Großer Beleg zum Thema

Das Framework „SalesPoint“: Technische Beschreibung der Version 2.0 und weiterer Ausbaumöglichkeiten

Bearbeitet von Steffen Zschaler
geb. am 07.12.1976 in Dresden

an der

TU Dresden	UniBw München
Fakultät Informatik	Fakultät Informatik
Lehrstuhl Softwaretechnologie	Institut II

Betreuer:	Dr. Birgit Demuth
	Dr. Lothar Schmitz
Verantwortlicher Hochschullehrer:	Prof. Dr. Heinrich Hussmann
Eingereicht am:	01.11.2000

1. Einleitung ... 3
2. Entwurf und Implementation ... 5
2.1. Begriffe und Konzepte ... 5
2.1.1. Applikationsverwaltung ... 5
2.1.2. Datenverwaltung ... 8
2.1.3. Benutzermanagement ... 10
2.1.4. Protokollverwaltung ... 11
2.1.5. GUI ... 11
2.2. Hinweise zur Implementation ... 12
2.2.1. Unterschiede zwischen Entwurf und Implementation ... 12
2.2.2. Style Guide ... 13
2.2.3. Implementationsmuster ... 13
2.2.3.1. Monitore ... 14
2.2.3.2. Sperren des Datenkorbs in der Pure Java Implementation der Datenklassen ... 14
2.2.3.3. `Listener` und Adapter ... 15
2.2.3.4. Partner und Kontexte ... 15
2.2.3.5. Standardformulare ... 16
2.2.3.6. Inhalte von Swing-Komponenten auswerten ... 16
2.2.4. Paketstruktur ... 16
2.3. Entwurfsmuster ... 17
2.3.1. Factory Method ... 17
2.3.2. Singleton ... 17
2.3.3. Adapter ... 18
2.3.4. Bridge ... 18
2.3.5. Composite ... 19
2.3.6. Decorator ... 19
2.3.7. Proxy ... 20
2.3.8. Command ... 20
2.3.9. Observer ... 20
2.3.10. Strategy ... 21
3. Verwandte Arbeiten ... 22
3.1. Taligent: Building Object-Oriented Frameworks ... 22
3.2. IBM San Francisco ... 23
4. Ausbaumöglichkeiten ... 24
4.1. JDBC – Anbindung ... 24
4.2. RMI – Einsatz ... 24
4.2.1. „Basar“ – Betrieb ... 24
4.2.2. Vertriebsketten ... 25
4.3. Werkzeug zur Anwendungsentwicklung ... 25
Literaturverzeichnis ... 26
Anhänge ... 27
A Framework – Doclet ... 27
B Style Guide ... 31
B.1 Namenskonventionen ... 31
B.1.1 Adaptierte Ungarische Notation ... 32
B.2 Klammern und Einrückungen ... 33
B.3 Kommentare ... 34
B.4 Modifikatoren ... 34
C Metriken ... 35
D Inhalt der CD-ROM ... 36

1. Einleitung

In der modernen Softwareentwicklung spielt die Verwendung objektorientierter Frameworks und die Orientierung auf objekt- und komponentenorientierte Programmierung eine zunehmende Rolle. Diese Konzepte erlauben sehr effiziente Wiederverwendung bereits erstellten Codes und sind damit eine wichtige Möglichkeit zur Senkung der Softwareentwicklungskosten. Die Verwendung von Frameworks stellt aber auch besondere Anforderungen an den Entwickler. Es erscheint daher sinnvoll, in einer softwaretechnologischen Ausbildung Studenten die Möglichkeit zu bieten, den Einsatz von Frameworks in der Praxis zu erproben.

Dieser Beleg beschreibt das Framework „SalesPoint“, welches im Rahmen des Softwarepraktikums an der TU Dresden sowie an der UniBw München eingesetzt wird. Die Darstellung wendet sich primär an Entwickler, die das System künftig warten und weiterentwickeln sollen. Die Übersicht und Begriffserläuterung in Abschnitt 2.1 könnte aber auch für Anwender des Frameworks nützlich sein.

Der Hauptteil des Belegs (Abschnitt 2) gibt einen Überblick über den Entwurf und die Implementation des Frameworks, wobei vor allem auf interne Konzepte, Strukturen und die Anwendung von Entwurfsmustern und weniger auf die Anwendung des Frameworks zur Entwicklung von Applikationen eingegangen wird. Weiterhin geht der Beleg auf verwendete Implementationsmuster, d.h. konsistent verwendete Implementationsidiome, ein. Abschnitt 3 versucht, das SalesPoint Framework in die existierende Literatur einzuordnen. In Abschnitt 4 schließlich werden einige angedachte Erweiterungsmöglichkeiten kurz vorgestellt. Dieser Abschnitt ist als eine Ideensammlung mit Hinweisen auf bereits vorbereitete Ansatzpunkte gedacht.

In den Anhängen findet sich eine kurze Vorstellung des zur Dokumentationsgenerierung erstellten Werkzeugs, eine Übersicht über die Namens- und Stilkonventionen, die im Quellcode des Frameworks verwendet wurden sowie ein Verzeichnis der auf der CD zum Beleg befindlichen Materialien.

Zunächst jedoch soll ein kurzer Abriß über Zustandekommen, Motivation und Entwicklung des Frameworks seit der Version 0.5 gegeben werden.

Im Jahr 1996 kam Herr Dr. Schmitz als Lehrstuhlvertretung an den Lehrstuhl für Softwaretechnologie der Fakultät für Informatik der Technischen Universität Dresden. Er hatte sich schon in München, wo er an der Universität der Bundeswehr arbeitet, mit dem Einsatz von Frameworks in studienbegleitenden Programmierpraktika beschäftigt. Bis zu diesem Zeitpunkt war es dabei jedoch stärker darum gegangen, daß die Studenten selbst über mehrere Stufen ein Framework entwickeln und die entsprechenden Auswirkungen auf die Anwendungsentwicklung beobachten. Die Aufgabe, ein komplettes Framework zu entwickeln, stellte sich jedoch als sehr komplex heraus, und so blieben die Ergebnisse der Praktika in dieser Hinsicht wenig befriedigend.

In Dresden gehörte ein einsemestriges Programmierpraktikum schon länger zum Studienplan. Dieses war bisher als C++ Praktikum durchgeführt wurden. Mit der wachsenden Bekanntheit von Java wurde der Entschluß gefaßt, die gesamte Ausbildung im Fach Softwaretechnologie auf Java umzustellen.

Im Rahmen dieser Umstellung ergab sich nun der Gedanke, den Studenten ein Framework als Unterstützung bei der Entwicklung ihrer eigenen Anwendungen an die Hand zu geben. Mehrere Punkte sprachen für diese Entscheidung:

- Anspruchsvolle Entwurfsaufgaben sprengen in ihrer Umsetzung leicht den zeitlichen Rahmen eines einsemestrigen Praktikums da die Studenten häufig noch sehr geringe Erfahrungen im objektorientierten Entwurf sowie in der Teamarbeit haben. Auf der anderen Seite neigen Aufgaben, die mit Sicherheit im zeitlichen Rahmen zu bewältigen sind, dazu, wenig interessant oder anspruchsvoll zu sein. Um diesen Konflikt zu entschärfen, kann man den Studenten zwar einerseits eine anspruchsvolle und interessante Aufgabe geben, ihnen andererseits aber gewissermaßen „auf halbem Wege entgegenkommen“, indem man ein Framework als Unterstützung anbietet.
- Gut konstruierte Frameworks haben auch einen didaktischen Effekt, indem sie guten objektorientierten Entwurf am Beispiel vormachen.
- Das Praktikum kann durch die Verwendung eines Frameworks und durch die damit verbundenen neuen oder geänderten Phasen der Anwendungsentwicklung (Einarbeitung in das Framework, Entwurf der eigenen Anwendung auf Basis des Frameworks,…) stärker realitätsorientiert werden, da es die auch in der Wirtschaft häufig auftretende Situation simuliert, daß man sich in ein existierendes System einarbeiten muß, auf dessen Basis man anschließend eine Anwendung programmiert.
- Es ergeben sich interessante Wettbewerbseffekte zwischen den einzelnen Gruppen, da diese sehr ähnliche, teilweise die selben, Aufgabenstellungen erhalten können (und müssen).

So wurde im Sommer 1997 die erste Version des Frameworks „SalesPoint“ erstellt. Diese erfüllte zumindest Punkt 2 der oben genannten Punkte (gutes Design durch gutes Vorbild) noch nicht und wurde deshalb im Nach-

hinein zu Version 0.5 umgetauft. Die Entwicklung der Version 0.5 war noch sehr stark von automatenorientierten Beispielen beeinflußt, wie zum Beispiel von der Vorstellung eines Fahrkartenautomaten als potentieller Anwendung. Die Framework-Architektur hatte stark automatenorientierten Charakter, es gab keine Unterstützung für Nebenläufigkeit, die Möglichkeiten der GUI-Verwendung waren sehr beschränkt. Außerdem war diese Version noch wenig flexibel und entsprechend mühsam in der Handhabung für die Anwendungsentwickler, d.h. die Studenten.

Der Quelltext zu dieser Version findet sich auf der beigefügten CD.

Aufbauend auf den Erfahrungen und Hinweisen der Studenten wurde im Sommer 1998 eine neue Version – Version 1.0 – entwickelt, deren Quellen ebenfalls auf der CD vorliegen. Diese hatte bereits einige der schlimmeren Probleme der Vorversion hinter sich gelassen. Die Verwendung einer grafischen Oberfläche wurde leichter, neue oder selbstentwickelte GUI-Komponenten konnten einfach integriert werden. Noch immer gab es jedoch keine echte Lösung für Nebenläufigkeit oder für ein echtes Transaktionskonzept. Insgesamt waren die Studenten zwar zufriedener als mit Version 0.5, aber die Flexibilität ließ noch immer zu wünschen übrig: Viele Teile des Frameworks waren einfach zu eng gekoppelt.

Um hier einen deutlichen Fortschritt zu erzielen, wurde im Zeitraum April bis August 1999 an der Universität der Bundeswehr München (UniBwM) ein Projekt unter Federführung von Dr. Schmitz durchgeführt. Neben dem Frameworkentwickler waren daran zwei Studenten der UniBwM beteiligt, die im Herbst 1998 selbst an einem Praktikum auf Basis der Version 1.0 des Frameworks teilgenommen hatten. Diese bekamen die Aufgabe, eine Beispielapplikation mit der entstehenden neuen Fassung des Frameworks zu entwickeln und in Form eines Tutorials zu beschreiben. Auf diese Weise konnten bei der Überarbeitung des Frameworks Wünsche und Probleme von Anwendern laufend beobachtet und berücksichtigt werden.

Das Ergebnis dieser Arbeit ist die Version 2.0, welche einen kompletten Neuentwurf des Systems darstellt. Nebenläufigkeit von Abläufen und ein gewisses Maß an Transaktionsmanagement sind ins Zentrum des Frameworks gerückt. Das GUI wurde vereinheitlicht, der Umgang damit nochmals durch Factory-Methods vereinfacht. Es wurde versucht, die Framework-Komponenten möglichst weitgehend zu entkoppeln, wo immer möglich wurde „Observer“- oder „Strategy“-artigen Mustern der Vorrang vor direkter Kopplung gegeben. Das Persistenzmanagement wird weitgehend vom Framework übernommen, so daß die endgültige Anwendung es nur noch anzustoßen braucht.

Diese Version wird im weiteren Thema des Belegs sein. Sie wird von der „technischen“ Seite beleuchtet werden und nicht so sehr von der Seite der Anwender, das heißt Entwurf und Implementation des Frameworks werden betrachtet, nicht jedoch die Anwendung des Frameworks zur Entwicklung spezifischer Applikationen. Dazu gibt es jedoch einiges Material auf der beigefügten CD-ROM.

2. Entwurf und Implementation

In den folgenden Kapiteln soll eine Übersicht über den Entwurf sowie die Implementation des Frameworks gegeben werden. Dazu werden zunächst wichtige Begriffe und Konzepte erläutert, die im Framework umgesetzt werden. Zum Schluß wird noch kurz auf die Paketstruktur des Frameworks eingegangen. Anschließend wird auf die Verwendung von Entwurfsmustern im Framework hingewiesen.

2.1. Begriffe und Konzepte

Das Framework zerfällt in zwei große, verhältnismäßig unabhängige Bereiche: die Datenverwaltung und die dynamische Applikationsverwaltung. Außerdem gibt es noch die kleineren Bereiche Benutzermanagement, Protokollverwaltung und GUI. In diesem Abschnitt werden wichtige Begriffe und Konzepte aus diesen Bereichen erläutert.

Eine wichtige Entwurfsentscheidung war die Betonung einer Simulationssicht. Das heißt, Anwendungen, die mit dem SalesPoint - Framework entwickelt werden, sind eher Simulationen einer Verkaufsstellenumgebung als reale Anwendungen zur Unterstützung von Verkäufer oder Kunde. Diese Entscheidung wurde als Kompromiß zwischen Leistungsfähigkeit und Komplexität getroffen. Der Eindruck der Framework – Entwickler war es, daß eine Orientierung des Frameworks in Hinblick auf reale Anwendungen die Unterstützung von Anwendungsarchitektur sehr stark eingeschränkt hätte. Es wäre nur noch möglich gewesen, Unterstützung für verschiedene Bereiche in Form relativ unabhängiger „BussinesObjects“ (s.a. IBM SanFrancisco) zu geben. Da das Framework auf noch relativ unerfahrene OO – Entwickler abzielt, wurde einer stärkeren architektonischen Führung der Vorzug über reale Anwendbarkeit gegeben.

2.1.1. Applikationsverwaltung

Beginnen wir mit dem Bereich der Applikationsverwaltung. Ein wichtiger Begriff ist hier der des **Ladens** (**Shop**)[1]. Jede Anwendung auf Basis des Frameworks „SalesPoint“ stellt im Prinzip einen mehr oder weniger großen Laden dar. Umgekehrt gibt es in einer Anwendung auch nur genau einen Laden. Ein Laden hat mindesten einen, möglicherweise mehrere **Verkaufsstände** (**SalesPoint**). Ein Verkaufsstand in unserem Sinne ist dabei alles wo ein Kunde eine Interaktion mit dem Laden hat, also z.B. Kassen, Fleischtheken, Pfandautomaten, Küche, Werkstatt, usw. Die einzelnen Verkaufsstände sind also gewissermaßen die Anlaufpunkte für die Kunden.

Kunden (**Customer**) wandern durch den Laden von Verkaufsstand zu Verkaufsstand, wobei Sie ihren **Einkaufskorb** (**DataBasket**) mit sich führen. Zum Einkaufskorb wird später im Abschnitt über Datenverwaltung noch mehr zu sagen sein. Für den Moment ist die Vorstellung eines Einkaufskorbes, in den Sachen hineingetan und auch wieder herausgenommen werden können, vollkommen ausreichend.

Abbildung 1 verdeutlicht diese Begriffe noch einmal grafisch. Außerdem kann man erkennen, daß Verkaufsstände eine Warteschlange (oder Queue) verwalten, in welche sich die Kunden einreihen können. Höchstens ein Kunde aus dieser Warteschlange ist der **aktuelle Kunde** (**current customer**), also der Kunde, der gerade bedient wird. Kunden müssen nicht an einem Verkaufsstand stehen. Das gilt aber nur, solange sie noch nicht im Laden sind. Umgekehrt können Kunden sich nur dann an einem Verkaufsstand anstellen, wenn sie den Laden bereits betreten haben.

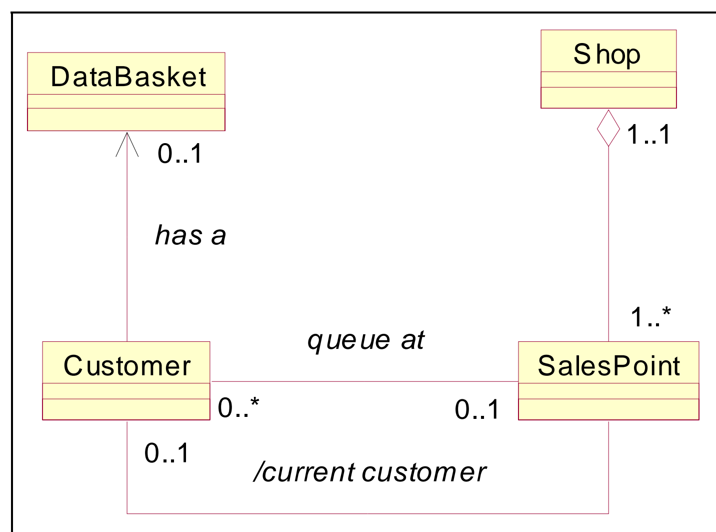

Abbildung 1 Ein Laden enthält Verkaufsstände, an denen sich Kunden mit ihren Einkaufskörben anstellen können.

Kunden agieren im Laden im Rahmen von **Prozessen** (**SaleProcess**). Ein Prozeß ist eine abwechselnde Folge von Kommunikation mit dem Kunden (z.B. ein Kunde an

[1] Die Begriffe werden in deutscher Sprache eingeführt, zusätzlich wird in Klammern die Bezeichnung der Klasse im Framework angegeben, die dieses Element implementiert. Dieser Name ist dem Englischen angelehnt und wird auch in den UML-Diagrammen verwendet werden.

der Fleischtheke wünscht 250g „Hackepeter“) und internen Bearbeitungsvorgängen (z.B. Entnahme, Abwiegen und Abpacken der gewünschten 250g „Hackepeter“). Diese wechselnden Vorgänge können als endliche Automaten interpretiert werden. Die Kommunikation mit dem Kunden findet in den **Zuständen** (**Gate**[1]) des Prozesses statt, die interne Bearbeitung wird in **Zustandsübergängen** (**Transition**) erledigt.

Abbildung 2 verdeutlicht diese Zusammenhänge nochmals grafisch.

Diese explizite Darstellung des Prozeßaufbaus im Objektmodell hat den zusätzlichen Vorteil, daß Prozesse von der Anwendung quasi „angefaßt“ werden können. Das heißt, Prozesse können jederzeit in einem definierten Zustand (nämlich an einem *Gate*) unterbrochen und später an der selben Stelle wieder fortgesetzt werden. Befindet sich der Prozeß zum Zeitpunkt der Unterbrechung gerade in einer Transition, so wird er einfach am nächsten Gate unterbrochen.[2] Damit dies aber überhaupt machbar ist, müssen ein paar Bedingungen eingehalten werden: Da das Unterbrechen von Prozessen nicht zur Systemblockade führen soll, fordert man, daß Transitionen verhältnismäßig kurz sind. Insbesondere dürfen sie keine „potentiell unendlich“ andauernden Aktivitäten umfassen. „Potentiell unendlich“ sind dabei alle Aktivitäten, deren Endtermin nicht vorherbestimmbar ist, zum Beispiel Benutzerkommunikation. Dabei könnte es vorkommen, daß der Benutzer einfach „vergißt“, einen Knopf zu wählen, was dazu führen würde, daß die Transition nicht verlassen und damit der Prozeß nicht unterbrochen werden kann. Deshalb sind Benutzerkommunikation und ähnliche langandauernde Vorgänge nur an einem Gate erlaubt. Ein Prozeß darf sich beliebig lange an einem Gate aufhalten, muß aber sicherstellen, daß er zu jedem Zeitpunkt unterbrochen werden kann. Im Gegenzug wird einem Prozeß zugesichert, daß er nie während einer Transition unterbrochen wird.

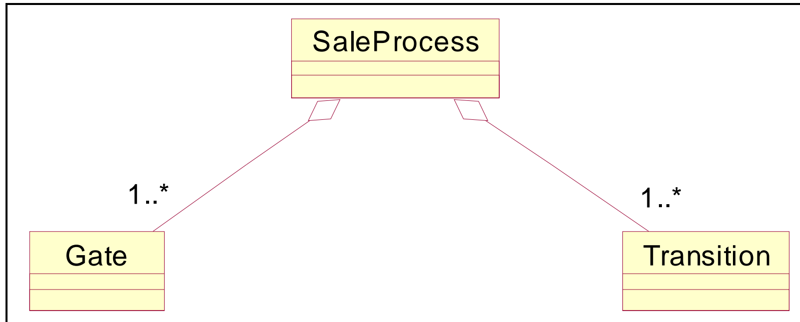

Abbildung 2 Ein Prozeß besteht aus Zuständen und Übergängen.

Prozesse können auf verschiedene Arten ausgeführt werden. Zum einen kann an jedem Verkaufsstand zu jedem Zeitpunkt höchstens ein Prozeß ablaufen. Zum anderen können Prozesse aber auch als Hintergrundprozesse wichtige Arbeit erledigen. Um den Prozeß davon abzukoppeln, ob er gerade als Hintergrundprozeß oder als Prozeß an einem Verkaufsstand abläuft, gibt es das Konzept der **Prozeß-Umgebung** (**ProcessContext**). Ein Prozeß kommuniziert nicht direkt mit dem Verkaufsstand auf dem er abläuft, oder mit dem Laden, in dem er stattfindet. Aus Prozeßsicht handelt es sich bei all dem um Prozeß-Umgebungen, über welche er die von ihm benötigten Ressourcen erhält bzw. ansteuern kann (siehe Abbildung 3).

[1] Erläuterung des vielleicht zunächst seltsamen Namens folgt.
[2] Daher der Name. Ein Gate ist etwas, das vom Prozeß „passiert“ werden muß. Gelegentlich, nämlich wenn der Prozeß unterbrochen werden soll, ist das Gate „geschlossen“.

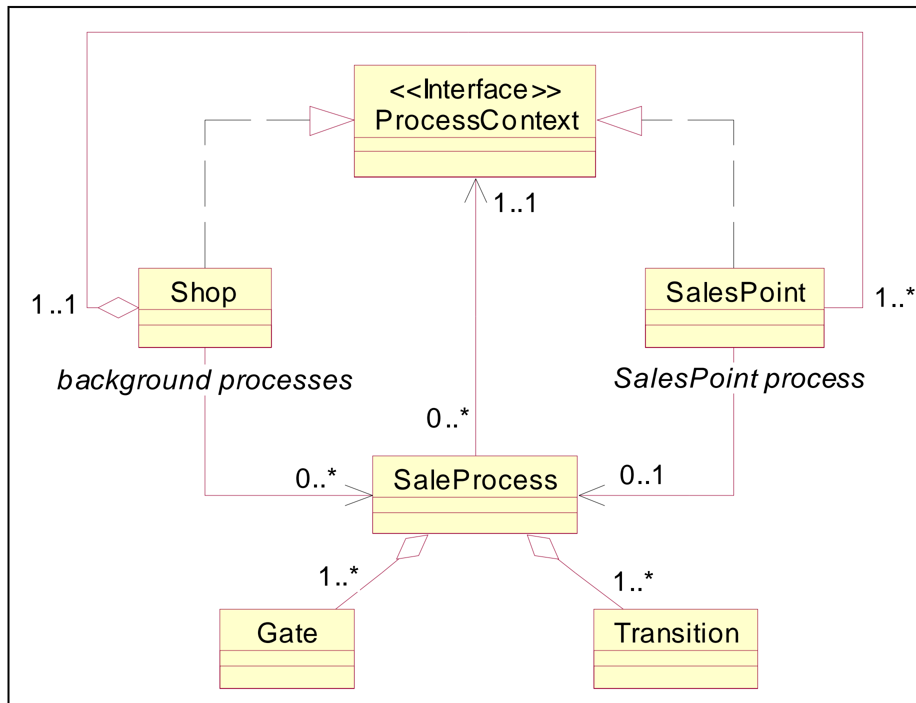

Abbildung 3 Prozesse können an Verkaufsständen und als Hintergrundprozesse laufen.

Wenn man mit dem Kunden kommunizieren will, so braucht man ein gemeinsames Medium. Im Framework gibt es dafür die **Anzeige** (**Display**). Über die Anzeige kommunizieren Kunde und Geschäft miteinander. Jeder Kunde, der den Laden betritt, bekommt eine Anzeige zugeteilt. Verläßt er den Laden wieder, so wird auch seine Anzeige vernichtet. Während der Kunde im Laden ist, „leiht“ er seine Anzeige an seine Partner (Verkaufsstände, Prozesse) aus, so daß diese darüber mit ihm kommunizieren können. Solange der Kunde nur an einer Verkaufsstelle ansteht, jedoch noch nicht der aktuelle Kunde ist, ist er selbst verantwortlich für das, was auf seiner Anzeige zu sehen ist. Er kann dort zum Beispiel ein paar Knöpfe unterbringen, die es dem Programmbenutzer gestatten, den Kunden an einen anderen Verkaufsstand zu bewegen. Sobald der Kunde zum aktuellen Kunden an einer Verkaufsstelle wird, übergibt er seine Anzeige in die Verantwortung der Verkaufsstelle. Diese kann jetzt Menüpunkte und Knöpfe anzeigen, mit denen der Benutzer Prozesse auslösen kann. Wird ein Prozeß gestartet, so erhält er die Verantwortung für den Inhalt der Anzeige. Er kann jetzt alle Formulare und Menüs anzeigen, die für den Prozeß wichtig sind.

Auf einer Anzeige können **Formulare** (**FormSheet**) und **Menüs** (**MenuSheet**) angezeigt werden. Menüs sehen aus, wie Menüs in anderen Anwendungen auch. Formulare zerfallen in zwei Bereiche: den Komponentenbereich und die Knopfleiste. Im Komponentenbereich, der gewöhnlich den oberen Teil der Anzeige einnimmt, kann jede beliebige Swing-Komponente untergebracht werden. Hier hat der Benutzer die Möglichkeit, Eingaben zu machen, aus Listen auszuwählen, etc. Gewöhnlich im unteren Bereich der Anzeige befindet sich die Knopfleiste, eine Zeile, die nur Schaltflächen enthält. Diese dienen dazu, gewisse Aktionen auszulösen. Obwohl die Formulare beliebige Swing-Komponenten enthalten können, sind Formulare und Menüs doch im wesentlichen eine Abstraktion von der konkreten Darstellungsform. Wenn man als Anwendungsentwickler ein Menü oder Formular verwendet, braucht man sich nicht darum zu kümmern, wie dieses letztendlich konkret dargestellt wird. Alles was man tun muß, ist die richtigen Angaben an der richtigen Stelle einzufügen und das ganze an die entsprechenden Stellen weiterzuleiten. Die Umsetzung in eine konkrete Oberfläche ist dann Aufgabe der Anzeige. In der im Framework enthaltenen Implementation ist diese Umsetzung die MultiWindow-Oberfläche, die jede Anzeige durch ein Unterfenster eines großen „Ladenfensters“ darstellt.

Im Rahmen des Persistenzmanagements wurde ein weiterer Begriff, der des **Formular-Inhaltserzeugers** (**FormSheetContentCreator**), eingeführt. Da die Swing-Klassen zum Teil erhebliche Schwächen bezüglich der Serialisierung aufweisen, mußte es vermieden werden, Swing-Klassen zu serialisieren. Dies wird erreicht, in dem man nur die Information speichert, die nötig ist um das GUI wieder zu regenerieren. Bei Menüs ist diese Information einfach das MenuSheet selbst (Achtung: nicht die Swing-Komponente! Das MenuSheet enthält nur die Information welche Menüknöpfe in welchem Menü zur Verfügung stehen.), bei Formularen ist es ein gesondertes Objekt, das die Strategy zur Erzeugung des Formular-Inhalts kapselt.

Formulare müssen gelegentlich mit ihrer Anzeige kommunizieren, um sicherzustellen, daß sich Änderungen am Formular auch sofort auf die Anzeige auswirken. Wenn beispielsweise ein Formular einen „Löschen“-Knopf hat, der ein gerade in einer Tabelle ausgewähltes Element löscht, so soll dieser vielleicht nicht selektierbar sein, wenn kein Eintrag in der Tabelle ausgewählt ist. Die Anwendung würde also den Zustand des Knopfes immer dann ändern, wenn sich die Auswahl in der Tabelle ändert. Um den Zustand des Knopfes zu verändern, verwendet die Anwendung eine entsprechende logische Operation, ohne sich um die Umsetzung auf der Ebene von Swing zu kümmern. Das Formular hat dann die Aufgabe, dafür zu sorgen, daß die Änderungen auch auf der

Oberfläche sichtbar werden. Um auch hier die bisherige enge Kopplung zwischen Formular und Oberfläche aufzuheben wurde der **Formular-Container** (**FormSheetContainer**) eingeführt. Er stellt ein Objekt dar, das ein Formular anzeigt. Das Formular erhält die Möglichkeit, den Formular-Container über Änderungen im Zustand des Formulars zu informieren. Der Formular-Container stellt gewissermaßen die Sicht des Formulars auf eine Anzeige dar und bietet dem Formular genau die begrenzte Schnittstelle, die es benötigt.

Mit den Knöpfen in der Knopfleiste eines Formulars oder in einem Menü werden **Aktionen** (**Action**) verknüpft, die ausgeführt werden, wenn der entsprechende Knopf gedrückt wurde.

Jedes Formular oder Menü wird immer in einem gewissen Kontext dargestellt. Dieser besteht aus dem Laden in dem es dargestellt wird, gegebenenfalls dem Verkaufsstand an dem es dargestellt wird und/oder gegebenenfalls dem Prozeß der es darstellt. Nur der Laden ist immer vorhanden. Aktionen, die mit Knöpfen eines Formulars oder Menüs verbunden sind, werden immer im Kontext dieses Formulars oder Menüs aufgerufen. Dies geschieht, in dem der entsprechenden Methode der Verkaufsstand und der Prozeß als Parameter übergeben werden. Der Laden ist ohnehin bekannt, da es nur einen pro Anwendung gibt. Abbildung 4 verdeutlicht diese Zusammenhänge noch einmal grafisch.

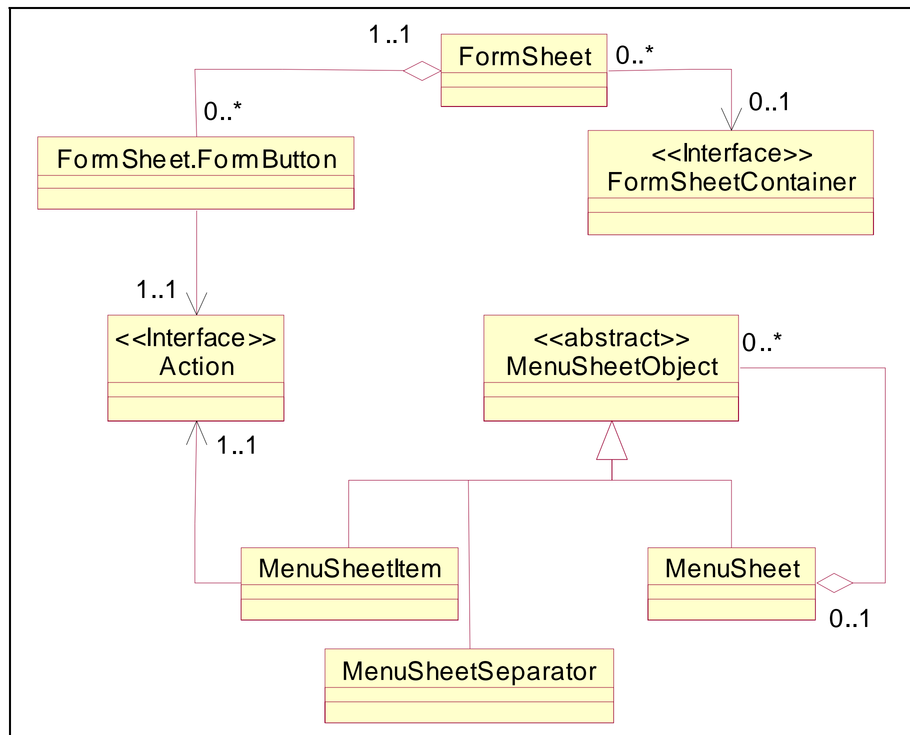

Abbildung 4 Formulare und Menüs verwenden Aktionen um Benutzereingaben mit Programmreaktionen zu verknüpfen.

Zum Abschluß dieses Abschnitts muß noch einmal der Laden erwähnt werden. Zusätzlich zu den hier geschilderten, eher konzeptionell begründeten Verantwortungen hat der Laden auch noch einige andere, eher technischer Natur. Auf abstraktester Ebene ließen sich diese etwa wie folgt motivieren: „Der Laden bildet das zentrale Element der Anwendung. Er bündelt sämtliche Aktivitäten und Verantwortungen.“ Das heißt, der Laden ist zentraler Ansprechpartner für alle Teile der Applikation. Er „kennt“, entweder durch direkte Assoziation oder weil sie ein „Singleton“ implementieren, sämtliche Elemente der Anwendung. Deshalb ist es nur natürlich, dem Laden auch die Verantwortung für das **Persistenzmanagement** zuzuordnen. Auf „Knopfdruck“, d.h. auf einen einfachen Methodenaufruf hin, kann der Laden den Zustand der gesamten Anwendung sichern, bzw. einen früher gespeicherten Zustand wiederherstellen.

2.1.2. *Datenverwaltung*

Ein weiterer, wichtiger und großer Bereich des Frameworks ist die Datenverwaltung. Diese stellt für Verkaufsanwendungen typische Datenstrukturen zur Verfügung. Außerdem stellt sie Mechanismen bereit, die zur Implementation von Transaktionseigenschaften wie Isolation und Atomizität verwendet werden können.

Ein Geschäft stellt die angebotenen Waren oder Dienstleistungen in Form eines oder mehrerer **Kataloge** (**Catalog**) dar. Ein Katalog ist im wesentlichen eine Liste von **Katalogeinträgen** (**CatalogItem**). Jeder Katalogeintrag beschreibt eine Ware oder Dienstleistung durch ihre Eigenschaften. Alle Katalogeinträge haben mindestens einen Namen, oder Schlüssel, und einen Wert. **Werte** (**Value**) sind Objekte, für die die Operationen Addition, Subtraktion, Multiplikation, Multiplikation mit Skalar und Division definiert sind. Die Menge aller Werte eines bestimmten Typs bilden mindestens ein Monoid bzgl. Addition und Multiplikation. Bzgl. der Multiplikation gibt es zusätzlich zum neutralen 1-Element noch ein 0-Element. Durch diese Objektifizierung von Werten wird erreicht, daß man immer den Typ verwenden kann, der der Anwendung am genauesten entspricht, dabei aber dennoch die Möglichkeit hat, Algorithmen, die immer wieder gleich bleiben, im voraus zu formulieren.

Kataloge stellen Listen von *potentiell* verfügbaren Objekten dar. Um *tatsächlich* verfügbare Objekte darzustellen, benötigt man **Bestände** (**Stock**). Ein Bestand bezieht sich auf einen Katalog und verzeichnet zu jedem Katalogeintrag die Anzahl tatsächlich verfügbarer Objekte diesen Typs. „Tatsächlich verfügbar“ heißt dabei nicht, daß Bestände nur verwendet werden können, um Lager- oder Kassenbestände darzustellen. Auch Bestellungen lassen sich als Bestände auffassen, wobei „tatsächlich verfügbar“ dann heißt „auf der Bestellung aufgeführt“.

Es gibt zwei grundlegende Typen von Beständen: **abzählende Bestände** (**CountingStock**) und **speichernde Bestände** (**StoringStock**). Abzählende Bestände speichern zu jedem Katalogeintrag tatsächlich nur die Anzahl verfügbarer Objekte, während speichernde Bestände für jedes tatsächlich verfügbare Objekt einen **Bestandseintrag** (**StockItem**) speichern. Dadurch ergibt sich die Möglichkeit, die einzelnen Objekte zu unterscheiden und ggf. mit individuellen Merkmalen anzureichern. Z.B. könnte es für einen Fahrzeughändler interessant sein, für

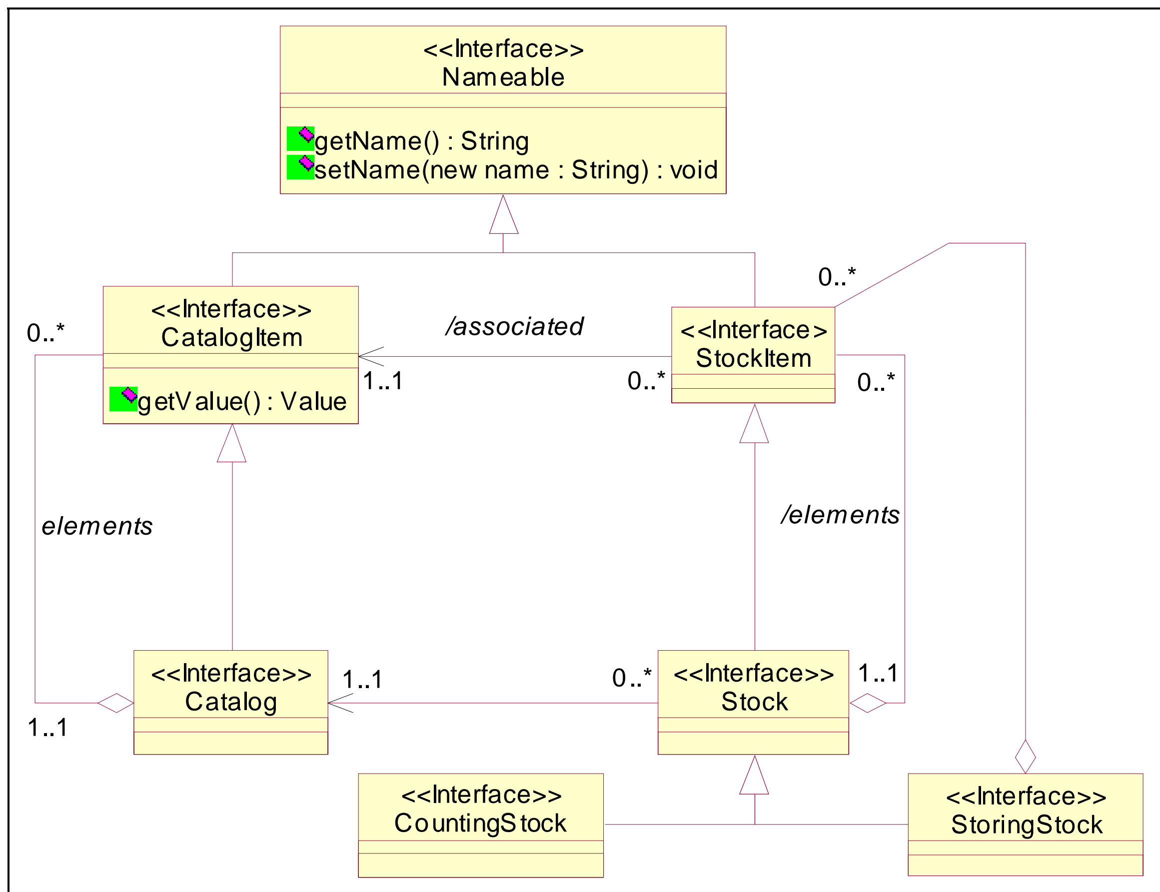

Abbildung 5 Bestände und Kataloge

jeden der 50 roten „Peugeot 406 HDI Break“, die er auf dem Hof stehen hat, zusätzlich noch die Fahrgestellnummer, die PIN der Wegfahrsperre etc. zu speichern. Dennoch sind alle 50 Autos rote „Peugeot 406 HDI Break“ und er möchte vermutlich nicht für jedes einen neuen Katalogeintrag anlegen. Deshalb kann er die zusätzlichen Informationen in einem Bestandseintrag speichern.

Abbildung 5 zeigt diese Zusammenhänge noch einmal grafisch. Sie zeigt außerdem zwei interessante zusätzliche Elemente: Kataloge bzw. Bestände sind selbst wieder Katalog- bzw. Bestandseinträge. Dadurch wird es möglich, auch mit geschachtelten Katalogen bzw. Beständen zu arbeiten. Diese können in manchen Anwendungen, die mit sehr komplexen Beständen arbeiten müssen, recht nützlich sein. Beispielsweise könnte eine Wechselstube, die Währungen zwischen denen sie wechseln kann, in einem Katalog anbieten. Die Währungen selbst sind wieder Kataloge, die die einzelnen Münzen und Banknoten beschreiben. In Analogie dazu gäbe es auch einen geschachtelten Bestand, der für jede Währung die Momentanbestände jedes einzelnen Schalters verwalten könnte. Werden geschachtelte Bestände verwandt, so müssen immer auch geschachtelte Kataloge benutzt werden. Der Katalog C, der von einem Bestand B referenziert wird, muß sich auf der gleichen Schachtelungsebene befinden, wie B. Er muß außerdem im Katalog des Bestandes enthalten sein, der B enthält. Das heißt, das Konzept unterstützt die Strukturierung von Katalogen und Beständen, jedoch nur so lange wie die Strukturierung nicht in sich selbst informationstragend ist.

Zweitens sieht man in Abbildung 5, daß die gemeinsame Eigenschaft, einen Namen zu haben, in dem Begriff des **Benennbaren** (**Nameable**) zusammengefaßt wurde. Benennbare Objekte haben einen Namen, welcher sie identifiziert. Dieser Name muß gewissen Regeln genügen. Die Regeln werden von einem **Namenskontext** (**NameContext**) festgelegt, welcher dem benennbaren Objekt zugeordnet werden kann. Vor jeder Namensänderung wird geprüft, ob der neue Name den Regeln des Namenskontextes genügt. Ist dem nicht so, so wird der Name nicht geändert.

Um Transaktionseigenschaften, wie Isolation und Atomizität, zu implementieren, existiert der **Datenkorb** (**DataBasket**). Im Abschnitt 2.1.1 „Applikationsverwaltung“ wurde er bereits unter dem Namen *Einkaufskorb* erwähnt. Datenkörbe arbeiten eng mit Datencontainern, wie Katalogen und Beständen, zusammen, um transaktionale Eigenschaften zu implementieren. Man kann sich einen Datenkorb auf zweierlei Weise vorstellen. Abstrakt ist es eine Art Transaktionshandle, also eine Struktur, die die aktuelle Transaktion kennzeichnet, und die an jede Routine, die Datencontainer manipuliert, übergeben werden muß. Etwas konkreter kann man sich Datenkörbe durchaus als eine Art Warenkorb vorstellen, in die beliebige Datenelemente hineingelegt werden können.

Datenkörbe arbeiten eng mit den entsprechenden Containern zusammen, um ihre Funktion zu erfüllen. Wenn aus einem Bestand ein Element entnommen wird, so wird es zunächst in den Datenkorb eingeordnet. Solange das Element im Datenkorb ist, wird es vom Bestand als „vorläufig gelöscht“ markiert. Erst wenn dem Datenkorb die Löschung des Elements bestätigt wurde (siehe unten), wird es endgültig aus dem Bestand gelöscht. Analog verhält es sich mit dem Hinzufügen von Elementen. Wird ein Element zum Bestand hinzugefügt, so wird es gleichzeitig in den entsprechenden Datenkorb eingefügt. Im Bestand wird es zunächst als „vorläufig hinzugefügt“ markiert und steht anderen Nutzern zunächst nicht zur Verfügung. Erst wenn das Hinzufügen dem Datenkorb bestätigt wurde, wird das Element als „tatsächlich enthalten“ markiert und wird für andere Nutzer sichtbar. Die Benutzung von Katalogen verläuft ganz analog.

Entsprechend der zwei möglichen Sichtweisen auf Datenkörbe gibt es auch zwei getrennte Untermengen der Datenkorbschnittstelle. Die eine bietet Funktionen, mit denen Transaktionseigenschaften implementiert werden können, die andere bietet Funktionen, um den Inhalt des Datenkorbs zu inspizieren und auszuwerten.

Die Transaktionsschnittstelle bietet verschiedene Variationen der Operationen *commit* und *rollback*. Diese werden verwendet um Änderungen, die im Datenkorb registriert sind, zu bestätigen und endgültig festzuschreiben (*commit*) oder um sie rückgängig zu machen (*rollback*). *Commit* bzw. *rollback* können unbedingt, also für alle vom Datenkorb beschriebenen Aktionen, oder nur für eine Auswahl dieser Aktionen durchgeführt werden. Außerdem ist es möglich, Unterbereiche des Datenkorbs zu definieren, die ähnlich wie *safe points* in Transaktionen verwendet werden können.

Die Inspektionsschnittstelle umfaßt Operationen zum Iterieren über den Inhalt des Datenkorbs sowie eine Funktion, um den Wert aller Elemente aufzusummieren, die einer bestimmten Bedingung genügen.

Außerdem können Datenkörbe gespeichert werden, was bedeutet, daß auch der Zustand von Transaktionen gesichert werden kann, so daß diese später wieder fortgesetzt werden können.

Faßt man die Aufgaben des Datenkorbs nochmals zusammen, so ergeben sich folgende Verantwortlichkeiten:

- **Transaktionsschnittstelle**: Ermöglichen von Atomizität durch *commit*- und *rollback*- Operationen.
- **Isolation**: Ermöglichen von Isolation (bis zu einem gewissen Grad (im Prinzip SQL Level 2: *read committed*)) durch Einschränkung der Sichtbarkeit von Datenelementen.
- **Inspektionsschnittstelle**: Verwendung des Datenkorbs als „Einkaufskorb“.

Bestände, Kataloge und Datenkörbe können Ereignisse auslösen, wenn ihr Inhalt geändert wird.

2.1.3. Benutzermanagement

Das Framework umfaßt auch einen kleineren Bereich, der sich mit der Verwaltung von Benutzern beschäftigt. Hier sind die folgenden Begriffe wichtig.

Benutzer (**User**) haben einen Namen, ein Paßwort sowie eine Liste von **Fähigkeiten** (**Capability**). Die Fähigkeiten bestimmen, welche Operationen ein Benutzer ausführen darf und welche ihm nicht gestattet sind. Der Name und das Paßwort dienen der Identifikation des Benutzers beim Log in.

Benutzer werden von einer **Benutzerverwaltung** (**UserManager**) verwaltet. Diese kann Benutzer aufnehmen, löschen und mit Hilfe einer **Benutzerfabrik** (**UserCreator**) sogar neue Benutzer erzeugen. Neu erzeugte Benutzer erhalten einen Satz von Fähigkeiten, die *Standardfähigkeiten* (*default capabilities*).

Kunden, wie sie bereits in 2.1.1 „Applikationsverwaltung“ eingeführt wurden, sind spezielle Benutzer im Sinne der Benutzerverwaltung.

2.1.4. Protokollverwaltung

Ein kleiner Bereich des Frameworks beschäftigt sich mit dem Erstellen und Auswerten von Protokollen.

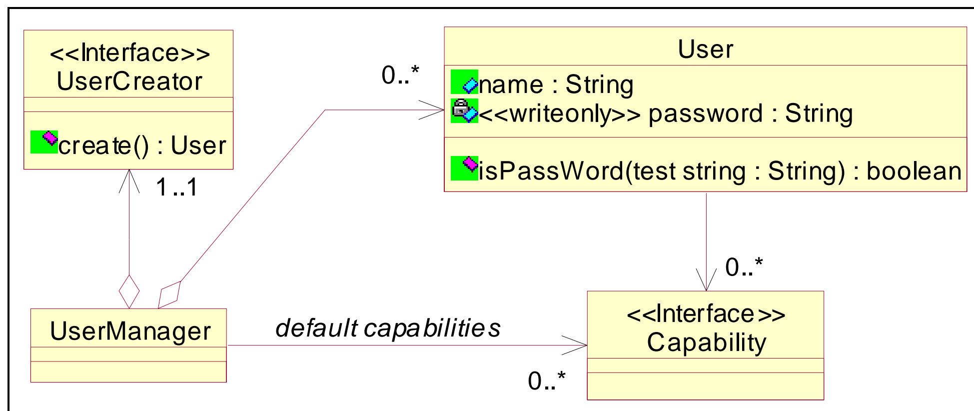

Abbildung 6 Benutzer haben Fähigkeiten und werden von Benutzermanagern verwaltet.

Protokolle (**Log**) sind Ströme von **Protokolleinträgen** (**LogEntry**). Ein Protokolleintrag verzeichnet den Zeitpunkt der Protokollierung sowie einen Text, der den zu protokollierenden Sachverhalt beschreibt. Er kann jedoch darüber hinaus noch weitere Informationen enthalten, die für den speziellen Typ von Protokolleintrag wichtig sind. **Protokollierbare Objekte oder Vorgänge** (**Loggable**) liefern auf Anfrage einen Protokolleintrag zurück, der dann ins Protokoll geschrieben wird.

Auf Protokolle besteht nur Schreibzugriff, um Protokollströme zu lesen und auszuwerten, benötigt man einen **Protokoll-Eingabestrom** (**LogInputStream**). Dieser kann mittels eines **Protokollfilters** (**LogEntryFilter**) gefiltert werden.

In zukünftigen Versionen sollte man über eine Speicherung der Protokollinformationen in einem XML – Format nachdenken.

2.1.5. GUI

Das Framework enthält bereits einige Standard-GUI-Komponenten. Neben einfachen Formularen wie **MsgForm**, **TextInputForm**, etc. gibt es auch Formulare zum Anzeigen und Editieren[1] des Inhalts von Datencontainern (siehe 2.1.2) und Protokoll-Eingabeströmen (siehe 2.1.4).

Bei den meisten dieser Formulare werden zunächst die entsprechenden Swing-Komponenten angeboten, die anschließend zu Standard-Formularen zusammengesetzt werden. Das erlaubt einerseits die flexible Verwendung dieser Komponenten in eigenen Formularen, bietet aber andererseits auch den Komfort bereits vorgefertigter Formulare für besondere Anwendungsfälle.

Ein wichtiges Element des Framework-GUI sind Tabellen. Dafür gibt es ein paar spezielle Komponenten, deren genauere Betrachtung sich an dieser Stelle lohnt. Die meisten im Rahmen des Frameworks darzustellenden Daten sind Listen von komplexen, aber gleichartigen Elementen. Um diese darzustellen, ist im Prinzip nur zu entscheiden, welche Attribute man darstellen möchte und in welcher Form. Man kann dann jedem darzustellenden Element eine Zeile und jedem darzustellenden Attribut eine Spalte einer Tabelle zuordnen.

[1] Außer bei Protokoll-Eingabeströmen

Für genau diese Art von Darstellung existieren im Framework die folgenden Konzepte. In Anlehnung an das von Swing verwendete MVC-Muster gibt es für jede Tabelle ein Modell. Die wesentlichen Eigenschaften von Tabellen, die Listen von Datensätzen verwalten, werden im **abstrakten Tabellenmodel** (**AbstractTableModel**) des Frameworks zusammengefaßt. Das abstrakte Tabellenmodel kennt eine Liste von **Datensätzen** (**Record**) und einen **Datensatzbeschreiber** (**TableEntryDescriptor**). Der Datensatzbeschreiber hat die Aufgabe, die darzustellenden Attribute auszuwählen und ihnen eine Spalte und eine Formatierung zuzuordnen. Die **abstrakte Tabelle** (**JAbstractTable**) des Frameworks bindet das ganze dann wieder zum MVC zusammen. In Abbildung 7 sind diese Zusammenhänge nochmals grafisch dargestellt. Unter diese abstrakten Klassen werden dann für jede Situation konkrete Unterklassen gehängt, die mit der konkreten Art darzustellender Daten umgehen können.

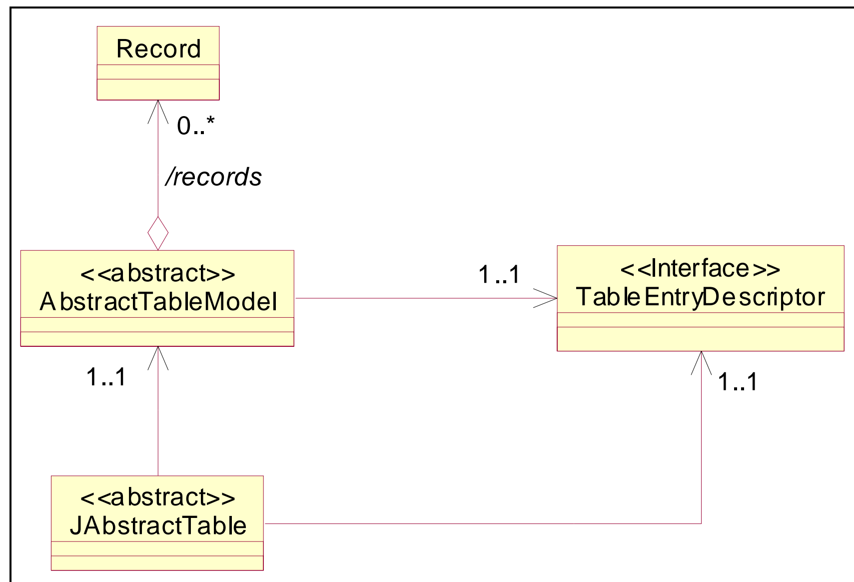

Abbildung 7 Tabellen beruhen auf einem Tabellenmodell, das eine Liste von Datensätzen mit Hilfe eines Datensatzbeschreibers auf Tabellenzeilen projiziert.

2.2. Hinweise zur Implementation

Dieser Abschnitt gibt Hinweise zur konkreten Implementation der Framework-Klassen. Zunächst werden wesentliche Unterschiede zwischen Entwurf und Implementation aufgezeigt. Anschließend werden Idiome und stilistische Regeln näher beleuchtet. Zum Abschluß des Abschnitts wird eine Übersicht über die Paketstruktur des Frameworks gegeben.

2.2.1. Unterschiede zwischen Entwurf und Implementation

Die Implementation ist im wesentlichen eine Eins-zu-eins-Umsetzung des hier vorgestellten Entwurfs. Es gibt jedoch zwei Stellen, an denen sich wichtige Abweichungen ergeben. Diese sind zum einen die Implementation der Datenhaltungsklassen und zum anderen die Implementation des Ladens als Prozeßumgebung.

Bei der Implementation der Datenhaltungsklassen wurde die eigentliche Implementation strikt von der Definition der Schnittstellen getrennt. Die gesamte Hierarchie, wie sie teilweise in Abbildung 5 dargestellt ist, wurde vollständig als Interfacehierarchie umgesetzt. Erst dann wurde eine Implementation dieser Schnittstellen erstellt und in einem gesonderten Paket abgelegt. Das Ziel dieser Technik ist es, im Rahmen der Anwendung oder der Weiterentwicklung des Frameworks die Entwicklung anderer Implementationen der Datenhaltungsklassen zu ermöglichen, unter welchen der Anwendungsentwickler dann auswählen kann. Eine alternative Implementation wäre z.B. eine Umsetzung auf der Grundlage von JDBC (siehe auch 4.1). Die im Framework enthaltene Implementation ist eine Pure Java Implementation. Sie verwaltet die Bestands- und Katalogeinträge in `Map`s und verwendet `Listener` zur Realisierung von referentieller Integrität.

Die Implementation des Ladens unterscheidet sich ebenfalls vom Entwurf. In Abbildung 3 wird der `Shop` als `ProcessContext` für Hintergrundprozesse dargestellt. In der Implementierung wurde diese Verantwortung von einem gesonderten Objekt übernommen. Der Laden ordnet jedem Hintergrundprozeß ein Handle zu, das als dessen Prozeßumgebung agiert (siehe Abbildung 8). Das hat den Vorteil, daß der Laden nicht erst bei jeder Anfrage durch einen Prozeß ein umständliches Lookup durchführen muß, um herauszufinden, welche konkreten Bedingungen für diesen bestimmten Prozeß gelten. Gleichzeitig ist `ProcessHandle` als innere Klasse implementiert, so daß sie Zugriff auf alle Funktionen von `Shop` hat. Dadurch kann sie sich konzeptionell wie ein Teil des Ladens verhalten. Der Entwurf wurde also nicht umgestoßen, es wurde nur eine effizientere Implementation benutzt.

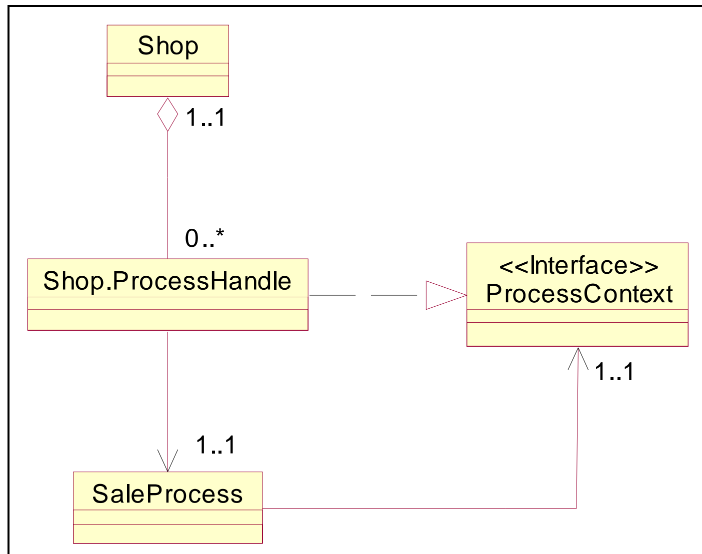

Abbildung 8 Tatsächliche Implementation des Ladens als Prozeßumgebung.

2.2.2. *Style Guide*

Für die Implementation des Frameworks wurden einige stilistische Regeln festgelegt. Diese dienen dazu, dem Quellcode ein einheitliches Erscheinungsbild zu geben und damit die Wartung und den weiteren Ausbau zu erleichtern. Zusätzlich sind einige der Regeln auch Hilfen, die gewisse Fehler bereits vor dem ersten Compilerlauf erkennen und beseitigen helfen.

An dieser Stelle soll nur ein kurzer Überblick über die wesentlichen Elemente des „Source Code Style Guide“ gegeben werden, der genaue Text kann im Anhang B „Style Guide“ nachgelesen werden.

Der „Style Guide“ enthält Angaben zu Namenskonventionen, Klammernsetzung, Verwendung von Einrückungen, Kommentaren und zu Modifikatoren. Die wichtigsten Punkte sind die Namenskonventionen sowie die Angaben zur Verwendung von Klammern und Einrückungen. Diese werden hier nochmals kurz von ihrer Zielsetzung her beleuchtet.

Die Namenskonventionen basieren im wesentlichen auf üblichen Konventionen ([SUNCode], [Lea2000]). Zusätzlich wird jedoch eine adaptierte Form der von Microsoft eingeführten Ungarischen Notation verwandt, die es erlaubt, den Typ eines Attributs oder einer Variable bereits aus dem Variablennamen zu erkennen. Die Verwendung von Kürzeln zur Kennzeichnung des Variablentyps führt zu einer Vereinfachung bei der Benennung, da sich die eigentlichen Namen nur noch auf die Semantik der Variable/des Attributs beziehen. Außerdem wird eine erste Typprüfung durch den Programmierer bereits beim Schreiben des Quellcodes ermöglicht. Bei Kollektionen, wie Listen oder Maps, kann die Typsicherheit sogar noch verstärkt werden. Diese Kollektionen sind normalerweise über `Object` definiert, was dem Compiler im wesentlichen die Möglichkeit nimmt, eine Typprüfung durchzuführen. Durch die Verwendung von Typkürzeln im Variablennamen hat der Programmierer die Möglichkeit bei jeder Verwendung einer Variable eine Typprüfung selbst durchzuführen.

Die Angaben zu Klammern und Einrückungen definieren, wie Klammern verwendet werden und wieviel Zeichen pro Blockebene eingerückt wird. Die hier verwendete Form der Einrückungen kann dem Programmierer helfen, Fehler wie das Vergessen einer schließenden Klammer zu vermeiden.

2.2.3. *Implementationsmuster*

Um die Wartbarkeit des Frameworks zu erhöhen, wurde versucht, analoge Probleme immer wieder auch analog zu lösen, so daß die Lösung im Prinzip nur einmal verstanden zu werden braucht. Bestimmte Konstruktionen tauchen an verschiedenen Stellen des Frameworks immer wieder auf. In diesem Abschnitt soll auf diese Konstruktionen hingewiesen und jeweils eine kurze Erläuterung der Struktur gegeben werden.

2.2.3.1. Monitore

An vielen Stellen des Frameworks tauchen **Monitore** auf. Monitore sind Attribute der Klasse `Object`, die den Zugriff auf andere Attribute bzw. die Ausführung von kritischen Abschnitten in Methoden synchronisieren, indem sie als Synchronisationsvariable in „`synchronized () {}`“ Blöcken eingesetzt werden.

Monitorattribute sind grundsätzlich `private transient`. Sie sind `transient`, da `Object` nicht serialisierbar ist. Sie sind `private`, damit der Zugriff grundsätzlich nur über eine Accessor-Methode möglich ist. Diese Accessor-Methoden sind immer `final` und können `public`, `private` oder `protected` sein, die Sichtbarkeit der Accessor-Methode bestimmt die Sichtbarkeit des Monitors.

Monitore werden ausschließlich über die Accessor-Methode benutzt. Dies ist wichtig, damit sichergestellt werden kann, daß zu jedem Zeitpunkt ein ordentlich initialisierter Monitor verwendet wird. Die Accessor-Methode überprüft zunächst, ob der Monitor einen von `null` verschiedenen Wert hat. Ist dies nicht der Fall, so erzeugt sie ein neues `Object` und weist es dem Monitorattribut zu. Anschließend gibt sie den Wert des Monitorattributs zurück.

Die Namensgebung von Monitoren und ihren Accessor-Methoden folgt einfachen Regeln: der Name des Monitors selbst hat die Syntax `m_o<MonitorID>Lock`. Die Accessor-Methode heißt `get<MonitorID>Lock`. `MonitorID` muß bei Monitor und dazugehöriger Accessor-Methode identisch sein.

Die Dokumentationskommentare folgen einem einheitlichen Schema: Für die Dokumentation des Monitorattributs wird folgender Kommentar verwendet: „Monitor synchronizing <MonitorZiel>.“ In Abhängigkeit vom tatsächlichen Ziel des Monitors, also ob er den Zugriff auf ein Attribut oder die Ausführung eines kritischen Abschnitts synchronisiert, wird `MonitorZiel` zu: „access to the <ZielAttribut>“ für Attributschutz oder zu: „the execution of <ZielAbschnitt>“. Der Dokumentationskommentar der Accessor-Methode enthält die Worte „Get the“ gefolgt vom Inhalt des Dokumentationskommentars des Monitorattributs.

Der folgende Code-Ausschnitt aus der Klasse `CatalogImpl.java` demonstriert das Implementationsmuster:

```
/**
  * Monitor synchronizing access to all the item maps.
  */
private transient Object m_oItemsLock;
/**
  * Get the monitor synchronizing access to all the item maps.
  *
  * @override Never
  */
protected final Object getItemsLock() {
  if (m_oItemsLock == null) {
    m_oItemsLock = new Object();
  }

  return m_oItemsLock;
}
```

2.2.3.2. Sperren des Datenkorbs in der Pure Java Implementation der Datenklassen

In der Pure Java Implementation der Datenhaltungsklassen ist es nötig, bei jeder Operation sowohl den verwendeten Datencontainer als auch den verwendeten Datenkorb zu sperren, um Thread-Sicherheit gewährleisten zu können.

Dabei gilt es zwei Bedingungen zu beachten. Erstens müssen Datenkorb und Datencontainer immer in der gleichen Reihenfolge gesperrt werden, um Dead-Locks zu vermeiden. Zweitens kann es vorkommen, daß statt eines Datenkorbs `null` übergeben wird, um zu signalisieren, daß die Operation einen unmittelbaren Effekt haben soll.

Da der Datenkorb bei *commit* und *rollback* Aktionen ohnehin zunächst sich selbst und anschließend die gerade bearbeiteten Kataloge und Bestände sperrt, wird diese Reihenfolge auch in den Operationen der Bestände beibehalten. Das zweite Problem wird durch das Erzeugen eines Dummy-Monitors gelöst, falls ein `null` Datenkorb übergeben wurde.

Das Implementationsmuster stellt sich im Code wie folgt dar:

```
public void operation (parameter..., DataBasket db) {
  Object oLock = ((db != null)?(db.getDBIMonitor()):(new Object()));

  synchronized (oLock) {
    synchronized (getItemsLock()) {
      // ... Code, der synchronisiert werden muß
    }
  }
}
```

2.2.3.3. `Listener` und Adapter

Viele Klassen des Frameworks feuern Ereignisse, um über Änderungen ihres Zustandes zu informieren. Diese Ereignisse werden von Implementationen spezieller Interfaces, sogenannten `Listeners` empfangen. Diese `Listeners` definieren für jeden Typ von Ereignis eine Methode, die bei Auftreten dieses Ereignisses aufgerufen wird. Gewöhnlich enthält ein `Listener` mehrere solche Methoden, jedoch kommt es häufig vor, daß ein Client eigentlich nur eine Methode definieren möchte. Deshalb werden für jedes `Listener` Interface mit mehr als einer Methode sogenannte Adapter definiert, die für jede Methode einen leeren Methodenkörper anbieten. Diese Adaptoren sind abstrakte, von `Object` abgeleitete Klassen, die sowohl das konkrete `Listener` Interface als auch das Interface `util.SerializableListener` implementieren. Die `Listener` Interfaces selbst erben nie von `SerializableListener`.

Zu jedem solchen Ereignis gibt es also ein Interface und bis zu zwei Klassen: den `Listener` selbst, das Ereignisobjekt, das als einziger Parameter an jede Methode im `Listener` übergeben wird und ggf. einen Adapter. Die Namen dieser drei Klassen haben einen gemeinsamen Stamm, der die Ereignisklasse beschreibt und an den die folgenden Endungen angehängt werden: „`Listener`“ für das `Listener`-Interface, „`Event`“ für das Ereignisobjekt und „`Adapter`“ für den Adapter.

Klassen, die Ereignisse feuern können, verwalten intern eine Liste aller momentan registrierten `Listener`. Diese Liste sollte von `util.ListenerHelper` abgeleitet sein. Damit wird sichergestellt, daß `Listener`, die nicht `SerializableListener` implementieren (insbesondere Swing-Klassen), nicht mit serialisiert werden. Wenn das die Ereignisse auslösende Objekt selbst Ereignisse empfängt, so ist es evtl. sinnvoll, `HelpableListener` zu implementieren und dem `ListenerHelper` eine Instanz des Objektes mitzugeben. Dadurch wird erreicht, daß das Objekt nur dann auf seine Quellen „hört“, wenn es auch `Listeners` gibt, die sich für das Objekt interessieren. Auf diese Weise können zyklische Verweise zwischen Objekt und Quelle aufgebrochen werden, wodurch ein besseres Garbage Collecting ermöglicht wird.

2.2.3.4. Partner und Kontexte

Einige Objekte im Framework erhalten einen Kontext oder einen Partner erst zu Lebzeiten zugeteilt. Dieser Partner kann ihnen gewöhnlich auch wieder weggenommen werden. Die Objekte haben aber zu jedem Zeitpunkt höchstens einen solchen Partner. Beispiele für solche Beziehungen sind Kunde – Datenkorb, Prozeß – Datenkorb, Verkaufsstand – Anzeige, etc.

Für die Methoden, die für das Zuteilen bzw. Wegnehmen des Partners zuständig sind, gibt es ein konsistentes Namensschema.

Die Methoden zum Zuteilen eines Partners heißen immer `attach`, sie unterscheiden sich nur durch Rückgabewert und Parametertyp. Diese Methoden haben nur einen Parameter, der Typ ist die Klasse des Partners. Der Rückgabewert hat ebenfalls die Klasse des Partners, zurückgeliefert wird der vorhergehende Partner. Der Dokumentationskommentar zu dieser Methode muß angeben, falls es illegal ist, wenn die Klasse keinen Partner hat.

Wenn es legal ist, daß die Klasse keinen Partner hat, so hat sie auch eine `detachXXX` Methode, um die Verbindung zum momentanen Partner zu entfernen. Der Name der Methode ergibt sich aus

```
detach<Klasse des Partners>
```

Die Methode hat keinen Parameter, ihr Rückgabewert hat die Klasse des Partners und bezeichnet den soeben entfernten Partner. Gewöhnlich wird diese Methode implementiert als:

```
return attach ((<Klasse des Partners>) null);
```

2.2.3.5. Standardformulare

Bei der Implementierung von Standardformularen wurde das „Strategy“-Muster verwendet, um das Anknüpfen anwendungsspezifischen Verhaltens an das Formular zu ermöglichen. Dabei gibt es zwei Punkte zu beachten:

Alle diese Strategien haben eine gemeinsame Wurzel in der Klassenhierarchie des Frameworks: `sale.stdforms.FormSheetStrategy`. Diese sorgt dafür, daß die Behandlung von Fehlern in jeder Strategie identisch durchgeführt werden kann. Sie definiert eine Methode `error`, deren Methodenkörper durch das Setzen einer weiteren Strategie (`sale.stdforms.FormSheetStrategy.ErrorHandler`) beeinflußt werden kann.

Zweitens definieren alle diese Strategien Unterprozesse des Prozesses, innerhalb dessen das Formular dargestellt wird. Das erzwingt zwar einerseits die Verwendung des Formulars innerhalb eines Prozesses, stellt aber andererseits Eindeutigkeit für die Strategie her. Die Strategie kann selbst entscheiden, welche Teile der durch sie definierten Operation an einem Gate oder in einer Transition ausgeführt werden sollen. Würde die Strategie nur die Operation bereitstellen, ohne den Prozeßrahmen zu definieren, so könnte die gesamte Operation nur entweder an einem Gate oder in einer Transition stattfinden.

Standardformulare verwenden weiterhin grundsätzlich `FormSheetContentCreator` zum Erzeugen des Formularinhalts. Dadurch wird sichergestellt, daß die Formulare immer serialisierbar sind.

2.2.3.6. Inhalte von Swing-Komponenten auswerten

Swing-Komponenten haben eine Menge von Problemen im Zusammenhang mit der Serialisierung. Deshalb wurde im gesamten Framework darauf Wert gelegt, daß Komponenten, die serialisiert werden, keine Referenzen auf Swing-Komponenten enthalten. In diesem Zusammenhang entstand auch das Konzept des `FormSheetContentCreator` (siehe 2.1.1).

Um den Inhalt von Swing-Komponenten zu bestimmen, ohne eine direkte Referenz auf diese Komponenten halten zu müssen, werden `Listener` verwendet, die auf jede Änderung des Inhalts einer Komponente reagieren und entsprechende Informationen in eine vorbereitete Variable schreiben. Beispiele dafür finden sich in den Klassen `SingleTableFormSheet` und `JTextInput`.

2.2.4. Paketstruktur

Entsprechend der logischen Struktur (siehe 2.1) zerfällt die Implementation des Frameworks in mehrere Pakete. Die Klassen zur Applikationsverwaltung (siehe 2.1.1) finden sich im Paket `sale`. Die Klassen für die Datenverwaltung (siehe 2.1.2) finden sich im Paket `data`. Das Benutzermanagement (siehe 2.1.3) wurde im Paket `users` implementiert, die Protokollverwaltung (siehe 2.1.4) im Paket `log`. Das Paket `util` enthält allgemein nützliche Klassen sowie – in `util.swing` – GUI (siehe 2.1.5) Basisklassen, die sich keinem der anderen Pakete sinnvoll zuordnen ließen.

Innerhalb der Pakete wurde versucht, die Benennung der Unterpakete einheitlich zu gestalten. Die folgende Tabelle gibt einen Überblick über immer wieder auftauchende Namen und die in diesen Paketen zu erwartenden Klassen:

Paketname	Enthaltene Klassen
`events`	`Listener` und Ereignisobjekte mit Bezug zu den Klassen im enthaltenden Paket.
`swing`	Swing-Komponenten zur Darstellung/Bearbeitung von Aspekten der Klassen des enthaltenden Pakets.
`stdforms`	Standardformulare, die meist Komponenten aus dem entsprechenden `swing` Paket kombinieren.

Einige Pakete definieren noch weitere Unterpakete. `sale` definiert das Unterpaket `multiwindow`, das die Klassen zur Verwaltung der `MultiWindows` der Standard-Applikations-Oberfläche enthält. `data` enthält die Unterpakete `filters` – mit Bestands- und Katalogfiltern – und `ooimpl` mit den Klassen der Pure Java Implementation der Datencontainer. `data.stdforms` enthält noch zwei weitere Hilfspakete: `singletableformsheet` und `twotableformsheet`. In diesen finden sich die Strategien für die gleichnamigen Standardformulare.

2.3. Entwurfsmuster

Im Framework werden einige der in [GHJV94] vorgestellten Entwurfsmuster angewendet. Eine Auswahl der wichtigsten soll hier aufgezeigt werden. Das „Template Method“ Muster wird dabei nicht beachtet, da es im Rahmen eines Frameworks praktisch ständig verwendet wird. Die graphische Notation lehnt sich an die in [Gamm95] vorgeschlagene an. Allerdings werden die Namen der jeweiligen Muster nicht im Diagramm dargestellt.

2.3.1. Factory Method

„Factory Method“ wird im Framework zu zwei Zwecken verwendet. Einmal, zusammen mit dem „Strategy“ Muster, um die Erzeugung von Protokollen zu kontrollieren. Zum anderen wird das Muster in Beständen und Katalogen verwendet, um den Klon-Algorithmus von der Bestimmung der konkreten Unterklasse zu trennen.

Die Klasse `Log`, die Protokolle implementiert, hat eine Methode `createLog`, die ein neues Protokoll erzeugt. Um diese statische Methode an neue Unterklassen von `Log` anpassen zu können, gleichzeitig aber nicht den Vorteil einer statischen Methode einzubüßen, daß sie nämlich ohne Kenntnis der konkreten Unterklasse angewandt werden kann[1], definiert das Framework eine Klasse `LogCreator`, die als „Strategy“ zum Erzeugen des neuen `Log`s dient. Diese definiert die „Factory Method“ `createLog`.

Außerdem wird das Muster auch in Katalogen und Beständen angewandt. Die Pure Java Implementationen von Beständen und Katalogen müssen die Fähigkeit haben, Klons zu erzeugen. Das Füllen dieser Klons läßt sich auf sehr abstrakter Ebene definieren, Es handelt sich einfach darum, die Elemente aus dem Originalcontainer in den neu erzeugten zu kopieren. Was sich jedoch auf so abstrakter Ebene nicht so leicht definieren läßt, ist die konkrete Klasse des zu erzeugenden Klons. Da dem Anwendungsentwickler die Möglichkeit gegeben werden soll, falls nötig Unterklassen von den Bestands- bzw. Katalogimplementationen zu definieren, muß auch die Klasse des Klons parametrisierbar sein. Dazu definieren diese Implementationen eine Methode `createPeer`, die einen neuen, leeren Bestand/Katalog von der selben Klasse wie der Empfänger der Nachricht erzeugt und zurückliefert. Die einzelnen Elemente des Musters sind nochmals in Abbildung 9 dargestellt.

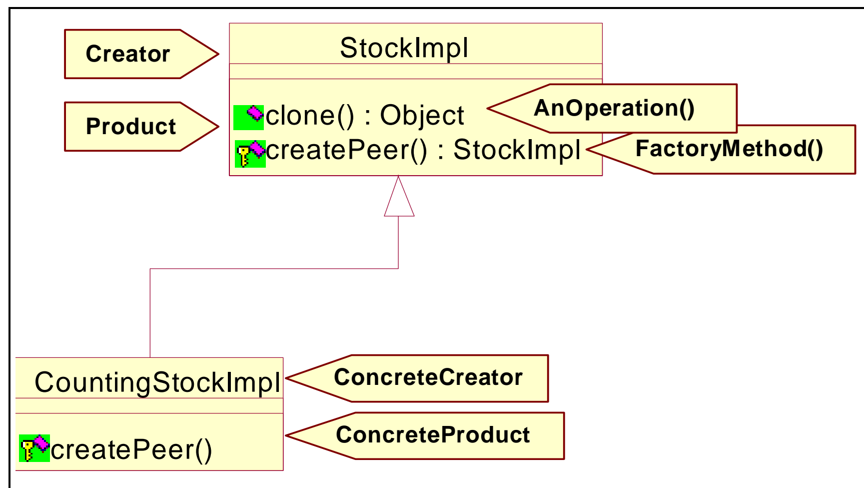

Abbildung 9 Anwendung des „Factory Method“ Musters in der Implementation von Beständen.

2.3.2. Singleton

„Singleton“ wird im Framework eingesetzt, um sicherzustellen, daß es immer genau einen Laden gibt. Der Konstruktor der `Shop` Klasse ist `protected`, und die Klasse enthält eine Methode `getTheShop` mit welcher man auf die einzige Instanz dieser Klasse zugreifen kann. Außerdem enthält die Klasse eine Methode `setTheShop`, so daß Instanzen von Unterklassen von `Shop` den Platz der einzigen `Shop`-Instanz einnehmen können. Abbildung 10 verdeutlicht die Verwendung des Musters nochmals grafisch.

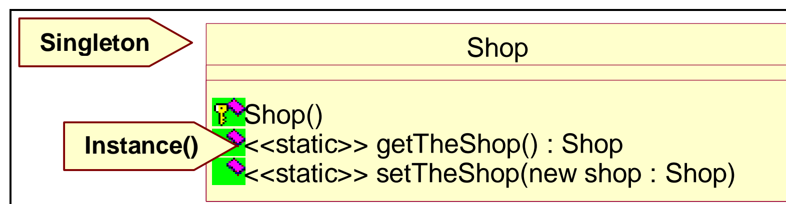

Abbildung 10 Anwendung des „Singleton“ Musters bei der Implementation des Ladens.

[1] Dies ist auch eine etwas verkrüppelte Anwendung von „Abstract Factory“

2.3.3. Adapter

Knöpfe in der Knopfleiste von Formularen können mit Aktionen verknüpft werden. Gern würde man auch Knöpfe, die sich im Komponententeil des Formulars befinden, mit Aktionen verknüpfen. Da diese Knöpfe aber als echte Swing-Komponenten behandelt werden müssen, bieten sie diese Möglichkeit nicht. Sie lassen sich nur mit einem `ActionListener` verknüpfen. Für solche Fälle bietet das Framework einen Adapter, `ActionActionListener`, der die `ActionListener` Schnittstelle in die `Action` Schnittstelle des Frameworks übersetzt. Der Adapter ist so gestaltet, daß er je nach Bedarf als „class adapter“ oder als „object adapter“ verwendet werden kann.

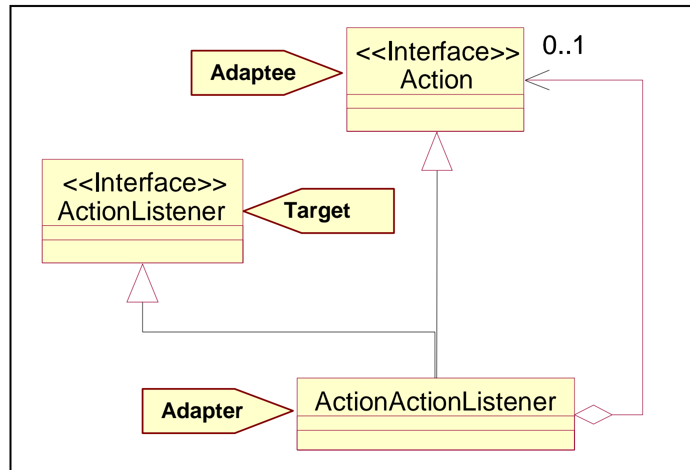

Abbildung 11 `ActionActionListener` kann sowohl als „class adapter“ als auch als „object adapter“ verwandt werden.

2.3.4. Bridge

Das „Bridge“ Muster wird im Framework eingesetzt, um die Zeitverwaltung zu implementieren. Die Zeitverwaltung kennt **Zeitgeber** (**Timer**) welche die Zeit verwalten. Sie kennen solche Operationen wie „Zeit lesen“, „Zeit ein Intervall weiterschalten“ etc. Es sind sowohl verschiedene Formen von Zeitgebern (explizite Auslösung, Auslösung durch Hintergrund-Thread in konstanten Intervallen, etc.) als auch verschiedene Formen von Zeitdarstellungen (Datum, einfache Zahl, künstliche Benennung der Zeitpunkte – etwa ‚Montag‘, ‚Dienstag‘, …) vorstellbar. Deshalb wurde der Begriff der **Zeit** (**Time**) eingeführt und die beiden durch eine „Bridge“ verbunden. Das gesamte Muster (und gleichzeitig die gesamte Klassenhierarchie der Zeitverwaltung) findet sich in Abbildung 12.

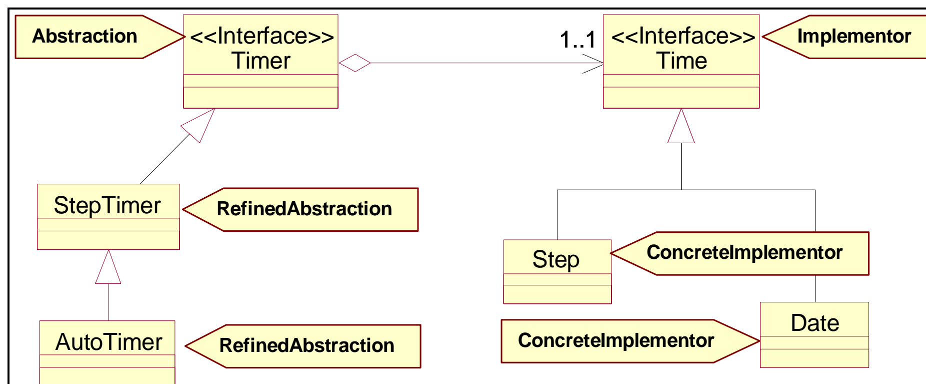

Abbildung 12 Das „Bridge“ Muster, angewendet auf die Zeitverwaltung im Framework.

2.3.5. Composite

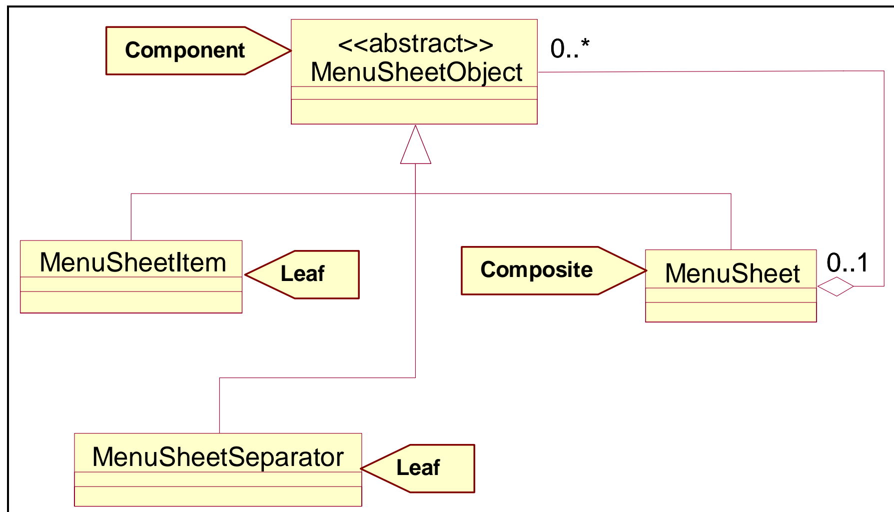

Abbildung 13 Das „Composite“ Muster angewandt auf Menüs.

Das „Composite“ Muster wird im Framework sowohl bei den Datenhaltungsklassen, als auch bei Menüs eingesetzt. Diese Beschreibung beschränkt sich auf den Einsatz bei Menüs, der Einsatz bei den Interfaces der Datenhaltungsklassen wurde bereits in 2.1.2 „Datenverwaltung„ angedeutet.

Menüs enthalten sowohl Untermenüs als auch einzelne Menüpunkte. Um diese auf einheitliche Art und Weise anzusprechen, bietet sich der Einsatz des „Composite“ Patterns an. Die abstrakte Komponente ist dabei `MenuSheetComponent`, das Komposit `MenuSheet` und die Blätter sind `MenuSheetItem` und `MenuSheetSeparator`.

2.3.6. Decorator

„Decorator“ wird verwendet, um Kataloge und Bestände zu filtern. Statt eine Funktion aufzurufen, die den übergebenen Bestand oder Katalog filtert und einen neuen Bestand oder Katalog zurückliefert, „dekoriert“ man einfach den gewünschten Bestand oder Katalog mit einem entsprechenden Filter. Die abstrakte Implementation dieser Filter existiert bereits, konkrete Filter müssen nur noch eine Methode überschreiben, die die Filterbedingung beschreibt.

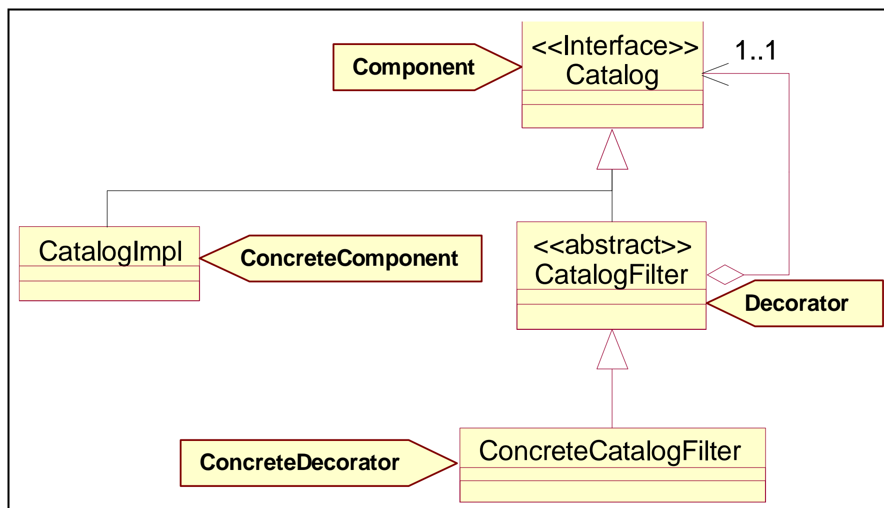

Abbildung 14 Das „Decorator“ Muster angewandt auf Katalogfilter. Die Klasse `ConcreteCatalogFilter` muß vom Anwendungsentwickler bereitgestellt werden.

Im Gegensatz zum originalen „Decorator“ Muster, lassen sich die so erzeugten gefilterten Bestände bzw. Kataloge allerdings nicht wieder als Teil eines anderen Bestands bzw. Katalogs verwenden. Dies hängt damit zusammen, daß zumindest die Pure Java Implementation von Beständen und Katalogen sich bei der Implementation von referentieller Integrität auf Objektidentität verläßt. Wenn jetzt ein „dekorierter“ Bestand verwendet wird, kann es vorkommen, daß dieser nicht korrekt erkannt wird (es ist ja tatsächlich ein anderes Objekt), was zu Problemen mit der referentiellen Integrität führen kann.

2.3.7. Proxy

Benutzer, wie sie in der Benutzerverwaltung (siehe 2.1.3) definiert sind, haben gewisse Fähigkeiten oder Rechte. Zu diesen kann es gehören, daß ein Benutzer gewisse Operationen ausführen darf, während ein anderer diese Operationen nicht ausführen darf. Operationen, die mit Menüpunkten bzw. Knöpfen in der Knopfleiste von Formularen verknüpft sind, werden durch Aktionen dargestellt („Command“ Muster, siehe 2.3.8). Solche Aktionen können nun mit Hilfe des „Proxy“ Musters um Rechte erweitert werden. Das Recht (`ActionCapability`, siehe Abbildung 15) „bewacht“ quasi die eigentliche Aktion.

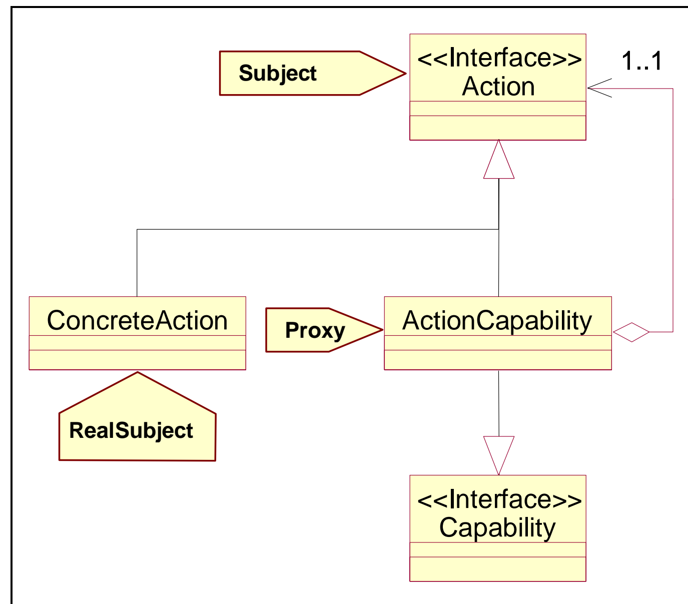

Abbildung 15 Eine `Capability` „bewacht“ eine `Action`.

2.3.8. Command

Menüknöpfe und Knöpfe in der Knopfleiste eines Formulars sind mit Aktionen verknüpft, die ausgeführt werden, wenn der entsprechende Knopf gedrückt wird. Dies ist eine Anwendung des „Command“ Musters. Auf eine grafische Darstellung wird wegen der extremen Einfachheit verzichtet.

2.3.9. Observer

„Observer“ ist wohl das am häufigsten im Framework verwendete Muster. Viele Klassen, darunter `Catalog`, `Stock`, `DataBasket`, `SalesPoint` und `Display` verschicken Ereignisse, wenn sich ihr Zustand ändert.

Dieses Muster wird hauptsächlich verwendet, um zu enge Kopplung zwischen Klassen zu vermeiden und um dem Anwendungsentwickler mehr Zugangsmöglichkeiten zu inneren Abläufen im Framework zu geben.

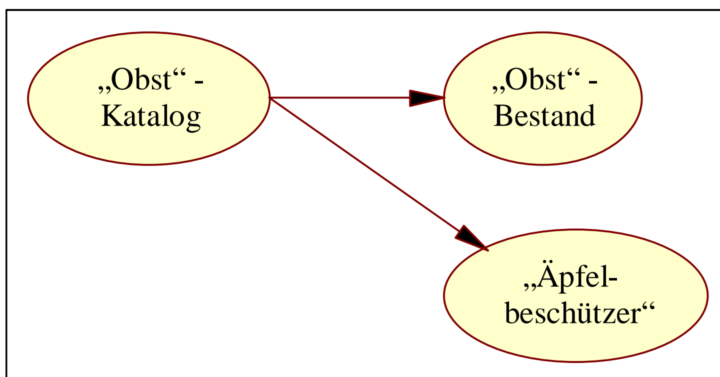

Abbildung 16 Ein Katalog wird von zwei Observern beobachtet.

Im Zusammenhang mit Beständen und Katalogen wird ein erweitertes „Observer“ Muster verwendet, um referentielle Integrität zu sichern. Observer von Beständen und Katalogen haben die Möglichkeit das Entfernen und Bearbeiten von Einträgen zu verhindern. Dazu wird vor dem Ausführen der Operation eine Nachricht an alle momentan registrierten Observer gesandt, die ggf. mit einer `VetoException` beantwortet wird. Hat ein Observer eine solche `VetoException` geworfen, so werden alle Observer, die bereits gefragt wurden, davon informiert, daß die Operation nun doch nicht durchgeführt wird. Anschließend wird die Exception aus der Methode in `Catalog` oder `Stock` propagiert, um den Aufrufer vom Abbruch zu informieren. Dieses Vorgehen sichert, daß Observer, die ihren Zustand änderten, um kein Veto einlegen zu müssen, den alten Zustand wieder herstellen können, falls die Operation dennoch abgebrochen wurde. Ein Beispiel wird die Notwendigkeit dieses Vorgehens näher erläutern: Nehmen wir an, in einem Katalog „Obst“ gibt es Einträge für „Apfel“ und „Birne“. In einem damit verbundenen Bestand liegt die Information vor, daß 7 Äpfel und 9 Birnen vorhanden sind. Außerdem gibt es im Rahmen der Anwendung die Bedingung, daß Äpfel, die als Grundnahrungsmittel betrachtet werden, nie aus dem Angebot entfernt werden dürfen. Die Objektreferenzen können Abbildung 16 entnommen werden. Die Ellipsen stehen für konkrete Objekte, die Pfeile bezeichnen den Fluß von Ereignissen. Wird nun versucht den Katalogeintrag „Birne“ aus dem Katalog zu löschen, so sendet dieser zunächst eine Anfrage an den Bestand sowie an den „Äpfelbeschützer“. Der Bestand stellt fest, daß er zwar 9 Birnen enthält, er kann die referentielle Integrität aber aufrecht erhalten, indem er diese löscht. Der „Äpfelbeschützer“ interessiert sich überhaupt nicht für Birnen und ignoriert das Ereignis. Also kann der Katalogeintrag gelöscht werden. Nun soll der Katalogeintrag „Apfel“ gelöscht werden. Wiederum wird eine Anfrage ausgelöst und wieder löscht der Bestand seine 7 Äpfel, um referentielle Integrität zu gewährleisten. Der „Äpfelbe-

schützer“ legt nun aber sein Veto ein. Würde jetzt die Löschoperation einfach abgebrochen, wären die 7 Äpfel gelöscht worden, obwohl der entsprechende Katalogeintrag gar nicht entfernt wurde. Deshalb müssen zunächst alle, die schon um Erlaubnis gefragt wurden, davon informiert werden, daß die Löschoperation nun doch nicht stattfindet. Wenn der Bestand dieses Ereignis erhält, stellt er die 7 Äpfel wieder her.

2.3.10. Strategy

„Strategy“ wird an vielen Punkten im Framework verwendet. Viele wichtige Anwendungen findet man im Bereich der Datenklassen, so das Bestimmen des Wertes eines Katalogeintrags (→ `CatalogItemValue`) und das Stückeln von Werten (→ `StockFromValueCreator`).

In beiden Fällen gibt es eine Menge vorstellbarer Algorithmen, so daß eine festverdrahtete Lösung nicht wünschenswert erscheint. Der Wert eines Katalogeintrags kann sein Standardwert sein, wie er durch die `getValue` Methode zurückgeliefert wird, man kann sich aber auch vorstellen, daß unter bestimmten Bedingungen der Wert aus dem Eintrag erst berechnet werden muß. Gleichzeitig gibt es aber Algorithmen, die bis auf die Bestimmung des Wertes eines Katalogeintrags immer gleich sind, wie z.B. das Bestimmen des Wertes eines Bestandes. Hierzu muß man nur für jeden Eintrag des entsprechenden Katalogs den Wert bestimmen, diesen mit der Anzahl entsprechender Bestandseinträge multiplizieren und das Ergebnis aufaddieren.

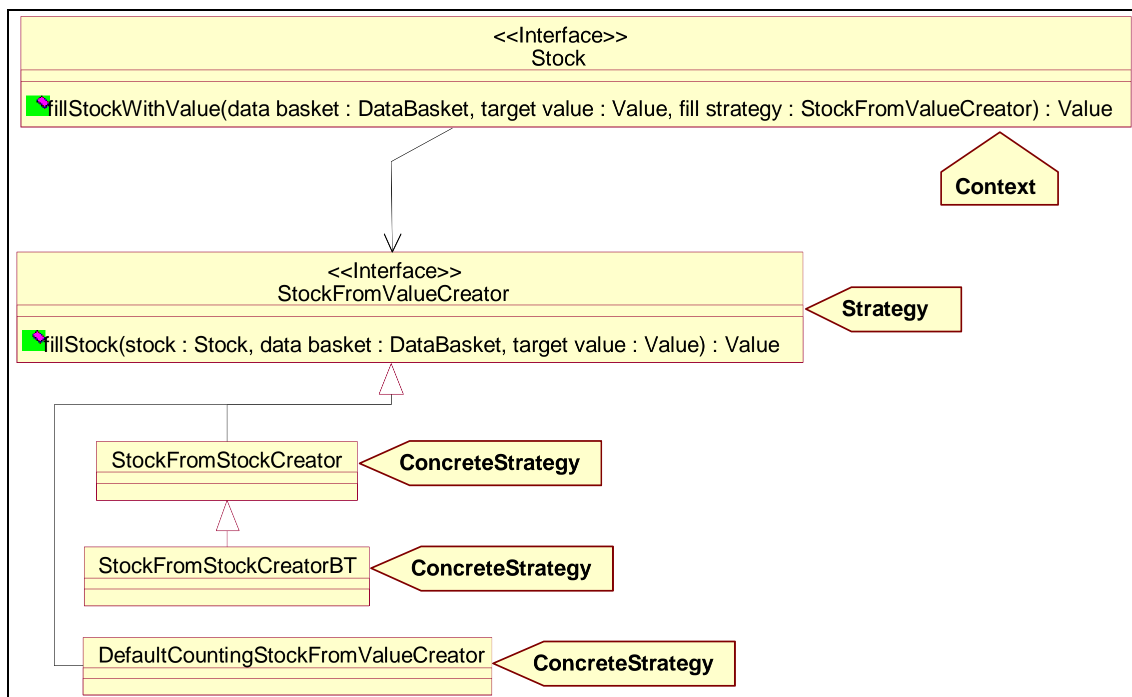

Abbildung 17 „Strategy“ Muster zum Stückeln von Werten und Auffüllen von Beständen.

Ähnliches gilt für das Stückeln von Werten. Während es eine Unzahl vorstellbarer Algorithmen gibt (allein im Framework sind bereits drei verschiedene implementiert), gibt es einige Elemente, die immer wieder gleich sind. So möchte beispielsweise der Zielbestand seinen Inhalt auf geeignete Weise sperren, so daß die Stückelung als atomare Operation ablaufen kann. Deshalb bietet sich auch in diesem Fall die Verwendung von „Strategy“ an. Dieser Fall wird in Abbildung 17 dargestellt.

Ein anderer wichtiger Bereich in dem „Strategy“ im Framework verwendet wird, sind die Standardformulare. `SingleTableFormSheet` und `TwoTableFormSheet` bieten eine relativ komplexe Benutzerschnittstelle zum Visualisieren und Editieren von Katalogen und Beständen. Das Editieren erfolgt, indem der Benutzer Einträge auswählt und eine Taste drückt. Dieses Verhalten ist standardisiert. Die Reaktion der Anwendung auf den Tastendruck des Benutzers ist jedoch anwendungsabhängig und wird daher als Strategy implementiert. Beim Erzeugen des Formulars kann die Anwendung eine Strategy übergeben, die definiert, wie das Formular auf Tastendruck durch den Benutzer reagieren soll. Aus Ergonomiegründen existieren jedoch auch Standardimplementationen für die meisten Fälle.

Die hier verwendeten Strategien sind sogar „Doppelstrategien“: Im Rahmen der Reaktion auf eine gedrückte Taste kann es zu Fehlern kommen. Die Fehlerbehandlung wird nun durch eine andere Strategy, den `ErrorHandler` übernommen.

3. Verwandte Arbeiten

Dieser Abschnitt versucht das Framework in die Landschaft bereits existierender Frameworks einzuordnen bzw. anhand existierender Artikel zu charakterisieren. Er erhebt dabei keinerlei Anspruch auf Vollständigkeit, sondern ist mehr als ein weiteres Mittel zur Charakterisierung des Frameworks gedacht.

Leider gibt es nur sehr wenig Literatur zum Thema Frameworks, die über die Grundlagen von OOP und die Aussage, daß Frameworks „wichtig, aber schwierig“ sind, hinausgehen. Daher soll das Framework kurz anhand von von Taligent Inc. in [TaliOOB] eingeführter Begriffe charakterisiert und mit einem anderen Framework (IBM San Francisco) ähnlicher thematischer Richtung verglichen werden.

3.1. Taligent: Building Object-Oriented Frameworks

In [TaliOOB] beschreiben Entwickler von Taligent Inc. Vorgehensweisen zur Erstellung objektorientierter Frameworks. Sie definieren dabei auch verschiedene Kategorisierungsmerkmale für Frameworks. In diesem Abschnitt soll versucht werden, diese Merkmale auf das SalesPoint Framework anzuwenden.

Taligent charakterisieren Frameworks in zwei Dimensionen: Anwendungsdomäne (problem domain) und interne Struktur (internal framework structure).

Bezüglich Anwendungsdomäne des Frameworks werden drei Kategorien unterschieden:

1. **Application frameworks** sind Frameworks, die allgemeine, aufgabenübergreifende Anwendungsfunktionalität bieten. Sie stellen eine horizontale Schicht der Anwendung zur Verfügung, welche für verschiedene Problemstellungen nützlich ist. Beispiele sind MFC, Borland OWL und andere GUI Frameworks.
2. **Domain frameworks** sind Frameworks, die problemspezifische Funktionalität bereitstellen. Sie stellen eine vertikale Schicht der Anwendung dar. Beispiele umfassen das JDK 1.2 Collection API, Multimedia Frameworks, etc.
3. **Support frameworks** stellen Systemdienste zur Verfügung, wie etwa Dateisysteme, Netzwerkzugriff u.ä. Sie werden im allgemeinen direkt und ohne Anpassung verwendet, es sei denn, um neue Dateisysteme o.ä. zu entwickeln.

Entsprechend der angebotenen Funktionalität läßt sich das Framework als Domain Framework einordnen. Es bietet zwar Anwendungsstruktur (d.h. eher eine Charakteristik des Application Frameworks) durch die Shop/SalesPoint Architektur, diese ist jedoch immer noch sehr stark auf die konkrete Domäne Verkaufsanwendung hin orientiert. Alle anderen Dienste (Datenhaltung, Transaktionsmanagement…) sind jedoch eindeutig als domänenspezifisch zu beurteilen.

Eine zweite Betrachtungsdimension bei Taligent ist die interne Struktur des Frameworks. Hier werden folgende Charakterisierungen verwendet (2./3. sind die zwei Pole eines Merkmals, 1. stellt ein zusätzliches Merkmal dar):

1. **Manager driven frameworks** haben eine zentrale Stelle (Funktion/Objekt), von der im wesentlichen alle Abläufe gesteuert werden. Das SalesPoint Framework läßt sich eindeutig als ein Manager driven framework einordnen, der Manager ist hierbei der Shop, von dem jede SalesPoint Anwendung ausgeht.
2. **Architecture-driven** ist ein Framework, wenn die bevorzugte Anpassungsstrategie das Ableiten eigener Klassen von Framework-Klassen ist. Daher werden solche Frameworks auch als **inheritance-focused** bezeichnet.
3. Ein Framework ist **data-driven** (**composition-focused**)**,** wenn die bevorzugte Anpassungsstrategie die Kombination von vordefinierten Werten und Objekten ist.

Die Unterscheidung Architecture-driven/Data-driven ist dabei ähnlich Prees Unterscheidung in „White Box“/„Black Box“ Frameworks (siehe [Pree97]). Eine wichtige Aussage ist dabei, daß Architecture-driven Frameworks größere Flexibilität erreichen können, gleichzeitig jedoch schwieriger anzuwenden sind. Data-driven Frameworks hingegen sind verhältnismäßig leicht anzuwenden, können aber die Flexibilität erheblich einschränken. Um von beiden Seiten zu profitieren, wird daher eine Kombination einer architekturgetriebenen Basis mit einer datengetriebenen Schicht für leichte Anpassung in Standardfällen empfohlen. Dies ist auch genau der Weg, der im SalesPoint Framework gegangen wurde. Der größte Teil des Frameworks muß als Architecture-driven angesehen werden, Anpassungen erfolgen fast nur durch Unterklassenbildung. Einige Teile jedoch (z.B. Standardformulare, Standard-Swingelemente, weite Teile der Datenhaltung) können auch durch einfaches „Zusammenstöpseln“ vorgefertigter Komponenten angepaßt werden.

Eine weiter wichtige Anregung aus dem Taligent Artikel ist der Gedanke, größere Frameworks in Gruppen kleinerer, entkoppelter, miteinander kommunizierender Frameworks zu zerlegen. Dies steigert die Wiederverwend-

barkeit, da auch einzelne der kleinen Frameworks unabhängig vom gesamten Rest in völlig anderem Kontext verwendet werden können. Dieser Gedanke sollte bei weiterer Wartung und Entwicklung des Frameworks unbedingt Beachtung finden.

3.2. IBM San Francisco

IBM’s San Francisco ist ein sehr allgemeines und umfangreiches Framework für wirtschaftliche Anwendungen auf Basis von Java. Dabei sind nicht nur Kauf und Verkauf von Interesse, sondern der gesamte betriebswirtschaftliche Sektor von Buchführung und Kostenrechnung bis hin zu verkaufsunterstützenden Anwendungen (bspw. Kassensysteme).

San Francisco bietet sehr umfangreiche grundlegende Dienste, wie z.B. Client-/Server-Unterstützung, Transaktionsmanagement, Persistenz, Sicherheit, Konfigurationsmanagement. Aufbauend auf diesen Diensten existieren sogenannte BusinessObjects, die allgemein in betriebswirtschaftlichen Anwendungen benötigte Objekte kapseln. Dies ist eine fundamental andere Herangehensweise als bei SalesPoint. Während San Francisco grundsätzliche Infrastruktur und darauf implementierte Funktionalität strikt trennt, existiert eine solchen Unterteilung in SalesPoint nicht. Die Unterstützung von Transaktionalität (`DataBaskets`) ist beispielsweise eng an die Transaktionalität verwendenden Klassen (`Catalogs/Stocks`) gebunden.

Soweit [Bohr98] erkennen läßt, bietet San Francisco keine Unterstützung für die eigentliche Anwendungsarchitektur. Es ist eher komponentenorientiert und besteht aus verschiedenen, relativ frei kombinierbaren Komponenten. Dadurch wird größere Flexibilität erreicht, allerdings auf Kosten höherer Anforderungen an den Programmierer und eines höheren Aufwands für die Einarbeitung.

Aus [Fend2000] wird ersichtlich, daß San Francisco eher als data-driven Framework nach [TaliOOB] zu betrachten ist. Viele der Anpassungen werden durch entsprechende Parametrisierung durchgeführt. Ein extremer Fall ist dabei die Möglichkeit des Ersetzens ganzer Klassen (sogenanntes „class replacement“).

Weiterhin weist [Fend2000] darauf hin, daß San Francisco mit kompletten ROSE Design Modellen für das Framework ausgeliefert wird, was die Einarbeitung erleichtert. Entsprechendes sollte für das SalesPoint Framework ebenfalls angedacht werden.

4. Ausbaumöglichkeiten

In diesem Abschnitt werden einige angedachte Ausbaumöglichkeiten für das Framework beschrieben.

4.1. JDBC – Anbindung

Die momentane Implementation der Datenhaltungsklassen des Frameworks ist eine Pure Java Implementation. Neben dem daraus resultierenden großen Umfang und der relativ hohen Komplexität der Klassen, gibt es auch gewisse Probleme, die diese Implementation nicht mehr abdeckt. In sehr speziellen Fällen kann es zu einer Verletzung der referentiellen Integrität kommen. Speziell kann dies immer dann vorkommen, wenn ein Benutzer einen Katalogeintrag editiert, zu dem es noch Bestandseinträge gibt. Wenn er anschließend einzelne Bestandseinträge löscht und danach den Namen des Katalogeintrags ändert, kann es vorkommen, daß das Löschen der Bestandseinträge nicht mehr rückgängig gemacht werden kann. Dies ist eine Folge eines Kompromisses zwischen Komplexität der Klassen und Stabilität des Systems. Das beschriebene Problem ist jedoch so speziell, daß es im normalen Betrieb des Frameworks keine Schwierigkeiten bereiten dürfte.

Gleichzeitig jedoch bieten sich die Strukturen von Katalogen und Beständen für eine Implementation in relationalen Datenbanken direkt an. Eine solche Implementierung würde auch die momentan noch existierenden Probleme beheben, da die gesamte Konsistenzsicherung in die Verantwortung der Datenbank überginge.

Obwohl eine solche Implementierung noch nicht existiert, ist doch schon einiges getan, um sie zu ermöglichen. Sämtliche Datenhaltungsklassen existieren zunächst als eine Schicht von Interfaces. Die Implementierung auf Basis einer relationalen Datenbank müßte nur noch diese Interfaces implementieren. Dabei bietet es sich an, den Datenkorb als Träger eines Transaktionsmerkmals (`Connection` in JDBC) zu interpretieren.

Da relationale Datenbanken zwar geschachtelte Transaktionen, gewöhnlich aber nicht beliebiges Hin- und Herwechseln zwischen den Transaktionsebenen erlauben, sind die Unterkorb bezogenen Operationen und die Operationen für bedingtes `commit` und `rollback` im Datenkorb als optional markiert. Die Pure Java Implementation implementiert alle diese Operationen, eine Datenbank-basierte Implementation kann selbst entscheiden, welche der Operationen sie implementiert.

Die Operationen zum Inspizieren eines Datenkorbs sind jedoch nicht optional, so daß eine Datenbank-basierte Implementation für jeden Datenkorb auch eine Tabelle haben muß, in welche sie die Elemente in diesem Datenkorb einträgt. Dies ist nötig, damit der Datenkorb tatsächlich wie ein Einkaufskorb behandelt werden kann, und Anwendungen konstruiert werden können, in denen der Kunde von Stand zu Stand geht und erst zum Schluß abrechnet.

4.2. RMI – Einsatz

Verteilte Anwendungen sind zu einem wichtigen Schlagwort geworden. Auch das Framework wird vermutlich nicht mehr lange ohne Unterstützung für Verteilung auskommen. Hierzu könnte man sich zwei verschiedene Vorgehensweisen vorstellen.

4.2.1. „Basar“ – Betrieb

In dieser Variante stellt man sich vor, daß eine Client-Anwendung einer Server-Anwendung einen SalesPoint zur Verfügung stellt, auf welchem sie die Ausgaben ihrer Prozesse darstellen kann. Der eigentliche Prozeß verbleibt dabei auf der Server-Seite.

Ein möglicher Anwendungsfall wäre die Implementation einer Messe, wobei die Messe als Client und die einzelnen Anbieter als Server auftreten. Kunden könnten dann durch die Messe spazieren, sich Dinge ansehen und anschließend direkt bei einem Anbieter eine Bestellung aufgeben. Ein ähnlicher Fall ist eine Markthalle, die verschiedenen Kleinhändlern einen Stellplatz zur Verfügung stellt.

Ein anderer Anwendungsfall ist bereits eine Art von Vertriebskette. Angenommen, ein Kunde kommt in einen Buchladen und wünscht das Buch „Design Pattern“ von Gamma et al. Dieses Buch ist aber leider im Moment nicht im Laden vorrätig. Der Kunde wünscht eine Bestellung und diese wird angelegt. Dazu loggt sich die Verkäuferin bei Addison-Wesley ein und startet dort einen Prozeß, der Bestellungen aufnimmt. Die Ausgabe dieses Prozesses wird aber auf ihr eigenes Display umgeleitet. Die Bestellung wird dadurch direkt bei Addison-Wesley aufgenommen.

Für die Implementierung dieser Variante gibt es bereits Ansätze im Framework. Ein wichtiger Punkt ist die Entkopplung von Prozessen und ihrer Umgebung durch die Einführung des Interfaces `ProcessContext`. Dadurch wird es möglich, einen Prozeß auch quasi „im Leeren“ also auf `Shop`-Ebene laufen zu lassen. Die Tren-

nung der Anzeige (`Display`) von den anderen Aspekten der Prozeßumgebung ermöglicht es, die Anzeige auf einen anderen Rechner zu verlagern, obwohl der Rest der Prozeßumgebung auf dem Ausgangsrechner verbleibt. Last but not least, die Methode `runProcess` in der Klasse `Shop` erlaubt bereits die Definition der vollständigen Umgebung eines Prozesses, der anschließend auf `Shop`-Ebene abläuft.

Es verbleibt aber auch noch eine Menge zu tun. Zunächst muß die Methode `runProcess` für den RMI-Zugriff „freigeschaltet“ werden. Anschließend mag es durchaus noch zu klärende Probleme mit RMI geben. Es kann zum Beispiel passieren, daß die Formulare und Menüs, die auf einer entfernten Anzeige dargestellt werden sollen, im Rahmen des Marshalling serialisiert werden, anstatt nur einen Proxy für diese zu verschicken. Die RMI-Spezifikation ist nicht sehr eindeutig, welches der beiden Interfaces `Remote` und `Serializable` im Zweifel Vorrang hat. Würden die Formulare und Menüs serialisiert, so verlöre man in diesem einen Fall plötzlich die Möglichkeit, das Aussehen eines bereits angezeigten Formulars oder Menüs zu verändern.

Insgesamt stellt sich aber diese Variante als relativ unkompliziert und durchaus realisierbar dar. Möglicherweise müssen zwar kleinere Änderungen auch an den Schnittstellen von Framework-Klassen vorgenommen werden, die Änderungen sollten aber in einem überschaubaren Rahmen bleiben.

4.2.2. Vertriebsketten

Eine zweite Variante ist der Aufbau von Vertriebsketten mit automatischer Bestellung und Lieferung. Der wichtige Punkt hierbei ist nicht so sehr „Aufbau von Vertriebsketten“ als vielmehr „automatische Bestellung und Lieferung“. Dies umfaßt die Möglichkeit, daß mehrere Anwendungen, zum Beispiel ein Einzelhändler und ein Großhändler, automatisch und ohne Benutzerintervention miteinander Informationen austauschen und Verträge abschließen.

Damit zwei Anwendungen Informationen über Waren austauschen können, benötigen sie zunächst eine gemeinsame Gesprächsbasis, also einen gemeinsamen Katalog. Allein dies ist schon ein sehr umfangreiches Problem, das hier nur kurz angerissen werden soll. Gewöhnlich wird ein solcher gemeinsamer Katalog nicht vorliegen, sondern es wird ein Übersetzungsmechanismus von dem einen in den anderen Katalog definiert werden müssen. Die Übersetzung kann entweder vom Anwendungsbenutzer vorgegeben sein, oder die beiden Anwendungen können sie selbständig untereinander aushandeln.

Wenn es einen gemeinsamen Katalog, bzw. eine Übersetzung zwischen den Katalogen gibt, verbleibt noch der Ablauf der Transaktionen zu klären. Bei Verwendung von Datenkörben stellt sich sofort die Frage, wem der verwendete Datenkorb gehören soll. Außerdem ist das Datenkorb-Konzept vermutlich für Transaktionen, die mehr als zwei Anwendungen überspannen, ungeeignet. Hier müßten neue Transaktionskonzepte entwickelt und implementiert werden. Mögliche Grundlagen dafür wären OSI-TP, Enterprise JavaBeans, JavaSpaces.

Insgesamt ist diese Variante viel aufwendiger als die in 4.2.1 geschilderte „Basar“-Variante. Die Architektur des Frameworks müßte vermutlich nochmals vollständig überarbeitet werden, um Unterstützung für solche Anwendungen zu bieten.

4.3. Werkzeug zur Anwendungsentwicklung

Im Zusammenhang mit Frameworks ist die Frage nach der Unterstützung des Anwendungsentwicklers ein wichtiges Kriterium. Das Framework befindet sich in seiner jetzigen Version auf dem Weg von einem „White-Box“-Framework zu einem „Black-Box“-Framework (siehe [Pree97]).

Eine interessante Vision für die Zukunft stellt die Entwicklung eines Werkzeugs dar, mit dem das Gerüst einer Verkaufsanwendung auf der Basis des Frameworks praktisch ohne Programmierarbeit entwickelt werden könnte. Die Eingaben des Programms wären Dinge wie Problemspezifikation in Form von Use Cases mit Zustandsdiagrammen, Formular- und Menüentwürfe etc. Die Ausgabe wäre generierter Quellcode, der vom Framework zur Verfügung gestellte Komponenten adaptieren und kombinieren würde, um die entsprechende Anwendung zu erzeugen.

Ähnlich wie bei einer mit einem GUI-Builder erstellten Oberfläche wären anwendungsspezifische Programmstücke anschließend von Hand zu ergänzen.

Literaturverzeichnis

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1994.

[FHLS97] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson. *Hooking Into Object-Oriented Application Frameworks.* In Proceedings of the 1997 International Conference on Software Engineering. Boston, Mass., 1997.

[Schmi98] Lothar Schmitz. *WWW-Seiten zu JaCCIE.* http://www2.informatik.unibw-muenchen.de/Research/Tools/JACCIE/.index.html

[Pree97] Wolfgang Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks.* dpunkt, Heidelberg, 1997.

[Bohr98] Kathy A. Bohrer. *Architecture of the San Francisco frameworks.* In IBM Systems Journal 37, 2, 1998.

[TaliOOB] Taligent Inc. *Building Object-Oriented Frameworks.* http://www.ibm.com/java/education/oobuilding.

[Fend2000] Kerstin Fender. *Vergleichende Untersuchungen SalesPoint – IBM San Francisco.* Entwurf zur Diplomarbeit. Dresden, 2000.

[Gamm95] Erich Gamma. *Applying Design Patterns in Java.* In Java Gems, SIGS Reference Library, 1997.

[SUNCode] Sun Microsystems, Inc. *Code Conventions for the JavaTM Programming Language.* http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[Lea2000] Doug Lea. *Draft Java Coding Standard.* http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

Anhänge

A Framework – Doclet

Zur Generierung der JavaDoc Dokumentation des Frameworks wurde ein spezielles Doclet implementiert. Dieses Doclet ist eine Erweiterung des Standard-Doclets und unterstützt folgende zwei Zusatzfunktionen:

1. Neues Tag: „@override“ zur Dokumentation von Methoden. Das @override-Tag gibt an, inwiefern eine Methode zum Überschreiben durch Anwendungsentwickler vorgesehen ist. Methoden werden in die drei Kategorien Always, Sometimes und Never unterteilt.
2. Dokumentation von Hooks und HotSpots nach [FHLS97].

Zur Dokumentation der Hooks und HotSpots wurde eine eigene kleine Sprache entworfen, in welcher die Hooks und HotSpots beschrieben werden können. Eine Datei die Text in dieser Sprache enthält, wird vom Framework – Doclet gelesen und in die entsprechende JavaDoc Dokumentation umgesetzt. Der „Parser“ zur Interpretation der Sprache wurde mit JaCCIE, einem Java Compiler Compiler von der Universität der Bundeswehr München, entwickelt (siehe [Schmi98]).

Der Rest dieses Anhangs beschreibt die Syntax dieser Sprache auf informelle Weise. Die originale Definitionsdatei für die Hook-Dokumentation des Frameworks sowie die Quelldateien des Doclets sind auf der CD-ROM verfügbar.

Die Sprache besteht aus Klartext und einfachen Kommandos. Kommandos beginnen stets mit dem Zeichen ‚@‘. Alle Zeichen nach einem ‚#‘ bis zum Zeilenende, werden als Kommentare interpretiert. Leerzeichen am Zeilenanfang und –ende sowie Leerzeilen werden ignoriert. Textbereiche, die keine Kommandos sind, werden dem letzten Kommando zugeschlagen, falls dessen Syntax dies erlaubt. Ansonsten kommt es zu einem Fehler.

Klartextbereiche können beliebigen HTML-Code enthalten. Es sollte jedoch beachtet werden, daß bei der Generierung ein vollständig formatiertes Dokument entsteht, und daß unachtsam angewandte HTML-Tags die Formatierung einer Seite negativ beeinflussen könnten. Innerhalb dieser Texte können weiterhin die speziellen Tags {@link …}, {@href …}, {@hooklink …} und {@hotspotlink …} verwendet werden. Syntax und Semantik dieser Tags werden weiter unten erläutert.

Die Kernelemente der Dokumentation sind Hooks und HotSpots. HotSpots beschreiben großflächige Anpassungsgebiete innerhalb des Frameworks. Sie können kaskadiert sein, d.h. ein HotSpot kann wiederum andere HotSpots enthalten. HotSpots enthalten Hooks. Ein einzelner Hook beschreibt notwendige Anpassungsschritte, um ein bestimmtes, genau spezifiziertes Ziel zu erreichen. Ein Hook gehört immer zu dem HotSpot der zuletzt definiert wurde. Hooks müssen immer Teil eines HotSpots sein.

Die folgenden Kommandos sind definiert[1]:

Syntax	Erläuterungen
@hotspot:<label>[**;**<level>] <name>	Definiert einen neuen HotSpot. Alle folgenden Kommandos, bis zum nächsten @hotspot Kommando, beziehen sich auf diesen HotSpot. *<label>* ist der interne Name des HotSpots. Er darf keine Leerzeichen enthalten und muß als Dateiname geeignet sein. Dieser Name wird immer dann verwendet, wenn man sich auf den HotSpot beziehen möchte, beispielsweise in einem {@hotspotlink …} Tag. *<level>* ist ein numerischer Wert, der den HotSpot innerhalb der Hierarchie plaziert. Der HotSpot ist Teil des zuletzt definierten HotSpots mit einem kleineren Level. *<name>* ist der Name des HotSpots wie er in der Dokumentation erscheint. Kann Leerzeichen enthalten.

[1] Die Syntax wird als regulärer Ausdruck beschrieben. **Fettgedruckte** Zeichen beschreiben Literale. Zeichen in <spitzen Klammern> beschreiben einzusetzende Werte; die Bedeutung wird in den Erläuterungen erklärt. [Eckige Klammern] beschreiben optionale Angaben.

@description:<description_text>	Definiert eine Beschreibung für den zuletzt definierten HotSpot. Dieses Kommando darf nur zwischen der Definition des HotSpots und des ersten Hooks stehen. *<description_text>* die Beschreibung des HotSpots. Kann mehrere Zeilen einnehmen und {@…} Tags sowie beliebige HTML Tags enthalten. Der erste Satz sollte eine Zusammenfassung der gesamten Beschreibung sein.
@hook:<label> <name>	Definiert einen neuen Hook. Der Hook gehört zum zuletzt definierten HotSpot. *<label>* ist der interne Name des Hooks. Er darf keine Leerzeichen enthalten und muß als Dateiname geeignet sein. Dieser Name wird immer dann verwendet, wenn man sich auf einen Hook bezieht, z.B. in einem {@hooklink …} Tag. *<name>* ist der Name des Hooks, wie er in der Dokumentation erscheint. Kann Leerzeichen enthalten.
@requirement:<requirement_text>	Gibt eine kurze Beschreibung der Bedingungen unter denen der zuletzt definierte Hook angewandt werden kann. *<requirement_text>* die Beschreibung des Hooks. Kann mehrere Zeilen einnehmen und {@…} Tags sowie beliebige HTML Tags enthalten. Der erste Satz sollte eine Zusammenfassung der gesamten Beschreibung sein.
@type:<adaptation> <support>	Definiert den Typ des zuletzt definierten Hooks. *<adaptation>* Die Anpassungsmethode. Muß einer der folgenden Werte sein: • **enable_feature** • **disable_feature** • **replace_feature** • **augment_feature** • **add_feature** Für eine Erläuterung dieser Werte siehe [FHLS97]. *<support>* Die vom Framework gebotene Unterstützung. Muß einer der folgenden Werte sein: • **single_option** • **multi_option** • **parameter_pattern** • **collaboration_pattern** • **open_ended** Für eine Erläuterung dieser Werte siehe [FHLS97].
@uses:<hook_label>	Benennt einen Hook, der vom zuletzt definierten Hook verwendet wird. Es kann mehrere @uses Tags je Hook geben. *<hook_label>* Das label des Hooks, der verwendet wird.

<table>
<tr><td>@participant:[<participant>][; <ref>]</td><td>Benennt eine Klasse, eine Methode, ein Attribut, etc. welches im zuletzt definierten Hook verwendet wird. Es kann mehrere @participant Tags je Hook geben.

<participant> der Name des Teilnehmers wie er in der Hook-Dokumentation erscheint. Wenn nicht definiert, so erscheint dieser Teilnehmer nicht in der Hook-Definition.

<ref> Eine JavaDoc-Referenz wie man sie in einem @see Tag verwenden könnte. Beschreibt die Klasse, die Methode oder das Attribut, welches dem Teilnehmer entspricht. Wenn dieses Feld definiert ist, so wird ein Link auf die Hook-Dokumentation in der Dokumentation des entsprechenden Teilnehmers aufgenommen. Falls <participant> ebenfalls definiert ist, so wird auch ein Link von der Hook-Dokumentation zur Dokumentation des Teilnehmers erstellt.

Mindestens eines von <participant> und <ref> muß definiert sein.</td></tr>
<tr><td>@change:[<level>;]<text></td><td>Spezifiziert einen Änderungsschritt im zuletzt definierten Hook. Es kann mehrere @change Tags pro Hook geben. Die Reihenfolge der Änderungsschritte ergibt sich aus der Reihenfolge der Tags.

<level> Numerischer Wert, der zum Erzeugen hierarchischer Änderungen genutzt werden kann. Verhält sich analog zu <level> bei der Definition von HotSpots. Wenn nicht definiert, wird ‚0‘ angenommen.

<text> Text, der die Änderung beschreibt. Kann mehrere Zeilen einnehmen und {@...} Tags sowie beliebige HTML Tags enthalten.</td></tr>
<tr><td>@constraint:<text></td><td>Definiert eine Bedingung für die Anwendung des zuletzt definierten Hooks. Es kann mehrere @constraint Tags pro Hook geben.

<text> Text, der die Bedingung beschreibt. Kann mehrere Zeilen einnehmen und {@...} Tags sowie beliebige HTML Tags enthalten.</td></tr>
<tr><td>@comment:<text></td><td>Definiert einen Kommentar zum zuletzt definierten Hook. Es kann mehrere @comment Tags pro Hook geben.

<text> Kommentartext. Kann mehrere Zeilen einnehmen und {@...} Tags sowie beliebige HTML Tags enthalten.</td></tr>
<tr><td>{@link <ref> [„<text>”]}</td><td>Fügt einen Link zur JavaDoc Dokumentation der spezifizierten Klasse, Methode oder des spezifizierten Attributs ein.

<ref> Eine JavaDoc-Referenz wie man sie in einem @see Tag verwenden könnte. Beschreibt die Klasse, die Methode oder das Attribut, zu dem ein Link erzeugt werden soll.

<text> Beliebiger Text, der in der Dokumentation erscheint. Wenn dieser Text nicht angegeben wird, wird der Name des Ziels verwendet.</td></tr>
</table>

{@href „<ref>**"** [**„**<text>**"**]**}**	Fügt einen Link zu einem beliebigen Dokument ein. *<ref>* der einzufügende Link. Die URL muß relativ zum Zielverzeichnis der JavaDoc-Dokumentation sein. *<text>* Beliebiger Text, der in der Dokumentation erscheint.
{@hotspotlink <label>**}**	Fügt einen Link zur Dokumentation eines HotSpots ein. *<label>* Das Label des Ziel-HotSpots.
{@hooklink <label>**}**	Fügt einen Link zur Dokumentation eines Hooks ein. *<label>* Das Label des Ziel-Hooks.

Bei der Definition von Hooks und HotSpots muß man Sorgfalt walten lassen, da die Fehlerbehandlung des „Parsers“ sehr minimal ist. Das hängt damit zusammen, daß es sich eigentlich nur um JaCCIEs Scanner handelt, der um eine State-Machine zur Trennung von Analysesituationen angereichert wurde.

B Style Guide

B.1 Namenskonventionen

Der Quellcode des Frameworks folgt, allgemein üblichen Konventionen ([SUNCode], [Lea2000]). Im einzelnen werden die folgenden Punkte beachtet:

- Alle Bezeichner sind an die englische Sprache angelehnt.
- *Namen von Klassen und Interfaces* beginnen mit einem Großbuchstaben und verwenden gemischte Groß- und Kleinschreibung. Das heißt, der Anfangsbuchstabe jedes Wortes, aus dem der Klassenname besteht, wird groß geschrieben. Ein Klassenname enthält normalerweise keine Unterstriche (_). Teile von gewöhnlich zusammengesetzten Substantiven müssen nicht groß geschrieben werden, z.B. würde eine Klasse, die eine Hintergrundaktivität bezeichnet, normalerweise `BackgroundTask` und nicht `Back`**`G`**`roundTask` benannt. Klassennamen sind gewöhnlich Substantive oder Substantivgruppen in nicht abgekürzter Form.

  Eine spezielle Klassen- oder Interfacemarkierung existiert nicht.

  Beispiele für Klassennamen sind:

  ```
  StockItem
  UserManager
  UserManagerListModel
  AbstractListenerList
  ```

  aber ***NICHT***:

  `C``StockItem` (Klassenmarkierung verwendet)
  `UserManager_ListModel` (Unterstriche verwendet)
  `usermanagerList` (falsche Verwendung der gemischten Schreibweise)

- *Methodennamen* beginnen mit einem Kleinbuchstaben verwenden ansonsten aber gemischte Schreibung. Unterstriche sind gewöhnlich nicht Teil eines Methodennamens, können aber zur Erhöhung der Lesbarkeit verwendet werden.
  Methodennamen sollten die Tätigkeit der Methode beschreiben, jedoch nicht zu lang sein. Obwohl es keine Maximallänge für Methodennamen gibt, sind die meisten Namen doch auf eine Länge von höchstens 20 bis 30 Zeichen beschränkt.
  Informationen zur Verwendung von Klammern finden sich in B.2 „Klammern und Einrückungen“.

  Beispiele für Methodennamen sind:

  ```
  getName
  multiplyWithScalar
  setErrorTracking
  isValid
  ```

  aber ***NICHT***:

  `GetName` (Großbuchstabe am Anfang)
  `a` (wenig informativ)
  `getnamewithlengthlowercase` (keine gemischte Schreibung)

- *Namen von Attributen und Variablen* beginnen mit einem Kleinbuchstaben und verwenden gemischte Schreibweise. Unterstriche werden gewöhnlich verwendet, um Bereichsmodifikatoren (‚m‘ und ‚s‘) vom eigentlichen Namen zu trennen. Sie können jedoch auch – sparsam – im Variablennamen selbst verwendet werden. Wiederum sollten Variablennamen informativ, jedoch nicht zu lang sein. Zusätzlich zum Java Standard wird eine adaptierte Form der Ungarischen Notation verwendet. Informationen darüber sowie Beispiele gültiger Variablennamen finden sich in B.1.1 „Adaptierte Ungarische Notation“.
- *Paketnamen* bestehen nur aus Kleinbuchstaben.

  Beispiele für Paketnamen sind:

```
sale
sale.stdforms
log
util
```

B.1.1 Adaptierte Ungarische Notation

Die Ungarische Notation wurde nach dem ungarischen Microsoft Programmierer benannt, der sie zuerst einführte. Ursprünglich handelt es sich um ein C++ Idiom. Eine angepaßte Form dieses Idioms wird im gesamten Quelltext verwendet.

Die Grundidee der Ungarischen Notation ist, den Namen jeder Variable, jeder Klasse und jeden Attributs mit einer kurzen Markierung zu versehen. Diese Markierung beschreibt den Typ des Identifikators und steht am Anfang des Namens. Klassen und Interfaces bekommen in der adaptierten Version keine Markierungen, wohl aber Variablen und Attribute.

Zwei Arten von Markierungen werden verwendet: Bereichs- und Typmarkierungen. **Bereichsmarkierungen** beschreiben, ob es sich um ein Instanz- oder Klassenattribut handelt oder ob es nur eine einfache Variable ist. **Typmarkierungen** beschreiben den Typ der Variable. Dadurch kann eine Typbeschreibung im eigentlichen Variablennamen entfallen.

Drei Arten von Bereichsmarkierungen werden verwendet:

Markierung	Bedeutung
`m_`	Instanzattribut (m für engl. member)
`s_`	Statisches Attribut. (s für statisch)
<keine Markierung>	Einfache Variable.

Typmarkierungen für die einfachen Typen sind in der folgenden Tabelle aufgeführt. Für Objekttypen ergibt sich die Typmarkierung aus allen Buchstaben, die im Klassennamen groß geschrieben werden, nur jetzt in Kleinbuchstaben. Eine Variable namens `Current` vom Typ `StockItem` hieße also **`si`**`Current`. Es gibt jedoch auch Ausnahmen, in denen diese allgemeine Regel nicht gilt. Eine Auswahl dieser Fälle findet sich ebenfalls in der folgenden Tabelle.

Markierung	Typ
`f`	`boolean` (flag).
`n`	`int` (natürlich)
`fl`	`float`
`dl`	`double`
`l`	`long`
`c`	`char`
`s`	`String`
`st`	`Stock`, auch `Set`
`trd`	`Thread`
`sh`	`Shop`
`p`	`SaleProcess`
`usr`	`User`
`vc`	`Vector`
`mp`	`Map`

Diese Vorsätze können auch zusammengesetzt werden, um komplexere Typen zu beschreiben. Ein `Vector` von `StockItems`, welcher `Current` heißt, erhielte den Namen `vcsiCurrent`. Solche Zusammensetzungen

sind nur in Ausnahmefällen länger als zwei Markierungen. Die folgenden Vorsilben haben eine besondere Bedeutung, wenn sie am Anfang einer Zusammensetzung erscheinen:

Vorsatz	Bedeutung
`a`	Array
`c`	Zähler, d.h. ein int, der als Zähler verwendet wird. Beispiele: `ccCharacters` ein Zeichenzähler. `csiItems` ein Zähler von `StockItems`.

Variablen- und Attributnamen setzen sich also wie folgt zusammen[1]:

`<bereichsmarkierung>(<typmarkierung>)+[<name>]`

Beispiele gültiger Namen sind:

`m_civEvaluator`

`s_umGlobal`

`umPersonalUserManager`

`um`

Die letztere Version (also nur Typmarkierungen) wird verwendet, um lokale Variablen und Methodenparameter, die nur zeitweilig verwendet werden und für die kein sinnvoller Name gefunden werden konnte, zu benennen. Weiterhin können die speziellen Bezeichner `i` und `j` als Schleifenindex beliebigen Typs (vorzugsweise numerisch oder Iterator) verwendet werden.

B.2 Klammern und Einrückungen

Klammern, die eine Parameterliste eröffnen, werden gewöhnlich direkt hinter den Methodennamen geschrieben, wenn es keine Parameter gibt. Anderenfalls werden sie erst nach einem Leerzeichen geschrieben. Dies gilt sowohl für Definition als Aufruf einer Methode bzw. eines Konstruktors.

Geschweifte Klammern, die einen Block eröffnen, erscheinen auf der selben Zeile, wie der Blockanfang (z.B. eine `if <bedingung>` Anweisung). Schließende geschweifte Klammern werden an der Anweisung ausgerichtet, die den Block eröffnete. Das ist gewöhnlich eine andere Spalte, als wenn die Klammer mit der öffnenden Klammer ausgerichtet würde.

Jeder Blockinhalt wird um zwei Leerzeichen eingerückt.

[1] Wieder ein regulärer Ausdruck. <Spitze Klammern> bezeichnen Nichtterminale, [eckige Klammern] optionale Anteile, ()+ fordert mindestens einmaliges Vorkommen des geklammerten Ausdrucks.

Der folgende Ausschnitt aus einem Stück Quelltext verdeutlicht noch einmal alle Konventionen:

```
package sale;

import java.io.*;

public class MyClass extends Object implements Serializable {

  protected Object m_oSomeData;

  public MyClass() {
    this (new Object());
  }

  public MyClass (Object oSomeData) {
    super();

    m_oSomeData = oSomeData;
  }

  public void lock() {
    if (m_oSomeData != null) {
      synchronized (m_oSomeData) {
        try {
          m_oSomeData.wait();
        }
        catch (InterruptedException ie) {}
      }
    }
  }
}
```

B.3 Kommentare

Innerhalb des Codes ist die bevorzugte Form des Kommentars der Zeilenkommentar, der mit zwei Schrägstrichen beginnt. Mehrere einzeilige Kommentare können aufeinander Bezug nehmen, um wichtige Entscheidungspunkte im Code zu markieren und zu beschreiben.

Mehrzeilige Kommentare zwischen `/*` und `*/` werden gelegentlich verwendet, um einen Codeabschnitt allgemein zu beschreiben. Sie enthalten Informationen darüber was der Code tut und eine abstrakte Sicht darauf, wie er es tut.

Dokumentationskommentare zwischen `/**` und `*/` werden nur vor Elementen verwendet, die dokumentiert werden werden.

Alle Kommentare sind in englischer Sprache gehalten.

B.4 Modifikatoren

Es gibt keine genaue Regel über die Reihenfolge von Modifikatoren vor einem Element, außer daß die Sichtbarkeitsmodifizierer (`public`, `protected`, `private`) immer an erster Stelle stehen.

C Metriken

Die folgenden Metriken wurden mit dem JavaCount – Tool ermittelt, daß auch von den Praktikumsteilnehmern zum Ermitteln der projektspezifischen Metriken verwendet wurde.

Leider hat dieses Tool einige Probleme mit manchen Quelldateien (insges. 11 der 206 Dateien), so daß für diese Dateien keine exakten Werte ermittelt werden konnten. Die entsprechenden Werte wurden anhand der Mittelwerte der analysierten Dateien hochgerechnet. Bei der geringen Prozentzahl (nur ca. 5%) der nicht analysierten Dateien erscheint dieses Vorgehen gerechtfertigt. Das detaillierte Protokoll der Zählung findet sich in der Datei LOC.out auf der beiliegenden CD-ROM. Die beiden Perl-Skripte LOC.pl und TraversDir.pm wurden verwendet, um die Zählung durchzuführen.

Package	Anzahl								
	Dateien			Klassen			LOC		
	analysiert	überspr.	gesamt	analysiert	geschätzt	gesamt	analysiert	geschätzt	gesamt
data	37	3	40	37	3	40	1075	87,2	1162,2
data.events	10	0	10	10	0	10	146	0	146
data.filters	7	0	7	8	0	8	976	0	976
data.ooimpl	19	0	19	26	0	26	3989	0	3989
data.stdforms	1	1	2	1	1	2	1151	1151	2302
…single table fomsheet	3	0	3	3	0	3	152	0	152
…two table fomsheet	12	0	12	12	0	12	1266	0	1266
data.swing	18	2	20	19	2,1	21,1	1026	114	1140
log	7	1	8	7	1	8	166	23,7	189,7
log.stdforms	1	0	1	1	0	1	51	0	51
log.swing	3	0	3	3	0	3	115	0	115
sale	32	0	32	38	0	38	3874	0	3874
sale.events	9	0	9	9	0	9	90	0	90
….multiwindow	2	0	2	3	0	3	840	0	840
sale.stdforms	1	2	3	1	2	3	20	40	60
sale.swing	2	0	2	3	0	3	67	0	67
users	9	0	9	9	0	9	430	0	430
users.events	6	0	6	6	0	6	61	0	61
users.stdforms	1	0	1	1	0	1	89	0	89
users.swing	4	0	4	5	0	5	178	0	178
util	5	2	7	5	2	7	174	69,6	243,6
util.swing	6	0	6	6	0	6	250	0	250
Gesamt	195	11	206	213	12	225	16186	913	17099

Achtung: Die Schätzungen für die Gesamtwerte ergeben sich nicht aus der Summe der Schätzungen für die einzelnen Pakete. Diese Werte wurden anhand der Summe der ermittelten Werte hochgerechnet.

D Inhalt der CD-ROM

- Beleg — Großer Beleg in elektronischer Form
  - Grosser Beleg.doc — Der Beleg
  - LOC.pl — PERL Skript zur Ermittlung der LOC im Framework
  - TraverseDir.pm — PERL Skript zur Ermittlung der LOC im Framework
- Doclet — Quelltext des Doclets zur Generierung der Javadoc zu Version 2.0 des Frameworks
  - doclet — Hauptpaket der Doclet-Klassen
- Framework — Sämtliche Quellen der Frameworks, Versionen 0.5 bis 2.0
  - 0.5 — Quelltext der Version 0.5
  - 1.0 — Quelltext der Version 1.0
  - 2.0 — Quelltext der Version 2.0 (Stand 10/30/2000)
- JavaDoc — Javadoc zu Version 2.0 des Frameworks
  - JavaDoc-Quellen — Quelldateien, die zur JavaDoc-Generierung verwendet wurden
  - v2.0 — JavaDoc zu Version 2.0 des Frameworks
    - javadoc — eigentliche JavaDoc Dateien
    - hooks — Hook und Hotspot - Doku
- UML — Together 4.0 Projekt mit Reverse Engineered Diagrams für SalesPoint v2.0
- Examples — Anwendungsbeispiele zu Version 2.0 des Frameworks
  - tests — Testprogramme der Datenklassen
  - salespointtutor — Tutorial-Code

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur/Hilfsmittel erstellt habe.

Dresden, 8. Juli 2004 ________________________________