Studienarbeit
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Betreuung: Ingrid Wetzel
Mai 1996

Konzeption und Implementierung eines "Reaktionsmusters" für objektorientierte Softwaresysteme

Stefan Roock
Vaasastr. 12
24109 Kiel
Matr.Nr.: 4176074
Email: 1roock@informatik.uni-hamburg.de
roock@t-online.de

Henning Wolf
Eichenweg 67
21493 Schwarzenbek
Matr.Nr.: 4175993
Email: 1wolf@informatik.uni-hamburg.de
henning.wolf@t-online.de

Wir bestätigen hiermit, daß wir die hier vorgelegte Studienarbeit ausschließlich mit den aufgeführten Mitteln erstellt haben.

Inhaltsverzeichnis

1. Motivation und Überblick über die Arbeit 3
1.1. Einleitung und Motivation 3
1.2. Ausgangspunkt reaktives System 3
1.3. Zielsetzung 4
1.4. Überblick über die Arbeit 5
1.5. Sprachliche und graphische Konventionen 5

2. Einführung der Terminologie sowie in die Werkzeug&Material-Metapher 9
2.1. Allgemeine Terminologie 9
2.2. Leitbild und die Werkzeug&Material-Metapher 15
2.2.1. Materialien 16
2.2.2. Werkzeuge 16
2.2.3. Trennung von Funktion und Interaktion 17
2.2.4. Automaten 17
2.3. Zusammenhang von Werkzeug und Material: Aspekte 17
2.4. Die Architektur von Werkzeugen 18
2.5. Konstruktion komplexer Werkzeuge 20

3. Verschiedene Reaktionsmechanismen im Vergleich 23
3.1. Ein Beispiel 23
3.2. Die Struktur zur Beschreibung der Muster 24
3.3. Beobachter-Muster: OBSERVER 26
3.4. Kommando-Muster: COMMAND 33
3.5. Ereignis-Muster: EVENT-NOTIFICATION 40
3.6. Tabellarische Gegenüberstellung der Muster 50

4. Probleme, Besonderheiten und Anforderungen beim Einsatz von Reaktionsmechanismen 51
4.1. Ungünstige Benachrichtigungsstrukturen 51
4.2. Ein Leitbild für den Reaktionsmechanismus 57
4.3. State-Charts 59
4.3.1. Fachliche Modellierung von State-Charts 60
4.3.2. Spezialisierung in State-Charts 62
4.3.3. Zusammenfassung State-Charts 64
4.4. Ereignis-Parameter 64
4.5. Reaktion auf Benutzeraktion 65
4.6. Propagieren von Benachrichtigungen 66
4.7. Anforderungen 68
4.7.1. Unterscheidbare Ereignisse 68
4.7.2. Explizitmachung 68
4.7.3. Netzwerkfähigkeit 69
4.7.4. Ereignis-Parameter 69
4.7.5. Event-Manager für die globale Ereignisfluß-Steuerung 70
4.7.6. Dynamische Überprüfung von reaktiven Änderungen 70
4.7.7. Unterstützung der State-Chart-Sichtweise 70
4.7.8. Kommando-Muster für Reaktion auf Benutzeraktion 70
4.7.9. Portierbarkeit des Konzeptes 71
4.7.10. Bewertung der vorgestellten Reaktionsmuster 71

5. Konzeption eines neuen Musters: EVENT-OBSERVER 72
5.1. Die Klassen OBSERVER und OBSERVABLE 73
5.2. Die Event-Klassen 73
5.3. Die Dynamik des Musters 75
5.4. IAT-Events 77
5.5. Der Event-Manager 78
5.6. Dynamische Überprüfung reaktiver Änderungen 81
5.7. Die Implementierung 82
5.8. Ein modifiziertes COMMAND-Muster 85

6. Die Umstellung der Bibliothek 87
6.1. Umstellung der Bibliotheksklassen 87
6.2. Umstellung vorhandener Anwendungen 87
6.3. Migrationsmuster 87

7. Diskussion und Abschluß 88

8. Literatur 90

Anhang A. Implementation des Event-Observer-Musters (C++) 93

Anhang B. Migrationsmuster für den Notifier-Mechanismus 125

Anhang C. Stichwortverzeichnis 127

1. Motivation und Überblick über die Arbeit

In diesem Kapitel soll kurz die Motivation für diese Arbeit erläutert sowie ein Überblick über die Struktur der Arbeit gegeben werden.

1.1. Einleitung und Motivation

Diese Arbeit entstand am Arbeitsbereich Softwaretechnik des Fachbereiches Informatik der Universität Hamburg. Das C++-Rahmenwerk (Framework) des Arbeitsbereiches stellt eine einfache Implementierung eines Reaktionsmechanismus[1] (siehe [Rie93a] und [Rie93b]) zur Verfügung. Die Erfahrungen der Vergangenheit haben gezeigt, daß dieser Mechanismus zur Realisierung komplexer Anwendungen zu wenig mächtig ist.
Die Umstellung des Rahmenwerkes auf ein neues Reaktionsmuster stellt einen Teil größerer Umstrukturierungsmaßnahmen an diesem Rahmenwerk dar.

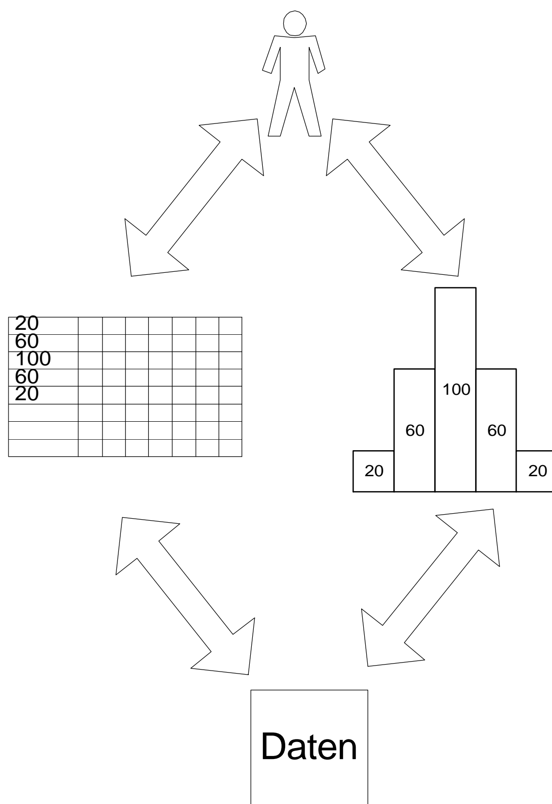
1.2. Ausgangspunkt reaktives System

Wir gehen davon aus, daß angemessene und nützliche Anwendungssysteme durch interaktive, reaktive Software unter grafischen Benutzungsoberflächen realisiert werden.
Der Begriff des reaktiven Systems umfaßt insbesondere, daß das Anwendungssystem nur als Reaktion auf Benutzeraktionen aktiv wird. Auf der anderen Seite erhält der Benutzer immer eine Reaktion auf seine Aktion (Rückkopplung). Betätigt er z.B. einen Knopf in einem Programmfenster, so wird eine entsprechende Aktion vom Softwaresystem ausgeführt und der Benutzer erhält eine Reaktion (beispielsweise eine geänderte Bildschirmdarstellung). An dieser Rückkopplung kann er die Folgen seines Handelns erkennen, einschätzen und überprüfen.
Wenn wir weiter davon ausgehen, daß komplexe Softwaresysteme in miteinander kommunizierende Komponenten[2] aufgeteilt werden, so besteht die Notwendigkeit, daß die Komponenten bidirektional miteinander kommunizieren, um die Eigenschaften eines reaktiven Systems zu realisieren.

---

[1] Daß wir anstelle des gebäuchlichen Begriffes “Beobachtungsmechanismus” den Begriff “Reaktionsmechanismus” benutzen, stellt einen Teil der Begriffsklärung dar. Näheres dazu findet sich in Kapitel 2.

[2] In objektorientierten Systemen sind dies die Objekte.



Ein reaktives System

Wir wollen dies an einem Beispiel verdeutlichen (siehe obige Abbildung). Die unterste Komponente “Daten” stellt statistische Werte bereit, welche durch die beiden anderen Komponenten visualisiert werden und zwar als Kalkulationsblatt und als Balkendiagramm. Diese beiden Komponenten sind auch für die Interpretation und Umsetzung der Benutzereingaben zuständig. Verändert der Benutzer die Daten im Kalkulationsblatt, soll sich diese Änderung auch im Balkendiagramm bemerkbar machen. Folglich reicht es nicht aus, wenn die interaktiven Komponenten Zugriff auf die Datenkomponente haben. Vielmehr muß auch die Datenkomponente Zugriff auf die interaktiven Komponenten haben, um diese über Änderungen informieren zu können. Um das System möglichst flexibel zu halten, sollte die Datenkomponente außerdem keine konkrete Kenntnis über die interaktiven Komponenten haben. Nur dann ist es möglich, weitere interaktive Komponenten, z.B. eine Ansicht als Tortendiagramm, problemlos ohne Änderung der Datenkomponente hinzuzufügen.
Für diese Aufgabe wird ein Reaktionsmechanismus benötigt, über welchen sichergestellt werden kann, daß die interaktiven Komponenten auf Änderungen an der Datenkomponente *reagieren* können.

1.3. Zielsetzung

Zielsetzung unserer Arbeit ist die Auswahl oder Erstellung eines Reaktionsmusters für die Verwendung am Arbeitsbereich Softwaretechnik. Dieses Muster soll auch als C++-Implementierung im Rahmenwerk des Arbeitsbereiches verfügbar sein.
Während erster Betrachtungen ergab sich die dringende Notwendigkeit, eine Klarstellung der Begrifflichkeit und Grundlagen von Reaktionsmechanismen herbeizuführen und auf die Probleme beim Einsatz von Reaktionsmechanismen einzugehen.

1.4. Überblick über die Arbeit

Um eine gemeinsame terminologische Grundlage mit dem Leser zu erreichen, stellen wir in Kapitel 2 kurz die Grundlagen der Werkzeug&Material-Metapher vor, sowie die weitere Terminologie, welche für unsere Arbeit wichtig ist. In diesem Kapitel erfolgt auch die Klärung der grundlegenden Begriffe.

In Kapitel 3 erfolgt eine Aufstellung der drei häufigsten Realisierungen des Reaktionsmechanismusses in objektorientierten Softwaresystemen.

Kapitel 4 stellt die Probleme vor, welche im Zusammenhang mit Reaktionsmechanismen auftreten, sowie Lösungsansätze. In diesem Kapitel erarbeiten wir auf der Grundlage dieser Problemstellungen Kriterien, welche von einem Reaktionsmuster erfüllt werden sollten.

Auf Grundlage dieser Analyse und Bewertung entwickeln wir in Kapitel 5 unser neues Muster und stellen außerdem die verschiedenen Implementationsaspekte dieses Musters vor.

Eine Reflektion über die Umstellungsarbeiten an dem C++-Rahmenwerk findet sich in Kapitel 6. Hier beschreiben wir kurz unsere Erfahrungen bei der Umstellungsarbeit.

Wir diskutieren und bewerten die Ergebnisse unserer Arbeit in Kapitel 7, wo wir auch die noch offenen Fragen ansprechen.

Abgeschlossen wird die Arbeit durch zwei Anhänge. In Anhang A finden sich Teile unserer Implementation des neuen Reaktionsmusters, wie es in Kapitel 5 vorgestellt wurde.
Der Anhang B enthält schließlich Handlungsanweisungen für die Umstellung von Anwendungen, welche auf dem klassischen Observer-Muster[3] basieren.

1.5. Sprachliche und graphische Konventionen

Wir benutzen in dieser Arbeit Klassendiagramme, Objektdiagramme, kombinierte Klassen-/Objektdiagramme sowie Interaktionsdiagramme und State-Charts. Wir werden die verwendeten Notationen hier kurz soweit erklären, wie wir sie benutzen werden.
Die State-Charts führen wir bei der ersten Benutzung ein.

Klassendiagramme

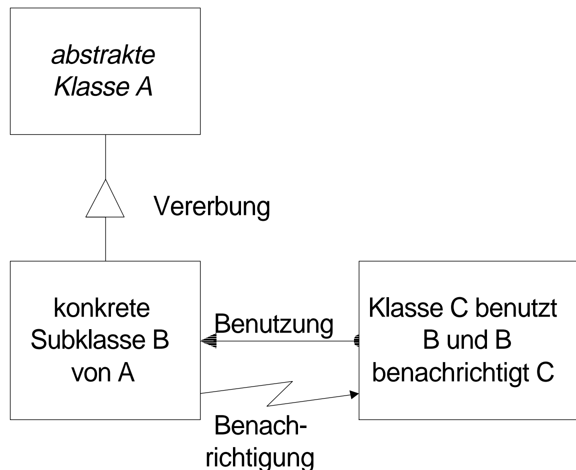
In Klassendiagrammen werden einzelne Klassen dargestellt sowie die Beziehungen zwischen ihnen.
Wir stellen Klassen durch Rechtecke dar. In den Rechtecken steht der Klassenname. Aufgeschobene Klassen sind in *kursiv* gesetzt. Es können drei Typen von Beziehungen dargestellt werden:

- *Vererbungsbeziehungen* werden durch vertikale Pfeillinien dargestellt, wobei der Pfeil in der Mitte der Pfeillinie von der erbenden Klasse zur Oberklasse verweist.
- *Benutztbeziehungen* werden durch in der Regel horizontale Pfeillinien bezeichnet. Der Pfeil am Ende der Pfeillinie verweist von der benutzenden zur benutzten Klasse.

---

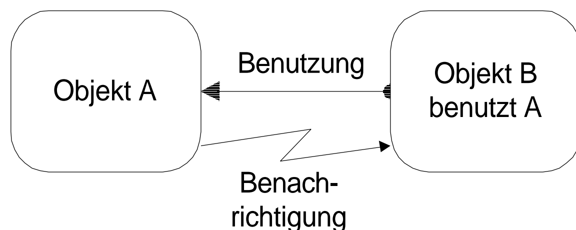
[3] Siehe dazu Kapitel 3.1.

- *Benachrichtigungsbeziehungen* stellen wir durch einen Blitz dar, welcher von der benachrichtigenden zur benachrichtigten Klasse zeigt.



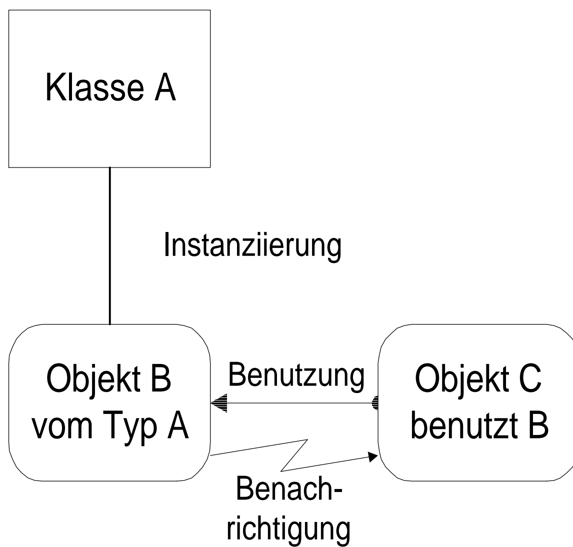
Klassendiagramm

Objektdiagramme
Objektdiagramme beschreiben die Beziehungen zwischen Objekten, also den Instanzen der Klassen.
Objekte werden durch Rechtecke mit abgerundeten Ecken dargestellt. Ähnlich wie bei den Klassendiagrammen können in Objektdiagrammen die Benutzt- und die Benachrichtigungs-beziehungen dargestellt werden.



Objektdiagramm

Klassen-/Objektdiagramme
Neben den Klassen- und Objektdiagrammen gibt es auch noch kombinierte Klassen-/Objektdiagramme, welche es erlauben, auch Vererbungsbeziehungen in Objektdiagrammen darzustellen.
Dazu werden die Objektdiagramme so erweitert, daß eine Vererbung von einer Klasse zu einem Objekt dargestellt werden kann. Dies bedeutet dann, daß dieses Objekt eine direkte oder indirekte Instanz der Oberklasse ist.

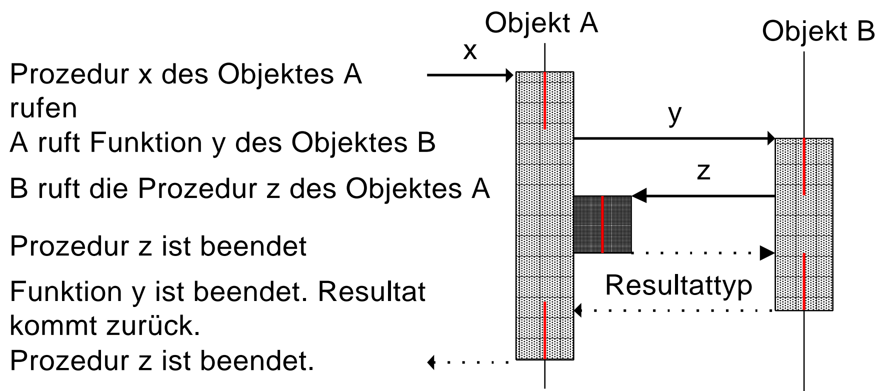


Klassen-/Objektdiagramm

Interaktionsdiagramme

Interaktionsdiagramme veranschaulichen das dynamische Verhalten und das Zusammenspiel von mehreren Objekten. Diese Darstellung ist allerdings immer nur beispielhaft kann daher nicht als vollständige Spezifikation verwendet werden.

Jedes Objekt wird durch eine vertikale Linie dargestellt. Die Zeit verläuft von oben nach unten. Rechtecke auf diesen Linien zeigen an, daß gerade eine Routine des zugehörigen Objektes aktiv ist. Die hellen Linien auf diesen Rechtecken zeigen an, wo sich zur Zeit der Kontrollfluß bewegt. Die durchgezogenen horizontalen Linien stellen Routinenaufrufe dar. Die gestrichelten Linien deuten den Kontrollrückfluß bei Beendigung einer Routine an. Auf ihnen werden Ergebnistypen markiert, wenn es sich um einen Funktionsaufruf handelt. Objektrekursion liegt vor, wenn eine Routine eines Objektes aufgerufen wird, während noch eine andere Routine dieses Objektes aktiv ist. Dies wird durch ein weiteres Rechteck neben der Objektlinie dargestellt. Links neben den Objektlinien kann ein erklärender Text zu der Abbildung stehen.



Interaktionsdiagramm

Code

Code ist immer in der Schriftart `Courier` gesetzt. Dies gilt auch für die Nennung von Codeteilen (wie z.B. Prozedurnamen) im Text. Bei der Vorstellung der bekannten Muster verwenden wir eine Eiffel-ähnliche Notation für die Codeteile.
Unsere Implementierung ist allerdings genauso wie das Rahmenwerk des Arbeitsbereiches in C++ geschrieben.

Sprachliche Konvention

Wir verwenden in dieser Arbeit durchgängig die “maskuline” Form, wenn wir von dem Benutzer etc. sprechen. Dies scheint uns zur Zeit die praktikabelste Lösung zu sein, da alle anderen uns bekannten Formen den Lesefluß doch erheblich stören.[4]

---

[4]Wenn jemandem eine praktikable Lösung bekannt ist, welche auch das weibliche Geschlecht nicht zu kurz kommen läßt, möge er uns dies bitte mitteilen. Wir werden diese Lösung dann in unserer Diplomarbeit verwenden.

2. Einführung der Terminologie sowie in die Werkzeug&Material-Metapher

In diesem Kapitel führen wir die unserer Arbeit zugrundeliegende Terminologie ein, und geben einen Überblick über die Werkzeug&Material-Metapher. Wir haben diese beiden Punkte hier zusammengefaßt, weil ein Großteil der verwendeten Termini aus der Werkzeug&Material-Metapher entstammen. Wir folgen bei der Darstellung der Werkzeug&Material-Metapher [BBS+95], [KGZ93] sowie [Rie95].

2.1. Allgemeine Terminologie

Wir beginnen mit den Begriffen *Mechanismus*, *Muster*, *Realisierung*, *Implementation* und *Codierung*. Diese Begriffe haben für die meisten Leser wahrscheinlich eine intuitive Bedeutung. Wir wollen aber an dieser Stelle erläutern, in welcher konkreten Bedeutung wir diese Begriffe in unserer Arbeit verwenden.

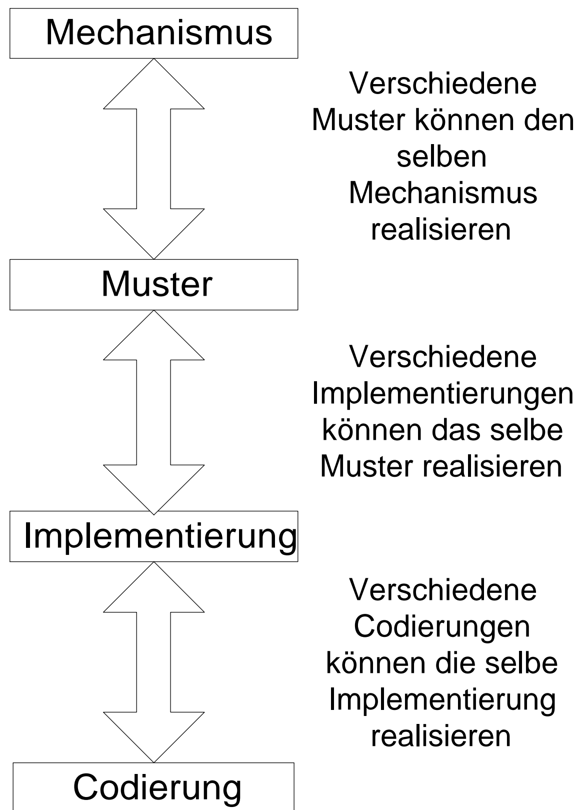
Ein *Mechanismus* ist für uns der allgemeine Aspekt einer Problemlösung. Der Mechanismus beschreibt einen Aufgabenbereich, den eine konkrete Problemlösung abdecken muß. Zu einem Mechanismus sind in der Regel mehrere Problemlösungen denkbar.

Problemlösungen können durch *Muster* beschrieben werden. Sie beschreiben die allgemeine Struktur der Problemlösung und abstrahieren aber noch von der konkreten Implementierung. Wiederum sind in der Regel verschiedene Implementierungen zu einem Muster denkbar.

Eine *Implementierung* gibt die konkrete Einteilung in Klassen und Routinen an. Zu einer Implementierung sind außerdem wieder verschiedene Codierungen denkbar. Codierungen geben den tatsächlich benutzen Quellcode an. Während eine Implementierung in diesem Sinne portabel über Programmiersprachen ist (sofern diese gleiche oder ähnliche Möglichkeiten aufweisen), ist dies bei der Codierung nicht mehr der Fall. Wir befassen uns in unserer Arbeit mit der Codierung nur am Rande.

Muster sind *Realisierungen* von Mechanismen, Implementierungen sind Realisierungen von Mustern und Codierungen können schließlich als Realisierungen von Implementierungen angesehen werden.

Diese Struktur ist in der folgenden Abbildung wiedergegeben.



Mechanismus, Muster, Implementierung, Codierung

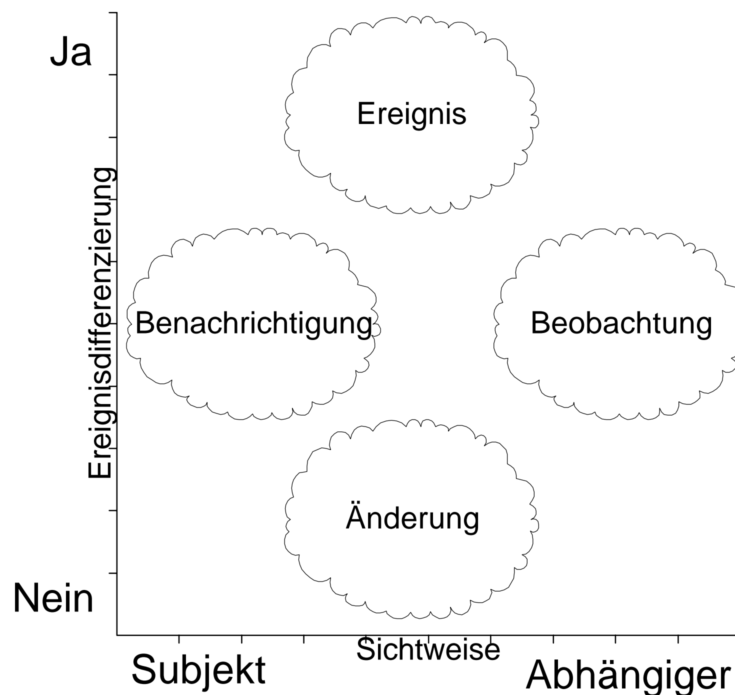
Klassen beinhalten Variablen und Routinen, wobei sich die Routinen weiter in Prozeduren und Funktionen aufteilen lassen. Prozeduren verändern den Objektzustand und liefern kein Resultat. Funktionen liefern ein Resultat und laufen seiteneffektfrei (sie verändern den Objektzustand nicht) (nach [Mey90]).

Des weiteren wollen wir die Begrifflichkeit von Reaktionsmechanismen klarstellen:
Ein *Reaktionsmechanismus* stellt die Möglichkeit dar, Komponenten abstrakt über Änderungen an einer anderen Komponente zu informieren, so daß diese informierten Komponenten auf diese Änderungen *reagieren* können.
Für die Namensgebung *Reaktionsmechanismus* sind folgende Gründe ausschlaggebend:

- Der Mechanismus dient einer Komponente zur *Reaktion* auf Veränderungen einer anderen Komponente.
- Diese Reaktionsmöglichkeit ist Grundvoraussetzung für die Konstruktion von *reaktiven* Softwaresystemen.

Die informierten Komponenten nennen wir auch *abhängige Komponenten*, da ihr Zustand von dem Zustand der informierenden Komponente abhängt.
Leider gibt es im deutschen Sprachgebrauch kein griffiges Wort für die informierende Komponente, welche das Gegenstück zu der abhängigen Komponente darstellt. Wir wählen daher aus der englischen Fachterminologie entlehnt den Begriff *Subjekt*. Die abhängige Komponente ist also vom Subjekt abhängig.

Die konkreten Reaktionsmuster lassen sich jetzt weiter klassifizieren, indem man einen zweidimensionalen Merkmalsraum gemäß der folgenden Abbildung aufspannt.



Reaktionsmechanismen

Reaktionsmechanismen können demnach danach unterschieden werden, ob sie in der Sichtweise vom Subjekt oder vom Abhängigen ausgehen und ob sie verschiedene Ereignisse differenzieren oder nur von allgemeiner Änderung sprechen. Es gibt hier also vier Fälle:

a) Bei der Änderungs-Benachrichtigung benachrichtigt das Subjekt die abhängige Komponente von einer Änderung seines Zustandes.
b) Bei der Änderungs-Beobachtung beobachtet die abhängige Komponente das Subjekt und nimmt dabei nur allgemeine Änderungen des Subjektzustands zur Kenntnis.
c) Bei der Ereignis-Benachrichtigung benachrichtigt das Subjekt die abhängige Komponente und informiert diese dabei über die Art der Änderung.
d) Bei der Ereignis-Beobachtung beobachtet die abhängige Komponente das Subjekt und nimmt die speziellen Zustandsänderungen an dieser wahr.

Diese Klassifizierung greift nur bei der konzeptionellen Sichtweise auf den Mechanismus und nicht mehr bei der konkreten Implementierung. Wir werden in Kapitel 3 sehen, daß bei den bekannten Reaktionsmustern immer die Benachrichtigung vom Subjekt ausgeht.

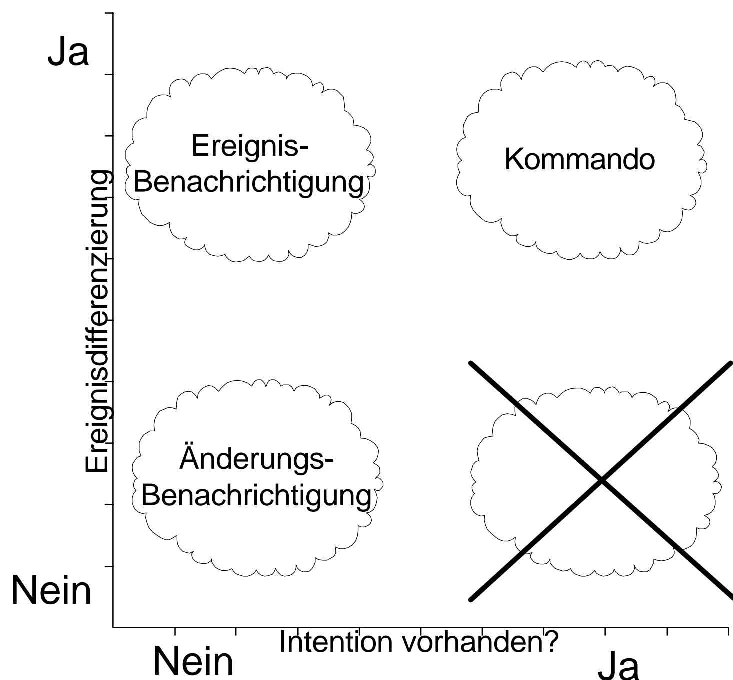
Bei der Beobachtungssichtweise (Punkt b und d) nennt man das Subjekt auch *Beobachteten* und den abhängigen *Beobachter* und geht implizit davon aus, daß der Beobachter die aktive und der Beobachtete die passive Komponente ist.
Bei der Benachrichtigungssichtweise (Punkt a und c) spricht man auch vom *Benachrichtiger* und dem *Benachrichtigten* und geht implizit davon aus, daß der Benachrichtiger die aktive und der Benachrichtigte die passive Komponente ist.
Damit sind einerseits Subjekt, Beobachteter und Benachrichtiger äquivalent und anderer-

seits Abhängiger, Beobachter und Benachrichtigter. Je nach Sichtweise ist mal das Subjekt aktiv und die abhängige Komponente passiv (Benachrichtigung) oder genau umgekehrt (Beobachtung).

Es sei hier noch darauf hingewiesen, daß die Klassifikation nach der Sichtweise eine rein konzeptionelle ist. Im konkreten Fall gehören Beobachter und Beobachteter bzw. Benachrichtiger und Benachrichtigter immer zusammen. Der Eine kann ohne den Anderen nicht existieren. Die Einteilung in allgemeine Änderungen und unterscheidbare Ereignisse ist dagegen tiefergreifend. Die Entscheidung für das Eine oder das Andere hat auch immer Einfluß auf den konkreten Mechanismus und muß in der Implementation berücksichtigt werden.

Während abhängige Komponenten von vornherein bezüglich des Reaktionsmechanismusses eine klar festgelegte Intention haben[5], verhält es sich beim Subjekt nicht so. Dort lassen sich die folgenden beiden Fälle unterscheiden:

- Keine Intention: Das Subjekt teilt abhängigen Komponenten zwar Änderungen mit, verfolgt mit dieser Benachrichtigung aber keine weiteren Ziele.
- Intention: Bei Änderungen erteilt das Subjekt den abhängigen Komponenten *Kommandos*, auf welche die abhängigen Komponenten reagieren. Hierbei verfolgt das Subjekt das abstrakte Ziel, daß die abhängigen Komponenten eine bestimmte (wenn auch dem Subjekt nicht konkret bekannte) zum Kommando passende Aktion ausführen.



Reaktionsmechanismen aus Subjektsicht mit und ohne Intention

---

[5]Nämlich die Reaktion auf Subjektänderungen in definierter Art und Weise um ihren vom Subjekt abhängigen Zustand konsistent mit dem Subjektzustand zu halten.

Ein beliebiges System von abhängigen Objekten und Subjekten wollen wir im folgenden *reaktive Komponente* nennen.

Wir wollen nun die beiden Begriffe Zustandsänderung und Ereignis konkretisieren. Wir verwenden dazu allgemeine Definitionen aus dem Brockhaus:

Def. Zustandsänderung (nach [Bro88])
"Änderung des thermodynamischen Zustandes eines Stoffes oder thermodynamischen Systems, verursacht durch Änderung einer Zustandsgröße. Eine Zustandsänderung kann reversibel oder irreversibel sein, [...]. Quasistatische Zustandsänderungen verlaufen so langsam, daß das System in jedem Augenblick als im Gleichgewicht befindlich betrachtet werden kann."

Wir adaptieren diese Definition für unseren Zusammenhang und erhalten so:

Unter einer *Zustandsänderung* verstehen wir die Änderung eines *Objektzustandes*, verursacht durch die Änderung einer *Zustandsgröße*. Zustandsänderungen können *reversibel* oder *irreversibel* sein.

Wir beschränken uns damit auf die Zustände von Objekten als Instanzen von Klassen. Zustandsgrößen sind nicht unbedingt mit Instanzvariablen gleich zu setzen. Vielmehr hängt es von der abstrakten Modellierung des Objektes ab, was unter einer Zustandsgröße zu verstehen ist. Eine Zustandsgröße wird aber immer durch eine oder mehrere Instanzvariablen gebildet.[6] Wir sprechen von *reversiblen Zustandsänderungen*, wenn das Objekt Prozeduren anbietet, welche der Wiederherstellung des vorigen Zustandes dienen. *Irreversible Zustandsänderungen* liegen vor, wenn solche Prozeduren nicht vorhanden sind.

Def. Ereignis (nach [Bro88])
"Physik: physikalischer Vorgang, dessen räumliche und zeitliche Dauer vernachlässigbar klein sind; in der Theorie dargestellt als ein Raum-Zeit-Punkt."

Wir wandeln auch diese Definition für unsere Zwecke etwas ab:

Ein *Ereignis* ist ein Vorgang in einem Softwaresystem, welches von einem Objekt ausgeht und von anderen Objekten wahrgenommen werden kann. Diese Objekte können auf Ereignisse reagieren. Räumliche und zeitliche Dauer eines Ereignisses sind vernachlässigbar klein. Deshalb kann es als Raum-Zeit-Punkt dargestellt und verstanden werden.

Wir beschränken uns bei der obigen Definition auf Objekte eines objektorientierten Softwaresystems. In dieser Definition finden sich direkt Anknüpfungspunkte zu dem bisher über Reaktionsmechanismen Gesagten. Reaktionsmechanismen dienen genau der Vermittlung von Ereignissen zwischen Objekten. Wir haben oben gesehen, daß eine Differenzie-

---

[6] Dies umfaßt ggfs. auch Klassenvariablen, wenn diese für das Objekt zugreifbar sind.

rung nach Ereignissen in Reaktionsmechanismen nicht notwendig ist. In diesem Fall kann man von der Existenz nur eines allgemeinen Ereignisses “Veränderung hat stattgefunden” ausgehen, während bei ausdifferenzierten Ereignissen von unterscheidbaren Ereignissen ausgegangen wird, welche bereits eine Klassifizierung der Veränderung enthalten, z.B. “Element wurde selektiert”.

Ereignisse werden durch *Signale* oder *Nachrichten* übermittelt:

Def. Signal (nach [Bro88])
“Allg.: Zeichen mit vereinbarter Bedeutung.”

Die vereinbarte Bedeutung kann dabei offenbar sehr allgemein oder auch sehr speziell sein. Wir verwenden den Begriff Signal, wenn Ereignisse allgemein bekannt gemacht werden sollen. Signale beziehen sich dabei nur auf das Ereignis selbst und nicht auf deren Kontext. Kontextinformationen werden nach unserem Sprachgebrauch durch Nachrichten übermittelt.

Def. Nachricht (nach [Bro88])
“Zusammenstellung von Zeichen oder Zuständen, die zur Übertragung von Information dient. [...]”

Zusätzlich zu den Signalen enthalten Nachrichten Kontextinformationen, die den Kontext des Ereignisses konkreter beschreiben.

Beispiel
Stellen wir folgendes *Ereignis* vor:
“Es hat eine Bundestagswahl stattgefunden und die Partei X hat gewonnen.”

Dieses Ereignis kann durch *Signale* unterschiedlicher Art übermittelt werden:
- Sehr allgemein: “Es hat sich etwas ereignet.”
- Konkreter: “Es hat eine Wahl stattgefunden.”
- Sehr konkret: “Es hat eine Bundestagswahl stattgefunden.”

Man kann auch durch eine *Nachricht* von demselben Ereignis informiert werden:
- “Bundestagswahl hat stattgefunden: Partei X hat gewonnen.”

Im Gegensatz zum üblichen Sprachgebrauch gehen wir davon aus,
- daß Ereignisse, wie in der obigen Definition, keine zeitliche Ausdehnung haben,
- daß die Übermittlung von Ereignissen durch Signale oder Nachrichten erst dann erfolgt, wenn das Ereignis abgeschlossen ist,
- daß die Übermittlung eines Ereignisses durch Signale oder Nachrichten ebenso wie die Ereignisse keine zeitliche Ausdehnung haben,
- und daher das Auftreten und die Beendigung eines Ereignisses sowie das Versenden und das Empfangen von Signal bzw. Nachricht gleichzeitig geschieht.

In einem realen Softwaresystem benötigt das Übermitteln eines Signals bzw. einer Nachricht Zeit. Wir halten dennoch an unserer Forderung fest, zumindest konzeptionell für den Sprachgebrauch und die Verwendung realer Implementierungen von Reaktionsmechanismen auch weiterhin davon auszugehen, daß keine zeitliche Ausdehnung vorhanden ist.

Des weiteren bekommen mehrere Empfänger desselben Signals bzw. derselben Nachricht in einem realen System nacheinander das Signal bzw. die Nachricht zugestellt. Konzeptionell gehen wir jedoch davon aus, daß dies gleichzeitig geschieht.

Während wir eine Trennung zwischen Signal und Nachricht für sinnvoll halten, so ist eine entsprechende Trennung auf Ebene der zugehörigen Verben *Signalisieren* und *Benachrichtigen* sprachlich nicht durchzuhalten. Dies liegt sicher auch daran, daß wir in der Regel davon ausgehen, daß wir *etwas signalisieren*, jedoch *jemanden benachrichtigen.* Aus diesem Grund benutzen wir die Verben *Signalisieren, Benachrichtigen* und *Informieren* synonym für die Übermittlung eines Signals oder einer Nachricht.

Mit den oben gegebenen Definitionen und den eingeführten Begriffen können wir jetzt sagen:

Der Objektzustand eines Objektes A kann von dem Objektzustand eines anderen Objektes S abhängen. Dann ist A der Abhängige vom Subjekt S. Bei jeder Zustandsänderung des Subjektes kann das abhängige Objekt darauf reagieren. Damit dies möglich wird, wird mit jeder Zustandsänderung in einem Subjekt ein Ereignis assoziiert, welches das abhängige Objekt wahrnehmen und so darauf reagieren kann. Das Subjekt S benachrichtigt das Objekt A, wenn die Zustandsänderung vollzogen ist.

2.2. Leitbild und die Werkzeug&Material-Metapher

Bei der Objektorientierung geht es zunächst darum, Objekte des Gegenstandsbereiches in Objekte des Entwurfs abzubilden. Damit ist aber noch keine Aussage darüber gemacht, welche Objekte des Gegenstandsbereiches in Objekte des Entwurfs abzubilden sind und wie dies zweckmäßig zu geschehen hat. Wir benötigen ein Leitbild, um die notwendige Orientierung und Gestaltungsperspektiven für den Entwurf von Softwaresystemen zu erhalten. “Wir wählen ein Leitbild, das uns hilft, Softwaresysteme menschengerecht und aufgabenangemessen zu gestalten: Den Arbeitsplatz für qualifizierte menschliche Tätigkeiten. Dies kann die Büroumgebung, der Schreibtisch oder eine Werkbank sein.” [BBS+95]. Innerhalb dieses Leitbildes benötigen wir weiterhin Metaphern, welche es uns erleichtern, die wichtigen Objekte des Gegenstandsbereiches zu erkennen und adäquat in Objekte des Softwaresystems abzubilden. Wir verwenden die eng verwobenen Metaphern von *Werkzeug* und *Material* und sprechen auch von der Werkzeug&Material-Metapher.

Wir beobachten den Gegenstandsbereich zuerst unter dem Blickwinkel der dort bearbeiteten Materialien. Dies können z.B. Rechnungen, Ordner oder auch abstraktere Dinge wie Zinssätze sein. Diese Materialien werden im Gegenstandsbereich auf eine bestimmte Art und Weise mit speziellen Werkzeugen bearbeitet. Betrachtet man einen Arbeitsplatz ohne EDV-Unterstützung, so werden meist relativ einfache Werkzeuge, wie Bleistifte, Locher, Anspitzer etc. benutzt. Wir bilden diese gefundenen Werkzeuge und Materialien nicht direkt und unverändert in die Objekte unseres Softwaresystemes ab. Vielmehr ist dieser Prozeß immer begleitet von Abstraktionsprozessen, um einen Entwurf zu finden, welcher den Anforderungen des Gegenstandsbereiches ebenso gerecht wird wie den Forderungen auf der softwaretechnischen Ebene (Wiederverwendung, Wartbarkeit etc.). Außerdem ist es in der Praxis einfach nicht möglich, einen Bleistift mit all seinen Einsatzmöglichkeiten zu modellieren.

Die Werkzeug&Material-Metapher läßt sich als Sammlung von *Analysemustern* begreifen, welche musterhaft vorgeben, worauf bei der Analyse des Gegenstandsbereiches zu achten

ist. Mit Hilfe dieser Analyse-Muster wird eine Handlungsanweisung für das Auffinden "guter" Objekte angegeben. In einem weiteren Schritt wird die Werkzeug&Material-Metapher um eine Reihe allgemeiner sowie spezifischer *Entwurfsmuster* erweitert, um auch Anleitungen für die Überführung der Analyseergebnisse in den Entwurf zu erhalten.

2.2.1. Materialien

Materialien werden entsprechend den Arbeitsgegenständen des Gegenstandsbereiches modelliert. Dabei betrachten wir den charakteristischen Umgang mit diesen Materialien. Unter dem charakteristischen Umgang verstehen wir die Art und Weise, wie der Benutzer Informationen an den Materialien erkennen kann und wie er Veränderungen an diesen Materialien vornimmt. Wir sprechen in diesem Zusammenhang auch von *fachlichen Umgangsformen*. Typische Umgangsformen für eine Rechnung könnten etwa das Hinzufügen einer Position oder das Ablesen des Gesamtrechnungsbetrages sein.
Materialien werden unabhängig von den Werkzeugen entworfen, um sie nicht auf die Benutzung mit bestimmten Werkzeugen einzuschränken. Materialien enthalten nur die fachlichen Umgangsformen, welche von der Repräsentation auf dem Bildschirm unabhängig sind. Zu ihrer Anzeige und Bearbeitung am Bildschirm müssen immer Softwarewerkzeuge herangezogen werden.

2.2.2. Werkzeuge

Werkzeuge zeichnen sich in erster Linie dadurch aus, daß sie zur Bearbeitung von Materialien benutzt werden. In [BBS+95] wird darauf hingewiesen, daß während der Analyse gefundene Werkzeuge[7] meist nicht unverändert in Softwarewerkzeuge überführt werden können. Im Gegenstandsbereich stoßen wir oft auf sehr universelle Werkzeuge, wie z.B. Bleistifte. Bleistife ermöglichen uns fast alle Formen der Textverarbeitung, Grafik etc. Ein so universelles Werkzeug in ein softwaretechnisches Objekt abzubilden, ist schlechterdings unmöglich.

Neben diesen sehr universellen Werkzeugen gibt es weiterhin auch eine Reihe von Werkzeugen, welche nur eine Aufgabe erfüllen und daher besser durch eine Operation als durch eine ganze Werkzeugklasse abzubilden sind. Ein typisches Beispiel für so ein Werkzeug ist z.B. ein Kopierer.
Da eine direkte Abbildung von Werkzeugen des Gegenstandsbereiches in Softwarewerkzeuge wie oben gezeigt nicht möglich ist, konzentrieren wir uns beim Entwurf der Softwarewerkzeuge auf die Handhabungen und Tätigkeiten, die sich im Arbeitszusammenhang als (lästige) Routinen herauskristallisiert haben. So kommen wir z.B. von der allgemeinen Form der Textverarbeitung zu speziellen Rechnungseditoren, welche der Bearbeitung und Anzeige von Rechnungen dienen.
Die von den Werkzeugen zur Verfügung gestellten Dienstleistungen werden vom Benutzer durch Interaktion an der Benutzungsoberfläche ausgelöst. Diese Benutzeraktion wird daraufhin vom Werkzeug in eine entsprechende Operation auf dem bearbeiteten Material umgesetzt.

---

[7]Hier sind die tatsächlich vorgefundene Werkzeuge wie Bleistifte und Locher gemeint.

2.2.3. Trennung von Funktion und Interaktion

In der Werkzeug&Material-Metapher findet sich ein wichtiges Architekturprinzip von interaktiven Softwaresystemen - die Trennung von Interaktion und Funktion. Für Interaktion sind nur Werkzeuge zuständig und in Materialien findet sich nur Funktionalität.
Werkzeuge können neben der Interaktion auch Funktionalität enthalten, welche über die Funktionalität der Materialien hinausgeht. Die Trennung von Interaktion und Funktion setzt sich daher auch innerhalb der Werkzeuge fort. Diese werden in einen funktionellen und interaktiven Teil getrennt.
Der funktionelle Teil kann so unabhängig vom interaktiven Teil entworfen werden. Interaktive und funktionelle Teile können weitgehend unabhängig voneinerander variiert werden. Durch einfaches Austauschen der interaktiven Teile kann zwischen Fensteroberflächen oder GUI-Rahmenwerken gewechselt werden, ohne daß an den funktionellen Teilen Änderungen notwendig werden.
Zusammenfassend stellen wir noch einmal die Aufgaben von Funktion und Interaktion gegenüber:

Funktion: Was bewirkt ein Software-Werkzeug fachlich?
Interaktion: Wie wird ein Software-Werkzeug gehandhabt?

Für konkrete Werkzeuge kommt noch die Präsentation hinzu, unter der wir die konkrete Darstellung des Werkzeuges und des bearbeiteten Materials verstehen. Die Präsentation ist abhängig von der konkreten Benutzungsoberfläche.

Präsentation: Wie wird ein Software-Werkzeug und das bearbeitete Material dargestellt?

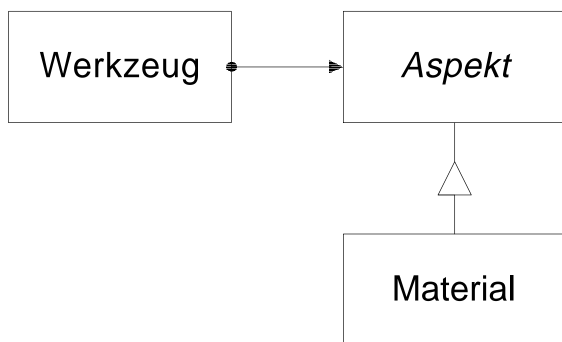
Neben den hier vorgestellten Werkzeugen, wird die Werkzeug&Material-Metapher um Automaten erweitert: “Automaten sind vorrangig ‘im Hintergrund’ laufende, aktive Komponenten, die eine länger andauernde Routineaufgabe erledigen” [BBS+95]. Automaten sind Werkzeugen ähnlich. Sie bieten ebenfalls einen funktionellen und einen interaktiven Teil. Über den interaktiven Teil kann der Automat eingestellt werden. Die Durchführung der Aufgabe erfolgt beim Automaten im Gegensatz zum Werkzeug automatisch. Für unseren Zusammenhang können wir Automaten mit Werkzeugen gleichsetzen.
Nach dieser Erweiterung können wir die Werkzeug&Material-Metapher auch WAM-Metapher nennen, wobei das *W* für Werkzeug, das *M* für Material und das *A* wahlweise für Automaten oder die im nächsten Kapitel noch einzuführende Aspektklassen stehen kann.

2.3. Zusammenhang von Werkzeug und Material: Aspekte

Wir haben in den vorigen beiden Abschnitten gesehen, daß Materialen unabhängig von den auf ihnen arbeitenden Werkzeugen entworfen werden sollten. Außerdem sollen sowohl Werkzeuge auf verschiedenen Materialien anwendbar sein, wie auch mehrere Werkzeuge dasselbe Material bearbeiten können.
Dennoch ist eine Kombination von Werkzeug und Material nicht beliebig. Mögliche Kombinationen müssen festgelegt werden, ebenso wie der dann entstehende Arbeitszusammenhang zwischen Werkzeug und Material.

Schließlich läßt sich erst durch den Arbeitszusammenhang endgültig entscheiden, was als Werkzeug und was als Material anzusehen ist. “Beim Sägen ist eine Säge Werkzeug, beim Wechseln des Sägeblattes ist sie Material für das Werkzeug Schraubendreher, mit dem das Blatt gespannt wird.” [BBS+95].

Diese Festlegung des Arbeitszusammenhanges treffen wir in *Aspektklassen*. Die möglichen Kombinationen von Werkzeugen und Materialien manifestieren sich in den Beziehungen, die Werkzeuge und Materialien zu den Aspektklassen haben. Es ist sogar so, daß Werkzeuge und Materialien immer nur indirekt über Aspektklassen in einer Beziehung zueinander stehen. Der Begiff “Aspektklasse” signalisiert, daß sich ein Werkzeug nur auf bestimmte Eigenschaften (Aspekte) des bearbeiteten Materials bezieht. Aspektklassen beschreiben jene Operationen, welche das Werkzeug benutzt und das Material daher bereitstellen muß. Folgende Definition stellt noch einmal die Bedeutung von Aspektklassen heraus:
“Eine Klasse ist in einem durch eine Aspektklasse bezeichneten Arbeitszusammenhang Werkzeug, wenn sie diese Aspektklasse benutzt; eine Klasse ist Material, wenn sie Unterklasse dieser Aspektklasse ist.” [BBS+95].



Klassendiagramm Materialbenutzung über Aspektklassen

2.4. Die Architektur von Werkzeugen

Wir haben in den letzten Abschnitten Werkzeuge, Materialien, Automaten und Aspektklassen vorgestellt. Für unsere weitere Arbeit konzentrieren wir uns auf *spezielle Entwurfsaspekte von Werkzeugen.*
Einem Werkzeug kommen die drei Aufgaben Funktion, Interaktion und Präsentation zu. Dazu teilen wir jedes Werkzeug in eine Interaktionskomponente (IAK) und eine Funktionskomponente (FK). Die Präsentation wird durch die Verbindung der Interaktionskomponente mit speziellen Interaktionstypen[8] (IAT) realisiert, welche die Funktionalität der Benutzungsoberfläche kapseln.

---

[8]*Interaktionstypen* realisieren bestimmte *Interaktionsformen*, wie z.B. eine “Eins aus N”-Auswahl. Für jede Interaktionsform kann es verschiedene Interaktionstypen geben. So kann die besagte “Eins aus N”-Auswahl durch eine “Listbox” oder durch eine “Radio-Group” realisiert werden.

Jede Interaktionskomponente, jede Funktionskomponente und jeder Interaktionstyp werden durch jeweils eine Klasse realisiert. In der Regel benutzt eine Interaktionskomponente eine Funktionskomponente sowie mehrere Interaktionstypen, mit deren Hilfe die Interaktionskomponente auf Benutzeraktionen reagieren kann. Dieser Sachverhalt wird durch die folgende Abbildung veranschaulicht:

Benutzbeziehungen der Interaktionskomponente

Zwischen Interaktionskomponente und Interaktionstypen einerseits sowie zwischen Interaktionskomponente und Funktionskomponente andererseits besteht ein Abhängigkeitsverhältnis, welches über die Benutzt-Beziehung hinausgeht, weil sichergestellt werden muß, daß Abhängigkeiten zwischen den Komponenten eingehalten werden:

- Die Interaktionskomponente benutzt nicht nur ihre Interaktionstypen, sondern muß auch auf Benutzeraktionen reagieren. Wird ein Interaktionstyp durch den Benutzer verändert, indem z.B. ein Listenelement selektiert wird, so muß die Interaktionskomponente hiervon Kenntnis erlangen, um eine entsprechende Aktion ausführen zu können.
- Ähnlich verhält es sich zwischen Interaktionskomponente und Funktionskomponente. Ruft die Interaktionskomponente eine Operation der Funktionskomponente auf, so kann dies zu einer Veränderung in der Funktionskomponente führen. Die Interaktionskomponente muß auf diese Veränderung reagieren können.

Es gibt generell drei Möglichkeiten, das Zusammenspiel von IAK, IATs und FK zu organisieren:

- *Bidirektionale Benutztbeziehungen*: Die einzelnen Komponenten benutzen sich gegenseitig und rufen gegenseitig beieinander Methoden auf. Als Konsequenz können IATs nicht ohne Anpassung für weitere IAKs wiederverwendet werden. Ebenso ist die FK sehr speziell auf eine IAK zugeschnitten (was die Frage aufkommen läßt, warum dann noch Interaktion und Funktion voneinander getrennt sein sollen, da die FK ihre IAK konkret kennt). Sollte eine FK von mehreren IAKs benutzt werden, so müßte diese FK speziell an alle benutzenden IAKs angepaßt sein und die Anzahl der benutzenden IAKs dürfte nicht variieren.
- *Polling*: Die Interaktionskomponente fragt in regelmäßigen Abständen IATs und FK ab, um auf Änderungen reagieren zu können. Außer der Ineffektivität dieses Verfahrens kommt als weitere Konsequenz hinzu, daß die FK nur von einer IAK gepollt werden kann, da sich der Hauptkontrollfluß in dieser einen IAK befindet[9].

---

[9]Bei der Realisierung von Interaktionstypen in modernen grafischen Benutzungsoberflächen, wird der Ansatz des Polling bereits von dem API (Application Programming Interface) der Benutzungsoberfläche verhindert. Der Hauptkontrollfluß befindet sich in der Benutzungsoberfläche und wird nur kurzzeitig bei Benutzeraktionen an das Werkzeug abgegeben.

- *Reaktionsmechanismus*: Die IAK wird von IATs bzw. ihrer FK von Änderungen unterrichtet ohne daß IATs und FK die IAK konkret kennen[10].
  Die IAK muß die Änderungen an der FK nicht voraussehen, und kann daher unabhängig von der konkreten FK bleiben. Die IATs müssen ihre IAK ebenfalls nicht konkret kennen und können daher auch für andere IAKs verwendet werden.

Aus den aufgeführten Gründen kommen bidirektionale Benutztbeziehungen und Polling nicht für die Konstruktion von Werkzeugen nach der Werkzeug&Material-Metapher in Frage. Wir benötigen demnach einen Reaktionsmechanismus. Die folgende Abbildung veranschaulicht den Einsatz eines Reaktionsmechanismusses:

Benutzungs- und Reaktionsbeziehungen in Werkzeugen

Es gibt eine ganze Reihe von Möglichkeiten, diesen Reaktionsmechanismus zu realisieren. Wir stellen im dritten Kapitel exemplarisch drei verschiedene Muster vor, welche alle zur Realisierung eines Reaktionsmechanismus herangezogen werden können. Allerdings haben sie jeweils unterschiedliche Schwerpunkte.

2.5. Konstruktion komplexer Werkzeuge

Würde man Werkzeuge allein mit Hilfe der oben beschriebenen Trennung von Interaktions- und Funktionskomponente konstruieren, so könnte man entweder nur relativ einfache Werkzeuge erstellen oder aber die Interaktions- und Funktionskomponente würden unhandlich groß. Es fehlt eine Konstruktionsanleitung für den Bau komplexer Werkzeuge.
Die Konstruktionsanleitung für komplexe Werkzeuge nach der Werkzeug&Material-Metapher erfüllt zwei softwaretechnische Grundanforderungen:

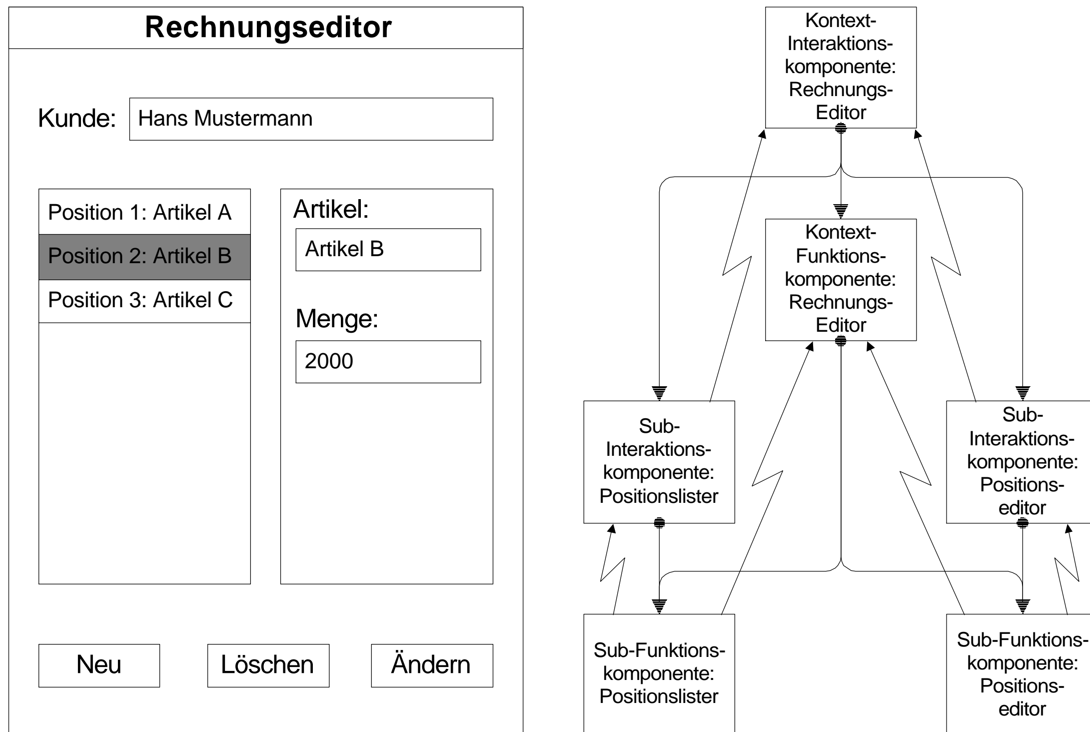
- Teile die Funktionalität so in Komponenten resp. Klassen auf, daß diese Komponenten noch übersichtlich und handhabbar sind.
- Wähle die Einteilung in Komponenten so, daß sich die einzelnen Komponenten mit möglichst geringem Aufwand wiederverwenden lassen.

Komplexe Werkzeuge werden nach der Werkzeug&Material-Metapher aus *Subwerkzeugen* zusammengesetzt.
Beispielsweise könnte ein Rechnungseditor nicht nur Funktionalität zum Editieren einzelner Positionen anbieten, sondern zusätzlich die Möglichkeit, die zu editierende Rechnungsposition aus einer Positionsliste auszuwählen. In diesem Beispiel können der Positionslister und der Positionseditor jeweils als einzelne Subwerkzeuge modelliert werden.

---

[10] Auch wenn wir bei der bidirektionalen Benutzung und dem Polling von Reaktionsmöglichkeiten sprechen, so stellen sie doch keine Reaktionsmechanismen dar. Nach der unter 2.1. gegebenen Definition muß die Information über die Änderung *abstrakt* an die abhängige Komponente gegeben werden.

Diese Subwerkzeuge können in beliebigen anderen komplexen Werkzeugen wiederverwendet werden, da die Einzelheiten der Kommunikation zwischen den einzelnen Subwerkzeugen in sogenannten Kontextwerkzeugen realisiert werden. Das Kontextwerkzeug hat die Aufgabe seine Subwerkzeuge einzubetten und die Kommunikation zwischen ihnen zu regeln. Außerdem kann im Kontextwerkzeug Funktionalität angesiedelt werden, welche nicht in den Subwerkzeugen enthalten ist, aber trotzdem für das Werkzeug benötigt wird.
Die folgende Abbildung demonstriert die Konstruktion komplexer Werkzeuge anhand des oben eingeführten Rechnungseditors.



Rechnungseditor: Ein Werkzeug mit Subwerkzeugen

Auf der linken Seite ist eine Bildschirmhardcopy[11] des komplexen Werkzeuges Rechnungseditor[12] abgebildet. Links in der Hardcopy ist das Subwerkzeug Positionslister zu sehen, auf der rechten Seite ist das Subwerkzeug Positionseditor dargestellt.
Links kann eine Position durch Doppelklick ausgewählt werden. Eine so selektierte Position kann dann auf der rechten Seite angesehen und bearbeitet werden.

---

[11]Es ist leicht zu sehen, daß es sich hierbei nicht wirklich um eine Hardcopy eines realen Werkzeuges handelt. Dies ist nur eine Skizze eines noch nicht existierenden Werkzeuges. Zusammen mit der Erklärung der geplanten Funktionalität und Handhabung stellt dies nach der Werkzeug&Material-Metapher eine System-Vision dar.

[12]Das hier dargestellte Werkzeug ist stark vereinfacht. In einem realen Werkzeug würde man weitere Daten der Rechnung und Positionen bearbeiten.

Nach der Bearbeitung können die Änderungen durch Betätigung des Ändern-Knopfes übernommen werden. Wurde der Artikel geändert, so wird auch die Liste auf der linken Seite angepaßt.

- Durch den Löschen-Knopf kann die selektierte Position gelöscht werden.
- Durch den Neu-Knopf wird eine neue Position in die Liste eingefügt und gleichzeitig im Editor zur Bearbeitung angeboten.
- Über den beiden Subwerkzeugen ist ein Eingabefeld für den Kundennamen zu sehen.

Das Feld für den Kundennamen und die drei Knöpfe gehören zum Kontextwerkzeug[13].

In dem rechten Teil der Abbildung haben wir die Struktur des Gesamtwerkzeuges dargestellt:
Jede Interaktionskomponente benutzt ihre Funktionskomponente. Daneben benutzt die Kontext-Funktionskomponente ihre Sub-Funktionskomponenten und die Kontext-Interaktionskomponente benutzt ihre Sub-Interaktionskomponenten. Entsprechend zu den Benutztbeziehungen existiert jeweils eine Reaktionsbeziehung. Die Interaktionstypen haben wir an dieser Stelle der Übersichtlichkeit wegen weggelassen. Jede Interaktionskomponente benutzt ihre Interaktionstypen und reagiert auf Änderungen in diesen.

Innerhalb der Werkzeugkonstruktion existiert zu jeder Benutztbeziehung zwischen Interaktionstypen, Interaktionskomponente und Funktionskomponente eine entsprechende Reaktionsbeziehung.

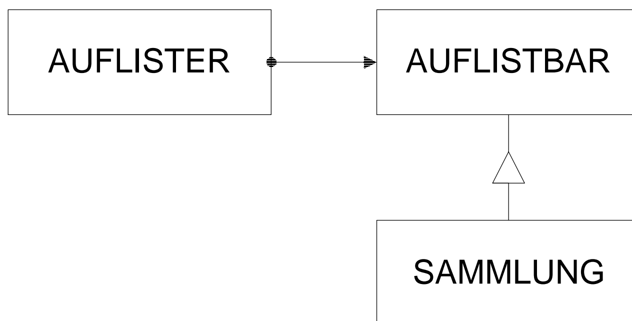
[13] Um ein begriffliches Gegenstück zu den Subwerkzeugen zu haben, wird gelegentlich auch von Hauptwerkzeug gesprochen. Der Begriff Kontextwerkzeug ist unserer Meinung nach aber geeigneter, da das Kontextwerkzeug gerade die Aufgabe hat, seine Subwerkzeuge in einen gemeinsamen *Kontext* zu stellen.

3. Verschiedene Reaktionsmechanismen im Vergleich

In diesem Kapitel werden verschiedene Varianten des Reaktionsmechanismusses vorgestellt: das Observer-Muster (als Beobachtermechanismus), das Command-Muster (als Kommandomechanismus) und das Event-Notification-Muster (als Ereignismechanismus). Wir verwenden die in [GHJ+94] und [Vli95] eingeführte Struktur zur Beschreibung von Mustern, da diese sich um eine einheitliche Form der Darstellung von Mustern verdient gemacht hat und eine breite Akzeptanz genießt.

3.1. Ein Beispiel

Als durchgängiges Beispiel verwenden wir ein Werkzeug, welches als Teil eines größeren Beispiels innerhalb eines Seminars am Arbeitsbereich Softwaretechnik entstanden ist. Das Beispiel ist ein einfaches Werkzeug "Auflister", welches eine Liste von Einträgen auf dem Bildschirm anzeigt. Einzelne Einträge können ausgewählt und die Liste kann verändert werden.
Nach der Werkzeug&Material-Metapher konstruieren wir zunächst die Liste als Material "Sammlung". In dieser werden die einzelnen Einträge verwaltet. Der Arbeitszusammenhang wird durch eine entsprechende Aspektklasse "Auflistbar" definiert.



Klassendiagramm: Auflister

Danach konstruieren wir das Werkzeug "Auflister" als Kombination von Interaktions- und Funktionskomponente (IAK_Auflister, FK_Auflister).

- Benutzereingaben werden von der Interaktionskomponente entgegengenommen und in entsprechende Operationen auf der Funktionskomponente umgesetzt.
- Weiterhin ist die Interaktionskomponente für die Präsentation des Werkzeugs und die Darstellung des bearbeiteten Materials verantwortlich.

Die Interaktionskomponente benutzt einen Interaktionstyp IAT_List, um die Liste auf dem Bildschirm darzustellen und listenbezogene Benutzereingaben (wie Selektion eines Eintrags) entgegenzunehmen.

Aspektklasse und Material sind hier vernachlässigt, weil wir uns hier auf die Kommunikation zwischen Interaktions- und Funktionskomponente des Werkzeugs konzentrieren wollen.

Klassendiagramm: Interaktionskomponente benutzt Interaktionstyp und Funktionskomponente

Der Interaktionstyp benachrichtigt die Interaktionskomponente bei jeder Benutzeraktion. Die Funktionskomponente benachrichtigt die Interaktionskomponente bei jeder Änderung ihres Zustandes, damit diese die Werkzeugpräsentation und die Darstellung des Materials aktualisieren kann.

Klassendiagramm: Benachrichtigung

3.2. Die Struktur zur Beschreibung der Muster

Die Struktur, in welcher wir die Muster beschreiben, gliedert sich nach [GHJ+94] wie folgt:

Mustername (Pattern Name)
Hier steht der Name des Musters, welcher immer in der Kapitelüberschrift enthalten ist.

Zielsetzung (Intent)
Hier wird die Frage beantwortet, wozu das Muster nützlich ist und welches Ziel mit dem Einsatz dieses Musters erreicht werden kann.

Auch bekannt als (Also Known As)
Hier stehen alle Namen, unter welchen das Muster sonst noch bekannt ist.

Motivation (Motivation)
Die Motivation erfolgt über ein Szenario, welches ein Entwurfsproblem beschreibt und erklärt, wie die Klassen- und Objektstrukturen in dem Muster das Problem lösen. Das Szenario soll helfen, die später folgende abstrakte Beschreibung besser zu verstehen.

Anwendbarkeit (Applicability)
Hier sind die Situationen beschrieben, in welchen der Mechanismus angewendet werden kann.

Struktur (Structure)
In diesem Abschnitt wird eine grafische Repräsentation der Klassen und Objekte des Musters in Form von Klassen- bzw. Objektdiagrammen gegeben.

Teilnehmer (Participants)
Hier werden die Klassen und/oder Objekte aufgeführt, welche das Muster bilden sowie deren jeweiligen Verantwortlichkeiten. Hierfür wird eine rein textuelle Beschreibungsform benutzt.

Zusammenarbeit (Collaborations)
In diesem Abschnitt wird dargestellt, wie die einzelnen Teilnehmer zusammenarbeiten, um ihre Aufgaben zu erfüllen. Hierzu verwenden wir Interaktionsdiagramme, welche wir für unsere Zwecke erweitert und angepaßt haben. Wir zeigen immer die drei Aufgaben Anmeldung, Abmeldung und Benachrichtigung.

Konsequenzen (Consequences)
Hier beschreiben wir die positiven und negativen Konsequenzen, welche aus der Benutzung des Musters erwachsen können.
Positive Konsequenzen sind durch das Symbol ☺ gekennzeichnet, negative Konsequenzen durch ☹. Konsequenzen, die nicht als positiv oder negativ beurteilt werden können, kennzeichnen wir durch die Kombination beider Symbole ☺☹.

Implementation (Implementation)
Hier werden nach [GHJ+94] Fallen, Hinweise und Techniken beschrieben, welche bei der Implementierung des Musters beachtet werden sollten. Wir haben diesen Teil bei der Gegenüberstellung komplett ausgelassen, da wir keine konkrete Anleitung zur Implementation der Muster geben wollten. Dazu sei auf die jeweils angeführte Literatur verwiesen.

Beispielcode (Sample Code)
In diesem Abschnitt werden Codefragmente vorgestellt, welche zum Verständnis der Musters hilfreich sind. Wir verwenden dazu eine eiffel-ähnliche Syntax und Namenskonvention, welche wir bei Bedarf den Erfordernissen anpassen. Aufgeschobene Klassen sind in der Klassendeklaration an dem Schlüsselwort "**`deferred`**" zu erkennen, aufgeschobene Routinen an dem Schlüsselwort "**`is deferred`**" nach der Routinendefinition. Klassennamen schreiben wir vollständig groß, Objektnamen beginnen mit einem Großbuchstaben.

Bekannte Benutzungen (Known Uses)
Hier werden Fälle zitiert, in welchen dieses Muster benutzt wurde. Außerdem verweisen wir an dieser Stelle auf weitere Literatur zu dem jeweiligen Muster.

Verwandte Muster (Related Patterns)
Hier werden Verweise auf verwandte Muster gegeben, sowie auf Muster, die in einer bestimmten Beziehung zu dem vorgestellten Muster stehen.

3.3. Beobachter-Muster: OBSERVER

Zielsetzung (Intent)
Das Ziel ist die Definition einer Abhängigkeitsbeziehung zwischen Objekten, so daß alle abhängigen Objekte (Beobachter) benachrichtigt werden, wenn sich das Subjekt (Beobachteter) ändert.
Des weiteren soll das Subjekt die abhängigen Objekte nicht konkret kennen, damit die Anzahl der abhängigen Objekte beliebig variiert werden kann und einzelne abhängige Objekte ausgetauscht werden können.

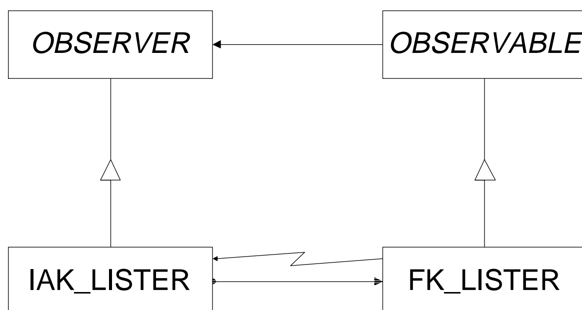
Auch bekannt als (Also Known As)
Dieses Muster ist in vielfältiger Form beschrieben und realisiert worden. Weitere Namen sind: Dependents, Publish-Subscribe, Change-Update, Notifier, Tell-Notify.

Motivation (Motivation)
In reaktiven Systemen, welche eine Trennung zwischen Interaktion und Funktion realisieren, tritt das Problem auf, daß die für Interaktion und Funktion zuständigen Objekte (Interaktions- und Funktionskomponente) in einem gegenseitigen Abhängigkeitsverhältnis stehen. Dabei soll die Funktionskomponente keine Kenntnis über die zugehörige Interaktionskomponente erhalten. Sie darf also die Interaktionskomponente nicht konkret kennen (siehe hierzu Kapitel 2).
Aus unserem Beispiel "Auflister" aus Abschnitt 3.1. soll die Funktionskomponente der Interaktionskomponente allgemein mitteilen, daß sich ihr Zustand geändert hat. Sie gibt der Interaktionskomponente keine weiteren Informationen. Die Interaktionskomponente muß durch Sondierung bei der Funktionskomponente selbst herausfinden, was sich geändert hat.

Anwendbarkeit (Applicability)
Verwende das Beobachtermuster in folgenden Situationen:

- Objekte sind von einem Subjekt abhängig und sollen benachrichtigt werden, wenn dieses Subjekt sich ändert.
- Das Subjekt soll keine konkrete Kenntnis von den abhängigen Objekten haben. Die Kopplung in diese Richtung soll also möglichst lose sein.
- Es ist nicht bekannt, wieviele Objekte vom Subjekt abhängig sind.

Struktur (Structure)



Klassendiagramm: OBSERVER

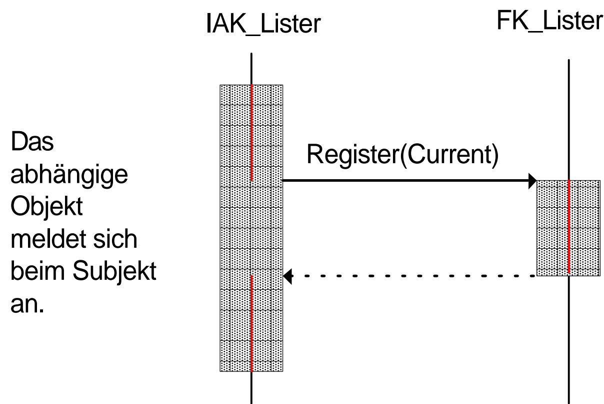
Die konkrete Benutzbeziehung zwischen OBSERVABLE und OBSERVER wird auf der Ebene der Klassen FK_LISTER und IAK_LISTER zu einer abstrakten Benachrichtigungsbeziehung, weil die Klasse IAK_LISTER der Klasse FK_LISTER nur abstrakt unter der Schnittstelle OBSERVER bekannt ist. Das Protokoll für den Reaktionsmechanismus wird in den aufgeschobenen Oberklassen OBSERVABLE und OBSERVER definiert.

Teilnehmer (Participants)

- *OBSERVABLE*
  - kennt seine Beobachter abstrakt unter der von OBSERVER definierten Schnittstelle, die Anzahl der Beobachter ist beliebig
  - stellt ein Interface zum An- und Abmelden von Beobachtern bereit
- *OBSERVER*
  - definiert eine Prozedur `update` in seinem Interface, über welche eine Zustandsänderung am OBSERVABLE mitgeteilt werden kann
- *FK_LISTER*
  - enthält Zustand, von welchem IAK_LISTER abhängig ist
  - benachrichtigt Beobachter über die in OBSERVABLE definierte Prozedur `notify`, wenn sich sein Zustand ändert
- *IAK_LISTER*
  - enthält Status, welcher konsistent mit dem des FK_LISTER sein muß
  - implementiert die in OBSERVER deklarierte Prozedur `update`

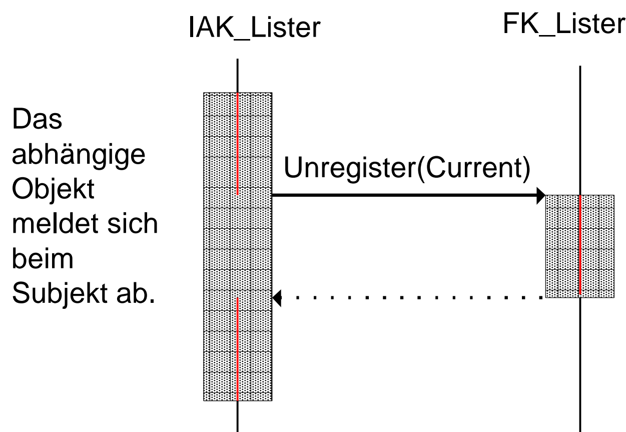
Zusammenarbeit (Collaborations)

- *Anmeldung*
  IAK_LISTER meldet sich bei FK_LISTER an, um über alle Veränderungen in FK_LISTER benachrichtigt zu werden.
- *Abmeldung*
  IAK_LISTER meldet sich bei FK_LISTER ab.
- *Benachrichtigung*
  FK_LISTER benachrichtigt IAK_LISTER allgemein über die in OBSERVER spezifizierte Prozedur `update`, wenn sich ihr Zustand ändert. Die Prozedur `update` bekommt als Parameter eine Referenz auf FK_LISTER übergeben, damit bei mehreren beobachteten Objekten die Benachrichtigung dem tatsächlichen Benachrichtiger zugeordnet werden kann.
  IAK_LISTER sondiert daraufhin FK_LISTER, um näheres über die Zustandsänderung zu erfahren.



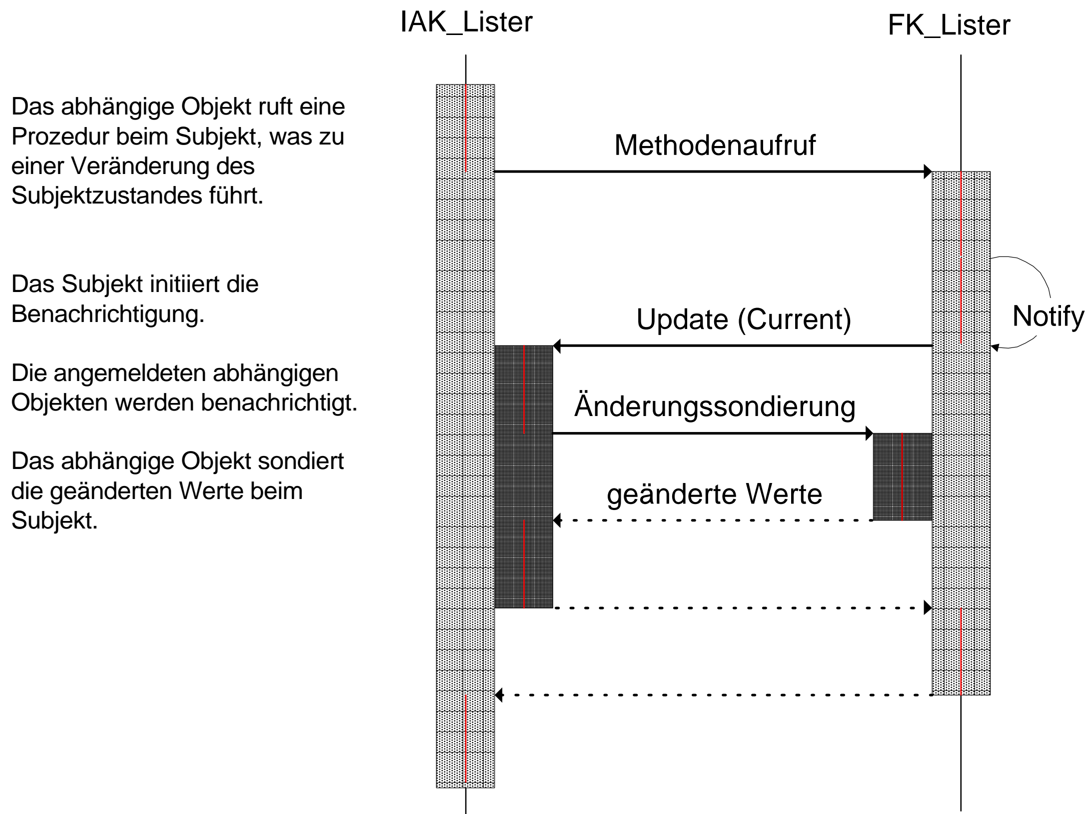
Interaktionsdiagramm: Anmeldevorgang OBSERVER

Bei der Anmeldung ruft die Interaktionskomponente eine spezielle Anmeldeprozedur der Funktionskomponente auf und übergibt sich selbst als Parameter.



Interaktionsdigramm: Abmeldevorgang OBSERVER

Bei der Abmeldung ruft die Interaktionskomponente eine spezielle Abmeldeprozedur der Funktionskomponte auf und übergibt wiederum sich selbst als Parameter, damit die Funktionskomponente die Interaktionskomponente aus der Liste der angemeldeten Beobachter entfernen kann.



Interaktionsdiagramm: Benachrichtigung OBSERVER

Findet in der Funktionskomponente eine Zustandsänderung statt, so werden alle angemeldeten Beobachter benachrichtigt, indem bei den Beobachtern die Prozedur `update` aufgerufen wird. Die benachrichtigten Objekte sondieren daraufhin die Funktionskomponente, um die geänderten Werte zu erfahren.

Konsequenzen (Consequences)

☺ Klassenschnittstellen sind einfach zu verstehen.

☺ Mechanismus ist einfach, wodurch die Realisierung und Wartung des Musters ebenfalls erleichtert wird.

☹☺ Es wird immer nur das sendende Objekt als Parameter übergeben, wodurch mitunter aufwendiges Sondieren notwendig wird. Zusätzlich kann eine einfache Konstante übergeben werden, welche die Art der Zustandsänderung näher charakterisiert. Schließlich ist auch die Übergabe von geänderten Werten denkbar. (Dann muß man allerdings in diesem Punkt auf statische Typsicherheit verzichten, da lediglich ein Pointer (C++) oder ein Objekt der Klasse ANY (Eiffel) übergeben werden kann. Eine entsprechende typkorrekte Typkonversion liegt dann im Verantwortungsbereich des konkreten Beobachters.)

☹ In Sprachen mit Einfachvererbung muß das Protokoll des OBSERVERs und OBSERVABLEs in einer Klassen realisiert werden. Dies macht die Klassenschnittstelle schwerer zu durchschauen.

☹ In der Update-Prozedur des OBSERVERs ist mitunter ein großes Case-Konstrukt notwendig, um die jeweils geänderten Werte beim OBSERVABLE zu sondieren und entsprechende Aktionen vornehmen zu können. In der Praxis zeigen sich hier schnell Wartungsprobleme, da die Wachstumsraten dieser Case-Konstrukte oft unterschätzt werden und daher die eigentlichen Abhandlungen anfangs oft nicht in eigene Prozeduren gekapselt werden.

☹ Möchte man eine Undo/Redo-Funktionalität realisieren (z.B. bei Beobachtung von grafischen Oberflächenobjekten), so muß man zu weiteren Mustern greifen (wie Memento in [GHJ+94]).

Beispielcode (Sample Code)

```
deferred class OBSERVER
     update (changed_observable : OBSERVABLE) is deferred
          -- wird gerufen, wenn sich ein beobachtetes
          -- Objekt geändert hat,
          -- muß von der konkreten Klasse implementiert
          -- werden
end

class OBSERVABLE

     register (o : OBSERVER) is
          -- melde 'o' als Beobachter an
     do
          observer_list.add(o)
     end

     unregister (o : OBSERVER) is
          -- melde 'o' als Beobachter ab
     do
          observer_list.remove(o)
     end
```

```
feature {NONE} -- private Features

     notify is
          -- melde eine Veränderung an alle angemeldeten
          -- Beobachter
     do
          from
               pos := observer_list.first_pos
          until
               pos > observer_list.last_pos
          loop
               observer_list.item(pos).update(Current)
               pos := observer_list.next_pos(pos)
          end
     end

     observer_list : LIST[OBSERVER]
          -- Liste mit den angemeldeten Beobachtern

end

class FK_LISTER
inherit OBSERVABLE

     select (idx : INTEGER) is
          -- selektiere Eintrag an der Position 'idx'
     do
          ...
          notify
     end

     selected : INTEGER
          -- der aktuell selektierte Eintrag

...
end
```

```
class IAK_LISTER
inherit OBSERVER

     update (changed_observable : OBSERVABLE) is
          -- wird gerufen, wenn sich ein beobachtetes
          -- Objekt geändert hat
     do
          if changed_observable = fk_lister then
               if fk_lister.selected /= iat_list.selected
               then
                    iat_list.select( fk_lister.selected )
               elsif ... -- Reaktion auf weiteres
               end
          end
     end

     fk_lister : FK_LISTER
     iat_list : IAT_LIST

end
```

Bekannte Benutzungen (Known Uses)
Das Muster ist [GHJ+94] entnommen. Es ist in einer Vielzahl von Klassenbibliotheken und Frameworks so oder ähnlich realisiert.
So findet sich z.B. in Smalltalk-80 [GR93] die klassische Lösung, in welcher sowohl das Observer- wie auch das Observable-Protokoll in einer Klasse OBJECT implementiert sind. Da in Smalltalk-80 alle Klassen automatisch von dieser Klasse OBJECT erben, steht der Mechanismus auch allen Klassen automatisch im vollen Umfang zur Verfügung.
In vielen C++-Realisierungen findet sich ebenfalls eine Vereinigung der beiden Protokolle in einer Klasse, z.B. ET++ [Gam92], als Klasse Notifier im Rahmenwerk des Arbeitsbereichs Softwaretechnik der Universität Hamburg [Rie93a][Rie93b], Unidraw. Beim Notifier-Mechanismus wird dem Observer als Standard-Parameter ein Integer-Wert mitgegeben, welcher die Zustandsänderung des Beobachteten genauer beschreibt. Für diese Werte sind entsprechende Konstanten definiert, welche allen Notifier-Benutzern zur Verfügung stehen.
Weiterhin ist das Muster in InterViews und dem AndrewToolkit realisiert. Laut [GHJ+94] ist das Muster außerdem in der THINK-Bibliothek realisert.

Verwandte Muster (Related Patterns)
- Singleton (nach [GHJ+94]): Die Benachrichtigung kann über einen CHANGE_MANAGER geschehen, welcher nur einmal im ganzen System vorhanden und außerdem global für alle Subjekte zugreifbar ist.
- Mediator (nach [GHJ+94]): Ein evt. vorhandener CHANGE_MANAGER spielt dann die Rolle des Mediators, welcher zwischen abhängigen Objekten und Subjekten vermittelt.

3.4. Kommando-Muster: COMMAND

Zielsetzung (Intent)

Beim Command-Muster wird eine Kommandoklasse definiert, deren Objekte quasi Prozeduren realisieren und als Erste-Klasse-Objekte zur Verfügung stellen. Diese werden bei einem Subjekt angemeldet, so daß dieses bei Bedarf das Kommando-Objekt "ausführen" kann.

Auch bekannt als (Also Known As)

Das Muster ist auch unter den Namen Action und Transaction bekannt.

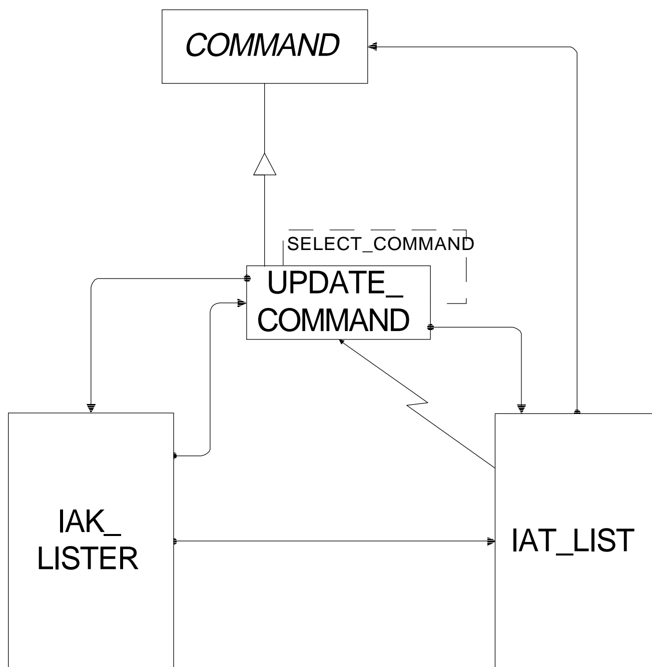
Motivation (Motivation)

In fensterbasierten Systemen mit grafischen Benutzungsoberflächen und Menüsystemen tritt das Problem auf, daß die komplexe Funktionalität der Benutzungsberfläche in der Regel in Bibliotheken gekapselt ist. Die Ausführung eines Kommandos als Reaktion auf eine Benutzeraktion (Auswahl eines Menüeintrages oder Anklicken eines Knopfes) kann daher nicht fest in die Bibliothek einprogrammiert werden, muß dieser aber abstrakt zur Verfügung gestellt werden.
Weiterhin ist es meist wünschenswert, eine prinzipiell unbeschränke Undo-/Redo-Funktionalität für durch den Benutzer ausgelöste Kommandos zur Verfügung zu stellen. Die Lösung des Problemes besteht hier in der Einführung einer aufgeschobenen Kommandoklasse. Konkrete Kommandoklassen werden von dieser abgeleitet und implementieren die gewünschte Funktionalität. Neben einer Execute-Prozedur kann hier weiterhin eine Unexecute-Prozedur definiert werdem, welche für die Rücknahme des Kommandos zuständig ist.

Anwendbarkeit (Applicability)

Verwende dieses Muster in folgenden Situationen:

- Als Reaktion auf eine Benutzeraktion soll eine bestimmte Aktion ausgeführt werden.
- Die Elemente der Benutzungsoberfläche sollen keine konkrete Kenntnis von den auszuführenden Kommandos haben. Die Kopplung in diese Richtung soll also möglichst lose sein.
- Es soll eine Undo-/Redo-Funktionalität zur Verfügung stehen.



Klassendiagramm: COMMAND

Die konkrete Benutztbeziehung zwischen IAT_LIST und COMMAND wird zu einer abstrakten Benachrichtigungsbeziehung zwischen IAT_LIST und UPDATE_COMMAND, weil der Klasse IAT_LIST das UPDATE_COMMAND nur unter der Schnittstelle der Klasse COMMAND bekannt ist. Für jede Benutzeraktion, auf die IAK_LISTER reagieren will, muß eine entsprechende Kommandoklasse implementiert werden. Die bidirektionale Benutztbeziehung zwischen IAK_LISTER und UPDATE_COMMAND ist unkritisch, da beide Klassen konzeptionell nicht voneinander zu trennen sind.

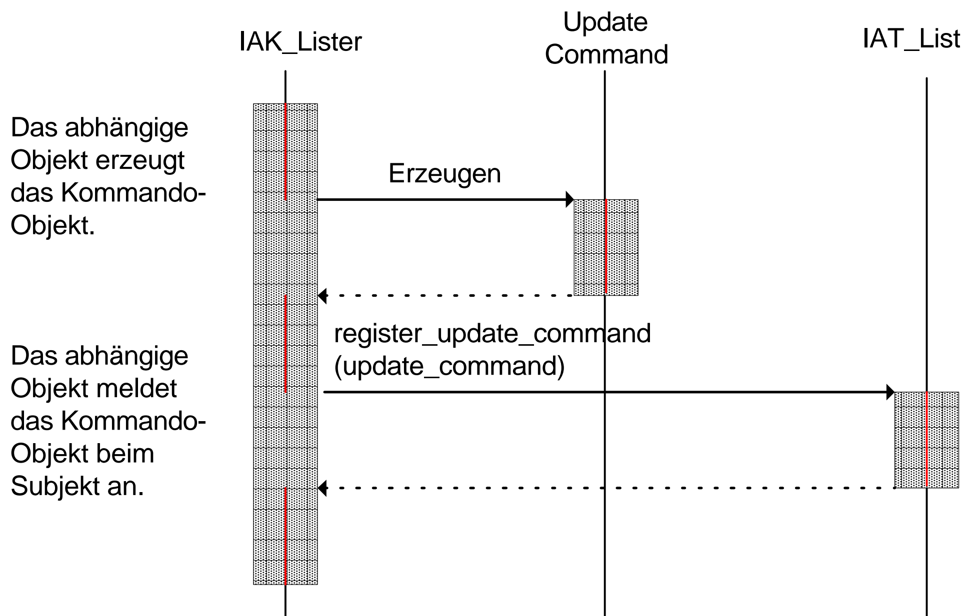
Teilnehmer (Participants)

- *COMMAND*
  - definiert ein Interface zum Aufruf einer Prozedur `execute`
- *IAT_LIST*
  - enthält Zustand, von welchem IAK_LISTER abhängig ist
  - kennt die angemeldeten Kommandoobjekte abstrakt unter der Schnittstelle COMMAND
  - stellt je möglicher Benutzeraktion am IAT_LIST ein Interface zum An- und Abmelden von Kommandoobjekten unter der Schnittstelle COMMAND bereit
  - ruft das angemeldete Exemplar der Klasse UPDATE_COMMAND, wenn der Benutzer die Liste verändert

- *UPDATE_COMMAND*
  - implementiert die Prozedur `execute` von COMMAND, in welcher als Reaktion auf eine Benutzeraktion die entsprechenden Operationen durchgeführt und/oder Routinen aus IAK_LISTER aufgerufen werden, so daß IAK_LISTER auf die Benutzeraktion reagieren kann
- *IAK_LISTER*
  - meldet ein Exemplar der Klasse UPDATE_COMMAND beim IAT_LIST an
  - stellt eine oder mehrere Prozeduren zur Verfügung, mit deren Hilfe UPDATE_COMMAND auf Benutzeraktionen in IAT_LIST reagieren kann

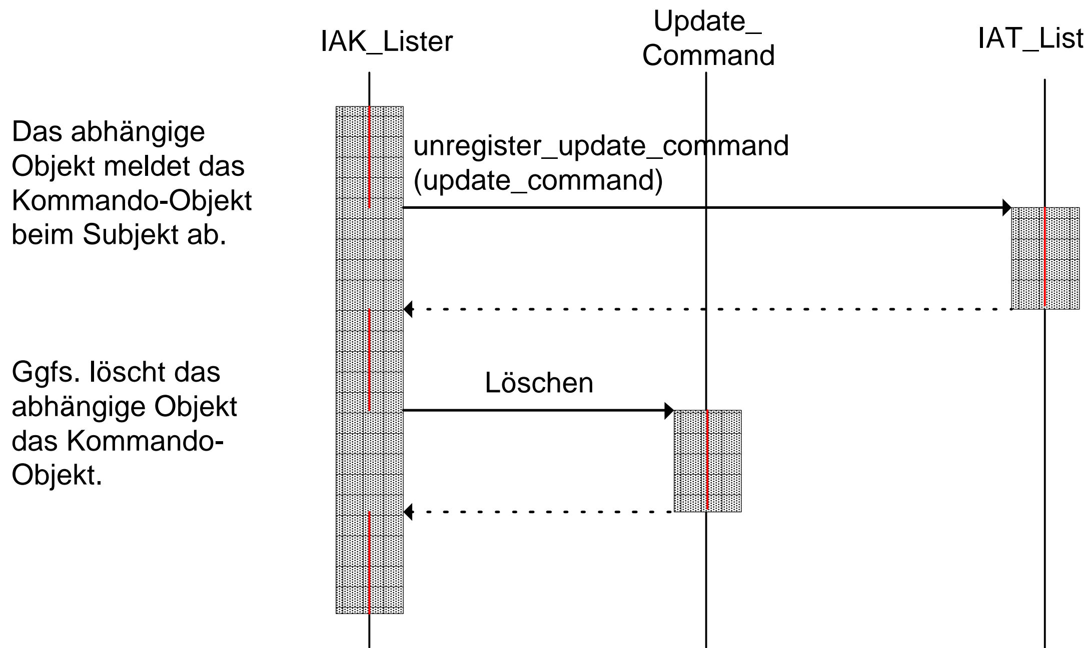
Zusammenarbeit (Collaborations)

- *Anmeldung*
  IAK_LISTER erzeugt ein Exemplar der Klasse UPDATE_COMMAND und meldet dieses bei IAT_LIST an.
- *Abmeldung*
  IAK_LISTER meldet das Exemplar der Klasse UPDATE_COMMAND bei IAT_LIST ab und löscht ggf. dieses Exemplar
- *Benachrichtigung*
  IAT_LIST "aktiviert" das für die Benutzeraktion angemeldete Kommando, welches die jeweilige Aktion bzw. jeweiligen Aktionen in IAK_LISTER aufruft.



Interaktionsdigramm: Anmeldevorgang COMMAND

Die Interaktionskomponente erzeugt je Benutzeraktion, für welche sie sich interessiert, ein spezielles Kommando-Objekt. Dieses wird beim Interaktionstyp angemeldet. Der Interaktionstyp stellt je möglicher Benutzeraktion eine spezielle Anmeldeprozedur zur Verfügung.



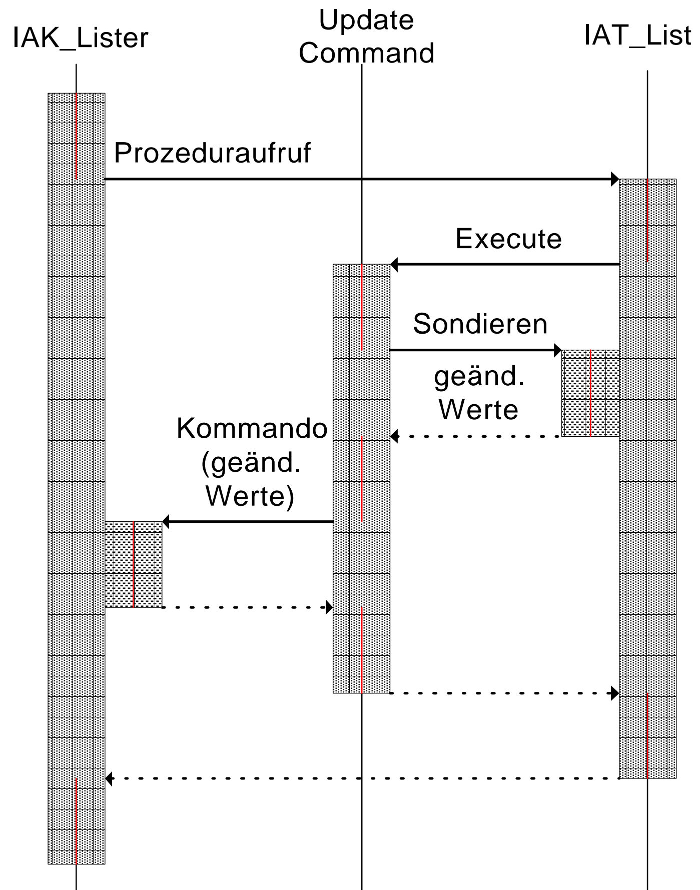
Interaktionsdigramm: Abmeldevorgang COMMAND

Bei der Abmeldung ruft die Interaktionskomponente eine spezielle Abmeldeprozedur beim Interaktionstyp auf und übergibt das entsprechende Kommando-Objekt an diese Prozedur. In Programmiersprachen ohne Garbage-Collector kann danach die Löschung des Kommando-Objektes durch die Interaktionskomponente erfolgen. Der Interaktionstyp stellt je Benutzeraktion eine spezielle Abmeldeprozedur zur Verfügung.

Das abhängige Objekt ruft eine Prozedur des Subjektes auf, was zu einer Veränderung des Subjektes führt.

Daraufhin aktiviert das Subjekt ein entsprechendes Kommando-Objekt. Das Kommando-Objekt kann dann die Veränderungen beim Subjekt sondieren. Dies kann auch durch das abhängige Objekt geschehen.

Das Kommando-Objekt ruft eine oder mehrere Prozeduren beim abhängigen Objekt auf.



Interaktionsdigramm: Benachrichtigung COMMAND

Findet am Interaktionstyp eine Benutzeraktion statt, so wird daraufhin das Kommando-Objekt aufgerufen, welches für diese Benutzeraktion angemeldet wurde. Das Kommando-Objekt kann daraufhin die geänderten Werte beim Interaktionstyp sondieren und einen speziellen Prozeduraufruf bei der Interaktionskomponente vornehmen. Bei diesem Prozeduraufruf können die geänderten Werte als Parameter mitgegeben werden. Alternativ könnte das Kommando-Objekt auch einfach eine spezielle Prozedur ohne Parameter bei der Interaktionskomponente aufrufen und ihr die Sondierung beim Interaktionstyp überlassen.

Konsequenzen (Consequences)

- ☺ Das große CASE-Konstrukt beim abhängigen Objekt entfällt, da die Fälle der Fallunterscheidung als Klassen realisiert werden.
- ☺ Die Kommandoklasse kann die Veränderungen beim Subjekt sondieren und diese dem abhängigen Objekt zur Verfügung stellen.
- ☺ Eine Undo-/Redo-Funktionalität läßt sich mit diesem Muster leicht realisieren.

- ☹ Es gibt keine Oberklassen OBSERVER und OBSERVABLE. Dadurch kann nicht am Klassenbaum erkannt werden, welche Klassen abhängig und welche Subjekte sind.
- ☹ Es muß je interessierender Benutzeraktion eine eigene Kommandoklasse implementiert werden. Von diesen Klassen wird in der Regel nur jeweils ein Exemplar erzeugt.

Beispielcode (Sample Code)

```
deferred class COMMAND
     execute () is deferred
          -- führe bestimmte Aktionen beim abhängigen
          -- Objekt aus
     unexecute () is deferred
          -- mache letzte Aktion rückgängig
end

class UPDATE_COMMAND
inherit COMMAND

     make (owner : IAK_LISTER; iat : IAT_LIST) is
          -- initialisiere
     do
          iak_lister := owner
          iat_lister := iat
     end

     execute () is
          -- führe bestimmte Aktionen bei IAK_LISTER aus
     do
          ... -- rufe Prozedur(en) bei IAK_LISTER auf
     end

     unexecute () is
          -- mache letzte Aktion rückgängig
     do
          ... -- rufe Prozedur(en) bei IAK_LISTER auf
     end

feature {NONE} -- private Features

     iak_lister : IAK_LISTER
     iat_list : IAT_LIST

end
```

```
class IAT_LIST

     register_update_command (c : COMMAND) is
          -- melde 'c' als Kommando an
     do
          update_command := c
     end

     unregister_update_command (c : COMMAND) is
          -- melde 'c' als Kommando ab
     do
          update_command := Void
     end

     -- entsprechend An- und Abmelderoutinen für
     -- select_command

feature {NONE}

     update_command : COMMAND
     select_command : COMMAND
...
end

class IAK_LISTER

     make is
     do
          !! iat_list.make
          !! update_command.make(Current, iat_list)
          iat_list.register_update_command(update_command)
          !! select_command.make(Current, iat_list)
          iat_list.register_select_command(select_command)
     end

     -- hier stehen weitere Prozeduren, welche die
     -- Kommandoobjekte aufrufen können
...
end
```

Bekannte Benutzungen (Known Uses)

Das Muster ist in Eiffel-Vision ([EV94]) realisiert. Außerdem ist das Muster in [GHJ+94] beschrieben und demnach in MacApp, ET++, InterViews, Unidraw und THINK realisiert.

Verwandte Muster (Related Patterns)

Man kann durch die Verwendung dieses Musters mitunter auf die Verwendung des Memento-Musters ([GHJ+95]) verzichten.

Innerhalb der Werkzeug&Material-Metapher kann die Kommunikation zwischen Interaktionskomponenten und Interaktionstypen mit diesem Muster realisiert werden. Es bietet sich überall dort an, wo es nur ein abhängiges Objekt zu einem Subjekt gibt.

3.5. Ereignis-Muster: EVENT-NOTIFICATION

Zielsetzung (Intent)

Bei diesem Muster stehen im Gegensatz zum Observer-Muster nicht die Beobachtungen im Vordergrund, sondern die Ereignisse, welche verschickt, empfangen und interpretiert werden. Das hier vorgestellte Muster ([Rie95]) basiert auf dem Ansatz von [NGG+93], auch wobei der Aspekt des impliziten Prozeduraufrufes in den Hintergrund tritt.
Grundidee bei diesem Muster ist die Auslagerung des Reaktionsmechanismusses in Ereignisklassen, was zur Folge hat, daß keine expliziten OBSERVER- und OBSERVABLE-Oberklassen mehr benötigt werden.
Die Anmeldung erfolgt bei den einzelnen Ereignissen. Außerdem kann man diesen in C++ die aufzurufenden Prozeduren typsicher übergeben. Weiterhin können diese Ereignisse auch die geänderten Werte direkt als Ereignis-Parameter an die abhängigen Objekte weitergeben.

Auch bekannt als (Also Known As)

Das ursprüngliche Muster von Sullivan&Notkin heißt Implicit-Invocation.

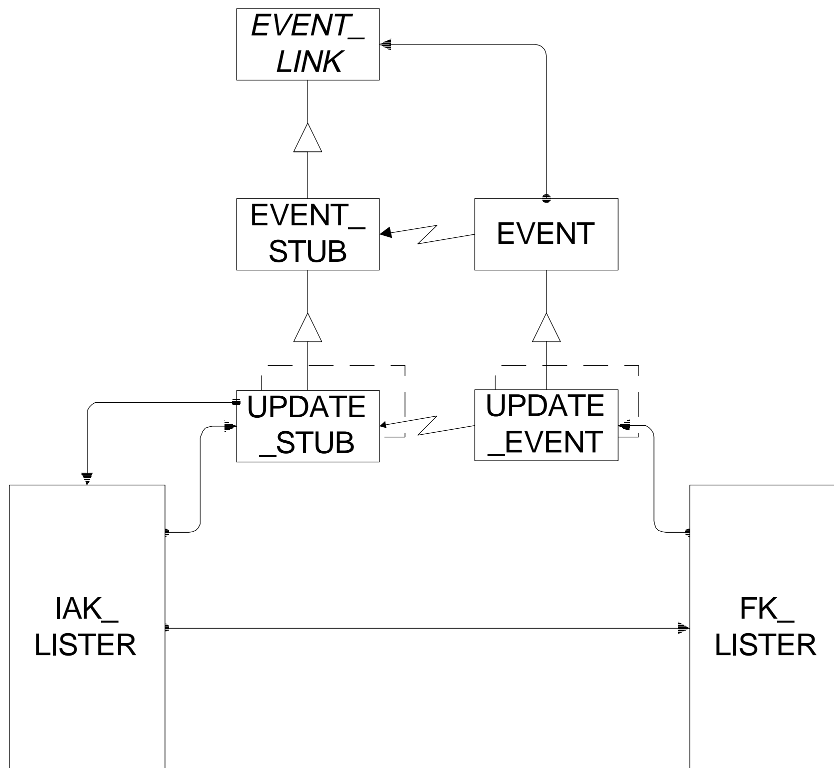
Motivation (Motivation)

Beim Event-Notification-Muster liegt die Motivation ähnlich wie beim Observer-Muster (siehe Abschnitt 3.3.).
Bei dem hier vorgestellten Event-Notification-Muster hat man allerdings die Möglichkeit, verschiedene Ereignisse durch Ereignisklassen zu differenzieren und zu parametrisieren.

Anwendbarkeit (Applicability)

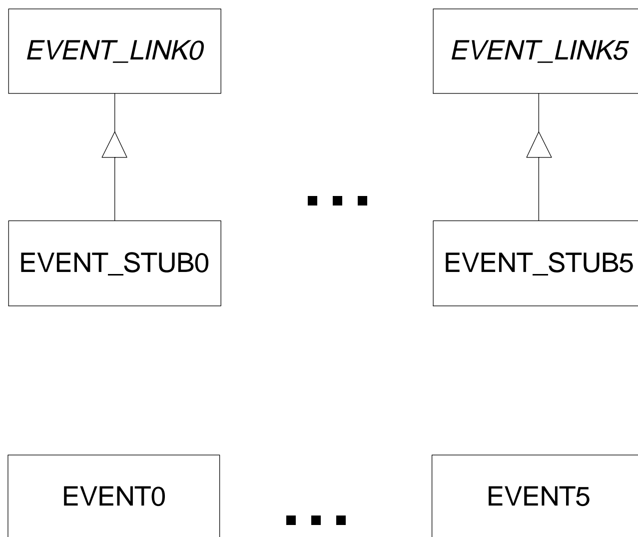
Verwende dieses Muster in folgenden Situationen:
Es gilt die gleiche Anwendbarkeit wie beim Observer-Muster (siehe Abschnitt 3.3.), sowie:

- Die Reaktionsbeziehungen sind komplex und vielfältig. Hier hilft die explizite Deklaration der verschickten Ereignisse bei den Subjekten sowie ggfs. der empfangenen Ereignisse bei den abhängigen Klassen, die Abhängigkeitsbeziehungen zwischen den Klassen deutlicher zu machen.
- Es sollen bei der Verschickung von Ereignissen zur Aktualisierung der abhängigen Objekte die geänderten Werte als Ereignis-Parameter mitgegeben werden. Dies ist z.B. dann wichtig, wenn man asynchron über Adreßraumgrenzen hinweg Ereignisse verschicken möchte, da Sondierung grundsätzlich nur bei synchroner Kommunikation sinnvoll möglich ist.



Klassendiagramm: EVENT-NOTIFICATION

Die konkrete Benutztbeziehung zwischen EVENT und EVENT_LINK wird zu einer abstrakten Benachrichtigungsbeziehung zwischen EVENT und EVENT_STUB (ebenso zwischen UPDATE_EVENT und UPDATE_STUB), da EVENT_STUB der Klasse EVENT nur unter der Schnittstelle EVENT_LINK bekannt ist.



Klassendiagramm Events, Event-Links, Event-Stubs

Es gibt je Parameteranzahl eine EVENT-LINK-, eine EVENT-STUB- sowie eine EVENT-Klasse. EVENT2 steht z.B. für einen Event mit zwei Parametern.

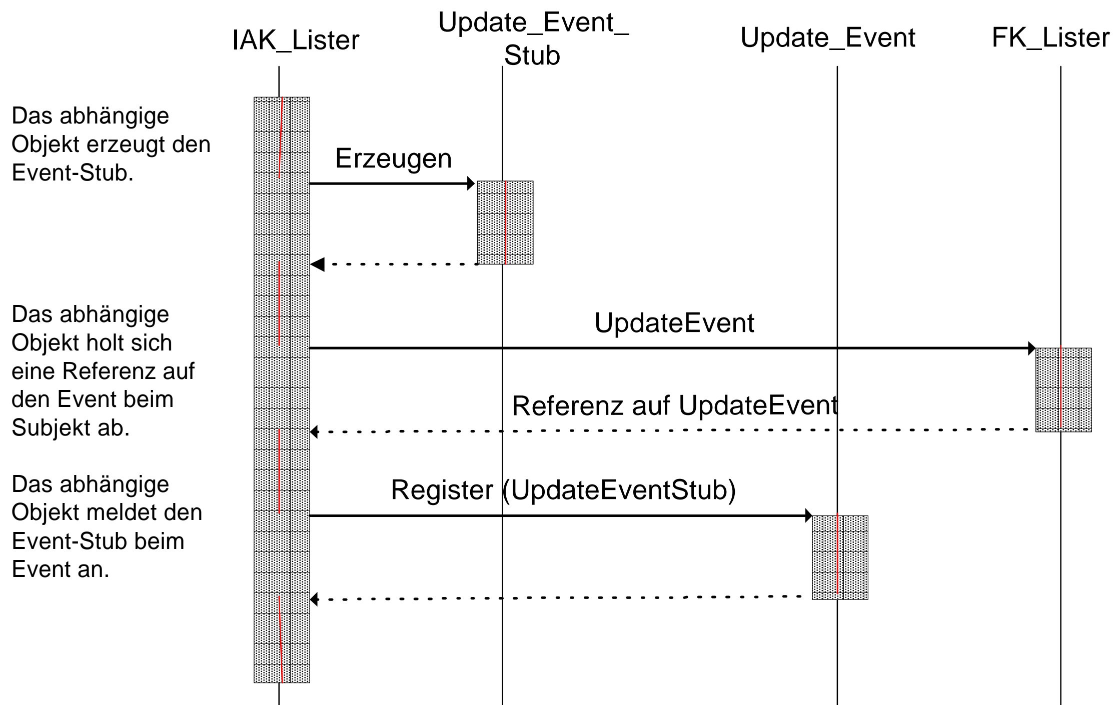
Teilnehmer (Participants)

- *EVENT0 bis EVENT5*
  - stellt ein Interface zum An- und Abmelden von Event-Links bereit
  - kennt die angemeldeten Event-Stubs unter der Schnittstelle der entsprechenden Event-Links
  - benachrichtigt auf Kommando alle angemeldeten Event-Links
- *UPDATE_EVENT*
  - wird vom FK_LISTER erzeugt
  - wird bei Veränderung des Subjekts aktiviert
- *EVENT_LINK0 bis EVENT_LINK5*
  - stellen Interface zur Benachrichtigung zur Verfügung
- *EVENT_STUB0 bis EVENT_STUB5*
  - wird beim Erzeugen mit einer bei Benachrichtigung auszuführenden Prozedur initialisiert
  - realisiert das Interface zur Benachrichtigung durch Prozeduraufruf der übergebenen Prozedur
- *UPDATE_STUB*
  - wird von IAK_LISTER erzeugt und beim UPDATE_EVENT angemeldet
  - wird von UPDATE_EVENT aktiviert und ruft daraufhin die bei ihr angemeldete Prozedur von IAK_LISTER
- *FK_LISTER*
  - enthält Zustand, von welchem IAK_LISTER abhängig ist
  - stößt UPDATE_EVENT an, wenn sich sein Zustand ändert

- *IAK_LISTER*
  - erzeugt UPDATE_STUB und meldet dabei die bei Benachrichtigung aufzurufende Prozedur an
  - meldet UPDATE_STUB bei UPDATE_EVENT an
  - enthält Status, welcher konsistent mit dem des FK_LISTER sein muß

Zusammenarbeit (Collaborations)

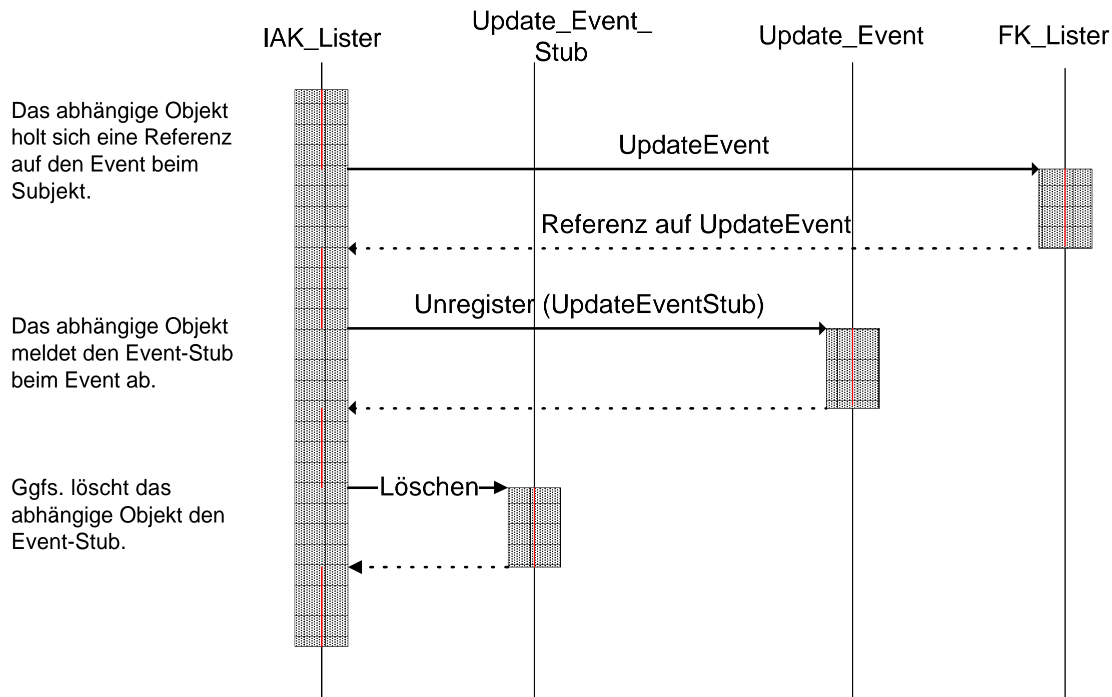
- *Anmeldung*
  IAK_LISTER erzeugt UPDATE_STUB und meldet die aufzurufende Prozedur bei diesem an. Daraufhin meldet IAK_LISTER UPDATE_STUB bei UPDATE_EVENT an.
- *Abmeldung*
  IAK_LISTER meldet UPDATE_STUB bei UPDATE_EVENT ab und löscht ggfs. UPDATE_STUB.
- *Benachrichtigung*
  FK_LISTER löst UPDATE_EVENT aus, welcher wiederum alle angemeldeten EVENT_LINKs benachrichtigt. UPDATE_STUB ruft daraufhin die angemeldete Prozedur auf. Die ganze Kette über werden die geänderten Werte als Parameter mit übergeben.



Interaktionsdiagramm: Anmeldevorgang EVENT-NOTIFICATION

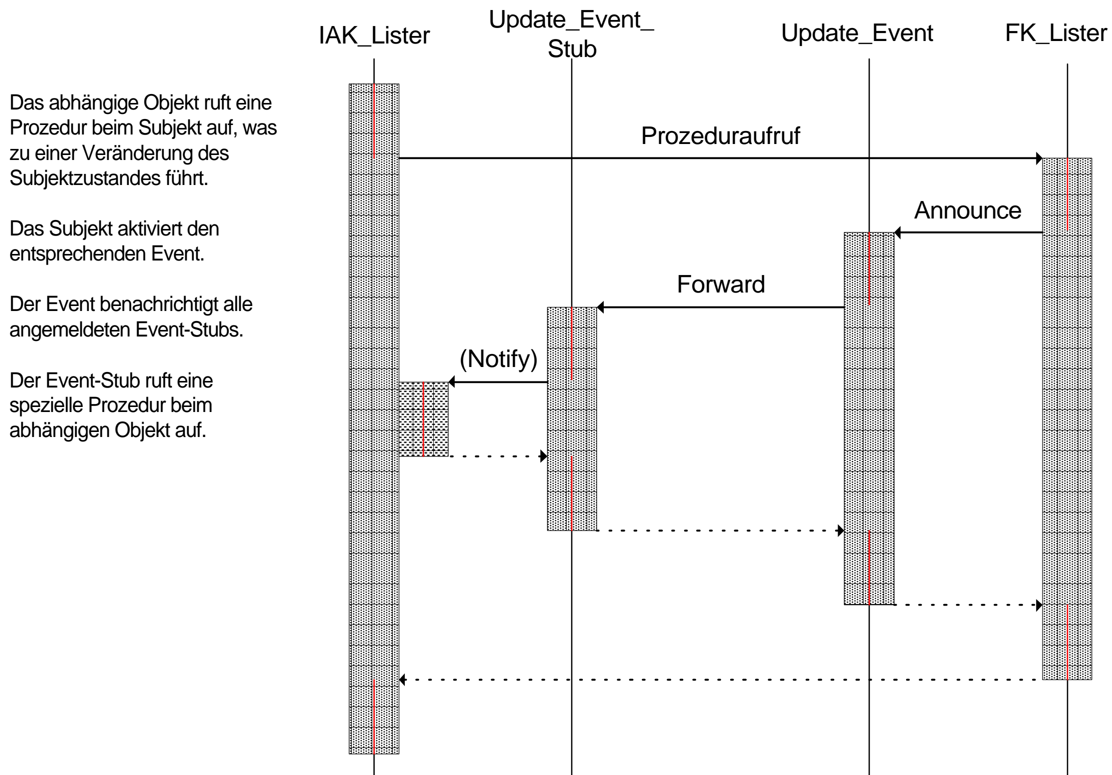
Die Funktionskomponente erzeugt je Event, welchen sie aussendet, einen Event, welchen sie an ihrer Schnittstelle deklariert.
Die Interaktionskomponente erzeugt je Event, für welchen sie sich interessiert einen Event-Stub. Diesem übergibt sie eine ihrer Prozeduren. Diese Prozedur soll aufgerufen werden, wenn der Event-Stub aktiviert wird.

Danach holt sich die Interaktionskomponente bei der Funktionskomponente eine Referenz auf den entsprechenden Event und meldet einen passenden Event-Stub bei diesem Event an, indem sie den Event-Stub an die Anmeldeprozedur des Events übergibt.
Der Event hat eine Liste der zu benachrichtigen Event-Stubs. Der Event, der Event-Stub und die aufzurufende Prozedur der Interaktionskomponente müssen die gleichen Parameter haben.



Interaktionsdiagramm: Abmeldevorgang EVENT-NOTIFICATION

Die Interaktionskomponente holt sich bei der Funktionskomponente eine Referenz auf den Event und meldet den entsprechenden Event-Stub beim Event ab. Der Event entfernt daraufhin diesen Event-Stub aus der Liste der zu benachrichtigen Event-Stubs. Danach kann der Event-Stub von der Interaktionskomponente gelöscht werden.



Interaktionsdiagramm: Benachrichtigung EVENT-NOTIFICATION

Bei einer Zustandsänderung in der Funktionskomponente stößt diese die Event-Benachrichtigung bei dem entsprechenden Event an (announce). Der Event benachrichtigt daraufhin alle bei ihm angemeldeten Event-Stubs. Der Event-Stub ruft daraufhin die bei ihm angemeldete Prozedur bei der Interaktionskomponente auf. Die ganze Kette entlang werden die geänderten Werte mit übergeben, damit die Interaktionskomponente nicht mehr bei der Funktionskomponente sondieren muß.

Konsequenzen (Consequences)

- ☺ Kein CASE-Konstrukt notwendig, weil direkt die aufzurufende Prozedur übergeben wird.
- ☺ Durch die Event-Links ist der Grundstein für asynchrone Ereignismeldung in Netzwerken gelegt. Die Event-Links können dann als Proxies (siehe dazu auch [GHJ+95]) realisiert werden.
- ☺ Es wird an der Klassenschnittstelle explizit gemacht, welche Ereignisse verschickt werden und welche Ereignisse empfangen werden. Letzteres gilt nicht für den in [NGG+93] beschriebenen Ansatz.
- ☺ Es können die geänderten Werte direkt als Event-Parameter mitgegeben werden. Dadurch wird explizit gemacht, welche geänderten Werte zu der Zustandsänderung gehören.
- ☹ Läßt sich in Programmiersprachen, welche Routinen nicht als Objekte erster Klasse sehen, so nicht implementieren, weil sich keine Prozedur an die Event-Stubs übergeben läßt.

- ☹ Durch das Fehlen allgemeiner Oberklassen für Beobachter und Beobachtete, läßt sich am Klassenbaum nicht erkennen, welche Klassen Events verschicken oder empfangen.
- ☹ Die Prozedur `announce` der Klasse EVENT ist für alle Objekte aufrufbar, welche Zugriff auf das Subjekt haben.
- ☹ Die in [NGG+93] beschriebene Variante benutzt C++Makros, um die hier vorgestellte Funktionalität zu realisieren, was zu einer erschwerten Verständlichkeit und Wartbarkeit führt.

Beispielcode (Sample Code)

```
class EVENT0
     -- Event ohne Parameter

     register ( link : EVENT_LINK0 )
          -- registriere 'link' als Empfänger
     unregister ( link : EVENT_LINK0 )
          -- mache Registrierung für 'link' rückgängig
     announce ( )
          -- melde Ereignis
end

class EVENT1 [ ARG_TYPE1 ]
     -- Event mit einem Parameter

     register ( link : EVENT_LINK1[ ARG_TYPE1 ] )
          -- registriere 'link' als Empfänger
     unregister ( link : EVENT_LINK1[ ARG_TYPE1 ] )
          -- mache Registrierung für 'link' rückgängig
     announce ( arg : ARG_TYPE1 )
          -- melde Ereignis mit Parameter 'arg'
end

...

class EVENT5 [ ARG_TYPE1, ARG_TYPE2, ARG_TYPE3, ARG_TYPE4,
               ARG_TYPE5 ]
     -- Event mit fünf Parametern
...
end

deferred class EVENT_LINK0
     -- Event-Link ohne Parameter
     forward ( ) is deferred
     -- melde Ereignis weiter
end
```

```
deferred class EVENT_LINK1 [ ARG_TYPE1 ]
     -- Event-Link für einen Parameter
     forward ( arg : ARG_TYPE1 ) is deferred
     -- melde Ereignis mit Parameter 'arg' weiter
end


...

deferred class EVENT_LINK5 [ ARG_TYPE1, ARG_TYPE2,
                             ARG_TYPE3, ARG_TYPE4,
                             ARG_TYPE5 ]
     -- Event-Link für fünf Parameter
...
end

class EVENT_STUB0
     -- Event-Stub ohne Parameter

inherit EVENT_LINK0

     make ( proc() : PROCEDURE[14] )
     -- Konstruktor
     -- 'proc' soll bei Aktivierung des Stubs gerufen werden

     forward ( )
     -- melde Ereignis weiter
end

class EVENT_STUB1 [ ARG_TYPE1 ]
     -- Event-Stub für einen Parameter

inherit EVENT_LINK1 [ARG_TYPE1]

     make ( proc(arg1 : ARG_TYPE1) : PROCEDURE )
     -- Konstruktor
     -- 'proc' soll bei Aktivierung des Stubs gerufen werden

     forward ( arg : ARG_TYPE1 )
     -- melde Ereignis mit Parameter 'arg' weiter
end

...
```

[14]Diese Notation gibt es in Eiffel nicht. Sie soll andeuten, daß hier eine Prozedur zu übergeben ist.

```
class EVENT_STUB5 [ ARG_TYPE1, ARG_TYPE2, ARG_TYPE3,
                    ARG_TYPE4, ARG_TYPE5 ]
     -- Event-Stub für fünf Parameter
...
end

class FK_LISTER

     do_something is
     do
          ... -- Zustand verändern
          update_event.announce(selected)
     end

     update_event : EVENT1[ INTEGER ]
          -- Update-Event mit einem Integer-Parameter

     selected : INTEGER
          -- selektierter Eintrag in der Sammlung

...
end

class IAK_LISTER
     update_procedure (idx : INTEGER)
          -- zu rufende Prozedur mit einem Integer-Parameter
     update_stub : EVENT_STUB_1[ INTEGER ]
          -- Empfänger-Stub für Update_Event
...
end
```

Es ist hier darauf hinzuweisen, daß die obige Implementation aus zwei Gründen so nicht in Eiffel zu implementieren ist:

- In Eiffel sind Prozeduren keine Erste-Klasse-Objekte und können daher nicht als Parameter übergeben werden.
- Da generische Klassen in Eiffel grundsätzlich von C++-Templates verschieden sind, läßt sich das Muster in C++ mit Hilfe von Templates implementieren, aber nicht in Eiffel unter Verwendung generischer Klassen. In Eiffel führt dies zu Typproblemen.[15]

Bekannte Benutzungen (Known Uses)
Das makrobasierte Muster ist in [NGG+93] und [Gra95] beschrieben und das darauf aufbauende Muster von Riehle ist in [Rie95a] beschrieben. Konkrete Realisierungen sind uns bisher nicht bekannt.

[15]In C++ gibt es deshalb keine Typprobleme, weil Templates nicht in das Typkonzept eingebunden sind und für jede instanziierte Template-Klasse eine expandierte Klasse erzeugt wird.

Verwandte Muster (Related Patterns)

- Singleton (nach [GHJ+95]): Das Versenden der Events kann über einen CHANGE_MANAGER geschehen, welcher nur einmal im ganzen System vorhanden und außerdem global für alle Events zugreifbar ist.
- Mediator (nach [GHJ+95]): Ein evt. vorhandener CHANGE_MANAGER spielt dann die Rolle des Mediators, welcher zwischen Events und Event-Links vermittelt.
- Kapselung von Window-System-spezifischen Oberflächenelementen in Interaktionstypen: Kapselung von Window-System-spezifischen Oberflächenelemente in Iteraktionstypen, welche der Interaktionskomponente zugeordnet sind, setzt einen Reaktiosmechanismus voraus. Die Interaktionskomponente muß auf Benutzeraktionen am Interaktionstyp reagieren. Diese Reaktionsbeziehung läßt sich über das Event-Notification-Muster realisieren. Hier wirkt sich vorteilhaft aus, daß man bei Interaktionstypen davon ausgehen kann, daß (fast) alle geänderten Werte auch für den Beobachter interessant sind. Es ist daher hier von Vorteil, die geänderten Werte der Interaktionskomponente als Event-Parameter zur Verfügung stellen zu können.

3.6. Tabellarische Gegenüberstellung der Muster

Zusammenfassend stellt die folgende Tabelle die vorgestellten Muster anhand der in Kapitel 2.1 beschriebenen Unterscheidungsmerkmale gegenüber[16]:

	Beobachtermuster	*Kommandomuster*	*Ereignismuster*
Wer ist (konzeptio-nell) aktiv?	Abhängiger	Subjekt	Subjekt oder Ab-hängiger
Wieviele Abhängi-ge kann es geben?	0 bis n	0 oder 1	0 bis n
Intention beim Be-nachrichtigen vor-handen?	Nein	Ja	Nein
Unterscheidbare Ereignisse?	Nein	Ja	Ja
Art der Inkennt-nissetzung?	Signal	Nachricht	Signal oder Nach-richt

[16]Wir verzichten an dieser Stelle auf eine detailliertere Beschreibung, da diese bereits in Kapitel 2.1 gegeben wurde.

4. Probleme, Besonderheiten und Anforderungen beim Einsatz von Reaktionsmechanismen

Wir wollen in diesem Kapitel einige allgemeine Probleme betrachten, welche beim Einsatz von Reaktionsmechanismen auftreten (können). Diese hier diskutierten Problematiken sind weitgehend unabhängig von dem konkret gewählten Muster. Einige Muster erleichtern allerdings den Umgang mit ihnen.
Wir beschäftigen uns in den folgenden Abschnitten mit den folgenden Fragen:

- Welche Benachrichtigungsbeziehungen können zu einem Fehlverhalten des Systems führen?
- Auf welcher formalen und theoretischen Grundlage können Benachrichtigungen modelliert werden?
- Wie soll auf Benutzeraktionen reagiert werden?

4.1. Ungünstige Benachrichtigungsstrukturen

Wir sprechen in diesem Abschnitt ganz bewußt von *ungünstigen* und nicht von *ungültigen* Benachrichtigungsstrukturen, da die gemeinten Benachrichtigungsstrukturen oftmals nicht zu einem Fehlverhalten des Systems führen. Dieses kann später auftreten, wenn das System erweitert wird. Diese Eigenschaft macht diese Strukturen besonders schwer handhabbar, da ein offenbar funktionierendes System durch eine beliebig kleine Änderung instabil werden kann und ein schwer nachvollziehbares Verhalten zeigt. Daher sollte man sich der potentiellen Widrigkeiten bewußt werden, welche bestimmte Systemstrukturen nach sich ziehen.

Die reaktive Änderung

Unter einer *reaktiven Änderung* verstehen wir eine Änderung, welche als Reaktion auf eine Benachrichtigung an dem benachrichtigendem Objekt (Subjekt) vorgenommen wird.

Eine reaktive Änderung läge z.B. vor, wenn die Funktionskomponente ihrer Interaktionskomponente eine Zustandsänderung signalisiert und die Interaktionskomponente als Reaktion auf diese Benachrichtigung den Zustand der Funktionskomponente durch einen Prozeduraufruf verändert.

Eine reaktive Änderung kann gleich zwei Arten von Fehlverhalten hervorrufen:

- Unendliche Änderung-Benachrichtigungs-Schleifen (Change-Update-Loops) können entstehen, wenn ein Objekt A ein Objekt B über eine Zustandsänderung benachrichtigt und das Objekt B als Reaktion darauf das Objekt A verändert, welches dann wiederum das Objekt B von einer Zustandsänderung informiert usw.
- *Ungültige Objektzustände* erhalten wir, wenn ein Objekt A mehrere Objekte B_1 bis B_n (in aufsteigender Reihenfolge) von einer Zustandsänderung benachrichtigt und ein Objekt B_i daraufhin das Objekt A verändert. Die Objekte B_{i+1} bis B_n finden das Objekt A jetzt nicht mehr in dem Zustand vor, welchen Objekt A immer noch signalisiert.[17]

---

[17]Es ist hier anzumerken, daß dieser Fall nur dann zu einem Fehlverhalten führen kann, wenn mit einem Ereignis-Mechanismus gearbeitet wird und nicht mit einem einfachen Änderungs-Mechanismus, da im letzteren Fall die Benachrichtigung auch für

Um dem oben beschriebenen Fehlverhalten vorzubeugen, gibt es zwei Richtlinien für den Entwurf von Klassen, welche an Reaktionsbeziehungen teilnehmen ([Rie95] und [Rie95a][18]).

> ***Richtlinie für das Design abhängiger Klassen***
> **Entwerfe und implementiere abhängige Klassen immer so, daß sie als Reaktion auf eine Benachrichtigung nur sondierende Operationen auf dem Subjekt aufrufen, aber niemals Prozeduren mit Seiteneffekten.**
> **Diese Richtlinie ist transitiv anzuwenden, in dem Sinne, daß das Subjekt auch nicht indirekt über andere Objekte verändert werden darf.**

> ***Richtlinie für das Design von Subjekten***
> **Entwerfe und implementiere Subjekte immer so, daß eine klare Unterscheidung zwischen sondierenden Operationen (Funktionen) und verändernden Operationen (Prozeduren) möglich ist.**

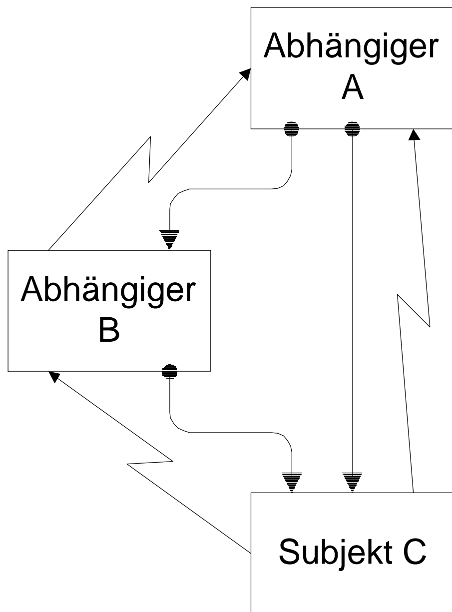
Der eigentliche Kern befindet sich in der ersten Richtlinie. Die zweite Richtlinie stellt nur sicher, daß die abhängige Klasse auch zwischen sondierenden und verändernden Operationen unterscheiden kann. Die Einhaltung der zweiten Richtlinie gehört sowieso "zum guten Ton" in der Softwareentwicklung und sollte daher auf alle Komponenten angewendet werden.

Die erste Richtlinie hat jedoch einen Schwachpunkt, welcher sie für den praktischen Einsatz unpraktikabel werden läßt: Die transitive Anwendung der Richtlinie führt dazu, daß immer die Gesamtstruktur des Systems betrachtet werden muß und Subkomponenten nicht einfach in komplexere Komponenten "eingestöpselt" werden können. Es muß immer wieder an der entstandenen Gesamtstruktur überprüft werden, ob diese Struktur noch der Richtlinie genügt. Es kann sogar der Fall auftreten, daß eine Subkomponente nicht ohne Änderung in eine komplexere Komponente eingebunden werden kann.

---

den neuen Objektzustand Gültigkeit hätte.

[18]Riehle schreibt, er habe von diesen Richtlinien zum ersten mal durch Karl-Heinz Sylla erfahren, welcher sie in einer Diskussion nannte.



Ungünstige Dreiecks-Reaktionsbeziehung

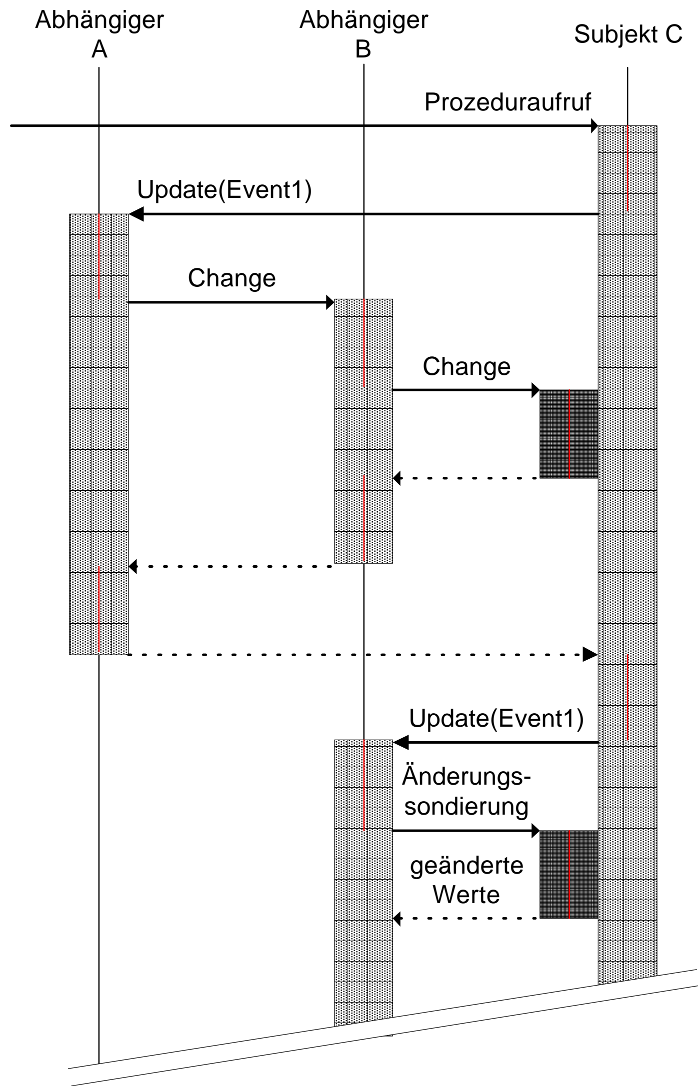
Wenn eine Reaktionsbeziehung existiert, welche eine Dreiecks-Reaktionsbeziehung wie in der Abbildung dargestellt, enthält, kann es zu einer *indirekten reaktiven Änderung* kommen. Dabei spielt es keine Rolle, ob das Objekt A direkt oder indirekt auf das Subjekt C zugreift. Wichtig ist nur, daß es zwei Benutzungswege von A nach C und umgekehrt zwei Benachrichtigungswege von C nach A gibt. Das folgende Interaktionsdiagramm veranschaulicht den prinzipiellen Ablauf einer indirekten reaktiven Änderung.

Der Subjektzustand wird durch einen Prozeduraufruf verändert.

Daraufhin wird Objekt A benachrichtigt, welches als Reaktion eine Veränderung an Objekt B vornimmt.

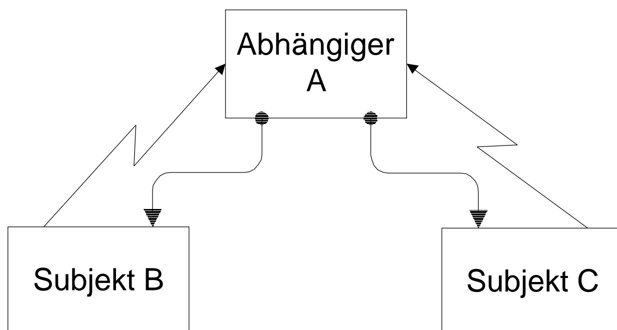
Objekt B verändert wiederum das Subjekt C. An dieser Stelle haben wir bereits eine *indirekte reaktive Änderung.*

Jetzt wird Objekt B von der ursprünglichen Änderung benachrichtigt.
Beim Sondieren findet B das Subjekt C nicht mehr in dem signalisierten Zustand vor.



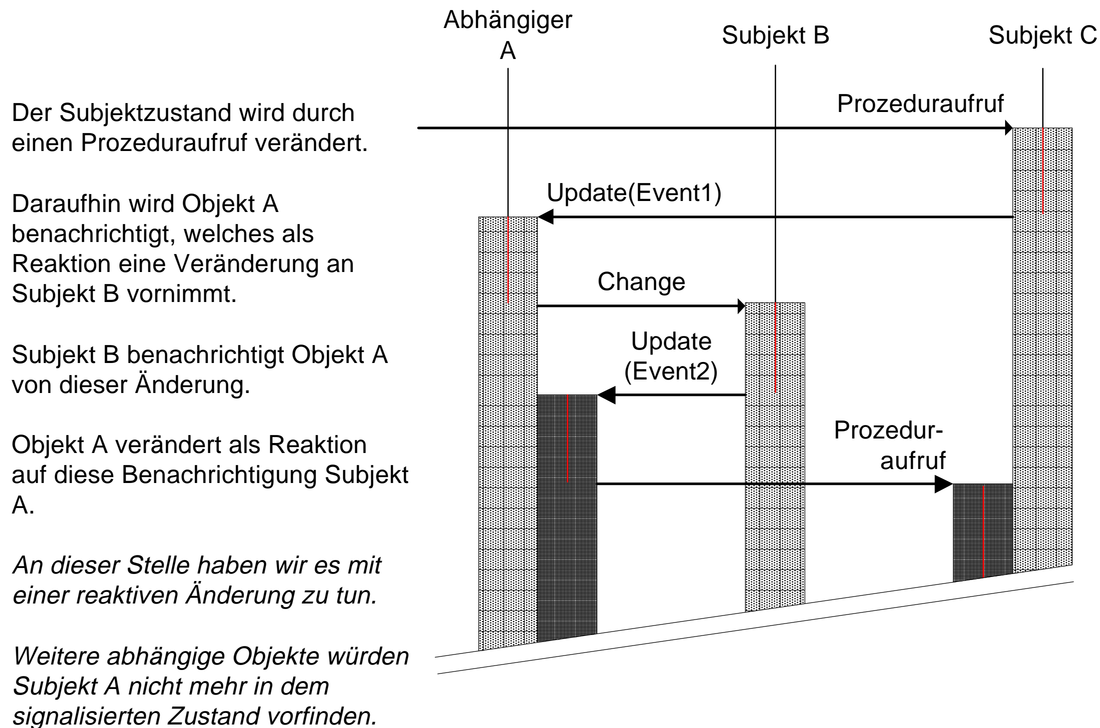
Eine indirekte reaktive Änderung am Subjekt

Ein weiteres Problem kann auftreten, wenn ein abhängiges Objekt von mehreren Subjekten abhängt, wie in dem folgenden Klassendiagramm dargestellt:



Klassendiagramm: Mehrfache Abhängigkeit

Hängt ein abhängiges Objekt A von den Subjekten B und C ab, so kann es passieren, daß A von C benachrichtigt wird und daraufhin B ändert, welches A von dieser Änderung benachrichtigt. Als Reaktion auf diese Benachrichtigung von B an A ändert A jetzt C, welches sich immer noch im Zustand des Signalisierens befindet. Dies haben wir in dem folgenden Interaktionsdiagramm dargestellt:



Direkte reaktive Änderung

Es ist somit klar, daß es keinen Sinn macht, nach einer Konstruktionsvorschrift für reaktive Komponenten zu suchen. Diese müßte unter anderem verhindern, daß ein abhängiges Objekt von mehreren Subjekten abhängig ist. Wenn eine solche Abhängigkeitsbeziehung nicht mehr erlaubt wäre, würde dies die Möglichkeiten bei der Konstruktion reaktiver Komponenten zu sehr einschränken.

Eine Erleichterung zum Finden ungünstiger Reaktionsstrukturen kann hier ein Mechanismus bringen, welcher dynamisch prüft, ob eine reaktive Änderung stattgefunden hat und ggfs. eine Ausnahmebehandlung auslöst.
Voraussetzung für einen solchen Mechanismus ist, daß bei jeder Änderung des abstrakten Subjektzustandes eine Benachrichtigung stattfindet. In diesem Fall kann bei der Benachrichtigung geprüft werden, ob sich das Subjekt bereits im Zustand des Signalisierens befindet. Ist dies der Fall, so liegt offenbar ein Fehlverhalten vor und es wird eine Ausnahmebehandlung ausgelöst (in der Regel ein Programmabbruch mit Ausgabe einer Fehlermeldung).

Wenden wir uns an dieser Stelle einem weiteren Problem zu, welches bei komplexen reaktiven Systemen auftreten kann, dem Ereignis-Wettlauf:

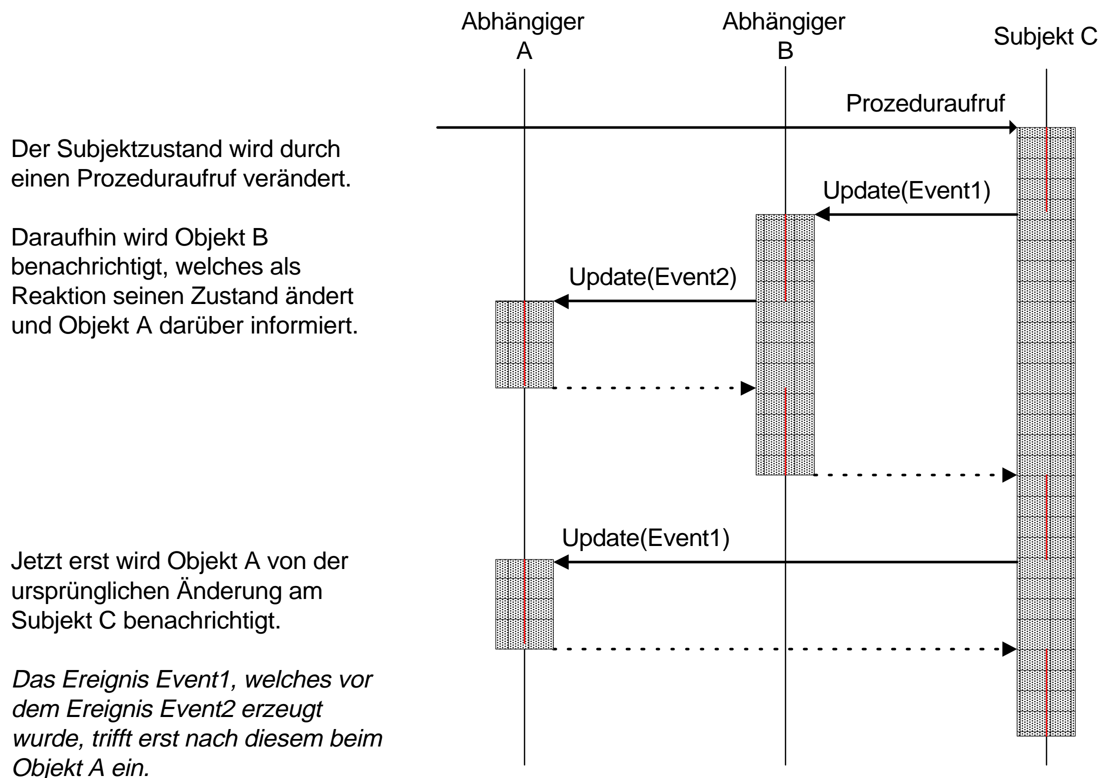
Der Ereignis-Wettlauf (Event-Race)

Von einem *Ereignis-Wettlauf* sprechen wir, wenn während der Benachrichtigung eines Ereignisses E_1 ein weiteres Ereignis E_2 ausgelöst wird und das Ereignis E_2 das Ereignis E_1 überholen kann.

Ein solcher Ereignis-Wettlauf kann z.B. auftreten, wenn ein "Dreiecks"-Benachrichtigungsbeziehung vorhanden ist: Objekt A benachrichtigt Objekt B und C. Als Reaktion auf die Benachrichtigung benachrichtigt Objekt B das Objekt C. Objekt C wird somit sowohl von einer Zustandsänderung in A, wie auch von einer Zustandsänderung in B benachrichtigt. Entsprechend ihrer chronologischen Reihenfolge, sollte zuerst das Ereignis des Objektes A und danach das Ereignis des Objektes B gemeldet werden.

Auch wenn wir hier von einem Ereignis-Wettlauf sprechen, kann dieses Problem genauso bei einem einfachen Änderungs-Mechanismus wie bei einem Ereignis-Mechanismus auftreten.

Interessanterweise kann dieser Fall ebenfalls nur dann auftreten, wenn zwei Benutztwege von A nach C existieren und umgekehrt zwei Benachrichtigungswege von C nach A. Das folgende Interaktionsdiagramm veranschaulicht dies:



Ereignis-Wettlauf

Es muß also sichergestellt werden, daß die chronologische Reihenfolge der Benachrichtigungen nicht verändert wird.
Dies kann durch den Einsatz eines *Change-Managers*[19] geschehen.

Der Change-Manager ist für die globale Benachrichtigungs-Flußkontrolle zuständig. Alle Benachrichtigungen werden beim Change-Manager angemeldet, welche dieser in einer Queue verwaltet. Entsprechend ihres chronologschen Auftretens, werden die Benachrichtigungen vom Change-Manager aktiviert. Erst zu diesem Zeitpunkt, werden die Änderungen tatsächlich an die abhängigen Objekte gemeldet. Ein Interaktionsdiagramm, das den Einsatz des Event-Managers demonstriert, findet sich in Abschnitt 5.5[20].

4.2. Ein Leitbild für den Reaktionsmechanismus

Eine Frage, welche sich beim Entwurf von Subjekten regelmäßig stellt, betrifft die Zeitpunkte der Benachrichtigung der abhängigen Objekte. Wir benötigen hierfür ein Leitbild, an welchem wir uns beim Einsatz des Reaktionsmechanismusses orientieren können. Hier ist die Frage zu klären, bei welchen Veränderungen das Subjekt seine abhängigen Objekte benachrichtigen soll.
Es ist sicherlich einzusehen, daß das Subjekt nicht bei jeder noch so kleinen Änderung seines konkreten Zustandes seine abhängigen Objekte benachrichtigen kann und soll. Dies kommt für die meisten Subjekte schon allein aus Komplexitätsgründen nicht in Frage. Außerdem hat es wohl wenig Sinn, die abhängigen Objekte von einer Zustandsänderung zu unterrichten, wenn diese nur nicht-öffentliche Instanzvariablen betrifft, welche auch nicht durch Funktionen zugegriffen werden können und deshalb von den abhängigen Objekten gar nicht nachvollzogen werden kann.
Die Lösung, welche in den meisten Fällen angewendet wurde, war schließlich auch eine rein pragmatische: Die abhängigen Objekte wurden immer dann benachrichtigt, wenn dies für den aktuellen Kontext gerade notwendig schien. Dadurch entsteht allerdings das Problem, daß es schwierig ist, neue abhängige Objekte hinzuzufügen, da dazu fast immer auch in das Subjekt eingegriffen werden muß, um die jetzt notwendigen Benachrichtigungen einzubauen.
Es fehlt also ein Leitbild, welches eine Anleitung gibt, wann eine Benachrichtigung zu erfolgen hat. Dafür ist es notwendig, daß das Subjekt "isoliert" von den abhängigen Objekten betrachtet und entworfen wird:

Leitbild
Abhängige Objekte werden vom Subjekt immer dann benachrichtigt, wenn sich der *abstrakte* Zustand des Subjektes geändert hat.

Diese Aussage hat zwei Hauptaspekte.

- Zum einen erfolgt eine Benachrichtigung immer dann, wenn sich der *akstrakte Zustand* des Subjektes ändert.

---

[19] Bei Ereignis-Mechanismen sprechen wir auch von *Event-Manager*.

[20] Das Problem des Ereignis-Wettlaufes tritt auch auf, wenn man die Ereignisse mit den geänderten Werten parametrisiert und nicht mehr beim Subjekt sondieren muß.

- Zum anderen soll mit der Benachrichtigung der abhängigen Objekte erst dann begonnen werden, wenn die Veränderung *vollständig abgeschlossen* ist. Nur so kann sichergestellt werden, daß die abhängigen Objekte beim Sondieren des Subjektes einen konsistenten Subjektzustand vorfinden.

Der abstrakte Zustand ist vom konkreten Zustand zu unterscheiden. Der konkrete Zustand wird durch die aktuellen Werte aller Instanzvariablen des Subjektes gebildet. Der abstrakte Zustand wird durch den *fachlichen Zustand* des Subjektes auf Grundlage der *Umgangsformen* gebildet. Das Problem besteht nun darin, die abstrakten Zustände und ihr Zusammenspiel formal festzulegen und zu definieren. Wir werden im folgenden Abschnitt mittels State-Charts eine grafische Darstellungsmöglichkeit abstrakter Zustände vorstellen.

Neben der hier vorgestellten Möglichkeit, bei jeder Änderung des abstrakten Subjektzustandes die abhängigen Objekte zu benachrichtigen, gäbe es auch noch zwei weitere Möglichkeiten, welche hier nur kurz angedeutet werden sollen:

- Bei jeder *Änderung einer Instanzvariablen* werden die abhängigen Objekte benachrichtigt. Hierbei ist aber sicherlich eine Unterscheidung in *relevante* und *nicht relevante* Instanzvariablen zu treffen. Man könnte z.B. definieren, daß alle die Instanzvariablen *relevant* sind, welche von außen abfragbar sind.
  Dies würde allerdings dazu führen, daß viel zu oft Änderungen an die abhängigen Objekte gemeldet werden, da die Änderung *einer* Instanzvariablen das Resultat *mehrerer* Funktionen beeinflussen kann. Außerdem ist es möglich, daß abhängige Objekte benachrichtigt werden, während sich das Subjekt in einem inkonsistenten Zustand befindet, weil z.B. eine Änderung mehrere Instanzvariablen betreffen kann und diese Änderung als *atomar* anzusehen ist.
- Bei jedem *Aufruf einer öffentlichen Prozedur* werden die abhängigen Objekte benachrichtigt. Hierhinter verbirgt sich die Annahme, daß jeder Aufruf einer öffentlichen Prozedur *relevante* Instanzvariablen verändert.
  In komplexeren Klassen kommt es jedoch häufig vor, daß Prozeduren weitere öffentliche Prozeduren derselben Klasse aufrufen, um eine bestimmte Aufgabe zu erledigen. Auch beim Aufruf dieser Prozeduren würden die abhängigen Objekte benachrichtigt. Auf diese Weise würde man erstens zu viele Ereignisse[21] versenden und zweitens *einen Teil der Klassenimplementation nach außen sichtbar machen*.
  Dieses Problem kann beseitigt werden, wenn man festlegt, daß öffentliche Prozeduren keine weiteren öffentlichen Prozeduren ihrer Klasse aufrufen dürfen.
  Bei diesem Ansatz hat man weiterhin das Problem, daß die versendeten Ereignisse ihre Namen auf Grundlage der Prozedurnamen erhalten würden und deshalb keinen Rückschluß auf den fachlichen Zustand des Subjektes zulassen würden.

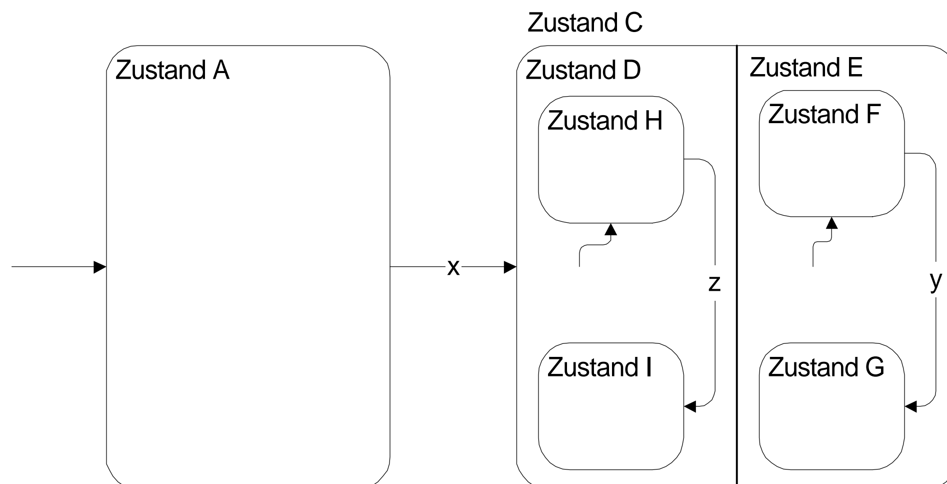
---

[21] Man stelle sich hierzu einen Auflister vor, welcher eine Liste von Einträgen enthält und einen Knopf zum Entleeren der gesamten Liste zur Verfügung stellt. Bei Betätigung dieses Knopfes wird im Lister eine Prozedur `clear` gerufen. Existiert in der vom Lister verwendeten Liste keine Prozedur zum Entleeren der gesamten Liste, muß in einer Programmschleife die gesamte Liste durch wiederholtes Aufrufen der Prozedur `remove` entleert werden. Bei jedem Aufruf von `remove` würden die abhängigen Objekte benachrichtigt.

Auch wenn wir diese beiden Möglichkeiten für *nicht adäquat* halten, um eine Grundlage für das Versenden von Ereignissen darzustellen, wollen wir dennoch darauf hinweisen, daß beide Möglichkeiten einen Vorteil haben: Das Versenden der Ereignisse könnte prinzipiell durch einen Automatismus geschehen, welcher z.B. in der Programmiersprache vorgesehen sein könnte.

4.3. State-Charts

State-Charts (nach [HAR88]) stellen eine Erweiterung der Zustandsübergangsdiagramme (nach [Min67]) dar. Wie diese stellen sie das dynamische Verhalten einer Komponente dar. Es werden die Zustände der Komponente und die Zustandsübergänge zwischen ihnen visualisiert. Im Gegensatz zu den Zustandsübergangsdiagrammen sind State-Charts in der Notation reichhaltiger und daher einfacher zu handhaben.
Zustände werden durch abgerundete Rechtecke dargestellt, in welchen die Zustandsbezeichnung steht. In Erweiterung zu den Zustandsübergangsdiagrammen ist es möglich, Zustände hierarchisch zu verfeinern. Dazu werden die Unterzustände in den Oberzustand gezeichnet. Wird ein Zustand durch Linien geteilt, so hat dieser orthogonale Subzustände, welche alle gleichzeitig aktiv sind, wenn der Oberzustand aktiv ist. In diesem Fall wird der Name des Oberzustandes über den Zustand geschrieben.



State-Chart

In diesem Beispiel gibt es einen sehr komplexen Zustand C. Dieser hat zwei orthogonale Subzustände D und E. Wenn der Zustand C aktiv ist, sind auch immer die beiden Zustände D und E aktiv. In D und E befinden sich jeweils wieder zwei Subzustände, welche allerdings nicht orthogonal sind. Es kann jeweils nur einer von ihnen aktiv sein. Pfeile mit einem ausgefüllten unbenannten Anfangszustand und ohne Prozedurnamen bezeichnen spontane Standard-Übergänge. Ein unbenannter Anfangsbuchstabe mit einem ‘H’ in der Mitte bezeichnet einen spontanen Standard-Übergang mit Historie. Wir verdeutlichen dies anhand eines möglich Ablaufes:
Wenn das beschriebene Objekt erzeugt wird, findet der spontane Übergang (links) statt, so daß sich das Objekt im Zustand A befindet. Wird die Prozedur x gerufen, werden gleichzeitig die Zustände C, D, E, H und F aktiv. Durch einen Aufruf der Prozedur y wird F inaktiv und G aktiv. Wird z gerufen, wird H inaktiv und I aktiv. Bei einem Zustands-

wechsel von H nach I bzw. von F nach G bleiben die umschließenden Zustände auch immer aktiv. Wird Zustand C verlassen und danach wieder betreten, so wird nicht unbedingt Zustand F aktiv, sondern der Zustand, welcher zuletzt aktiv war.

Prinzipiell sind State-Charts nicht-deterministisch. Es ist möglich, daß zwei gleichnamige Zustandsübergänge von gleichen Zustand parallel in mehrere verschiedene Endzustände mündet. In der Praxis tritt dieser Fall sogar relativ häufig auf. Solche nicht-deterministischen State-Charts können deterministisch gemacht werden, indem hinter den Namen der Zuständsübergänge Prädikate (boolsche Ausdrücke) angegeben werden, welche bestimmen, unter welchen Bedingungen welcher Zustandsübergang gewählt wird. In den Prädikaten können zwei Arten von Variablen auftreten:

- Zustände im State-Chart: Zustandsübergänge können davon abhängig gemacht werden, ob gerade bestimmte Zustände aktiv sind.
- Konkrete Variablen/Funktionsresultate: Abhängig von den konkreten Werten von Programmvariablen und Funktionsresultaten[22] können bestimmte Zustandsübergänge vollzogen werden.

Als Leitbild für Benachrichtigung haben wir abstrakte Zustandsänderung (Kapitel 4.2.) eingeführt, mit dem Ziel, die von den Subjekten vorgenmmenen Benachrichtigungen unabhängig von den abhängigen Objekten modellieren zu können. Um diese abstrakten Zustände und ihre Übergänge zu definieren, wollen wir State-Charts benutzen, wie dies von [Rie95a] vorgeschlagen wird. Für jede Subjektklasse wird ein State-Chart definiert. Nur so kann eine bidirektionale Transformation zwischen Klassenschnittstelle und State-Chart vorgenommen werden.
Bei jeder Änderung eines abstrakten Subjektzustandes (entspricht im State-Chart einem Zustandsübergang) sollen die abhängigen Objekte benachrichtigt werden. Wir definieren pro abstraktem Zustand ein zugehöriges Ereignis, welches immer dann ausgelöst wird, wenn der abstrakte Zustand erreicht wird. Dies soll auch dann geschehen, wenn der Anfangs- und Endzustand eines Zustandsüberganges identisch sind[23].

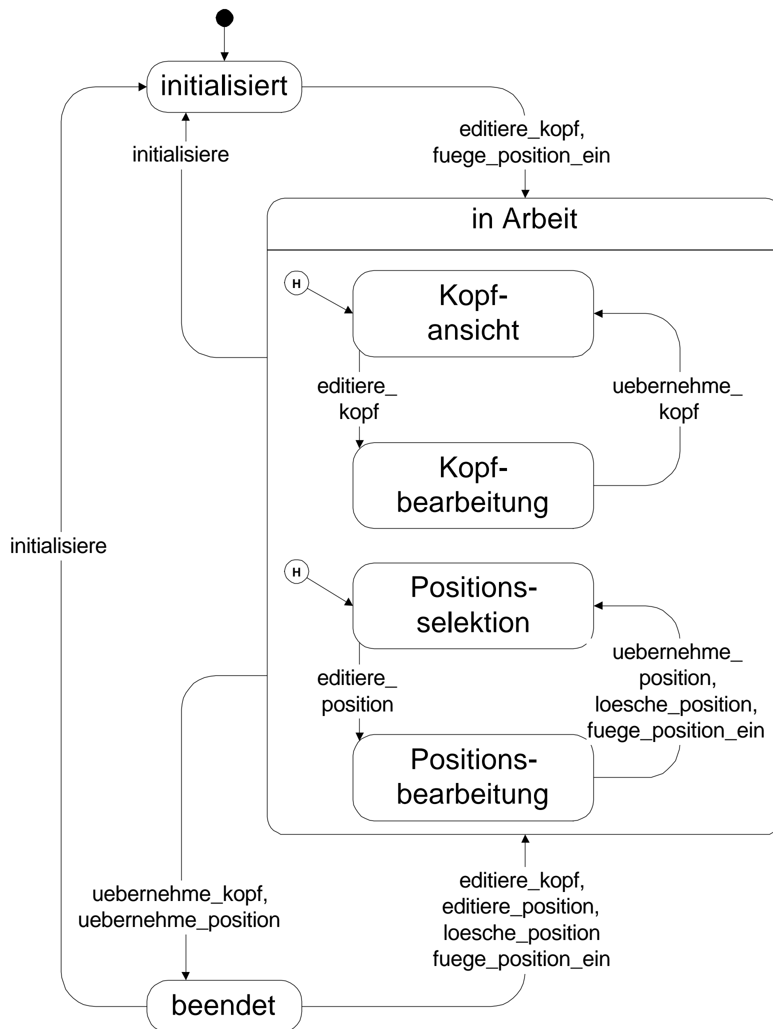
4.3.1. Fachliche Modellierung von State-Charts

Die modellierten abstrakten Zustände sollten fachlich motiviert sein, so daß man ein fachliches State-Chart erhält. State-Charts können auf der Grundlage des Objekt-Lebensweges (Object-Life-Cycle) erstellt werden. Dazu definiert man die wichtigen fachlichen Zustände in der Lebenszeit eines Objektes als abstrakte Zustände. Die relevanten fachlichen Prozeduren werden in Zustandsübergänge abgebildet.

Wir haben hier einmal den Rechnungseditor aus Kapitel 2 als State-Chart modelliert:

[22] In objektorientierten Systemen benutzt man in der Regel Exemplarvariablen oder Funktionsresultate.

[23] Im State-Chart entspricht dies einer Schleife.



State-Chart für den Rechnungseditor

Eine Rechnung kann sich in den drei Hauptzuständen *initialisiert*, *in Arbeit* und *beendet* befinden. Beim Start des Rechnungseditors befindet sich die Rechnung im Zustand initialisiert. Beginnt der Benutzer mit dem Eintrag von Rechnungsdaten oder dem Zufügen von Positionen, so wechselt das Werkzeug in den Zustand *in Arbeit*. Wenn die Rechnung fertig erstellt ist, wechselt das Werkzeug in den Zustand *beendet*. Befindet man sich im Zustand *in Arbeit*, so befindet man sich zusätzlich noch in zwei der vier Subzustände. Entweder man sieht sich die Rechnungskopfdaten an oder man editiert sie; entweder man selktiert eine der Positionen oder man editiert eine der Positionen.
An dem Beispiel kann man sehen, daß es keinen Sinn macht, bei der Modellierung des State-Charts den Zustand des Werkzeuges und den Zustand des gekapselten Materials voneinander zu trennen. Außerdem ist aus dem Beispiel ersichtlich, wie nützlich die Historien-Anfangszustände sein können. Eine Modellierung ohne diese wäre sehr unübersichtlich und umfangreich geworden.

4.3.2. Spezialisierung in State-Charts

Da Zustände in State-Charts *hierarchisch* angeordnet sein können, können bei einem Prozeduraufruf *mehrere Zustandsübergänge* vollzogen werden. Weil wir in einem Einprozeßraum nicht mehrere Zuständsübergänge gleichzeitig signalisieren können, müssen sie sequentiell signalisiert werden und zwar zuerst die allgemeinen, gefolgt von den speziellen Zustandsübergängen.

Es stellt sich hier die Frage, in welchem Zustand sich das signalisierende Objekt befindet, wenn das allgemeine Ereignis, aber noch nicht das spezielle Ereignis verschickt wurde. Ist der spezielle Zustandsübergang bereits vollzogen, oder noch nicht? Es gibt hier drei Implementationsvarianten:

- *Alle Ereignisse werden am Ende der sendenden Prozedur verschickt*. In diesem Fall ist der spezielle Zustandsübergang bereits beim Signalisieren des allgemeinen Zustandsüberganges vollzogen. Um dieses Verhalten zu erreichen, müssen mitunter erhebliche Anstrengungen bzgl. der Implementation unternommen werden, um die Erweiterung via Vererbung nicht einzuschränken. Man wäre dann gezwungen, Prozeduren von Superklassen immer am Ende der aufrufenden Prozedur, aber noch vor Versenden des spezielleren Events aufzurufen. Oder es müssen andere aufwendige Vorkehrungen bereits in der Superklasse vorgenommen werden.

```
class SUBJEKT
     betrete_zustand_a is
     do
          -- Aktionen
          event_a.announce
     end
end

class UNTER_SUBJEKT
inherit SUBJEKT
     betrete_zustand_b is
     -- b ist Subzustand von a
     -- diese Prozedur benutzt 'betrete_zustand_a', um
     -- seine Aufgabe zu erledigen
     do
          -- Aktionen
          betrete_zustand_a   -- dieser Aufruf muß an
                              -- dieser Stelle stehen
          event_b.announce
     end
end
```

- *In einer Prozedur wird erst der allgemeine Zustandsübergang vollzogen und per Ereignis bekanntgemacht und danach der spezielle*. In diesem Fall ist der spezielle Zustandsübergang noch nicht vollzogen, wenn der allgemeine Zustandsübergang bekannt gemacht wird. Diese Variante ist einfacher als die erste zu implementieren. Allerdings muß hier beachtet werden, daß vor dem Aufruf einer Prozedur der Superklasse keine Veränderungen vorgenommen werden, welche das Objekt bzgl. des spezielleren Zustandes inkonsistent werden lassen.

```
class SUBJEKT
     betrete_zustand_a is
     do
          -- Aktionen
          event_a.announce
     end
end

class UNTER_SUBJEKT
inherit SUBJEKT
     betrete_zustand_b is
     -- b ist Subzustand von a
     -- diese Prozedur benutzt 'betrete_zustand_a', um
     -- seine Aufgabe zu erledigen
     do
          betrete_zustand_a   -- dieser Aufruf muß an
                              -- dieser Stelle stehen
          -- Aktionen
          event_b.announce
     end
end
```

- *Undefiniert*. In der Implementation wird das allgemeine Ereignis dann verschickt, wenn der allgemeine Zustandsübergang vollzogen ist. Dies kann vor dem speziellen Zustandsübergang geschehen, danach oder währenddessen. Dies hat zur Folge, daß beim Versenden des allgemeinen Ereignisses keine Aussage über den Vollzug des speziellen Zustandsüberganges gemacht werden kann. Diese Implementation ist am leichtesten durchzuhalten, hat allerdings das Problem, daß man als Reaktion auf Erhalt des allgemeinen Ereignisses nicht bzgl. des spezielleren Zustandes sondieren darf, da dieser evtl. undefiniert und inkonsistent ist.

```
class UNTER_SUBJEKT
inherit SUBJEKT
     betrete_zustand_b is
     -- b ist Subzustand von a
     -- diese Prozedur benutzt 'betrete_zustand_a', um
     -- seine Aufgabe zu erledigen
     do
          -- Aktionen erster Teil
          betrete_zustand_a   -- dieser Aufruf kann
                              -- irgendwo vor
                              -- event_b.announce
                              -- stehen
          -- Aktionen zweiter Teil
          event_b.announce
     end
end
```

Wenn ein abhängiges Objekt von einer Änderung benachrichtigt wird und somit die Kontrolle erhält, sollte garantiert sein, daß das Subjekt bzgl. *jedes* Zustandes konsistent ist, was mit den letzten beiden Lösungen gerade nicht möglich ist. Es ist hier also die erste Lösung zu wählen, auch wenn sich dadurch im Zusammenhang mit Vererbung möglicherweise Probleme ergeben.

4.3.3. Zusammenfassung State-Charts

Bei der Konstruktion objektorientierter Software kann mehrfacher Nutzen aus State-Charts gezogen werden:

- State-Charts spezifizieren die abstrakten Zustände von Objekten. Auf ihrer Grundlage kann das konkrete Verhalten von Klassen implementiert werden.
- State-Charts sind zur Dokumentation des Verhaltens von Klassen geeignet.
- Auf Grundlage der mittels State-Chart festgelegten abstrakten Zustände kann die Benachrichtigung abhängiger Objekte auf einer formal sauberen Grundlage modelliert werden, ohne konkrete Kenntnis der abhängigen Objekte.
- Auf Grundlage von State-Charts können Vor- und Nachbedingungen für Routinen sowie Klasseninvarianten formuliert werden[24]. Im Sinne des Vertragsmodells ist es durchaus sinnvoll, auch "Zustandsübergänge" für Funktionen zu modellieren. Es muß sich hierbei immer im Schleifen handeln. Da Ausgangs- und Endzustand bei Funktionen immer identisch sind, schlagen wir vor, diese durch Schleifen ohne Pfeilspitze an die jeweiligen Zustände zu zeichnen. Ein Funktionsübergang darf natürlich nicht das Versenden von Ereignissen zur Folge haben. Um das State-Chart nicht unnötig kompliziert zu machen, sollten nur Funktionen an die Zustände geschrieben werden, welche nicht in jedem Zustand aufgerufen werden können (partielle Funktionen). Totale Funktionen sollten neben oder unter dem eigentliche State-Chart notiert werden.

  Es ist jedoch noch eine offene Frage, ob Vor- und Nachbedingungen alle Informationen enthalten sollten, welche im State-Chart verhanden sind. Dazu wäre es notwendig, die State-Chart-Zustände als Funktionen in der Klasse zu modellieren.

Weiteres zu State-Charts in objektorientierten Systemen findet sich in [RBP+91].

4.4. Ereignis-Parameter

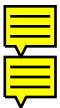
Wie wir oben beschrieben haben, werden die abhängigen Objekte vom Subjekt immer dann benachrichtigt, wenn sich der abstrakte Zustand des Subjektes geändert hat. Als Reaktion auf eine Benachrichtigung wird das abhängige Objekt in den meisten Fällen das Subjekt sondieren, um die geänderten Werte in Erfahrung zu bringen. Wenn man dem abhängigen Objekt bei der Benachrichtigung die geänderten Werte mitgäbe, so könnte dieses Sondieren entfallen[25].

---

[24] Dieser Punkt sei hier nur als Hinweis gedacht. An dieser Stelle ist unserer Meinung nach noch einiges an Arbeit notwendig.

[25] Nach unserer in Kapitel 2 gegebenen Definition handelt es sich bei parametrisierten Ereignissen um *Nachrichten*.

Es steht außer Frage, daß sich bei verschiedenen Zustandsänderungen verschiedene geänderte Werte ergeben. Das Parametrisieren von Ereignissen mit den geänderten Werten ist also nur dann sinnvoll, wenn verschiedene Ereignisse unterschieden werden, wie dies z.B. bei dem EVENT-NOTIFICATION-Muster der Fall ist.
Möglicherweise ließe sich durch Ereignis-Parameter eine Verbindung zwischen abstraktem und konkretem Subjektzustand herstellen und explizit machen. Wird ein Element aus einer Liste ausgewählt, geht das Subjekt in den abstrakten Zustand *selektiert* über und benachrichtigt seine abhängigen Objekte darüber. Bei dieser Benachrichtigung gibt das Subjekt z.B. einen Integerwert mit, welcher die Nummer des selektierten Elementes angibt. Die abhängigen Objekte können jetzt sofort auf die Veränderung reagieren, ohne das Subjekt sondieren zu müssen.
So interessant diese Möglichkeit auch sein mag, sie birgt auch einige Gefahren:
Für unerfahrene Entwickler wird möglicherweise die Versuchung sehr groß sein, diesen Mechanismus als eine andere Art des Prozeduraufrufes zu verstehen und zu benutzen. Dies hätte dann die unausweichliche Folge verminderter Wiederverwendbarkeit und Erweiterbarkeit.
Weiterhin stellt sich die Frage, ob die oben beschriebene Abbildung zwischen abstraktem und konkretem Zustand immer so ohne weiteres beschrieben werden kann. Wenn dies nicht gelingt, wären die Event-Parameter rein willkürlich und müßten oft geändert werden.
Wir können hier kein abschliessendes Urteil über die Praktikabilität von Event-Parametern abgeben. Hier ist sicherlich weitere Arbeit und vor allem das Sammeln von Erfahrungen notwendig.
Ein potentieller Vorteil von Ereignis-Parametern soll hier jedoch noch kurz angeschnitten werden: Wenn es möglich ist, eina Abbildung zwischen konkreten und abstrakten Zuständen durchzuführen, so könnte das Sondieren des Subjektes vom abhängigen Objekt entfallen. Gleichzeitig entfiele damit das Problem der reaktiven Änderung. Wenn die abhängigen Objekte das Subjekt nicht sondieren, können sie es auch nicht in einem veränderten oder inkonsistenten Zustand vorfinden.



4.5. Reaktion auf Benutzeraktion

In diesem Abschnitt wollen wir untersuchen, wie auf Benutzereingaben reagiert werden kann. Es gibt in objektorientierten Systemen im wesentlichen zwei Ansätze. Zum einen können die Elemente der Benutzungsoberfläche[26] über einen Benachrichtigungsmechanismus a la OBSERVER oder EVENT-NOTIFICATION angeschlossen werden. Zum anderen können Kommando-Objekte[27] Verwendung finden.
Die Praxis zeigt, daß beide Wege möglich sind. Trotzdem tendieren wir zur Verwendung von Kommando-Objekten. Dies hat die folgenden Gründe:

- Das COMMAND-Muster bietet eine elegante Möglichkeit zur Realisierung von Undo- und Redo-Funktionalität.
  Verzichtet man auf das COMMAND-Muster, muß die Undo- und Redo-Funktionalität über andere Muster wie z.B. MEMENTO realisiert werden. Dieses Muster hat aber z.B. den Nachteil, daß es das Geheimnisprinzip an einigen Stellen aufbricht.

---

[26]Im Sprachgebrauch der Werkzeug&Material-Metapher sprechen wir hier von *Interaktionstypen (IAT)*. Üblich ist außerdem der Begriff *Widget*.

[27]Siehe OBSERVER-Muster weiter oben.

Bei Verwendung des EVENT-NOTIFICATION-Mustern bestünde die Möglichkeit, Undo- und Redo-Funktionalität in das Muster zu integrieren. Dadurch wüchse aber Umfang und Mächtigkeit des Musters an. Es stellt sich hier die Frage, ob das Muster dadurch nicht zu groß und unhandlich wird.

- Die Reaktionsbeziehung zwischen Oberflächenelementen und dem eigentlichen Softwaresystem unterscheidet sich stark von den Reaktionsbeziehungen innerhalb eines Softwaresystems:
  - Auf Veränderungen von Oberfächenelementen reagiert immer nur *ein* Objekt, während allgemein auf die Veränderung eines Subjektes beliebig viele abhängige Objekte reagieren können.
  - Wie wir oben gesehen haben, ist es ungünstig, eine reaktive Änderung an einem Subjekt vorzunehmen. Eine reaktive Änderung an einem Oberflächenelement ist aber oftmals notwendig. Dies ist auch nicht weiter gefährlich, da niemals mehr als ein Objekt auf eine Benutzereingabe reagiert.
- Wir haben weiter oben gezeigt, daß ein Subjekt *jede* Änderung seines abstrakten Zustandes bekanntmachen soll. Dies kann zwar auch für Oberflächenelemente geschehen, ist aber nicht notwendig und außerdem umständlich. In diesem Fall, würde das Oberflächenelement das Setzen eines Textfeldes melden. Dies ist überflüssig, weil das setzende Objekt natürlich weiß, daß es das Textfeld verändert; eine entsprechende Reaktion ist nicht notwendig. Es ist umständlich, weil das verändernde Objekt bei jeder Meldung des Oberflächenelementes prüfen muß, ob die Veränderung vom Benutzer oder von ihm selbst erfolgte. Es scheint somit sinnvoll, nur solche Ereignisse zu melden, welche vom Benutzer direkt ausgelöst wurden.
- Im vorigen Abschnitt haben wir Ereignis-Parameter diskutiert und gesehen, daß ihre Verwendung mitunter problematisch sein kann. Das Verwenden von Parametern im COMMAND-Muster hingegen ist unproblematisch und kann hier eine Erleichterung bei der Softwareentwicklung mit sich bringen.

4.6. Propagieren von Benachrichtigungen

Wir wollen in diesem Kapitel untersuchen, wie Benachrichtigungen propagiert werden können.

Bei der Konstruktion objektorientierter Software ist es oftmals üblich, nicht verarbeitete Benachrichtigungen an eine Oberklasse weiterzuleiten.

Dies sieht dann in der Regel folgendermaßen aus:

```
notify (event : EVENT) is
-- nehme Benachrichtigung entgegen und verarbeite sie
do
     if event = x then
          -- do something
     else
          ^notify(event)
     end
end
```

Das Pfeil-Hoch-Zeichen ^ soll hier andeuten, daß die Prozedur der Superklasse aufgerufen wird.

Gleichzeitig ist es aber auch denkbar, Benachrichtigungen nicht nach oben sondern" zur Seite hin" zu propagieren. In diesem Fall würde die nicht verarbeitete Benachrichtigung nicht an die Superklasse gehen, sondern an das einbettende Objekt. In reaktiven System kommt es häufig vor, daß Komponenten visuell in andere Komponenten eingebettet sind[28]. In solchen Fällen scheint das Propagieren "zur Seite hin" sinnvoll.

```
notify (event : EVENT) is
-- nehme Benachrichtigung entgegen und verarbeite sie
do
     if event = x then
          -- do something
     else
          parent.notify(event)
     end
end
```

`parent` bezeichnet dabei die einbettende Komponente.

Da beide Möglichkeiten wünschenswert sind, erscheint eine Kombination sinnvoll. Wir schlagen daher vor, nicht verarbeitete Ereignisse zunächst an die Superklasse zu propagieren. In der obersten Superklasse, welche noch Kenntnis von der einbettenden Komponente hat, wird die Benachrichtigung schließlich an die einbettende Komponente propagiert, wenn sie die Benachrichtigung nicht selbst verarbeitet.

Während dies beim Observer- und beim Command-Muster problemlos funktioniert, ist das Propagieren von Ereignissen beim Event-Notification-Muster nicht ohne weiteres möglich, da es keine Standard-Benachrichtigungsprozedur geben kann, welche für die verschiedenen Parameterzahlen einsetzbar ist. Für ein Propagieren an die einbettende Komponente müßte die eingebettete Komponente die Ereignisse, welche sie propagiert, an ihrer Schnittstelle bekannt machen. Nur so kann die einbettende Komponente das jeweilige Ereignis identifizieren[29]. Hier ist sicherlich weitere Arbeit notwendig, welche den Rahmen dieser Arbeit sprengen würde.

---

[28]Ein Subwerkzeug ist z.B. häufig auch visuell in sein Kontextwerkzeug eingebettet.

[29]Auch in diesem Bereich ist noch Arbeit notwendig, welche den Rahmen dieser Arbeit sicher sprengen würde.

4.7. Anforderungen

In diesem Kapitel wollen wir unsere Anforderungen an Reaktionsmechanismen vorstellen[30]. Wir werden jeweils die in Kapitel 3 vorgesstellen Muster an ihnen messen.
Wie wir zeigen werden, entspricht keines der Muster exakt unseren Vorstellungen. Wir werden daher im nächsten Kapitel aus den positiven Eigenschaften der Muster ein neues Muster "synthetisieren", welches für unsere Zwecke besser geeignet ist.

4.7.1. Unterscheidbare Ereignisse

Unsere ursprüngliche Zielsetzung war das Konzeption eines Reaktionsmechanismusses für das Rahmenwerk am Arbeitsbereich Softwaretechnik der Universität Hamburg.
Da hier recht anspuchsvolle und komplexe Softwaresysteme zu realisieren sind, sollte der Mechanismus entsprechend mächtig sein. Da in solchen komplexen Systemen in der Regel eine Vielzahl von Ereignissen anzutreffen sind, sollte es eine Möglichkeit geben, diese zu strukturieren. Dies kann und sollte durch die Einführung unterscheidbarer Ereignisse geschehen, so daß ein abhängiges Objekt bereits aufgrund des erhaltenen Ereignisses Rückschlüsse über die Art der Änderung am Subjekt ziehen kann. Die Art und Weise der Strukturierung sollte vom Entwickler bestimmt werden. Es reicht also nicht aus, eine bestimmte Anzahl unterschiedlicher Ereignisse anzubieten. Vielmehr muß der Entwickler in der Lage sein, selbst Ereignisse zu definieren.

Das Muster sollte die Möglichkeit bieten, beliebige unterscheidbare Ereignisse zu definieren.

Das Observer-Muster bietet in seiner Grundform keine unterscheidbaren Ereignisse. Es kann zwar entsprechend erweitert werden, indem eine Unterscheidung durch Konstantenparameter angeboten wird. Aber dies hat zur Folge, daß die Ereignisse an einem zentralen, global zugänglichen Ort definiert werden müssen, was zu Schwierigkeiten bei der Zuordnung von Ereignissen zu den entsprechenden Subjekten führt.
Das Event-Notification-Muster bietet die Möglichkeit an, verschiedene Ereignisse zu definieren.

4.7.2. Explizitmachung

Wie im vorigen Abschnitt beschrieben, benötigen wir die Möglichkeiten der Strukturierung von Ereignissen, um das System überschaubarer und leichter handhabbar zu machen. Aus diesem Grund sollte auch möglichst viel des Konzeptes in der Implementation wiederzufinden sein.

Es sollte explizit gemacht werden, welche Klassen Subjektklassen sind und welche Klassen von diesen Subjektklassen abhängen. Außerdem sollte deutlich werden, über welche Ereignisse sie von den Subjekten abhängen.

Es ist also sinnvoll, Oberklassen für Subjekte und abhängige Objekte zu haben. Außerdem sollten die Ereignisse an der Schnittstelle des Subjketes deklariert werden, damit die Beziehungen zwischen Subjekt und den abhängigen Objekten deutlich werden.
Das Observer-Muster beitet die beiden Oberklassen OBSERVER und OBSERVABLE an.

---

[30]Zum Teil wurden diese Anforderungen schon im vorigen Kapitel vorgestellt. Sie seien der Vollständigkeit hier dennoch kurz wiederholt.

Das Command- und das Event-Notification-Muster bieten keine entsprechenden Oberklassen.
Beim Observer-Muster werden keine Ereignisse an der Subjektschnittstelle definiert. Beim Command-Muster geschieht dies durch das Bereitstellen der An- und Abmeldeprozeduren. Über eine entsprechende Namenskonvention kann dies noch deutlicher gemacht werden. Das Event-Notification-Muster beitet die Möglichkeit, sowohl die Ereignisse beim Subjekt sowie die zugehörigen Ereignis-Stubs bei den abhängigen Objekten explizit zu machen.

4.7.3. Netzwerkfähigkeit

Dem Bereich der Netzwerkfähigkeit kommt eine wachsende Bedeutung zu. (Auch im Arbeitsbereich Softwaretechnik wird intensiv an diesem Thema gearbeitet.) Auch wenn die Problematik noch lange nicht vollständig aufgearbeitet ist, so scheint eines bereits sicher:

Um zwischen Objekten auf verschiedenen Rechnern kommunizieren zu können, wird ein netzwerkfähiger Ereignis-Mechanismus benötigt.

Gerade weil hier noch viel Entwicklungs- und Forschungsarbeit zu leisten ist, sollte ein Reaktionsmuster zumindest die konzeptionelle Netzwerkfähigkeit beeinhalten. Dies bedeutet:

- Damit Objekte auch über weite Entfernungen miteindander kommunizieren können, muß die Möglichkeit der asynchronen Kommunikation bestehen. Dies hat wiederum die Notwendigkeit von Ereignis-Parametern zur Folge, da nicht mehr beim Subjekt sondiert werden kann.
- Der konkrete Ort der abhängigen Objekte muß vor dem Subjekt verborgen werden können.

Weder das Observer- noch das Command-Muster bieten in ihrer Konzeption eine entsprechende Möglichkeit an. Das Event-Notification-Muster bietet über die Event-Links eine entsprechende konzeptionelle Grundlage: Durch Ableitung einer Event-Stub-Proxy-

Klasse[31] kann Netzwerk-Kommunikation realisiert werden.

4.7.4. Ereignis-Parameter

Wie wir im vorigen Kapitel bereits gesehen haben, ist die Verwendung von Ereignis-Parametern noch nicht ausreichend erforscht, um sagen zu können, ob sie wirklich nützlich sein können.

Um eine geeignete Grundlage für weitere Untersuchungen hinsichtlich der Verwendung von Ereignis-Parameter zu ermöglichen, sollte das Muster Ereignis-Parameter anbieten, ihre Verwendung aber nicht erzwingen.

Das Observer-Muster bietet keine typsichere Möglichkeit, Ereignis-Parameter zu realisieren. Das Command-Muster ließe sich entsprechend erweitern. Das Event-Notification-

Muster bietet Ereignis-Parameter, wobei die Verwendung der Ereignis-Parameter jedoch nicht erzwungen wird.

---

[31] Näheres zum Proxy-Muster findet sich in [GHJ+94].

4.7.5. Event-Manager für die globale Ereignisfluß-Steuerung

Wir haben im vorigen Kapitel unter anderem auch das Problem der Ereignis-Wettläufe untersucht und dabei gezeigt, daß ein Event-Manager zur Ereignisfluß-Kontrolle notwendig ist.

Das neue Muster sollte einen Event-Manager schon in der Konzeption enthalten.

Es ist ohne weiteres möglich, alle drei behandelten Muster um einen Event- bzw. Change-Manager zu erweitern, auch wenn beim Command-Muster nicht explizit darauf hingewiesen wird. Beim Observer- und Event-Notification-Muster wird der Change-Manager allerdings als Option dargestellt. Wir haben jedoch gesehen, daß seine Existenz in komplexen System *notwendig* ist.

4.7.6. Dynamische Überprüfung von reaktiven Änderungen

Im vorigen Kapitel haben wir das Problem der reaktiven Änderungen beschrieben und gesehen, daß es keine praktikable Möglichkeit gibt, diese präventiv zu verhindern. Als einzige Möglichkeit haben wir eine dynamische Überprüfung der reaktiven Änderungen herausgearbeitet, welche eine Ausnahmebehandlung bei Auftreten einer reaktiven Änderung auslöst.

Das neue Muster sollte die Möglichkeit bieten, reaktive Änderungen zu überprüfen und ggfs. eine Ausnahmebehandlung auszulösen.

In alle drei behandelten Muster kann eine solche Möglichkeit implementiert werden, auch wenn diese in keiner der Musterbeschriebungen erwähnt wurde. Wie der Ereignis-Mechanismus sollte auch die Möglichkeit der Überprüfung als existentieller Teil eines Reaktionsmechanismusses angesehen werden.

4.7.7. Unterstützung der State-Chart-Sichtweise

Wir haben oben gesehen, daß State-Charts eine Möglichkeit bieten, abstrakte Zustände als Grundlage für Ereignisse zu modellieren. Es sollte daher möglich sein, eine entsprechende Unterstützung in das Muster zu integrieren. Es könnte z.B. hilfreich sein, den jeweiligen Vorzustand zur Verfügung zu stellen.

Es sollte die Möglichkeit bestehen, spezielle Protokolle für die Unterstützung von State-Chart-Modellierungen zu implementieren.

Das Observer-Muster läßt sich zwar prinzipiell mit State-Chart-Modellierung vereinbaren, schöpft aber aufgrund der nicht-unterscheidbaren Ereignissen die Möglichkeiten nicht voll aus. Beim Command- und beim Event-Notification-Muster ist es aufgrund der fehlenden Oberklassen nur schwer möglich, eine entsprechende Unterstützung zu integrieren, da diese in den Command- bzw. Event-Klassen implementiert werden müßte.

4.7.8. Kommando-Muster für Reaktion auf Benutzeraktion

Wir haben in Abschnitt 4.5. gezeigt, daß das Command-Muster sich für die Reaktion auf Benutzeraktion am besten eignet. Dieses sollte um Parameter erweitert werden.

Es sollte mit dem Command-Muster auf Benutzeraktionen reagiert werden.

4.7.9. Portierbarkeit des Konzeptes

Das Muster sollte nach Möglichkeit mit den Mitteln der (objektorientierten) Programmiersprache realisierbar sein, ohne weitere Mittel wie Makros, Preprozessoren oder Codegeneratoren einsetzen zu müssen. In den letzteren Fällen, wäre das Muster schwer zu durchschauen und zu warten. Außerdem wären Erweiterungen und Modifikationen nur schwer durchführbar. Außerdem sollte das Muster so konzipiert sein, daß es sich problemlos in verschiedenen Programmiersprachen realisieren läßt.

Das Muster sollte mit den Mitteln der (objektorientierten) Programmiersprache realisierbar sein und außerdem unabhängig von der Programmiersprache sein.

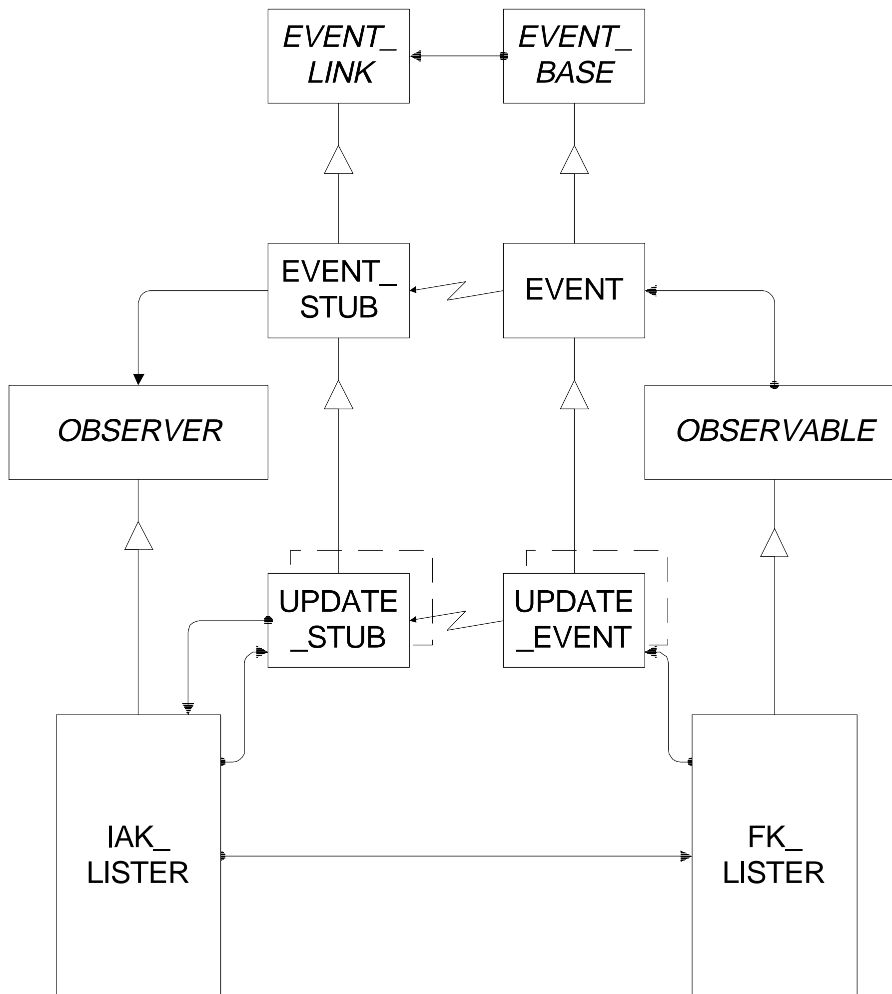
Das Observer- und das Command-Muster sind in den verbreiteten objektorientierten Programmiersprachen Eiffel, Smalltalk und C++ realisierbar. Das Event-Notification-Muster ist in der vorgestellten Form nicht in Eiffel implementierbar.

4.7.10. Bewertung der vorgestellten Reaktionsmuster

Die Einordnung der drei vorgestellten Muster in die oben aufgestellten Kategorien hat gezeigt, daß sie sich in wesentlichen Punkten unterscheiden, daß aber (fast) alle wünschenswerten Eigenschaften in der Summe der Eigenschaften der Muster enthalten sind. Es gibt kein Reaktionsmuster, welches unseren Anforderungen vollkommen gerecht wird. Es bleibt aber die Hoffnung bestehen, daß sich aus den oben aufgeführten Mustern ein für unsere Zwecke besser geeignetes Muster “synthetisieren” läßt. Die Konzeption und Realisierung dieses Musters wird im nächsten Kapitel Gegenstand der Untersuchung sein.

5. Konzeption eines neuen Musters: EVENT-OBSERVER

In diesem Kapitel wird die Konzeption für den neuen Reaktions-Mechanismus vorgestellt. Wir haben hier die Bezeichnung Event-Observer-Pattern (Ereignis-Beobachter-Muster) gewählt, weil der Name verdeutlicht, daß es sich bei diesem Muster um eine Kombination herkömmlicher Observer-Muster mit den neueren Ereignis-Mustern handelt. Das Command-Muster kommt hier in der Namensgebung nicht vor, da dieses in dem neuen Muster nicht mehr erkennbar ist. Wie wir oben gezeigt haben, halten wir eine Trennung von Benachrichtigungs- und Kommando-Muster für sinnvoll.
Wir beschreiben nachfolgend zunächst die beiden Klassen OBSERVER und OBSERVABLE, welche als allgemeine Oberklassen für unseren Mechanismus dienen, ähnlich wie dies beim Observer-Muster der Fall ist. Allerdings ist die eigentliche Funktionalität nicht in diesen Klassen enthalten, sondern in den einzelnen Event-Klassen (ähnlich wie beim Event-Notification-Muster), welche im darauffolgenden Abschnitt erläutert werden.



Klassendiagramm: EVENT-OBSERVER

5.1. Die Klassen OBSERVER und OBSERVABLE

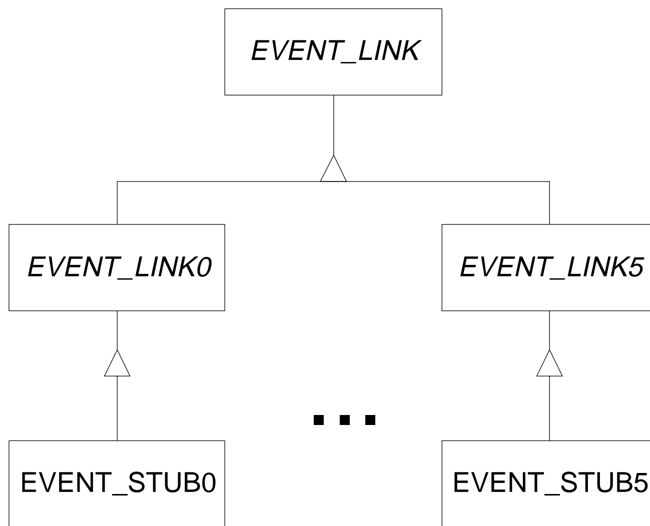
Wir definieren die beiden aufgeschobenen Oberklassen OBSERVER und OBSERVABLE, um auch auf der Ebene der Klassenhierarchie deutlich zu machen, welche Klassen Beobachter (Abhängige) und welche Klassen Beobachtete (Subjekte) sind. In einigen Fällen kann es sogar sinnvoll sein, in diesen Klassen Funktionalität zu implementieren oder Routinen zu spezifizieren. Wir haben in der Klasse OBSERVABLE z.B. Funktionen implementiert, welche den jeweils vorigen Event zur Verfügung stellen.

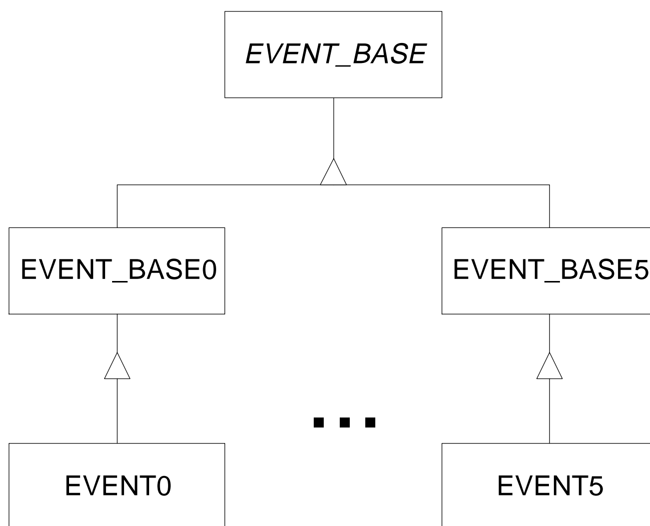
5.2. Die Event-Klassen

Wir deklarieren Events an der Klassenschnittstelle der Subjektklasse als über Funktionen abfragbare Instanzvariablen. Auf der anderen Seite deklasieren wir Event-Stubs je Event an der Klassenschnittstelle der beobachtenden Klasse. Die Event-Stubs werden vom abhängigen Objekt bei den Events des Subjektes angemeldet. Außerdem meldet sich das abhängige Objekt bei seinen Event-Stubs mit jeweils einer Routine an, welche bei Benachrichtigung des Event-Stubs gerufen werden soll. Löst jetzt das Subjekt einen Event aus, benachrichtigt dieser alle angemeldeten Event-Stubs, welche abstrakt als EVENT_LINKs bekannt sind. Durch Ableitung weiterer Klassen von EVENT_LINK können weitere Sendemethoden implementiert werden, welche z.B. asynchrone Kommunikation in Netzwerken erlauben könnten[32]. Die Events sind für die abhängigen Klassen nur als EVENT_BASEs bekannt, damit nur die Subjektklasse die Prozedur `announce` aufrufen kann.

---

[32]Es könnte dann ein Event-Stub-Proxy von Event-Link abgeleitet werden. Näheres zum Proxy-Muster findet sich in [GHJ+94].



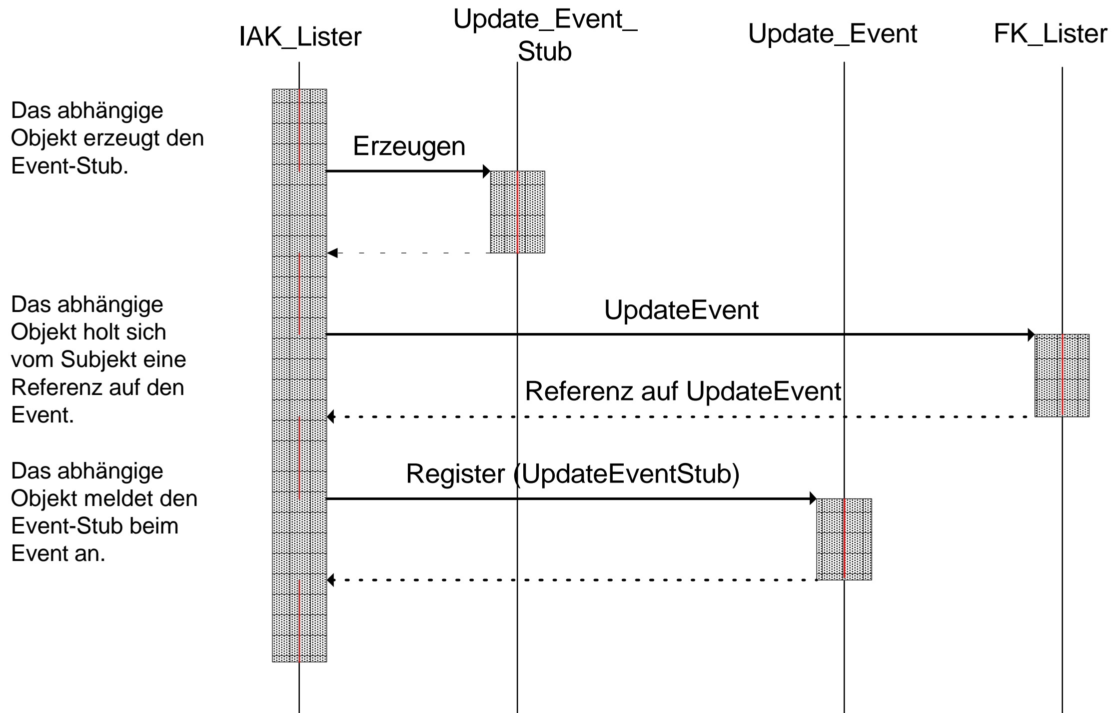


Klassendiagramm: EVENT-OBSERVER

Wir haben die Möglichkeit vorgesehen, bis zu fünf Parameter bei der Aktivierung eines Events mitzugeben, wie in dem obigen Klassendiagramm zu erkennen ist. In einem der vorherigen Kapitel hatten wir bereits kurz die Problematik der Event-Parameter angedeutet. Wir haben diese Möglichkeit geschaffen, damit Erfahrungen in diesem Gebiet gesammelt werden können. Möchte man keine Event-Parameter nutzen, so kann man einfach auf die Events ohne Parameter (tEvent0) zurückgreifen.

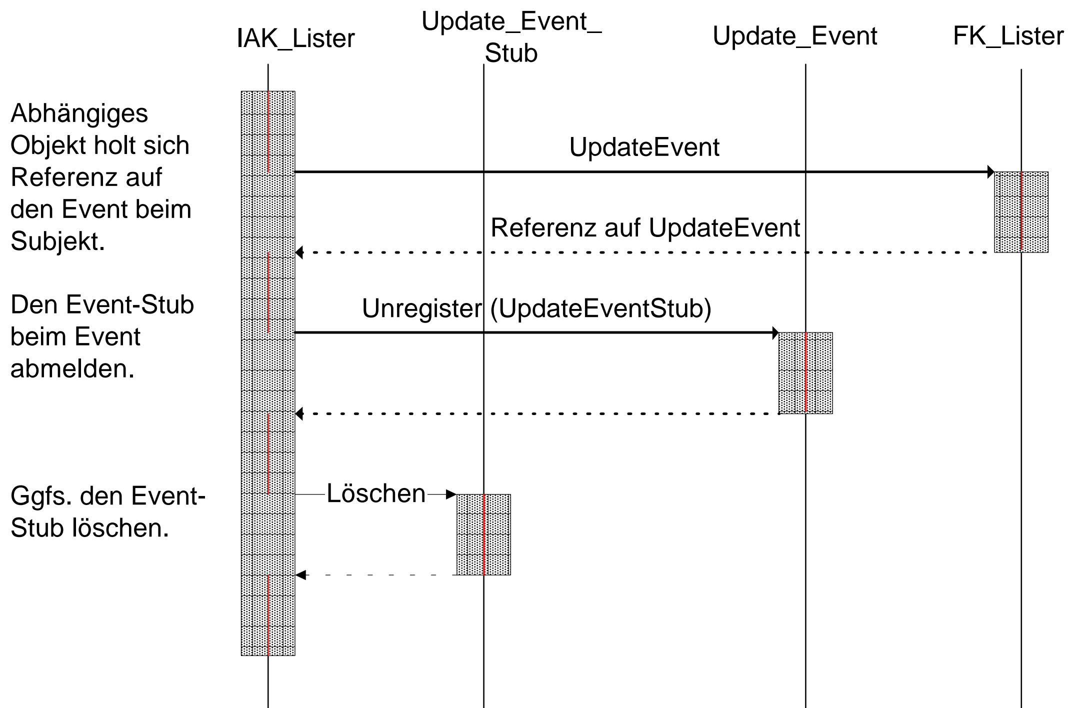
5.3. Die Dynamik des Musters

Wir stellen hier jeweils ein Interaktionsdiagramm für den An- und Abmeldevorgang sowie die Benachrichtigung vor.



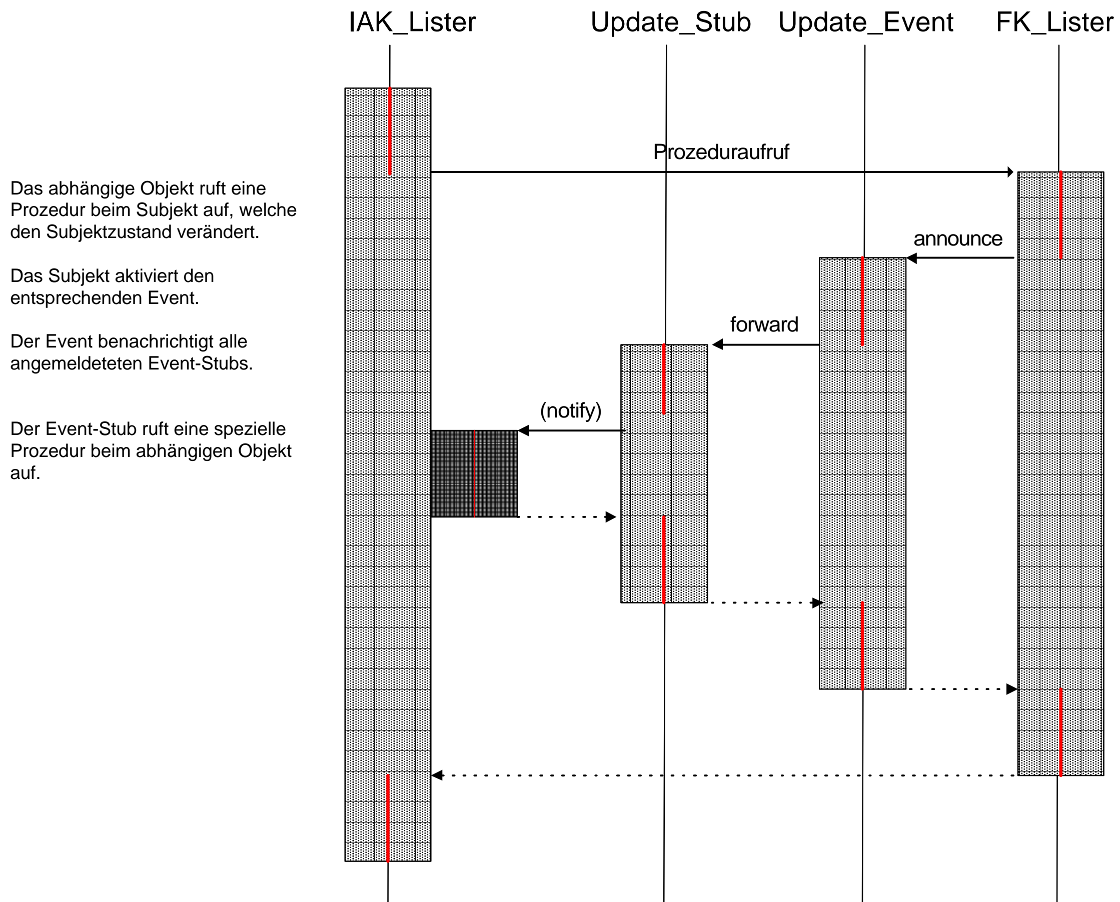
Interaktionsdiagramm: Anmeldevorgang EVENT-OBSERVER

Die Interaktionskomponente erzeugt je Event, für welchen sie sich interessiert einen Event-Stub. Diesem übergibt sie eine ihrer Prozeduren als Parameter. Diese Prozedur soll aufgerufen werden, wenn der Event-Stub aktiviert wird.
Danach holt sich die Interaktionskomponente bei der Funktionskomponente eine Referenz auf den entsprechenden Event (Update_Event) und meldet einen passenden Event-Stub (Update_Stub) bei diesem Event an.
Der Event hat eine Liste der zu benachrichtigen Event-Stubs. Der Event, der Event-Stub und die aufzurufende Prozedur der Interaktionskomponente müssen die gleichen Parameter haben.



Interaktionsdiagramm: Abmeldevorgang EVENT-OBSERVER

Die Interaktionskomponente holt sich bei der Funktionskomponente eine Referenz auf den Event und meldet den entsprechenden Event-Stub beim Event ab. Der Event entfernt daraufhin diesen Event-Stub aus der Liste der zu benachrichtigen Event-Stubs. Danach kann der Event-Stub von der Interaktionskomponente gelöscht werden.



Interaktionsdiagramm: Benachrichtigung EVENT-OBSERVER

Bei einer Zustandsänderung in der Funktionskomponente stößt diese die Event-Benachrichtigung bei dem entsprechenden Event an (`announce`). Der Event benachrichtigt daraufhin alle bei ihm angemeldeten Event-Stubs. Der Event-Stub ruft dann die bei ihm angemeldete Prozedur bei der Interaktionskomponente auf.
Die ganze Kette entlang können die geänderten Werte mit übergeben werden, damit die Interaktionskomponente nicht mehr bei der Funktionskomponente sondieren muß.

5.4. IAT-Events

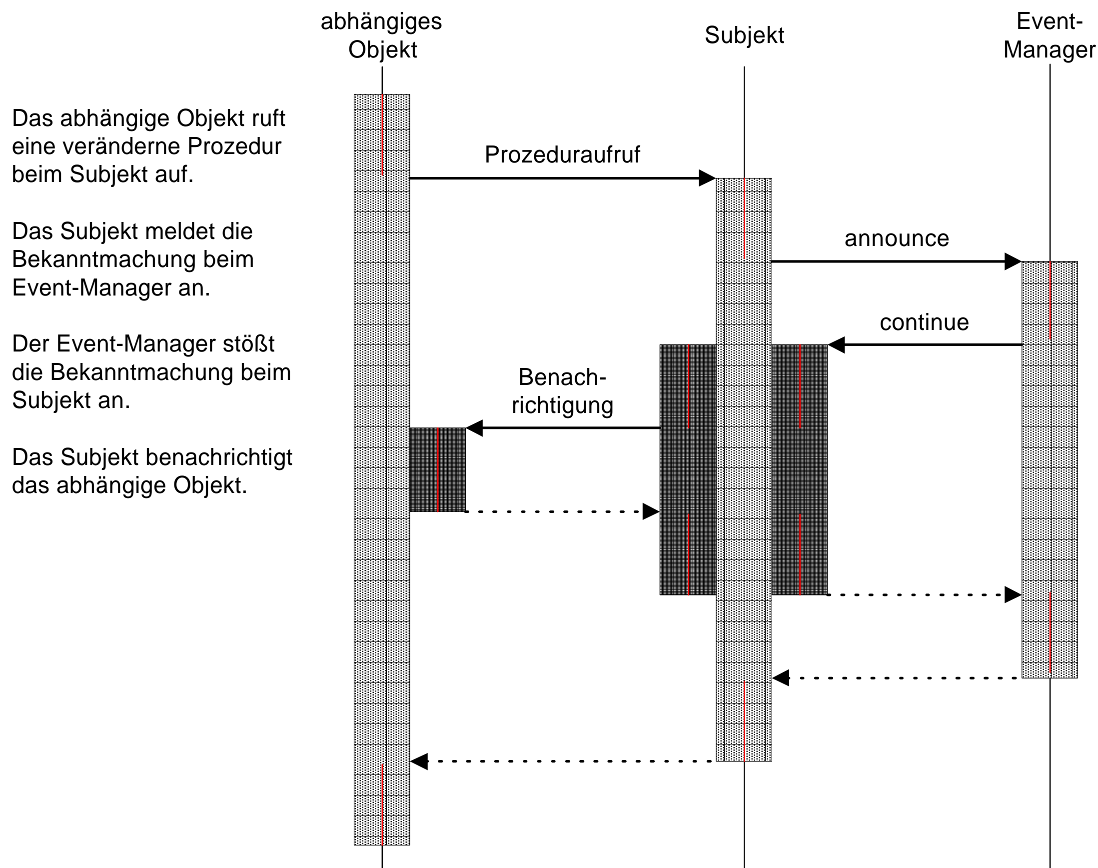
Obwohl wir uns weiter oben dafür ausgesprochen haben, auf Benutzeraktionen durch den Einsatz des COMMAND-Musters zu reagieren, wollen wir hier kurz die Möglichkeiten andeuten, welche unser EVENT-OBSERVER-Muster hier bietet.
Wir haben bereits beschrieben, welche Besonderheiten bei der Reaktion auf Benutzereingaben zu beachten sind.
Es scheint daher sinnvoll, von den Event- und den Event-Stub-Klassen spezielle IAT-Event- und IAT-Event-Stub-Klassen abzuleiten, welche die folgende Besonderheit aufweisen: Es kann nur ein abhängiges Objekt bei einem Subjekt (hier Oberflächenobjekt) angemeldet werden.

5.5. Der Event-Manager

Wie wir oben beschrieben haben, kann das Auftreten von Ereignis-Wettläufen durch den Einsatz eines Change-Managers verhindert werden. Wir wollen uns hier etwas näher mit dem Change-Manager beschäftigen, welchen wir im folgenden *Event-Manager* nennen. Schließlich unterscheiden wir in unserem Muster differenzierte Ereignisse.

Der Event-Manager hat die Aufgabe, die bei ihm eintreffenden Ereignisse zu serialisieren, so daß sie in der chronologischen Reihenfolge beim abhängigen Objekt ankommen, in welcher sie ausgelöst wurden.

Wir haben den Event-Manager als Singleton (siehe [GHJ+94]) realisiert, damit er nur einmal im gesamten System vorhanden ist. Alle Events haben Zugriff auf den Event-Manager. Wird ein Event ausgelöst, meldet dieser seine Aktivierung nicht direkt an die angemeldeteten Event-Stubs, sondern an den Event-Manager. Wenn alle vorher beim Event-Manager eingetroffenen Ereignisse abgearbeitet sind, aktiviert der Event-Manager erneut den Event, welcher jetzt die angemeldeteten Event-Stubs benachrichtigt. Das folgende Interaktionsdiagramm demonstriert die neue Objektinteraktion unter Einbeziehung des Event-Managers.



Objekt-Interaktion unter Einbeziehung des Event-Managers

Es wäre ebensogut auch möglich gewesen, die Verwaltung der angemeldeten Event-Stubs vom Event-Manager und nicht den Events vornehmen zu lassen. Für diese Lösung spricht vor allem die Tatsache, daß durch eine Verwaltung bei den Events potentiell mehr Speicherplatz verbraucht wird, als bei einer Verwaltung beim Event-Manager. Der Event-Manager stellt nur genau dann Speicherplatz zur Verfügung, wenn dies auch notwendig ist. Ist bei einem Event kein Event-Stub angemeldet, so wird hier auch kein Speicherplatz

verbraucht. Siedelt man hingegen die Verwaltung bei den einzelnen Events an, so enthalten diese immer zumindest eine leere Liste.
Wir haben die Verwaltung dennoch bei den einzelnen Events belassen, weil der Speicherplatzverlust eher gering ausfällt und diese außerdem leichter zu implementieren und warten ist.
Hat man es mit einem speicherkritischen System zu tun, sollte die Verwaltung der angemeldeteten Event-Stubs aber auf jeden Fall beim Event-Manager geschehen..

Daß die Benutzung des Event-Managers tatsächlich die Ereignisse serialisiert, zeigt das folgende Interaktionsdiagramm, welches den weiter oben beschriebenen Ereignis-Wettlauf unter Einbeziehung des Event-Managers darstellt[33].

[33]Das Interaktionsdiagramm ist recht komplex. Es mag hilfreich sein, sich das weiter oben angegebene Interaktionsdiagramm zum Ereignis-Wettlauf nochmal anzusehen.

Abhängiger A

Abhängiger B

Subjekt C

Event-Manager

Prozeduraufruf

Der Subjektzustand wird durch einen Prozeduraufruf verändert.

announce

Das Subjekt meldet sich beim Event-Manager an.

continue

Der Event-Manager löst die Bekanntmachung beim Subjekt aus.

(notify)

Das Subjekt benachrichtigt Objekt B, welches darufhin seinen Zustand ändert.

announce

Objekt B meldet sich beim Event-Manager an.

(notify)

Das Subjekt benachrichtigt Objekt A.

continue

Der Event-Manager löst die Bekanntmachung bei Objekt B aus.

(notify)

Objekt B benachrichtigt Objekt A.

Ereignis-Wettlauf mit Event-Manager

Das folgende Code-Beispiel[34] stellt die Hauptprozedur des `announce` Event-Managers dar. Diese wird vom Event gerufen, wenn dieser sich bekanntmachen soll. Sobald der Event an der Reihe ist, ruft der Event-Manager das bei ihm implementierte `continue`, welches für die eigentliche Verschickung an alle angemeldeten Event-Stubs zuständig ist[35].

[34] Es handelt sich hierbei um Eiffel. Die C++-Implementation findet sich in Anhang A.

[35] In dem Codebeispiel verwenden wir die LINKED_QUEUE-Klasse aus der Eiffel-Base-Bibliothek.

```
class EVENT_MANAGER
     -- der Event-Manager

feature

     is_announcing : BOOLEAN
          -- wird gerade ein Event im System bekanntgemacht?

     announce (event : EVENT) is
          -- mache 'event' bekannt
          -- ggfs. ist die Bekanntmachung zu verzögern
     require
          not_void: event /= Void
     do
          if is_announcing then    -- es wird gerade ein
                                   -- Ereignis
                                   -- bekanntgemacht
               queue.extend(event)
          else -- keine Ereignisse in der Queue vorhanden
               queue.extend(event)
               from
               until
                    queue.empty
               loop
                    is_annoucing := True
                    queue.item.continue
                         -- ältestes Ereignis bekanntmachen
                    is_announcing := False
                    queue.remove
                         -- zuletzt bekanntgemachtes
                         -- Ereignis entfernen
               end
          end
     end

...

end -- class EVENT_MANAGER
```

5.6. Dynamische Überprüfung reaktiver Änderungen

In diesem Abschnitt wollen wir darstellen, wie wir den Mechanismus zur Überprüfung reaktiver Änderungen konzipiert haben.

Wie wir oben gesehen haben, können wir davon ausgehen, daß eine reaktive Änderung dann vorliegt, wenn ein Ereignis beim Subjekt ausgelöst wird, während sich das Subjekt bereits im Zustand des Signalisierens befindet.

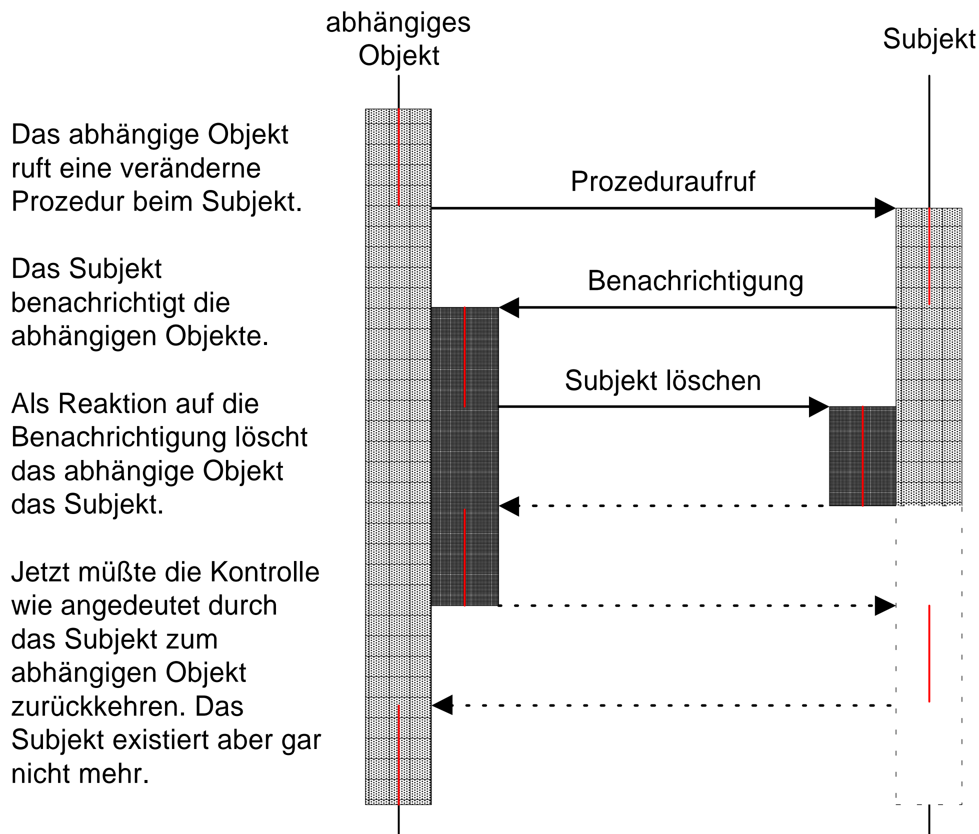
Dies haben wir so realisiert, daß das sich Subjekt merkt, ob es sich gerade im Zustands des Signalisierens befindet. Wird ein Event ausgelöst, so befragt er zunächst das Subjekt, ob es sich bereits im Zustand des Signalisierens befindet. Ist dies der Fall, so löst der Event eine entsprechende Ausnahmebehandlung aus, welche in der Regel zu einem Programmabbruch oder einer Protokollierung des Fehlverhaltens führt. Befindet sich das Subjekt nicht im

Zustand des Signalisierens, so versetzt der Event das Subjekt in diesen und benachrichtigt die angemeldeten Event-Stubs. Danach setzt er das Subjekt wieder zurück.
Wenn die Zielsprache Vorbedingungen unterstützt, kann diese Überprüfung aus dem eigentlichen Programmcode in die Vorbedingung herausgezogen werden. Dadurch wird die Forderung nach Vermeidung reaktiver Änderungen explizit gemacht.

5.7. Die Implementierung

In diesem Abschnitt wollen wir die implementationsspezifischen Aspekte unseres Musters beschreiben. Wir haben die Implementation in C++ vorgenommen und das Werkzeug&Material-Rahmenwerk des Arbeitsbereiches Softwaretechnik der Universität Hamburg auf unsere Implementation umgestellt.
In Programmiersprachen, welche Prozeduren nicht als Erste-Klasse-Objekt behandeln, müssen die Event-Stub-Klassen jeweils passend zur abhängigen Klasse implementiert werden. Dies bedeutet, daß für jedes Ereignis, auf welches reagiert werden soll, eine Klasse zu implementieren ist. Dies mag auf viele abschreckend wirken, aber es entspricht auf jeden Fall eher dem Gedanken der Objektorientierung als ein langes CASE-Konstrukt, welches das Dispatching vornimmt.
In C++ besteht die Möglichkeit, klassengebundene Prozeduren unter Zuhilfenahme von Templates, als Parameter zu übergeben. Genau dies haben wir in unserer Implementierung gatan. So muß nicht für jedes Ereignis eine Klasse implementiert werden. Im Konstruktor des Event-Stubs wird einfach die aufzurufende Prozedur übergeben.
Weiterhin war die Frage zu klären, was passieren soll, wenn eine An- oder Abmeldung während der Bekanntmachung eines Events initiiert wird. Da eine sofortige Manipulation der Liste mit den angemeldeten Event-Stubs zu nicht vorhersehbaren Effekten geführt hätte, wird die An- bzw. Abmeldung nicht sofort in der Liste vorgenommen. Vielmehr werden zwei weitere Listen geführt. In der ersten Liste sind alle diejenigen Event-Stubs vermerkt, welche vor der nächsten Bekanntmachung noch in die Stub-Liste einzutragen sind. Die zweite Liste enthält diejenigen Event-Stubs, welche vor der nächsten Bekanntmachung noch aus der Stub-Liste entfernt werden müssen.

Ein Problem bei Reaktionsmechanismen besteht in der Zerstörung von Subjekten. Wird ein Subjekt als Reaktion auf eine Benachrichtigung zerstört, so muß sich der Kontrollfluß durch das eben gelöschte Objekt zurückziehen. Wir sprechen in diesem Fall auch vom reaktiven Löschen. Das folgende Interaktionsdiagramm demonstriert diese Problematik:



Reaktives Löschen

Zunächst ist hier zu bemerken, daß dieses Problem auch in Systemen mit Garbage-Collector auftreten kann. In diesem Fall wird das Subjekt nicht explizit vom abhängigen Objekt gelöscht. Es wird lediglich die Referenz auf das Subjekt gelöst. Wird das Subjekt jetzt von keinem weiteren Objekt referenziert kann es gelöscht werden. Auch wenn der Garbage-Collector meistens nicht sofort aktiv wird, gibt es keine Garantie dafür, daß das Subjekt noch lange genug existiert, damit sich die Kontrolle noch zurückziehen kann.
Die Lösung des Problemes ist stark von der Programmiersprache abhängig. In C++ besteht die Lösung darin, nach dem Löschen des Subjektes keine neuen Objekte zu erzeugen, bis die Kontrolle durch das Subjekt zum abhängigen Objekt zurückgekehrt ist. Auch wenn das Subjekt eigentlich gar nicht mehr existiert, so führt diese Technik dennoch nicht zu einem Fehler, weil das Subjekt noch als “Leiche” im Speicher zur Verfügung steht.
Ist in der Programmiersprache ein Garbage-Collector vorhanden, kann diese Technik mitunter nicht angewendet werden, da nicht vorhersehbar ist, wann der Garbage-Collector das Subjekt löscht. In den verschiedenen Garbage-Collector-Implementationen wird dieser meist erst aktiv, wenn Speicherplatz für ein neues Objekt angefordert wird. Kann man sich auf ein solches Verhalten des Garbage-Collectors nicht verlassen, so hat man in der Regel auch die Möglichkeit, den Garbage-Collector zeitweise zu deaktivieren. Dies kann man dann für die Zeit des Kontrollrückflusses tun.

Hat man es mit einem Garbage-Collector zu tun, welcher sich nicht deaktivieren läßt, so muß man bereits auf der konzeptionellen Ebene geeignete Maßnahmen zu treffen. Denkbar wäre hier beispielsweise eine Lösung, welche des Lösen der Subjektreferenz solange aufschiebt, bis die Kontrolle zum abhängigen Objekt zurückgekehrt ist. Aus Platzgründen können wir jedoch an dieser Stelle nicht näher auf diese Möglichkeit eingehen[36].

Es wäre in einigen Fällen hilfreich, wenn zuerst das auslösende abhängige Objekt Rückkopplung via Ereignis bekommen würde und erst dann die anderen abhängigen Objekte (siehe [P+88]). Dies ist insbesondere dann wichtig, wenn die Benachrichtigung aller abhängigen Objekte eine relevante Zeitspanne in Anspruch nimmt und somit der Benutzer eine merkliche Zeit auf die Aktualisierung seiner Anzeige warten muß. Dieses Problem tritt vor allem dann auf, wenn mehrere Benutzer - verbunden über ein Netzwerk - das selbe Material im Zugriff haben. Dann sollte ebenfalls die Veränderung zuerst bei dem Benutzer sichtbar werden, welcher diese auch ausgelöst hat. Hier ergibt sich allerdings das technische Problem, daß das Subjekt gar nicht weiß, von welchem Objekt eine bestimmte ihrer Prozeduren aufgerufen wurde. Dies ist so ohne weiteres nur mit entsprechender Laufzeitunterstützung der verwendeten Programmiersprache möglich. Als Lösung ist hier evt. eine Lösung mit Prioritäten zu wählen. In einer solchen Implementation könnte man zum einen Ereignissen eine Priorität zuweisen, so daß Ereignisse mit einer hohen Priorität bevorzugt abgehandelt werden, was wiederum in den Aufgabenbereich des EVENT_MANAGER fallen würde. Zum anderen könnten sich abhängige Objekte mit einer Priorität bei den Ereignissen anmelden. So könnte ein abhängiges Objekt, welches eine Veränderung an einem Subjekt vornehmen will, vorher seine Priorität bei den Events des Subjektes erhöhen, damit es zuerst benachricht wird. Als Reaktion auf die Benachrichtigung könnte er die Priorität wieder zurücksetzen. Das Zurücksetzen könnte auch automatisch durch die Events bzw. den Event-Manager geschehen. Es wäre aber in jedem Fall ein erhöhter Programmieraufwand notwendig, da das abhängige Objekt ständig die Priorität erhöhen müßte. Wir haben aus diesen Gründen diese Variante bisher nicht implementiert.

Die eigentliche Event-Klasse ist in zwei Klassen aufgeteilt. In der Klasse EVENT_BASE sind die beiden Prozeduren `register` und `unregister` zum An- und Abmelden von Beobachtern implementiert. Die Klasse EVENT erbt von EVENT_BASE und implementiert die Prozedur `announce` zum Verschicken von Events. Durch diese Aufteilung ist es möglich, für die abhängigen Objekte nur die Prozeduren der Klasse EVENT_BASE zugänglich zu machen, während nur das Subjekt die Prozedur `announce` aufrufen kann. Die Event-Stubs der abhängigen Objekte sind für Klienten der Klasse versteckt (hidden), so daß die `forward`-Prozedur nicht aufgerufen werden kann. Das abhängige Objekt übergibt bei der Anmeldung des Event-Stubs beim Event diesen, so daß der Event Zugriff auf die `forward`-Prozedur hat.
Laut [GHJ+94] können redundante Updates durch geschickte Implementierung des Benachrichtigungsmechnismusses umgangen werden. Dies trifft bei Ereignis-Mechanismen leider nicht zu. In dem in [GHJ+94] beschriebenen Observer-Muster ist dies möglich, da der Change-Manager bei weiteren ankommenden Update-Ereignissen diese nur einmal an die abhängigen Objekte zustellen kann. Bei unserer Sichtweise ist eine solche Wegoptimierung von Ereignissen unzulässig, da damit auch vollzogene Zustandsübergänge wegopti-

[36]Unseres Wissens nach hat Wolf-Gideon Bleeck eine solche Möglichkeit des *späten Zerstörens* für C++ implementiert.

miert würden. Dies ist vor allem mit den formalen Grundlagen des Musters nicht zu vereinbaren.

Unser Event-Observer-Muster übertrifft alle vorgestellten Muster an Komplexität. Daher ist für die sinnvolle Benutzung der Implementierung ein grundlegendes Verständnis notwendig.

5.8. Ein modifiziertes COMMAND-Muster

Wir haben neben dem EVENT-OBSERVER-Muster außerdem das Command-Muster modifiziert und implementiert. Das modifizierte Command-Muster haben wir wie oben gefordert für die Reaktion auf Benutzeraktionen verwendet.
Die Modifikationen betreffen im wesentlichen die Möglichkeit, die Kommando-Objekte zu parametrisieren. In der modifizierten Variante ist es jetzt möglich, die Kommando-Objekt mit den geänderten Werten zu parametrisieren. Dazu erben von einer allgemeinen aufgeschobenen COMMAND-Klasse fünf weitere: COMMAND0 bis COMMAND5. Die Ziffer am Ende des Klassennamens bezeichnet die Anzahl der Parameter. Die Klassenschnittstellen sehen wie folgt aus:

```
deferred class COMMAND

     execute is deferred
     end

     undo is
     do
          -- leere Standardimplementation
     end

     redo is
     do
          execute  -- Standardimplementation
     end

end

deferred class COMMAND0
inherit COMMAND
end

deferred class COMMAND1 [ARG_TYPE1]
inherit COMMAND

     execute (arg1 : ARG_TYPE1) is deferred
     end

end

...
```

Die abhängige Klasse muß die jeweiligen Commando-Klassen implementieren und beim Oberflächen-Subjekt anmelden. In C++ können die Command-Klassen so implementiert werden, daß im Konstruktor Prozedurreferenzen übergeben werden, ähnlich wie wir dies auch schon bei den Event-Stub-Klassen realisiert haben. Diese Prozeduren werden dann beim Aufruf der einzelnen Prozeduren der Command-Klassen aufgerufen. Somit kann man auf das Implementieren der Command-Klassen verzichten. In dieser Variante ist es aber mitunter schwerer eine entsprechende Undo-Funktionalität zu realisieren, weil diese ebenfalls in der abhängigen Klasse implementiert werden müßte.

Command-Klassen bieten eine gute Ausgangsbasis für die Realisierung von Undo-/Redo-Funktionalität. Insbesondere bei komplexen reaktiven Systemen, reicht diese Möglichkeit jedoch mitunter nicht aus. Dies gilt auch für komplexe Werkzeuge nach der Werkzeug&Material-Metapher. Führt eine Benutzeraktion in einem Kontextwerkzeug zu einer Änderung einer Sub-Funktionskomponente, so reichen die in dem zugehörigen Command-Objekt gespeicherten Werte mitunter nicht aus, um den Zustand der Sub-Funktionskomponente wieder herzustellen. An dieser Stelle ist mit Sicherheit noch Klärungsbedarf vorhanden.

Wir haben beim Event-Notification-Muster gesehen, daß es in der vorgestellten Form nicht in allen objektorientierten Programmiersprachen realisierbar ist. Da unser Muster stark von dem Event-Notification-Muster beeinflußt wurde, gilt dies auch hier. Wenn man allerdings auf die Ereignis-Parameter verzichtet, ist das Muster in anderen Programmiersprachen wie z.B. Eiffel implementierbar. Dann müßte man die Event-Stub-Klassen jeweils implementieren, da Prozeduren nicht als Parameter übergeben werden können. Ob die dadurch entstehende Menge von Event-Stub-Klassen auch in großen Systemen noch handhabbar ist, muß die Zukunft zeigen.

6. Die Umstellung der Bibliothek

In diesem Kapitel wollen wir die Umstellung der Bibliothek beschreiben und die Erfahrungen, welche wir dabei gewonnen haben.

6.1. Umstellung der Bibliotheksklassen

Wir haben die Bibliothek des Arbeitsbereiches Softwaretechnik der Universität Hamburg auf die Reaktionsmuster *Event-Observer* und *Command* umgestellt.
Die Reaktion auf Objektänderungen geschieht mit dem Event-Observer-Muster. Wir haben dabei auf die Anwendung von Ereignis-Parameter verzichtet. Die Reaktion auf Benutzeraktion wird mit einem modifizierten Command-Muster realisiert. Dabei haben wir Command-Parameter benutzt, um die geänderten Werte an das abhängige Objekt mitzugeben.

6.2. Umstellung vorhandener Anwendungen

Die Umstellung vorhandener Anwendungen ist im wesentlichen unproblematisch, weil hier keine Besonderheiten wie die in Abschnitt 6.1. zu beachten sind. Die Umstellung ist hier lediglich eine “Fleißarbeit”. Im Anhang B stellen wir ein Migrationsmuster vor, mit welchem die Umstellung vorhandener Anwendungen erleichtert werden soll.

6.3. Migrationsmuster

Wir haben im Anhang B ein Migrationsmuster erstellt, welches bei der Umstellung von Anwendungen helfen soll, welche mit dem Notifier-Mechanismus der BibV21 konstruiert wurden. Wir haben diese Umstellungsanleitung Muster genannt, weil sie immerhin ein Muster vorgibt, nach welchem vorgegangen werden kann. Dieses Muster ist aus unserer Erfahrung mit der Umstellung der Bibliothek BibV21 sowie einer darauf basierenden Beispielanwendung (der Aufgabenverwalter) entstanden. Wir hoffen, daß es den Benutzern behilflich ist, Umstellungsarbeiten schnell und störungsfrei durchführen zu können.

7. Diskussion und Abschluß

Wir haben in unserer Arbeit verschiedene Reaktionsmuster vorgestellt, analysiert und bewertet. Auf Grundlage dieser Analyse haben wir ein neues Muster, das Event-Observer-Muster, aus den vorgestellten Mustern "synthetisiert".
Wir haben weiterhin die konzeptionellen und formalen Randbedingungen von Reaktionsmechanismen untersucht und die dabei gewonnenen Ergebnisse in die Konzeption und Realisierung des Event-Observer-Musters einfließen lassen.

Wir hoffen, durch unsere Arbeit einen Beitrag zur Klärung der Konzepte von Reaktionsmechanismen geliefert zu haben. Wenn die Konzepte ausreichend klar ausgearbeitet und Konsens über sie hergestellt wurde, ist zu überlegen, ob diese als Mittel der Programmiersprache selbst bereitgestellt werden sollten.

Im Laufe der Arbeit haben wir einige Themenbereiche aufgezeigt, an denen offenbar noch eine Menge weiterer Arbeit zu leisten ist, Arbeit, welche wir in diesem Rahmen nicht leisten konnten. Die wichtigsten offenen Fragen betreffen die folgenden Punkte:

Konzeptionelle Fragen

- Das Command-Muster bietet eine gute Grundlage für die Realisierung von Undo-/Redo-Funktionalität. Welche zusätzlichen Mittel für komplexe Werkzeuge nach der Werkzeug&Material-Metapher notwendig und sinnvoll sind, ist noch offen.
- Ereignis-Parameter erscheinen attraktiv, ihre Verwendbarkeit muß allerdings noch erprobt werden.
- Ob und wie die hier vorgestellten Reaktionsmechanismen in verteilten Softwaresystemen einsetzbar sind, ist weiterhin eine offene Frage, die wir im Rahmen unserer Arbeit nicht untersuchen konnten.
- Wir haben in einem Abschnitt die Möglichkeiten zum Propagieren von Benachrichtigungen diskutiert. Insbesondere im Bereich der Ereignis-Mechanismen sind hier jedoch noch Schwierigkeiten vorhanden, welche gelöst werden müssen.

Verwendung von State-Charts für die Ereignis-Modellierung

- Die Modellierung von Ereignissen auf Grundlage von State-Charts wurde nur kurz und keinesfalls umfassend dargestellt. Hier fehlt nach wie vor eine umfassende Betrachtung.
- Wir konnten in dieser Arbeit die Zusammenhänge zwischen State-Charts und Vererbung einerseits sowie State-Charts und dem Vertragsmodell andererseits nur kurz andeuten. In diesem Bereich sind mit Sicherheit weitere Arbeiten interessant und sinnvoll.
- Die Verwendung von State-Charts zur Spezifikation, Konstruktion und Dokumentation objektorientierter Softwaresysteme bedarf weiterer Untersuchung.

Visualisierung und Dokumentation

- Unserer Meinung nach existiert im Bereich der Visualisierung und Spezifikation von dynamischem Objektverhalten noch keine wirklich gute Notation. Interaktionsdiagramme sind zwar anschaulich, zeigen aber immer nur einen exemplarischen Interaktionsverlauf und können daher nicht zur Spezifikation eingesetzt werden. Außerdem werden sie unübersichtlich, wenn eine größere Anzahl von Objekten betrachtet werden soll. Andere Notationen, wie z.B. State-Charts lassen sich zwar

zur Spezifikation verwenden, sind aber z.T. wenig anschaulich und werden schnell unhandlich. Objektdiagramme können zwar mehr Objekte aufnehmen, als Interaktionsdiagramme. Bei vielfältiger Objektkommunikation verliert man hier jedoch ebenfalls schnell die Übersicht, weil nicht ohne weiteres die jeweils nächste Objektkommunikation ausfindig gemacht werden kann.
Möglicherweise bieten die in [WN95] vorgestellten Notationen nützliche Ansatzpunkte in diesem Bereich[37].

- Damit Muster eine weite Verbreitung finden, müssen sie gut dokumentiert sein. Hypertextsysteme wie z.B. das World-Wide-Web (WWW) sind dafür gut geeignet. Während eine gute Form zu Musterpräsentation in Textform von [GHJ+94] vorgeschlagen wurde, fehlt eine solche noch für Hypertextsysteme.

Wir möchten an dieser Stelle all jenen danken, welche uns in vielerlei Form bei der Erstellung dieser Arbeit geholfen haben. Besonderer Dank gebührt den Mitgliedern des MoMo-Seminars, welche sich nicht nur bereits am frühen Montag Morgen an der Uni eingefunden haben, sondern uns immer wieder durch Anregungen und Kritik neue Denkanstöße gegeben haben. Stellvertretend für alle seien namentlich Heinz Züllighoven, Carola Lilienthal und Ingrid Wetzel erwähnt. Bei Ingrid Wetzel möchten wir uns außerdem für die anregenden Diskussionen im Rahmen der Betreuung dieser Arbeit bedanken. Ihr ist es zu verdanken, daß wir uns detailliert mit der Terminologie zum Thema auseinandergesetzt haben.
Ein besonderer Dank geht auch an Dirk Riehle, dessen Event-Notification-Muster uns als Grundlage diente und der uns immer zu einer EMail-Diskussion zur Verfügung stand.

---

[37]Da das Buch erst spät zu unserer Verfügung stand, sind die dort vorgeschlagenen Notationen nicht in diese Studienarbeit eingeflossen.

8. Literatur

BBS+95
Dirk Bäumer, Reinhard Budde, Karl-Heinz Sylla, Guido Gryczan, Heinz Züllighoven: *Objektorientierte Konstruktion von Software-Werkzeugen und -Materialien.* Hamburg: Unveröffentlicht. 1995.
Boo91
Grady Booch. *Object Oriented Design With Applications.* Redwood City, Californien, 1991.
Bro88
Brockhaus. Mannheim: DTV, 1988
EV94
Interactive Software Engineering Inc. *Eiffel-Vision.* Coleta, Californien, 1994.
Gam92
Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++.* Berlin Heidelberg: Springer-Verlag, 1992.
GHJ+94
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley, 1994.
GN91
Garlan, Notkin: “Formalizing Design Spaces: Implicit Invocation Mechanisms” in *VDM ‘91, Formal Software Development Methods. Lecture Notes in Computer Science 551*, Springer Verlag, New York, 1991, 31-44
GR93
Adele Goldberg, David Robson: *Smalltalk-80: The language and Its Implementation.* Reading, Massachusetts: Addison-Wesley, 1993.
Gra95
Sven Grand. *Trennung von Interaktion und Funktion in der Werkzeug und Material Metapher.* Diplomarbeit. Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1995.
Har88
David Harel. “On visual formalisms” in *Communications of ACM 31*, 5 (Mai 1988), 514-530.
KGZ93
Klaus Kilberth, Guido Gryczan, Hein Züllighoven. *Objektorientierte Anwendungsentwicklung.* Braunschweig/Wiesbaden: Vieweg, 1993.
KP88
Glenn E. Krasner, Stephen T. Pope. “A cookbook for using the model-view controller user interface paradigm in Smalltalk-80", in *Journal of Object-Oriented Programming*, 1(3):26-49, August/September 1988.
Lam88
Günther Lamprecht. *Simula. Einführung in die Programmiersprache.* Braunschweig/Wiebaden: Vieweg, 1988.
Lie85
Henry Liebermann. “There’s more to menu systems than meets the screen” in *ACM SIGGRAPH Computer Graphics*. San Francisco, CA, July 1985. 181-189l.

LVC89
Mark A. Linton, John Vlissides, Paul R. Calder. “Composing user Interfaces with Inter-Views.” in *Computer*, 22(2): 8-22, February 1989.
Mey90
Bertrand Meyer. *Objektorientierte Softwareentwicklung.* London: Prentice-Hall, 1990.
Mey91
Betrand Meyer. *Introduction to the Theory of Programming Languages.* New York, London: Prentice-Hall, 1991.
Mey92
Betrand Meyer. *Eiffel. The Language.* New York, London: Prentice-Hall, 1992.
Mey94
Betrand Meyer. *Reusable Software. The Base Object-Oriented Component Libraries.* Hemel Hempstead, Herfordshire: Prentice Hall, 1994.
Min67
Marvin Minsky. *Computation: Finite and Infinite Machines.* Englewood-Cliffs, New Jersey: Prentice Hall, 1967.
NGG+93
David Notkin, David Garlan, William G. Griswold, Kevin Sullivan. “Adding Implicit Invocation to Languages: Three Approaches”. JSSST-93, LNCS-742, *Object Technology for Advanced Software.* Edited by Shojiro Nishio and Akinori Yonezawa. New York: Springer-Verlag, 1993. 489-510.
P+88
Andrew J. Palay et al. “The Andrew Toolkit: An Overview”. *Proceedings of the 1988 Winter USENIX Technical Conference.* Dallas, Texas: USENIX Association, 1988. 9-21.
Rie93a
Dirk Riehle. *Dokumentation zur IATMotif-Bibliothek.* Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1993.
Rie93b
Dirk Riehle. *Dokumentation zur FIAK-Bibliothek.* Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1993.
Rie95
Dirk Riehle. *Muster am Beispiel der Werkzeug und Material Metapher.* Diplomarbeit. Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1995.
Rie95a
Dirk Riehle. *An Event Notification Pattern.* Unveröffentlicht. Zürich, Schweiz, 1995.
RBP+91
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Objekt-Oriented Modelling and Design.* Englewood-Cliffs, New Jersey: Prentice Hall, 1991.
RZ94
Dirk Riehle, Heinz Züllighoven. “A Pattern Language for Tool Construction and Integration Based on the Tools&Materials Metaphor”. *PloP.* Reading, Massasuchets, 1994.
Str91
Bjarne Stroustrup. *The C++ Programming Language.* Reading, Massachusetts: Addison Wesley, 1991.
SN92
Sullivan, Notkin: “Reconciling Environment Integration and Software Eveolution” in *ACM Transactions on Software Engineering and Methodology*. Vol.1, No. 3, Juli 1992, 229-

268.

SV94
Star-Division. *Star-View-Framework Version 2.21.* Hamburg, 1994.
VL90
John M. Vlissides, Mark A. Linton. “Unidraw: A framework for building domain-specific graphical editors” in *ACM Transactions on Information Systems*, 8(3): 237-268, Juli 1990.
Vli95
John Vlissides. *Using Design Patterns: Elements of Reusable Architectures.* New York: IBM T.J. Watson Research Center, 1995.
WN95
Kim Waldén, Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems.* Englewood Cliffs, New Jersey: Prentice Hall, 1995.
WWW90
Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software.* Englewood Cliffs, New Jersey: Prentice Hall, 1990.

Im folgenden findet sich ein Großteil unserer Implementation des Event-Oberserver-Musters in C++. Da wir die Quelltexte direkt übernommen haben, fehlen jegliche deutschen Umlaute und Sonderzeichen.

Wir haben versucht, Teile des Vertragsmodells in Form von Vorbedingungen und Klasseninvarianten nachzubilden. In der aktuellen Form haben sie nur dokumentatorischen Chararakter. Die in der Datei "Contract.H" zu diesem Zweck definierten Makros sind leer. Um eine Überprüfung der Vorbedingungen zu erreichen, müßten entsprechende Makros zusätzlich in den Routinenimplementationen benutzt werden. Für eine Überprüfung der Klasseninvarianten müßten zusätzliche Makros am Ende einer jeden Routinenimplementation benutzt werden.

// Datei Oberserver.H

```
//
// Klasse: Observer
// Muster: Event-Observer
//
// Jede Klasse, welche andere Klassen beobachten will,
// muß von dieser Klasse erben.
//
// Es koennen nur Klassen beobachtet werden, welche
// von Observable erben.
//

#ifndef _Observer
#define _Observer

class tObserver
{
protected:

  // Konstruktor
  tObserver ( );

  // Destruktor
  virtual ~tObserver ( );
};

#endif
```

```
// Datei Observer.C

#include "Observer.H"

// Konstruktor
tObserver::tObserver ()
{
}

// Destruktor
tObserver::~tObserver ()
{
}
```

```
// Datei Observable.H

//
// Klasse: Observable
// Muster: Event-Observer
//
// Jede Klasse, welche von anderen beobachtet werden soll,
// muß von dieser Klasse erben.
//
// Klassen koennen nur von Klassen beobachtet werden,
// welche von Observer erben.
//

#ifndef _Observable
#define _Observable

class tEventBase; // Forward-Deklaration

class tObservable
{
public:

  // Kontruktoren & Destruktoren
  tObservable ( );
  virtual ~tObservable ( );

  // War "anEvent" der vorige Event?
  virtual bool WasPreviousEvent (tEventBase* anEvent);

  // Beginne Announce-Ablauf.
  virtual void StartAnnounce(tEventBase* announcedEvent);

  // Beende Announce-Ablauf.
  virtual void EndAnnounce( tEventBase* announcedEvent );

  // Befindet sich das Objekt gerade im Zustand des
  // Signalisierens?
  virtual bool IsAnnouncing ();

protected:

  tEventBase* pActualEvent;
  // der Event, welcher gerade bekannt gemacht wird oder
  // zuletzt wurde

  tEventBase* pPreviousEvent;
  // der vor dem aktuellen Event bekanntgemachte Event
```

```
  bool bIsAnnouncing;
  // befindet sich das Objekt gerade im Zustand des
  // Signalisierens

};

#endif
```

```
// Datei Observable.C

#include "Observable.H"
#include "EventBase.H"

// Konstruktor
tObservable::tObservable ()
{
  pActualEvent = 0;
  pPreviousEvent = 0;
  bIsAnnouncing = false;
}

// Destruktor
tObservable::~tObservable ()
{
  pActualEvent = 0;
  pPreviousEvent = 0;
  bIsAnnouncing = false;
}

// Beginne mit dem Bekanntmachen von 'announcedEvent'
void tObservable::StartAnnounce( tEventBase*
                                 announcedEvent )
{
  bIsAnnouncing = true;
  pPreviousEvent = pActualEvent;
  pActualEvent = announcedEvent;
}

// Beende Ereignisversendung
void tObservable::EndAnnounce( tEventBase* announcedEvent
)
{
  bIsAnnouncing = false;
};

// Wird gerade ein Ereignis versendet?
bool tObservable::IsAnnouncing ()
{
  return bIsAnnouncing;
};

// War 'anEvent' der vorige Event?
bool tObservable::WasPreviousEvent (tEventBase* anEvent)
{
  return anEvent == pPreviousEvent;
};
```

// Datei EventBase.H

```
//
// Klasse: tEventBase
// Muster: Event-Observer
//
// Diese Klasse ist aufgeschoben.
// Beim Beobachteten sollte es fuer jeden Event eine
// Funktion geben, welche den Event zurueckgibt. Die
// Event-Instanz-Variable muss dynamisch vom Typ tEvent#
// sein, die Funktion sollte aber nur tEventBase# liefern,
// damit das Announce nicht von unbefugten aufgerufen
// werden kann.
//
// Events muessen beim Beobachteten deklariert werden.
// Fuer einen Event mit einem Parameter ist tEventBase1 zu
// benutzen, fuer einen Event mit zwei Parametern
// tEventBase2 usw.
//

#ifndef _EventBase
#define _EventBase

#include "set.h"
#include "Contract.H"
#include "Observable.H"

class tEventBase
{
public:

  // der Besitzer dieses Events
  virtual tObservable* Owner ( ) = 0;

  // Hashcode
  virtual unsigned hash () const;

  // Jetzt Ereignis tatsaechlich melden.
  virtual void Continue ();

protected:

  // Protected-Konstruktor, um Instanziierung von Objekten
  // dieser Klasse zu verhindern
  tEventBase ( );

  // Protected-Destruktor symmetrisch zum
  // Protected-Konstruktor
  virtual ~tEventBase ( );

  // Existiert dieses Objekt ueberhaupt noch?
  virtual bool DoIExist ();
```

```
};

#endif
```

// Datei EventBase.C

```
#include "EventBase.H"

// Liefert die Menge aller existierender Instanzen dieser
// Klasse.
static tSet<tEventBase*>* Table()
{
  static tIDSetAsHashBag<tEventBase*> pTable;
  return &pTable;
};

// Hashcode
unsigned tEventBase::hash () const
{
  return unsigned(this);
};

// Protected-Konstruktor, um Instanziierung von Objekten
// dieser Klasse zu verhindern
tEventBase::tEventBase ( )
{
  Table()->Add( this );
};

// Protected-Destruktor symmetrisch zum
// Protected-Konstruktor
tEventBase::~tEventBase ( )
{
  Table()->Remove( this );
};

// Existiert dieses Objekt ueberhaupt noch?
bool tEventBase::DoIExist ()
{
  return Table()->Contains( this );
};

// Jetzt Ereignis tatsaechlich melden
void tEventBase::Continue ()
{
  // ist von den Subklassen zu implementieren
};
```

// Datei EventBase0.H

```
//
// Klasse: tEventBase0
// Muster: Event-Observer
//
// Diese Klasse ist aufgeschoben.
//
// Beim Beobachteten sollte es fuer jeden Event eine
// Funktion geben, welche den Event zurueckgibt. Die
// Event-Exemplarariable muss dynamisch vom Typ tEvent#
// sein, die Funktion sollte aber nur tEventBase#
// liefern, damit das Announce nicht von unbefugten
// aufgerufen werden kann.
//
// Events muessen beim Beobachteten deklariert werden.
// Fuer einen Event mit einem Parameter ist tEventBase1 zu
// benutzen, fuer einen Event mit zwei Parametern
// tEventBase2 usw.
//

#ifndef _EventBase0
#define _EventBase0

#include "Contract.H"
#include "EventLink0.H"
#include "EventStub0.H"
#include "EventBase.H"

class tEventBase0 : public tEventBase
{
public:

  // registriere 'eventLink' als Beobachter
  virtual void Register ( tEventLink0* eventLink ) = 0;
    require( eventLink != NULL )

  // mache Registrierung rueckgaengig
  // require: eventLink /= NULL
  virtual void Unregister ( tEventLink0* eventLink ) = 0;

};

#endif
```

// Datei EventBase1.H

```
//
// Klasse: tEventBase1
// Muster: Event-Observer
//
// Diese Klasse ist aufgeschoben.
// Beim Beobachteten sollte es fuer jeden Event eine
// Funktion geben, welche den Event zurueckgibt. Die
// Event-Exemplarvariable muss dynamisch vom
// Typ tEvent# sein, die Funktion sollte aber nur
// tEventBase# liefern, damit das Announce nicht von
// unbefugten aufgerufen werden kann.
//
// Events muessen beim Beobachteten deklariert werden.
// Fuer einen Event mit einem Parameter ist tEventBase1 zu
// benutzen, fuer einen Event mit zwei Parametern
// tEventBase2 usw.
//

#ifndef _EventBase1
#define _EventBase1

#include "Contract.H"
#include "EventStub1.H"
#include "EventBase.H"

template <class A1>
class tEventBase1 : public tEventBase
{
public:

  // registriere 'eventLink' als Beobachter
  virtual void Register (tEventLink1<A1>* eventLink) = 0;
    require( eventLink != NULL )

  // mache Registrierung rueckgaengig
  // require: eventLink /= NULL
  virtual void Unregister(tEventLink1<A1>* eventLink) = 0;
    require( eventLink != NULL )

};

#endif

// Klassen tEventBase2 bis tEventBase5 entsprechend
```

// Datei Event0.H

```
//
// Klasse: tEvent0
// Muster: Event-Observer
//
// Beim Beobachteten sollte es fuer jeden Event eine
// Funktion geben, welche den Event zurueckgibt. Die
// Event-Exemplarariable muss dynamisch vom Typ tEvent#
// sein, die Funktion sollte aber nur tEventBase# liefern,
// damit das Announce nicht von Unbefugten aufgerufen
// werden kann.
//
// Events muessen beim Beobachteten deklariert werden.
// Fuer einen Event ohne Parameter ist tEvent0 zu
// benutzen, fuer einen Event mit einem Parameter tEvent1
// usw.
//

#ifndef _Event0
#define _Event0

#include "Contract.H"
#include "collect.h"
#include "list.h"
#include "EventManager.H"
#include "EventStub0.H"
#include "EventBase0.H"

class tEvent0 : public tEventBase0
{
public:

  // Konstruktor
  // 'owner' ist der Ausloeser des Events
  tEvent0 ( tObservable* owner );
     require(owner != NULL);

  // Destruktor
  virtual ~tEvent0 ( );

  // registriere 'eventLink' als Beobachter
  virtual void Register ( tEventLink0* eventLink );
     require(eventLink != NULL);

  // mache Registrierung rueckgaengig
  virtual void Unregister ( tEventLink0* eventLink );
     require(eventLink != NULL);

  // melde Event an alle Beobachter
  virtual void Announce ( );
```

```
  // setze durch Announce angestossene Benachrichtigung
  // fort
  // wird vom Event-Manager benutzt
  virtual void Continue ( );

  // der Besitzer dieses Events
  virtual tObservable* Owner ( );

protected:

  // loese pToAdd auf
  virtual void AddLate ();

  // loese pToRemove auf
  virtual void RemoveLate ();

  tObservable* pOwner;
  // Besitzer des Events

  tCollection< tEventLink0* >* pLinks;
  // angemeldete EventLinks

  tCollection< tEventLink0* >* pToAdd;
  // noch anzumeldende EventLinks

  tCollection< tEventLink0* >* pToRemove;
  // noch abzumeldende EventLinks

  invariant( pOwner /= NULL && pLinks /= NULL )

};

#endif
```

```
// Datei Event0.C

#include <fstream.h>
#include "Event0.H"

// loese pToAdd auf
void tEvent0::AddLate ()
{
  pLinks->AddAll( pToAdd );
  pToAdd->MakeEmpty();
};

// loese pToRemove auf
void tEvent0::RemoveLate ()
{
  pLinks->RemoveAll( pToRemove );
  pToRemove->MakeEmpty();
};

// 'owner' ist der Ausloeser des Events
tEvent0::tEvent0 ( tObservable* owner )
{
  pOwner = owner;
  pLinks = new tIDLinkedList< tEventLink0* >;
  pToAdd = new tIDLinkedList< tEventLink0* >;
  pToRemove = new tIDLinkedList< tEventLink0* >;
};

// Destruktor
tEvent0::~tEvent0 ( )
{
  delete pLinks;
  delete pToAdd;
  delete pToRemove;
};

// registriere 'eventLink' als Beobachter
void tEvent0::Register ( tEventLink0* eventLink )
{
  if  ( pToRemove->Contains( eventLink ) )
  {
     pToRemove -> Remove( eventLink );
  };

  pToAdd -> Add( eventLink );
};
```

```
// mache Registrierung rueckgaengig
void tEvent0::Unregister  ( tEventLink0* eventLink )
{
  if ( pToAdd->Contains( eventLink ) )
  {
     pToAdd -> Remove( eventLink );
  };

  pToRemove -> Add( eventLink );
};

// Melde Ereignis an alle angemeldeten Event-Stubs
void tEvent0::Announce ()
{
  if ( pOwner->IsAnnouncing() )
  {
     cout << "Reaktive Aenderung bei Event " << this <<
             " des Subjektes " << pOwner << "\n";
     exit(1);
  }
  tEventManager::Instance() -> Announce(this);
};

// setze die durch Announce begonnene Benachrichtigung
// fort
void tEvent0::Continue ( )
{
  // aktuellen und letzten Event neu setzen
  pOwner->StartAnnounce( this );
  RemoveLate();
  AddLate();
  tCursor< tEventLink0* >* pCursor;
  pCursor = new tCursor<tEventLink0*>( pLinks );
  while ( pCursor -> Forth() )
  {
     if ( DoIExist() )
     {
       pCursor->Current()->Forward( pOwner );
     };
  };
  pCursor -> Logout();
  delete pCursor;
  pOwner->EndAnnounce( this );
};

// der Besitzer dieses Events
tObservable* tEvent0::Owner ( )
{
  return pOwner;
};
```

// Datei Event1.H

```
//
// Klasse: tEvent1
// Muster: Event-Observer
//
// Beim Beobachteten sollte es fuer jeden Event eine
// Funktion geben, welche den Event zurueckgibt. Die
// Event-Exemplarariable muss dynamisch vom Typ tEvent#
// sein, die Funktion sollte aber nur tEventBase# liefern,
// damit das Announce nicht von unbefugten aufgerufen
// werden kann.
//
// Events muessen beim Beobachteten deklariert werden.
// Fuer einen Event ohne Parameter ist tEvent0 zu
// benutzen, fuer einen Event mit einem Parameter tEvent1
// usw.
//
// Aufgrund von Problemen mit dem Gnu-Linker, stehen
// Klassendefinition und Implementation bei allen
// Template-Klassen in einer Datei.
//

#ifndef _Event1
#define _Event1

#include "Contract.H"
#include "collect.h"
#include "list.h"
#include "EventManager.H"
#include "EventStub1.H"
#include "EventBase1.H"

template <class A1> class tEvent1 : public tEventBase1<A1>
{
public:

  // Konstruktor
  // 'owner' ist der Ausloeser des Events
  tEvent1 ( tObservable* owner );
    require( owner != NULL )

  // Destruktor
  virtual ~tEvent1 ( );

  // registriere 'eventLink' als Beobachter
  virtual void Register ( tEventLink1<A1>* eventLink );
    require( eventLink != NULL )

  // mache Registrierung rueckgaengig
  virtual void Unregister ( tEventLink1<A1>* eventLink );
    require( eventLink != NULL )
```

```
  // melde Event an alle Beobachter
  virtual void Announce ( A1 arg1 );

  // setze Benachrichtigung fort
  virtual void Continue ();

  // der Besitzer dieses Events
  virtual tObservable* Owner ( );

protected:

  // loese pToAdd auf
  virtual void AddLate ();

  // loese pToRemove auf
  virtual void RemoveLate ();

  tObservable* pOwner;
  // Besitzer des Events

  tCollection< tEventLink1<A1>* >* pLinks;
  // angemeldete EventLinks

  tCollection< tEventLink1<A1>* >* pToAdd;
  // noch anzumeldende EventLinks

  tCollection< tEventLink1<A1>* >* pToRemove;
  // noch abzumeldende EventLinks

  A1 argument1;
  // Ereignis-Parameter

  invariant( pOwner != NULL && pLinks != NULL )

};

//
// Implementation
//

// loese pToAdd auf
template <class A1>
void tEvent1< A1 >::AddLate ()
{
  pLinks->AddAll( pToAdd );
  pToAdd->MakeEmpty();
};
```

```
// loese pToRemove auf
template <class A1>
void tEvent1< A1 >::RemoveLate ()
{
  pLinks->RemoveAll( pToRemove );
  pToRemove->MakeEmpty();
};

// 'owner' ist der Ausloeser des Events
template <class A1>
tEvent1< A1 >::tEvent1 ( tObservable* owner )
{
  pLinks = new tIDLinkedList< tEventLink1<A1>* >;
  pOwner = owner;
  pToAdd = new tIDLinkedList< tEventLink1<A1>* >;
  pToRemove = new tIDLinkedList< tEventLink1<A1>* >;
};

// Destruktor
template <class A1>
tEvent1< A1 >::~tEvent1 ( )
{
  delete pLinks;
  delete pToAdd;
  delete pToRemove;
};

// registriere 'eventLink' als Beobachter
template <class A1>
void tEvent1< A1 >::Register (tEventLink1<A1>* eventLink)
{
  if    ( pToRemove->Contains( eventLink ) )
  {
     pToRemove -> Remove( eventLink );
  };
  pToAdd -> Add( eventLink );
};

// mache Registrierung rueckgaengig
template <class A1>
void tEvent1< A1 >::Unregister ( tEventLink1<A1>* event-
Link )
{
  if    ( pToAdd->Contains( eventLink ) )
  {
     pToAdd -> Remove( eventLink );
  };
  pToRemove -> Add( eventLink );
};
```

```
// melde Event an alle Beobachter
template <class A1>
void tEvent1< A1 >::Announce ( A1 arg1 )
{
  argument1 = arg1;
  tEventManager::Instance() -> Announce(this);
};

// setze Benachrichtigung fort
template <class A1>
void tEvent1< A1 >::Continue ()
{
  if ( pOwner->IsAnnouncing() )
  {
     cout << "Reaktive Aenderung bei Event " << this <<
              " des Subjektes " << pOwner;
     exit(1);
  }
  // aktuellen und letzten Event neu setzen
  pOwner->StartAnnounce( this );
  RemoveLate();
  AddLate();
  tCursor< tEventLink1<A1>* >* pCursor;
  pCursor = new tCursor<tEventLink1<A1>*>( pLinks );
  while ( pCursor->Forth() )
  {
     if ( DoIExist() )
     {
       pCursor->Current()->Forward( pOwner, argument1 );
     };
  };
  pCursor -> Logout();
  delete pCursor;
  pOwner->EndAnnounce( this );
};

// der Besitzer dieses Events
template <class A1>
tObservable* tEvent1< A1 >::Owner ( )
{
  return pOwner;
};

#endif

// Klassen tEvent2 bis tEvent5 entsprechend
```

// Datei EventLink.H

```
//
// Klasse: tEventLink
// Muster: Event-Observer
//
// Ein EventLink stellt die Verbindung zwischen einem
// Event und weiteren EventLinks dar.
//
// Für einen Event mit einem Parameter ist tEventLink1 zu
// benutzen, für einen Event mit zwei Parametern
// tEventLink2 usw.
//

#ifndef _EventLink
#define _EventLink

class tEventLink
{
protected:

  // Konstruktor
  tEventLink ( );

  // Destruktor
  virtual ~tEventLink ( );
};

#endif
```

```
// Datei EventLink.C

#include "EventLink.H"

// Konstruktor
tEventLink::tEventLink ()
{
};

// Destruktor
tEventLink::~tEventLink ()
{
};
```

```
// Datei EventLink0.H

//
// Klasse: tEventLink0
// Muster: Event-Observer
//
// Ein EventLink stellt die Verbindung zwischen einem
// Event und weiteren EventLinks statt.
//
// Für einen Event mit einem Parameter ist tEventLink1 zu
// benutzen, für einen Event mit zwei Parametern
// tEventLink2 usw.
//

#ifndef _EventLink0
#define _EventLink0

#include "Observable.H"
#include "EventLink.H"

class tEventLink0 : public tEventLink
{
public:

  // Konstruktor
  tEventLink0 ( );

  // Destruktor
  virtual ~tEventLink0 ( );

  // melde Ereignis weiter
  virtual void Forward ( tObservable* sender );

};

#endif
```

```
// Datei EventLink0.C

#include "EventLink0.H"

// Konstruktor
tEventLink0::tEventLink0 ( )
{
};

// Destruktor
tEventLink0::~tEventLink0 ( )
{
};

// melde Ereignis weiter
void tEventLink0::Forward ( tObservable* sender )
{
};
```

// Datei EventLink1.H

```
//
// Klasse: tEventLink1
// Muster: Event-Observer
//
// Ein EventLink stellt die Verbindung zwischen einem
// Event und weiteren EventLinks statt.
//
// Für einen Event mit einem Parameter ist tEventLink1 zu
// benutzen, für einen Event mit zwei Parametern
// tEventLink2 usw.
//

#ifndef _EventLink1
#define _EventLink1

#include "Observable.H"
#include "EventLink.H"

template<class A1>
class tEventLink1 : public tEventLink
{
public:

  // Konstruktor
  tEventLink1 ( );

  // Destruktor
  virtual ~tEventLink1 ( );

  // gebe Parameter 'arg1' weiter
  virtual void Forward ( tObservable* sender, A1 arg1 );

};

//
// Implementation
//

// Konstruktor
template< class A1 >
tEventLink1< A1 >::tEventLink1 ( )
{
};

// Destruktor
template< class A1 >
tEventLink1< A1 >::~tEventLink1 ( )
{
};
```

```
// gebe Parameter 'arg1' weiter
template< class A1 >
void tEventLink1< A1 >::Forward ( tObservable* sender,
                                  A1 arg1 )
{
};

#endif

// Klassen tEventLink2 bis tEventLink5 entsprechend
```

// Datei EventStub0.H

```
//
// Klasse: tEventStub0
// Muster: Event-Observer
//
// Der EventStub bildet den Abschluß einer Kette von
// EventLinks.
//
// EventStubs sind immer beim Beobachter zu deklarieren.
//

#ifndef _EventStub0
#define _EventStub0

#include "Contract.H"
#include "Observer.H"
#include "EventLink0.H"

template<class O>
class tEventStub0 : public tEventLink0
{
public:

  // Konstruktor
  tEventStub0 ( tObserver* observer, O* concreteObserver,
                void(O::*notify) (tObservable*) );
    require( observer != NULL && obs != NULL &&
             observer == obs && notify != NULL )

  // Destruktor
  virtual ~tEventStub0 ( );

  // rufe die angemeldete Prozedur auf
  virtual void Forward ( tObservable* sender );

protected:

  tObserver* pObserver;  // Beobachter
  O* pConcreteObserver;  // == pObserver

  void(O::*pNotify) (tObservable*);
  // aufzurufende Prozedur

  invariant( tObserver != NULL && pNotify != NULL )

};
```

```
//
// Implementation
//

// Konstruktor
template <class O>
tEventStub0< O >::tEventStub0(tObserver* observer, O*
                              concreteObserver,
                              void(O::*notify) (tObserva-
ble*) )
{
  pObserver = observer;
  pConcreteObserver = concreteObserver;
  pNotify = notify;
};

// Destruktor
template <class O>
tEventStub0< O >::~tEventStub0 ( )
{
  pObserver = NULL;
  pConcreteObserver = NULL;
};

// rufe die angemeldete Prozedur auf
template <class O>
void tEventStub0< O >::Forward ( tObservable* sender )
{
  (pConcreteObserver->*pNotify) ( sender );
};

#endif
```

```
// Datei EventStub1.H

//
// Klasse: tEventStub1
// Muster: Event-Observer
//
// Der EventStub bildet den Abschluß einer Kette von
// EventLinks.
//
// EventStubs sind immer beim Beobachter zu deklarieren.
//

#ifndef _EventStub1
#define _EventStub1

#include "Contract.H"
#include "Observer.H"
#include "EventLink1.H"

template<class O, class A1>
class tEventStub1 : public tEventLink1<A1>
{
public:

  // Konstruktor
  tEventStub1 ( tObserver* observer, O* concreteObserver,
                void(O::*notify) (tObservable*, A1) );
    require( observer != NULL && obs != NULL &&
             observer == obs && notify != NULL )

  // Destruktor
  virtual ~tEventStub1 ( );

  // rufe die angemeldete Prozedur mit Parameter 'arg1'
 // auf
  virtual void Forward ( tObservable* sender, A1 arg1 );

protected:

  tObserver* pObserver; // Beobachter
  O* pConcreteObserver; // == pObserver

  void(O::*pNotify) (tObservable*, A1);
  // aufzurufende Prozedur

  invariant( tObserver != NULL && pNotify != NULL )

};
```

```
//
// Implementation
//

// Konstruktor
template< class O, class A1 >
tEventStub1< O, A1 >::tEventStub1(tObserver* observer,
                                  O* concreteObserver,
                                  void(O::*notify)
                                    (tObservable*, A1) )
{
  pObserver = observer;
  pConcreteObserver = concreteObserver;
  pNotify = notify;
};

// Destruktor
template< class O, class A1 >
tEventStub1< O, A1 >::~tEventStub1 ( )
{
  pObserver = NULL;
  pConcreteObserver = NULL;
};

// rufe die angemeldete Prozedur mit Parameter 'arg1' auf
template< class O, class A1 >
void tEventStub1< O, A1 >::Forward ( tObservable* sender,
                                     A1 arg1 )
{
  (pConcreteObserver->*pNotify) ( sender, arg1 );
};

#endif

// Klassen tEventStub2 bis tEventStub5 entsprechend.
```

// Datei EventManager.H

```
//
// Klasse: EventManager
// Muster: Event-Observer
//
// Der Event-Observer kontrolliert den globalen
// Event-Fluss. Die eigentliche Benachrichtigung wird
// jedoch von den einzelnen Event-Klassen durchgefuehrt.
//
// Damit alle Objekte denselben Event-Manager benutzen,
// ist dieser als SINGLETON realisiert..
//

#ifndef _EventManager
#define _EventManager

#include "Contract.H"
#include "queue.h"
#include "cursor.h"
#include "EventBase.H"

class tEventManager
{
public:

  // Hier gibt`s eine Referenz auf die einzige Instanz
  static tEventManager* Instance();

  // sorge fuer die Verschickung von `anEvent`
  virtual void Announce ( tEventBase* anEvent );
    require( anEvent != NULL )

protected:

  // Konstruktor
  tEventManager ( );

  // Destruktor
  virtual ~tEventManager ( );

  // verschicke direkt den Inhalt der Event-Queue
  virtual void AnnounceQueue ();

  tQueue<tEventBase*>* pQueue;
  // die Event-Queue

  bool bIsAnnouncing;
  // werden gerade Events verschickt?
```

```
  static tEventManager* pInstance;
  // einzige Instanz des Event-Managers

};
#endif
```

```
// Datei EventManager.C

#include "EventManager.H"

tEventManager* tEventManager::pInstance = 0;
// Initialisierung auf 0

// liefert einziges Singleton-Exemplar
tEventManager* tEventManager::Instance()
{
  if (pInstance == 0)
  {
     pInstance = new tEventManager;
  };
  return pInstance;
}

// Konstruktor
tEventManager::tEventManager ()
{
  bIsAnnouncing = false;
  pQueue = new tIDQueueAsLinkedList<tEventBase*>();
}

// Destruktor
tEventManager::~tEventManager ()
{
  bIsAnnouncing = false;
  delete pQueue;
  pQueue = NULL;
  delete pInstance;
  pInstance = NULL;
}

// sorge fuer die Verschickung von `anEvent`
void tEventManager::Announce ( tEventBase* anEvent )
{
  pQueue -> Put( anEvent );
  // Events werden in einer Queue (FIFO)
  // verwaltet, damit sich keine Events
  // ueberholen koennen.

  // wenn gerade kein `Announce` laeuft, jetzt
  // Verschickung starten
  if (! bIsAnnouncing)
  {
     AnnounceQueue();
  };
}
```

```
// alle Events, welche sich in der Queue befinden,
// verschicken
void tEventManager::AnnounceQueue ()
{
  bIsAnnouncing = true;

  while (! pQueue->IsEmpty() )
  {
     pQueue->Get()->Continue();
     // Get liefert hier das aelteste Element und entfernt
     // es dann aus der Queue
  };

  bIsAnnouncing = false;
}
```

Anhang B. Migrationsmuster für den Notifier-Mechanismus

Es folgt hier eine kleine Anleitung zur Umstellung der bisherigen Notifier-basierten Anwendungen, welche mit Hilfe der BibV21 oder früher realisiert wurden. Diese Anleitung ist rein technischer Natur, in dem Sinne, daß sie zu läuffähigen Anwendungen mit dem neuen Mechanismus führt. Die formalen Möglichkeiten der State-Charts sind damit natürlich noch lange nicht berührt. Um auch daraus Vorteile zu ziehen, müßten für die Subjektklassen State-Charts erstellt werden. Die bisher versendeten Ereignisse müßten dann mit den Zustandsübergängen in den State-Charts abgeglichen werden. Außerdem sind ggfs. weitere Ereignisse zu definieren. In der Praxis zeigt sich, daß die meisten existierenden Werkzeuge nur einen sehr kleinen Teil der tatsächlichen Zustandsänderungen tatsächlich via Ereignis signalisieren. Dieser kleine Teil deckt in der Regel gerade den Bereich ab, welchen man mit Interaktionskomponenten oder Kontext-Funktionskomponenten gerade benötigt. Wollte man weitere abhängige Klassen anschließen, müßten in der Subjektklasse nachträglich weitere Ereignisse verschickt werden.

Zuerst einmal ist anzumerken, daß für die Umstellung des Mechanismusses die BibV21, sowie die Klassen des Event-Observer-Musters notwendig sind. So ist sowhl der alte wie auch der neue Mechanismus parallel vorhanden. So bleiben die Anwendungen auch während der Umstellungsphase lauffähig. Erst nach der Umstellung kann auf die neue Bibliothek BibV30 umgestiegen werden, welche auf dem neuen Event-Observer-Muster beruht.

Umstellungsanleitung für beobachtete Klassen

Im der Definition (H-File) die Dateien “Event.H”, “EventBase.H” und “Observable.H” includen.

Klasse muß von tObservable erben (H-File).

In der Implementation (C-File) nach Tell-Aufrufen suchen.

In der Definition (H-File) im Protected-Teil einen entsprechenden Event mit den passenden Parametern definieren.

In der Definition eine Funkion zur Rückgabe des Events unter dem statischen Typ EventBase# definieren (wieder mit den passenden Parametertypen).

In der Implementation (C-File) den Tell-Aufruf ersetzen durch Announce mit den entsprechenden Parametern.

In der Definition (C-File) im Init den Event erzeugen und im Destruktor wieder löschen.

Umstellungsanleitung für Beobachter-Klassen:

In der Definition (H-File) die Dateien “Observer.H” und “EventStub.H” includen.

In der Definition (H-File) die Klasse von tObserver erben lassen.

In der Definition (H-File) eine Instanzvariable vom Typ EventStub mit den passenden Parametern definieren.

Neue Funktion zur Reaktion in der Definition (H-File) definieren mit den passenden Parametern.

Diese neue Funktion in der Implementation (C-File) implementieren mit der bisherigen Aktion auf diesen Event (siehe Notify-Reaktionszweig).

In der Implementation (C-File) aus dem Notify den Reaktionszweig für den Event entfernen.

EventStub in der Implementation (C-File) im Init erzeugen (mit der neuen Reaktionsprozedur als Parameter) und im Destruktor wieder löschen.

In der Implementation (C-File) EventStub im Konstruktor beim Event anmelden.

Allgemein

Wenn alle Events auf den neuen Mechanismus umgestellt sind, so kann die Vererbungsbeziehung zu tNotifier gelöst und die Prozedur `notify` gelöscht werden. Jetzt kann mit einer Bibliotheksversion (BibV30 oder später) gearbeitet werden, welche nicht mehr auf tNotifier beruht.

abhängige Komponente 10
abhängig 50
Abhängiger 12, 50
Änderungen 70, 81
reaktiv 81
reaktive 70
Änderungs-Benachrichtigung 11
Änderungs-Beobachtung 11
Anwendungen 87
Umstellung 87
Arbeitszusammenhang 17
Aspektklassen 17
Automat 17
Benachrichtiger 11
Benachrichtigter 12
Benachrichtigungen 66
Propagieren 66
Benachrichtigungs-Flußkontrolle 57
Benachrichtigungsbeziehung 6
Benachrichtigungsstrukturen 51
Ungünstige 51
Benutzeraktion 65, 70
Benutztbeziehung 5
Beobachter 12
Beobachter-Muster 26
Beobachtermuster 50
Beobachteter 11
Bibliothek 87
Umstellung 87
Change-Manager 57
Change-Update-Loop 51
Code 8
COMMAND 33
COMMAND-Muster 65, 70, 85
Dreiecks-Reaktionsbeziehung 53
Entwurfsmuster 16
Ereignis 13, 14
Ereignis-Benachrichtigung 11
Ereignis-Beobachter-Muster 72
Ereignis-Beobachtung 11
Ereignis-Muster 40
Ereignis-Parameter 64, 69
Ereignis-Wettlauf 56
Ereignisfluß-Steuerung 70
Ereignismuster 50
Ereignisse 68
Unterscheidbare 68

Event 73, 84
Event-Klassen 73
Event-Manager 70, 78
EVENT-NOTIFICATION 40
EVENT-OBSERVER 72
Event-Race 56
Event-Stub 73
Event-Stub-Klassen 82
EVENT_BASE 84
EVENT_LINK 73
Explizitmachung 68
forward-Prozedur 84
Funktion 17, 18
IAT-Events 77
Implementierung 9, 82
indirekten reaktiven Änderung 53
informierende Komponente 10
Interaktion 17, 18
Interaktionsdiagramm 7
Interaktionstyp 18
Klassen-/Objektdiagramm 6
Klassendiagramm 5
Klassendiagramme 5
Kommando-Muster 33, 70
Kommandomuster 50
Komponente 10
    abhängig 10
    informierende 10
    reaktive 13
Kontextwerkzeug 21
Leitbild 15, 57
    Reaktionsmechanismus 57
Materialien 16
Mechanismus 9
Metapher 15
Migrationsmuster 87
Muster 9
Nachricht 14, 50
Nachrichten 13
Netzwerkfähigkeit 69
Objekt 6
Objektdiagramm 6
Objektzustand 15
OBSERVABLE 72, 73
OBSERVER 26, 72, 73
Portierbarkeit 71
Präsentation 17, 18
Prioritäten 84
Propagieren 66

Proxy 69
Reaktionsmechanismus 4, 10
reaktive Änderung 51
reaktive Komponente 13
reaktiven Änderungen 70
reaktiver Änderungen 81
reaktives System 3
Realisierung 9
Signal 14, 50
State-Chart 59, 70
Subjekt 10, 50
Subjektzustand 65
    abstrakt 65
    konkret 65
Subwerkzeug 20
Ungültige Objektzustände 51
Vererbung 6
Vererbungsbeziehung 5
Verhalten
    dynamisch 7
WAM 17
Werkzeug 20
    komplex 20
Werkzeug&Material-Metapher 9
Werkzeugarchitektur 18
Werkzeuge 16, 17
Zustand 57
    abstrakt 57
Zustandsänderung 13
Zustandsübergangsdiagramm 59