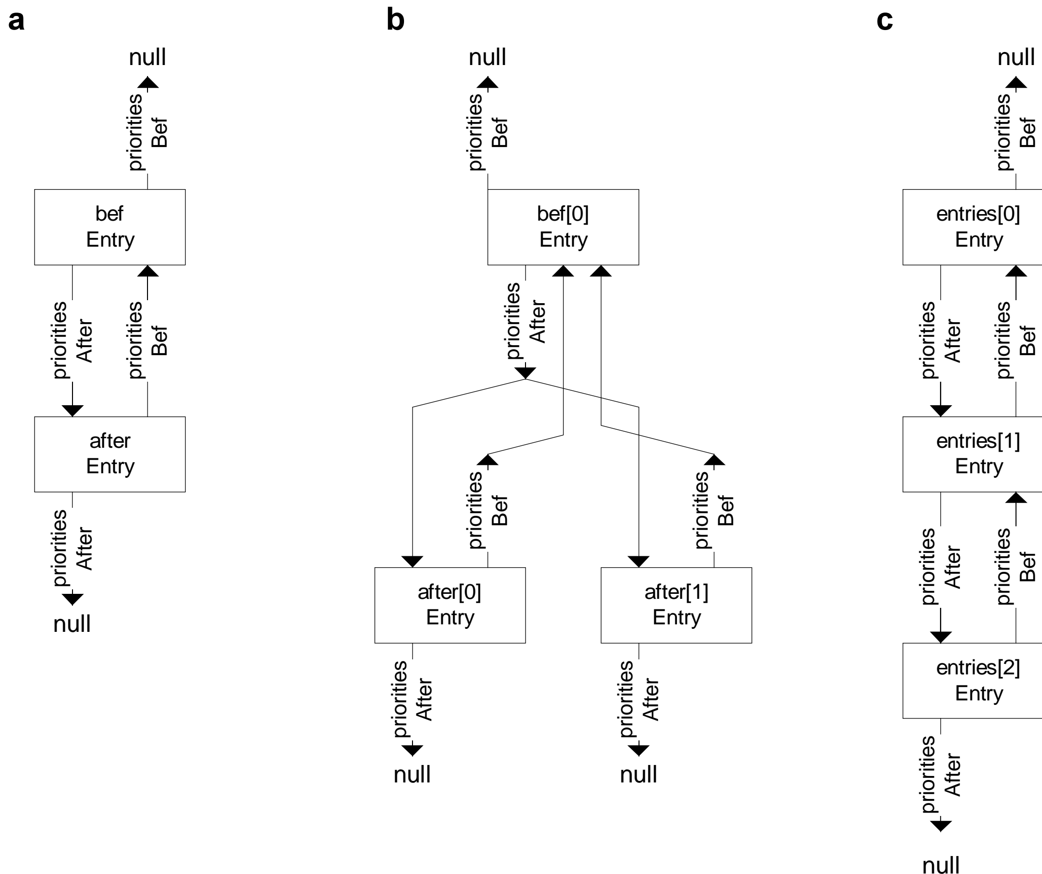
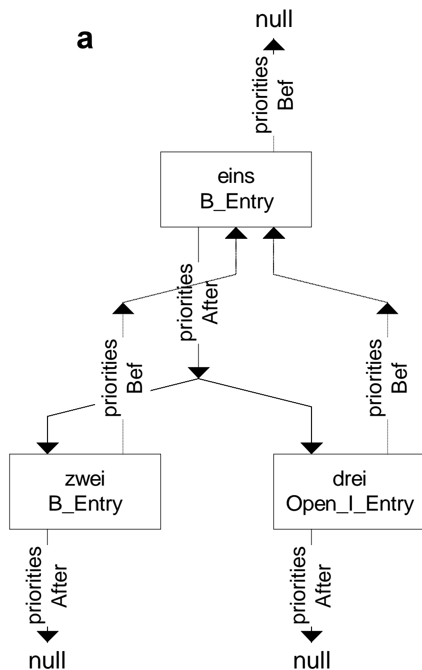


Abbildung 4.6: Definieren von Prioritäten

a: prior(bef: Entry, after: Entry): void
b: allPriorOthers(bef: Entry[], after: Entry[]): void
c: priorInRow(entries: Entry[]): void

Bei folgendem Aufruf `priorInRow(entries[3])` werden die Beziehungen (entries[0], entries[1]) und (entries[1], entries[2]) eingetragen. Dabei muß beachtet werden, daß nicht die transitive Hülle gebildet wird. Ansonsten müßte zusätzlich die Beziehung (entries[0], entries[2]) eingetragen werden. Mit der Methode `priorInRow(...)` wird also eine durchlaufende Reihe von Prioritätsbeziehungen erstellt.
Beim Aufruf `allPriorOthers(bef[1], after[2])` wird zwischen jedem der Entries im Array `bef` und allen Entries im Array `after` eine Prioritätsbeziehung definiert. In diesem Fall werden die Beziehungen (bef[0], after[0]) und (bef[0], after[1]) erzeugt.

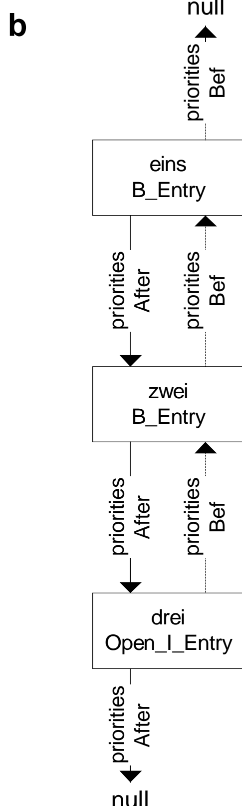
Die Verwendung dieser Methoden und die resultierenden Prioritätsbeziehungen werden in Abbildung 4.7 vereinfacht dargestellt.




```
//Code aus Abbildung 4.5

Zwei Möglichkeiten zum Erstellen:
1.        itinerary.prior(eins, zwei);
          itinerary.prior(eins, drei);

2.        Entry[] bef = {eins};
          Entry[] after = {zwei, drei};
          itinerary.allPriorOthers(bef, after);
```




```
//Code aus Abbildung 4.5

Entry[] row = {eins, zwei, drei};
itinerary.priorInRow(row);
```

Möglichkeiten um Priorität zwischen Entry eins und zwei zu löschen:

```
1.        itinerary.delPrior(eins, zwei);

2.        Entry[] befDel = {eins};
          Entry[] afterDel = {zwei};
          itinerary.delAllPriorOthers(befDel, afterDel);

3.        Entry[] row = {eins, zwei};
          itinerary.delPriorInRow(row);
```

Falsch wäre zum Beispiel:

```
          Entry[] befDel = {eins};
          Entry[] afterDel = {zwei, drei};
          itinerary.allPriorOthers(befDel, afterDel);
```

Die Priorität zwischen Entry eins und drei existiert nicht und somit wird nichts gelöscht.

Abbildung 4.7: Definition und Löschen von Prioritäten

Für die drei Methoden, die Prioritätsbeziehungen setzen, gibt es entsprechende Löschoperationen, die diese Beziehungen wieder aus der Reiseroute entfernen. Dies sind:
- `delPrior(bef: Entry, after: Entry): void`
- `delPriorInRow(entries: Entry[]): void`
- `delAllPriorOthers(bef: Entry[], after: Entry[]): void`

Dabei ist es auch möglich nur Teilmengen von erstellten Prioritäten zu löschen. Die Methode zum Löschen von Prioritäten muß also nicht zwingend mit denselben Parametern wie die entsprechende Erstellungsmethode aufgerufen werden. Prioritäten werden aber nur entfernt wenn alle zu löschenden Prioritäten vorhanden sind.
In Abbildung 4.7 b wird beschrieben welche Möglichkeiten es gibt um die Prioritätsbeziehung zwischen B_Entry „eins“ und „zwei“ zu löschen.

Vorbedingungen:

Zum Erstellen von Vorbedingungen sind ebenfalls Methoden vorhanden. Mit den Methoden `before(bef: Entry, after: Entry)` und `after(after: Entry, bef: Entry)` läßt sich festlegen, daß das Entry `bef` vor dem Entry `after` ausgeführt wird.
Die Methode `atMost(noMore: int, entries: Entry[])` gibt einem die Möglichkeit die Anzahl (`noMore`) maximal auszuführender Entries aus einer Menge (`entries`) festzulegen.
Bei `afterAtLeast(afterEntries:Entry[], atLeast: int, befEntries: Entry[])` wird eine Beziehung zwischen zwei Mengen von Entries (`afterEntries, befEntries`) festgelegt. Die Entries des Parameters `afterEntries` werden erst ausgeführt wenn mindestens `atLeast` Entries aus dem Parameter `befEntries` ausgeführt wurden.

Bei den vier zuletzt genannten Methoden muß wieder darauf geachtet werden, daß alle Entries, die in Parametern übergeben werden im ausführenden I_Entry enthalten sind. Andernfalls wird wieder eine NotInThisEntryException erzeugt.
Ein Beispiel für die Verwendung von `before(...)` und `afterAtLeast(...)` zeigen die Abbildungen 4.18 und 4.22.

Auch bei den Vorbedingungen gibt es die Möglichkeit Änderungen rückgängig zu machen. Dazu stehen die Methoden `delBefore(bef: Entry, after: Entry)`, `delAfter(after: Entry, bef: Entry)`, `delAtMost(noMore: int, entries: Entry[])` und `delAfterAtLeast(afterEntries:Entry[], atLeast: int, befEntries: Entry[])` zur Verfügung.

Diese werden im Folgenden nacheinander genauer betrachtet:
- `delBefore(bef: Entry, after: Entry): void` bzw. `delAfter(after: Entry, bef: Entry): void`

Durch die Methode `before(...)` bzw. `after(...)` wird ein Predicate Objekt erzeugt, welches dann an die bereits bestehende Precondition des `after` Entry‘s über ein PrecondAnd Objekt angehängt wird (s. Abbildung 4.18). Nun gilt es dieses Predicate Objekt zu finden und aus der Precondition des `after` Objekts wieder zu entfernen.

In der Methode `delBefore(...)` bzw. `delAfter(...)` wird `delPredicate(...)` (s. Abschnitt 4.3.1) für das `after` Entry aufgerufen. Wenn das Predicate Objekt nicht gefunden wird so wird eine PrecondDoesNotExistException ausgelöst.

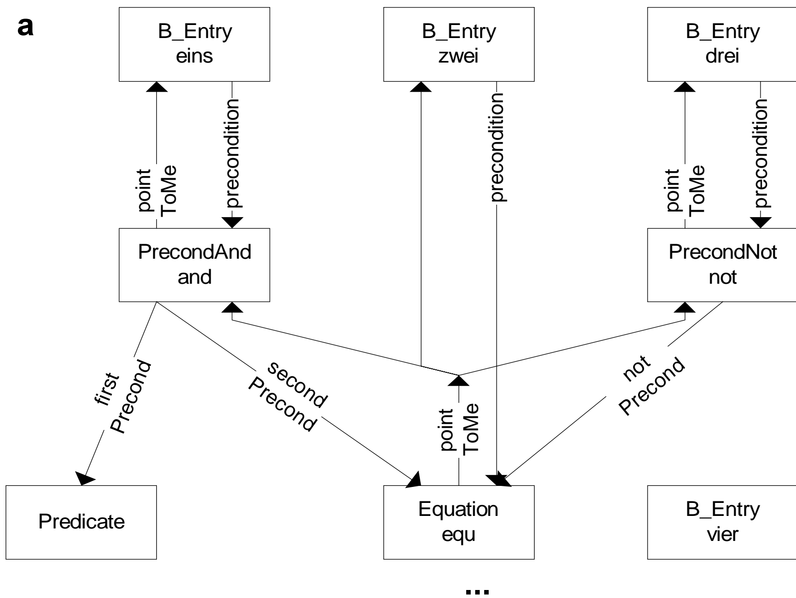
- `delAtMost(noMore: int, entries: Entry[]): void`

Diese Methode muß ein Equation Objekt (s. Abschnitt 4.3.5), das durch die Methode `atMost(...)` erstellt wurde finden und löschen. Um die Löschaktion korrekt ausführen zu können müssen die Parameter beim Löschen exakt mit denen beim Erstellen des Equation Objekts mit der Methode `atMost(...)` übereinstimmen.
Für das Finden der korrekten Equation wird die Methode `Precond.findEquation(i: int, operator: Operator, entries Entry[], method: int): Equation` (s. Abschnitt 4.3.1) benutzt.
Die Methode `delAtMost(...)` ruft `findEquation(...)` für die Precondition des ersten Entries im Feld `entries` auf. Wird das Equation Objekt nicht gefunden wird eine PrecondDoesNotExistException ausgelöst. Ansonsten wird das Equation Objekt aus allen Precondition Objekten, die darauf verweisen gelöscht. Dies wird in Abbildung 4.8 dargestellt.

- `delAfterAtLeast(afterEntries: Entry[], atLeast: int, befEntries: Entry[]):void`

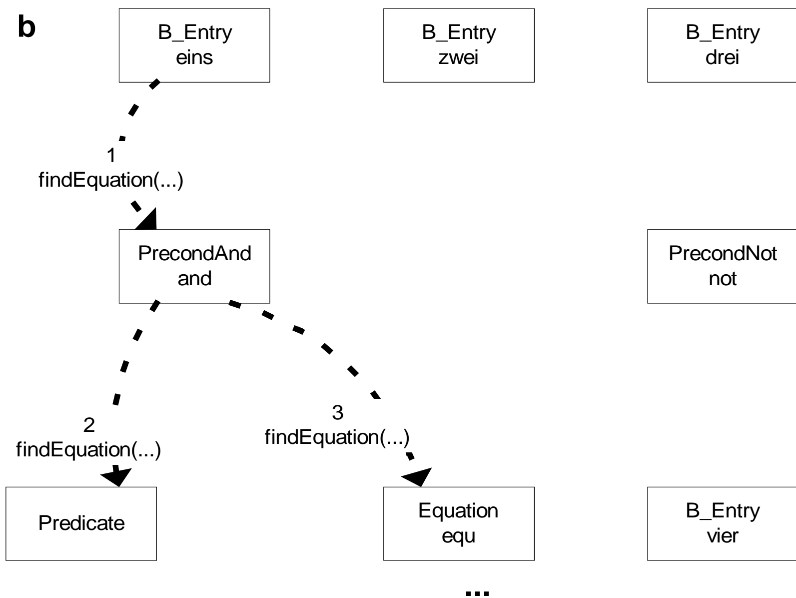
Wie bei `delAtMost(...)` muß auch hier ein Equation Objekt gesucht und gelöscht werden.
Die Parameter `atLeast` und `befEntries` müssen mit denen beim Erzeugen der Equation übereinstimmen, denn nur durch diese Werte ist die Equation eindeutig zu identifizieren. Im Parameter `afterEntries` können aber auch weniger Entries als beim Aufruf von `afterAtLeast(...)` angegeben werden. Nur in den Preconditions der in `afterEntries` enthaltenen Entries wird das Equation Objekt entfernt.
Um dies zu gewährleisten wird hier die Funktion `findEquation(caller: Object, i: int, operator: Operator, entries Entry[], method: int, pointTo: Vector): Equation` (s. Abschnitt 4.3.1) benutzt. Die Methodenaufrufe von findEquation(...) und die entstehende Struktur nach dem Löschen sind in Abbildung 4.9 dargestellt. Die beiden gepunkteten Objekte sind die Objekte, die die Methode `findEquation(...)` im Vector `pointTo` zurückliefert. Diesen Objekten muß die Nachricht geschickt werden, daß die Equation gelöscht werden soll.

Um Informationen über die in einem I_Entry enthaltenen Entries zu erhalten, werden die Methoden `getAllEntries()` und `getAllB_Entries()` zur Verfügung gestellt. Mit `getAllEntries()` wird eine „flache“ Kopie des Vectors entries des I_Entries zurückgeliefert. Man erhält nur die direkt in die I_Entry eingefügten Entries. Wenn man auch die indirekt enthaltenen Entries bekommen will muß man für die erhaltenen I_Entries rekursiv `getAllEntries()` aufrufen. Diese Vorgehensweise wird bei der Methode `getAllB_Entries()` verwendet. Sie liefert alle B_Entries ob direkt oder indirekt enthalten zurück. Dabei wird die weiter oben vorgestellte Methode `getAllB_Entries(allEntries: Vector)` benutzt. In Abbildung 4.5 wird verdeutlicht was direkt und indirekt enthalten bedeutet.



Diese Struktur könnte durch folgende Codezeilen entstanden sein:

```
//Erzeugen eines I_Entry names
//i_Entry. Einfügen in Itinerary
//B_Entries erzeugen und in
//I_Entry einfügen.
...
iEntry.after(eins, vier);
Entry[] atMost = {eins, zwei, drei};
i_Entry.atMost(2, atMost);
drei.precondNot(drei.precondition);
```



Jetzt wird folgende Zeile ausgeführt:

```
iEntry.delAtMost(2, atMost);
```

Bild zeigt die Nachrichten, die benötigt werden um das Equation Objekt zu finden.

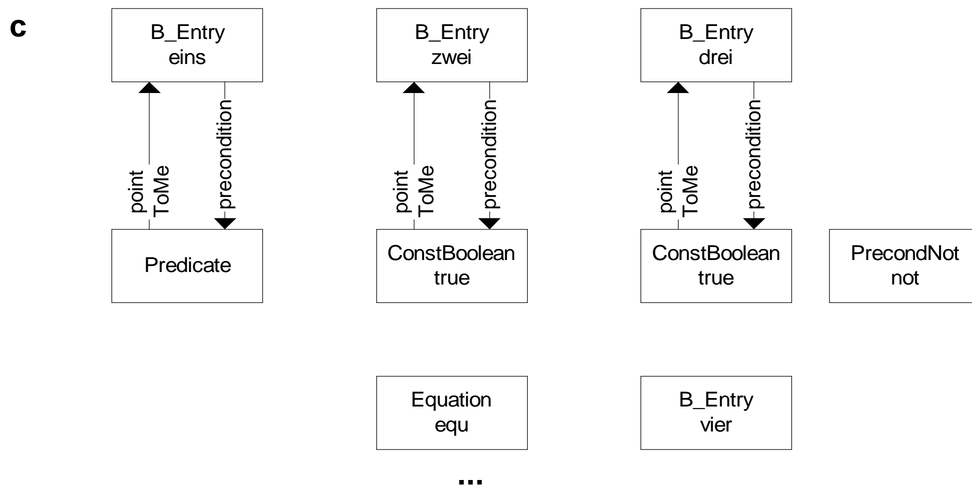
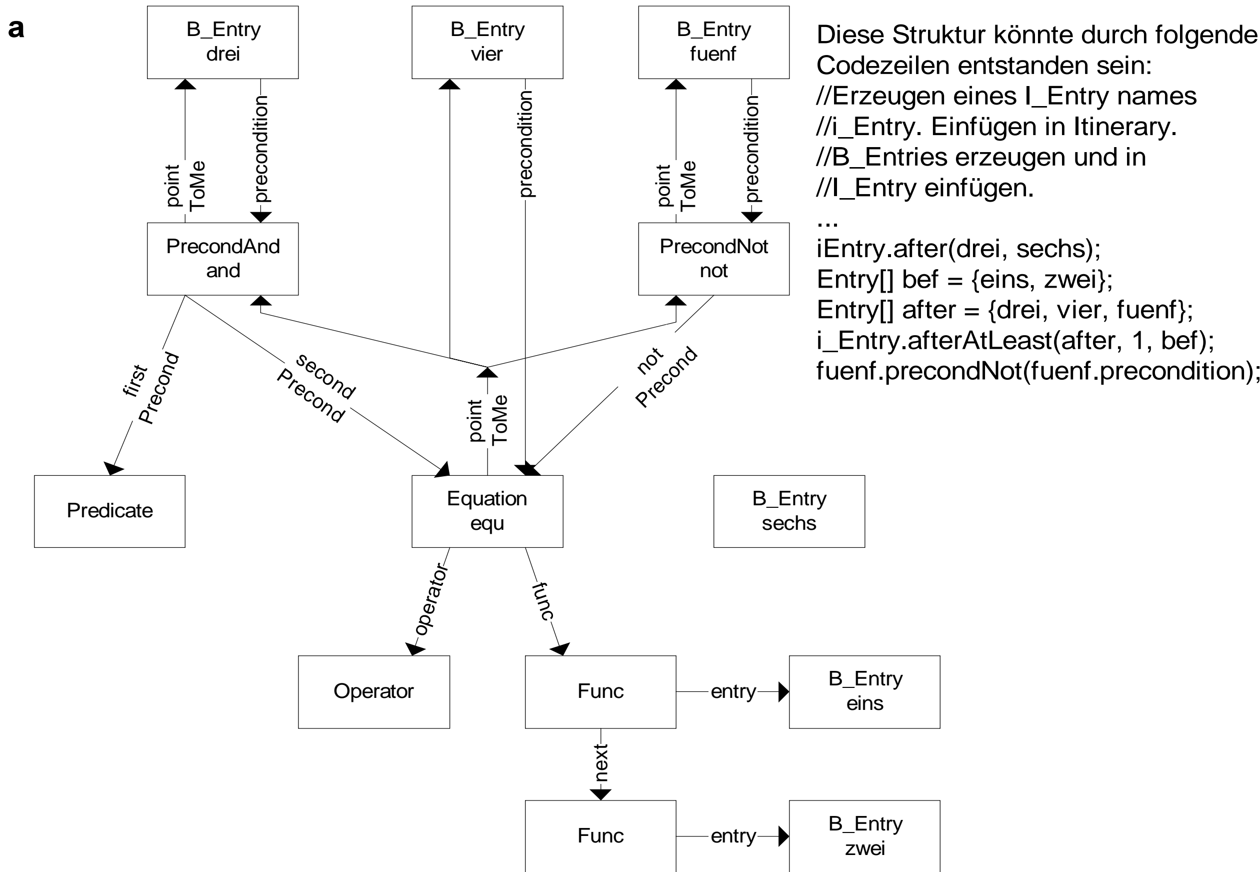
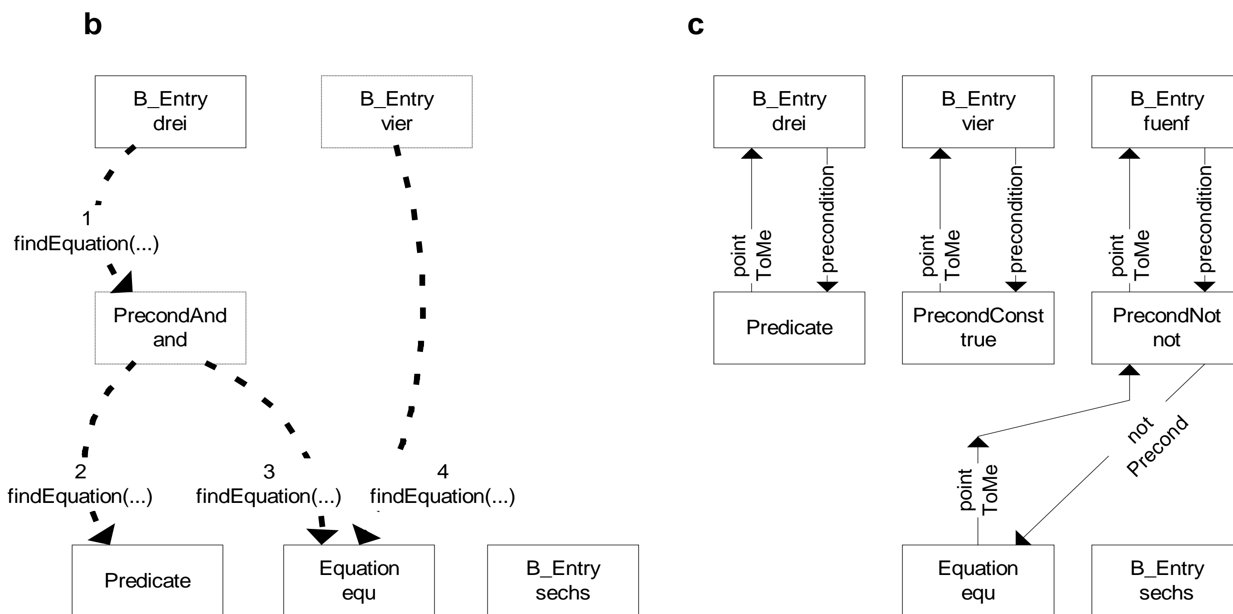


Abbildung 4.8: Methode delAtMost(...)

a: Vorhandene Struktur
b: Versendete Nachrichten
c: Struktur nach Abarbeitung von delAtMost





...

Jetzt werden folgende Zeilen ausgeführt:
Entry[] afterNew = {drei, vier};
i_Entry.delAfterAtLeast(afterNew, 1, bef);

Somit wird das Equation Objekt in der
Precondition von Entry fuenf beibehalten.

Abbildung 4.9: Methode delAfterAtLeast(...)

a: Vorhandene Struktur
b: Versendete Nachrichten
c: Struktur nach Abarbeitung von delAfterAtLeast

4.2.4 Die Klassen Closed/Open_ I_Entry

Abbildung 4.10: Klassen Closed/Open I_Entry

Wegen der verschiedenen Verhaltensweisen der Closed_I_Entry und der Open_I_Entry Klasse wurden diese beiden Klassen erstellt.

4.2.5 Die Klasse Itinerary

Diese Klasse stellt die gesamte Reiseroute dar.

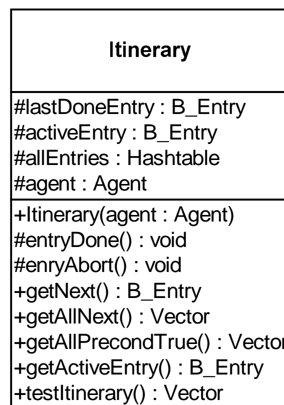


Abbildung 4.11: Klasse Itinerary

Im Attribut `lastDoneEntry` wird das zuletzt ausgeführte B_Entry gehalten. Wozu das benötigt wird, wird im Kapitel 5.3.1 erwähnt.
Das momentan aktivierte B_Entry wird im Attribut `activeEntry` gespeichert.
Um das mehrmalige Einfügen des gleichen Entries zu vermeiden wird die Hashtable `allEntries` benutzt. In dieser Hashtable sollten sich immer alle Entries der Reiseroute befinden. Beim Migrieren des Agenten aber wird dieses Attribut nicht mit transportiert. Somit kann das Attribut auch null werden. Beim Hinzufügen oder Entfernen von Entries wird die Hashtable dann wieder in den korrekten Zustand gebracht. Dies geschieht mit der Methode `Entry.getAbsAllEntries(...)`. Sie liest alle Entries der Itinerary in die Hashtable ein.
Nach dem Hinzufügen eines Entries in ein I_Entry der Reiseroute wird das hinzugefügte Entry in die Hashtable `allEntries` eingetragen. Beim Löschen von Entries werden diese aus der Hashtable entfernt. Somit ist sichergestellt, daß in der Hashtable alle Entries der Itinerary enthalten sind und beim Hinzufügen kann dadurch überprüft werden ob der hinzuzufügende Entry schon existiert.
Ein B_Entry enthält die Information welche Methode des Agenten beim Abarbeiten des B_Entries ausgeführt werden soll. Diese Information wird als String im Attribut toDo gespeichert. Um nun beim Setzen dieses Attributs in einem B_Entry überprüfen zu können ob diese Methode im Agent existiert wird in der Itinerary Klasse das Attribut `agent` aufgenommen.

An dieser Stelle wird noch mal erwähnt welche Kriterien erfüllt sein müssen, so daß ein B_Entry e_1 ausführbar ist:

1. Open-I-Entries, in denen sich e_1 befindet müssen entweder schon im Zustand STARTED sein oder ihre Precondition muß erfüllt sein.
2. Der B-Entry e_1 darf sich nicht außerhalb eines Closed-I-Enty, der den Status STARTED hat, befinden.
3. Die Precondition von e_1 muß erfüllt sein, d.h. sie muß zu TRUE ausgewertet werden.
4. Es darf keinen anderen B-Entry e_2 mit höherer Priorität geben, für den die Kriterien 1-3 ebenfalls erfüllt sind. In diesem Fall wäre e_2 ausführbar falls er Kriterium 4 erfüllt.

Um Informationen darüber zu erhalten welche B_Entries ausgeführt werden können, gibt es mehrere Methoden, die im Folgenden beschrieben werden.
Um nur ein ausführbares B_Entry zu bekommen wird die Methode `getNext()` bereitgestellt. Diese sucht sich den ersten B_Entry dessen Vorbedingung wahr ist und gibt ihn zurück. Wird kein ausführbarer B_Entry gefunden wird eine NoEntryCanRunException ausgelöst. Wie hier genau vorgegangen wird um die korrekte Reihenfolge durchzusetzen wird in Kapitel 5.3.1 genauer beschrieben.
Die Methode `getAllNext()` ist eine Erweiterung der zuletzt genannten Methode. Sie liefert alle ausführbaren B_Entries in einem Vector zurück.
Im Gegensatz zu `getNext()` und `getAllNext()` liefert die Methode `getAllPrecondTrue()` alle B_Entries, die die oben erwähnten Kriterien 1-3 erfüllen. Die Prioritäten werden also bei dieser Methode nicht beachtet. Diese Methode wird unter anderem beim Implementieren eines Protokolls zur fehlertoleranten Ausführung von Agenten benötigt.
Als letzte wichtige Methode ist `testItinerary()` zu nennen. Sie geht vom jetzigen Zustand aus und überprüft welche B_Entries wegen wahrscheinlich falsch definierter Vorbedingungen nicht ausgeführt werden können. Sie liefert alle nicht ausführbaren B_Entries in einem Vector zurück.
Genaueres zur Methode `testItinerary()` wird im Kapitel 5.3.2 folgen.

4.3 Precondition Klassen

Folgende Abbildung zeigt die Vererbungshierarchie aller Klassen, die beim Erstellen einer Vorbedingung verwendet werden können.

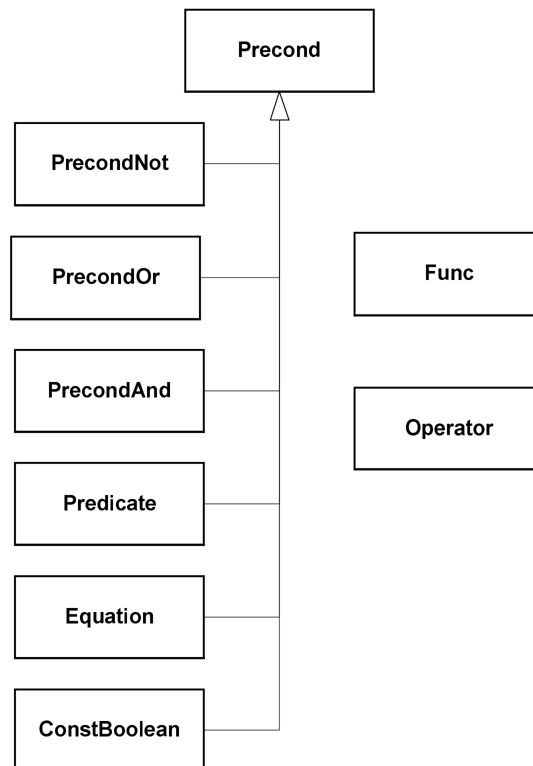


Abbildung 4.12: Precondition Klassen

4.3.1 Die Klasse Precond

«abstract» **Precond**
#countEvaluate : int = -1 #result : boolean = false #pointToMe : Vector = null
#evaluate(countEvaluate : int) : boolean #addPointToMe(add : Object) : void #addPointToMe(add : Vector) : void #delPointToMe(del : Object) : void #getEntries(entries : Vector) : void #entriesInPrecondOk(entry : I_Entry) : boolean #getI_Entry() : I_Entry #del() : void #delPrecond(del : Precond) : void #replace(set : Precond) : void #replacePrecond(replace : Precond, set : Precond) : void #delPredicate(caller : Object, method : int, entry : Entry) : boolean #findEquation(caller : Object, i : int, operator : Operator, entries : Entry[], method : int, pointTo : Vector) : Equation #findEquation(i : int, operator : Operator, entries : Entry[], method : int) : Equation +and(first : Precond, second : Precond) : Precond +or(first : Precond, second : Precond) : Precond +not(not : Precond) : Precond

Abbildung 4.13: Klasse Precond

Diese Klasse ist die abstrakte Basisklasse für alle Klassen, die eine Vorbedingung darstellen.
Das Attribut `countEvaluate` wird verwendet um das mehrfache Auswerten einer Precondition zu vermeiden. Wie dies genau gemacht wird, wird im Kapitel 5.2.2 erklärt.
In der Eigenschaft `result` wird das Ergebnis der Precondition beim Auswerten gespeichert.
Im Vector `pointToMe` werden alle Objekte eingetragen, die dieses Precondition Objekt benutzen. Hierbei kann es sich um Objekte vom Typ Entry oder Precond handeln. Dieser Vector wird benutzt um beim Verändern der Precondition diese Objekte benachrichtigen zu können.

In diesem Beispiel wird für das PrecondAnd Objekt and die Methode getEntries(...) aufgerufen.
Das Bild zeigt die daraufhin versendeten Nachrichten.

Letztendlich kommt der Vector {drei, eins, zwei} zurück.

Abbildung 4.14: Funktionsweise von getEntries(entries: Vector)

Protected Methoden:

Die Methode `evaluate(countEvaluate: int)` wertet die Precondition aus und liefert das Ergebnis als booleschen Wert zurück. Der Parameter `countEvaluate` wird übergeben und mit dem gleichnamigen Attribut des Objekts verglichen um zu ermitteln ob die Precondition neu ausgewertet werden muß. Dies wird im Kapitel 5.2.2 verdeutlicht.

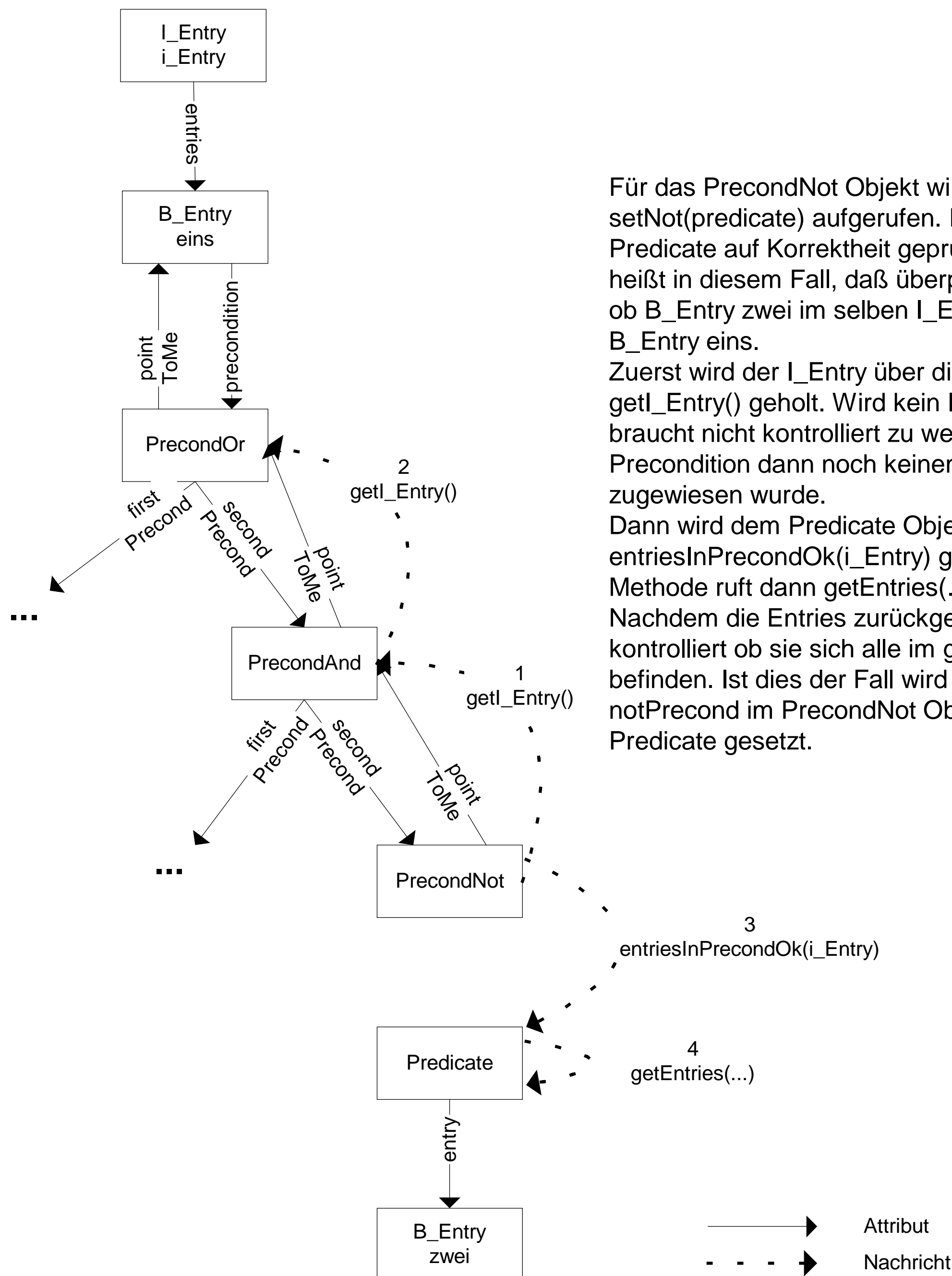
Objekte, die auf das Precond Objekt verweisen, müssen dies mit der Nachricht `addPointToMe(add: Object)` oder `addPointToMe(add: Vector)` bekanntgeben. Diese Methoden sorgen dafür, daß die übergebenen Objekte im Vector `pointToMe` eingetragen werden.
Mit `delPointToMe(del: Object)` kann sich ein Objekt wieder aus dem `pointToMe` Vector entfernen lassen. Dies geschieht zum Beispiel beim Umsetzen der Precondition eines Entries mit der Methode `setPrecond(precond: Precond)`. Der Entry muß der alten Precondition die Nachricht `delPointToMe(this)` und der neu zu setzenden Precondition die Nachricht `addPointToMe(this)` schicken.

Beim Auswerten von Precondition Objekten werden unter anderem die Methoden `D()`, `S()`, `d_int()` und `s_int()` von verschiedenen Entries aufgerufen. Um alle Entries zu erhalten, die so mit einer Precondition verbunden sind wird die Methode `getEntries(entries: Vector)` benutzt. Diese Methode wird rekursiv für alle miteinander verbundenen Precondition Objekte aufgerufen. Immer wenn die ausführende Precondition auf ein Entry verweist trägt sie dieses im Vector `entries` ein. Der Nachrichtenfluß und der sich ergebende Vector `entries` werden in Abbildung 4.14 an Hand einer beispielhaften Precondition dargestellt.

Eine Precondition darf einem Entry nur zugewiesen werden, wenn sich der Entry bereits in einem I_Entry befindet. Somit kann getestet werden ob auch alle Entries, auf die die Precondition verweist, direkt in diesem I_Entry liegen. Für diesen Test wird die Methode `entriesInPrecondOk(entry: I_Entry)` benutzt.

Wird ein Teil einer Precondition geändert muß für diesen veränderten Teil überprüft werden ob er korrekt ist. Dies wird ebenfalls mit der Methode `entriesInPrecondOk(...)` erledigt. Zuerst muß aber das entsprechende I_Entry Objekt gefunden werden. Diese Aufgabe übernimmt die Methode `getI_Entry()`. Dazu wird rekursiv von Precondition Objekt zu Precondition Objekt nach oben gewandert bis der Entry gefunden wird, dem die gesamte Precondition zugeordnet ist. Bei der I_Entry im Attribut `belongsTo` dieser Entry handelt es sich um die gesuchte I_Entry. Daraufhin wird mit `entriesInPrecondOk(...)` überprüft ob die Entries alle im gefundenen I_Entry liegen.
Durch diese Vorgehensweise ist sichergestellt, daß beim Verändern der Preconditions keine Fehler entstehen. Ein Beispiel für den Nachrichtenfluß bei Veränderung einer Precondition zeigt Abbildung 4.15.

Wenn ein Precond Objekt gelöscht werden soll, so wird die Methode `del()` benutzt. Sie durchläuft den Vektor `pointToMe` und sendet jedem Objekt darin eine Nachricht. Handelt es sich dabei um ein Entry wird ihm die Nachricht `delPrecond()` geschickt. Im anderen Fall, wenn es sich um ein Precond Objekt handelt wird die Nachricht `delPrecond(this)` geschickt.



Für das PrecondNot Objekt wird die Methode setNot(predicate) aufgerufen. Dadurch muß das Predicate auf Korrektheit geprüft werden. Das heißt in diesem Fall, daß überprüft werden muß ob B_Entry zwei im selben I_Entry liegt wie B_Entry eins.
Zuerst wird der I_Entry über die Methode getI_Entry() geholt. Wird kein I_Entry gefunden braucht nicht kontrolliert zu werden, weil die Precondition dann noch keinem Entry zugewiesen wurde.
Dann wird dem Predicate Objekt die Nachricht entriesInPrecondOk(i_Entry) geschickt. Diese Methode ruft dann getEntries(...) auf.
Nachdem die Entries zurückgeliefert wurden wird kontrolliert ob sie sich alle im gefundenen I_Entry befinden. Ist dies der Fall wird das Attribut notPrecond im PrecondNot Objekt auf das Predicate gesetzt.

Abbildung 4.15: Verändern einer Precondition

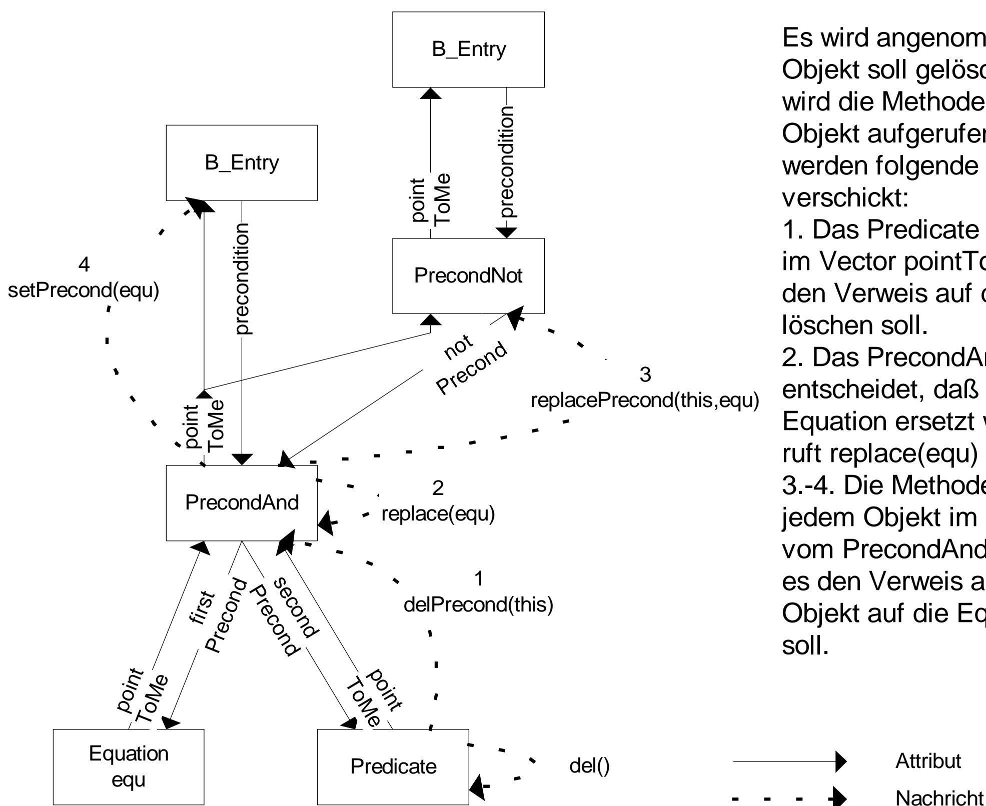
Mit der Methode `delPrecond(del: Precond)` wird dem Precond Objekt mitgeteilt, daß es den Verweis auf die Precondition `del` löschen soll. Die Methode sorgt für das Löschen der Verbindung. Danach teilt die Methode allen Objekte, die auf das Precond Objekt zeigen mit, daß sich die Precondition verändert. Objekte verschiedener Klassen werden hier jeweils anders reagieren.

Um einem Precond Objekt mitzuteilen, daß es sich durch ein anderes Precond Objekt ersetzen soll wird die Nachricht `replace(set: Precond)` versendet. Diese Methode sorgt zuerst dafür, daß alle Objekte, die auf das Precond Objekt zeigen ihre Verweise auf die Precondition `set` umsetzen. Wenn das Precond Objekt selbst auf

andere Precondition Objekte verweist, sollten diese Verweise vor dem Aufruf von `replace(...)` schon entfernt werden.

Durch die Methode `replacePrecond(replace: Precond, set: Precond)` wird einem Precond Objekt mitgeteilt, daß es den Verweis auf die Precondition `replace` auf die Precondition `set` umhängen soll. Diese Methode wird in der oben erwähnten Methode `replace(...)` verwendet.

In Abbildung 4.16 wird das Verhalten eines PrecondAnd Objektes und die Verwendung der eben kurz beschriebenen Methoden gezeigt.



Es wird angenommen das Predicate Objekt soll gelöscht werden. Also wird die Methode del() für dieses Objekt aufgerufen. Daraufhin werden folgende Nachrichten verschickt:
1. Das Predicate teilt jedem Objekt im Vector pointToMe mit, daß es den Verweis auf das Predicate löschen soll.
2. Das PrecondAnd Objekt entscheidet, daß es durch die Equation ersetzt werden soll und ruft replace(equ) auf.
3.-4. Die Methode replace(...) teilt jedem Objekt im Vektor pointToMe vom PrecondAnd Objekt mit, daß es den Verweis auf das PrecondAnd Objekt auf die Equation umsetzen soll.

Abbildung 4.16: Nachrichtenfluß bei Precond.del()

Mit der Methode `delPredicate(caller: Object; method: int, entry: Entry)` wird gezielt ein Predicate Objekt in der Precondition gesucht und aus der Precondition entfernt. Das zu suchende Predicate Objekt wird über die Parameter `method` und `entry` genau spezifiziert. Die Methode läuft rekursiv durch die Precondition. Wird das gesuchte Precondition Objekt gefunden, wird dem Objekt im Parameter `caller` eine Nachricht geschickt, daß es die Verbindung zum Predicate löschen soll. Im Parameter `caller` befindet sich immer das Objekt, das auf das im Moment die Methode `delPredicate(...)` ausführende Precondition Objekt verweist.
Man könnte auch allen Objekten im Vector `pointToMe` eine Nachricht schicken, daß das Predicate gelöscht werden soll. Dann würde das Predicate aber aus allen Precondition Objekten gelöscht werden. Mit dem Ansatz der hier gewählt wurde kann das Predicate gezielt aus einer Precondition entfernt werden.

Wenn ein bestimmtes Equation Objekt gefunden werden soll steht die Methode `findEquation(...)` in folgenden Varianten zur Verfügung:

- `findEquation(caller: Object, i: int, operator: Operator, entries: Entry[], method: int, pointTo: Vector): Equation`
- `findEquation(i: int, operator: Operator, entries: Entry[], method: int): Equation`

Diese Methode sucht rekursiv alle untergeordneten Precondition Objekte nach der gesuchten Equation durch. Wenn ein Equation Objekt gefunden wird so wird es mit dem gesuchten verglichen. Bei Erfolg wird das Objekt zurückgeliefert ansonsten wird weitergesucht.
Die Variante `findEquation(caller: Object,..., pointTo: Vector): Equation` macht noch mehr als nur das Equation Objekt zu suchen. Das Objekt, das die Nachricht weitergibt trägt sich im Parameter `caller` ein. Wenn dann ein Equation Objekt feststellt, daß es das Gesuchte ist, trägt es das im Parameter `caller` eingetragene Objekt in den Vector `pointToMe` ein. Somit ist gewährleistet , daß alle Objekte, die auf die Equation verweisen im Vector `pointToMe` stehen.

Public Methoden:

Die Klasse Precond enthält drei statische Methoden zur Erzeugung von Preocondition Objekten:

- `and(first: Precond, second: Precond): Precond`
- `or(first: Precond, second: Precond): Precond`
- `not(not: Precond): Precond`

Die Methode `and(...)` erzeugt eine neue Precondition indem es die Precondition Objekte `first` und `second` mit dem booleschen Operator „and“ verbindet.
Die beiden anderen Methoden machen Vergleichbares nur mit den Operatoren „or“ und „not“.

4.3.2 Die Klasse Predicate

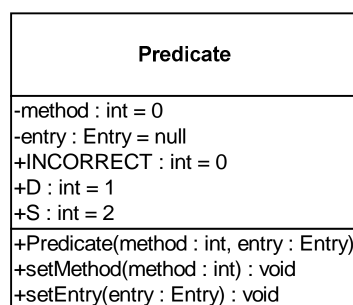


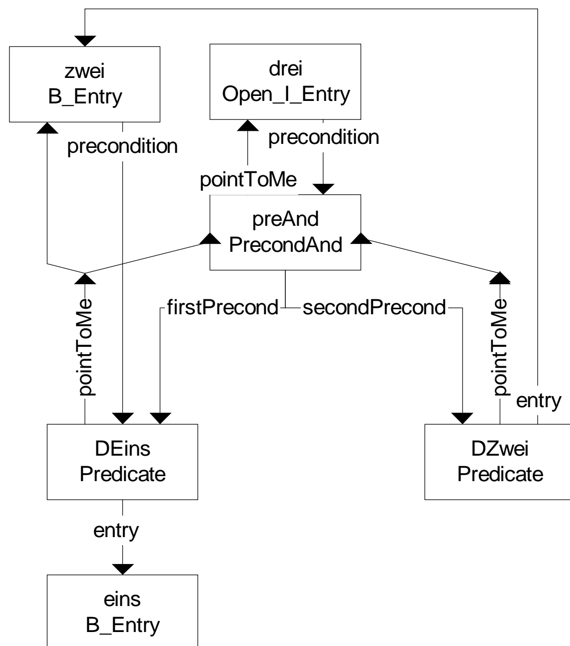
Abbildung 4.17: Klasse Predicate

Durch Objekte dieser Klasse läßt sich festlegen, daß ein Entry erst nach Ausführen oder Starten eines Anderen an der Reihe ist.

Durch das Attribut `method` wird festgelegt welche Methode (`D()` oder `S()`) im enthaltenen Entry (`entry`) ausgeführt wird (s. Abschnitt 4.2.1).

Mit den Methoden `setMethod(method: int)` und `setEntry(entry: Entry)` werden die eben erwähnten Attribute gesetzt.

Die Abbildung 4.18 zeigt die Verwendung dieser Klasse und die daraus resultierende Vorbedingung. Dabei wird auch noch eine Instanz der Klasse PrecondAnd verwendet, die erst weiter unten erwähnt wird.



<u>Bsp:</u> Entry zwei soll erst nach Entry eins ausgeführt werden. Und Entry drei ist erst nach Entry eins und nach Entry zwei an der Reihe.

```
//Code aus Abbildung 4.5

Predicate DEins = new Predicate(Predicate.D, eins);
Predicate DZwei = new Predicate(Predicate.D, zwei);
Precond preAnd = Precond.and(DEins, DZwei);

zwei.setPrecond(DEins);
drei.setPrecond(preAnd);
```

<u>Einfacher über I_Entry Methoden:</u>

```
itinerary.before(eins, zwei);
itinerary.before(eins, drei);
itinerary.before(zwei, drei);
```

Abbildung 4.18: Precondition mit Predicate und PrecondAnd

4.3.3 Die Klasse Func

Diese Klasse ist nicht abgeleitet von der Precond Klasse, weil sie keine eigenständige Vorbedingung darstellt. Objekte dieser Klasse sind nur im Zusammenhang mit Instanzen der Klasse Equation sinnvoll.

Func
#method : int = 0 -entry : Entry = null -next : Func = null +INCORRECT : int = 0 +d : int = 1 +s : int = 2
+Func(method : int, entry : Entry) +Func(method : int, entry : Entry, next : Func) #evaluate() : int #getEntries(entries : Vector) : void +setMethod(method : int) : void +setEntry(entry : Entry) : void +setNext(next : Func) : void

Abbildung 4.19: Klasse Func

Diese Klasse ist der Predicate Klasse sehr ähnlich. Sie enthält als Attribut ebenfalls ein Entry (`entry`) nur können hier die Methoden `d_int()` oder `s_int()` als `method` gesetzt werden. Wenn mehrere Func Objekte hintereinander gehängt werden sollen, so werden sie über das Attribut `next` verbunden. Das letzte Func Objekt einer solchen Reihe wird daran erkannt, daß `next` gleich null ist.

Die Methode `evaluate()` ruft die in `method` eingetragene Methode für den im Attribut `entry` stehenden Entry auf. Da es sich dabei nur um die Methoden `d_int()` und `s_int()` handeln kann, ist das Ergebnis ein Integerwert. Wenn mehrere Func Objekte hintereinander gereiht sind so durchläuft `evaluate()` rekursiv alle in dieser Reihe und addiert die Ergebnisse. Durch das Bilden solcher Reihen werden also Summen erzeugt.
Um alle Entries zu erhalten, auf die in einer solchen Reihe von Func Objekten verwiesen wird, wird die Methode `getEntries(entries: Vector)` benutzt.

Die Methoden `setMethod(method: int)`, `setEntry(entry: Entry)` und `setNext(next: Func)` sind zum Setzen der Attribute `method`, `entry` und `next`.

4.3.4 Die Klasse Operator

Diese Klasse stellt einen Vergleichsoperator aus der Menge {<, <=, =, >=, >} dar. Sie ist wie die Func Klasse nicht von Precond abgeleitet, da auch sie keine vollständige Vorbedingung ist. Sie wird ebenfalls mit Objekten der Klasse Equation verwendet.

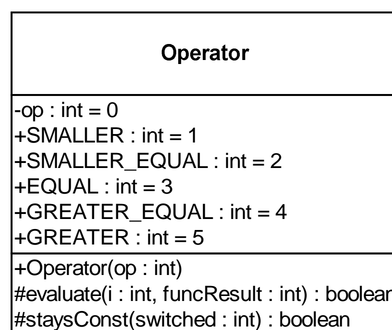


Abbildung 4.20: Klasse Operator

Das Attribut `op` kann einen der Werte aus {`SMALLER, SMALLER_EQUAL, EQUAL, GREATER_EQUAL, GREATER`} annehmen.
Die Methode `evaluate(i: int, funcResult: int)` vergleicht zwei übergebene Integerwerte mit dem gesetzten Operator und liefert das Ergebnis des Vergleichs zurück.
Über die Methode `staysConst(switched: int)` wird ermittelt ob sich das Ergebnis des Vergleichs im Folgenden noch einmal ändern kann. Dabei gibt der Parameter `switched` an wie oft sich das Ergebnis des Vergleichs schon geändert hat. Diese Funktionalität wird benutzt um die Precondition Struktur so klein wie möglich zu halten. Weiteres dazu findet sich im Kapitel 5.2.2.

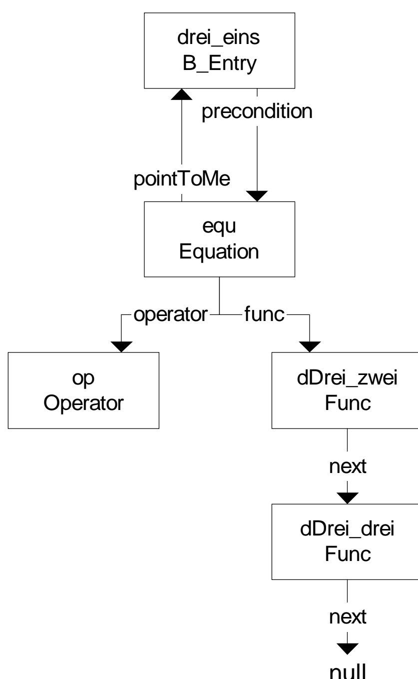
4.3.5 Die Klasse Equation

Equation
-i : int = 0 -operator : Operator = null -switched : int = 0 -func : Func = null
+Equation(i : int, operator : Operator, func : Func) +setI(i : int) : void +setOperator(operator : Operator) : void +setFunc(func : Func) : void

Abbildung 4.21: Klasse Equation

Die Klasse Equation besteht aus einem Integer `i`, einem Operator und einem Func Objekt. Als weiteres Attribut (`switched`) wird festgehalten wie oft sich das Ergebnis der Equation bereits geändert hat.

Beim Ausführen der Methode `evaluate(countEvaluate: int)` wird zuerst die Methode `evaluate()` für das Attribut `func` aufgerufen. Dann wird für den Operator die Methode `evaluate(...)` mit dem Attribut `i` und dem Ergebnis des Func Objekts als Parameter aufgerufen. Das Ergebnis wird als boolescher Wert zurückgeliefert. Ein Beispiel für ein Equation Objekt und die sich ergebende Struktur werden in Abbildung 4.22 gezeigt.



<u>Bsp:</u> Das Entry drei_eins soll erst ausgeführt werden, wenn ein Entry aus der Menge {drei_zwei, drei_drei} bereits ausgeführt wurde.

```
//Code aus Abbildung 4.5

Func dDrei_drei = new Func(Func.d, drei_drei);
Func dDrei_zwei = new Func(Func.d, drei_zwei, dDrei_drei);
Operator op = new Operator(Operator.SMALLER_EQUAL);
Equation equ = new Equation(1, op, dDrei_zwei);

drei_eins.setPrecond(equ);
```

<u>Einfacher über I_Entry Methoden:</u>

```
Entry[] after = {drei_eins};
Entry[] bef = {drei_zwei, drei_drei};
drei.afterAtLeast(after, 1, bef);
```

Abbildung 4.22: Verwendung von Equation und die sich ergebende Struktur

4.3.6 Weitere Klassen zur Beschreibung von Vorbedingungen

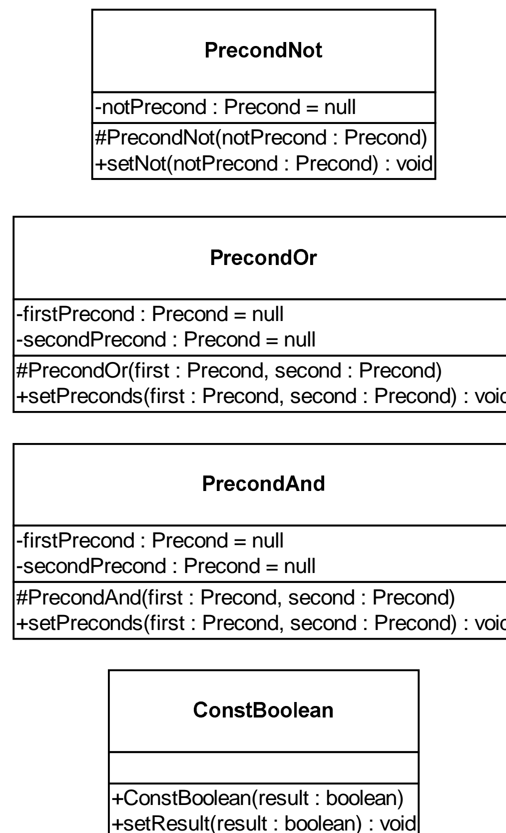


Abbildung 4.23: Weitere Precondition Klassen

PrecondNot
Die Klasse PrecondNot dient lediglich dazu vorhandene Preconditions zu negieren. Die zu negierende Precondition wird im Attribut `notPrecond` gespeichert.
Der Konstruktor dieser Klasse sowie der beiden Klassen PrecondOr und PrecondAnd sind auf „protected“ gesetzt, weil es in der Precond als auch in der Entry Klasse Methoden zum automatischen Erzeugen von Instanzen dieser Klassen gibt.

PrecondOr / PrecondAnd
Diese Klasse verbindet zwei bereits bestehende Preconditions (`firstPrecond`, `secondPrecond`) zu einer neuen Precondition, in dem die vorhandenen mit dem booleschen Operator „oder“/ „und“ verknüpft werden.

ConstBoolean
Diese Klasse wird verwendet um eine Precondition mit einem konstanten booleschen Wert zu erzeugen. Entries, denen nicht explizit eine Precondition zugewiesen wird erhalten automatisch eine Precondition, die sich immer zu „wahr“ auswertet.

An einem Beispiel wird kurz die Verwendung von Objekten der zuletzt erwähnten Klassen erläutert.

Beispiel: Wir nehmen an wir haben ein I_Entry Objekt names i_Entry. In diesem I_Entry befinden sich die Entries eins, zwei, drei und vier.

Entry zwei soll nach eins ausgeführt werden. Dies wird durch folgende Zeile Code erreicht:

```
i_Entry.after(zwei, eins);
```

Entry drei soll nicht nach eins ausgeführt werden. Die Precondition von drei ist also einfach die negierte Precondition von Entry zwei.

```
drei.setPrecond(Precond.not(zwei.getPrecond()));
```

Der Entry vier soll nach eins und nach drei ausgeführt werden.

```
i_Entry.after(vier, drei);
vier.setPrecond(Precond.and(zwei.getPrecond(),
                             vier.getPrecond()));
```

Anstatt der letzten Zeile könnte auch einfacher folgendes benutzt werden:

```
i_Entry.after(vier, eins);
```

Dadurch würde allerdings ein Predicate Objekt mehr erzeugt werden.

Beim Erzeugen eines Entries wird die Precondition automatisch auf ein PrecondConst Objekt, das sich immer zu TRUE auswertet gesetzt. Somit ist die Precondition von eins, da diese nicht verändert wurde immer noch dieses PrecondConst Objekt.

Die hieraus resultierende Struktur wird in Abbildung 4.24 gezeigt.

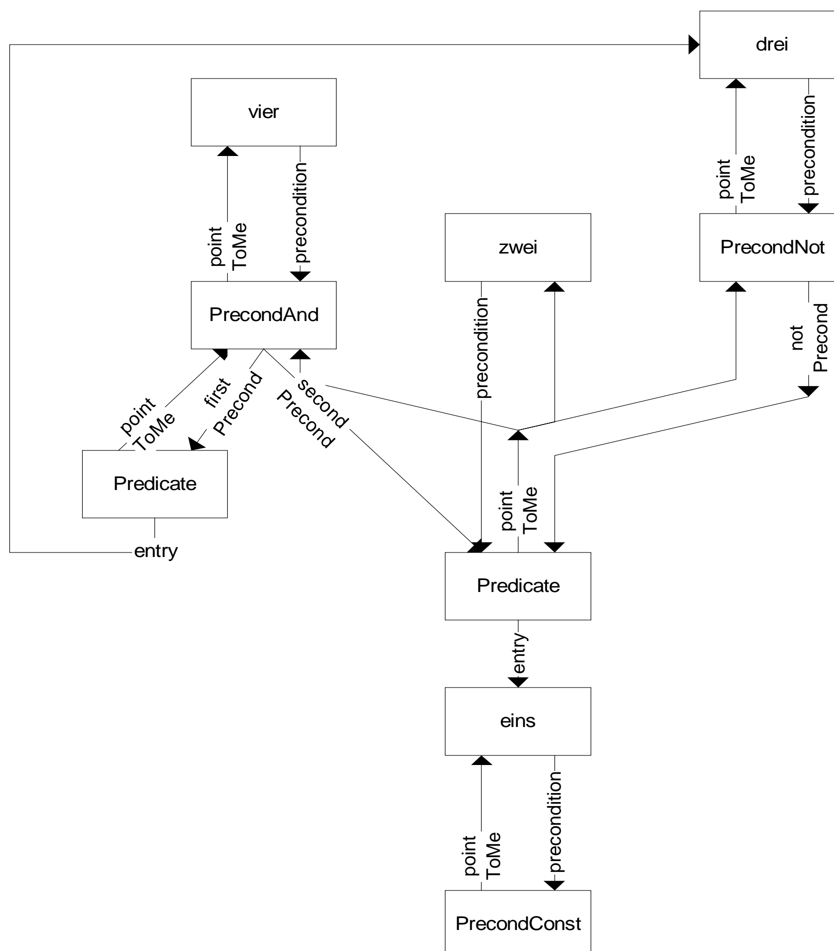


Abbildung 4.24: Klassen PrecondNot, PrecondAnd und PrecondConst

Am Schluß dieses Kapitels soll an einem Beispiel der Aufbau einer umfangreicheren Reiseroute erklärt werden.

4.4 Erstellen einer kompletten Reiseroute

Eine Firma plant einen Besuch auf der Systems in München. Dieser soll mit einem Betriebsausflug verbunden werden. Am Samstag wird mit dem Bus Richtung München losgefahren. Um die Mittagszeit soll entweder eine Besichtigung eines Schlosses oder aber eine Floßfahrt stattfinden. Danach wird je nach gewähltem Programm, Besichtigung oder Floßfahrt, eine Gaststätte aufgesucht und zu Mittag gegessen. Am Abend soll wie jedes Jahr im Hotel Adler übernachtet werden. Sonntags wird dann die Systems besucht und gegen Abend wieder nach Hause gefahren. Im folgenden wird an Hand von Pseudo-Code gezeigt wie eine Reiseroute für einen Agenten aufgestellt werden kann, der die notwendigen Vorbereitungen für den Ausflug treffen soll.

Zunächst wird ein Agent erzeugt, der dann später die Buchungen erledigen soll. Er muß Methoden besitzen, welche die benötigten Aktionen auf den zu besuchenden Rechnern erledigen werden. Dann wird eine Reiseroute erzeugt.

```
AgentBuchen agentAusflug = new AgentBuchen();

Itinerary reiseroute = new Itinerary(agentAusflug);
```

Um Karten für den Besuch der Messe kaufen zu können, muß zuerst ein Scheck bei einer Bank geholt werden. Da die Firma bei zwei Banken ein Konto besitzt kann zwischen diesen beiden gewählt werden. Also werden zu der Reiseroute zwei B_Entries names bank1 und bank2 hinzugefügt. Um die Struktur etwas übersichtlich zu gestalten, werden diese B_Entries in ein Open_I_Entry namens scheck eingefügt. Dazu muß zuerst das Open_I_Entry erzeugt und zur Reiseroute hinzugefügt werden. Erst dann können die beiden B_Entries in den Open_I_Entry eingefügt werden.

```
Open_I_Entry scheck = new Open_I_Entry();
reiseroute.add(scheck);
B_Entry bank1 = new B_Entry(holeScheck, Kreissparkasse);
B_Entry bank2 = new B_Entry(holeScheck, Volksbank);
scheck.add(bank1);
scheck.add(bank2);
```

Es muß ausgeschlossen werden, daß bei beiden Banken ein Scheck geholt wird. Somit wird in beiden Vorbedingungen aufgenommen, daß das B_Entry nur ausgeführt werden kann, wenn das andere noch nicht ausgeführt wurde. Oder einfacher ausgedrückt darf maximal einer der beiden B_Entries ausgeführt werden. Weiterhin wird die Kreissparkasse der Volksbank vorgezogen.

```
Entry[] banken = {bank1, bank2};
scheck.atMost(1, banken);

scheck.prior(bank1, bank2);
```

Nach dem der Scheck auf einer der Banken geholt wurde kann der Kauf der Karten ausgeführt werden. Dies wird durch den B_Entry karten erledigt. Dazu wird eine Precondition erzeugt, die dafür sorgt, daß karten erst nach erledigen der Open_I_Entry scheck ausgeführt werden darf.

```
Precond nachScheck = new Predicate(Predicate.D, scheck);
B_Entry karten = new B_Entry(kaufeKarten, Systems);
karten.setPrecond(nachScheck);
reiseroute.add(karten);
```

Das Programm am Samstag ist entweder eine Besichtigung oder eine Floßfahrt. Je nach dem was letztendlich gebucht wird, werden Tische in einer nahe gelegenen Gaststätte reserviert, um dort das Mittagessen einzunehmen. Um die Verwendung von Closed_I_Entries zu zeigen werden die B_Entries, die diese Aufgaben erledigen in zwei Closed_I_Entries programm1 und programm2 gepackt. Dabei wird dafür gesorgt, daß zuerst versucht wird die Veranstaltung zu buchen bevor die Tische reserviert werden.

```
Closed_I_Entry programm1 = new Closed_I_Entry();
reiseroute.add(programm1);
B_Entry unternehmen1 = new B_Entry(reserviereBesichtigung,
                                  Reisebüro);
B_Entry essen1 = new B_Entry(reserviereTische, Gasthof
                                  Rössle);
programm1.add(unternehmen1);
programm1.add(essen1);
programm1.before(unternehmen1, essen1);

Closed_I_Entry programm2 = new Closed_I_Entry();
reiseroute.add(programm2);
B_Entry unternehmen2 = new B_Entry(reserviereFloßfahrt,
                                  Reisebüro);
B_Entry essen2 = new B_Entry(reserviereTische, Gasthof
                             Hirschen);
programm2.add(unternehmen2);
programm2.add(essen2);
programm2.before(unternehmen2, essen2);
```

Da bei einer Umfrage unter den Beschäftigten der Firma klar wurde, daß sich mehr Personen für die Besichtigung interessieren wird eine Priorität zwischen programm1 und programm2 definiert, die dafür sorgt, daß programm1 Vorrang hat.

```
reiseroute.prior(programm1, programm2);
```

Das Buchen beider Programme wird durch gegenseitigen Ausschluß der beiden Closed_I_Entries verhindert. Dies geschieht hier mit der Methode atMost.

```
Entry[] programme = {programm1, programm2};
reiseroute.atMost(1, programme);
```

Das Hotel für die Übernachtung steht schon fest, also wird einfach ein B_Entry uebernachten eingefügt, der die Reservierung der Zimmer erledigen soll. Da dies

ohne Beachten von Vorbedingungen geschehen kann muß die Precondition dieses B_Entries ein PrecondConst sein, welches immer zu TRUE ausgewertet wird. Da beim Erstellen eines Entries das Attribut `precondition` automatisch auf ein PrecondConst Objekt dieser Art gesetzt wird, muß an dieser Stelle keine Veränderung der Precondition erfolgen.

```
B_Entry uebernachten = new B_Entry(reserviereZimmer, Hotel
                                       Adler);
reiseroute.add(uebernachten);
```

Ebenfalls unabhängig kann das Buchen des Busses vor sich gehen. Dabei kommen drei verschiedene Busunternehmen in Frage. Also werden drei B_Entries eines für jedes der drei Unternehmen erzeugt. Wegen der Übersichtlichkeit wird wieder eine Open_I_Entry busBuchen in die Reiseroute eingefügt. In diese werden dann die B_Entries gesetzt.

```
Open_I_Entry busBuchen = new Open_I_Entry();
reiseroute.add(busBuchen);
B_Entry bus1 = new B_Entry(bestelleBus, Busunternehmen1);
B_Entry bus2 = new B_Entry(bestelleBus, Busunternehmen2);
B_Entry bus3 = new B_Entry(bestelleBus, Busunternehmen3);
busBuchen.add(bus1);
busBuchen.add(bus2);
busBuchen.add(bus3);
```

Um das Bestellen mehrerer Busse auszuschließen wird wieder mit atMost festgelegt, daß nur eines der B_Entries bus1, bus2 und bus3 ausgeführt werden.

```
Entry[] busse = {bus1, bus2, bus3};
busBuchen.atMost(1, busse);
```

Da man schon Erfahrungen mit den drei Busunternehmen hat und das erste den beiden anderen klar vorzieht werden Prioritäten definiert.

```
Entry[] vorziehen = {bus1};
Entry[] schlechter = {bus2, bus3};
busBuchen.allPriorOthers(vorziehen, schlechter);
```

Um sich am Sonntag auf der Messe zurechtzufinden sollen am Schluß noch Informationen über den Standort verschiedener Stände gesammelt werden. Dabei ist man an den Ständen von vier Firmen interessiert bei diesen man also die Anfragen ausführen wird. Das Sammeln von Informationen soll ganz am Schluß geschehen. Um mit weniger Preconditions auszukommen werden diese B_Entries wieder in ein Open_I_Entry gepackt.

```
Open_I_Entry infos = new Open_I_Entry();
reiseroute.add(infos);

B_Entry stand1 = new B_Entry(holeInformationen, Firma1);
B_Entry stand2 = new B_Entry(holeInformationen, Firma2);
B_Entry stand3 = new B_Entry(holeInformationen, Firma3);
B_Entry stand4 = new B_Entry(holeInformationen, Firma4);
```

```
infos.add(stand1);
infos.add(stand2);
infos.add(stand3);
infos.add(stand4);
```

Um das „zuletzt Ausführen“ des Open_I_Entries zu gewährleisten wird definiert, daß es erst nach scheck, karten, uebernachten und busBuchen drankommt.

```
reiseroute.after(infos, scheck);
reiseroute.after(infos, karten);
reiseroute.after(infos, uebernachten);
reiseroute.after(infos, busBuchen);
```

Man darf aber nicht definieren, daß es nach programm1 und nach programm2 ausgeführt werden soll, weil nie beide Closed_I_Entries abgearbeitet werden. Somit würde das Open_I_Entry infos nie abgearbeitet werden. Also wird definiert, daß die Vorbedingung von infos nach Ausführung mindestens einer der beiden Closed_I_Entries wahr ist. Dies könnte auch umgangen werden in dem man um die beiden Closed_I_Entries noch ein Open_I_Entry einfügen würde.

```
Entry[] vor = {programm1, programm2};
Entry[] after = {infos};
reiseroute.afterAtLeast(after, 1, vor);
```

Nachdem mindestens drei der Firmen nach dem Standort ihres Standes befragt wurden soll der Agent zurückkehren und die Ergebnisse ausgeben. Da die Vorbedingung der zuletzt definierten B_Entries auf diesen B_Entry zugreifen muß er ebenfalls in das Open_I_Entry eingefügt werden.

```
B_Entry home = new B_Entry(zeigeErgebnisse, BenutzerKnoten);
infos.add(home);

Entry[] staende = {stand1, stand2, stand3, stand4};
Entry[] nachInfos = {home};

infos.afterAtLeast(nachInfos, 3, staende);
```

Als letztes wird die Reiseroute in den Agenten gesetzt. Jetzt kann der Agent gestartet werden und sehr wahrscheinlich steht dem Ausflug dann bald nichts mehr im Wege.

```
agentAusflug.setItinerary(reiseroute);
```

5. Implementierung

In diesem Kapitel werden die wichtigsten Algorithmen beim Aufbauen einer Reisroute erklärt. Als erstes wird das Eintragen von Prioritäten und die Besonderheiten dabei betrachtet. Danach werden die Vorbedingungen genauer unter die Lupe genommen. Mit diesen Grundlagen werden dann am Schluß noch die Vorgehensweisen, die im Zusammenhang mit Entries vorkommen erklärt.

5.1 Prioritäten

Erst soll kurz erläutert werden wie eine Prioritätsbeziehung in die Itinerary Struktur eingetragen wird. Um eine Priorität (e_1,e_2) zwischen den Entries e_1 und e_2 auszudrücken, wird wie folgt vorgegangen. Der Entry e_2 hat in diesem Fall eine niedrigere Priorität als Entry e_1. Er wird also im Vector `prioritiesAfter` von Entry e_1 eingetragen. Da Entry e_1 eine höhere Priorität erhält als Entry e_2 wird er im Vector `prioritiesBef` von Entry e_2 eingetragen. Im Attribut `prioritiesAfter` befinden sich also die Entries mit niedrigerer Priorität und in `prioritiesBef` befinden sich Entries mit höherer Priorität. Die Struktur nach eintragen einer Priorität zwischen zwei Entries wird in Abbildung 5.1 gezeigt.

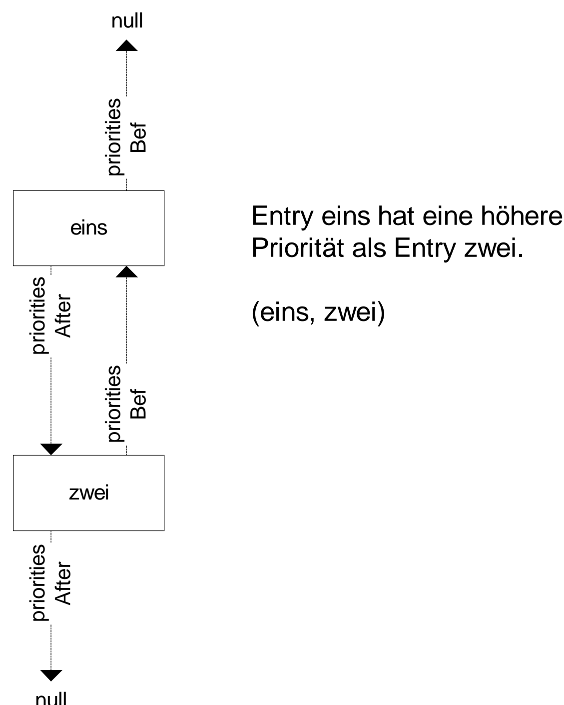


Abbildung 5.1: Prioritätsbeziehung

Bevor eine Prioritätsbeziehung eingetragen wird, wird zuerst überprüft ob sie schon existiert. Ist dies der Fall wird sie nicht noch einmal erzeugt.

Prioritäten dürfen nur zwischen Entries definiert werden, die sich direkt im gleichen I_Entry befinden. Die Methoden, die Prioritätsbeziehungen erstellen befinden sich

alle in der Klasse I_Entry. Somit kann ein I_Entry Objekt vor dem Einfügen einer Prioritätsbeziehung zwischen zwei Entries testen, ob beide Entries im Attribut `entries` der I_Entry enthalten sind. Da im Vector `entries` nur die direkt enthaltenen Entries einer I_Entry eingetragen werden, ist sicher gestellt, daß nur Prioritäten zwischen Entries, die direkt im gleichen I_Entry liegen erzeugt werden. In Abbildung 5.2 ist dargestellt welche Prioritätsbeziehungen nicht möglich sind.

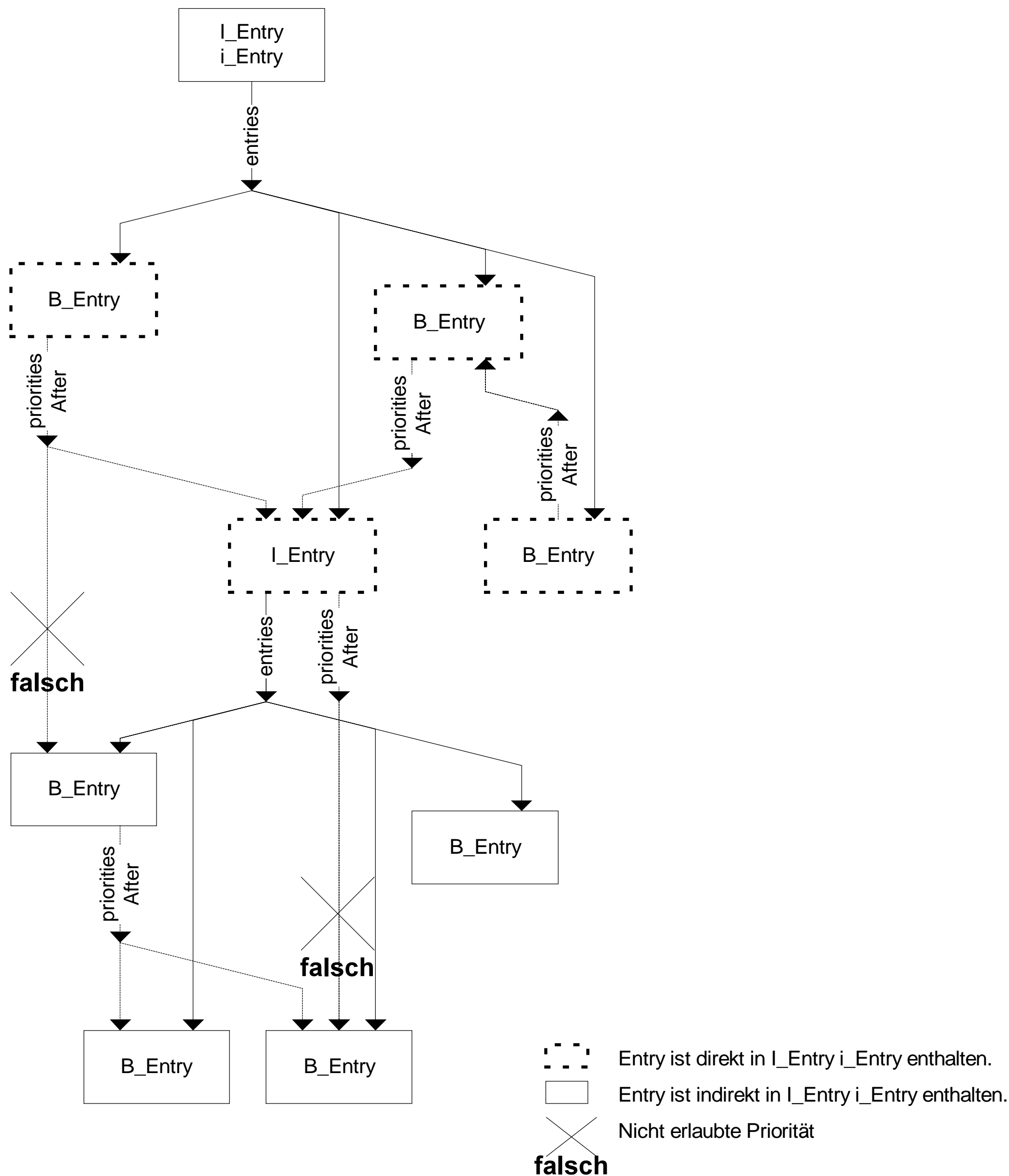


Abbildung 5.2: Nicht erlaubte Prioritäten

Beim Deklarieren von Prioritäten könnten Zyklen entstehen. In Abbildung 5.3 wird dies kurz gezeigt. Dabei werden nur die Vectoren `prioritiesAfter` der Entries dargestellt. Durch die Vectoren `prioritiesBef` würde genau so ein Zyklus entstehen. Somit reicht es also aus einen dieser Vectoren zu betrachten.

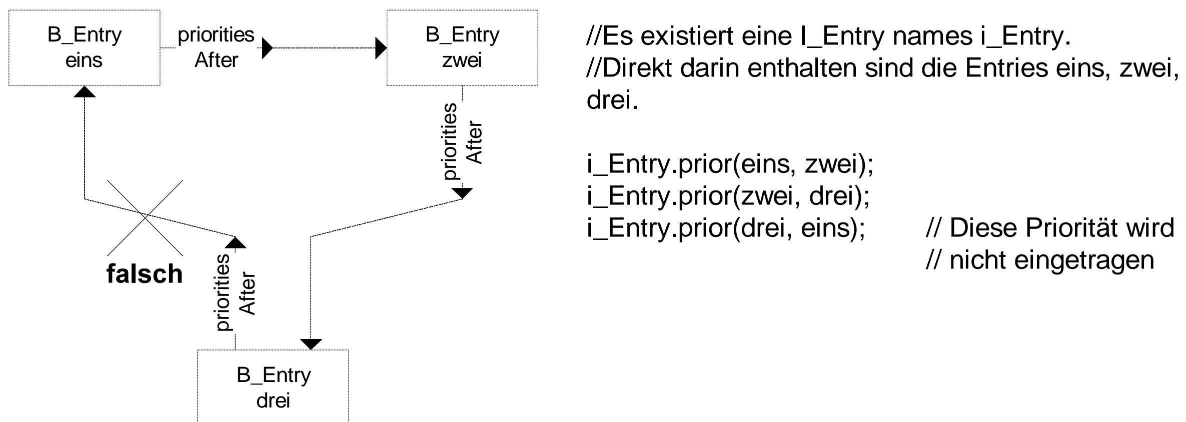


Abbildung 5.3: Zyklus in Prioritätsbeziehungen

Solche Zyklen werden in der Reiseroute verhindert. Bevor eine Prioritätsbeziehung eingetragen wird, wird zuerst kontrolliert ob ein Zyklus entsteht. Wenn ein Zyklus entstehen würde wird die Priorität nicht eingetragen.
Ein Zyklus kann nur durch hinzufügen einer Priorität z.B. über die Methode `I_Entry.prior(bef, after)` entstehen. Dieser entsteht aber nur dann, wenn bereits eine Verbindung zwischen Entry `after` und Entry `bef` über die Attribute `prioritiesAfter` besteht (s. Abbildung 5.4). Also muß der Algorithmus nur prüfen ob ein „Weg" von Entry `after` zu Entry `bef` über die Vectoren `prioritiesAfter` besteht.
Dabei wird wie folgt vorgegangen:
Es werden rekursiv alle Vektoren `prioritiesAfter` von Entry `after` ausgehend nach dem Entry `bef` durchsucht. Wird dabei irgendwann der Entry `bef` gefunden so würde beim Hinzufügen der Priorität (`bef`, `after`) ein Zyklus entstehen. Der Algorithmus merkt sich die besuchten Entries um nicht dieselben Entries mehrmals zu untersuchen. In Abbildung 5.4 könnte es z.B. vorkommen, daß man den Entry sechs mehrfach untersucht. Er ist sowohl im Vector `prioritiesAfter` von Entry fuenf als auch in dem von vier enthalten.

Wenn mehrere Prioritäten auf einmal definiert werden sollen (z.B. durch die Methode `priorInRow(...)`), muß der Algorithmus etwas verändert werden. Jetzt können nicht zuerst alle hinzuzufügenden Prioritäten einzeln geprüft werden, denn es kann vorkommen, daß nur ein Zyklus durch mehrere der hinzuzufügenden Prioritäten entsteht. Dies wird in Abbildung 5.5 verdeutlicht.

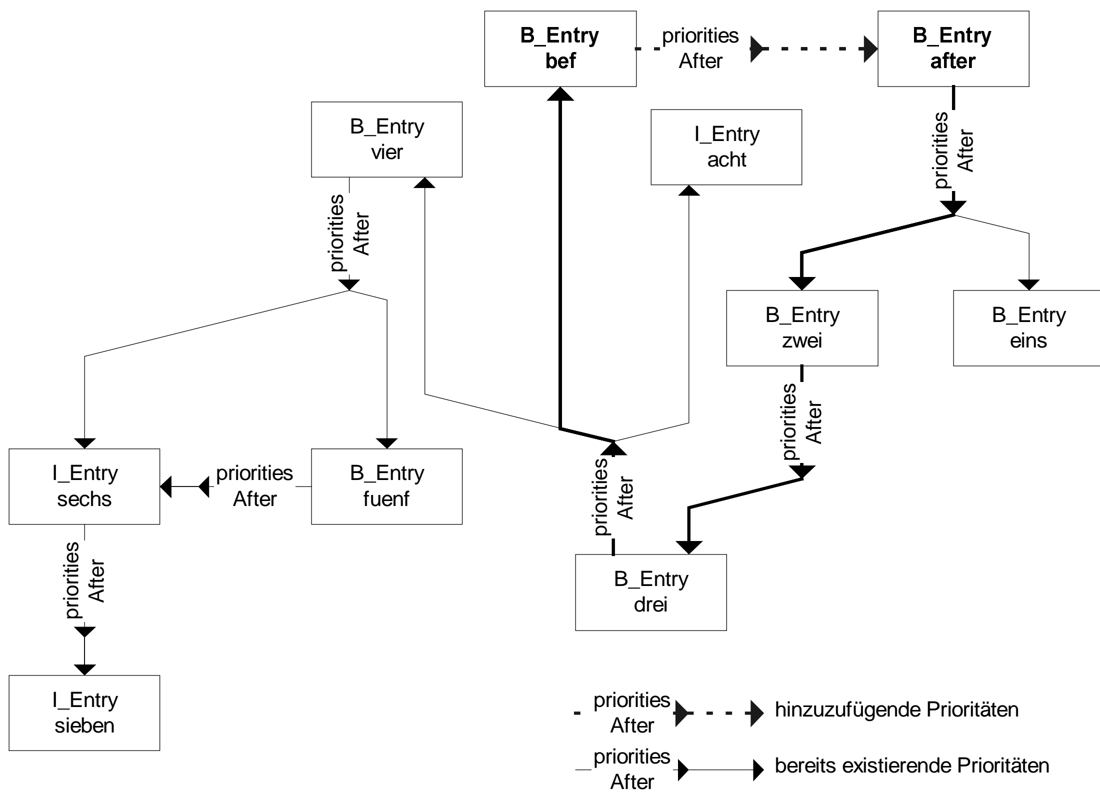


Abbildung 5.4: Erkennen von Zyklen vor dem Einfügen einer Priorität

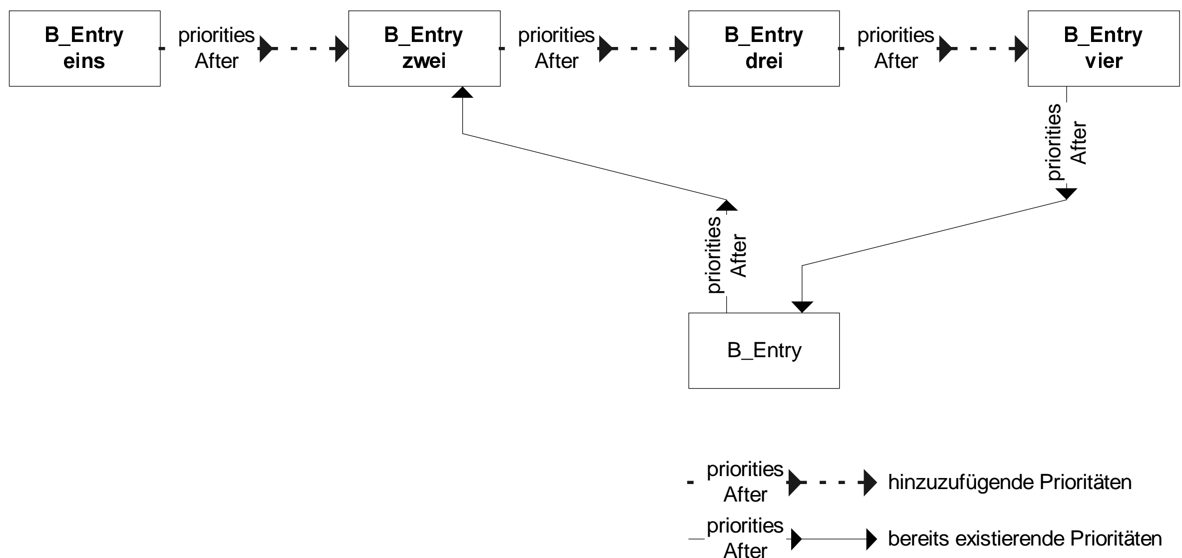


Abbildung 5.5: Zyklus durch mehrere auf einmal hinzugefügte Prioritäten

Wenn für jede hinzuzufügende Priorität einzeln getestet würde ob ein Zyklus entsteht, würde man das entstehen eines solchen nicht bemerken.
Aus diesem Grund wird beim Versuch mehrere Prioritäten einzufügen wie folgt vorgegangen:
Man führt den Test ob ein Zyklus entsteht für jede hinzuzufügende Prioritätsbeziehung nacheinander durch. Wenn bei einer einzufügenden Priorität bemerkt wird, daß kein Zyklus entstehen würde, wird diese Priorität in der Struktur eingetragen. Dabei merkt man sich welche Prioritäten bereits hinzugefügt wurden. Bemerkt man dann irgendwann, daß doch ein Zyklus entsteht, so löscht man alle bereits eingetragenen Prioritäten wieder aus der Itinerary Struktur heraus.

5.2 Vorbedingungen

5.2.1 Erstellen und Verändern von Vorbedingungen

Die Vorbedingung eines Entries darf nur auf Entries verweisen, die sich direkt im gleichen I_Entry befinden wie das Entry selbst. Um dies für eine Precondition kontrollieren zu können muß das I_Entry Objekt verfügbar sein. Dies ist aber erst dann verfügbar wenn das Precondition Objekt dem Attribut `precondition` eines Entries zugewiesen wurde. Dann kann über dieses Entry das I_Entry Objekt geholt werden, vorausgesetzt der Entry ist bereits in ein I_Entry eingefügt worden.
Somit wird festgelegt, daß die Vorbedingung eines Entries nur gesetzt werden kann wenn es sich bereits in einem I_Entry befindet.
Wenn nach dem Setzen des `precondition` Attributs ein Teil der Precondition geändert wird, so wird der hinzugefügte bzw. veränderte Teil überprüft. Dadurch ist sichergestellt, daß Vorbedingungen von Entries sich immer nur auf Entries derselben I_Entry beziehen.

Wie kontrolliert wird ob alle Entries auf die sich eine Precondition bezieht im richtigen I_Entry liegen wurde bereits unter 4.3.1 beschrieben.

5.2.2 Auswerten von Vorbedingungen

Um eine Precondition auszuwerten, wird die Methode `Precond.evaluate(...)` benutzt. Jedes Precond Objekt ruft für alle Precond Objekte, auf die es über seine Attribute verweist diese Methode auf. Dann wird aus den erhaltenen Werten das Resultat des Objekts berechnet. Dies wird an einem Beispiel in Abbildung 5.6 gezeigt.

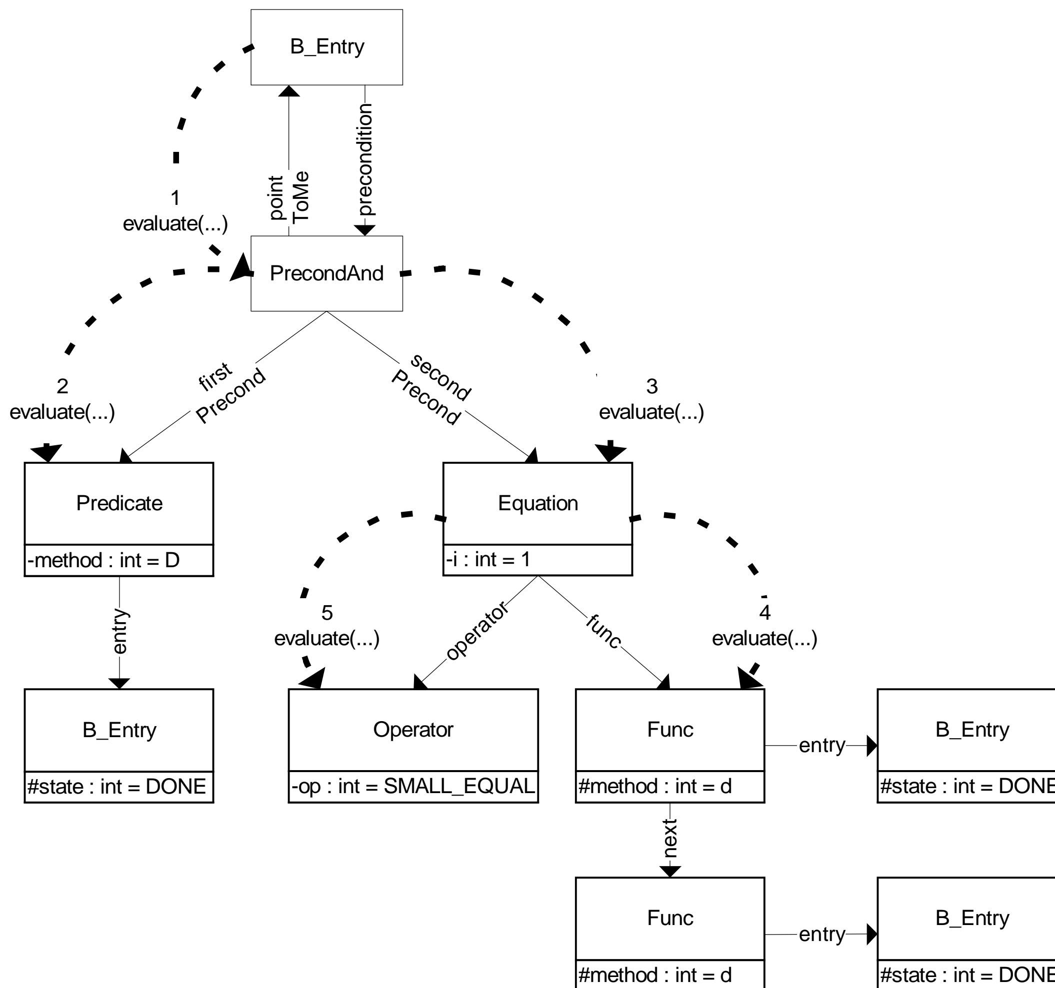


Abbildung 5.6: Methode evaluate(...)

Da Teile von Preconditions in anderen Preconditions ebenfalls benutzt werden können, könnte es vorkommen, daß beim Suchen des nächsten Entries mit `getNext()` das gleiche Precond Objekt mehrmals ausgewertet werden würde. Um dies zu verhindern hat sowohl die Precond als auch die I_Entry Klasse ein Attribut namens `countEvaluate`.

Um die Auswertung der Preconditions zu beschleunigen werden folgende Optimierungen vorgenommen:

- Precond Objekte werden nur so oft wie notwendig ausgewertet.

Am Ergebnis der Preconditions kann sich nur etwas ändern wenn der Zustand eines B_Entries auf `DONE` gesetzt wird. Somit wird jedesmal in der Methode `B_Entry.done()` das Attribut `countEvaluate` des I_Entries in dem sich der B_Entry befindet um eins erhöht. Wenn jetzt ein Precond Objekt ausgewertet werden soll, so wird als Parameter dieses `countEvaluate` mitgegeben. Ist der Wert des gleichnamigen Attributs des Precond Objekts kleiner als der Wert im Parameter so muß das Precond Objekt neu ausgewertet werden. Ansonsten wird direkt der im Attribut `result` gemerkte boolesche Wert zurückgegeben. Wenn Änderungen an Precond Objekten durchgeführt werden so wird `countEvaluate` auf –1 gesetzt um sicherzustellen, daß dieses Objekt auf jeden Fall beim nächsten Auswerten neu berechnet wird.

- Beim Auswerten von PrecondAnd und PrecondOr Objekten wird bei Feststehen des Ergebnisses abgebrochen.

Ein PrecondAnd Objekt wird nur dann wahr werden wenn beide Preconditions auf die es zeigt wahr werden. Somit wird das Auswerten von `secondPrecond` gespart wenn `firstPrecond` nicht wahr ist.
Ein PrecondOr Objekt hingegen ist auf jeden Fall wahr wenn `firstPrecond` wahr ist. Also wird auch in diesem Fall die Berechnung abgebrochen.

- Verkleinern der Precondition wenn Precond Objekte existieren, die ihr Ergebnis nicht mehr verändern.

Wenn ein Precond Objekt ausgewertet wird, untersucht es ob sich sein Ergebnis im weiteren Verlauf überhaupt noch mal ändern kann. Ist dies nicht der Fall werden alle Precond Objekte die dieses Precond Objekt benutzen benachrichtigt. Dann wird das Precond Objekt durch ein ConstBoolean Objekt ersetzt. In Frage kommen dabei nur Equation oder Predicate Objekte.
Bei einem Predicate wird sich das Ergebnis nicht mehr ändern sobald es einmal wahr ist. In diesem Fall wird an alle Objekte in `pointToMe` eine Nachricht geschickt, daß sie dieses Objekt durch ein ConstBoolean Objekt ersetzen sollen. Die Objekte sind entweder Instanzen der Klasse Entry oder Precond. Entry Objekten wird die Nachricht `setPrecond(new ConstBoolean(true))`, Precond Objekten die Nachricht `replacePrecond(Precond(this), new ConstBoolean(true))` geschickt.
Bei einer Equation ist es nicht ganz so einfach. Je nach Operator kann sich das Ergebnis ein oder zweimal ändern bevor es konstant bleibt.

Beispiel:

Wir gehen von dem Equation Objekt in Abbildung 5.7 aus. Das Attribut i der Equation sei 1. Das Ergebnis der Reihe von den drei Func Objekten ist entweder 0,1,2 oder 3, je nachdem in welchem Zustand sich die Entries eins, zwei und drei befinden. Dies ist unabhängig vom Attribut `method` der Func Objekte. Deshalb sind sie in der Abbildung nicht eingetragen.
In den folgenden Tabellen wird das Ergebnis der Equation in Abhängigkeit von i dem Operator und dem Ergebnis der Func-Reihe dargestellt.

i	Operator	Func-Ergebnis	Ergebnis der Equation
1	=	0	false
1	=	1	true
1	=	2	false
1	=	3	false

i	Operator	Func-Ergebnis	Ergebnis der Equation
1	<	0	false
1	<	1	false
1	<	2	true
1	<	3	true

Man sieht, daß sich das Ergebnis beim ersten Operator (=) zweimal verändert bis es konstant bleibt, beim zweiten (<) dagegen ändert es sich nur einmal.

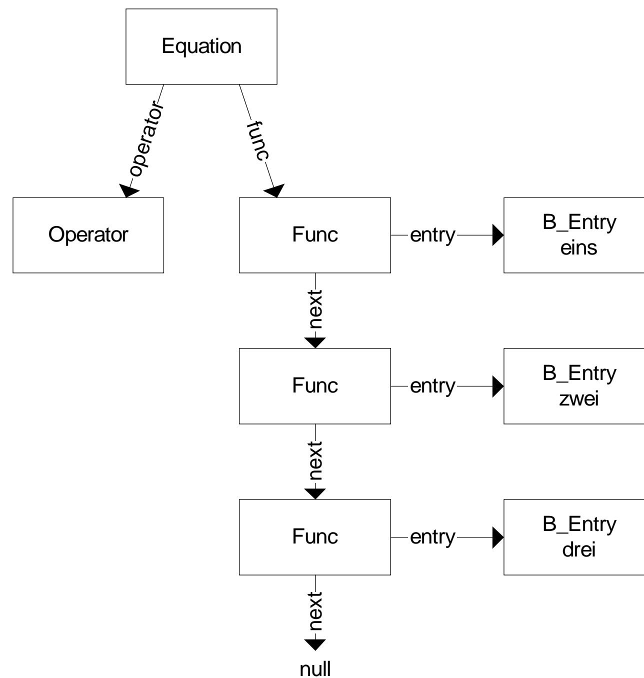


Abbildung 5.7: Equation

Um feststellen zu können ob sich das Ergebnis der Equation noch ändern wird, hat die Equation Klasse das Attribut `switched` das angibt wie oft sich das Ergebnis bereits geändert hat. Der Operator hat die Methode `staysConst(...)`, die dann an Hand von dem Parameter `switched` entscheidet ob sich das Ergebnis noch ändern kann oder nicht. Wenn sich das Ergebnis nicht mehr ändern kann wird wie beim Predicate Objekt vorgegangen.

Durch die so eingefügten ConstBoolean Objekte entstehen wieder Strukturen, die noch verkleinert werden können. Wie sich dabei die Objekte PrecondAnd, PrecondOr verhalten und welche Strukturen letztlich entstehen zeigt Abbildung 5.8. Dazu ist Folgendes zu sagen:
Ein PrecondAnd Objekt, das auf ein Precond Objekt zeigt welches sich konstant zu TRUE auswertet, kann ersetzt werden durch das zweite Precond Objekt auf das es zeigt. In der Abbildung wird das PrecondAnd Objekt and ersetzt durch die Equation equ.
Ein PrecondOr Objekt, das auf ein ConstBoolean Objekt, welches sich zu TRUE auswertet zeigt, wird im Folgenden nur noch zu TRUE ausgewertet werden. Folglich kann es durch ein konstant TRUE lieferndes ConstBoolean Objekt ersetzt werden.

In Abbildung 5.9 ist dargestellt wie sich ein PrecondNot Objekt verhält wenn es auf ein Predicate Objekt verweist, dessen Ergebnis sich nicht mehr verändert.

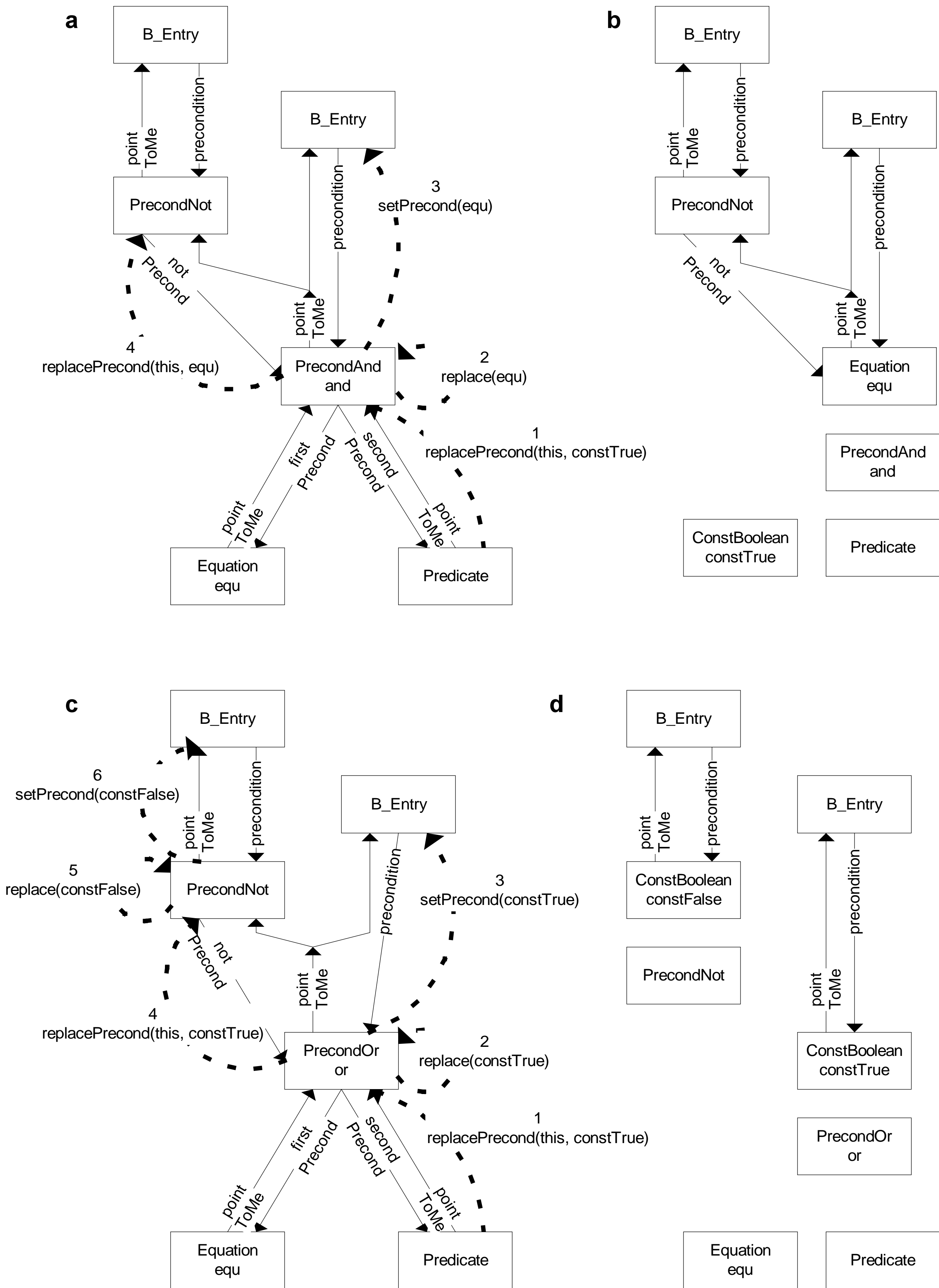


Abbildung 5.8: Veränderung des PrecondAnd/PrecondOr bei wahrem Predicate

a, c: Vorhandene Struktur und Nachrichten
b, d: Entstandene Struktur

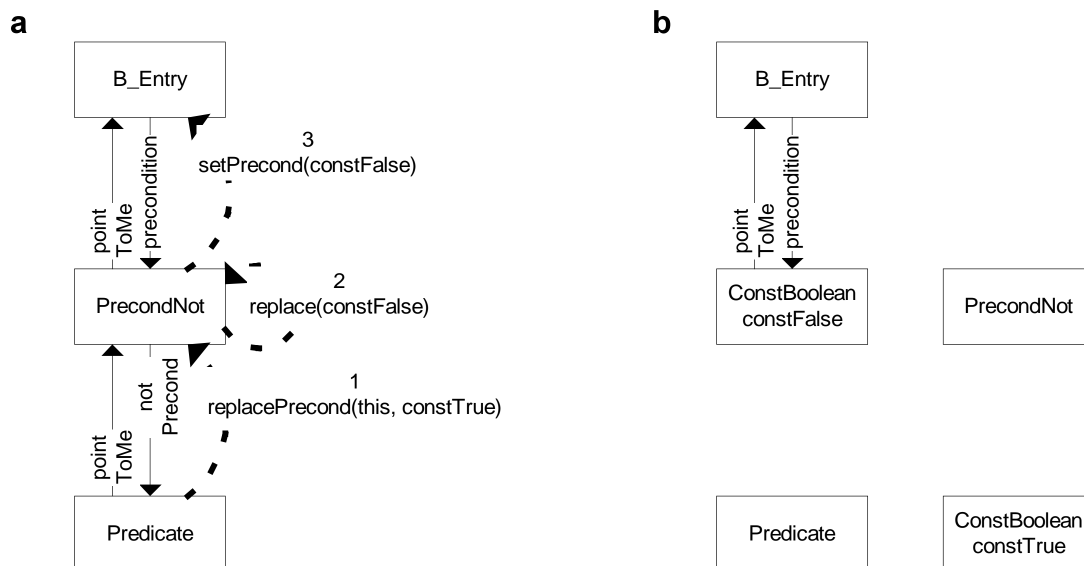


Abbildung 5.9: Veränderung des PrecondNot bei wahrem Predicate

a: Vorhandene Struktur und Nachrichten
b: Entstandene Struktur

5.3 Entries

5.3.1 Ausführbare Entries holen:

Wenn die Reiseroute komplett zusammengebaut wurde soll jetzt ein Entry nach dem anderen aus der Reiseroute gelesen und abgearbeitet werden.
Dazu werden rekursiv alle Entries der Reiseroute besucht. Dabei wird bei den Entries im Attribut `highestPriorities` begonnen. Wie sich die verschiedenen Entry Typen dabei verhalten wird im Folgenden beschrieben.

- B_Entry:

Ein B_Entry darf nur dann ausgeführt werden wenn kein Entry, das sich im Attribut `prioritiesBef` befindet ausführbar ist und wenn die Precondition wahr ist. Somit muß der Algorithmus zuerst für alle Entries aus dem Vector `prioritiesBef` aufgerufen werden. Wenn dann noch kein B_Entry zum Ausführen gefunden wurde so gibt es kein Entry mit höherer Priorität, das gestartet werden darf. Also wertet das B_Entry seine Precondition aus. Ist diese erfüllt, so darf das B_Entry ausgeführt werden. Die Precondition wird aber nur ausgewertet wenn das B_Entry nicht im Zustand DONE, also noch nicht ausgeführt ist. Wenn das Entry nicht ausgeführt werden kann wird im Vector `prioritiesAfter` weitergesucht. Dies wird in Abbildung 5.11 a und c an Hand der verwendeten Methode `getNext(...)` erläutert. Mit dem bisher beschriebenen Vorgehen kann es dazu kommen, daß Entries mehrfach untersucht werden. Im schlimmsten Fall endet die Rekursion nie (s. Abbildung 5.11 b). Um das zu vermeiden merkt sich der Algorithmus alle bereits untersuchten Entries im Vector `evalList`. Wenn auf diese Weise bemerkt wird, daß ein Entry schon einmal besucht wurde, wird die Rekursion an dieser Stelle

abgebrochen. Auch die Entries, die bereits ausgeführt wurden, werden in einem Vector gehalten. Dadurch wird das mehrfache Besuchen von bereits ausgeführten Entries umgangen.

- Open_I_Entry:

Entries aus einem Open_I_Entry dürfen ausgeführt werden wenn die Precondition der Open_I_Entry wahr ist und kein Entry mit höherer Priorität ausführbar ist. Nach Ausführen eines B_Entries aus der Open_I_Entry sollen unabhängig der Precondition, die anderen Entries der Open_I_Entry ausgeführt werden dürfen. Natürlich nur wenn deren Precondition wahr ist. Dies wird dadurch realisiert, daß beim Ausführen eines B_Entries, die Precondition aller Open_I_Entries, in denen er sich befindet auf wahr gesetzt werden.
Der Algorithmus durchsucht wieder alle Entries, die sich in `prioritiesBef` befinden. Falls dabei kein geeignetes B_Entry gefunden wurde wird geschaut ob sich die Open_I_Entry bereits im Zustand `DONE` befindet. Wenn dies der Fall ist wird direkt für alle Entries im Vector `prioritiesAfter` nach einem Ausführbaren gesucht. Ansonsten wird zuerst die Precondition ausgewertet. Wenn diese wahr ist wird nun im Attribut `highestPriorities` der Open_I_Entry weitergesucht. Da die Entries, die sich in diesem Vector befinden die höchste Priorität haben, befinden sich im Vector `prioritiesBef` dieser Entries keine Einträge. Dadurch wird der Algorithmus sehr stark beschleunigt. Die Methodenaufrufe der Methode `getNext(...)`, die zum Finden von ausführbaren Entries benutzt werden, sind in Abbildung 5.10 dargestellt.

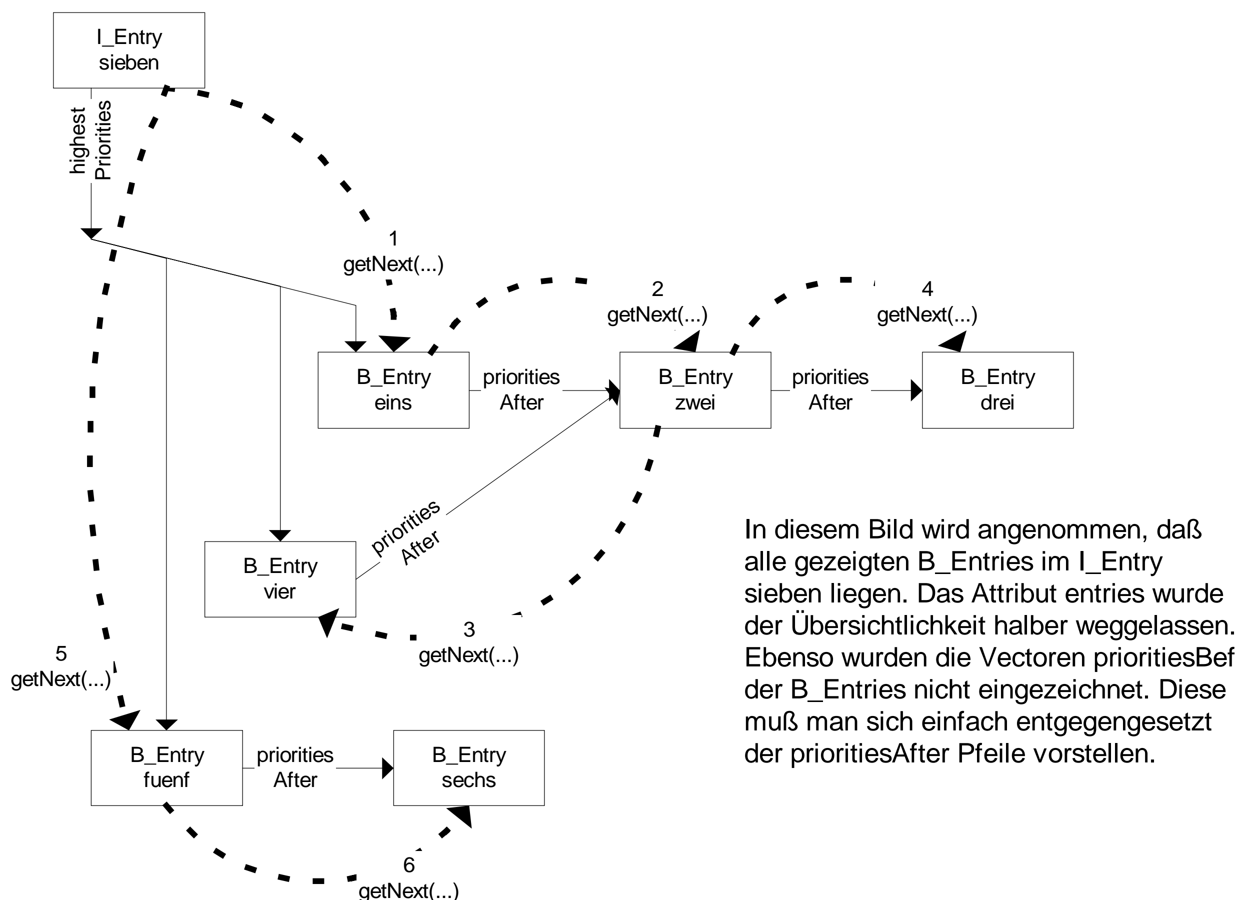
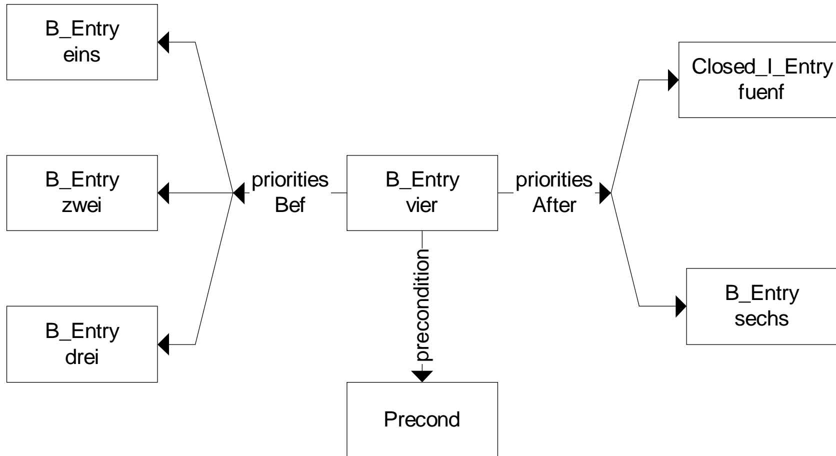


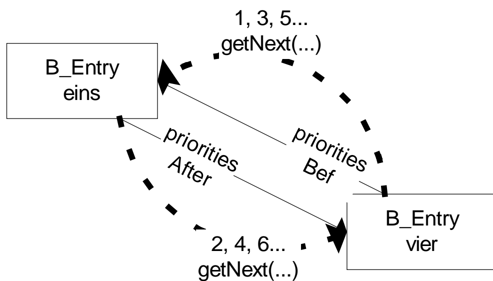
Abbildung 5.10: Methode getNext(...) bei I_Entry und B_Entry

a



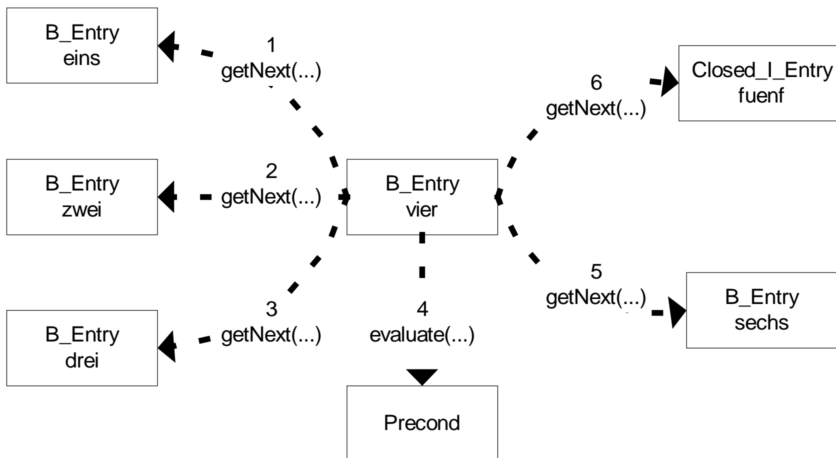
In diesem Bild wird wieder angenommen, daß alle gezeigten Entries in einem I_Entry liegen. Es sind nur die Objekte eingezeichnet, die zum Erklären des Sachverhalts notwendig sind.

b



B_Entry vier ruft getNext von B_Entry eins auf. Da dieser keine Entries in prioritiesBef hat und angenommen wird, daß die Precondition nicht wahr ist so ruft er für alle in prioritiesAfter getNext auf. B_Entry vier ruft wieder B_Entry eins auf und so weiter.

c



Es werden nicht unbedingt alle diese Nachrichten abgesetzt. So bald ein ausführbares B_Entry gefunden wurde wird abgebrochen.

Abbildung 5.11: Methode getNext() für B_Entry

a: Vorhandene Struktur
b: Nicht abbrechende Rekursion
c: Nachrichten

- Closed_I_Entry:

Wenn keine Entries mit höherer Priorität ausgeführt werden können und die Precondion erfüllt ist können die Entries der Closed_I_Entry ausgeführt werden. Sobald ein Closed_I_Entry gestartet wurde so werden solange nur noch Entries aus der Closed_I_Entry ausgeführt, bis sich diese im Zustand `DONE` befindet.
Der Algorithmus überprüft also zuerst ob der Closed_I_Entry schon gestartet wurde. In diesem Fall wird direkt im Attribut `highestPriorities` der Closed_I_Entry weitergesucht. Wenn er noch nicht gestartet wurde so wird gleich vorgegangen wie bei der Open_I_Entry. Um durchzusetzen, daß nach dem Start eines Closed_I_Entry nur noch Entries daraus ausgeführt werden, wurde für die Itinerary Klasse das Attribut `lastDoneEntry` eingefügt. In diesem Attribut wird das zuletzt ausgeführte B_Entry gehalten. Wenn der Algorithmus gestartet wird, überprüft er zuerst ob sich das `lastDoneEntry` der Itinerary in einem Closed_I_Entry befindet, welcher noch nicht im Zustand DONE ist. Ist dies der Fall so wird nur in diesem Closed_I_Entry nach ausführbaren B_Entries gesucht.. Ansonsten wird jedes Entry im Attribut `highestPriorities` der Itinerary untersucht.

5.3.2 Nicht ausführbare Entries ermitteln:

Wenn Vorbedingungen zyklische Abhängigkeiten besitzen, so kann es vorkommen, daß mehrere B_Entries nicht ausgeführt werden können. Das einfachste Beispiel dafür würde durch den folgenden Code entstehen. Dabei wird angenommen, daß die B_Entries eins, zwei und drei gemeinsam im Itinerary Objekt iti liegen.

```
iti.before(eins, zwei);
iti.before(zwei, drei);
iti.before(drei, eins);
```

Solche nicht ausführbaren B_Entries sollen gefunden werden. Dabei wird wie folgt vorgegangen:
Das Holen und Ausführen der B_Entries der gesamten Itinerary wird simuliert. Dabei holt man sich zuerst alle B_Entries, die noch nicht ausgeführt wurden. Diese Entries werden im Vector `entriesNotRun` gespeichert. Danach werden alle ausführbaren B_Entries geholt. Jetzt wird einer nach dem Anderen der ausführbaren Entries auf den Zustand `DONE` gesetzt und aus dem Vector `entriesNotRun` entfernt. Nachdem wieder alle ausführbaren Entries geholt wurden, wird wieder einer nach dem Anderen von diesen auf DONE gesetzt und so weiter. Dies wird rekursiv für alle ausführbaren B_Entries gemacht. Es werden also alle Kombinationen, die beim Ausführen möglich sind durchgetestet. Die Entries, die am Schluß immer noch im Vector `entriesNotRun` stehen, können nie ausgeführt werden. Am Schluß muß noch darauf geachtet werden, daß die Zustände zurückgesetzt werden, nicht nur die der B_Entries auch die der Closed- und Open_I_Entries, die automatisch beim Auswerten der Preconditions auf `DONE` gesetzt wurden.

Diese Vorgehensweise wurde durch die Methode `testItinerary(iti: Itinerary, entriesNotRun: Vector)` implementiert. In Abbildung 5.12 sind die Methodenaufrufe und der Vector `entriesNotRun` an einem Beispiel verdeutlicht. Dabei wurden nicht alle Nachrichten eingezeichnet. Nach dem 6. Aufruf von `testItinerary(...)` würde die ganze Itinerary noch mal für

den Fall durchgetestet werden, daß als erstes Entry 5 anstatt Entry 4 ausgeführt wird.

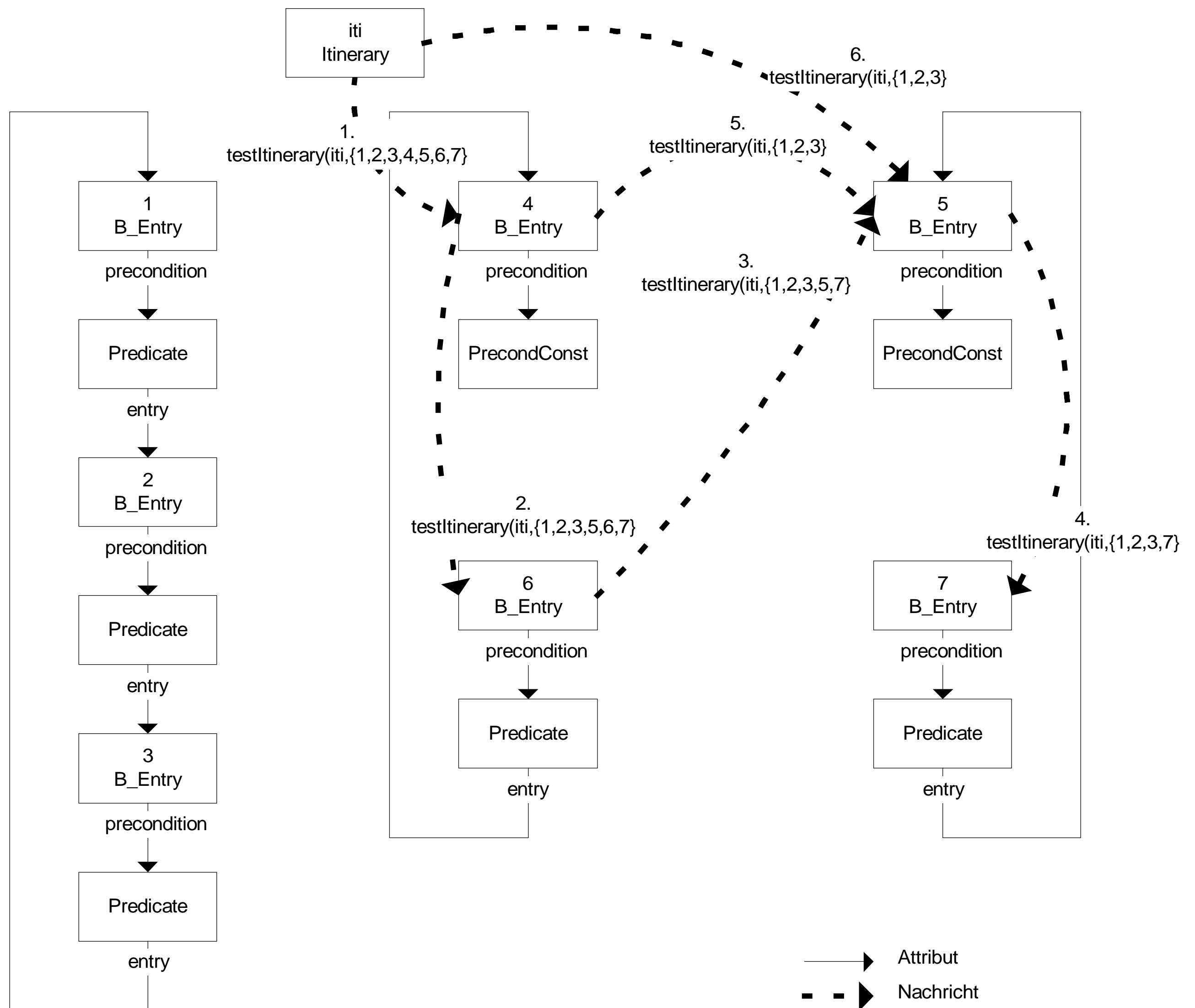


Abbildung 5.12: Nachrichten von testItinerary(...)

6. Mole Integration

In diesem Kapitel werden die Veränderungen an Mole beschrieben. Es wurde darauf geachtet, daß das alte Migrationkonzept weiterhin genutzt werden kann.

Erweiterungen der Klasse Agent

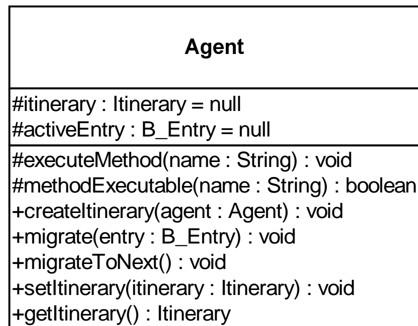


Abbildung 8.1: Erweiterungen der Klasse Agent

Dem Agent wurde ein Itinerary Objekt hinzugefügt, das durch die Methode `createItinerary(agent: Agent)` erzeugt wird. Dabei wird der Agent, der zur Reiseroute gehört übergeben.
Weiterhin erhielt er ein Attribut `activeEntry` in dem immer das auszuführende Entry gespeichert ist. Somit weiß der Agent beim Ankommen auf der nächsten Location, wessen `toDo` also welche Methode er dort auszuführen hat.
Die Methode `executeMethod(name: String)` bietet die Möglichkeit, die über einen String festgelegte Methode des Agenten aufzurufen. Dieser String wird per Parameter übergeben. Dies wird über das Class Objekt erreicht. Das Class Objekt liefert ein Methoden Objekt. Die Namen der in diesem Objekt enthaltenen Methoden werden jetzt mit dem String verglichen. Existiert die Methode, so wird sie durch `invoke(...)` ausgeführt. Die Methode `executeMethod(...)` wird benutzt um beim Ankommen auf einer Location, die im `activeEntry` im Attribut `toDo` beschriebene Methode auszuführen.
Die Methode `methodExecutable(name: String)` bekommt einen String übergeben und prüft ob eine Methode mit dem übergebenen Namen im Agenten existiert. Dabei wird auch darauf geachtet, daß die Methode public ist und keine Parameter erwartet. Existiert eine solche Methode wird der Wert true zurückgeliefert. Dies wird beim Setzen von `toDo` in einem B_Entry benutzt, um nicht ausführbare Methoden direkt ausschließen zu können.
Das Migrieren des Agenten zur nächsten Location wird durch die Methode `migrateToNext()` realisiert. Sie überprüft zuerst ob eine Itinerary vorhanden ist. Wenn dies nicht der Fall ist, wird eine ItineraryException ausgelöst. Dann wird mit der Methode itinerary.getNext ein B_Entry aus der Reiseroute des Agenten geholt der ausgeführt werden darf. Dieser wird dann der Methode `migrate(...)` übergeben.
Die Methode `migrate(...)` ruft für die aktuelle Location des Agenten die Methode `goTo(...)` mit dem im B_Entry im Attribut `dest` gesetzten LocationName auf. Zuerst wird jedoch kontrolliert ob eine Reiseroute für den Agenten erzeugt wurde. Ist dies nicht der Fall wird eine ItineraryException ausgelöst.

Bevor dann auf die neue Location migriert wird, wird das `activeEntry` des Agenten auf das übergebene B_Entry gesetzt. Wenn das `activeEntry` der Itinerary ungleich null ist wird dies mit der Methode `entryDone()` in den Zustand `DONE` gesetzt. Warum dies notwendig ist wird im nächsten Abschnitt beschrieben.
Zusätzlich existieren noch die Methoden `setItinerary(...)` und `getItinerary()` zum Schreiben und Lesen des Itinerary Objekts.

Änderungen an der privaten Klasse StandardAgentThread

Um die Änderungen klar zu machen wird zuerst beschrieben wie die Klasse Standard AgentThread bisher beim Migrieren verwendet wurde. Diese Beschreibung ist sehr vereinfacht und dient nur dazu die hier wichtigen Abläufe zu verstehen.
Dann erst wird erklärt was verändert wurde.

In der oben beschriebenen Methode `migrate(...)` wird für die aktuelle Location des Agenten die Methode `goTo(...)` aufgerufen. Als Parameter wird der Name, der zu besuchenden Location übergeben.
Wenn der Agent auf der Location ankommt wird zuerst seine `prepare()` Methode aufgerufen. In der Methode `prepare()` wird standardmäßig kein Code ausgeführt. Sie ist zum Überschreiben gedacht um auf der angekommenen Location irgendwelche Initialisierungen vorzunehmen. Dann wird ein StandardAgentThread erzeugt der dann direkt mit `start()` gestartet wird. Daraufhin wird die für uns wichtige Methode `runSubclass()` des StandardAgentThread's ausgeführt.
Diese Methode sorgt nun dafür, daß die Methode `start()` des Agenten aufgerufen wird.

In der veränderten Version muß in der Methode `runSubclass()` nicht `start()` sondern die `toDo` Methode des aktiven B_Entries `activeEntry` des Agenten ausgeführt werden. Um nun die alte Funktionalität weiter unterstützen zu können wird in `runSubclass()` zuerst überprüft ob das Itinerary Objekt gleich null ist. Ist dies der Fall wird wie bisher auch die Methode `start()` des Agenten aufgerufen. Ist aber ein Itinerary Objekt im Agenten enthalten, so wird mit der Methode `executeMethod()` des Agenten die Methode, die im aktiven B_Entry beschrieben ist ausgeführt. Davor wird das B_Entry über die Methode `activate()` in den Zustand `ACTIVE` gesetzt. Wenn der Agent aus der aufgerufenen Methode wieder zurückkehrt, also nicht zu einer anderen Location geschickt wurde, wird er durch Aufruf der Methode `migrateToNext()` auf die nächste Location geschickt. Bisher wurde verpaßt den aktiven B_Entry mit der Methode `done()` in den Zustand `DONE` zu setzen. Da aber nicht sicher ist, daß der Agent aus der aufzurufenden Methode zurückkehrt wird das nicht an dieser Stelle gemacht. Das Aufrufen der Methode `Itinerary.entryDone()` wird beim darauffolgenden Methodenaufruf von `migrate(...)` erledigt. Dies wurde weiter oben in diesem Kapitel bei der Beschreibung von `migrate(...)` kurz erwähnt.

Ein Agent kann auch weiterhin mit migrateTo(...) auf eine andere Location geschickt werden. Es ist aber nicht möglich das alte (migrateTo()) und das neue (migrate(...), migrateToNext(...)) Migrationskonzept gleichzeitig zu verwenden. Wenn für den Agenten ein Reiseroutenobjekt erzeugt wird. So dürfen nur die Methoden `migrate(...)` und `migrateToNext(...)` verwendet werden. Wird in diesem Fall

die Methode `migrateTo(...)` verwendet wird beim ankommen auf der neuen Location eine ItineraryException ausgelöst. Dies liegt daran, daß an Hand des Vorhandenseins eines Itinerary Objekts entschieden wird ob die Methode `start()` des Agenten oder die Methode `toDo`, die im aktiven B_Entry festgelegt ist im Agent ausgeführt wird. Existiert also eine Reiseroute wird versucht das Attribut `activeEntry` des Agenten zu lesen, dies wird aber in der Methode `migrateTo(...)` nicht gesetzt. Somit würde eine NullPointerException entstehen, die durch die ItineraryException verhindert wird.

7. Fazit

Die Kommunikation zwischen Agent und Reiseroute sollte noch erweitert werden. Der Agent sollte der Reiseroute mitteilen können wenn ein Rechner nicht erreichbar ist. Daraufhin könnte die Reiseroute den zugehörigen Entry in irgendeiner Weise zurückstellen, so daß der Agent die Möglichkeit bekommt erst andere Rechner zu besuchen. Erst wenn kein anderer Entry mehr ausführbar ist würde die Reiseroute den zurückgestellten Entry wieder freigeben.

Literaturverzeichnis

[Eckel 98] Bruce Eckel (1998): Thinking in Java, Prentice Hall PTR

[GronSun 97] Daniel Groner & Todd Sunsted (1997): Java API Superbible: Die definitive Sprachreferenz zu Java 1.1, SAMS

[StraRoth 98] M. Straßer and K. Rothermel (1998): Reliability Concept for Mobile Agents, International Journal of Cooperative Information Systems (IJCIS)

[BaumEA 98] J. Baumann, F. Hohl, K. Rothermel and M. Straßer (1998): Mole - Concepts of a Mobile Agent System

[Oester 98] Bernd Oestereich (1998): Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language, Oldenburg

Anhang

Projektplan für Studienarbeiten

Titel:	Reiserouten-Konzepte für Mobile Agenten
Autor:	Jürgen Buschle
Datum:	17.05.99

1 Einleitung

Der Projektplan dient dem Bearbeiter und dem Betreuer als Grundlage für die Projektüberwachung und -steuerung. Er legt den Projektablauf aber nicht ein für allemal fest, sondern soll im Laufe der Arbeit ergänzt und angepaßt werden. So ist es z.B. möglich, daß anfänglich spezifizierte Arbeitspakete weiter verfeinert oder Arbeitspakete zugunsten anderer aufgegeben werden.

Der Projektplan kann in Absprache mit dem Betreuer in beiderseitigem Einverständnis in den folgenden Fällen angepaßt werden:

- bei Verzug,
- bei vorzeitigem Abschluß eines Arbeitspakets oder
- bei Gewinn neuer Erkenntnisse.

Der aktualisierte Projektplan soll in den Anhang der Ausarbeitung einfließen.

Projektbeschreibung:

Bei diesem Projekt handelt es sich um eine Studienarbeit in der Abteilung Verteilte Systeme am Institut für Parallele und Verteilte Höchstleistungsrechner der Universität Stuttgart.

Reiserouten-Konzepte für Mobile Agenten

Unter Mobilen Agenten versteht man "Programmkonstrukte", die die Fähigkeit haben, autonom zu handeln und von Rechner zu Rechner zu migrieren, falls auf beiden Seiten ein "Agentensystem" installiert ist. Das Modell der Mobilen Agenten ist ein neuer, vielversprechender Ansatz im Bereich der Verteilten Systeme. Die Abteilung Verteilte Systeme forscht auf diesem Gebiet und erstellt im Rahmen des Projektes "Mole" ein System, mit dem Mobile Agenten eingesetzt werden können.

Die Reiseroute, die ein Agent während der Erledigung seiner Aufgabe zurücklegt, kann innerhalb des Agenten prinzipiell auf zwei unterschiedliche Arten "repräsentiert" sein. Eine Möglichkeit ist die implizite Repräsentation der Reiseroute durch im Programmcode des Agenten enthaltene "Gehe zu Rechner x"-Anweisungen. Die andere Möglichkeit ist die explizite Repräsentation der Reiseroute. Hierbei wird die Reiseroute des Agenten (zumindest teilweise) im voraus festgelegt und in einem Objekt (oder einer Datenstruktur) abgelegt. Der Code des Agenten enthält in diesem Falle nur "Gehe zum nächsten Rechner"-Anweisungen. Um den Agenten in seiner

Flexibilität nicht einzuschränken, kann die Reiseroute im Laufe der Ausführung des Agenten abgeändert werden.
Der Ansatz der impliziten Repräsentation der Reiseroute, welcher momentan in "Mole" Verwendung findet, ist sehr einfach anwendbar und sehr flexibel. Er bietet jedoch den Nachteil, daß das den Agenten ausführende Agentensystem kein Wissen über die Reiseroute des Agenten besitzt. Ist die Definition einer Reiseroute ausreichend flexibel, so könnte das Agentensystem dieses Wissen nutzen, um bestimmte Optimierungen durchzuführen. Eine mögliche Optimierung hierbei wäre beispielsweise die Bestimmung des kürzest möglichen Weges. Eine andere Optimierung wäre, daß Rechner, die besucht werden müssen aber gerade nicht erreichbar sind (Netzwerkprobleme, Rechnerausfall), erst "möglichst spät" besucht werden.
Aufgabe dieser Arbeit ist es, ein gegebenes Konzept zur expliziten, sehr flexiblen Definition einer Reiseroute auf Erweiterungsmöglichkeiten zu evaluieren. Die Implementierung dieses erweiterten Konzepts in "Mole" ist zu konzipieren und durchzuführen. Weiterhin soll die Integration des Reiserouten-Konzeptes in ein Protokoll zur fehlertoleranten Ausführung von Agenten vorbereitet werden.

Die Implementierung erfolgt in Java. Bei der Implementierung sind die Projektrichtlinien der Abteilung zu beachten. Die Ergebnisse der Arbeit sind in einem Abschlußvortrag zu präsentieren.

Bearbeitungszeitraum: 1.5.99- 31.10.99

Betreuer: Markus Straßer

In den nachfolgenden Abschnitten werden zunächst die Arbeitspakete identifiziert und beschrieben, dann wird ein Zeitplan zu deren Durchführung festgelegt und schließlich erfolgt die Definition der Meilensteine und der dafür zu erstellenden Dokumente.

2 Beschreibung der Arbeitspakete

Dieser Abschnitt umfaßt die Definition der wesentlichen Arbeitspakete, die Abhängigkeiten zwischen einzelnen Paketen und die Identifikation von kritischen Punkten, die die Durchführung des Projektes gefährden könnten.

2.1 Definition der Arbeitspakete

Die im Rahmen der Diplomarbeit durchzuführenden Tätigkeiten lassen sich in folgende, zusammenhängende Arbeitspakete aufgliedern:

AP1, Projektplan: Erstellung eines initialen Projektplanes.

Status: abgeschlossen

AP2, Einarbeitung: Genaue Einarbeitung in die Aufgabenstellung und die relevante Literatur (Mole, Java, UML).

Status: abgeschlossen

AP3, Erweiterung: Gegebenes Reiseroutenkonzept auf Erweiterungsmöglichkeiten untersuchen und gegebenenfalls Erweiterungen vornehmen. Plausibilitätstests für die Reiseroute. Vorbereitung der Integration des Reiseroutenkonzeptes in ein Protokoll zur fehlertoleranten Ausführung von Agenten. Festlegen der genauen Funktionalität.

Status: abgeschlossen

AP4, Entwurf: Klassenentwurf mit UML.

Status: abgeschlossen

AP5, Implementierung: Implementierung der Klassen mit Java.

Status: abgeschlossen

AP6, Klassentest: Testen der Klassen unabhängig von Mole.

Status: abgeschlossen

AP7, Mole-Integration: Integrieren des Reiseroutenkonzepts in Mole mit Entwurf dafür.

Status: abgeschlossen

AP8, Mole-Test: Test auf korrekte Funktionalität mit Mole.

Status: abgeschlossen

AP9, Ausarbeitung: Zusammenfassen der Ergebnisse in schriftlicher Form.

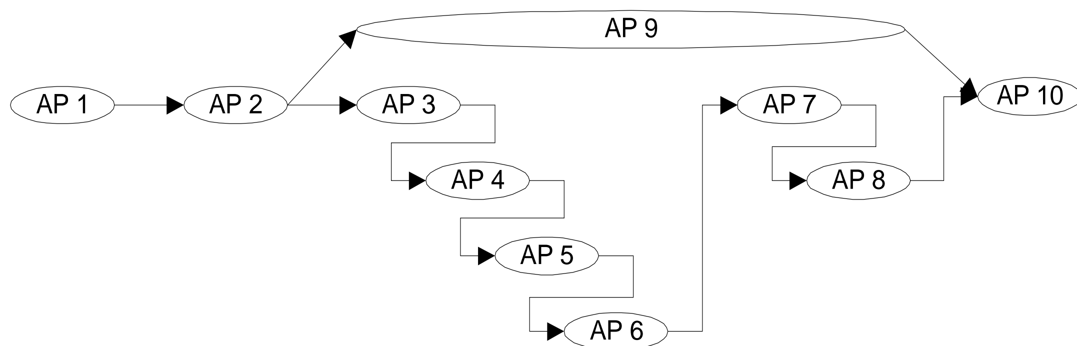
Status: abgeschlossen

AP10, Pflege des Projektplans: Überarbeiten des Projektplans.

Status: abgeschlossen

2.2 Abhängigkeiten der Arbeitspakete

Zwischen den Arbeitspaketen der Studienarbeit bestehen die folgenden Abhängigkeiten:



Zuerst wird AP1 (Projektplan) durchgeführt. Danach folgt AP2 (Einarbeitung). Beide Arbeitspakete haben keine Vorbedingungen. Darauf wird mit AP3 (Erweiterung) begonnen. Um dieses Arbeitspaket erledigen zu können, muß man sich vorher in Java und in die Aufgabe eingearbeitet haben. Es ist möglich, daß Teile dieses Arbeitspakets erst später erledigt werden, z.B. während des Implementierens, wenn man feststellt, daß die festgelegte Funktionalität nicht ausreicht. Wenn die genaue Funktionalität festgelegt wurde wird AP4 (Entwurf) gestartet. Hier ist es möglich, daß man erst beim Implementieren oder beim Testen Fehler im Entwurf feststellt und dann zu diesem Arbeitspaket zurückspringt. Um mit AP5 (Implementierung) beginnen zu können, sollte der Entwurf möglichst vollständig sein. Doch teilweise können Entwurfsentscheidungen erst während des Implementierens getroffen werden. So kann es vorkommen, daß man den Entwurf verändern muß. Wenn die Implementierung abgeschlossen ist, folgt AP6 (Klassentest). Teile des Tests werden schon während der Implementierung erledigt.Um mit AP7 (Mole-Integration) fortfahren zu können, müssen die AP's 3-6 vollständig erledigt sein. Auch hier wäre es theoretisch möglich, daß die entworfene Reiseroute in Mole nicht genügend Funktionalität liefert und erweitert werden muß. So kann es vorkommen, daß man AP3 wieder aufnimmt. Danach folgt AP8 (Mole-Test). Dieser Test wird wieder teilweise parallel zur Mole-Integration ablaufen. Teilweise parallel zu den Arbeitspaketen 3-8 soll AP9 (Ausarbeitung) ablaufen.Um dieses Paket abzuschließen, müssen alle vorigen Arbeitspakete beendet sein. Wenn dies der Fall ist folgt AP10 (Pflege des Projektplans).

2.3 Kritische Punkte / Risiken

Der kritische Punkt war, daß ich mit der Programmierung sehr zügig voran kam aber die Ausarbeitung immer weiter nach hinten geschoben haben. So wurde es am Schluß sehr knapp.

3 Zeitplan

In diesem Abschnitt erfolgt die zeitliche Planung der Arbeitspakete und Meilensteine. Der Bearbeitungszeitraum ist 6 Monate oder 26 Wochen. Wenn sich der Zeitraum mehrerer Arbeitspakete überlappt, dann werden diese parallel bearbeitet. Am Ende des Bearbeitungszeitraums ist ein Puffer eingeplant worden, der für Verzögerungen oder andere unvorhergesehene Tätigkeiten genutzt werden soll.

geplanter Zeitraum	Aufwand	Arbeitspakete und Meilensteine	tatsächlicher Zeitraum
01.05. - 21.05.	3 PW	*AP1 Projektplan, AP2 Einarbeitung, Meilenstein 1*	*01.05.-21.05.*
22.05. - 16.07.	4 PW	AP3 Erweiterung, Meilenstein 2	*22.05.-16.07.*
05.06. - 23.07.	4 PW	AP4 Entwurf, Meilenstein 3	05.06.-23.07.
19.06. - 13.08.	6 PW	AP5 Implementierung, AP6 Klassentest, Meilenstein 4	19.06.-13.08.
14.08. - 03.09.	3 PW	AP7 Mole-Integration, AP8 Mole-Test, Meilenstein 5	14.08.-01.09.
22.05. - 08.10.	4 PW	AP9 Ausarbeitung, AP10 Pflege des Projektplans, Meilenstein 6	20.09.-30.10.
9.10. - 30.10.	3 PW	Puffer	

(PW = Personenwoche)

4 Meilensteine und Dokumente

Als Ergebnis der jeweiligen Arbeitspakete sollen zu den Meilensteinen folgende Dokumente entstehen, die nach diesen einer Versionskontrolle unterliegen und nur in Absprache mit dem Betreuer geändert werden sollen.

21.05.1999, Meilenstein 1: Projektplan, Literaturliste

16.07.1999, Meilenstein 2: Beschreibung des Reiseroutenkonzepts mit Erweiterungen

23.07.1999, Meilenstein 3: UML - Klassendiagramme

13.08.1999, Meilenstein 4: Testdaten, Quellcode

03.09.1999, Meilenstein 5: Testdaten, Quellcode, Entwurfsbeschreibung

31.10.1999, Meilenstein 6: Ausarbeitung, abschließender Projektplan

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

____

(Jürgen Buschle)