

Schriftliche Hausarbeit

vorgelegt im Rahmen der Ersten Staatsprüfung für das Lehramt für die
Sekundarstufe I/II in Informatik von

Ulrich Block

Thema:

Weiterentwicklung der Software „Schulkiosk“ und Konzeption einer
darauf basierenden Unterrichtseinheit für die Sekundarstufe II
nach dem fachdidaktischen Ansatz der Dekonstruktion

Paderborn, 15. Mai 2002

Gutachter: Prof. Dr. Johannes Magenheim
Universität Paderborn
Fachbereich 17: Didaktik der Informatik

Inhaltsverzeichnis

1	Einleitung.....	2
2	Dekonstruktion als Unterrichtsmethode.....	3
3	Vorstellung der Software Schulkiosk	7
3.1	Funktionsumfang der verschiedenen Ausbaustufen der Software Schulkiosk.....	7
3.1.1	Funktionalität der Version 1.0.....	7
3.1.2	Funktionalität der Version 1.1.....	12
3.1.3	Funktionalität der Version 1.2.....	17
3.2	UML-Design und Implementierungsdetails der Software Schulkiosk.....	19
4	'Didaktische Fenster' auf die Software Schulkiosk.....	27
4.1	Schluss von Funktionalität und Erprobung der Software Schulkiosk auf das ihr zugrunde liegende objektorientierte Design.....	27
4.2	Systemgestaltung als kooperativer Kommunikationsprozess.....	30
4.3	Software-Ergonomie: Die Kasse der Version 1.0 – Beispiel einer unpraktikablen Benutzerschnittstelle.....	32
4.4	Bidirektionale Assoziationen bzw. -Aggregationen im Quelltext.....	35
4.4.1	Bidirektionale 1:1-Assoziationen bzw. -Aggregationen.....	35
4.4.2	Bidirektionale 1:n-Assoziationen bzw. -Aggregationen.....	40
4.5	Das Observer-Pattern als Beispiel für Entwurfsmuster.....	47
4.5.1	Beschreibung des Observer-Patterns.....	47
4.5.2	Umsetzung des Observer-Patterns in Java.....	48
4.5.3	Das Observer Pattern in der Software Schulkiosk.....	49
4.6	Der Iterator – ein weiteres Muster.....	53
4.6.1	Der Iterator in Java.....	53
4.6.2	Der Iterator in der Software Schulkiosk.....	53
4.7	Ereignisbehandlung.....	57
4.7.1	Beschreibung der Ereignisbehandlung in Java.....	57
4.7.2	Ereignisbehandlung in der Software Schulkiosk.....	58
4.8	Ausnahmebehandlung.....	63
4.8.1	Ausnahmebehandlung in Java.....	63
4.8.2	Auslösen von Ausnahmen in eigenen Methoden.....	67
4.8.3	Fujaba-Aktivitätsdiagramme und Ausnahmebehandlung.....	68
4.9	Algorithmen – Sortieralgorithmus in der Software Schulkiosk.....	71
5	Resümee.....	73
6	Literatur.....	74
7	Anlage.....	75
8	Versicherung.....	76

1 Einleitung

Mit der im Rahmen dieser Examensarbeit (weiter-)entwickelten Software Schulkiosk soll ein relativ komplexes Softwaresystem bereitgestellt werden, das geeignet ist, im Informatikunterricht der Sekundarstufe II eingesetzt zu werden. Es ist mit Blick auf den von Johannes Magenheim vertretenen fachdidaktischen Ansatz der *Dekonstruktion von Informatiksystemen als Unterrichtsmethode*¹ entwickelt worden. Dabei sind u.a. auch Erfahrungen aus dem „Life³“-Projekt² (Life³ = Lernwerkzeuge für den Informatikunterricht: Einsetzen, Evaluieren und (Weiter)Entwickeln) eingeflossen, was sich zum Beispiel darin widerspiegelt, dass als Entwicklungsumgebung für die Fachklassen der Software Schulkiosk das an der Universität Paderborn entwickelte Tool Fujaba verwendet wurde.³ Zunächst einmal soll also der von Johannes Magenheim im Rahmen einer systemorientierten Didaktik der Informatik beschriebene fachdidaktische Ansatz der Dekonstruktion kurz vorgestellt werden. Anschließend stehen Umfang und Aufbau der Software Schulkiosk im Mittelpunkt. Im darauf folgenden Kapitel werden dann sog. 'didaktische Fenster'⁴ als mögliche Zugänge zur Software Schulkiosk beschrieben, wie sie Inhalte der Dekonstruktion von Informatiksystemen im Informatikunterricht sein können. Diese 'didaktischen Fenster' übernehmen schließlich die Rolle der im Thema dieser Arbeit beschriebenen Konzeption einer Unterrichtseinheit, da eine zu detailliert ausgearbeitete, auf der Software Schulkiosk basierende Unterrichtseinheit, ohne die konkrete Lerngruppe und ihren Leistungsstand bzw. ihr Vorwissen vor Augen zu haben, nicht sinnvoll umsetzbar erscheint und auch nur einen kleinen Einblick in das Potential der Software Schulkiosk zu geben vermag. Ein kurzes Resümee soll schließlich den Abschluss dieser Arbeit bilden.

1 Vgl. Hampel, Thorsten; Magenheim, Johannes; Schulte, Carsten: Dekonstruktion von Informatiksystemen als Unterrichtsmethode, 149ff.

2 Siehe: <http://www.uni-paderborn.de/cs/ag-magenheim/life/>

3 Siehe: <http://www.fujaba.de>

4 Magenheim, Johannes : Informatiksystem und Dekonstruktion als Didaktische Kategorien

2 Dekonstruktion als Unterrichtsmethode

Einen Informatikunterricht, der in weiten Teilen sehr stark an einen Programmierkurs erinnert, und dabei, „orientiert am Konzept einer Programmiersprache, systematisch neue Sprachelemente einzuführen versucht“⁵, trifft zu Recht der Vorwurf, dass der „für ein Unterrichtsfach an allgemeinbildenden Schulen übliche Anspruch auf Vermittlung von Allgemeinbildung [...] so kaum zu realisieren“⁶ ist. In ihrem Artikel „*Dekonstruktion von Informatiksystemen als Unterrichtsmethode*“ gehen Hampel, Magenheim und Schulte auf einige Gründe dafür ein: „Aufgaben und Beispiele weisen häufig einen geringen Komplexitätsgrad auf oder beziehen sich auf Standardalgorithmen mit engem Bezug zur Mathematik. Problemlösestrategien, Schulung analytischen Denkens, Abstraktionsvermögen, Zergliedern von Problemen in Teilprobleme und andere Problemlösestrategien können hierbei nur eingeschränkt eingeübt werden.“⁷

Magenheim betont zudem in seinem Artikel „*Informatiksystem und Dekonstruktion als didaktische Kategorien*“: „Bei einem ausschließlich an Formalismen orientierten Zugang entstehen Unterrichtskonzepte, bei denen die von Lehrplänen immer wieder eingeforderte Behandlung von gesellschaftlichen Auswirkungen von Informatiksystemen im Unterricht bestenfalls in Form von knappen Exkursen und als Alibiveranstaltungen an die 'wahren' informatischen Inhalte angefügt werden. Macht man hingegen die Modellierung von Informatiksystemen aus der Perspektive eines sozio-technischen Handlungssystems zum Ausgangspunkt des Informatikunterrichts, könnte es gelingen, formale Operationen und informatische Prinzipien als Elemente des Softwareentwicklungsprozesses mit Fragen von Anwendungen und Auswirkungen von Informatiksystemen zu verbinden.“⁸

Magenheim als Vertreter einer systemorientierten Didaktik spricht davon und geht wie diese „pragmatisch von einem zentralen Auftrag an die Informatik aus, wissenschaftliche Methoden und Verfahren bereitzustellen, die es ermöglichen, für gegebene Zwecke gebrauchsfähige Informatiksysteme erstellen [zu]

5 Hampel, Magenheim, Schulte: Dekonstruktion von Informatiksystemen als Unterrichtsmethode, 149

6 Ebd., 149

7 Ebd., 149

8 Magenheim: Informatiksystem und Dekonstruktion als didaktische Kategorien

können. Damit rücken Prozesse und Methoden der Systemgestaltung und Softwareentwicklung in den Mittelpunkt didaktischer Betrachtung.“⁹ Und damit rückt auch in gewissem Sinne die Objektorientierung in den Blickpunkt. Denn, so argumentieren Hampel, Magenheim und Schulte in Form einer These: „Objektorientierte Softwareentwicklung und Objektorientierung wird in manchen Publikationen zum Paradigma erhoben und ist eine fundamentale Idee der Fachwissenschaft, die es im wissenschaftspropädeutischen und allgemeinbildenden Sinn zu thematisieren gilt. Hierzu müsste es gelingen, die Komplexität einer Software, eines Informatiksystems für den unterrichtlichen Einsatz didaktisch zu reduzieren, ohne dass die Einführung objektorientierter Sichtweisen beim verwendeten Beispiel zur Trivialität verkommt.“¹⁰ In diesem Zusammenhang soll auch ein aktuelles Statement von der OO-Konferenz OOPSLA 2001 in Tampa, Florida nicht unerwähnt bleiben, denn dort betonte Krysten Nygaard, einer der Erfinder von Simula, im Rahmen des „Educators Symposium“, dass „die Lehrbeispiele für Objektorientierung 'ausreichend komplex' anstatt 'ausreichend einfach' sein sollten, um den Gedanken und die Vorteile von Objektorientierung besser zu erläutern.“¹¹

Eben hier will Dekonstruktion von Informatiksystemen als Unterrichtsmethode ansetzen, indem sie eine Möglichkeit eröffnet, die o.g. Prozesse und Methoden der Systemgestaltung geeignet im Informatikunterricht zu thematisieren. Dekonstruktion als Begriff ist dabei zu verstehen als „eine zentrale Kategorie des Konzepts einer systemorientierten Didaktik der Informatik. Er beschreibt einerseits eine wissenschaftstheoretische Methode zur Analyse und Erschließung von Informatiksystemen für didaktische Fragestellungen im Informatikunterricht. Andererseits kann daraus auch ein konkretes unterrichtsmethodisches Vorgehen abgeleitet werden.“¹²

Als wissenschaftstheoretische Methode lehnt sich die Dekonstruktion an eine von Jacques Derrida in den sechziger Jahren des vorigen Jahrhunderts entwickelte Methode zur Beurteilung von Texten an, die von ihm als Dekonstruktion bezeichnet wurde.¹³ Magenheim schreibt dazu: „Nicht ein externes Normensystem wird zur Bewertung und zum Vergleich von Texten

9 Ebd.

10 Hampel, Magenheim, Schulte: Dekonstruktion von Informatiksystemen als Unterrichtsmethode, 151

11 Josuttis, Nicolai: Grenzen. OO-Konferenz: Diskussion um Design und Methodik, 22

12 Magenheim: Informatiksystem und Dekonstruktion als didaktische Kategorien

13 Vgl. ebd.

herangezogen, sondern durch den intensiven Nachvollzug des argumentativen Aufbaus des Textes wird versucht, Widersprüchlichkeiten des im Text etablierten Begriffssystems zu entdecken. Dekonstruktion bedeutet Auflösung und Entstrukturalisierung des Textes. Vor allem Ungesagtes, Angedeutetes kann von zentraler Bedeutung sein. Dies gilt auch für die Software, deren Modellierungsprämissen und Entwurfsentscheidungen, wenn überhaupt, nur implizit oder gar nicht mehr erkennbar sind und quasi archäologisch in Schichten freigelegt werden müssen.“¹⁴

Für Software gilt also laut Magenheim in diesem Sinne, dass sie – verbunden mit gewissen Einschränkungen natürlich – „als Text zur Beschreibung und Steuerung von maschinellen Aktivitäten des Informatiksystems und von Mensch-Maschine Interaktion angesehen werden kann. In diese Beschreibung sind implizit Entwurfs- und Designentscheidungen der Entwickler eingeflossen, die jedoch z.T. irreversibel und daher nicht mehr reproduzierbar sind. Dekonstruktion von Software heißt insofern, in einer ersten Annäherung an den Begriff, interpretieren von Software, um bei den Lernenden im Informatikunterricht Sensibilität für Gestaltungsräume und für Prozesse der Softwareentwicklung zu erzeugen. Dekonstruktion ist daher mehr als das Lesen von Quellcode einer Software, zum Zwecke des Erlernens der Syntax einer Programmiersprache anhand eines Beispiels.“¹⁵

Was aber kann dies konkret für den Informatikunterricht bedeuten? - Nun, erforderlich ist eine didaktische Software mit hinreichender jedoch unter didaktischen Gesichtspunkten reduzierter Komplexität. Sie muss u.a. exemplarische 'didaktische Fenster' auf tieferliegende Schichten des durch die Software repräsentierten Informatiksystems möglich machen. Von den Lernenden sollen einzelne Zusatzmodule selbst entwickelt und in die Software integriert werden können, was eine entsprechende Offenheit der didaktischen Software voraussetzt.¹⁶ Zudem „sollten Transfermöglichkeiten zu einem realen Informatiksystem [...] bestehen, das auch unter Einbeziehung seines sozialen Umfeldes erkundet werden kann.“¹⁷

„Unterrichtsmethodisch gesehen bedeutet Dekonstruktion [...]: Erkunden der

14 Ebd.

15 Ebd.

16 Vgl. Hampel, Magenheim, Schulte: Dekonstruktion von Informatiksystemen als Unterrichtsmethode, 150

17 Ebd., 154

Systemfunktionalität der Software, von Gestaltungsmerkmalen ihrer Benutzungsoberflächen und partieller Analyse des implementierten Sprachcodes an ausgewählten Beispielen sowie das sukzessive Erlernen von Modellierungs- und Gestaltungsprozessen anhand des Beispiels.“¹⁸ Dabei hält Magenheim eine gleichberechtigte Verschränkung von konstruktiven und dekonstruktiven Arbeitsphasen für sinnvoll. Denn während „durch Dekonstruktion einer Software größere systemische Zusammenhänge erschlossen werden, können durch ergänzende themenbezogene kleinere Beispiele informatische Konzepte vertieft eingeübt werden. [...] Diese Unterrichtsmethodik kann als *Oszillieren* zwischen Dekonstruktion und Konstruktion, als Wechsel zwischen komplexerem System und kleinerer Aufgabenstellung mit Programmieranteilen, als inhaltlich aufeinander bezogene Abfolge zwischen 'Modellieren im Großen' und 'Programmieren im Kleinen' angesehen werden.“¹⁹

Auf die im Rahmen dieser Arbeit (weiter-)entwickelte Software Schulkiosk als eine wie oben beschriebene didaktische Software soll nun im Folgenden eingegangen werden.

18 Magenheim: Informatiksystem und Dekonstruktion als didaktische Kategorien

19 Ebd.

3 Vorstellung der Software Schulkiosk

Im Folgenden soll die Software Schulkiosk beschrieben werden, wie sie auf der beigelegten CD-Rom enthalten ist. Sie wurde in drei verschiedenen Ausbaustufen entwickelt, deren jeweiliger Funktionsumfang zunächst kurz dargestellt werden soll. Anschließend wird der Blick auf das zugrundeliegende UML-Design sowie auf einige Implementierungsdetails der Software gerichtet.

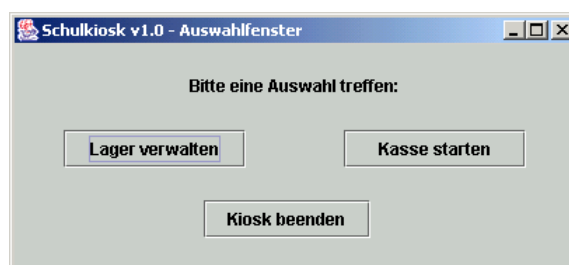
3.1 Funktionsumfang der verschiedenen Ausbaustufen der Software Schulkiosk

Beginnen wir mit der kleinsten Ausbaustufe der Software Schulkiosk in der Version 1.0. Diese bietet die Basisfunktionalität einer Software für das Betreiben eines Schulkiosks und wird im folgenden Abschnitt vorgestellt.

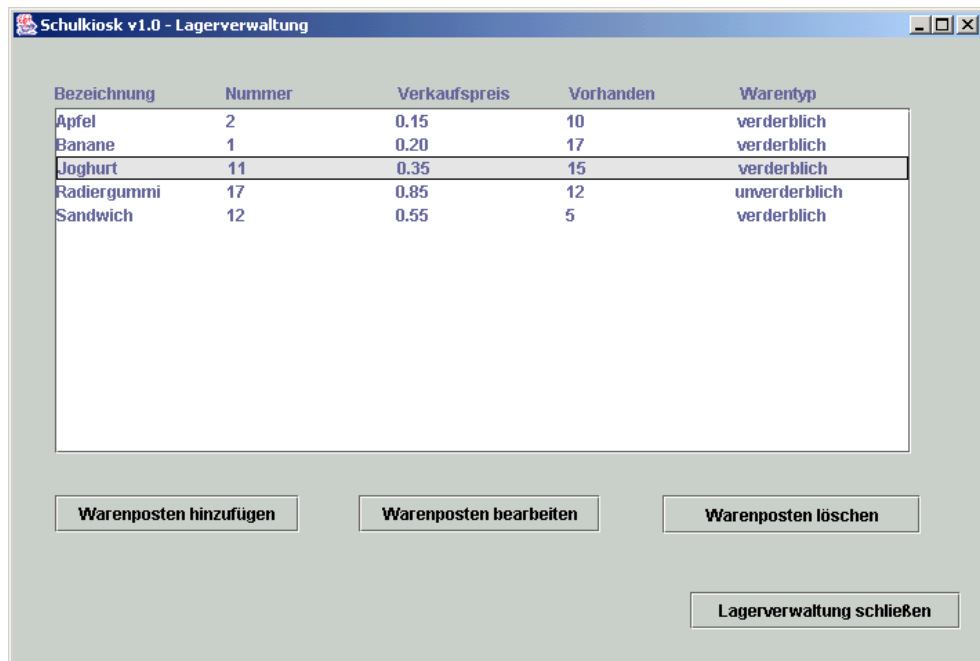
3.1.1 Funktionalität der Version 1.0

Die Version 1.0 besteht aus einer Lagerverwaltung und einer mit rudimentärer Funktionalität ausgestatteten Kasse. Diese Version ist gedacht als Ausgangspunkt, von dem aus Schülerinnen und Schülern die Möglichkeit gegeben werden soll, sie um zusätzliche Module bzw. Funktionalität zu ergänzen.

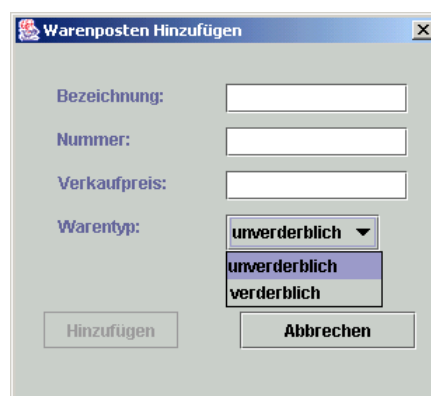
Nach dem Start der Software Schulkiosk in der Version 1.0 öffnet sich ein Auswahlfenster, das die Möglichkeiten bietet, die Lagerverwaltung bzw. die Kasse zu starten oder den Kiosk zu beenden.



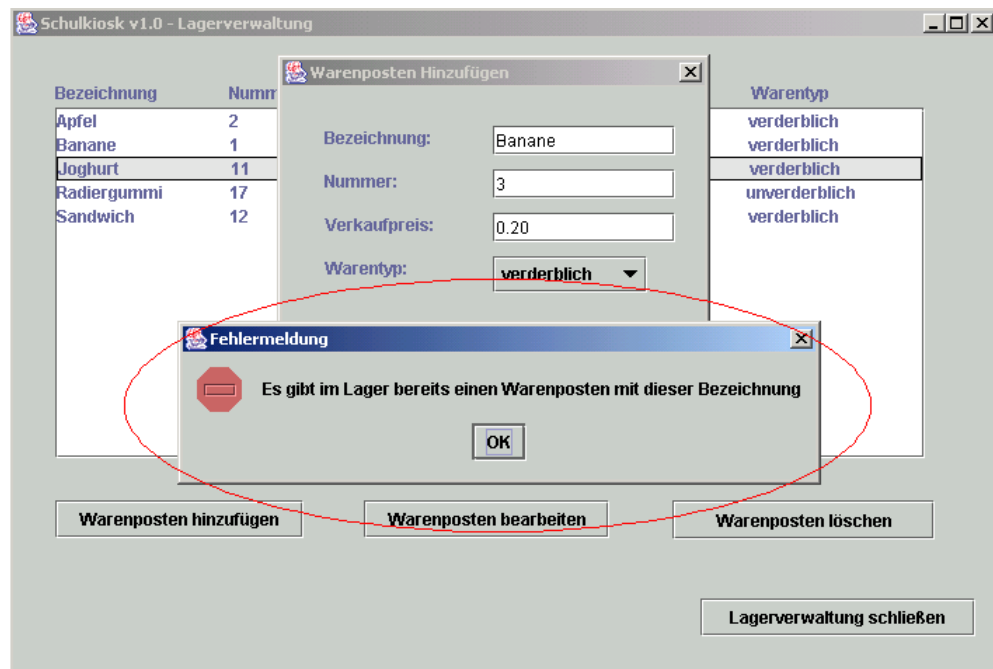
Wird die Lagerverwaltung gestartet, hat der Benutzer die Möglichkeit, einen neuen Warenposten hinzuzufügen, einen in der alphabetisch sortierten Liste ausgewählten Warenposten zu bearbeiten (d.h. ihm neue Waren hinzuzufügen oder Waren zu entfernen), ihn zu löschen oder die Lagerverwaltung zu schließen. Die Lagerverwaltung präsentiert sich folgendermaßen:



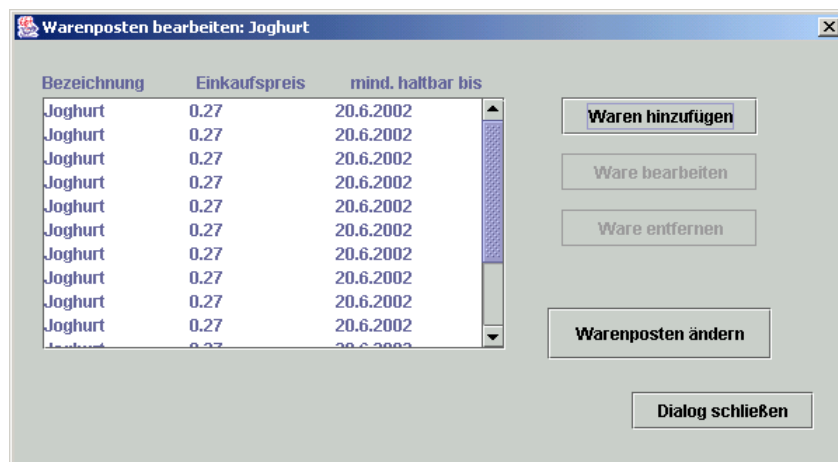
Der Dialog, der erscheint, wenn der Benutzer einen Warenposten hinzufügen möchte, hat die folgende Gestalt:



Hier werden die Attribute eines Warenpostens festgelegt: Bezeichnung, Nummer, Verkaufspreis, Warentyp. Wird nun der Button „Hinzufügen“ betätigt, so überprüft die Software, ob im Lager schon ein anderer Warenposten mit gleicher Bezeichnung oder Nummer vorhanden ist. Ist dies der Fall, erhält der Benutzer eine Fehlermeldung, welche auf der nächsten Seite zu sehen ist:

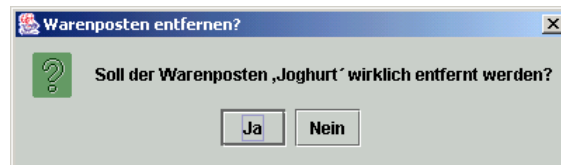


Nach einem Klick auf den Button „*Warenposten bearbeiten*“ in der Lagerverwaltung präsentiert sich dem Benutzer folgendes Dialogfenster, in dem er den in der Liste aller Warenposten ausgewählten Warenposten bearbeiten kann:

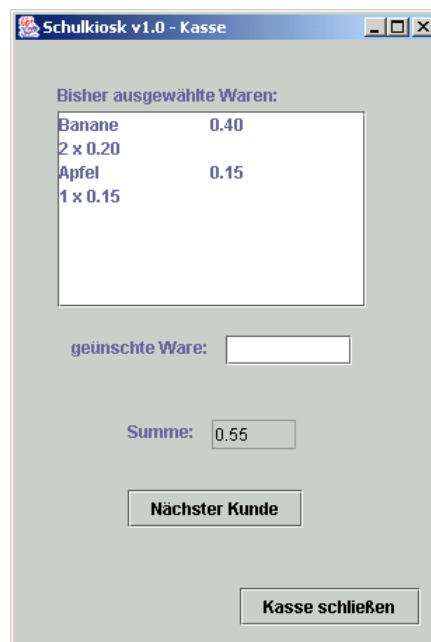


Hier werden alle im betreffenden Warenposten enthaltenen Waren einzeln – bei verderblichen Waren alphabetisch sortiert – angezeigt. Der Benutzer kann nun Waren hinzufügen, einzelne Waren bearbeiten bzw. entfernen oder über den Button „*Warenposten ändern*“ die Attribute des Warenpostens ändern.

Natürlich wird – viele kennen dies von Standardsoftware und mögen dies durchaus als lästig empfinden – der Benutzer, wenn er eine Ware aus einem Warenposten oder einen Warenposten aus dem Lager entfernen will, nochmals gefragt, ob er dies wirklich möchte:



Bleibt in der Version 1.0 noch die mit rudimentärer Funktionalität ausgestattete Kasse, auf die in Kapitel 4.3 noch näher eingegangen wird:

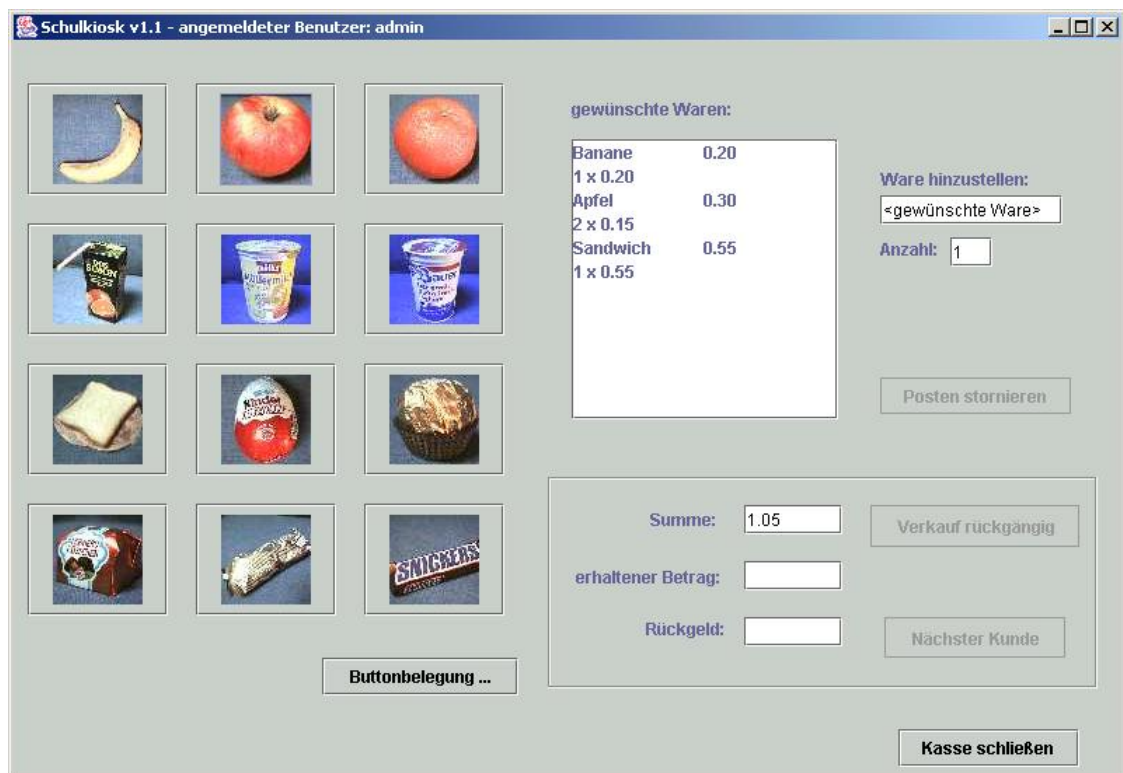


Der Benutzer hat bei dieser Kasse die Möglichkeit, eine neue Ware zu den ausgewählten Waren hinzuzufügen, indem er Bezeichnung oder Nummer der entsprechenden Ware in das Textfeld „gewünschte Ware“ eingibt und mit der Eingabetaste bestätigt. Ist der Verkauf abgeschlossen, drückt er den Button „Nächster Kunde“.

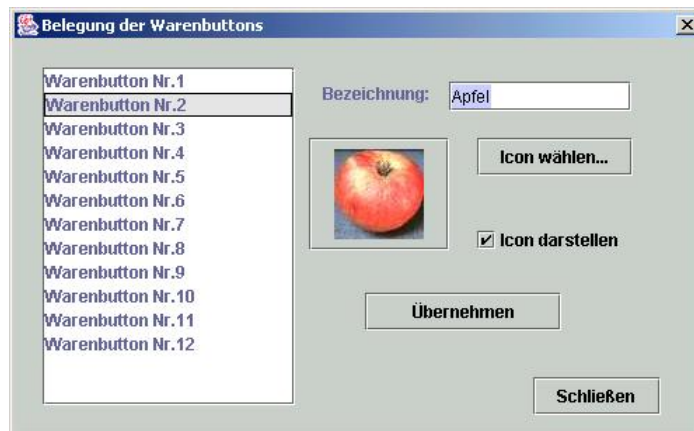
An dieser Stelle ist der Funktionsumfang der Software Schulkiosk in der Version 1.0 zunächst einmal grob umrissen.

3.1.2 Funktionalität der Version 1.1

Die Version 1.1 bietet gegenüber der Version 1.0 einige Erweiterungen. Die Kasse ist mit erweiterter Funktionalität ausgestattet, d.h. der Benutzer hat zusätzlich die Möglichkeit, ausgewählte Warenposten zu stornieren, den Verkauf rückgängig zu machen, es wird für einen eingegebenen Betrag das Wechselgeld berechnet. Außerdem gibt es zwölf Warenbuttons, die es ermöglichen, bestimmte Waren (z.B. häufig verkaufte Waren) schnell einzugeben. Werfen wir also einen Blick auf die Kasse, wie sie in den Versionen 1.1 und 1.2 enthalten ist:

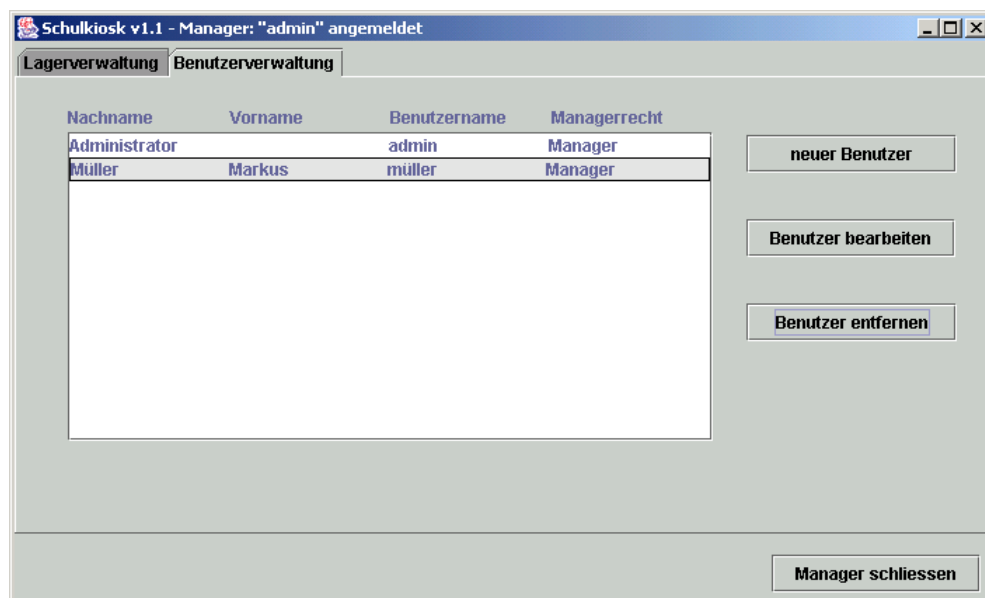


Hinter dem Button „*Buttonbelegung...*“ verbirgt sich ein Dialog, in dem die Belegung der zwölf Warenbuttons für die schnelle Eingabe von Waren vorgenommen und geändert werden kann. Der Dialog ist auf der nächsten Seite dargestellt:



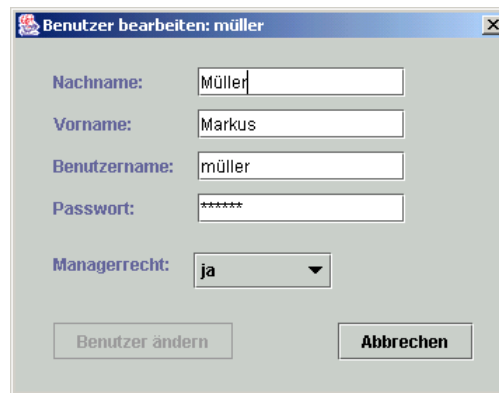
Über „Icon wählen“ kann in obigem Dialog jedem Button ein eigenes Icon, das z.B. als .jpg-Datei vorliegt, zugewiesen werden. Das Häkchen der Checkbox „Icon darstellen“ legt fest, ob das gewählte Icon oder aber die Bezeichnung der Ware auf dem betreffenden Warenbutton in der Kasse erscheinen soll.

Neben der erweiterten Funktionalität der Kasse wurde diese Version noch um eine Benutzerverwaltung ergänzt. Sie ist integriert in das Manager-Fenster, welches die Lagerverwaltung aus der Vorversion enthält und nun zusätzlich noch die Funktionen einer Benutzerverwaltung bereitstellt:



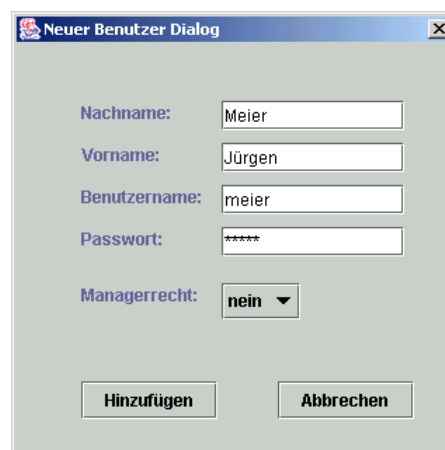
In der Benutzerverwaltung können neue Benutzer aufgenommen, bestehende Benutzer bearbeitet oder entfernt werden. Das Aufnehmen neuer und das

Bearbeiten bestehender Benutzer geschieht über die beiden folgende Dialoge:



The dialog box titled 'Benutzer bearbeiten: müller' contains the following fields and controls:

- Nachname: Müller
- Vorname: Markus
- Benutzername: müller
- Passwort: *****
- Managerrecht: ja (dropdown menu)
- Buttons: Benutzer ändern, Abbrechen

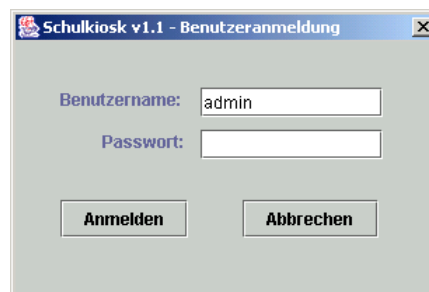


The dialog box titled 'Neuer Benutzer Dialog' contains the following fields and controls:

- Nachname: Meier
- Vorname: Jürgen
- Benutzername: meier
- Passwort: *****
- Managerrecht: nein (dropdown menu)
- Buttons: Hinzufügen, Abbrechen

Die Benutzer haben, wie man sieht, die Attribute Nachname, Vorname, Benutzername, Passwort. Außerdem wird unterschieden zwischen Benutzern mit Managerrecht und ohne selbiges. Letzteren ist es nicht möglich, den Manager zu starten.

Bedingt durch die Benutzerverwaltung erscheint beim Start der Software ein Anmeldedialog, der folgende Gestalt hat:



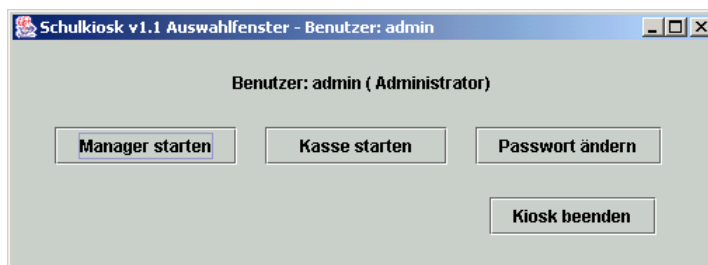
The dialog box titled 'Schulkiosk v1.1 - Benutzeranmeldung' contains the following fields and controls:

- Benutzername: admin
- Passwort: (empty)
- Buttons: Anmelden, Abbrechen

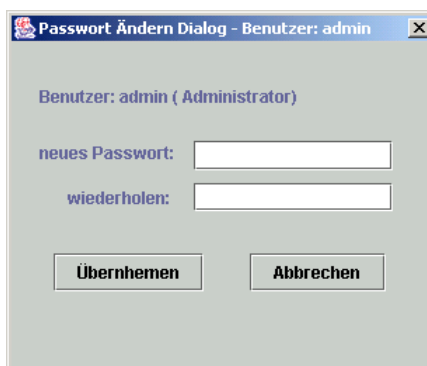
Wird die Software erstmals gestartet, ist also in der Benutzerverwaltung noch kein Benutzer eingetragen, so existiert standardmäßig der Benutzer

„Administrator“ mit dem Benutzernamen „admin“ und keinem Passwort.

Nach einem Klick auf „Anmelden“ erscheint in Abweichung von Version 1.0 folgendes Auswahlfenster:



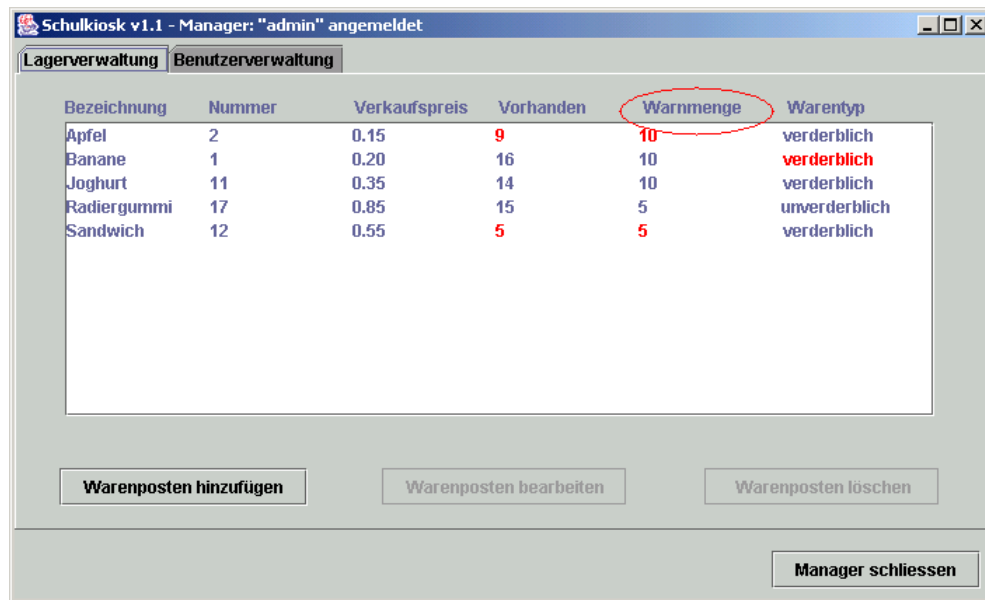
Hier hat der nun angemeldete Benutzer die Möglichkeit, einmal – sofern er Managerrechte besitzt – den Manager zu starten. Der Start der Kasse, das Ändern des Passworts und das Beenden des Kiosks sind weitere Optionen an dieser Stelle. Der Dialog zum Ändern des Passworts ist der folgende:



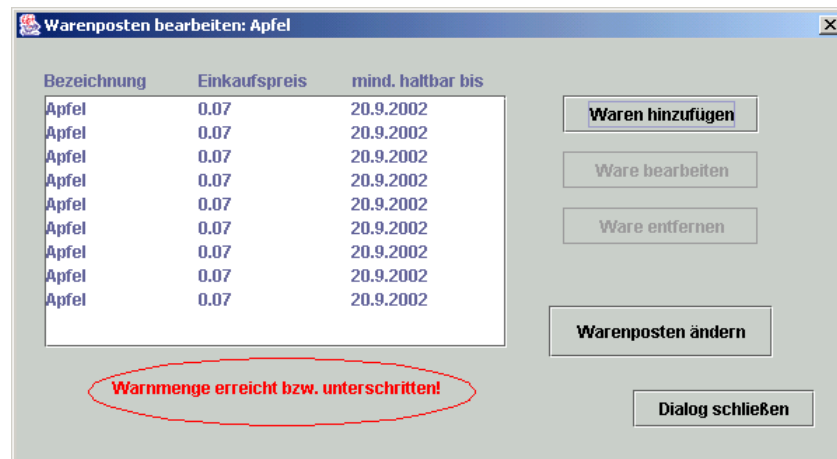
Eine weitere Neuerung im Vergleich zur Version 1.0 ist das zusätzliche Attribut Warnmenge des Warenpostens.

Es hat den Zweck, dass bei Unterschreitung der Warnmenge der Benutzer der Software darüber informiert wird und ggf. Waren frühzeitig nachbestellen kann. Dieses zusätzliche Attribut bedingt eine Erweiterung der Dialoge, in denen ein neuer Warenposten hinzugefügt bzw. ein bestehender geändert wird. Auf deren ausführliche Darstellung kann hier allerdings verzichtet werden, da sich außer einem zusätzlichem Eingabefeld für die Warnmenge keine Änderungen ergeben haben.

Allenfalls ein Blick auf die Repräsentation der Lagerverwaltung im Manager lohnt sich an dieser Stelle:



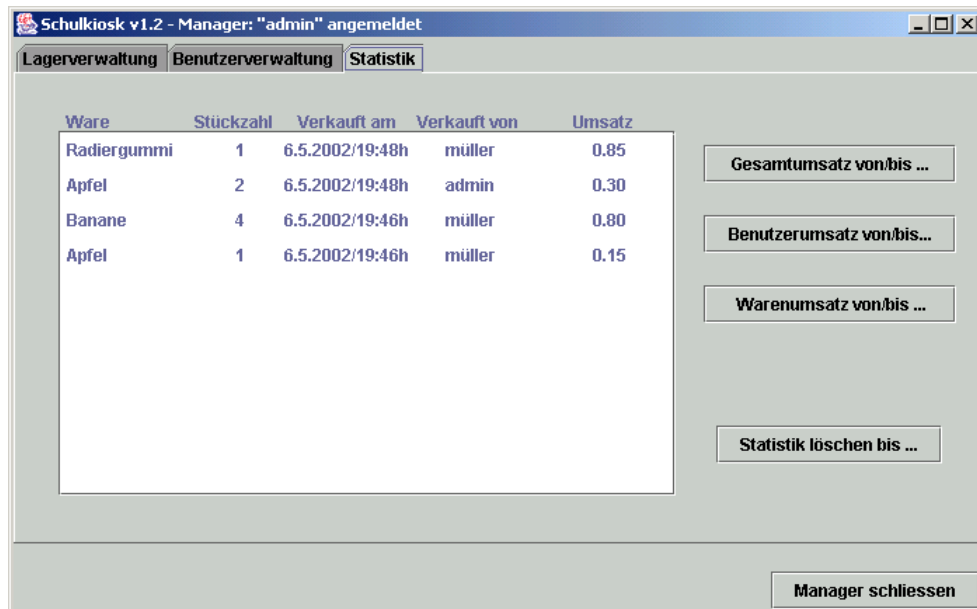
Ist hier die Warnmenge einer Ware erreicht bzw. unterschritten, so werden die entsprechenden Werte unter „vorhanden“ und „Warnmenge“ rot hervorgehoben. Im Dialog „Warenposten bearbeiten“ erscheint zusätzlich ein ebenfalls rot hervorgehobener Hinweis beim Erreichen bzw. Unterschreiten der Warnmenge:



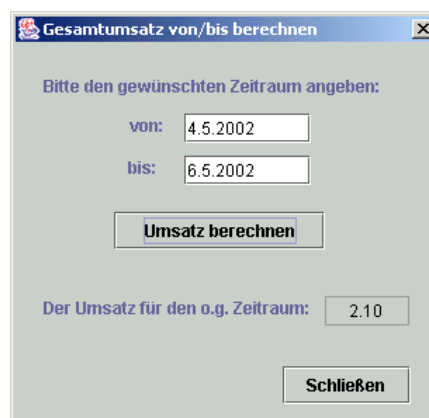
Eine weitere, kleinere Neuerung im Vergleich zur Version 1.0 schließlich stellt noch das Feature dar, dass in der Lagerverwaltung bei Warenposten mit abgelaufenen Waren, der Eintrag „verderblich“ unter „Warentyp“ rot hervorgehoben wird. Das Gleiche gilt für den obigen Dialog „Warenposten bearbeiten“. In ihm werden abgelaufene Waren durch ein rotes Haltbarkeitsdatum gekennzeichnet.

3.1.3 Funktionalität der Version 1.2

Kommen wir nun zur höchsten Ausbaustufe der Software Schulkiosk, zur Version 1.2. Sie wurde in Bezug auf Version 1.1 um ein Statistikmodul erweitert, in welchem sämtliche Verkaufsvorgänge erfasst werden:



Die Funktionalität dieses Statistikmoduls besteht neben der Darstellung aller Verkaufsvorgänge als Liste darin, für einen vorgegebenen Zeitraum den Gesamtumsatz des Schulkiosks, den von einem bestimmten Benutzer erwirtschafteten Umsatz oder den Umsatz eines bestimmten Artikels berechnen zu können. Die entsprechenden Dialoge sind die folgenden:



The dialog box titled "Benutzerumsatz von/bis berechnen" contains the following elements:

- A text input field for "Benutzername" with the value "admin".
- A label "Bitte den gewünschten Zeitraum angeben:" followed by two date input fields: "von: 4.5.2002" and "bis: 6.5.2002".
- A button labeled "Umsatz berechnen".
- A label "Der Umsatz für den o.g. Zeitraum:" followed by a text input field containing the value "0.30".
- A button labeled "Schließen".

The dialog box titled "Warenumsatz von/bis berechnen" contains the following elements:

- A text input field for "Warenbezeichnung" with the value "Banane".
- A label "Bitte den gewünschten Zeitraum angeben:" followed by two date input fields: "von: 4.5.2002" and "bis: 6.5.2002".
- A button labeled "Umsatz berechnen".
- A label "Der Umsatz für den o.g. Zeitraum:" followed by a text input field containing the value "0.80".
- A button labeled "Schließen".

Um alle erfassten Verkäufe bis zu einem bestimmten Datum zu löschen, steht folgender Dialog zur Verfügung:

The dialog box titled "Statistikdaten löschen bis" contains the following elements:

- A label "Bitte eingeben, bis zu welchem Datum die Statistik gelöscht werden soll:" followed by a date input field containing the value "6.5.2002".
- Two buttons: "Daten löschen" and "Abbrechen".

Damit ist nun insgesamt ein Überblick über den Funktionsumfang der Software Schulkiosk in ihren verschiedenen Ausbaustufen gegeben. Im Folgenden werden nun das UML-Design und einige Implementierungsdetails der Software Schulkiosk vorgestellt.

3.2 UML-Design und Implementierungsdetails der Software Schulkiosk

Da die verschiedenen Ausbaustufen der Software Schulkiosk aufeinander aufbauen, wird im Folgenden lediglich die höchste Ausbaustufe, die Version 1.2 vorgestellt:

Die Software insgesamt ist in Anlehnung an das MVC-Konzept²⁰ (MVC = Model/View/Controller) in zwei Schichten aufgebaut. Auf der einen Seite stehen die Fachklassen²¹ (Model) und auf der anderen die Klassen für die Benutzungsschnittstelle (View).

Als Entwicklungsumgebung für die Implementation der Fachklassen wurde das an der Universität Paderborn entwickelte *Fujaba* eingesetzt. Fujaba ist frei verfügbar (www.fujaba.de) und erlaubt das Erstellen von UML-konformen Klassendiagrammen und die graphische Implementierung von Methoden über sog. Aktivitätsdiagramme. Ein zusätzliches Highlight von Fujaba ist der integrierte graphische Debugger DOBS (Dynamic Object Browsing System), der eine graphische Darstellung der Objekte sowie der zwischen ihnen bestehenden Beziehungen zur Ausführungszeit der Software ermöglicht.

Da Fujaba im Rahmen des in der Einleitung bereits erwähnten Life³-Projekts im Informatikunterricht bereits recht erfolgreich eingesetzt wurde, bietet sie sich auch im Hinblick auf einen ersten 'Einsatz' der Software Schulkiosk in den am Projekt beteiligten Informatikkursen als geeignete Entwicklungsumgebung an.

Als Entwicklungsumgebung für die Klassen der Benutzungsschnittstelle fiel die Wahl auf *Forte for Java*. Forte ist ein Produkt der Firma SUN und in der Community Edition ebenfalls frei verfügbar (www.sun.com), es ermöglicht das relativ einfache Erzeugen von graphischen Oberflächen durch „Zusammenklicken“ und erzeugt dabei recht gut strukturierten, verständlichen Quelltext.

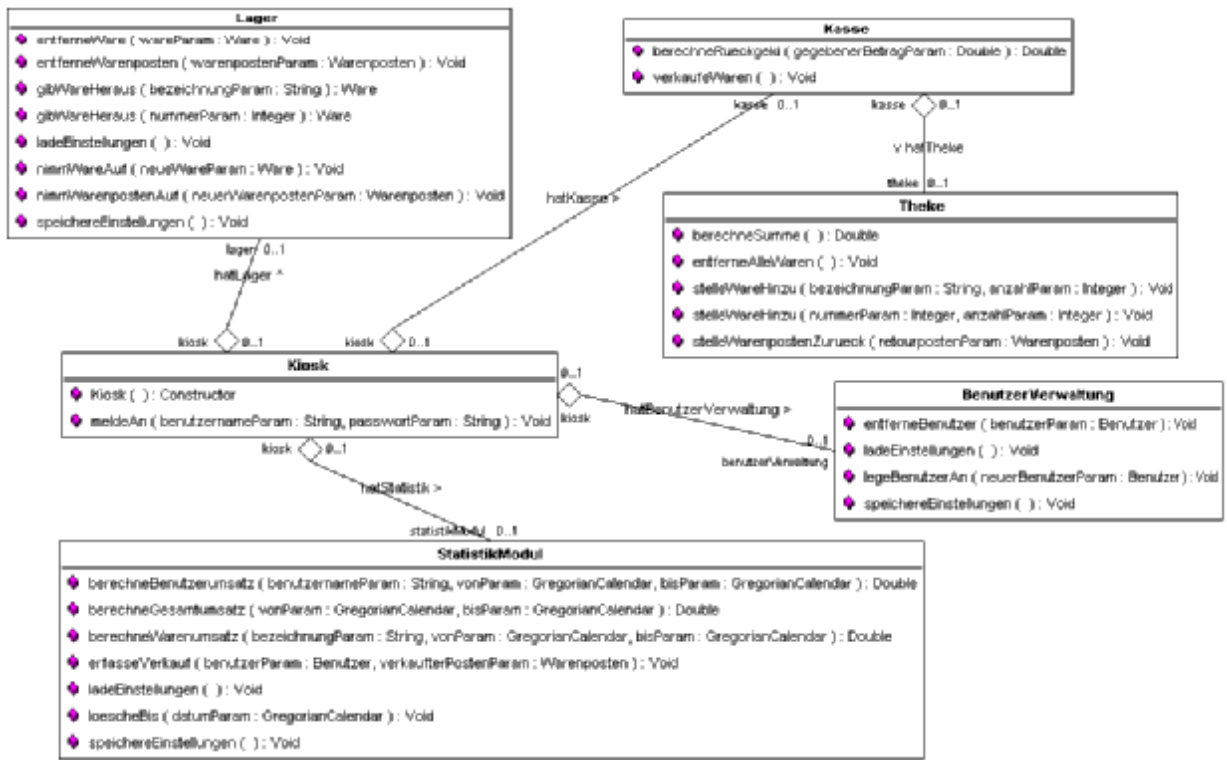
Für eine weitergehende Beschreibung der beiden Entwicklungsumgebungen Fujaba und Forte sei auf die oben angegebenen Webseiten verwiesen.

Zunächst aber erfolgt der Blick auf die Fachklassen.

²⁰ Vgl. z.B. Gamma, Erich: Entwurfsmuster, 5; oder vgl. Oestereich, Bernd: Objektorientierte Softwareentwicklung, 140

²¹ zum Begriff 'Fachklasse' siehe z.B. Oestereich, Bernd: Objektorientierte Softwareentwicklung, 91f.

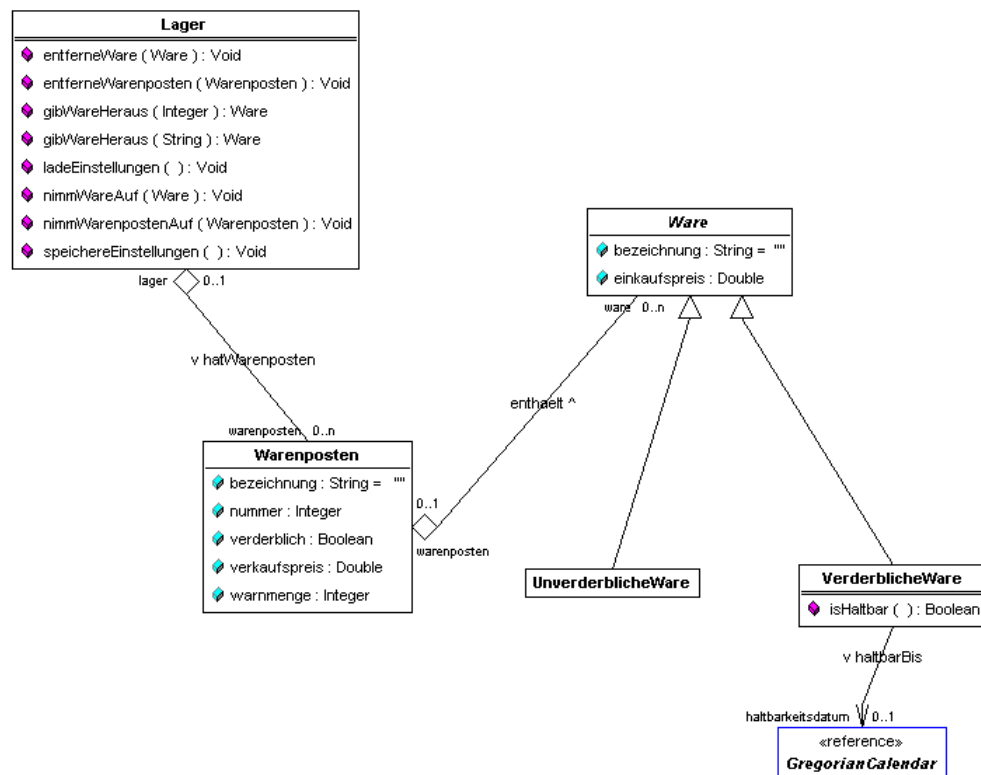
Das folgende Klassendiagramm zeigt die verschiedenen Module, aus denen sich die zentrale Klasse `Kiosk` zusammensetzt:



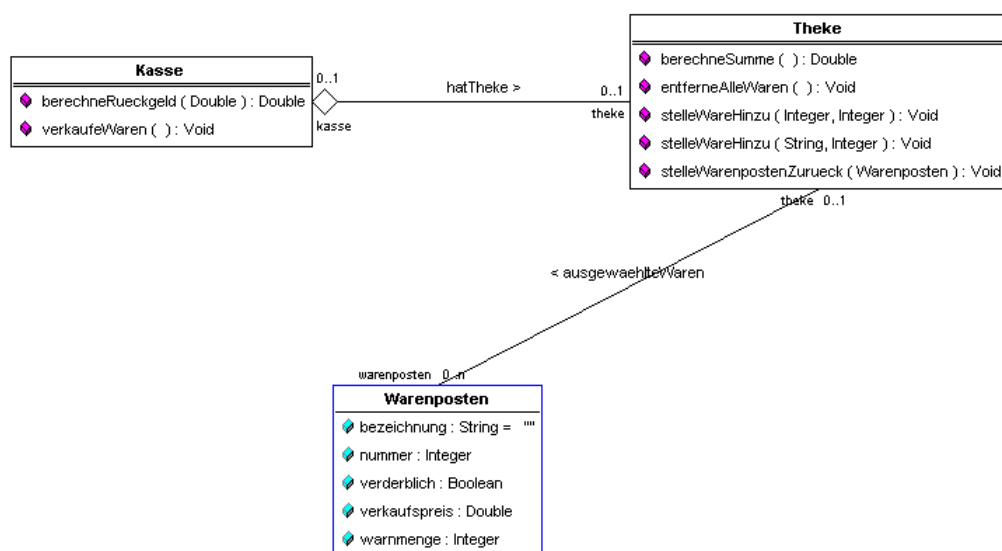
Die Klasse `Kiosk` ist eine Aggregation aus den Klassen `Lager`, `Kasse` (sie aggregiert ihrerseits die Klasse `Theke`), `BenutzerVerwaltung` und `StatistikModul`.

Wenden wir nun den Blick auf die vom `Kiosk` aggregierten Klassen im Einzelnen und beginnen mit der Klasse `Lager`, die eine 1:n-Aggregationsbeziehung zur Klasse `Warenposten` hat. Die Klasse `Warenposten` ihrerseits hat eine selbige zur Klasse `Ware`. `Ware` schließlich ist eine abstrakte Klasse, von der die beiden Klassen `UnverderblicheWare` und `VerderblicheWare` erben. `UnverderblicheWare` unterscheidet sich von `VerderblicheWare` durch die 1:1-Beziehung zur Klasse `GregorianCalendar`, dessen Rolle `haltbarkeitsdatum` ist. Diese 1:1-Beziehung ist also gleichzusetzen mit einem Attribut `haltbarkeitsdatum` vom Typ `GregorianCalendar`.

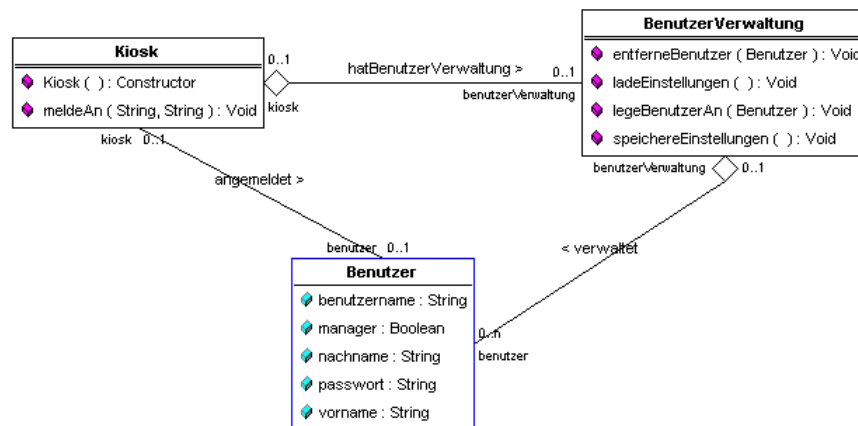
Im Folgenden ein Diagramm dieser Klassen:



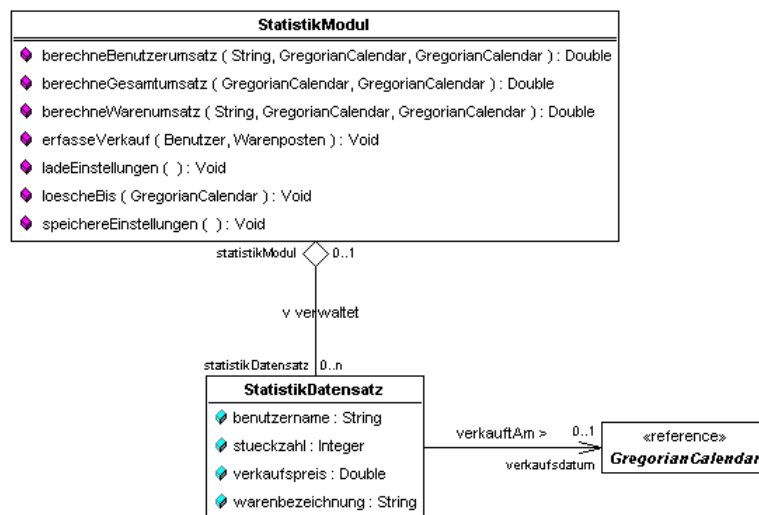
Als nächstes werfen wir einen Blick auf die Klasse *Kasse*, die ihrerseits eine Theke aggregiert. Die Theke wiederum kennt – bildlich gesprochen – die auf ihr liegenden Waren, was durch die 1:n-Assoziation „ausgewählteWaren“ zwischen den Klassen *Theke* und *Warenposten* zum Ausdruck gebracht wird:



Kommen wir nun zur Klasse `BenutzerVerwaltung`. Sie hat lediglich eine 1:n-Aggregation zur Klasse `Benutzer`. Und zwischen der Klasse `Kiosk` und der Klasse `Benutzer` besteht eine 1:1-Assoziation mit dem Namen „angemeldet“. Diese Assoziation repräsentiert das Faktum, dass sich ein `Benutzer` beim `Kiosk` anmelden kann und soll:

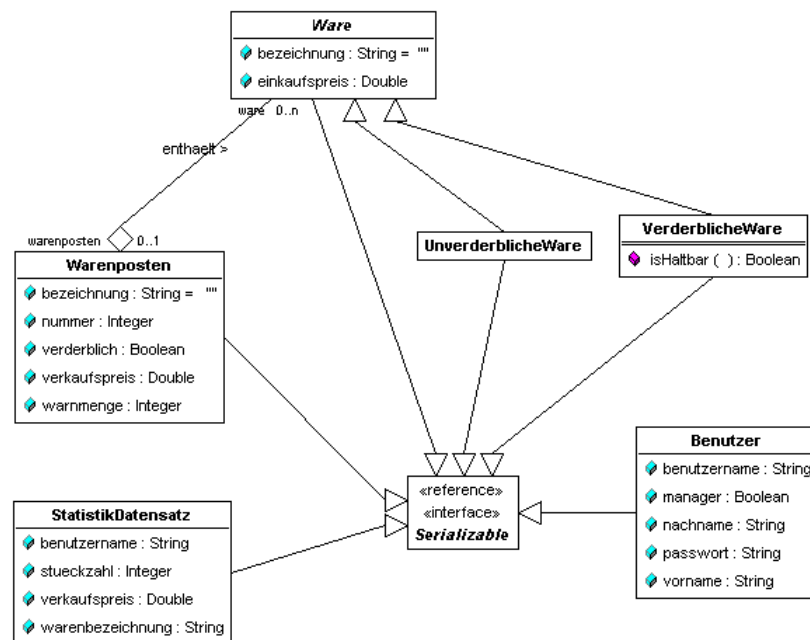


Bleibt noch die Klasse `StatistikModul`:

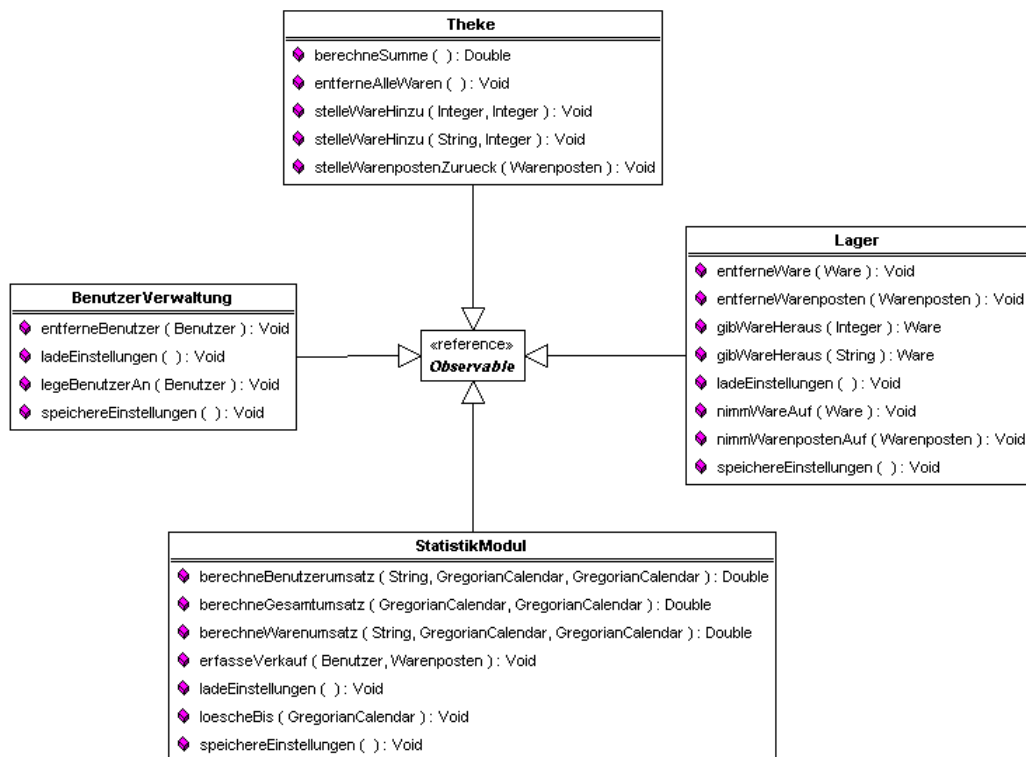


Zwischen der Klasse `StatistikModul` und der Klasse `StatistikDatensatz` existiert eine 1:n-Aggregation. Die Klasse `StatistikDatensatz` repräsentiert den erfassten Verkaufsvorgang einer Ware durch einen bestimmten `Benutzer`. Der Verkaufszeitpunkt ist dabei ausgedrückt durch die 1:1-Assoziation „verkauftAm“ zwischen `StatistikDatensatz` und `GregorianCalendar`.

Damit die Einstellungen des Lagers, des Statistikmoduls und der Benutzerverwaltung einfach über Objekt-Streams gespeichert werden können, ist es nun noch nötig, dass die Klassen aller Objekte, die gespeichert werden sollen das Interface `Serializable` implementieren. Das sind im einzelnen die Klassen `Ware`, `UnverderblicheWare`, `VerderblicheWare`, `Warenposten`, `Benutzer` und `StatistikDatensatz`:



Um die bereits angedeutete Trennung zwischen Fachklassen und den Klassen der Benutzungsschnittstelle (View) zu realisieren, wurde das in Kapitel 4.5 thematisierte Observer-Pattern verwendet. Dazu melden sich sog. Observer (Beobachter) bei solchen Objekten an, über deren Zustandsänderung sie informiert werden möchten. Dies kann ein View-Objekt sein, das z.B. eine Liste aller Warenposten des Lagers darstellt und natürlich benachrichtigt werden möchte, wenn beispielsweise ein neuer Warenposten zum Lager hinzukommt. Java bietet zur einfachen Umsetzung des Observer-Patterns das Interface `Observer` und die Klasse `Observable` an. Klassen, deren Objekte `Observer` benachrichtigen sollen, erben dazu von der Klasse `Observable`. Die Klassen der beobachtenden Objekte auf der View-Seite implementieren das Interface `Observer`. Aufschluss darüber, welche Fachklassen der Software Schulkiosk von der Klasse `Observable` erben, gibt das folgende Klassendiagramm:



Eine weitergehende Beschreibung der einzelnen Fachklassen der Software Schulkiosk im Detail würde den Rahmen dieser Arbeit sicherlich sprengen. Einige interessante Einzelheiten werden jedoch im Rahmen der 'didaktischen Fenster' im nachfolgenden Kapitel 4 erwähnt.

Bleiben noch die Klassen der Benutzungsschnittstelle. Sie befinden sich getrennt von den Fachklassen im Package `schulkiioskGUI`. Da auch hier eine detaillierte Beschreibung zu weit führen würde, mag folgende Tabelle Aufschluss über die Klassen geben:

<i>Klasse</i>	<i>Beschreibung</i>
AnmeldungDialog	Dialog zur Benutzeranmeldung am Kiosk
BenutzerBearbeitenDialog	Dialog zum Bearbeiten der Benutzerattribute
BenutzerHinzufuegenDialog	Dialog zum Erstellen und Hinzufügen eines neuen Benutzers
BenutzerlisteListCellRenderer	Renderer für die im ManagerFenster dargestellte Liste (JList) aller Benutzer
BenutzerlisteListModel	ListModel für die im ManagerFenster dargestellte Liste (JList) aller Benutzer

Klasse	Beschreibung
Buttonbelegung	Belegung eines Warenbuttons im KasseFenster (ab Version 1.1)
ButtonbelegungDialog	Dialog zur Belegung der Warenbuttons im KasseFenster, der die Möglichkeit bietet, jedem Warenbutton ein Icon und eine Warenbezeichnung zuzuweisen
ButtonbelegungsListeListCellRenderer	Renderer für die im ButtonbelegungDialog dargestellte Liste (JList) der Warenbuttons
KasseFenster	Version 1.0: rudimentär ausgestattete Oberfläche der Kasse, summiert die Preise eingebonnter Waren und stellt diese in einer Liste (JList) dar ab Version 1.1: Kassenoberfläche zusätzlich mit Warenbuttons zur schnellen Eingabe häufig verkaufter Waren, Storno-Funktion, Wechselgeldberechnung, Möglichkeit die Warenbuttons zu belegen, Verkauf-Rückgängig-Funktion
KioskFenster	Version 1.0: Auswahlfenster, um die Lagerverwaltung bzw. die Kasse zu starten oder den Kiosk zu beenden ab Version 1.1: Auswahlfenster, um den Manager oder die Kasse zu starten, das Passwort des angemeldeten Benutzers zu ändern oder den Kiosk zu beenden
LagerFenster	nur Version 1.0: Fenster, das die Lagerverwaltung repräsentiert, dazu eine Liste (JList) der im Lager enthaltenen Warenposten anzeigt und die Möglichkeit bietet, neue Warenposten hinzuzufügen, bestehende zu bearbeiten oder zu löschen sowie die Lagerverwaltung zu beenden
ManagerFenster	nur ab Version 1.1: ersetzt das LagerFenster der Version 1.0 und repräsentiert Lagerverwaltung, Benutzerverwaltung und Statistikmodul
PasswortAendernDialog	Dialog zum Ändern des Passworts des am Kiosk angemeldeten Benutzers
StatistikBenutzerUmsatzDialog	Dialog zum Berechnen des von einem Verkäufer/Benutzer erwirtschafteten Umsatzes in einem anzugebenden Zeitraum
StatistikDatenLoeschenBisDialog	Dialog zum Löschen aller Statistikdaten bis zu einem bestimmten Datum
StatistikGesamtumsatzDialog	Dialog zum Berechnen des Gesamtumsatzes in einem anzugebenden Zeitraum
StatistiklisteListCellRenderer	Renderer für die im ManagerFenster dargestellte Liste (JList) der Statistikdaten
StatistiklisteListModel	ListModel für die im ManagerFenster dargestellte Liste (JList) aller der Statistikdaten
StatistikWarenumsatzDialog	Dialog zum Berechnen des Umsatzes einer Ware in einem anzugebenden Zeitraum

Klasse	Beschreibung
ThekenlisteListCellRenderer	Renderer für die im KasseFenster dargestellte Liste (JList) ausgewählter Waren
ThekenlisteListModel	ListModel für die im KasseFenster dargestellte Liste (JList) ausgewählter Waren
UnverderblicheWareBearbeitenDialog	Dialog zum Bearbeiten der Attribute einer unverderblichen Ware
UnverderblicheWareHinzufuegenDialog	Dialog zum Erstellen und Hinzufügen einer neuen unverderblichen Ware
VerderblicheWareBearbeitenDialog	Dialog zum Bearbeiten der Attribute einer verderblichen Ware
VerderblicheWareHinzufuegenDialog	Dialog zum Erstellen und Hinzufügen einer neuen verderblichen Ware
WarenlisteListCellRenderer	Renderer für die im WarenpostenBearbeitenDialog dargestellte Liste (JList) enthaltener Waren, ab Version 1.1 wird von Waren, deren Haltbarkeitsdatum abgelaufen ist, dieses rot dargestellt
WarenlisteListModel	ListModel für die im WarenpostenBearbeitenDialog dargestellte Liste (JList) enthaltener Waren
WarenpostenAendernDialog	Dialog zum Ändern der Attribute eines existierenden Warenpostens, ab Version 1.1 zusätzlich das Attribut Warnmenge
WarenpostenBearbeitenDialog	Dialog zum Bearbeiten eines existierenden Warenpostens, der die Möglichkeit bietet, neue Waren hinzuzufügen, enthaltene Waren zu löschen, deren Attribute zu bearbeiten oder die Attribute des Warenpostens zu ändern, ab Version 1.1 wird zusätzlich angezeigt, ob die Warnmenge unterschritten wurde
WarenpostenHinzufuegenDialog	Dialog zum Erstellen und Hinzufügen eines neuen Warenpostens, aber Version 1.1 zusätzlich mit dem Attribut Warnmenge
WarenpostenlisteListCellRenderer	Renderer für die im ManagerFenster dargestellte Liste (JList) der im Lager enthaltenen Warenposten, ab Version 1.1 werden bei Warenposten, deren Warnmenge unterschritten ist oder die abgelaufene Waren enthalten, die entsprechenden Attribute rot dargestellt
WarenpostenlisteListModel	ListModel für die im ManagerFenster dargestellte Liste (JList) der im Lager enthaltenen Warenposten
Werkzeug	Klasse, die einige statische Methoden zum Konvertieren von Strings in double-, int-, long-Werte oder GregorianCalendar-Objekte und umgekehrt anbietet und zusätzlich noch Methoden die überprüfen ob sich ein String in die genannten Werte bzw. Objekte konvertieren lässt

4 'Didaktische Fenster' auf die Software Schulkiosk

In Form von sog. und bereits erwähnten 'didaktischen Fenstern' sollen in diesem Kapitel nun einige informatische Inhalte betrachtet werden, wie sie anhand der Software Schulkiosk im Rahmen der Dekonstruktion im Informatikunterricht thematisiert werden können. Diese didaktischen Fenster – wie in der Einleitung bereits beschrieben und begründet – übernehmen die Rolle der Konzeption einer auf der Software Schulkiosk basierenden Unterrichtseinheit, wie sie im Thema dieser Arbeit angekündigt ist.

Als erstes didaktisches Fenster soll nach der im letzten Kapitel erfolgten Vorstellung der Software Schulkiosk der Schluss von Funktionalität und Erprobung der Software auf das ihr zugrunde liegende objektorientierte Design erläutert werden.

4.1 Schluss von Funktionalität und Erprobung der Software Schulkiosk auf das ihr zugrunde liegende objektorientierte Design

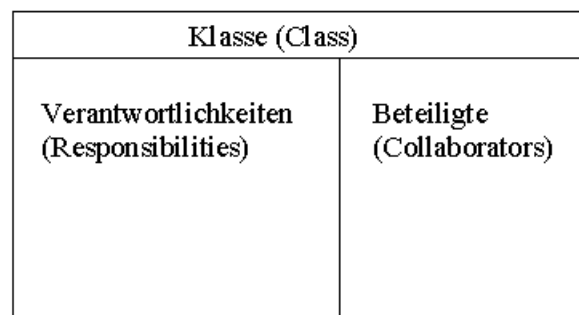
An eine erste Erkundungs- und Erprobungsphase der Software Schulkiosk durch die Schüler kann sich möglicherweise gut als erster Dekonstruktionsschritt der Schluss auf das der Software zugrunde liegende objektorientierte Design anbieten. Dies setzt natürlich auf Seiten der Schüler ein Basiswissen von objektorientierter Modellierung voraus. Insbesondere der Unterschied von Klasse und Objekt sollte verstanden, Assoziations- bzw. Aggregationsbeziehungen bekannt sein.

In gewissem Sinne ist die Aufgabenstellung damit für die Schüler eine Nachmodellierung der Fachklassen der Software Schulkiosk, an deren Beginn eine objektorientierte Analyse steht, deren Grundlage in diesem Fall eine fertige Software in einem bestimmten (gedachten) Einsatzkontext ist. Folglich bietet sich als Modellierungstechnik aufgrund ihrer Einfachheit eine Modellierung mit *CRC-Karten* an.

CRC-Karten werden von Oestereich beschrieben als „Karteikarten, auf denen die Klasse, ihre Verantwortlichkeiten und andere Beteiligte notiert werden. CRC

ist denn auch die Abkürzung für Class-Responsibilities-Collaborators (Klasse-Verantwortlichkeiten-Beteiligte).²² Verantwortlichkeiten werden folgendermaßen beschrieben: „Eine Klasse ist verantwortlich für das Wissen, über das ihre Objekte verfügen sollen (Attribute) und für die Operationen, die ihre Objekte ausführen müssen, um ihre Aufgaben und Ziele zu erreichen.“²³ Beteiligte schließlich sind „andere Klassen, zu denen Beziehungen existieren müssen, damit die Klasse ihre Aufgaben erfüllen kann. Nicht alle Verantwortlichkeiten kann eine Klasse aus eigener Kraft erfüllen, in vielen Fällen muß sie mit anderen Klassen kooperieren. Beteiligte sind also Kooperationspartner.“²⁴

Hier die Skizze einer CRC-Karte:



CRC-Karten eignen sich gut für Teamarbeit, ihr Einsatz wird von Oestereich im Rahmen der objektorientierten Analyse recht detailliert beschrieben.²⁵ Die Schüler könnten also in Gruppen das der Software Schulkiosk zugrundeliegende Modell mit Hilfe der CRC-Karten versuchen zu (nachzu-)modellieren. Dabei werden naturgemäß wahrscheinlich unterschiedliche Modelle als Ergebnis herauskommen. Dies aber kann und sollte im Unterricht thematisiert werden, laut Magenheim ist u.a. folgende Erkenntnis wesentliches Ziel unterrichtlichen Modellierens: „Systemgestaltung sollte auch im Unterricht als Prozess erkannt werden, der keinesfalls eindeutige Lösungen zu vorher definierten Systemanforderungen liefert und somit den scheinbar zweckrationalen Charakter von Informatiksystemen infrage stellt.“²⁶ Spätestens, wenn es darum geht, das der Software Schulkiosk tatsächlich zugrunde liegende Modell in Form des Klassendiagramms der Fachklassen in

²² Oestereich, Bernd: Objektorientierte Softwareentwicklung, 48

²³ Ebd., 48

²⁴ Ebd., 49

²⁵ Vgl. ebd., 152ff.

²⁶ Hampel, Magenheim, Schulte: Dekonstruktion von Informatiksystemen als Unterrichtsmethode, 153

Augenschein zu nehmen, werden die Schüler „vermutlich feststellen, dass andere Personen aufgrund divergierender Entscheidungen zu anderen Entwurfsergebnissen gekommen sind.“²⁷

Angedeutet sei an dieser Stelle noch die Möglichkeit, mittels einer noch zu erstellenden, natürlich fiktiven Dokumentation von Designentscheidungen den Entwurfsprozess dieser 'anderen Personen' nachvollziehen zu können.

Um Designentscheidungen und potentielle Konflikte geht es auch im nächsten Abschnitt.

27 Ebd., 154

4.2 Systemgestaltung als kooperativer Kommunikationsprozess

„Systeme zu modellieren und zu erkennen, dass ihre Gestaltung auf sorgfältiger Analyse sozialer Kommunikations- und Interaktionsstrukturen der mit dem System interagierenden Menschen und der technologischen Rahmenbedingungen ihres Handlungsfeldes beruht, dass Systemgestaltung infolgedessen nicht nur ein technischer sondern auch ein kooperativer Kommunikationsprozess ist, erscheint dann als eine Hauptaufgabe informatischer Bildung und einer systemorientierten Didaktik der Informatik.“²⁸

Offenheit in der Form, dass „einzelne Zusatzmodule der Software von den Lernenden selbst entwickelt und in die Software integriert werden“²⁹ können, war u.a. eine Anforderung an die Software Schulkiosk. Eine Möglichkeit nun, diese Eigenschaft und die in obigem Zitat geforderte Erkenntnis, nämlich Systemgestaltung als kooperativen Kommunikationsprozess zu begreifen, zu verknüpfen, soll im Folgenden umrissen werden:

In der Version 1.1 der Software Schulkiosk existiert noch kein Statistikmodul. Schüler und Lehrer könnten nun die Rollen bzw. Sichtweisen von Entwicklern, künftigen Benutzern und Auftraggebern einnehmen und die Anforderungen an ein Statistikmodul festlegen. Hierbei wird es höchstwahrscheinlich zu Interessenskollisionen kommen, da z.B. ein detailliertes Erfassen eines jeden Verkaufsvorgangs für den Auftraggeber, zum Beispiel den Hausmeister, der einen Kiosk betreibt, durchaus wünschenswert ist. Ein Mitarbeiter (z.B. Schüler) hingegen wird dies jedoch unter Umständen ablehnen, da er dies als Überwachung und Bruch eines Vertrauensverhältnisses empfinden könnte. Auch die Möglichkeit, Umsatz nach Mitarbeiter berechnen zu können, ist sicher nicht unproblematisch.

Ernsthaft relevant werden Designentscheidungen bzw. -konflikte, wenn der Transfer auf ein reales Informatiksystem, ein Warenwirtschaftssystem erfolgt. Hier nämlich spielen ohne Zweifel soziale, innerbetriebliche, rechtliche (insbesondere arbeits- und datenschutzrechtliche) Fragen bei der Entwicklung und Gestaltung von Informatiksystemen eine gewichtige Rolle.

Und all dies sollte und kann im Informatikunterricht an dieser Stelle thematisiert werden. Dazu sind geeignete Materialien (z.B. arbeits- oder

28 Ebd., 152f

29 Ebd., 150

datenschutzrechtliche Bestimmungen usw.) heranzuziehen. Auch kann sich gegebenenfalls eine Betriebserkundung anbieten, während der solche Fragen von kompetenter Seite und aus der Praxis heraus erörtert werden könnten.

Problematisieren lassen sich die gerade beschriebenen potentiellen Konflikte übrigens auch ausgehend vom Statistikmodul der Software Schulkiosk in der Version 1.2, das bei jedem Verkaufsvorgang unter anderem Verkäufer, Datum und Uhrzeit sowie den Umsatz erfasst und u.a. das Berechnen des Umsatzes für einen bestimmten Mitarbeiter vorsieht.

Natürlich ist „*Systemgestaltung als kooperativer Kommunikationsprozess*“ in diesem didaktischen Fenster noch längst nicht umfassend genug beschrieben und behandelt. Es versteht sich lediglich als Andeutung einer Möglichkeit, dieses Thema im Informatikunterricht zu problematisieren.

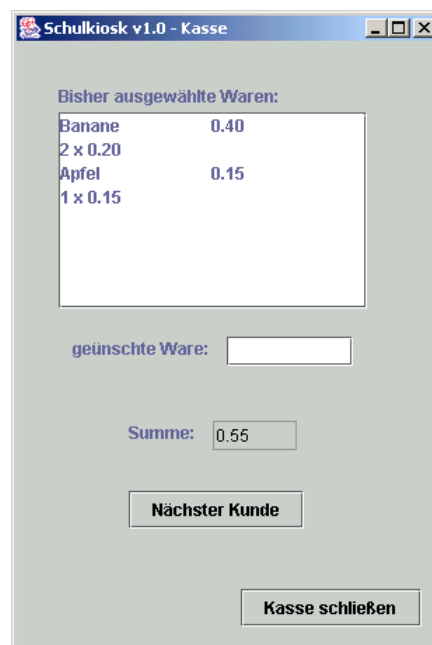
Was jedenfalls zu thematisieren wäre, beschreibt Magenheim so: „Subjektives Wissen und subjektive problembezogene Einstellungen (z.B. der Entwickler, Auftraggeber und Anwender) sind es auch, die Gestaltungsentscheidungen und die Nutzung von Informatiksystemen bestimmen. Diesen impliziten Prozessen nachzuspüren, die sich in der Software materialisiert haben, ist neben der formal-technischen Analyse von Informatiksystemen eine weitere Aufgabe der didaktischen Analyse einer systemorientierten Didaktik der Informatik. Dekonstruktion kann möglicherweise derartige Fragestellungen für den Informatikunterricht erschliessen.“³⁰

30 Magenheim: Informatiksystem und Dekonstruktion als didaktische Kategorien

4.3 Software-Ergonomie: Die Kasse der Version 1.0 – Beispiel einer unpraktikablen Benutzerschnittstelle

„Dekonstruktion bewegt sich [...] zwischen der Ebene der Formalismen und Regeln und der in der Phase der Modellierung beim Softwareentwurf und Design angelegten Funktionalität der Software im Kontext des Informatiksystems. Die funktionale Einbettung der Software in den Kontext sozialer Interaktionen des Informatiksystems erfolgt nicht zuletzt über die Benutzungsschnittstelle. Folglich muß Dekonstruktion auch die Medialität der Benutzungsschnittstelle berücksichtigen, um die in der Software typisiert angelegten Interaktionsschemata als Impulse für real ablaufende soziale Interaktionsprozesse im Informatiksystem sichtbar zu machen.“³¹

Als Ansatzpunkt, um das hier angedeutete Problem, das u.a. auch unter das Stichwort *Software-Ergonomie* eingeordnet werden kann, im Unterricht zu problematisieren, kann die Benutzeroberfläche der Kasse in der Version 1.0 der Software Schulkiosk herangezogen werden:

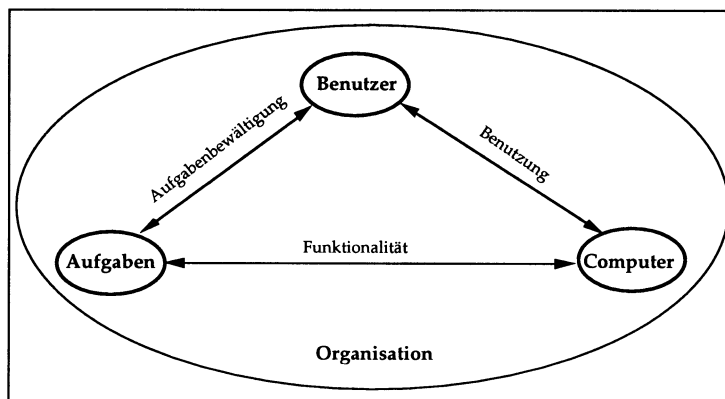


Als Elemente und Kriterien, die bei der Evaluation von Software unter dem Gesichtspunkt Software-Ergonomie zu berücksichtigen sind, nennen Oppermann und Reiterer unter der Überschrift „*Software-ergonomische Evaluation*“ in „*Einführung in die Software-Ergonomie*“ die Punkte *Aufgabenbewältigung*, *Benutzung* und *Funktionalität*.³² Sie stehen, wie die

31 Magenheim: Informatiksystem und Dekonstruktion als didaktische Kategorie

32 Vgl. Oppermann, Reinhard; Reiterer, Harald: Software-ergonomische Evaluation, 335-342

folgende Darstellung³³ zeigt, für die Beziehungen zwischen Benutzer, Computer und Aufgaben:



Elemente und Beziehungen

Die Beziehung *Aufgabenbewältigung* meint hierbei, „inwieweit der Benutzer in der Lage ist, die ihm anvertrauten Aufgaben zu erfüllen und ob er diese als „menschengerecht“ empfindet. [...] Bei der software-ergonomischen Bewertung des EDV-Systems interessiert vor allem, inwieweit die Aufgabenbewältigung durch das EDV-System unterstützt oder behindert wird[...]“³⁴. Die Beziehung *Benutzung* beschreibt, „mit welchem Interaktionsaufwand die Bedienung des EDV-Systems für den Benutzer verbunden ist. [...] Die ergonomische Qualität der Benutzung ist zentraler Bewertungsgegenstand der software-ergonomischen Evaluation.“³⁵ Unter *Funktionalität* schließlich fällt, inwieweit das EDV-System aufgabenrelevant und aufgabenangemessen ist, also wie groß das Ausmaß der Unterstützung der Aufgaben durch das EDV-System ist, ob es die bestehenden Arbeitsaufgaben hinreichend genau abbilden kann oder ob es diese entstellt und verkompliziert.³⁶

Nach dieser Erläuterung wenden wir uns nun wieder der bereits dargestellten Benutzeroberfläche der Kasse in der Software Schulkiosk zu: Stellt man sich einen realen Verkaufsvorgang in einem Schulkiosk vor, so vermag sie lediglich folgendes zu leisten: Sie stellt die bisher durch den Kunden ausgewählten und vom Verkäufer/Benutzer eingebonnten Waren in einer Liste dar und zeigt deren Summe an.

³³ Darstellung in ebd., 337

³⁴ Ebd., 337f.

³⁵ Ebd., 338

³⁶ Vgl. ebd., 338

Zu den Unzulänglichkeiten, die beim Einsatz in der Praxis jedoch auftreten werden und die im Informatikunterricht z.B. durch die Benutzung der Software von den Schülern zu erarbeiten sind, gehören sicherlich einerseits, dass der Fall, dass eine eingebonnte Ware vom Kunden nun doch nicht gewünscht ist, nicht berücksichtigt ist. Die Oberfläche stellt hierzu keinerlei Optionen zur Verfügung wie beispielsweise eine Storno-Funktion. Zudem bietet die so gestaltete Kasse keine Möglichkeit, das Wechselgeld zu berechnen, eine auf jeden Fall zu leistende Aufgabe eines typischen Verkaufsvorgangs. Wünschenswert könnte es außerdem sein, häufig verkaufte Waren, schneller eingeben zu können, beispielsweise über einen Button statt über die (mühsame) Eingabe von Nummer bzw. Bezeichnung der Ware.

Ausgehend von diesem Beispiel lassen sich nun im Informatikunterricht – in Analogie zu den oben genannten – Kriterien zur Bewertung von Software entwickeln. Darüber hinaus kann es sich auch anbieten, weiter in die Themen *Design von Benutzungsschnittstellen* und damit letztlich *HCI (Human Computer Interaction)* einzusteigen, indem beispielsweise Sekundärtexte herangezogen und bearbeitet werden, die dieses Thema behandeln. Transfermöglichkeiten hin zu einem realen Informatiksystem, nämlich einem Warenwirtschaftssystem, sind zudem gegeben, d.h. ein solches ließe sich auch und sogar unter Einbeziehung seines sozialen Umfeldes erkunden und würde somit das Verständnis von Informatiksystemen als sozio-technische Systeme weiter unterstützen helfen.³⁷

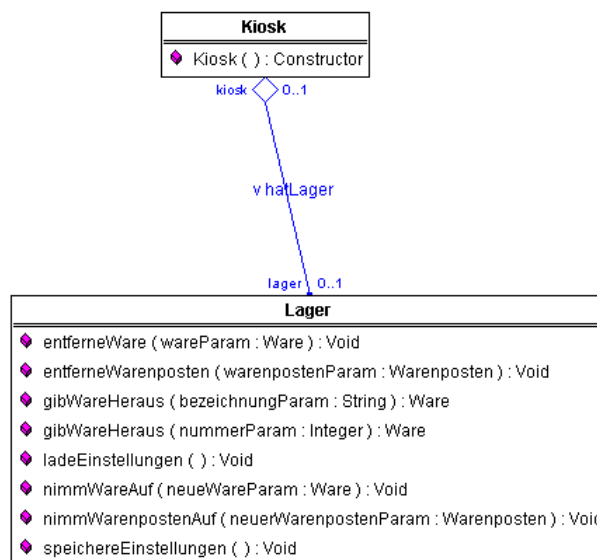
³⁷ Vgl. Hampel, Magenheimer, Schulte: Dekonstruktion von Informatiksystemen als Unterrichtsmethode, 153 u. 154

4.4 Bidirektionale Assoziationen bzw. -Aggregationen im Quelltext

Assoziation (Bekanntschaft bzw. „kennt“-Beziehung) und Aggregation („hat“-Beziehung) sind, was ihre Implementierung betrifft, häufig identisch, d.h. letztendlich „wird die Unterscheidung zwischen Bekanntschaft und Aggregation mehr durch ihre Verwendung und nicht durch die Nutzung expliziter Sprachmöglichkeiten getroffen.“³⁸ – Wie aber lassen sich bidirektionale 1:1- und 1:n-*Beziehungen*³⁹ zwischen Objekten verschiedener Klassen in Java realisieren? Das für Modellierung und Implementierung der Fachklassen der Software Schulkiosk verwendete Entwicklungswerkzeug Fujaba erzeugt aus einem vom Entwickler in UML-Notation angegebenen Klassendiagramm automatisch den zugehörigen Java-Quelltext. Es bietet sich daher an, sich anhand des von Fujaba erzeugten Quelltextes anzuschauen, wie zunächst einmal bidirektionale 1:1-Beziehungen sich in Java umsetzen lassen.

4.4.1 Bidirektionale 1:1-Assoziationen bzw. -Aggregationen

Als Beispiel soll hier die folgende 1:1-Beziehung analysiert werden:



³⁸ Gamma, Erich: Entwurfsmuster, 32

³⁹ Da, wie beschrieben, bezüglich der Umsetzung von Aggregationen und Assoziationen so gut wie kein Unterschied besteht, soll im Folgenden nur noch von Beziehungen die Rede sein, d.h. der Begriff 'Beziehung' steht stellvertretend für Assoziation als auch für Aggregation.

Ein Objekt der Klasse `Kiosk` kennt ein Objekt der Klasse `Lager`. Und das Objekt der Klasse `Lager` kennt umgekehrt das Objekt der Klasse `Kiosk`, zu dem es gehört. - Wie stellt sich dies im Quelltext dar?

```

56  /**
57  * <pre>
58  *      0..1 hatLager  0..1 /\
59  * Lager -----< > Kiosk
60  *      lager        kiosk  \/
61  * </pre>
62  */
63  private Lager lager;
64

```

1:1 INS

```

714 /**
715 * <pre>
716 *      /\  0..1 hatLager  0..1
717 * Kiosk < >----- Lager
718 *      \/  kiosk        lager
719 * </pre>
720 */
721 private Kiosk kiosk;
722

```

1:1 INS

Kiosk Lager

Nun, wie man sieht, verfügt die Klasse `Kiosk` über ein Attribut vom Typ `Lager` und die Klasse `Lager` umgekehrt über ein Attribut vom Typ `Kiosk`. Für beide Attribute existieren `get-` und `set-`Methoden. Die `get-`Methoden machen nichts anderes, als das jeweilige Objekt, zu dem eine Beziehung besteht, zurückzugeben. Ein Blick z.B. in die Methode `getLager()` der Klasse `Kiosk` zeigt es:

```

91  /**
92  * UMLMethod: '+ getLager (): Lager'.
93  */
94  public Lager getLager ()
95  {
96      return this.lager;
97  }
98
99

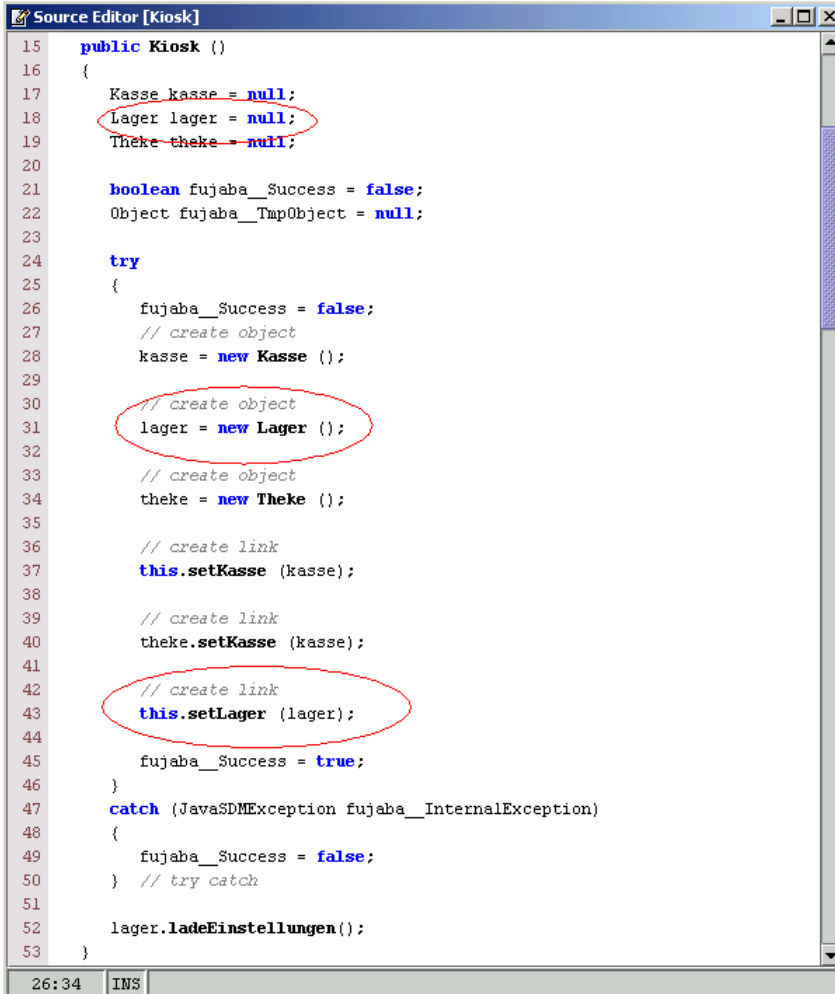
```

91:7 INS

Kiosk Lager

Interessanter ist es sicherlich, sich anzuschauen, wie eigentlich die

bidirektionale Beziehung zwischen einem Objekt der Klasse `Kiosk` und einem Objekt der Klasse `Lager` aufgebaut wird. Dazu empfiehlt sich ein Blick in den Konstruktor der Klasse `Kiosk`:



```
15 public Kiosk ()
16 {
17     Kasse kasse = null;
18     Lager lager = null;
19     Theke theke = null;
20
21     boolean fujaba_Success = false;
22     Object fujaba_TmpObject = null;
23
24     try
25     {
26         fujaba_Success = false;
27         // create object
28         kasse = new Kasse ();
29
30         // create object
31         lager = new Lager ();
32
33         // create object
34         theke = new Theke ();
35
36         // create link
37         this.setKasse (kasse);
38
39         // create link
40         theke.setKasse (kasse);
41
42         // create link
43         this.setLager (lager);
44
45         fujaba_Success = true;
46     }
47     catch (JavaSDMException fujaba_InternalException)
48     {
49         fujaba_Success = false;
50     } // try catch
51
52     lager.ladeEinstellungen();
53 }
```

Hier wird zunächst eine lokale Variable `lager` für ein Objekt der Klasse `Lager` deklariert. Anschließend wird diese Variablen durch `lager = new Lager()` die Referenz auf ein neu erzeugtes Objekt der Klasse `Lager` zugewiesen. Der eigentliche Aufbau der bidirektionalen Beziehung geschieht jedoch erst durch den Aufruf der Methode `this.setLager(lager)` auf dem `Kiosk`-Objekt.

Um zu begreifen, was die Methode `setLager(Lager value)` der Klasse `Kiosk` leistet, bietet sich eine Fallunterscheidung an, wobei es sich empfiehlt, jeden einzelnen Fall anhand des folgenden Quelltextes nachzuvollziehen:

```

Source Editor [Kiosk]
68 public boolean setLager (Lager value)
69 {
70     boolean changed = false;
71     if (this.lager != value)
72     {
73         if (this.lager != null)
74         {
75             Lager oldValue = this.lager;
76             this.lager = null;
77             oldValue.setKiosk (null);
78         }
79         this.lager = value;
80         if (value != null)
81         {
82             value.setKiosk (this);
83         }
84         changed = true;
85     }
86     return changed;
87 }
88 }
26:34 INS

```

1. Es besteht noch keine Beziehung zwischen dem `Kiosk`-Objekt und einem anderen `Lager`-Objekt, und der Parameter `value` ist nicht `null`:
 Nach Aufruf der Methode ist die bidirektionale Beziehung zwischen dem `Kiosk` und dem als Parameter übergebenen `Lager`-Objekt aufgebaut. Der Rückgabewert der Methode ist `true`. Er zeigt an, ob sich die Beziehung des `Kiosk`-Objekts zu einem `Lager`-Objekt geändert hat, sei es, weil sie neu aufgebaut, gelöscht oder geändert wurde.
2. Es besteht bereits eine Beziehung zwischen dem `Kiosk`-Objekt und einem anderen `Lager`-Objekt, und der Parameter `value` ist nicht `null`:
 Nach Aufruf der Methode ist die bidirektionale Beziehung zwischen dem `Kiosk` und dem als Parameter übergebenen `Lager`-Objekt aufgebaut. Die vorher bestehende bidirektionale Beziehung zu einem anderen Objekt der Klasse `Lager` ist gelöscht, d.h. weder der `Kiosk` kennt es, noch kennt es den `Kiosk`. Der Rückgabewert der Methode ist `true`.
3. Es besteht noch keine Beziehung zwischen dem `Kiosk`-Objekt und einem anderen `Lager`-Objekt, und der Parameter `value` ist `null`:
 Es ändert sich nichts. Der Rückgabewert ist `false`.
4. Es besteht bereits eine Beziehung zwischen dem `Kiosk`-Objekt und einem anderen `Lager`-Objekt, und der Parameter `value` ist `null`:
 Die Methode bewirkt das Löschen der bestehenden bidirektionalen Beziehung zu dem anderen `Lager`-Objekt. Der Rückgabewert ist `true`.

5. Es besteht bereits eine Beziehung zwischen dem `Kiosk`-Objekt und dem als Parameter übergebenen `Lager`-Objekt:

Es ändert sich nichts, die bestehende Beziehung bleibt bestehen. Der Rückgabewert der Methode ist `false`.

In der Methode `setLager(Lager value)` der Klasse `Kiosk` – die Methode `setKiosk(Kiosk value)` der Klasse `Lager` besitzt einen identischen Aufbau – passiert also Folgendes: Für das als Parameter übergebene Objekt der Klasse `Lager` (kann auch `null` sein!) wird zunächst überprüft, ob schon eine Beziehung zu ihm besteht, d.h. ob das Attribut `this.lager` und das als Parameter übergebene `Lager`-Objekt identisch sind, bzw beide `null` sind. Ist dies der Fall, besteht die Beziehung zwischen den beiden ja bereits, oder beide sind `null`, und es muss nichts weiter unternommen werden. Die Methode signalisiert das Auftreten dieses Falles, also wenn der Aufruf der Methode keine Beziehung verändert hat, schließlich durch den Rückgabewert, der dann natürlich `false` ist.

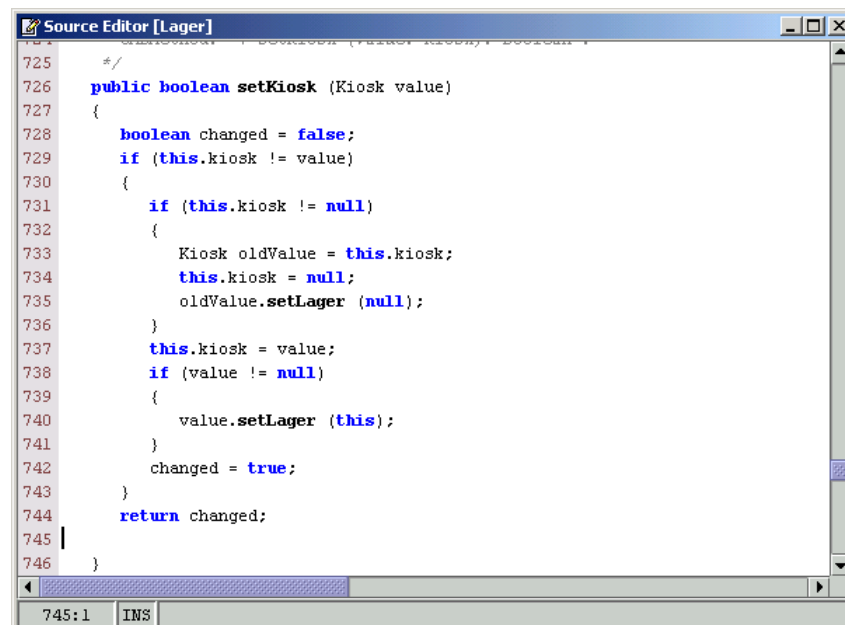
Besteht noch keine Beziehung zu dem als Parameter übergebenen `Lager`-Objekt, so wird als nächstes überprüft, ob bereits eine Beziehung zu einem anderen `Lager`-Objekt besteht. Ist dies nicht der Fall, so wird zunächst die Beziehung in Richtung des als Parameter übergebenen `Lager`-Objekts durch die Zuweisung `this.lager = value` etabliert. Anschließend bewirkt der Aufruf der Methode `setKiosk(this)` auf dem `Lager`-Objekt den Aufbau der Beziehung in umgekehrter Richtung. Die bidirektionale Beziehung ist also hergestellt.

Besteht hingegen beim Methodenaufruf schon eine Beziehung zu einem anderen `Lager`-Objekt, ist also die Bedingung (`this.lager != null`) erfüllt, so wird zunächst die bestehende Beziehung zu dem anderen `Lager`-Objekt gelöscht, indem dem Attribut `this.lager = null` zugewiesen und auf dem anderen `Lager`-Objekt die Methode `setKiosk(null)` aufgerufen wird, wodurch die Beziehung auch in Richtung vom anderen `Lager`-Objekt zum `Kiosk` gelöscht wird.

Erst dann – wenn der Parameter `value` nicht `null` ist – wird durch `this.lager = value` die Beziehung in Richtung des als Parameter

übergebenen Lager-Objekts aufgebaut und durch Aufruf der Methode `value.setKiosk(this)` auf dem Lager-Objekt auch in der umgekehrten Richtung. War der Parameter `value` hingegen `null`, so hatte dies lediglich ein Löschen der alten Beziehung zur Folge. In jedem Fall signalisiert die Methode über ihren Rückgabewert, falls sich an der bestehenden Beziehungen irgendetwas geändert hat. Dieser ist dann natürlich `true`.

Die bereits erwähnte Methode `setKiosk(Kiosk value)` der Klasse `Lager` ist, wie gesagt, identisch aufgebaut, was abschließend ein Blick auf ihren Quelltext zeigt:



```
725  /*
726  public boolean setKiosk (Kiosk value)
727  {
728      boolean changed = false;
729      if (this.kiosk != value)
730      {
731          if (this.kiosk != null)
732          {
733              Kiosk oldValue = this.kiosk;
734              this.kiosk = null;
735              oldValue.setLager (null);
736          }
737          this.kiosk = value;
738          if (value != null)
739          {
740              value.setLager (this);
741          }
742          changed = true;
743      }
744      return changed;
745  }
746  }
```

4.4.2 Bidirektionale 1:n-Assoziationen bzw. -Aggregationen

Bidirektionale 1:n-Beziehungen zwischen einem Objekt der Klasse A und mehreren Objekten der Klasse B sind dadurch gekennzeichnet, dass ein Objekt einer Klasse A mehrere Objekte einer anderen Klasse B kennt und umgekehrt alle Objekte dieser anderen Klasse B das eine Objekt der Klasse A.

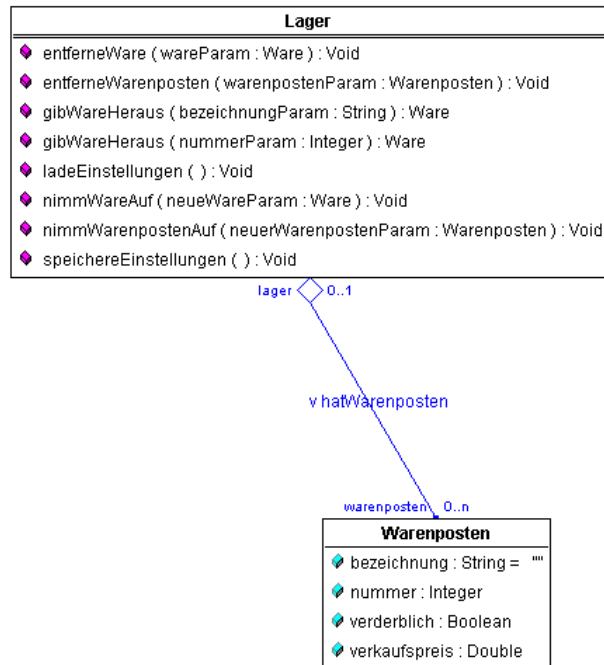
Zum Verwalten dieser Beziehungen wird auf Seiten der Klasse A häufig eine sog. Containerklasse (auch Sammlung/Collection genannt) verwendet. Hierbei handelt es sich um „zumeist in Standardklassenbibliotheken definierte Klassen, denen gemeinsam ist, daß sie eine Menge von Objekten ansammeln und verwalten können. [...] Sie verfügen daher alle über Operationen zum Anfügen und Entfernen von Objekten, zum Prüfen, ob ein gegebenes Objekt in der Menge enthalten ist und zum Ermitteln, wie viele Objekte momentan in der Menge enthalten sind.“⁴⁰

Fujaba verwendet standardmäßig zum Verwalten von Beziehungen die eigene Klasse `FHashSet`, die im Package `de.uni_paderborn.tools.fca` enthalten ist, und von den Methoden her weitgehend der Klasse `HashSet` im Package `java.util` entspricht, von dieser jedoch nicht erbt. Die wesentlichen Methoden von `FHashSet` sind die folgenden:

<i>Methodenname</i>	<i>Rückgabewert</i>	<i>Beschreibung</i>
<code>add(Object objekt)</code>	<code>boolean</code>	fügt das als Parameter übergebene Objekt hinzu
<code>contains(Object objekt)</code>	<code>boolean</code>	überprüft, ob das als Parameter übergebene Objekt, bereits im <code>FHashSet</code> enthalten ist
<code>iterator()</code>	<code>java.util.Iterator</code>	gibt ein Iterator-Objekt, eine Aufzählung der enthaltenen Objekte zurück
<code>size()</code>	<code>int</code>	gibt die Anzahl der im <code>FHashSet</code> enthaltenen Objekte zurück
<code>remove(Object objekt)</code>	<code>boolean</code>	entfernt das als Parameter übergebene Objekt aus dem <code>FHashSet</code>

Wie sich nun bidirektionale 1:n-Beziehungen in Java realisieren lassen, soll ebenfalls ein Blick in den von der Entwicklungsumgebung Fujaba generierten Quelltext zeigen. Als Beispiel dient die Aggregation „hatWarenposten“ zwischen den Klassen `Lager` und `Warenposten`, wie sie auf der nächsten Seite als Klassendiagramm dargestellt ist:

⁴⁰ Oestereich: Objektorientierte Softwareentwicklung, 57



Diese 1:n-Beziehung wird auf der Seite der Klasse `Lager` über ein Attribut der gerade beschriebenen Klasse `FHashSet` realisiert.

```

Source Editor [Lager]
758 /**
759  * <pre>
760  *      0..n hatWarenposten 0..1 /\
761  * Warenposten -----< > Lager
762  *      warenposten      lager  \/
763  * </pre>
764  */
765 private FHashSet warenposten;
766
789:1 INS
  
```

Zusätzlich werden von Fujaba in der Klasse `Lager` automatisch die Methoden

- `hasInWarenposten(Warenposten value)`
- `iteratorOfWarenposten()`
- `sizeofWarenposten()`
- `addToWarenposten(Warenposten value)`
- `removeFromWarenposten(Warenposten value)`
- `removeAllFromWarenposten()` generiert.

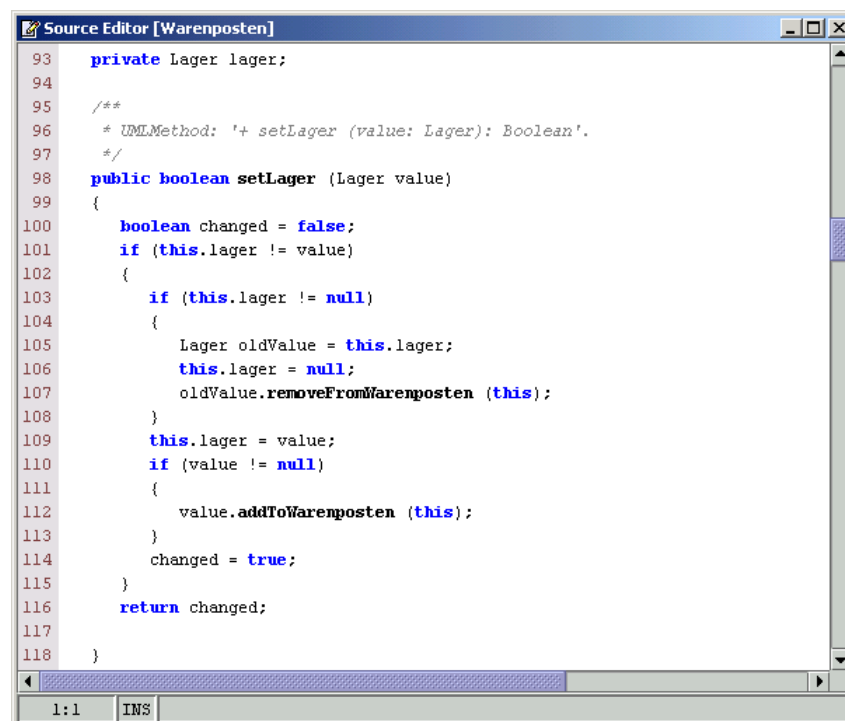
Dabei dient die Methode `addToWarenposten(Warenposten value)` dazu, eine bidirektionale Beziehung zu dem als Parameter übergebenen Objekt der Klasse `Warenposten` aufzubauen. Sie entspricht daher in gewisser Weise den im vorangegangenen Abschnitt beschriebenen `set`-Methoden der Klassen

Kiosk und Lager. Zum Löschen einer Beziehung jedoch benötigt man zusätzlich noch die Methode `removeFromWarenposten(Warenposten value)`, welche die Beziehung zu dem als Parameter übergebenen Objekt der Klasse `Warenposten` löscht. Sollen die Beziehungen eines Objektes der Klasse `Lager` zu allen Objekten der Klasse `Warenposten` auf einmal gelöscht werden, genügt ein Aufruf der Methode `removeAllFromWarenposten()`.

Die Rolle der `get`-Methode, wie es sie bei 1:1-Beziehungen gibt, übernimmt hier die Methode `iteratorOfWarenposten()`. Rückgabewert ist ein `Iterator`-Objekt⁴¹, also eine Aufzählung über alle Objekte der Klasse `Warenposten`, zu denen eine Beziehung besteht.

Die Methode `hasInWarenposten(Warenposten value)` überprüft, ob eine Beziehung zu dem als Parameter übergebenen Objekt der Klasse `Warenposten` besteht, und die Methode `sizeOfWarenposten()` gibt zurück, zu wie vielen Objekten der Klasse `Warenposten` eine Beziehung besteht.

Auf Seite der Klasse `Warenposten` schließlich ist die Realisierung der 1:n-Beziehung zwischen `Lager` und `Warenposten` fast identisch mit der im vorangegangenen Abschnitt beschriebenen bidirektionalen 1:1-Beziehung:



```

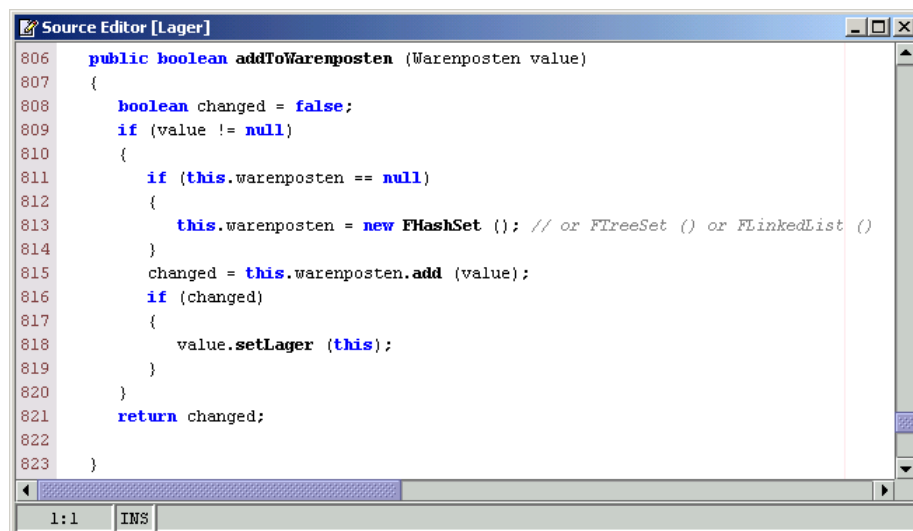
Source Editor [Warenposten]
93 private Lager lager;
94
95 /**
96  * UMLMethod: '+ setLager (value: Lager): Boolean'.
97  */
98 public boolean setLager (Lager value)
99 {
100     boolean changed = false;
101     if (this.lager != value)
102     {
103         if (this.lager != null)
104         {
105             Lager oldValue = this.lager;
106             this.lager = null;
107             oldValue.removeFromWarenposten (this);
108         }
109         this.lager = value;
110         if (value != null)
111         {
112             value.addToWarenposten (this);
113         }
114         changed = true;
115     }
116     return changed;
117 }
118 }
  
```

Die Beziehung in Richtung `Lager` erfolgt, wie in obigem Quelltext zu sehen ist,

⁴¹ zur Erläuterung des Begriffes „Iterator“ und des dahinter stehenden Musters siehe auch Kapitel 4.6

über ein Attribut der Klasse Lager. Die set-Methode ist daher im Wesentlichen identisch mit der bei zweiseitigen 1:1-Beziehungen, außer, dass nun in der Klasse Lager keine set-Methode zur Verfügung steht, sondern stattdessen die Methoden `removeFromWarenposten(Warenposten value)` und `addToWarenposten(Warenposten value)` zum Aufbau und zum Löschen der Beziehung benutzt werden müssen.

Doch auch die `addToWarenposten`-Methode der Klasse Lager ähnelt von ihrer Struktur her der im vorangegangenen Abschnitt beschriebenen set-Methode, wie ihr Quelltext zeigt:



```
Source Editor [Lager]
806 public boolean addToWarenposten (Warenposten value)
807 {
808     boolean changed = false;
809     if (value != null)
810     {
811         if (this.warenposten == null)
812         {
813             this.warenposten = new FHashSet (); // or FTreeSet () or FLinkedList ()
814         }
815         changed = this.warenposten.add (value);
816         if (changed)
817         {
818             value.setLager (this);
819         }
820     }
821     return changed;
822 }
823 }
```

In ihr wird zunächst überprüft, ob nicht `null` als Parameter übergeben wurde, dann nämlich ändert sich natürlich nichts. Anschließend wird – falls noch nicht geschehen – das Attribut `this.warenposten` vom Typ `FHashSet` mit einem neu erzeugten `FHashSet`-Objekt initialisiert.

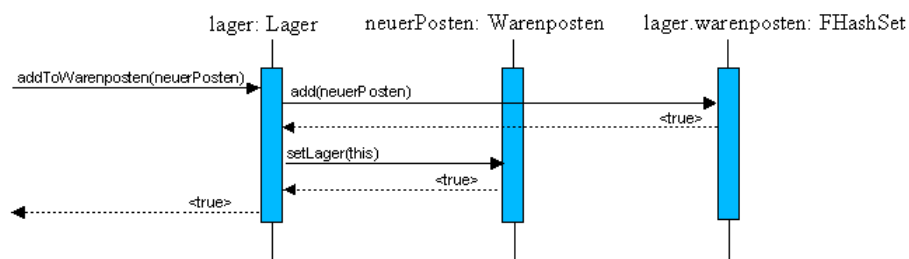
Dann schließlich wird durch `changed = this.warenposten.add(value)` der als Parameter übergebene `Warenposten` dem `FHashSet` hinzugefügt, also die Beziehung vom Lager in Richtung zum `Warenposten` aufgebaut. Die `boolean`-Variable `changed` hält dabei fest, ob der `Warenposten` tatsächlich hinzugefügt wurde bzw. werden konnte. Denn ist ein Objekt bereits im `FHashSet` enthalten, so gibt die `add`-Methode `false` zurück, in unserem Fall hätte die Beziehung zwischen dem Lager und dem als Parameter übergebenen `Warenposten` also bereits bestanden.

Bestand sie noch nicht, wird durch `value.setLager(this)` die Beziehung

auch in Richtung vom Warenposten zum Lager etabliert.

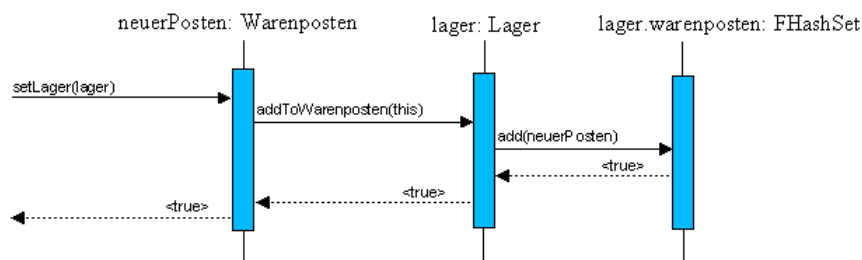
In jedem Fall gibt die `addToWare`-Methode über den `boolean`-Wert zurück, ob die Beziehung zum als Parameter übergebenen `Warenposten`-Objekt neu aufgebaut werden konnte.

Im Folgenden zur besseren Veranschaulichung noch als Sequenzdiagramme die verschiedenen Möglichkeiten, Beziehungen zwischen einem Objekt der Klasse `Lager` und einem Objekt der Klasse `Warenposten` aufzubauen bzw. zu löschen:

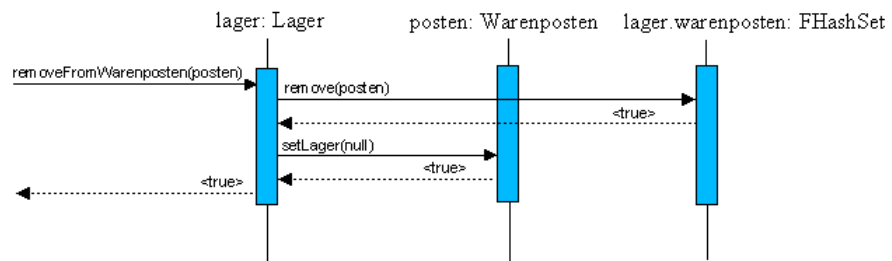


Hier wurde auf dem Objekt `lager` der Klasse `Lager` die Methode `addToWarenposten(neuerWarenposten)` aufgerufen, um eine bidirektionale Beziehung zwischen den beiden Objekten herzustellen.

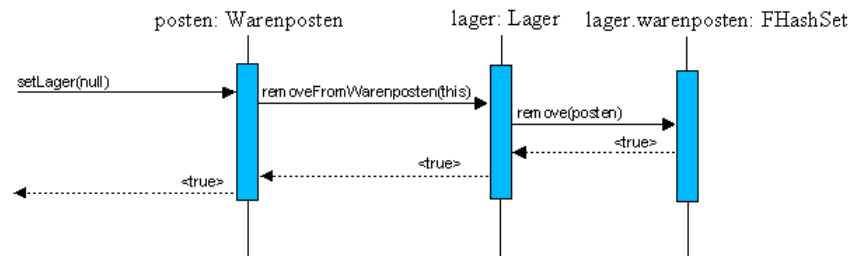
Im folgenden Diagramm wird die Beziehung zwischen den beiden Objekten hergestellt, indem die Methode `setLager(lager)` auf dem Objekt `neuerWarenposten` aufgerufen wird:



Die Sequenzdiagramme für das Löschen von Beziehungen zwischen `Lager` und `Warenposten` werden im Folgenden behandelt. Zunächst das Löschen der Beziehung durch Aufruf der `removeFromWarenposten`-Methode auf einem Objekt `lager` der Klasse `Lager`, dass zu einem Objekt `posten` der Klasse `Warenposten` eine Beziehung hat:



Und schließlich umgekehrt das Löschen der Beziehung durch Aufruf der `setLager`-Methode mit Parameter `null` auf dem Objekt `posten` der Klasse `Warenposten`, zum dem vom Lager `lager` eine Beziehung besteht:



4.5 Das Observer-Pattern als Beispiel für Entwurfsmuster

In der Einführung seines Buches *Entwurfsmuster* schreibt Gamma: „Jedes Entwurfsmuster benennt, erläutert und bewertet systematisch einen wichtigen und wiederkehrenden Entwurf in objektorientierten Systemen.“⁴² Entwurfsmuster gehören für ihn „entweder zum Allgemeinwissen objektorientierter Entwickler oder sie sind Teil erfolgreicher objektorientierter Systeme.“⁴³ Insofern ist es überaus sinnvoll, dieses 'Allgemeinwissen' zu nutzen und – bildlich gesprochen – das Rad nicht jedesmal neu zu erfinden.

Entwurfsmuster als aus den Erfahrungen vieler objektorientierter Entwickler gewonnenes 'Allgemeinwissen' sind daher sehr wohl auch im Informatikunterricht zu thematisieren. Entwurfsmuster können durchaus mit den von Andreas Schwill beschriebenen *fundamentalen Ideen der Informatik* als „grundlegende Prinzipien, Denkweisen und Methoden (den fundamentalen Ideen) der Informatik“⁴⁴ in Verbindung gebracht werden.

In der Software Schulkiosk wurde als Entwurfsmuster das Observer-Pattern umgesetzt, was im Folgenden beschrieben werden soll.

4.5.1 Beschreibung des Observer-Patterns

Das Observer-Pattern definiert „eine 1:n-Abhängigkeit zwischen Objekten, so daß die Änderung des Zustands eines Objekts dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.“⁴⁵ Die „Änderungen eines Objekts können sich auf andere Objekte auswirken, ohne daß das geänderte Objekt die anderen genauer kennen muß.“⁴⁶ Dies wird in der Software Schulkiosk – und oft auch im Allgemeinen⁴⁷ - dazu benutzt, um eine Trennung zwischen den Fachklassen (dem sog. Model) und der Benutzeroberfläche (der sog. View) zu realisieren. Ein auf das Observer-Pattern aufsetzendes⁴⁸ Entwurfsmuster bzw. Paradigma trägt denn auch das

42 Gamma: Entwurfsmuster, 2

43 Ebd., 2

44 Schwill, Andreas: Fundamentale Ideen der Informatik, 20

45 Gamma: Entwurfsmuster, 287

46 Ebd., 6

47 Vgl. ebd., 287ff.

48 Vgl. ebd., 300

Kürzel MVC (Model – View – Control) und soll genau diese Trennung zwischen Fachklassen und Oberfläche realisieren helfen, bei 'reinem' MVC ist noch eine sog. „Control-Schicht“ zwischengeschaltet. Die Fachklassen sollen dabei keine Kenntnis darüber benötigen, wie die Benutzungsoberfläche realisiert ist. Das bedeutet zum Beispiel für die Software Schulkiosk, welche Klassen die Aufgabe der Visualisierung des Lagers, der darin enthaltenen Warenposten oder der Kasse übernehmen, ist für die betreffenden Fachklassen ohne Bedeutung, sie müssen darüber nichts Genaueres wissen. Zur Umsetzung dieser Trennung von Fachklassen (Model) und Oberfläche (View) bietet sich also aus den oben genannten Gründen das Observer-Pattern an.

Was aber verbirgt sich hinter dem Observer-Pattern? - Nun, auf der einen Seite steht ein Observable-Objekt (also ein „beobachtbares“ Objekt). Auf der anderen Seite stehen eines oder auch mehrere Observer-Objekte (also die beobachtenden Objekte bzw. Beobachter). Dem Observable-Objekt muss mitgeteilt werden, dass und welche Observer über Zustandsänderungen benachrichtigt werden möchten, die Observer müssen ihm also bekannt gemacht werden. Anschließend, da dem Observable-Objekt nun seine Observer bekannt sind, wird es sie bei Zustandsänderungen benachrichtigen, und die Observer können dann ihrerseits beim von ihnen beobachteten Objekt nachfragen, was konkret sich am Zustand geändert hat und darauf reagieren. Über die Observer muss das Observable-Objekt tatsächlich nichts weiter wissen, außer, dass es Observer-Objekte sind. Die Eigenschaft, dass ein Objekt ein Observer ist, wird also typischerweise über Interfaces realisiert.⁴⁹

4.5.2 Umsetzung des Observer-Patterns in Java

Java bietet von Haus aus die Möglichkeit, das Observer-Pattern relativ einfach umzusetzen: Es stehen dazu die Klasse `Observable` und das Interface `Observer` zur Verfügung, beide enthalten im Package `java.util`.

Im wesentlichen wichtig für die Umsetzung sind die Methoden `addObserver()`, `setChanged()`, `notifyObservers()`, `deleteObserver(Observer o)` der Klasse `Observable` und die Methode

⁴⁹ Vgl. ebd., 291

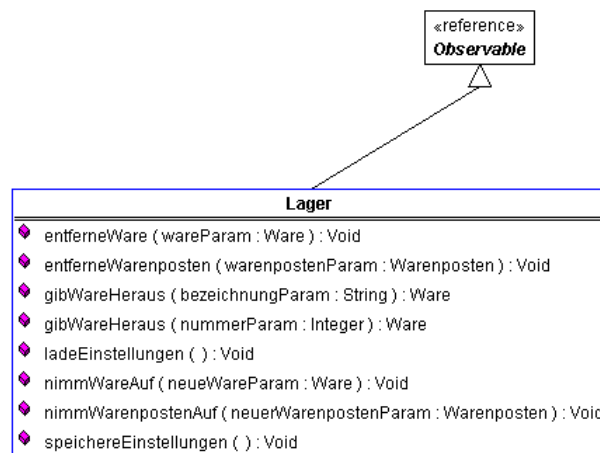
`update(Observable o, Object arg)` als einzige Methode des Interfaces `Observer`.

Die Klasse des Objektes, das an der Zustandsänderung eines anderen Objekts interessiert ist, muss nun das `Observer`-Interface implementieren. Damit wird das Objekt zu einem `Observer` (Beobachter). Das Objekt, welches seine `Observer` über Zustandsänderungen informieren will, also von `Observern` beobachtet wird, ist ein `Observable`-Objekt (also ein beobachtbares Objekt), seine Klasse erbt von der Klasse `Observable`, die die Funktionalität zum Verwalten und Benachrichtigen der `Observer` über die oben beschriebenen Methoden bereitstellt.

Natürlich kann sich ein Entwickler um diese Funktionalität auch selbst kümmern, ohne dass die betreffende zu beobachtende Klasse von `Observable` erbt. Dies ist jedoch mit einem Mehraufwand verbunden, den das Erben von der Klasse `Observable` im Allgemeinen überflüssig macht.

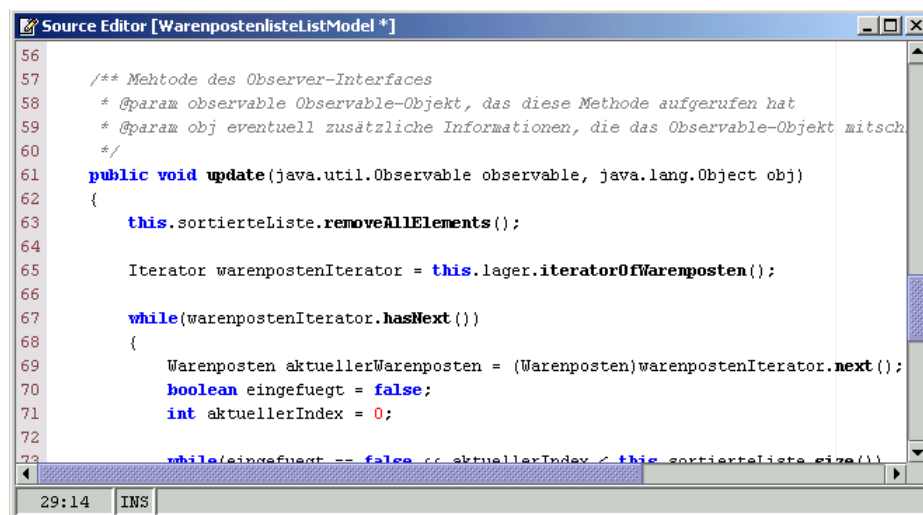
4.5.3 Das Observer Pattern in der Software Schulkiosk

Der Einsatz des `Observer`-Patterns soll im Folgenden an der Software Schulkiosk beschrieben werden. Als Beispiel dient die Klasse `Lager`. Sie erbt von der Klasse `java.util.Observable` die obigen Methoden. Neben der Klasse `Lager` erben auch die Klasse `Theke`, und in den Versionen 1.1 bzw. 1.2 die Klassen `BenutzerVerwaltung` und `StatistikModul` von der Klasse `java.util.Observable`.



Beim Lager meldet sich nun z.B. ein Objekt der Klasse WarenpostenlisteListModel als Observer an. Sie ist Bestandteil der Benutzeroberfläche und dazu da, die Daten für eine Liste im LagerFenster bereitzustellen, will und muss also informiert werden, wenn das Lager beispielsweise ein neues Objekt der Klasse Warenposten aufnimmt.

Um sich als Observer beim Lager anmelden zu können, muss die Klasse WarenpostenlisteListModel das Interface Observer implementieren, also die Methode `update(Observable o, Object arg)` bereitstellen:

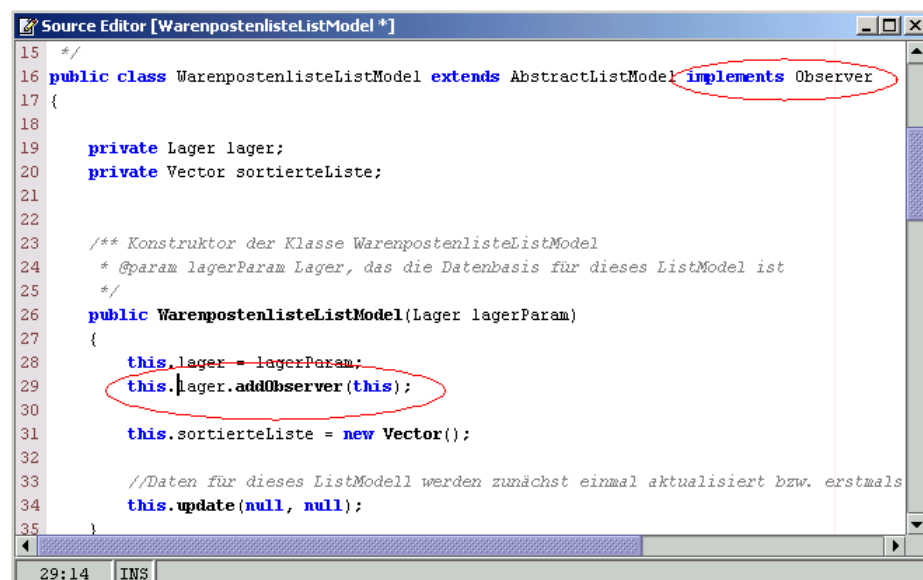


```

56
57  /** Methode des Observer-Interfaces
58  * @param observable Observable-Objekt, das diese Methode aufgerufen hat
59  * @param obj eventuell zusätzliche Informationen, die das Observable-Objekt mitsch
60  */
61  public void update(java.util.Observable observable, java.lang.Object obj)
62  {
63      this.sortierteListe.removeAlLElements();
64
65      Iterator warenpostenIterator = this.lager.iteratorOfWarenposten();
66
67      while(warenpostenIterator.hasNext())
68      {
69          Warenposten aktuellerWarenposten = (Warenposten)warenpostenIterator.next();
70          boolean eingefuegt = false;
71          int aktuellerIndex = 0;
72
73          while(eingefuegt == false && aktuellerIndex < this.sortierteListe.size())

```

Die Anmeldung beim Lager erfolgt schließlich durch Aufruf der Methode `addObserver(this)` auf dem Lager-Objekt durch das Objekt der Klasse WarenpostenlisteListModel, die ja das Observer-Interface implementiert:

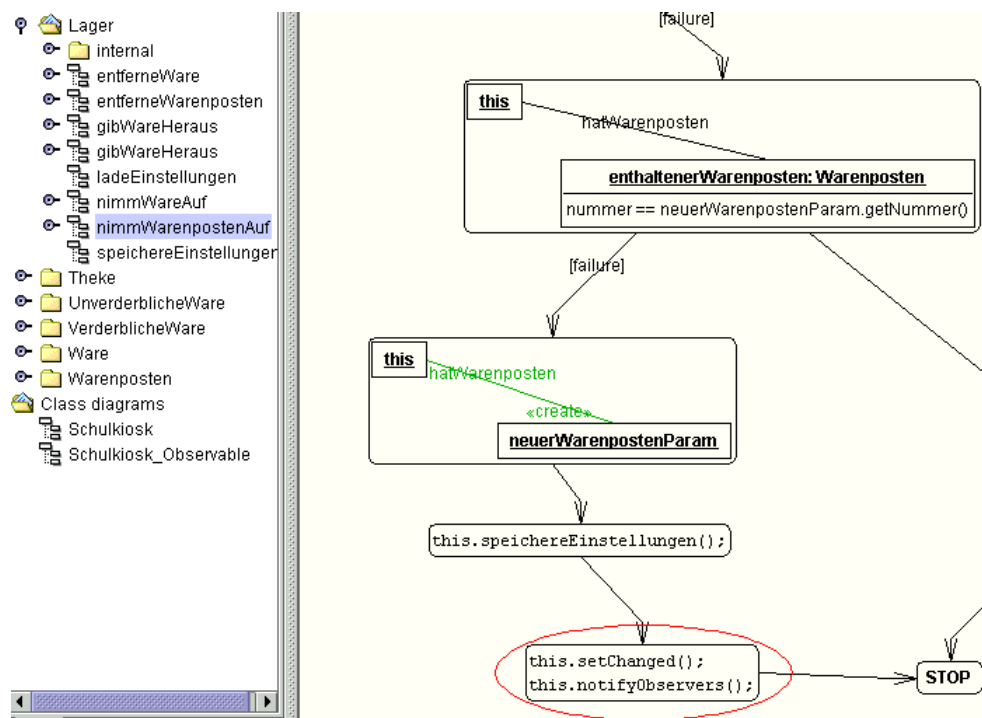


```

15  */
16  public class WarenpostenlisteListModel extends AbstractListModel implements Observer
17  {
18
19      private Lager lager;
20      private Vector sortierteListe;
21
22
23      /** Konstruktor der Klasse WarenpostenlisteListModel
24      * @param lagerParam Lager, das die Datenbasis für dieses ListModel ist
25      */
26      public WarenpostenlisteListModel(Lager lagerParam)
27      {
28          this.lager = lagerParam;
29          this.lager.addObserver(this);
30
31          this.sortierteListe = new Vector();
32
33          //Daten für dieses ListModell werden zunächst einmal aktualisiert bzw. erstmals
34          this.update(null, null);
35      }

```

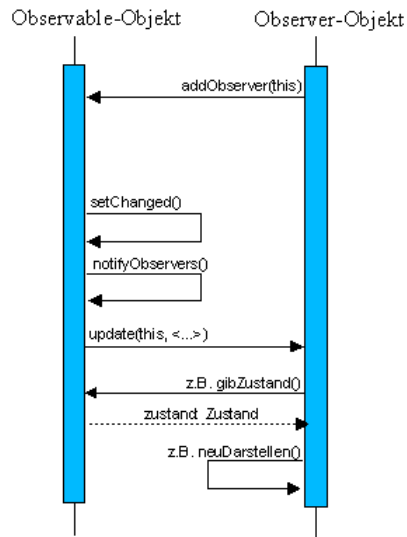
Wenn sich dann irgendwann am Zustand des Lager-Objekts etwas geändert hat, also beispielsweise ein neues Objekt der Klasse `Warenposten` aufgenommen wurde, ruft das Lager-Objekt auf sich selbst zunächst die Methode `this.setChanged()` auf, um zu signalisieren, dass sich sein Zustand geändert hat. Die eigentliche Benachrichtigung aller angemeldeten Observer erfolgt schließlich über den Aufruf der Methode `this.notifyObservers()`. Als Beispiel mag der folgende Ausschnitt aus dem Fujaba-Aktivitätsdiagramm der `nimmWarenpostenAuf`-Methode der Klasse `Lager` dienen:



Obiges `this.notifyObservers()` bewirkt, dass auf jedem angemeldeten Observer die Methode `update(Observable o, Object arg)` aufgerufen wird. In dieser `update`-Methode ist dann beim jeweiligen Observer festgelegt, wie bei einer Zustandsänderung des beobachteten Objekts zu verfahren ist. In unserem Fall wird das `WarenpostenlisteListModel`-Objekt dafür sorgen, dass die Liste im Lagerfenster aktualisiert wird und sich dazu vom Lager die entsprechenden Daten holen.

Möglicherweise ist zum besseren Verständnis des dem in Java umgesetzten

Observer-Pattern zugrunde liegenden Benachrichtigungsmechanismus noch ein Sequenzdiagramm hilfreich:



Besonders anschaulich wird das Funktionieren des obigen Benachrichtigungsmechanismus im Übrigen in der Software Schulkiosk, wenn man z.B. Kasse und Lagerverwaltung nebeneinander auf dem Bildschirm betrachtet: Gibt man in der Kasse eine Ware ein und erscheint sie in der Liste ausgewählter Waren, so läßt sich im selben Moment in der Lagerverwaltung beobachten, dass bei der betreffenden Ware unter „*vorhanden*“ der Wert um 1 verringert worden ist.

Im nächsten Abschnitt wird mit dem *Iterator* ein weiteres Beispiel für ein Muster vorgestellt.

4.6 Der Iterator – ein weiteres Muster

Der Iterator ist - wie bei Gamma beschrieben - ein objektbasiertes Verhaltensmuster. Es ermöglicht „den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen“⁵⁰.

4.6.1 Der Iterator in Java

Das oben genannte zusammengesetzte Objekt ist in Java häufig ein Objekt einer sog. Containerklasse wie die in `java.util` enthaltenen Klassen `Vector`, `ArrayList`, `HashSet` usw. Alle diese Containerklassen implementieren das Interface `Collection`, welches ebenfalls in `java.util` enthalten ist und die Methode `iterator()` definiert, die ein Objekt mit der Schnittstelle des `Iterator`-Interfaces als Rückgabewert liefert.

Das `Iterator`-Interface umfasst die folgenden drei Methoden, die den einheitlichen Zugriff auf alle Elemente von zusammengesetzten Objekten ermöglichen, ohne deren zugrunde liegende Repräsentation zu kennen:

Die Methode `next()` gibt das jeweils nächste Element des Iterators bzw. des zusammengesetzten Objekts, über das der Iterator iteriert, zurück. Die Methode `hasNext()` gibt an, ob es ein nächstes Element gibt. Und die Methode `remove()` schließlich löscht das zuletzt von `next()` zurückgegebene Objekt.

Das Interface `Iterator` bietet also zusammengefasst eine einheitliche Schnittstelle an, um auf die Elemente zusammengesetzter Objekte zuzugreifen.

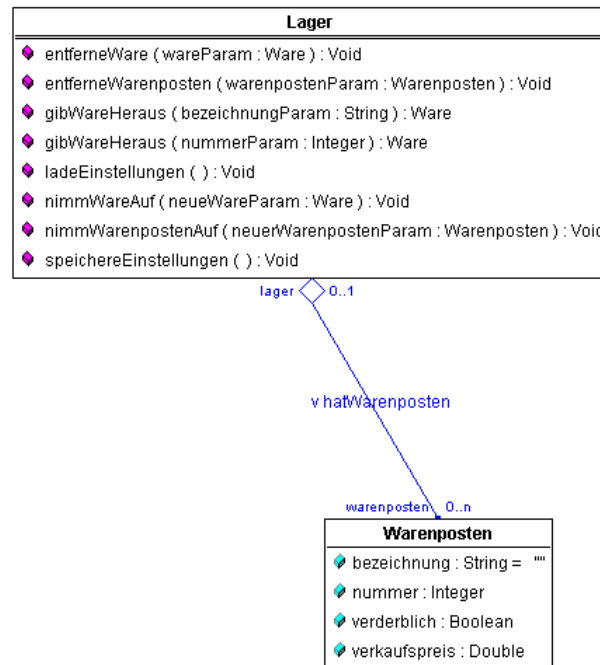
4.6.2 Der Iterator in der Software Schulkiosk

Auch in der Software Schulkiosk findet der Iterator Verwendung. So werden, wie in Kapitel 4.4 beschrieben, 1:n-Aggregationen bzw. 1:n-Assoziationen von Fujaba mittels der Containerklasse `FHashSet` umgesetzt. Um nun auf diese aggregierten bzw. assoziierten Objekte zuzugreifen, generiert Fujaba für jede dieser Beziehungen eine Zugriffsmethode `iteratorOf<<'Name'>>()`, deren

⁵⁰ Gamma: Entwurfsmuster, 335

Rückgabewert ein Objekt mit `Iterator`-Schnittstelle ist, über die so auf alle aggregierten bzw. assoziierten Objekte zugegriffen werden kann.

Als Beispiel sei auf die 1:n-Aggregation zwischen `Lager` und `Warenposten` verwiesen:



Fujaba erzeugt hier aus dem obigen Klassendiagramm unter anderem die Methode `iteratorOfWarenposten()`, deren Quelltext im Folgenden angegeben ist:

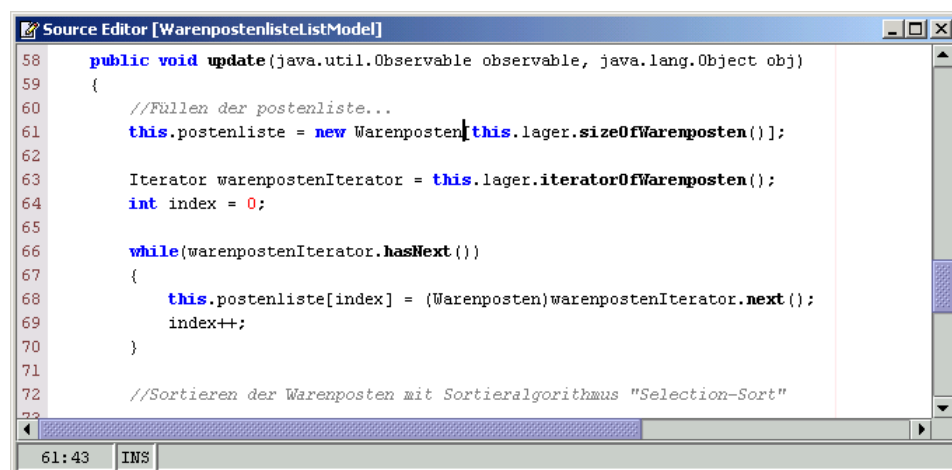
```

Source Editor [Lager]
778
779 /**
780  * UMLMethod: '+ iteratorOfWarenposten (): Iterator'.
781  */
782 public Iterator iteratorOfWarenposten ()
783 {
784     return ((this.warenposten == null)
785         ? FEmptyIterator.get ()
786         : this.warenposten.iterator ());
787 }
788
789 |
789:1  INS
  
```

Rückgabewert ist ein Objekt mit `Iterator`-Schnittstelle, das den Zugriff auf alle `Warenposten`-Objekte ermöglicht, zu denen eine Aggregationsbeziehung besteht. Und zwar besagt `return ((this.warenposten == 0) ? FEmptyIterator.get() : this.warenposten.iterator())` nichts anderes als: Wenn das Attribut `warenposten` der Klasse `FHashSet` null ist,

dann gib mit `EmptyIterator.get()` ein „leeres“ `Iterator`-Objekt zurück, denn dann besteht noch keine Beziehung vom Lager zu einem `Warenposten`-Objekt. Sonst gib mit `this.warenposten.iterator()` einen `Iterator` über den Container `warenposten` zurück.

Wie der Zugriff auf zusammengesetzte Objekte über die `Iterator`-Schnittstelle funktioniert, sei anhand der Klasse `WarenpostenlisteListModel` erklärt, die alle im Lager enthaltenen `Warenposten` für die Darstellung in einer Liste bereitstellt und sie dazu sortiert. Ein Objekt der Klasse `WarenpostenlisteListModel` meldet sich dazu zunächst als `Observer` beim Lager an⁵¹. Daraufhin wird es immer, wenn sich der Zustand vom Lager ändert (z.B. durch Aufnahme oder Entfernen von `Warenposten`), von diesem durch Aufruf der `update`-Methode benachrichtigt und aktualisiert dann in ihr seine Daten. Das aber sind die zu sortierenden `Warenposten` aus dem Lager. Nun, und auf eben diese erlangt es über die `iteratorOfWarenposten`-Methode der Klasse `Lager` Zugriff, in dem ein `Iterator` über alle `Warenposten`-Objekte zurückgegeben wird, zu denen das Lager eine Beziehung hat:



```
Source Editor [WarenpostenlisteListModel]
58 public void update(java.util.Observable observable, java.lang.Object obj)
59 {
60     //Füllen der postenliste...
61     this.postenliste = new Warenposten[this.lager.sizeOfWarenposten()];
62
63     Iterator warenpostenIterator = this.lager.iteratorOfWarenposten();
64     int index = 0;
65
66     while(warenpostenIterator.hasNext())
67     {
68         this.postenliste[index] = (Warenposten)warenpostenIterator.next();
69         index++;
70     }
71
72     //Sortieren der Warenposten mit Sortieralgorithmus "Selection-Sort"
73
```

Der obige Quelltext zeigt das Updaten des Arrays `postenliste`, in welchem das `WarenpostenlisteListModel` die `Warenposten` aus dem Lager ablegt und sortiert. Zunächst einmal wird der vom Lager über `this.lager.iteratorOfWarenposten()` als Rückgabewert erhaltene `Iterator` der lokalen Variablen `warenpostenIterator` zugewiesen.

⁵¹ siehe Kapitel 4.5: Das Observer-Pattern als Beispiel für Entwurfsmuster

Anschließend wird mit einer `while`-Schleife über die Elemente des `Iterators` iteriert, also über alle im Lager enthaltenen Warenposten. In dieser Schleife wird über `warenpostenIterator.next()` das jeweils nächste Element des `Iterators` zurückgegeben, welches dann nach einem Casting an die über die Zählvariable `index` definierte Position im Array gespeichert wird.

Das nächste didaktische Fenster greift das Thema Ereignisbehandlung auf.

4.7 Ereignisbehandlung

Moderne graphische Benutzeroberflächen werden aufgebaut u.a. nach dem Prinzip der Ereignisorientierung. Als Ereignisse gelten beispielsweise das Drücken eines Buttons durch den Benutzer, das Klicken und Bewegen der Maus, das Drücken von Tasten, das Auswählen eines Elements in einer Liste usw.

4.7.1 Beschreibung der Ereignisbehandlung in Java

Die Ereignisbehandlung in Java basiert auf dem sog. „*Listener-Konzept*“⁵², welches im Wesentlichen auf dem im vorigen Kapitel beschriebenen Observer-Pattern beruht. Es wird unterschieden zwischen Ereignisquellen, Ereignisempfängern und den Ereignissen selbst.

Ereignisquellen sind zum Beispiel Elemente der Benutzeroberfläche wie Buttons, Textfelder, Listen und viele mehr. Als Ereignisempfänger fungieren sog. 'Listener'-Objekte. Das sind Objekte, deren Klassen eine oder mehrere bestimmte Listener-Interfaces implementieren. Die Ereignisse selbst sind in Java ebenfalls Objekte, wofür es verschiedene Klassen passend für den jeweiligen Ereignistyp gibt. Um nur zwei wesentliche zu nennen: Ein `ActionEvent` ist ein z.B. durch das Drücken eines Buttons oder das Betätigen der Eingabetaste in einem Textfeld ausgelöstes Ereignis; ein `KeyEvent` ist ein z.B. durch das Eingeben eines Zeichens über die Tastatur ausgelöstes Ereignis.

Wie oben bereits angedeutet, beruht die Ereignisorientierung in Java im Wesentlichen auf der Anwendung des *Observer-Patterns*⁵³. Hier wie dort gibt es Objekte, die andere Objekte über ihre Zustandsänderung (z.B. „*Button wurde gedrückt*“) benachrichtigen. Solche Objekte sind die oben beschriebenen Ereignisquellen. Bei ihnen melden sich die Listener-Objekte an und werden daraufhin immer dann benachrichtigt, wenn ein Ereignis bei der Ereignisquelle, bei der sie angemeldet sind, auftritt. In Analogie zum Observer-Pattern übernehmen die Ereignisquellen also die Rolle der Observable-Objekte und die

⁵² Vgl. Budd, Timothy: Understanding object-oriented programming with Java, 91

⁵³ siehe Kapitel 4.5

Listener die Rolle der Observer-Objekte.

Hinzu kommen noch die Ereignisse selbst als eigene Objekte, die von den Ereignisquellen an die Ereignisempfänger geschickt werden: Jedes Event-Objekt bietet z.B die Methode `getSource()`, die das Objekt zurückgibt, welches das Ereignis ausgelöst hat (also die Ereignisquelle). Dies macht u.a. eine zentrale Behandlung von Ereignissen, die ja potentiell von mehreren Ereignisquellen kommen können, möglich und sinnvoll.

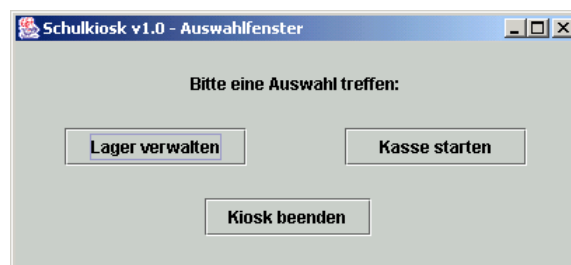
Java bietet, wie gerade bereits angedeutet, eine Vielzahl an möglichen Ereignistypen und zugehörigen Listnern. Sie befinden sich in den Packages `java.awt.event` und `javax.swing.event`.

Ereignisquellen sind, wie gesagt, typischerweise häufig Elemente der Benutzeroberfläche wie zum Beispiel Buttons, Textfelder, Listen, Scrollbars, Fenster, usw. Alle diese Elemente werden von den Packages `java.awt` und `java.swing` zur Verfügung gestellt.

Im Folgenden soll nun die Ereignisbehandlung in Java anhand eines Ausschnitts der Software Schulkiosk illustriert werden.

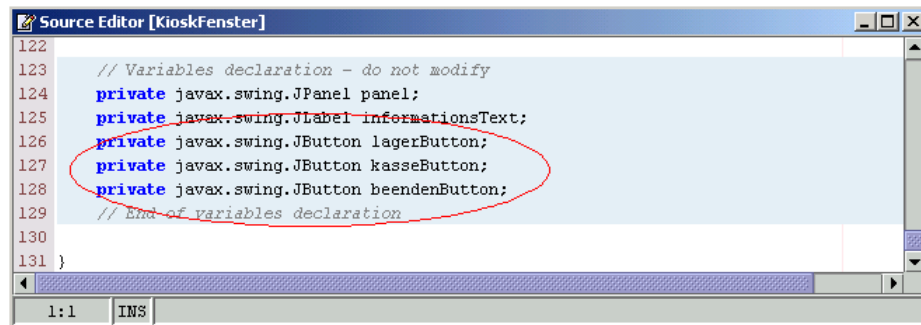
4.7.2 Ereignisbehandlung in der Software Schulkiosk

Als Beispiel herausgenommen sei hier seiner Einfachheit wegen die Klasse `KioskFenster` der Version 1.0. (Die Klasse `KioskFenster` der Versionen 1.1 und 1.2 ist allerdings ebenso einfach aufgebaut.) Es handelt sich hierbei um ein Fenster, ein `JFrame`-Objekt, das nach dem Start der Software Schulkiosk erscheint und über drei `JButton`-Objekte dem Benutzer die Auswahl anbietet, entweder die Lagerverwaltung oder die Kasse zu starten oder den Schulkiosk zu beenden.



Die Klasse `KioskFenster` verfügt also u.a. über die privaten Attribute

lagerButton, kasseButton und beendenButton, welche alle drei Objekte der Klasse Jutton sind:

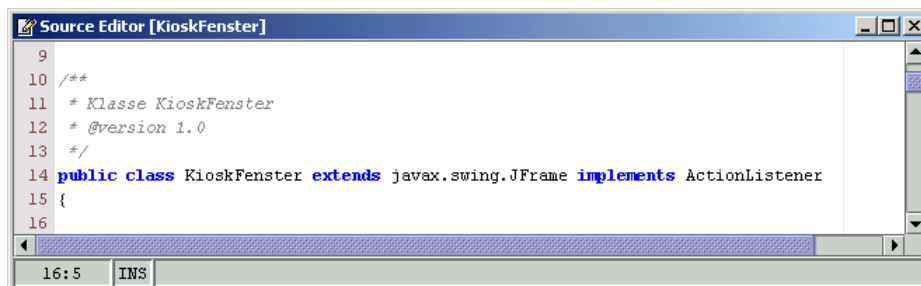


```

122
123 // Variables declaration - do not modify
124 private javax.swing.JPanel panel;
125 private javax.swing.JLabel informationsText;
126 private javax.swing.JButton lagerButton;
127 private javax.swing.JButton kasseButton;
128 private javax.swing.JButton beendenButton;
129 // End of variables declaration
130
131 }

```

Die Klasse KioskFenster implementiert selbst das ActionListener-Interface, kann sich somit bei Ereignisquellen, die ein ActionEvent auslösen können, als ActionListener anmelden, wird also zu einem Ereignisempfänger:

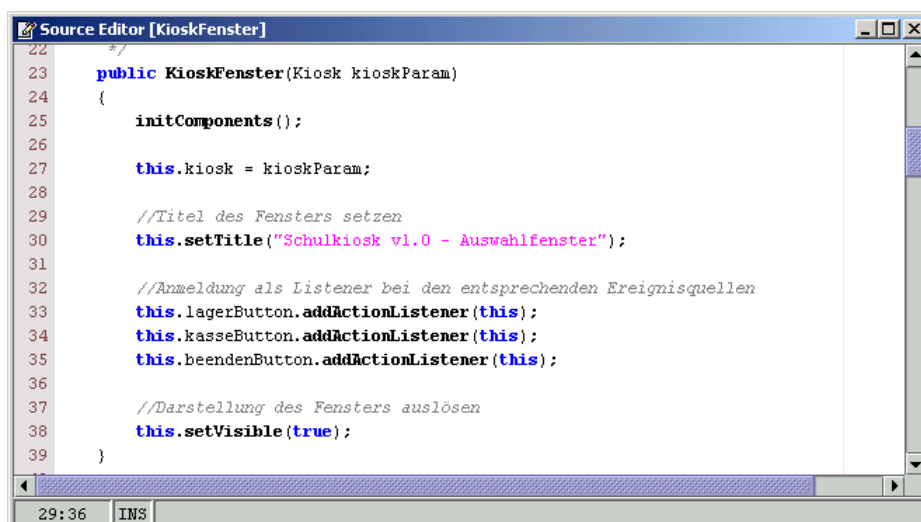


```

9
10 /**
11  * Klasse KioskFenster
12  * @version 1.0
13  */
14 public class KioskFenster extends javax.swing.JFrame implements ActionListener
15 {
16

```

Das Anmelden bei den JButton-Objekten erfolgt schließlich im Konstruktor der Klasse KioskFenster durch Aufruf der Methode addActionListener(ActionListener listener) auf den jeweiligen Buttons:

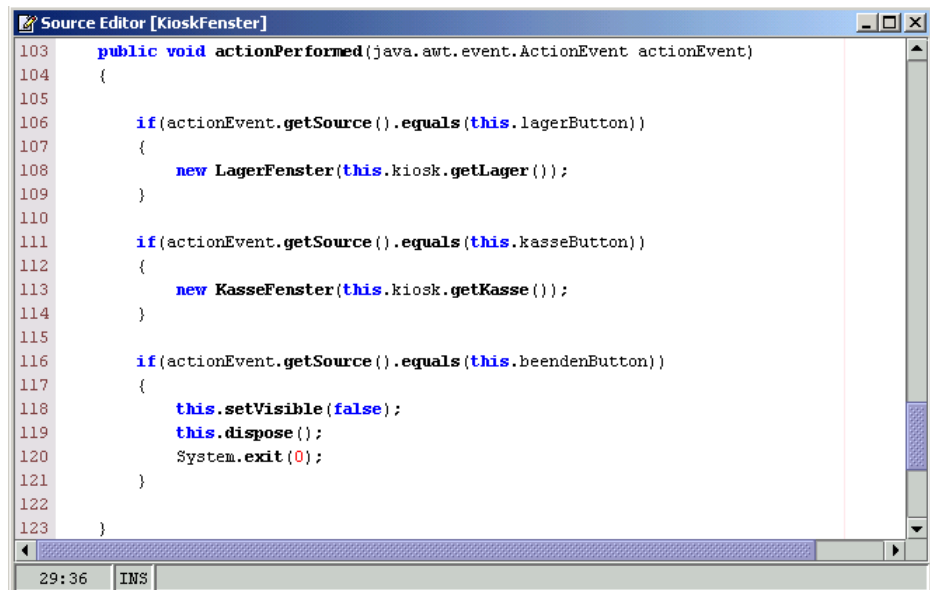


```

22 //
23 public KioskFenster(Kiosk kioskParam)
24 {
25     initComponents();
26
27     this.kiosk = kioskParam;
28
29     //Titel des Fensters setzen
30     this.setTitle("Schulkiosk v1.0 - Auswahlfenster");
31
32     //Anmeldung als Listener bei den entsprechenden Ereignisquellen
33     this.lagerButton.addActionListener(this);
34     this.kasseButton.addActionListener(this);
35     this.beendenButton.addActionListener(this);
36
37     //Darstellung des Fensters auslösen
38     this.setVisible(true);
39 }

```

Wird nun ein Button gedrückt, so werden alle angemeldeten ActionListener-Objekte benachrichtigt, indem auf ihnen die Methode `actionPerformed(ActionEvent actionEvent)` des ActionListener-Interfaces mit einem Objekt der Klasse `ActionEvent` als Parameter aufgerufen wird. In dieser Methode findet schließlich die eigentliche Ereignisbehandlung statt:



```

103 public void actionPerformed(java.awt.event.ActionEvent actionEvent)
104 {
105
106     if(actionEvent.getSource().equals(this.lagerButton))
107     {
108         new LagerFenster(this.kiosk.getLager());
109     }
110
111     if(actionEvent.getSource().equals(this.kasseButton))
112     {
113         new KasseFenster(this.kiosk.getKasse());
114     }
115
116     if(actionEvent.getSource().equals(this.beendenButton))
117     {
118         this.setVisible(false);
119         this.dispose();
120         System.exit(0);
121     }
122
123 }

```

Und zwar wird hier über eine Abfolge von `if`-Anweisungen und mit Hilfe der o.g. `getSource()`-Methode festgestellt, welches Objekt jeweils der Auslöser des Ereignisses (die Ereignisquelle) gewesen ist. Je nachdem, welcher Button also das Ereignis ausgelöst hat, wird entweder die Lagerverwaltung bzw. die Kasse gestartet, oder der Schulkiosk wird beendet.

Im Folgenden eine Übersicht über die Ereignisklassen und zugehörige Listener, wie sie in der Software Schulkiosk verwendet werden:

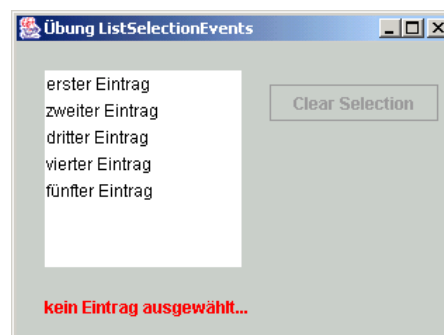
<i>Ereignisklasse</i>	<i>zugehöriges Listener-Interface</i>	<i>Java-Bibliothek</i>
ActionEvent	ActionListener	java.awt.event
FocusEvent	FocusListener	java.awt.event
KeyEvent	KeyListener	java.awt.event
ListSelectionEvent	ListSelectionListener	javax.swing.event

Die nächste Tabelle gibt Aufschluss darüber, welche Methoden die einzelnen Interfaces umfassen:

<i>Interface</i>	<i>Methoden</i>
ActionListener	public void actionPerformed(ActionEvent actionEvent)
FocusListener	public void focusGained(FocusEvent focusEvent) public void focusLost(FocusEvent focusEvent)
KeyListener	public void keyPressed(KeyEvent) public void keyReleased(KeyEvent) public void keyTyped(KeyEvent)
ListSelectionListener	public void valueChanged(ListSelectionEvent selectionEvent)

Ganz analog zum gerade illustrierten Umgang mit ActionEvents in der Software Schulkiosk, entpuppt sich die Behandlung von ListSelectionEvents. Hier zeigt sich, dass das Konzept der Ereignisbehandlung – einmal verstanden – universell einsetzbar ist. Eine Übungsaufgabe, wie sie im Unterricht eingesetzt werden kann, zeigt, wie dies aussehen könnte:

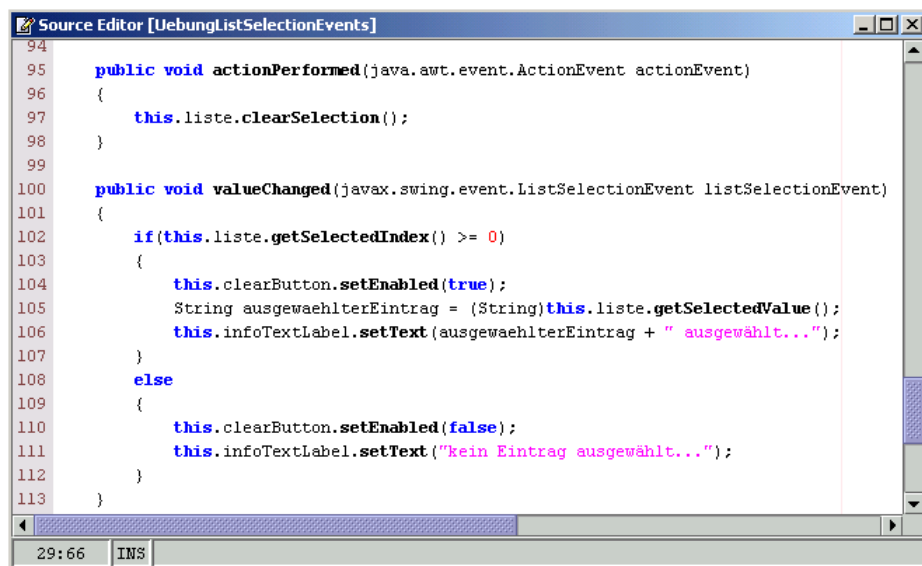
Die Schülerinnen und Schüler erhalten eine Klasse, die von der Klasse JFrame erbt und eine Liste, einen Button und ein Label als Attribute enthält. Sie verfügt zudem über eine main-Methode, die ein Objekt dieser Klasse erzeugt, das sich dann folgendermaßen auf dem Bildschirm darstellt:



Die Aufgabe ist es nun, die Klasse so zu implementieren, dass wenn ein Eintrag der Liste ausgewählt (selektiert) wird, das Label den Namen dieses Eintrags ausgibt. Wenn keines ausgewählt ist, soll dies ebenfalls durch das Label angezeigt werden: z.B. „kein Eintrag ausgewählt...“. Außerdem soll dann der Button „Clear Selection“ inaktiv sein. Die Funktionalität des „Clear Selection“-Buttons schließlich soll es sein, die Auswahl der Liste mittels `liste.clearSelection()` aufzuheben.

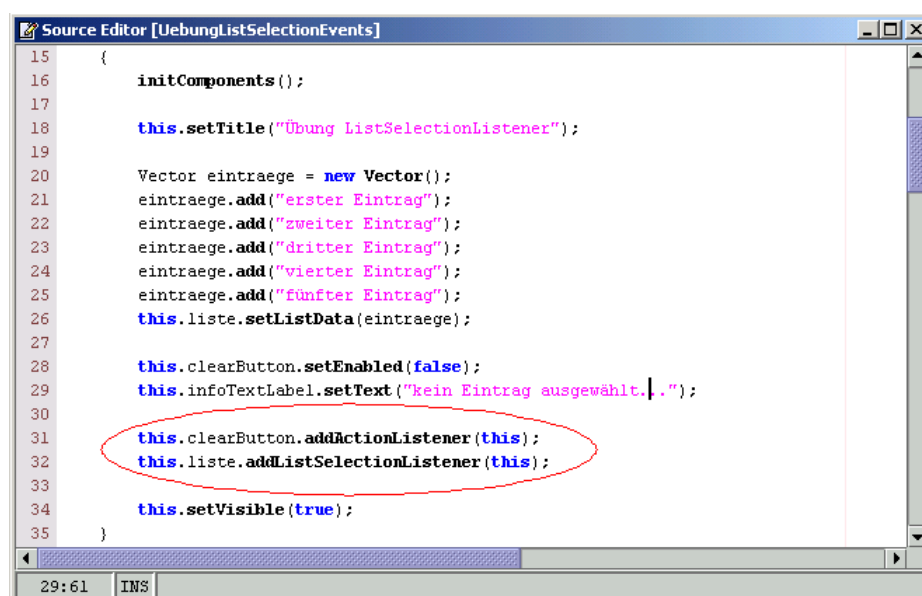
Zur Lösung: Die Klasse muss zunächst einmal die beiden Interfaces

ActionListener und ListSelectionListener implementieren. In den damit zu implementierenden Methoden `actionPerformed(ActionEvent actionEvent)` und `valueChanged(ListSelectionEvent listSelectionEvent)` muss nun für das in der Aufgabenstellung geforderte Verhalten gesorgt werden. Der entsprechende Quelltext könnte beispielsweise so aussehen:



```
Source Editor [UebungListSelectionEvents]
94
95 public void actionPerformed(java.awt.event.ActionEvent actionEvent)
96 {
97     this.liste.clearSelection();
98 }
99
100 public void valueChanged(javax.swing.event.ListSelectionEvent listSelectionEvent)
101 {
102     if(this.liste.getSelectedIndex() >= 0)
103     {
104         this.clearButton.setEnabled(true);
105         String ausgewaehlteEintrag = (String)this.liste.getSelectedValue();
106         this.infoTextLabel.setText(ausgewaehlteEintrag + " ausgewählt...");
107     }
108     else
109     {
110         this.clearButton.setEnabled(false);
111         this.infoTextLabel.setText("kein Eintrag ausgewählt...");
112     }
113 }
```

Schließlich, damit auf die entsprechenden Ereignisse reagiert werden kann, muss die Klasse (Sprechweise: eigentlich ein Objekt der Klasse) sich bei dem Button als ActionListener und bei der Liste als ListSelectionListener anmelden. Dies erfolgt sinnvollerweise im Konstruktor:



```
Source Editor [UebungListSelectionEvents]
15 {
16     initComponents();
17
18     this.setTitle("Übung ListSelectionListener");
19
20     Vector eintraege = new Vector();
21     eintraege.add("erster Eintrag");
22     eintraege.add("zweiter Eintrag");
23     eintraege.add("dritter Eintrag");
24     eintraege.add("vierter Eintrag");
25     eintraege.add("fünfter Eintrag");
26     this.liste.setListData(eintraege);
27
28     this.clearButton.setEnabled(false);
29     this.infoTextLabel.setText("kein Eintrag ausgewählt...");
30
31     this.clearButton.addActionListener(this);
32     this.liste.addListSelectionListener(this);
33
34     this.setVisible(true);
35 }
```

4.8 Ausnahmebehandlung

Ein wichtiger Aspekt bei der Implementierung von (größeren) Softwareprojekten ist die sog. Fehler- bzw. Ausnahmebehandlung (engl. Exception Handling).

Eine klassische Möglichkeit mit Fehlern umzugehen, ist es, sog. Flags zu verwenden. Das sind lokale bool'sche Variablen, die immer dann auf *true* gesetzt werden, wenn ein Fehler auftritt. Die eigentliche Fehlerbehandlung findet schließlich an einem geeigneten Ort im Quelltext statt. Eine andere Möglichkeit ist es, dass fehleranfällige Methoden (z.B. solche, die mit Dateien hantieren) über einen Rückgabewert mitteilen, ob bei ihrer Ausführung ein Fehler aufgetreten ist. In der Praxis werden solche Rückgabewerte jedoch häufig und gern ignoriert, niemand kann jedenfalls sicherstellen, dass sie überhaupt abgefragt werden.⁵⁴

Eine wesentlich elegantere, sicherere und effektivere Art mit Fehlern umzugehen, stellt das Auslösen und anschließende Behandeln von Ausnahmen dar. Dabei liegt schon in der Schnittstelle einer Methode selbst fest, ob und ggf. welche Ausnahmen die Methode auslösen kann. Jemand, der diese Methode schließlich aufruft, ist nun gezwungen, diese eventuelle Ausnahme auch zu behandeln. Sei es, dass die Ausnahme an Ort und Stelle behandelt wird oder dass sie dorthin weitergereicht wird, wo sie sinnvoller behandelt werden kann. Java selbst unterstützt diese effektive Form der Fehlerbehandlung von Haus aus, wie im Folgenden zu sehen ist:

4.8.1 Ausnahmebehandlung in Java

In Java sind Ausnahmen/Exceptions selbst Objekte. Alle Klassen, die eine Ausnahme beschreiben sollen, müssen dazu von der Klasse `Exception` erben.

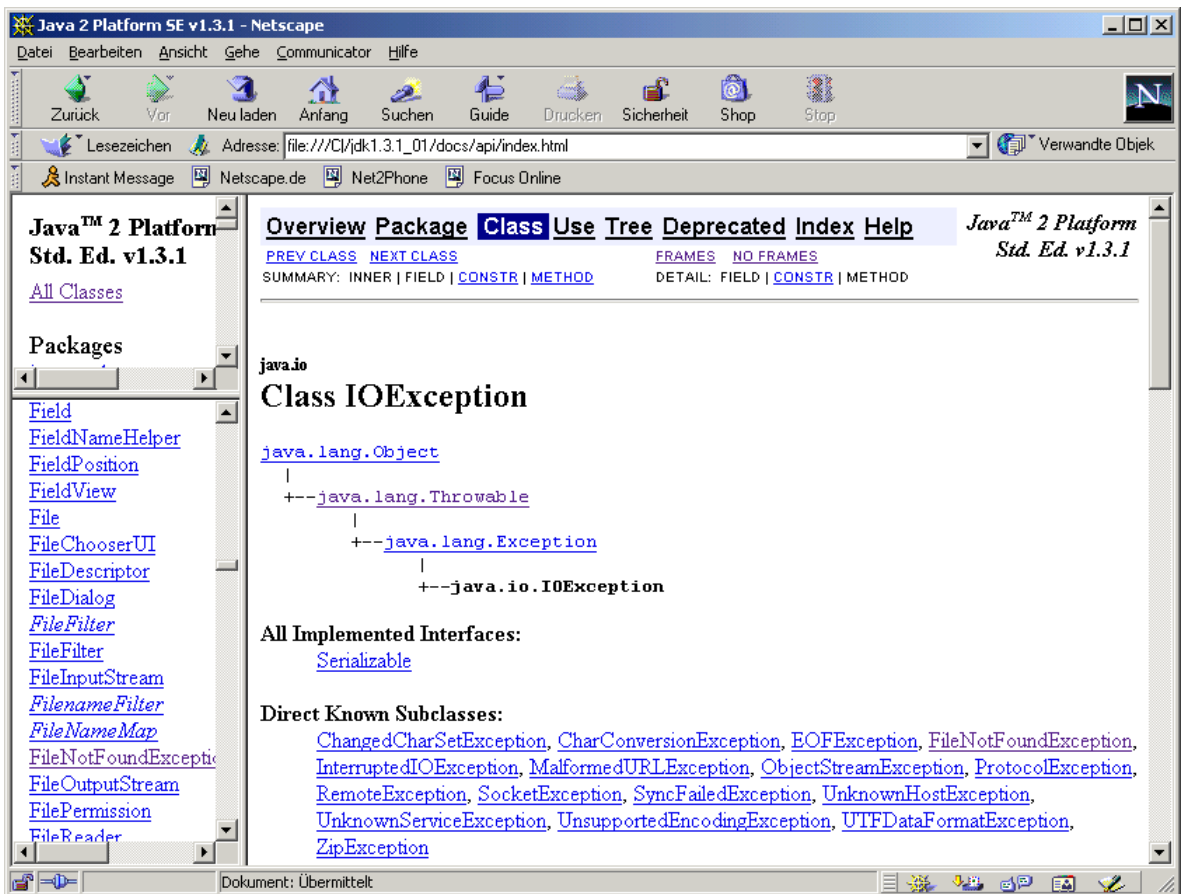
Diese stellt u.a. die Methode `getMessage()` bereit, die einen `String` mit einer Fehlermeldung zurückgibt, der beim Erzeugen/Auslösen der Ausnahme dem Konstruktor des `Exception`-Objekts übergeben werden kann.

In der Java-Klassenbibliothek selbst gibt es eine ganze Hierarchie von Ausnahmen. Als Beispiel seien hier die im Package `java.io` enthaltenen

⁵⁴ Warren, Nigel; Bishop, Philip: Java in Practice, 64f

Exception-Klassen angeführt.

Dort gibt es die Klasse `IOException`, die von `Exception` erbt und von der wiederum alle anderen Exception-Klassen im Package `java.io` erben:



Von `IOException` erben also – wie oben zu sehen ist – eine ganze Reihe von Klassen, die je eine spezielle Ausnahme beschreiben.

So beschreibt zum Beispiel die Klasse `FileNotFoundException` eine Ausnahme, die ausgelöst wird, falls versucht wird, auf eine nicht existierende Datei zuzugreifen.

Löst nun eine Methode wie beispielsweise der Konstruktor der Klasse `FileInputStream` potentiell eine Ausnahme aus, so wird dies in der Deklaration der Methode kenntlich gemacht, wie folgender Ausschnitt aus der Dokumentation zur Klasse `FileInputStream` zeigt:

Constructor Detail

FileInputStream

```
public FileInputStream(String name)
    throws FileNotFoundException
```

Creates a `FileInputStream` by opening a connection to an actual file, the file named by the path name `name` in the file system. A new `FileDescriptor` object is created to represent this file connection.

First, if there is a security manager, its `checkRead` method is called with the `name` argument as its argument.

If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a `FileNotFoundException` is thrown.

Parameters:

`name` - the system-dependent file name.

Throws:

[FileNotFoundException](#) - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

[SecurityException](#) - if a security manager exists and its `checkRead` method denies read access to the file.

See Also:

[SecurityManager.checkRead\(java.lang.String\)](#)

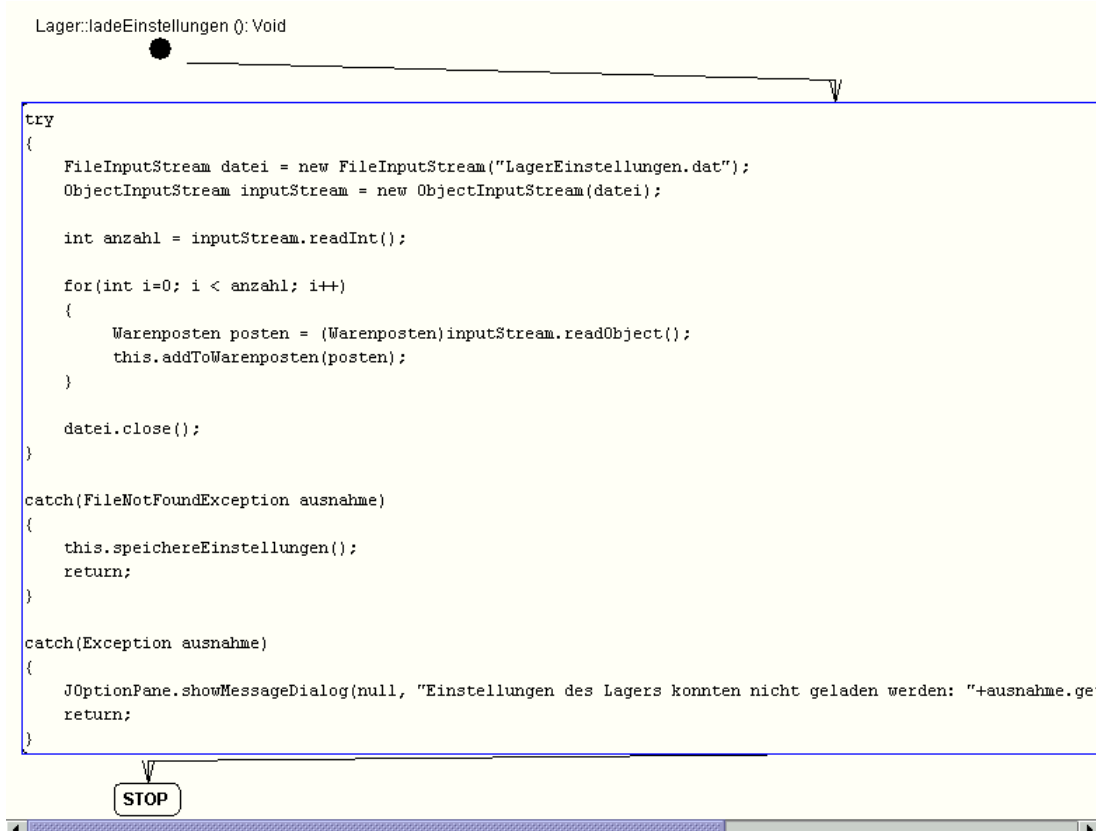
Der hier beschriebene Konstruktor erzeugt einen Input-Stream, um aus einer Datei mit dem als Parameter übergebenen Namen lesen zu können. Dabei kann natürlich der Fall vorkommen, dass die Datei nicht existiert oder aus einem anderen Grund nicht geöffnet werden kann. Diese Möglichkeiten signalisiert die Methode in ihrer Deklaration durch `throws FileNotFoundException` und fordert damit, dass bei Aufruf dieser Methode die potentielle Ausnahme behandelt werden muss.

Wie aber läuft die Behandlung von Ereignissen ab? - Nun dies geschieht in zwei bzw. drei Blöcken:

In einem `try`-Block (`try {...}`) steht der Code, bei dessen Ausführung potentiell Ausnahme ausgelöst werden können. Ihm folgen ein oder mehrere `catch`-Blöcke (z.B. `catch(Exception ausnahme) {...}`), die bei Auftreten der in den runden Klammern der `catch`-Anweisung beschriebenen Ausnahme ausgeführt werden. Schließlich kann noch ein `finally`-Block (`finally {...}`) stehen, dessen Anweisungen in jedem Fall ausgeführt werden, egal, ob eine Ausnahme ausgelöst wurde oder nicht.

Folgender Ausschnitt aus der `ladeEinstellungen`-Methode der Klasse `Lager` soll die Fehlerbehandlung in Java illustrieren:

```
Lager::ladeEinstellungen():Void  
  
try  
{  
    FileInputStream datei = new FileInputStream("LagerEinstellungen.dat");  
    ObjectInputStream inputStream = new ObjectInputStream(datei);  
  
    int anzahl = inputStream.readInt();  
  
    for(int i=0; i < anzahl; i++)  
    {  
        Warenposten posten = (Warenposten)inputStream.readObject();  
        this.addToWarenposten(posten);  
    }  
  
    datei.close();  
}  
  
catch(FileNotFoundException ausnahme)  
{  
    this.speichereEinstellungen();  
    return;  
}  
  
catch(Exception ausnahme)  
{  
    JOptionPane.showMessageDialog(null, "Einstellungen des Lagers konnten nicht geladen werden: "+ausnahme.get  
    return;  
}
```

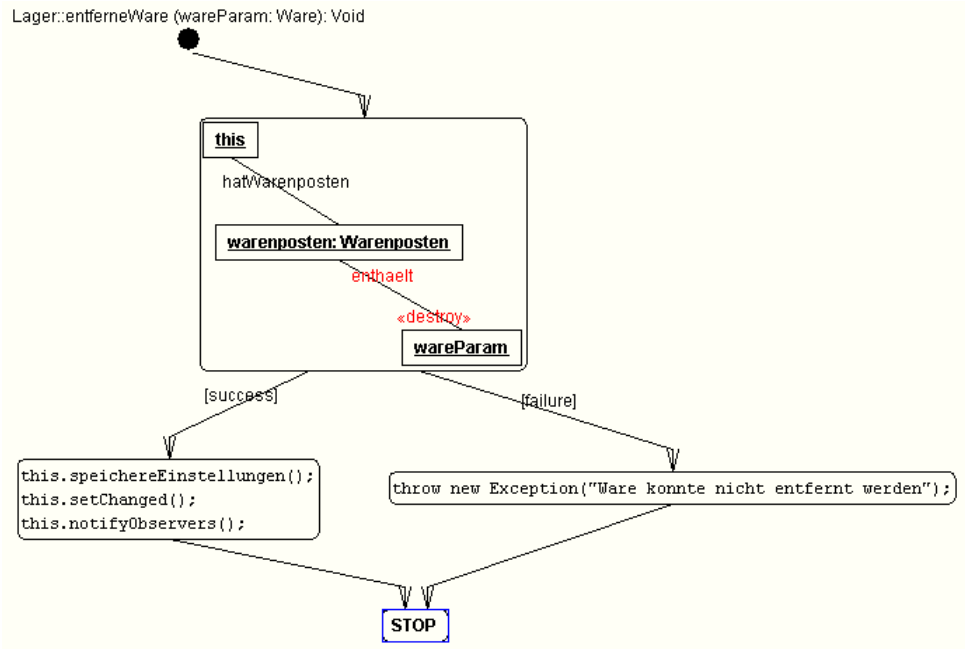


Der `try`-Block enthält die Anweisungen, bei deren Ausführung potentiell Ausnahmen auftreten können. Dies kann zum einen beim Aufruf des oben bereits beschriebenen Konstruktors der Klasse `FileInputStream` geschehen, aber auch andere Fehlerquellen sind zumindest denkbar. So kann (nur als Beispiel) auch das durchzuführende Casting `(Warenposten)inputStream.readObject()` eine Ausnahme auslösen, wenn das aus dem Stream gelesene Objekt kein `Warenposten` ist, also nicht gecastet werden kann.

Die Behandlung der Ausnahmen in der `ladeEinstellungen`-Methode erfolgt über zwei `catch`-Blöcke. Die Anweisungen im `catch(FileNotFoundException ausnahme)`-Block legen fest, dass, wenn noch keine Datei mit dem Namen „LagerEinstellungen.dat“ existiert, die Methode `speichereEinstellungen()` aufgerufen wird, in der dann die Datei erstmals neu angelegt wird. Der zweite `catch(Exception ausnahme)`-Block legt fest, was bei einem anderen Fehler geschehen soll: In diesem Fall wird ein Dialogfenster mit der Fehlermeldung auf dem Bildschirm angezeigt.

4.8.2 Auslösen von Ausnahmen in eigenen Methoden

Folgender Ausschnitt aus der Software Schulkiosk demonstriert, wie Exceptions in eigenen Methoden verwendet und ausgelöst werden:



Der zugehörige Quelltext der Methodendeklaration der obigen, als Fujaba-Aktivitätsdiagramm notierten Methode sieht so aus:

```

Source Editor [Lager]
22  /**
23  * entfernt die als Parameter übergebene Ware aus dem Lager, speichert die E
24  * @param wareParam Ware, die entfernt werden soll
25  * @throws java.lang.Exception Ausnahme, die ausgelöst wird, falls die als P
26  */
27  public void entferneWare (Ware wareParam) throws Exception
  
```

Eine Ausnahme kann also – das Fujaba-Aktivitätsdiagramm zeigt es – einfach über die `throw`-Anweisung ausgelöst werden. Beim Erzeugen der Ausnahme kann, wie bereits beschrieben, ein String mit der Fehlermeldung als Parameter übergeben werden. Dieser kann dann von der die Ausnahme behandelnden Methode über die ebenfalls bereits erwähnte Methode `getMessage()` ausgelesen werden.

Dies geschieht denn auch in der Klasse `WarenpostenBearbeitenDialog`, in der die von der gezeigten `entferneWare`-Methode der Klasse `Lager` potentiell ausgelöste Ausnahme behandelt wird. Der folgende Quelltext ist Bestandteil der `actionPerformed`-Methode der Klasse

WarenpostenBearbeitenDialog und wird ausgeführt, wenn der „Entfernen“-Button gedrückt wird und eine Ware in der JList warenliste selektiert ist:

```

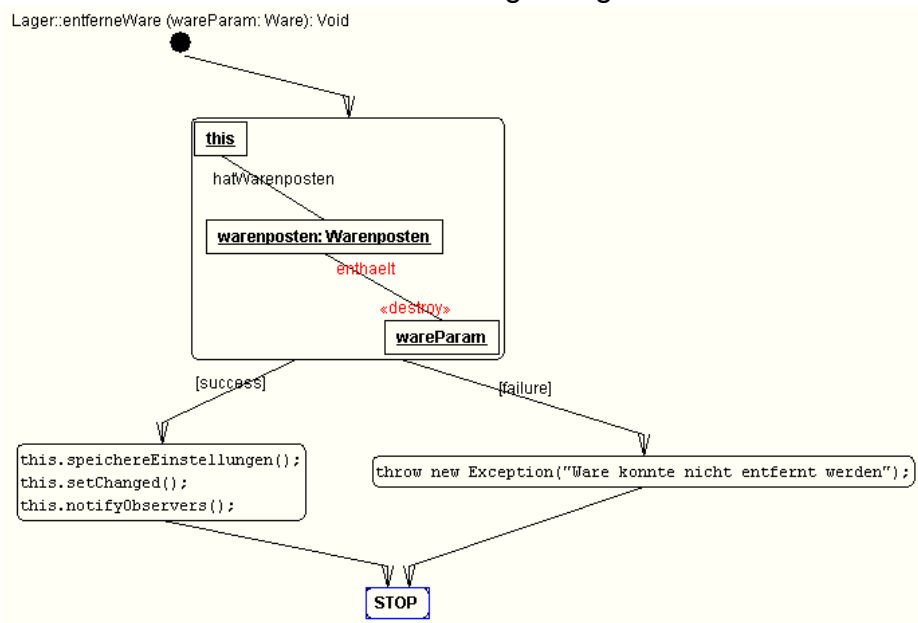
Source Editor [WarenpostenBearbeitenDialog]
204 {
205     try
206     {
207         this.lager.entferneWare(ware);
208         this.warenliste.clearSelection();
209     }
210     catch(Exception ausnahme)
211     {
212         JOptionPane.showMessageDialog(this, ausnahme.getMessage(), "Fehlermeldung");
213         return;
214     }
215 }
  
```

Wurde beim Entfernen der Ware eine Ausnahme ausgelöst, so wird ein Dialog mit der entsprechenden Fehlermeldung am Bildschirm angezeigt.

Nachdem nun klar ist, wie Ausnahmen ausgelöst und behandelt werden, folgt noch eine kurze Erläuterung, wie Fujaba von der Ausnahmebehandlung im generierten Quelltext Gebrauch macht.

4.8.3 Fujaba-Aktivitätsdiagramme und Ausnahmebehandlung

Zur Veranschaulichung sei nochmals das Aktivitätsdiagramm der entferneWare-Methode in der Klasse Lager angeführt:



Das Storypattern mit den Objekten `this`, `warenposten` und `wareParam` wird über die *success*- oder die *failure*-Kante verlassen, je nachdem, ob die in dem Storypattern abgebildete Situation vorgefunden bzw. hergestellt werden konnte oder nicht.

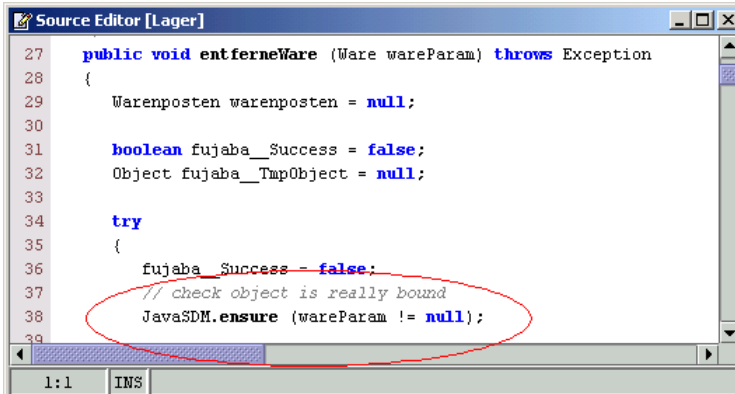
Im Quelltext entspricht das obige Storypattern einem `try`-Block. Wird nun hierin eine Ausnahme ausgelöst, d.h. konnte die beschriebene Situation nicht vorgefunden bzw. hergestellt werden, so wird eine Ausnahme ausgelöst, was automatisch zu einem Sprung in einen `catch`-Block führt und der Verzweigung der *failure*-Kante entspricht.

Eine Ausnahme müsste beispielsweise dann ausgelöst werden, wenn der Parameter `wareParam` `null` ist.

Natürlich ließe sich dies auch so ausdrücken:

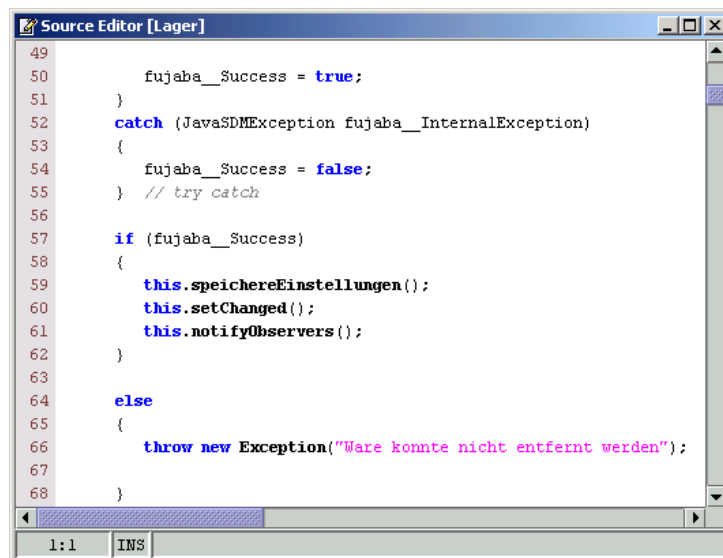
```
if (wareParam == null) throw new Exception();
```

Fujaba jedoch verwendet hierzu die Klasse `JavaSDM` und deren statische Methode `ensure(<<zu überprüfende Bedingung>>)`, die der Intention nach der obigen `if`-Anweisung entspricht:



```
Source Editor [Lager]
27 public void entferneWare (Ware wareParam) throws Exception
28 {
29     Warenposten warenposten = null;
30
31     boolean fujaba_Success = false;
32     Object fujaba_TmpObject = null;
33
34     try
35     {
36         fujaba_Success = false;
37         // check object is really bound
38         JavaSDM.ensure (wareParam != null);
39     }
```

Auf der nächsten Seite folgt das Ende des `try`-Blocks und damit die Umsetzung der *success*- und *failure*-Kante im Quelltext:



```
49     fujaba_success = true;
50   }
51   }
52   catch (JavaSDMException fujaba_InternalException)
53   {
54     fujaba_success = false;
55   } // try catch
56
57   if (fujaba_success)
58   {
59     this.speichereEinstellungen();
60     this.setChanged();
61     this.notifyObservers();
62   }
63
64   else
65   {
66     throw new Exception("Ware konnte nicht entfernt werden");
67   }
68 }
```

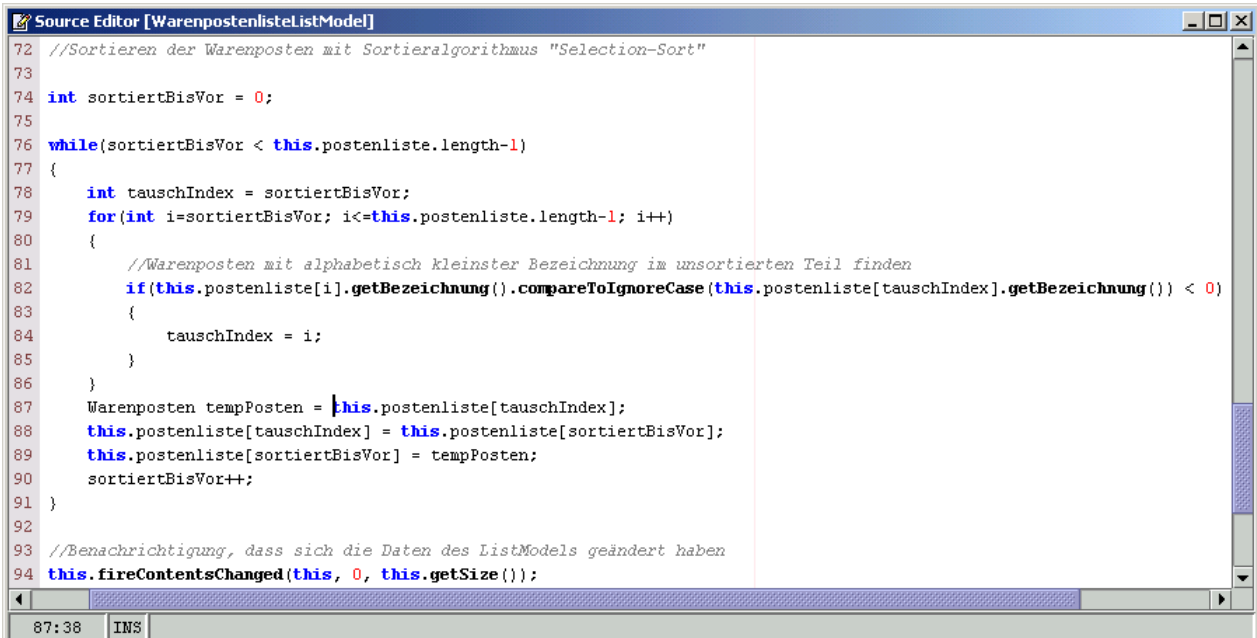
Am Ende des `try`-Blocks wird das Flag `fujaba_success` auf `true` gesetzt, was nur dann geschieht, wenn der ganze `try`-Block ohne das Auslösen einer Ausnahme durchlaufen werden konnte, d.h. die in dem entsprechenden Storypattern beschriebene Situation konnte gefunden bzw. hergestellt werden. Das Auslösen einer Ausnahme im `try`-Block hingegen führt zu einem Sprung in den `catch`-Block und entspricht dem Verlassen des Storypatterns über die *failure*-Kante. Im `catch`-Block schließlich wird das Flag `fujaba_success` auf `false` gesetzt. Alles weitere erklärt sich – das Fujaba-Aktivitätsdiagramm und den obigen Quelltextausschnitt vor Augen – quasi von selbst: Je nach Wert des Flags `fujaba_success` werden entweder die Anweisungen des Statements am Ende der *success*-Kante oder am Ende der *failure*-Kante ausgeführt.

4.9 Algorithmen – Sortieralgorithmus in der Software Schulkiosk

Algorithmen sind als Inhalte des Informatikunterrichts schon traditionell kaum wegzudenken. So gehört für Andreas Schwill die *Algorithmisierung* unbedingt zu den *fundamentalen Ideen der Informatik*, ist von daher elementarer Bestandteil des Informatikunterrichts⁵⁵. Im Rahmen einer Dekonstruktion der Software Schulkiosk sollten also auch Algorithmen Thema sein.

Als Beispiel sehen wir uns die bereits erwähnte Klasse `WarenpostenlisteListModel` an. Sie ist das zur dargestellten Liste (eine `JList` aus dem Package `javax.swing`) der Warenposten in der Klasse `LagerFenster` (in Version 1.0) bzw. `ManagerFenster` (ab Version 1.1) gehörige `ListModel`, stellt also alle im Lager enthaltenen Warenposten für diese Liste bereit und sortiert sie dazu. Der betreffende Sortieralgorithmus ist Bestandteil der `update`-Methode, die immer dann (siehe Observer-Pattern) vom Lager aufgerufen wird, wenn sich der Zustand vom Lager geändert hat, beispielsweise durch Aufnahme oder Löschen eines Objekts der Klasse `Warenposten`.

Hier nun der auf „Sortierung durch Auswählen“⁵⁶ (Selection-Sort) basierende Sortieralgorithmus:



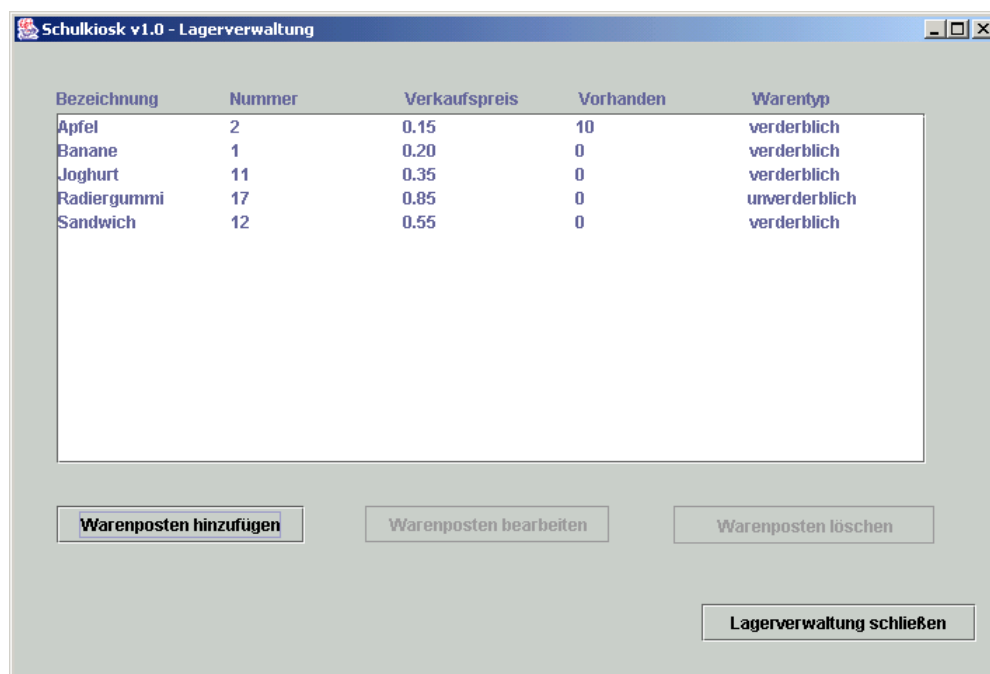
```
72 //Sortieren der Warenposten mit Sortieralgorithmus "Selection-Sort"
73
74 int sortiertBisVor = 0;
75
76 while(sortiertBisVor < this.postenliste.length-1)
77 {
78     int tauschIndex = sortiertBisVor;
79     for(int i=sortiertBisVor; i<=this.postenliste.length-1; i++)
80     {
81         //Warenposten mit alphabetisch kleinster Bezeichnung im unsortierten Teil finden
82         if(this.postenliste[i].getBezeichnung().compareToIgnoreCase(this.postenliste[tauschIndex].getBezeichnung()) < 0)
83         {
84             tauschIndex = i;
85         }
86     }
87     Warenposten tempPosten = this.postenliste[tauschIndex];
88     this.postenliste[tauschIndex] = this.postenliste[sortiertBisVor];
89     this.postenliste[sortiertBisVor] = tempPosten;
90     sortiertBisVor++;
91 }
92
93 //Benachrichtigung, dass sich die Daten des ListModels geändert haben
94 this.fireContentsChanged(this, 0, this.getSize());
```

55 Vgl. Schwill: Fundamentale Ideen der Informatik

56 Vgl. Nievergelt, Jürg; Hinrichs, Klaus H.: Algorithms & Data Structures, 177f.

Erläuterungen zur Funktionsweise des Algorithmus dürften sich seiner Einfachheit wegen wohl erübrigen.

Im Rahmen der Dekonstruktion der Software Schulkiosk im Unterricht bietet es sich an dieser Stelle an, andere Sortierverfahren (z.B. Bubble-Sort, Merge-Sort, Insertion-Sort, Quick-Sort) zu thematisieren und sie auch von den Schülern implementieren zu lassen. Dies entspricht denn auch dem im fachdidaktischen Ansatz der Dekonstruktion geforderten Oszillieren zwischen Dekonstruktion und Konstruktion. Das korrekte Funktionieren des obigen und auch der von Schülern implementierten Sortieralgorithmen lässt sich jedenfalls einfach durch Ausführen der Software Schulkiosk überprüfen:



Bezeichnung	Nummer	Verkaufspreis	Vorhanden	Warentyp
Apfel	2	0.15	10	verderblich
Banane	1	0.20	0	verderblich
Joghurt	11	0.35	0	verderblich
Radiergummi	17	0.85	0	unverderblich
Sandwich	12	0.55	0	verderblich

Eine andere Möglichkeit, das Sortieren und damit Sortieralgorithmen zum Thema des Unterrichts zu machen, ist es, den Schülern eine um den obigen Sortieralgorithmus reduzierte Klasse `WarenpostenlisteListModel` vorzugeben und ihnen das Entwickeln eines eigenen Sortierverfahrens zu überlassen.

5 Resümee

Nachdem nun anhand einiger didaktischer Fenster Ansätze und Möglichkeiten für die Dekonstruktion der Software Schulkiosk im Informatikunterricht angedeutet wurden, bleibt noch der Hinweis darauf, dass im Rahmen der Unterrichtsmethode der Dekonstruktion von Informatiksystemen noch weit mehr möglich ist und die beschriebenen didaktischen Fenster lediglich eine Auswahl darstellen. Die im Rahmen dieser Arbeit entwickelte Software Schulkiosk versucht den Ansprüchen der Dekonstruktion an eine didaktische Software weitgehend gerecht zu werden. Möglich und erwünscht ist es jedoch und zudem, sie um zusätzliche Module und Funktionalität zu erweitern.

Es dürfte deutlich geworden sein, dass für einen Informatikunterricht nach dem fachdidaktischen Ansatz der Dekonstruktion vieles spricht, es sei dazu an dieser Stelle speziell nochmals auf das Kapitel 2 *'Dekonstruktion als Unterrichtsmethode'* verwiesen. Die in Kapitel 4 beschriebenen didaktischen Fenster schließlich sollten je Hinweise auf die Vielzahl von Ansatzpunkten für die Dekonstruktion der Software Schulkiosk sein. Natürlich sind darüber hinaus für eine konkrete Unterrichtsplanung, also den Einsatz der Software im Informatikunterricht, weitere Schritte nötig, was insbesondere die Bereitstellung von Arbeitsmaterialien einschließt.

Den Abschluss dieser Arbeit soll ein Zitat von Johannes Magenheim bilden, der aus der Perspektive einer systemorientierten Didaktik der Informatik darauf verweist, „dass Informatik als einziges Fach (evtl. neben Arbeitslehre) mit technologischen und ingenieurwissenschaftlichen Bezügen an allgemeinbildenden Schulen in der Lage sein könnte, den Schülerinnen und Schülern den Umgang und die Auseinandersetzung mit Technologie insbesondere mit Informationstechnologien näher zu bringen.“⁵⁷ Sein Fazit für die Bedeutung des Informatikunterrichts:

„Informatikunterricht ist damit ein zentrales Fach mit allgemeinbildendem Anspruch in einer postindustriellen Informationsgesellschaft mit einem hohen Grad an Zukunftsrelevanz für Schülerinnen und Schüler.“⁵⁸

⁵⁷ Magenheim: Informatiksystem und Dekonstruktion als didaktische Kategorien

⁵⁸ Ebd.

6 Literatur

Budd, Timothy: *Understanding object-oriented programming with Java*. Reading [u.a] 1998

Gamma, Erich: *Entwurfsmuster. Elemente wiederverwendbarer Software*. Bonn 1996

Hampel, Thorsten; Magenheim, Johannes; Schulte Carsten: *Dekonstruktion von Informatiksystemen als Unterrichtsmethode – Zugang zu objektorientierten Sichtweisen im Informatikunterricht*. In: Schwill, Andreas [Hrsg.]: *Informatik und Schule*. Berlin [u.a.] 1999

Josuttis, Nicolai: *Grenzen. OO-Konferenz: Diskussion um Design und Methodik*. In: *Magazin für professionelle Informationstechnik – iX*, H.12, S. 22, 2001

Magenheim, Johannes: *Informatiksystem und Dekonstruktion als didaktische Kategorien. Theoretische Aspekte und unterrichtspraktische Implikationen einer systemorientierten Didaktik der Informatik*.
http://ddi.uni-paderborn.de/didaktik/Veroeffentlichungen/sytemorientierter_ansatz.pdf
(besucht am 14.5.2002)

Nievergelt, Jürg; Hinrichs, Klaus H.: *Algorithms & Data Structures. With Applications to Graphics and Geometry*. Zürich 1999

Oestereich, Bernd: *Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language*. München/Wien ⁴1998

Oppermann, Reinhard; Reiterer, Harald: *Software-ergonomische Evaluation*. In: Eberleh, Edmund [Hrsg.]: *Einführung in die Software-Ergonomie. Gestaltung graphisch-interaktiver Systeme: Prinzipien, Werkzeuge, Lösungen*. Berlin/New York ²1994

Schwill, A., *Fundamentale Ideen der Informatik*. In: *Zentralblatt für Didaktik der Mathematik - ZDM*, 25, H.1, S. 20-31, 1993

Warren, Nigel; Bishop, Philip: *Java in practice: design styles and idioms for effective Java*. Harlow [u.a] 1999

7 Anlage

CD-Rom mit folgendem Inhalt:

- die Software Schulkiosk
- Übungsbeispiel zu ListSelectionEvents
- JDK 1.3.1
- Fujaba/Life-Version 3.0.0
- Forte for Java in der Community Edition/Version 3.0

8 Versicherung

Versicherung:

Ich versichere, dass ich die schriftliche Hausarbeit selbständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Alle Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem einzelnen Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht. Das gleiche gilt auch für die beigegebenen Zeichnungen, Kartenskizzen und Darstellungen.

Datum _____

Unterschrift _____