

Hausarbeit im Modul Softwaredesign

„Lagerverwaltung“

Marius Mamsch
Gescherweg 165
48161 Münster

Niklas Devenish
Töpferstraße 91
48165 Münster

Jonas Elfering
Hagen 3
48624 Schöppingen

Hochschule Weserbergland
Studiengang: Wirtschaftsinformatik
Studiengruppe: WI44/14
Betreuender Dozent: Prof. Dr. Robert Mertens

Ausbildungsbetrieb:
Fiducia und GAD IT AG
GAD-Straße 2-6
48163 Münster



I Inhaltsverzeichnis

I	Inhaltsverzeichnis.....	I
II	Abkürzungsverzeichnis.....	II
III	Abbildungsverzeichnis.....	II
1	Verwendete Patterns	1
1.1	Model-View-Controller.....	1
1.2	Observable/Oberserver	2
1.3	Command	3
1.4	Strategy.....	3
2	Besonderheiten	4
3	Modelle und Skizzen	6
3.1	Skizzen	6
3.1.1	TreeView und DetailView.....	6
3.1.2	Übersicht aller Buchungen.....	6
3.1.3	Buchungsliste	7
3.2	UML-Modell.....	8

II Abkürzungsverzeichnis

Abkürzung	Beschreibung
GUI	Graphical User Interface
MVC	Model-View-Controller
UML	Unified Modelling Language

III Abbildungsverzeichnis

Abbildung 1: TreeView und DetailView	6
Abbildung 2: Buchungsübersicht	6
Abbildung 3: Buchungsliste mit Details	7
Abbildung 4: UML-Modell	8

1 Verwendete Patterns

1.1 Model-View-Controller

Das grundlegend verwendete Entwurfsmuster bei der Erstellung der Lagerverwaltung ist das Model-View-Controller-Entwurfsmuster (MVC). Dieses Entwurfsmuster ist für diese Anwendung geeignet, da es möglich ist, die fachliche Logik getrennt von der sichtbaren Oberfläche zu bearbeiten. Zudem wird durch das MVC-Entwurfsmuster ermöglicht, einen Zustand des Systems in einer Datei abzuspeichern und zu einem späteren Zeitpunkt neu zu laden.

Die fachliche Logik der Lagerverwaltung wird in verschiedenen Models abgebildet. Insgesamt werden in diesem Fall sechs Models eingesetzt. Das LagerVerwaltungsModel enthält alle notwendigen Methoden, um die Gesamtheit aller Buchungen zu verwalten. Zudem wird die Initialbefüllung der Lager in diesem Model vorgenommen. In Ergänzung zum LagerVerwaltungsModel existiert das LagerModel, welches dazu dient mit einzelnen Lagern zu interagieren und beispielsweise verwendet wird, um eine Buchung auf einzelnes Lager auszuführen. Des Weiteren existiert ein BuchungsModel. Darin sind sowohl alle Informationen, wie beispielsweise Anteile und Buchungstag, als auch die benötigten Methoden zur Erstellung und Verwaltung von Buchungen enthalten. Das BuchungsModel dient als abstrakte Oberklasse. Die konkrete Implementation der Methoden für eine Zu- oder Abbuchung erfolgt in zwei eigenen Models, dem ZuBuchungsModel und dem AbBuchungsModel. Abschließend existiert ein AnteilModel, welches benötigt wird, um eine Gesamtbuchung als Buchung mehrerer kleiner Anteile auf verschiedene Lager darstellen zu können. In Kombination mit dem Command-Pattern kann so auch das Speichern der Anteile ermöglicht werden.

Zur graphischen Darstellung der Informationen werden diverse View-Elemente genutzt. Diese View-Elemente kommunizieren über den LagerVerwaltungsController mit den Models. In dem Lagerverwaltungscontroller sind ebenfalls die Methoden für das Speichern, das Laden und einen Undo/Redo-Mechanismus implementiert. Das übergeordnete View-Element ist die VerwaltungsView, welche weitere View-Elemente enthält. Ein Beispiel dafür ist die detaillierte Übersicht eines Lagers. Die Informationen, wie viel Kapazität ein Lager besitzt oder welche Buchungen auf dieses Lager getätigt wurden, erhält diese DetailView vom Lagerverwaltungsmodel und dem

LagerModel aufgrund einer Observable/Oberserver-Beziehung. Der LagerVerwaltungsController dient dazu, den Models vorzugeben, welche View-Elemente sie benachrichtigen müssen, wenn sich ihr Informationsstand geändert hat. Falls der Benutzer beispielsweise eine neue Buchung tätigt, wird diese Information von der View über den LagerVerwaltungsController an die Models weitergegeben, dort wird der Informationsstand der Models aktualisiert und die neuen Informationen können in der View angezeigt werden.

Ein großer Vorteil bei dieser Vorgehensweise den Controller als Vermittlungsstelle zwischen Benutzeroberfläche und fachlicher Logik zu verwenden, ist die Unabhängigkeit von Logik und Oberfläche voneinander. Es ist möglich ein Model zu verändern, ohne dass die Benutzeroberfläche angepasst werden muss. Dies gilt auch umgekehrt. Des Weiteren ist es möglich den aktuellen Informationsstand der Models zu speichern und zu einem späteren Zeitpunkt neu zu laden, auch wenn sich der Informationsstand der Models verändert hat.

1.2 Observable/Oberserver

In der GUI wurde das Observable/Observer Pattern genutzt, um Veränderungen in den Models angemessen in der GUI darzustellen. Das LagerVerwaltungsModel implementiert das Interface Observable und wird von der LagerVerwaltungsView, die den Observer erweitert, observt. Das dient vor allem dazu, Änderungen an der laufenden Buchung im LagerVerwaltungsModel zu erkennen und die GUI hinsichtlich dieser Änderungen zu aktualisieren. Aber auch auf Änderungen der Struktur der Lager und auf Änderungen in der Buchungsliste wird hier geachtet, damit diese immer korrekt angezeigt werden.

Des Weiteren werden Observable/Observer-Beziehungen bei den LagerModels, die das Interface Observable implementieren, eingesetzt. So erweitert der LagerBaumKnoten den Observer und jeder Knoten überwacht ein Lager, welches er in der Baumstruktur repräsentiert. Hierdurch wird es ermöglicht, dass die Knoten immer den aktuellen Bestand des Lagers anzeigen, ohne immer den kompletten Baum aktualisieren zu müssen. Außerdem überwacht die DetailView als Observer das LagerModel, welches momentan angezeigt wird. Dadurch wird auch ermöglicht, dass die DetailView die aktuellen Informationen zum Lager anzeigt, ohne ständig neu erstellt oder komplett aktualisiert zu werden.

Eine weitere Observable/Observer-Beziehung ist zwischen dem Controller und der BuchungsBar zu finden. Der Controller ist in diesem Fall das Observable und die BuchungsBar der Observer. In diesem Fall teilt der Controller seinem Observer mit, wenn sich der Redo- oder der Undo-Stack geändert haben. Aufgrund von diesen Informationen entscheidet die BuchungsBar, ob die Buttons für den Redo-, bzw. Undo-Mechanismus aktiviert oder deaktiviert werden.

1.3 Command

Für die Realisierung eines Redo- und Undo-Mechanismus wird das Command Pattern verwendet. Der Controller verwaltet zwei Stacks. Einen Stack für die Redo-Commands und einen für die Undo-Commands. Der Command besitzt eine Execute- und eine Undo-Methode. Der Controller führt die Execute-Methode aus, um einen Anteil zu erzeugen bzw. die Undo-Methode um einen Anteil zu löschen. Wird ein Command erzeugt wird die Execute-Methode dieses Commands aufgerufen und das Command dem Undo-Stack hinzugefügt. Außerdem wird der Redo-Stack gelöscht. Beim Undo-Mechanismus wird die Undo-Methode des obersten Commands im Undo-Stack aufgerufen. Das Command wird aus dem Undo-Stack gelöscht und dem Redo-Stack hinzugefügt. Analog funktioniert der Redo-Mechanismus.

Vorteile bei der Anwendung des Command-Patterns sind die Modularität und Wiederverwendbarkeit von Befehlsobjekten sowie die Vermeidung von Coderedundanz und Inkonsistenzen durch zentrale Befehlsobjekte.

1.4 Strategy

Das Entwurfsmuster Strategy wurde im Kontext des Sortierers eingesetzt. Der Sortierer besitzt eine Sortierstrategie. Die Sortierstrategie ist ein Interface, welches die Methode `sortiere()` bereitstellt. Dieser Methode müssen zwei `BuchungsModel` mitgegeben werden, die verglichen werden sollen. Die Methode `sortiere()` liefert „true“ zurück wenn das zuerst mitgegebene `BuchungsModel` vor dem zweiten `BuchungsModel` angezeigt werden muss, ansonsten „false“. Es gibt mehrere konkrete Implementationen dieses Interface. Für jede mögliche Art die Buchungsliste zu sortieren jeweils eine. Diese konkreten Implementationen unterscheiden sich vor allem durch das Attribut nach welchem sie sortieren und durch die Sortierreihenfolge, wobei beides am Namen der Implementationen erkennbar ist. Das Entwurfsmuster

Strategy bietet den Vorteil, dass man nur einen Sortieralgorithmus implementieren muss, der mit dem Interface Sortierstrategie arbeitet, um alle Sortiermöglichkeiten abzubilden. Dadurch kann die konkrete Implementation der Sortierstrategie noch zur Laufzeit geändert werden. Diese Implementation ist sehr viel eleganter, als alternativ für jede Möglichkeit des Sortierens einen kompletten Sortieralgorithmus zu implementieren, denn das würde redundanten Code erzeugen.

2 Besonderheiten

Unsere Gruppe hat als dreier-Gruppe auch die Aufgaben für vierer-Gruppen bearbeitet. So ist es mit unserer Lagerverwaltung möglich beliebig viele Lager an jeder Stelle zu erstellen. Des Weiteren ist es möglich Lager zu löschen. Dafür sind jedoch bestimmte Voraussetzungen zu erfüllen. Außerdem ist es möglich den aktuellen Stand der Lagerverwaltung zu speichern und einen gespeicherten Stand wieder zu laden.

Die Ausarbeitung unseres Löschens wird beispielhaft an der folgenden Lagerstruktur erklärt.

- Lager 1
 - Lager 1.1
 - Lager 1.1.1
 - Lager 1.2
 - Lager 1.2.1
 - Lager 1.2.2
- Lager 2

Wird ein Lager gelöscht, welches Unterlager besitzt, wandern diese Unterlager eine Ebene nach oben. In diesem Beispiel wären das die Lager 1, 1.1 und 1.2.

Wird ein Lager, welches keine Unterlager besitzt, gelöscht, dann gibt es drei verschiedene Konstellationen. Besitzt das Lager ebenfalls kein Oberlager, muss es leer sein, um gelöscht werden zu können. In diesem Beispiel wäre das das Lager 2. Sind auf der Ebene des zu löschenden Lagers keine anderen Lager, so wird es gelöscht und die Anteile der Buchungen und der Bestand auf das Lager darüber übertragen. In diesem Beispiel ist das das Lager 1.1.1. Sind stattdessen auf der gleichen Ebene noch weitere Lager, kann das Lager nur gelöscht werden, wenn das Lager leer ist. In diesem Beispiel wären das die Lager 1.2.1 oder 1.2.2.

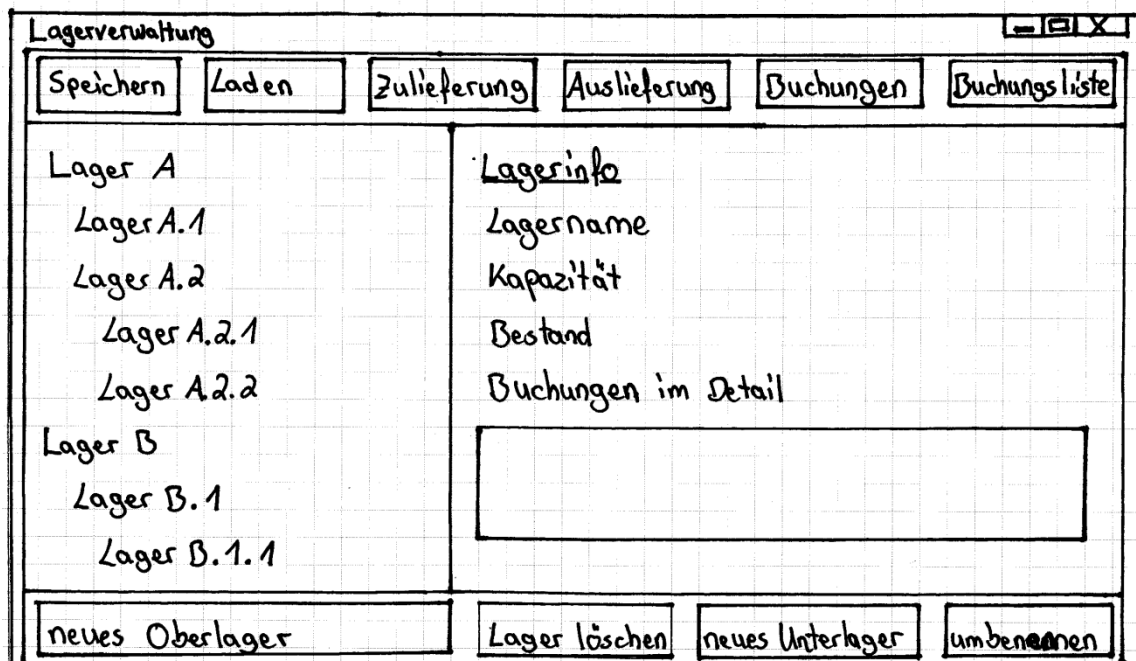
Des Weiteren wurden JUnit-Tests erstellt. Diese Tests gewährleisten die Kernfunktionalitäten der Models. Viele Methoden werden implizit durch den Test der

Funktionalität des LagerVerwaltungsModels getestet. Das Programm wurde ausführlich getestet. Es wurden Akzeptanztests und Oberflächentests durchgeführt.

3 Modelle und Skizzen

3.1 Skizzen

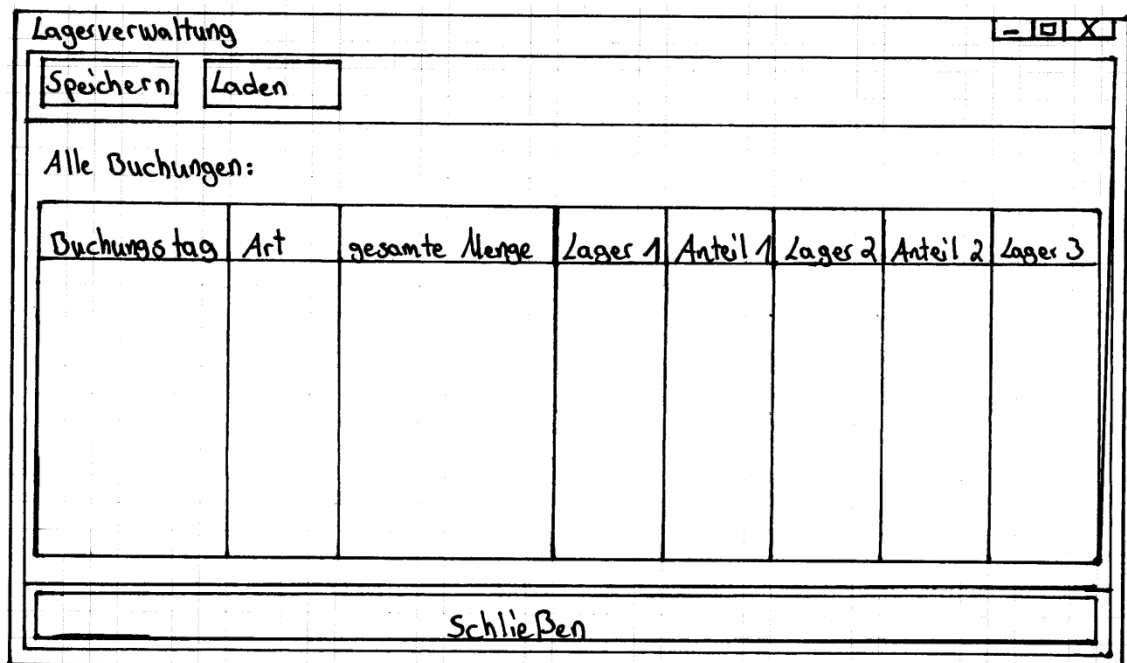
3.1.1 TreeView und DetailView



The sketch shows a window titled 'Lagerverwaltung' with standard window controls. Below the title bar is a menu bar with buttons: 'Speichern', 'Laden', 'Zulieferung', 'Auslieferung', 'Buchungen', and 'Buchungsliste'. The main area is split into two panes. The left pane contains a tree structure of warehouses: 'Lager A' (with sub-items 'Lager A.1' and 'Lager A.2', where 'Lager A.2' has further sub-items 'Lager A.2.1' and 'Lager A.2.2'), and 'Lager B' (with sub-items 'Lager B.1' and 'Lager B.1.1'). The right pane is titled 'Lagerinfo' and contains labels for 'Lagername', 'Kapazität', and 'Bestand', followed by the text 'Buchungen im Detail' and a large empty rectangular box. At the bottom of the window is a bar with four buttons: 'neues Oberlager', 'Lager löschen', 'neues Unterlager', and 'umbenennen'.

Abbildung 1: TreeView und DetailView

3.1.2 Übersicht aller Buchungen



The sketch shows a window titled 'Lagerverwaltung' with 'Speichern' and 'Laden' buttons. Below these is the text 'Alle Buchungen:'. A table follows with the following headers: 'Buchungstag', 'Art', 'gesamte Menge', 'Lager 1', 'Anteil 1', 'Lager 2', 'Anteil 2', and 'Lager 3'. The table body is empty. At the bottom of the window is a button labeled 'Schließen'.

Abbildung 2: Buchungsübersicht

3.1.3 Buchungsliste

Lagerverwaltung

Speichern

Laden

Buchungstag

Gesamte Menge

Buchungstag

Gesamte Menge

Buchungstag

Gesamte Menge

Buchungsdetails:

Buchungstag

Gesamte Menge

Lager 1

Lager 2

Lager 3

Lager 4

Schließen

Abbildung 3: Buchungsliste mit Details

