

**Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering**

Werkzeug zur Unterstützung von Interviews in der Prozessmodellierung

Bachelorarbeit

im Studiengang Informatik

von

Carl Volhard

**Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Prof. Dr.-Ing. Christian Grimm
Betreuer: Prof. Dr. Kurt Schneider**

Hannover, 18. Oktober 2006

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst habe. Es wurden keine anderen, als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet.

Hannover, 18.10.2006 Carl Volhard

Zusammenfassung

Use Cases dienen im Requirements Engineering zur Beschreibung einzelner Funktionen des vorhandenen (für Ist-Zustand) oder zu erstellenden Systems (für Soll-Zustand) aus Sicht der Benutzer. Dadurch sollen die Anforderungen an das System besser erhoben und dokumentiert werden. Durch Reflektion der erfassten Daten können weitere Anforderung aufgedeckt und formuliert werden. Oberflächenskizzen sind Grobentwürfe der vom System zur Benutzerinteraktion angezeigten Dialoge. Meist werden sie während eines Kundengesprächs gezeichnet, um dem Kunden eine ungefähre Vorstellung der späteren Benutzerschnittstelle zu geben. In Kombination mit den zugehörigen Use Cases können künftige Benutzer schon vorab durch das System geführt werden, und somit Schwachstellen in der Oberfläche als auch falsche oder fehlende Anforderungen aufgedeckt werden.

Zwischen dem Erfassen der Daten für Use Cases und Oberflächenskizzen und dem Auswerten dieser Daten gehen oftmals Informationen verloren oder werden im Nachhinein falsch interpretiert. Daher wäre es wünschenswert, am Ende des Interviews Use Cases und Oberflächenskizze soweit fertig gestellt zu haben, dass diese vom Kunden zu validiert werden können.

In dieser Bachelorarbeit wurde das Programm *MockUp Fast* entwickelt. Dieses Programm erlaubt es Use Cases zu erfassen, die dazu gehörigen Bedienoberflächen zu skizzieren und diese miteinander zu verknüpfen. Über diesen Zusammenhang zwischen Use Case und Oberfläche kann das spätere Programm simuliert werden.

Zusätzlich ermöglicht es das Erfassen der Use Cases und das Skizzieren der Oberflächen während eines Kundengesprächs durch zu führen. Dies soll eine unmittelbare Validierung durch den Kunden ermöglichen. Somit lassen sich weitere Kundengespräche zur Klärung ersparen.

Inhaltsverzeichnis

1	Einleitung	5
1.1	Motivation	5
1.2	Zielsetzung	5
1.3	Gliederung	6
2	Aufgabenstellung	7
2.1	Hauptanforderungen	7
2.2	Oberflächenskizzen	7
2.3	Anforderungen an die Simulation	7
3	Verknüpfung von Use Cases und Oberflächenskizzen	9
4	Vorbereitung	13
4.1	Sprachwahl	13
4.1.1	Unterschiede von C# im Vergleich zu Java	13
4.2	Prototypen	16
4.2.1	Zeichnen und Schrifterkennung auf Zeichenfläche	16
4.2.2	Ausfüllen des Use Cases	17
5	Entwicklung des Modells	19
5.1	Aufbau des darzustellenden Use Case	19
5.2	Klassen des Modells	20
5.2.1	UseCase	20
5.2.2	UseCaseProjekt	21
5.2.3	Action	21
5.2.4	Stakeholder	22
5.2.5	MockUp	22
5.2.6	MockUpBereich	22
5.2.7	MockUpBeschriftung	23
5.3	Zusammenspiel des Modells	23
5.3.1	Design By Contract	23
5.4	Grenzen des Modells	25
5.4.1	Globale Erweiterungen	25
5.4.2	Zuordnung der MockUps zu einem UseCase	25
6	Entwicklung der Oberfläche	27
6.1	Bedienbarkeit	27
6.1.1	Übersichtlichkeit und Aufbau	29
6.2	Intelligente Oberfläche	30
6.3	Umsetzung auf Tablet-PC	30
6.4	MVC	30
6.5	BeschreibungsTable und ErweiterungsTable	31

7	Betrachtung des fertigen Programms	34
7.1	Bedienung mit Tablet-PC.....	34
7.1.1	Zeichnen der Oberflächenskizzen	34
7.1.2	Ausfüllen der Use Cases	34
7.2	Export der Use Cases.....	35
8	Bedienungsanleitung.....	36
8.1	Voraussetzungen.....	36
8.2	Installation	36
8.3	Benutzung	37
9	Glossar	41
10	Quellenangaben	42

1 Einleitung

1.1 Motivation

Kundengespräche während der Anforderungsanalyse kosten sowohl auf Auftragsgeber- als auch auf Entwickler-Seite Zeit und Geld.

Bei diesen können – zum Beispiel wegen unterschiedliche Fachsprachen und irrtümlich angenommener Selbstverständlichkeiten – Missverständnisse auftreten. Die im Nachhinein aufgearbeiteten Gesprächsprotokolle können falsch interpretiert werden. Im besten Fall werden die daraus resultierenden falschen Anforderungen bei weiteren Gesprächen aufgedeckt, die aber das Zeit- und Geldbudget belasten. Häufiger es ist jedoch der Fall, dass die falschen Anforderungen erst bei Auslieferung des Programms offenbar werden. Wünschenswert wäre also ein Werkzeug, mit dem während des Kundengesprächs die Ergebnisse so festgehalten werden, dass man sie im direkten Anschluss zusammen mit dem Kunden überprüfen kann.

Interaktionen von Benutzern mit Programmen finden in der Regel über graphische Benutzeroberflächen statt. Oberflächenskizzen können einen groben Einblick in die Bedienung der späteren Software ermöglichen.

Ein reines Zeigen und Besprechen der Oberflächenskizzen gestattet jedoch keine Interaktion durch den Kunden, so bleibt die Gefahr von Missverständnissen bestehen. Kann hingegen der Kunde mit den Oberflächenskizzen aktiv agieren, können Missverständnisse aufgedeckt und geklärt werden. Dazu muss die in den Use Case vorhandene Information über den Ablauf der jeweiligen Teilaufgabe in die Oberflächenskizzen einfließen.

Eine Verknüpfung zwischen Use Cases und den Oberflächenskizzen kann also zur Simulation der zu entwickelnden Software verwendet werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist, es ein Werkzeug zu entwickeln, das es erlaubt, die in einem Use Case beschriebenen Interaktion zwischen Benutzer und System mit den Skizzen der Oberflächen für diese Interaktionen zu verknüpfen. Das Werkzeug soll es ermöglichen, das Erstellen des Use Cases und der Oberflächenskizze sowie die Verknüpfung zwischen den beiden während eines einzigen Kundengesprächs durchzuführen. Dies erfordert, dass die Bedienung des Programms hinreichend einfach und schnell geht, damit das Kundengespräch nicht beeinträchtigt wird.

Nachdem die Use Cases erstellt und mit den Oberflächenskizzen verknüpft sind, soll es möglich sein, eine Simulation der erfassten Abläufe zu starten, in der der Benutzer Eingaben im Sinne der simulierten Teilaufgabe tätigen kann.

1.3 Gliederung

In Kapitel 2 wird die Aufgabenstellung und die daraus resultierenden Anforderungen beschrieben.

In Kapitel 3 **Verknüpfung von Use Cases und Oberflächenskizzen** wird auf die Problemlösung bei der Visualisierung der Verknüpfungen zwischen Use Case und Oberflächenskizze eingegangen.

In Kapitel 4 wird auf die Vorbereitungen eingegangen, die für die Bachelor-Arbeit stattfanden. Diese betrafen in erster Linie die Entwicklung für den Tablet-PC und die Wahl der geeigneten Programmiersprache. Anschließend folgte die Entwicklung einiger Prototypen, auf die in diesem Kapitel auch eingegangen wird.

Im Kapitel 5 **Entwicklung des Modells** wird beschrieben, wie das Modell aus dem gewünschten Aufbau der Use Cases und deren Zusammenspiel mit den Oberflächenskizzen entwickelt wurde. Die Beschränkungen, die das entstandene Modell mit sich bringt, werden ebenfalls erläutert.

Im ersten Teil von Kapitel 6 **Entwicklung der Oberfläche** werden die verwendeten Entwicklungsmethoden erläutert. Im zweiten Teil wird näher auf die eigentliche Entwicklung der Oberfläche und auf die aufgetretenen programmier- und designtechnischen Probleme eingegangen.

In Kapitel 7 wird auf das fertige Programm eingegangen und beurteilt, in wie weit die Erwartungen an den Bedienungsvorteil durch ein Tablet-PC erfüllt werden konnten. Außerdem wird auf mögliche Verbesserungen und Erweiterungen eingegangen.

Kapitel 8 enthält die Bedienungsanleitung für das entwickelte Programm *MockUp Fast*.

Im Glossar beschriebene Begriffe sind kursiv dargestellt. Bei kursiven Begriffen, die nicht im Glossar auftauchen, handelt es sich um Klassen des Programms *MockUp Fast*. Eine Erläuterung zu diesen Klassen findet sich in Kapitel 5 **Entwicklung des Modells** und 6 **Entwicklung der Oberfläche**.

2 Aufgabenstellung

2.1 Hauptanforderungen

Das Programm *MockUp Fast* soll die Zeit, die für Kundengespräche aufgewendet werden muss, verringern und gleichzeitig die Gefahr von missverstanden oder fehlenden Anforderungen ausschließen.

Dazu muss das Programm *MockUp Fast* folgende Anforderung erfüllen:

- Erstellen von Use Cases
- Skizzieren von Oberflächen
- Simulation des zu erstellenden Programms mit Hilfe der Informationen aus den erfassten Use Cases und Oberflächenskizzen.
- Der gesamte Vorgang soll während eines Kundengesprächs stattfinden. Die Simulation soll vom Kunden selbst ausgeführt werden.
- Das Programm *MockUp Fast* darf das Kundengespräch nicht behindern und muss deshalb schnell bedienbar sein. Aus diesem Grund soll die Anwendung für einen Tablet-PC erstellt werden, der handschriftliche Eingaben akzeptiert.

2.2 Oberflächenskizzen

Die Art der Oberflächenskizzen sollte im Stil von Freihand-Skizzen auf Papier oder Whiteboards sein. Für die Simulation wäre vielleicht eine Oberfläche besser verständlich, die aus vorgefertigten Bedienelementen besteht. Allerdings besteht dann die Gefahr, dass das fertige Aussehen der Oberfläche das Gespräch zu sehr auf das Design der Oberfläche lenkt. Dies ist zu einem Zeitpunkt, in dem erst die eigentlichen Funktionen und Anforderungen der zu entwickelnden Software erhoben werden, nicht wünschenswert.

Zudem ist zu erwarten, dass das Freihandzeichnen der Oberflächen schneller geht, vor allem bei der Verwendung eines Tablet-PCs. Das Drag und Drop von fertigen Bedienelementen wäre zwar eine Möglichkeit, wird jedoch einerseits die Anzahl dieser Elemente zu hoch, leidet die Erreichbarkeit (Navigation durch mehrere Menü-Ebenen). Andererseits müssten fehlende Elemente mühselig per Hand nach gebaut werden.

Es sollte möglich sein, handschriftlichen Text auf den Oberflächenskizzen in Maschinentext umzuwandeln. Dies widerspricht zwar dem Skizzencharakter, erhöht aber die Übersichtlichkeit und Lesbarkeit der Oberflächenskizzen erheblich. Zudem lassen sich so während der Simulation die Benutzereingaben von den Beschriftungen der Oberfläche unterscheiden.

2.3 Anforderungen an die Simulation

Beschreibungen und Erweiterungen sind Teile des Use Case. Sie behandeln die Interaktion zwischen Benutzer und System. Werden sie mit den Oberflächenskizzen verknüpft, lässt sich die Benutzerschnittstelle simulieren.

Dabei soll dem Kunden die Möglichkeit geben werden, Eingaben im Sinne des Use Cases zu tätigen. Widerspricht die Eingabe den Erwartungen der Interviewer, offenbart sich Klärungsbedarf. Falls dem Kunden nicht klar sein sollte, was in diesem Schritt genau von ihm gefordert ist, erlaubt dies Rückschlüsse auf die Intuitivität der Oberflächen.

Für die Simulation eines Use Cases sollen die einzelnen Beschreibungen in ihrer Reihenfolge abgearbeitet werden. Zu jeder Beschreibung soll die verknüpfte Oberflächenskizze angezeigt werden. Der Bereich, in dem sich die Beschreibung auf der Oberflächenskizze abspielt, soll dabei hervorgehoben werden. Zusätzlich soll der Inhalt der Beschreibung angezeigt werden. Dieser Inhalt und die Hervorhebung des Bereiches auf der Oberfläche sollen ausgeblendet werden können. Dies soll den Erkenntnisgewinn über die Intuitivität der Oberfläche erhöhen.

Nach der Abhandlung einer Beschreibung wird zur nächsten Beschreibung gewechselt. Frühere Eingaben sollen auf späteren Oberflächenskizzen angezeigt werden, um den Ablauf besser nachvollziehbar zu machen.

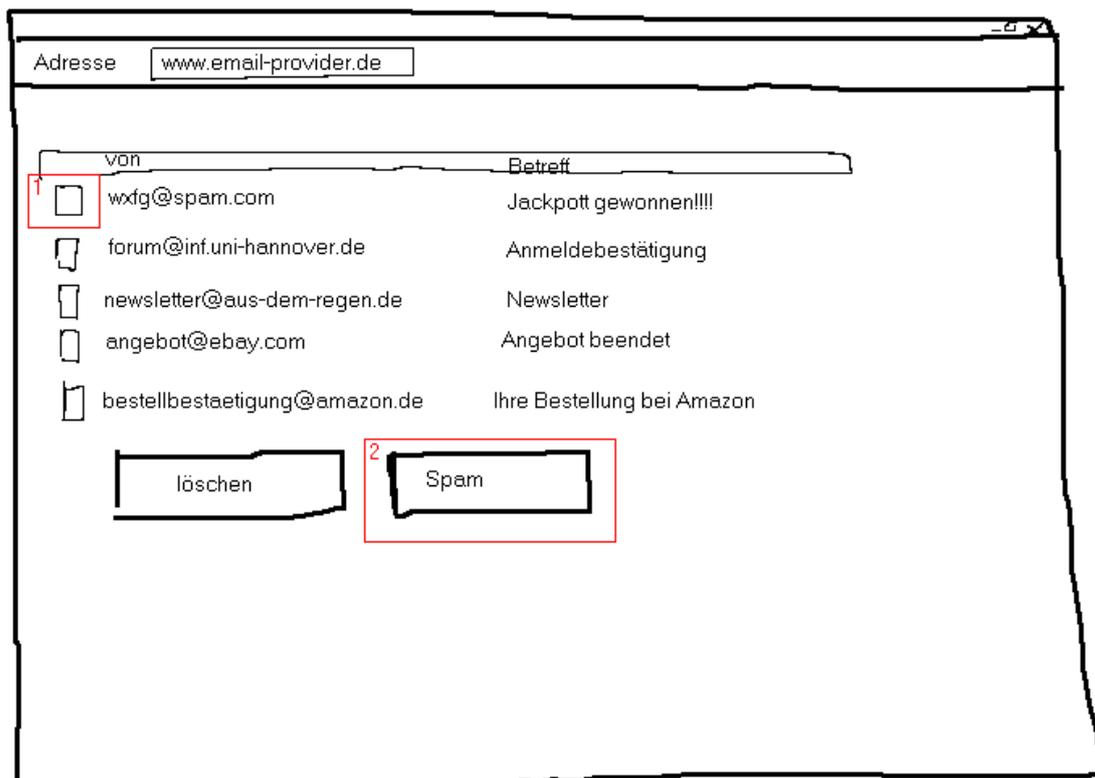
Während der Entwicklung des Programms *MockUp Fast* traten weitere Anforderungen zu Tage. Auf diese wird im Kapitel 5 Entwicklung des Modells eingegangen.

3 Verknüpfung von Use Cases und Oberflächen-skizzen

Im Folgenden wird der Begriff **Aktion** für Beschreibungen und Erweiterungen verwendet. Dialoge lassen in der Regel mehr als die Interaktion zu, die in einer Aktion beschrieben wird. Daher muss bei der Verknüpfung deutlich gemacht werden welche Bedienelemente durch die Aktion betroffen sind.

Beispiel für eine Oberfläche mit Verknüpfungen zu mehreren Aktionen:

	1.1	Email als Spam deklarieren	
Umfeld		Webseite des Email-Providers	
[...]			
Beschreibung	Wer handelt	Aktion	
	1	Benutzer	markiert die Spam-Mail
	2	Benutzer	drückt auf den "Spam" Button



Zur Realisierung der Verknüpfung bieten sich folgende Ansätze:

1. Für jede Aktion werden nur die Bedienelemente der Oberfläche skizziert, die zur Ausführung der Aktion nötig sind.
2. Für jede Aktion wird eine komplette Oberflächenskizze erstellt. Die zur Ausführung der Aktion nötigen Bedienelemente werden hervorgehoben.

3. Oberflächenskizzen können mit mehreren Aktionen eines Use Cases verknüpft werden. Dabei werden die zur Ausführung nötigen Bedienelemente so hervorgehoben, dass deutlich wird, bei welchen Aktionen sie benötigt werden. Dieser Lösungsansatz wurde gewählt.

ad 1.: Bei unvollständigen, nur die betroffenen Bedienelemente zeigenden Skizzen wäre unklar, wie die einzelnen Dialoge in ihrer Gesamtheit aussehen. Für die Entwicklung des Programms ließe sich dies eventuell noch aus dem Zusammenhang der Use Cases schließen. Eine an das Interview anschließende Validierung mittels Simulation durch den Kunden wäre jedoch sehr schwierig, da dem Kunden keine vollständigen Oberflächenskizzen präsentiert werden können.

Algorithmen, die aus den unvollständigen Oberflächenskizzen eine komplette Skizze fertigen, sind ohne zusätzliche Information über die Beschaffenheit dieser Skizze nicht realisierbar. Diese Information muss vom Benutzer eingegeben, was wiederum zusätzlichen Aufwand beim Erstellen der Oberflächenskizzen bedeutet. Daher ist dieser Lösungsansatz nicht praktikabel.

ad 2.: Bei diesem Lösungsansatz müsste für jede Aktion eine neue Oberflächenskizze gezeichnet werden. Bei Dialogen die in mehr als einer Aktion auftauchen, müssten in diesem Fall Oberflächenskizzen doppelt gezeichnet werden. Doppelte Oberflächen bedeuten jedoch auch doppelten Aufwand. Außerdem müssten Änderungen an einer Oberfläche auch an allen entsprechenden Oberflächen durchgeführt werden. In der ansonsten inkonsistenten Skizzensammlung wäre sonst später nicht mehr erkennbar, welches die gültige Oberflächenskizze ist.

Wenn das Programm *MockUp Fast* die Möglichkeit bieten würde, Oberflächen zu kopieren, würde das mehrfache Erstellen derselben Oberfläche kaum zusätzlichen Aufwand mit sich bringen. Zusätzlich kann das Programm in diesem Fall Änderungen an einer Oberfläche bei allen anderen gleichen Oberflächen übernehmen. Dies würde jedoch einen zusätzlichen Implementierungsaufwand nach sich ziehen. Zu dem würde die Komplexität des Modells erhöhen. Aus diesen Gründen wurde dieser Lösungsansatz nicht gewählt.

ad 3.: Bei diesem Lösungsansatz werden Oberflächenskizzen zunächst separat erstellt. Im Nachhinein werden verschiedene Aktionen mit dieser Oberfläche verknüpft. Insofern können (1) vollständige Oberflächenskizzen gezeichnet werden, ohne dass diese (2) doppelt für einen Use Cases gezeichnet werden müssen. Bei der Hervorhebung der verknüpften Bedienelemente muss nun jedoch deutlich gemacht werden, welche Bedienelemente mit welcher Aktion verknüpft sind. Die Hervorhebung der Bedienelemente ist daher ausschlaggebend für die Bewertung dieses Lösungsansatzes. Im Folgenden werden deshalb drei mögliche Lösungen für die Hervorhebung diskutiert:

- a. Aktionen können über ihre Nummer innerhalb eines Use Cases identifiziert werden. Die Nummer der Aktion wird neben den mit dieser Aktion verknüpften Bedienelementen platziert.

- b. Jede Aktion erhält bei der Darstellung eine eigene Farbe. Bedienelemente, die mit einer Aktion verknüpft sind, werden in der Farbe dieser Aktion dargestellt.
- c. Hervorhebung der Bedienelemente durch einen Rahmen, der alle zu einer Aktion gehörigen Elemente umfasst. Jeder Rahmen erhält zur Unterscheidung eine Identifikation für die verknüpfte Aktion. Dieser Lösungsansatz wurde für die Umsetzung gewählt.

ad a.: Bei diesem Lösungsansatz entsteht ein Problem, wenn mehrere Bedienelemente von einer Aktion benutzt werden. Um deutlich zu machen, welche Bedienelemente betroffen sind, müsste neben jedem betroffenen Bedienelement die Nummer der verknüpften Aktion platziert werden. Dies würde jedoch zu einer Überladung der Oberflächenskizze führen. Zudem kostet die Platzierung mehrere Identifikationen zusätzlichen Bedienungsaufwand. Kann nur eine Platzierung vergeben werden, dürfen die Aktionen nur mit einem Bedienelement verknüpft werden. Aktionen die sich auf mehrere Bedienelemente beziehen, müssten unterteilt werden. Das Erstellen würde mehr Zeit kosten und die Aktion wären detaillierter als nötig beschrieben. Aus diesen Gründen wurde dieser Lösungsansatz nicht gewählt.

Beispiel für eine durch mehrere Bedienelemente erzwungene Unterteilung einer Aktion:

Bei dem folgenden Use Case und der zugehörigen Oberflächenskizze lässt sich die Identifikation nicht sinnvoll platzieren.

	2.1 Ändern der Kontakt-Email	
[...]		
Beschreibung	Wer handelt	Aktion
[...]		
	4 Benutzer	gibt neue Kontakt-Email ein und klickt auf OK

Um die Identifikationen sinnvoll zu platzieren müsste die betroffene Aktion unterteilt werden, wie im folgenden Beispiel dargestellt.

	2.1 Ändern der Kontakt-Email	
[...]		
Beschreibung	Wer handelt	Aktion
[...]		
	4 Benutzer	gibt neue Kontakt-Email ein
	5 Benutzer	klickt auf OK

Eine Aktion, die beispielsweise den Ablauf einer Benutzeranmeldung mit "Benutzer: Gibt Benutzernamen und Passwort ein" beschreibt, müsste in zwei Aktionen unterteilt werden, nämlich in (1) "Benutzer: Gibt Benutzernamen ein" und in (2) "Benutzer: Gibt Passwort ein".

ad b.: Bei diesem Lösungsansatz ist die Farbwahl ein Problem. Da die Anzahl der Aktionen innerhalb eines Use Cases nicht beschränkt ist, müsste eine nicht beschränkte Palette an Farben zur Verfügung gestellt werden. So kann es passieren, dass zwar unterschiedliche Farben verwendet werden, die das menschliche Auge jedoch nicht unterscheiden kann. Dieses Problem wird durch die individuelle Farbwahrnehmung und die unterschiedliche Farbdarstellung verschiedener Anzeigegeräte noch verstärkt.

Zudem ist es möglich, dass ein Bedienelement mit mehreren Aktionen verknüpft ist. Dieses Bedienelement müsste also in unterschiedlichen Farben dargestellt werden. Eine *Stroke* kann aber in nur einer Farbe dargestellt werden, es sei denn man teilt sie auf und stellt die Bestandteile "bunt" dar. Dabei muss jedoch drauf geachtet werden, dass diese Bestandteile groß genug sind, dass ihre Einzelfarben erkennbar bleiben. Eine knallbunte Oberflächenskizze wäre zudem nicht sonderlich übersichtlich. Deshalb wurde dieser Lösungsansatz nicht gewählt.

ad c.: Hebt ein Rahmen die verknüpften Bedienelemente hervor, kann sichergestellt werden, dass alle betroffenen Elemente markiert sind. Es kann passieren, dass innerhalb des Rahmens auch Bedienelemente liegen die nicht mit dieser Aktion verknüpft werden sollen. Das kommt jedoch selten vor und wurde daher in Kauf genommen.

Ein Rahmen lässt sich zudem durch markieren zweier Eckpunkte schnell zeichnen.

4 Vorbereitung

4.1 Sprachwahl

Eine Basisentscheidung für jedes Softwareprojekt ist die Wahl der zu verwendenden Programmiersprache. Zunächst war von der Verwendung von Java ausgegangen worden. Für diese Programmiersprache existiert jedoch kein *API* für die Handschrifterkennung und die Anbindung des Tablet-PCs. Da die Programmierung einer eigenen *API* für diese Aufgaben den Rahmen einer Bachelor-Arbeit sprengen würde, musste auf Programmiersprachen zurückgegriffen werden, für die solch ein solches *API* bereits existiert ist.

Für das *.NET-Framework* existiert eine solches *API*. Somit wäre eine Programmierung in den Sprachen des *.NET-Frameworks* C#, C++, Visual Basic oder J# möglich.

Java bietet zwar die Möglichkeit über "native methods" (JNI) die C++-Klassen des *.NET-Frameworks* zu verwenden. Dabei entsteht aber bei jeder Interaktion des Benutzers zusätzlicher Overhead. Durch die Verwendung des *.NET-Frameworks* fällt jedoch der Vorteil der Plattformunabhängigkeit von Java weg. Daher wurde Java nicht verwendet.

C# ist die modernste Programmiersprache des *.NET-Frameworks*. Sie ist objektorientierter als C++ und verzichtet auf unsichere Elemente, wie zum Beispiel die Zeigerarithmetik von C++. Die Syntax ist zu Java recht ähnlich, was den Vorkenntnissen entgegen kam. Zudem ist C# laut Microsoft die neben Java am weitesten verbreitete Programmiersprache in der Softwareentwicklung. Kenntnisse in dieser Sprache werden in Zukunft sicherlich nützlicher sein als beispielsweise in Visual Basic. Aus diesen Gründen wurde C# gewählt.

4.1.1 Unterschiede von C# im Vergleich zu Java

C# bietet einige Features die Java nicht zur Verfügung stellt. Diese werden im Folgenden kurz vorgestellt.

4.1.1.1 Enums

Enums sind hilfreich, wenn einer Variable nur Werte eines bestimmten Wertebereichs zugeordnet werden soll. Dabei kann man dem Compiler die Wertevergabe überlassen oder sie selbst bestimmen.

```
public enum AmpelFarbe { Rot, Orange, Gruen }

// Festlegung der Werte
public enum AmpelFarbe (Rot=10, Orange=20, Gruen=30)

public class Ampel
{
    private AmpelFarbe _farbe = AmpelFarbe.Rot;
}
```

4.1.1.2 Structs

Structs sind so genannte Leichtgewichtsklassen. Sie gleichen den Klassen, nur können sie nicht abstrakt sein und bieten keine Möglichkeit der Vererbung. Structs werden im Gegensatz zu Klassen per Value übergeben. Dies und die fehlende Möglichkeit der Vererbung boten keinen sinnvollen Einsatz in dieser Bachelorarbeit.

4.1.1.3 Properties

Properties sind "syntaktischer Zucker", ohne zusätzliche semantische Funktionalität. Sie ersetzen die get- und set-Methoden für geschützte Instanzvariablen. Lesen und Schreiben erfolgt dabei wie ein Zugriff auf eine öffentliche Instanzvariable.

```
public class Student
{
    private string _name;

    public string Name
    {
        get { return this._name; }
        set { this._name = value; }
    }
}

// Zugriff
Student student = new Student();
student.Name = "Max";
Console.WriteLine(student.Name);
```

Durch Weglassen des set- bzw. get-Teils lassen sich Variablen nur les- oder schreibbar machen.

Die Verwendung von Properties macht den Code lesbarere und erleichtert das spätere Umbenennen, da nur der Name der Property und nicht sowohl get- als auch set-Methode geändert werden müssen. Zudem wird der logische Zusammenhang der get- und set-Methode nun auch optisch sichtbar.

4.1.1.4 Indexer

Indexer ermöglichen den Zugriff auf eine Klasse wie auf ein Array.

```
public class Institut
{
    private string[] _mitArbeiter = new string[5];

    public string this [int index]
    {
        get { return this._mitArbeiter[index]; }
        set { this._mitArbeiter[index] = value; }
    }
}

// Zugriff
Institut institut = new Institut();
Console.WriteLine(institut[0]);
```

4.1.1.5 Delegates und Events

Delegates sind Zeiger auf Funktionen. Mit ihnen kann zur Laufzeit festgelegt werden welche Methode(n) aufgerufen werden sollen. Die zugewiesenen Methoden müssen über die gleiche Signatur und denselben Rückgabotyp verfügen, wie bei der Deklaration des Delegates angegeben wurde.

Events sind Attribute von Klassen. Sie werden wie durch das Schlüsselwort `event` gefolgt von dem Typ des darzustellenden Delegates deklariert. Andere Klassen können bei diesen Events Methoden anmelden. Wenn das event ausgelöst wird, werden alle angemeldeten Methoden ausgeführt. Dieses Konzept ersetzt das Listener-Konzept aus Java.

```
// Deklaration des Delegetentyps NameChangedEventHandler
public delegate void NameChangedEventHandler(string newValue);

public class Student
{
    private string _name;

    // Deklaration eines Events vom Typ NameChangedEventHandler
    public event NameChangedEventHandler NameChanged;

    public string Name
    {
        get { return this._name; }
        set
        {
            this._name = value;
            // Auslösen des Events NameChanged, alle angemeldeten Methoden werden ausgeführt
            if(this.NameChanged != null)
            {
                this.NameChanged(this._name);
            }
        }
    }
}

public class StudentObserver
{
    public StudentObserver(Student student)
    {
        // Anmelden beim NameChanged Event
        student.NameChanged += new NameChangedEventHandler(this.WriteName);
    }

    private void WriteName(string name)
    {
        Console.WriteLine(name);
    }
}
```

Methoden lassen sich auch wieder von Events abmelden. Dazu wird der `-=` Operator mit dem Namen der abzumeldenden Methode verwendet.

Mit diesem Konzept lässt sich das Observer-Pattern sehr elegant umsetzen. Jede Klasse wird nur über Änderungen informiert die für sie relevant sind.

Sämtliche Observer-Observable Beziehungen in dieser Arbeit wurden mit Events und Delegates umgesetzt.

4.1.1.6 Polymorphismus

Im Gegensatz zu Java ist C# `early-binding`. Methoden die von Kindklassen überschrieben werden dürfen, müssen mit `virtual` deklariert werden. Die

überschreibende Methode der Kindklasse muss hingegen mit `override` deklariert werden.

Alternativ können nicht `virtual` deklarierte Methoden mit dem `new` Schlüsselwort überschrieben werden. Sollte die Kindklasse jedoch zur Elternklasse gecastet werden, wird die Methode der Elternklasse aufgerufen.

```
public class Parent
{
    public void PrintName()
    {
        Console.WriteLine("Parent");
    }
}

public class Child : Parent
{
    public new void NotVirtual()
    {
        Console.WriteLine("Child");
    }
}

class Test
{
    static void Main(string[] args)
    {
        Child c = new Child();
        c.NotVirtual();           // Ausgabe: Child
        ((Parent)c).PrintName(); // Ausgabe: Parent
    }
}
```

4.2 Prototypen

Weder im Fachgebiet für Software Engineering noch bei mir waren Kenntnisse über die Programmierung für Tablet-PCs vorhanden. Dies galt auch für die Programmiersprache C#. Um festzustellen, ob das Programm überhaupt umsetzbar ist, und welchen Aufwand dies erfordern würde, sollten im Voraus mehrere Prototypen gefertigt werden.

Im Folgenden werden die beiden angefertigten Prototypen erläutert.

4.2.1 Zeichnen und Schrifterkennung auf Zeichenfläche

Ein wesentlicher Grund für den Einsatz des Tablet-PCs war die Hoffnung, dass dieser die Geschwindigkeit beim Zeichnen der Oberflächen erhöht. Insofern lag es nahe, einen Prototypen zu fertigen, der sich mit dem Zeichnen per *Digitizer* beschäftigt.

Dies ließ sich erstaunlich einfach mit Hilfe des *InkOverlays*, einer Klasse des "Microsoft Tablet PC Platform SDK", realisieren. Das *InkOverlay* dient als eine Art Layer, die über ein *Control* (Elternklasse aller GUI-Elemente) gelegt wird. Auf einem solchen *Control* kann dann per *Digitizer* gezeichnet werden.

Das *InkOverlay* verfügt über drei verschiedene *EditingModes*. Mit diesen kann sehr einfach zwischen Zeichnen, Selektieren und dem Löschen der *Strokes* gewechselt werden. Über die *DefaultDrawingAttributes* des *InkOverlays* lassen sich Aussehen (Dicke, Farbe, usw.) sehr leicht verändern.

Für die Umwandlung von handschriftlichem Text in Maschinentext werden die betreffenden *Strokes* an ein *RecognizerContext* Objekt übergeben. Dieser liefert dann mögliche Interpretationen zurück. Da die in Maschinentext umgewandelten *Strokes* von der Zeichenfläche entfernt werden sollen, müssen diese von den *Strokes*, die die Oberfläche darstellen, unterschieden werden können. Diese Unterscheidung ist schon bei der Schrifterkennung nötig, da sonst *Strokes*, die eigentlich die Oberfläche darstellen, bei der Schrifterkennung mit berücksichtigt würden.

Bei dem Prototyp wurde diese Unterscheidung durch zwei unterschiedliche Tools, wie man es aus diversen Grafikprogrammen kennt, gewährleistet. Eine mögliche Alternative wäre ein spezieller Bereich für das Schreiben des Textes gewesen. Der erkannte Text müsste dann im Nachhinein auf der Oberflächenskizze platziert werden. Denkbar wäre auch, die *Strokes* die in Text umgewandelt werden sollen, zu markieren und sie so von den restlichen *Strokes* zu unterscheiden.

Für das Speichern, Laden und den Export als Grafik gab es einige sehr einfache Beispiele, daher wurden diese Funktionen nicht in den Prototyp implementiert. Beim Export als Grafik wurde dabei allerdings nicht bedacht, dass zusätzlich zu den *Strokes* auch der Text und die Visualisierung der Verknüpfungen mit den Beschreibungen und Erweiterungen exportiert werden mussten. Dies führte in der Endphase der Entwicklung nochmals zu einigen unerwarteten Problemen.

4.2.2 Ausfüllen des Use Cases

Der zweite Prototyp beschäftigte sich mit dem Ausfüllen des Use Case Formulars. Eine nahe liegende Möglichkeit war die Verwendung von *InkEdits*. Dabei handelt es sich um Eingabefelder, in die per *Digitizer* geschrieben werden kann.

Das Schreiben in die *InkEdits* stellte sich im Nachhinein als etwas umständlicher als das Schreiben auf dem *InkOverlay* heraus. Dies konnte leider während der Entwicklung der Prototypen nicht festgestellt werden, da zu diesem Zeitpunkt noch kein Tablet-PC zur Verfügung stand. Da der Text im Nachhinein noch editierbar sein sollte, boten sich allerdings nicht viele andere Möglichkeiten an.

In diesem Prototyp wurde außerdem getestet ob es möglich ist, die Höhe der Eingabefelder automatisch an ihren Inhalt anzupassen. Dies ließ sich sogar mit einer Begrenzung der Höhe verwirklichen. So werden ab einer bestimmten Höhe vertikale Scrollleisten angezeigt, statt das Eingabefeld weiter zu vergrößern.

Die Anpassung der Höhe soll verhindern, dass man beim Lesen des Use Cases in den einzelnen Eingabefelder auch dann scrollen muss wenn der Text nur über einige Zeilen verläuft.

Ab einer festgelegten Größe der Eingabefelder verhindern die Scrollleisten in den Eingabefeldern, dass der Use Case zu lang und zu unübersichtlich wird.

Des Weiteren wurde das Formatieren des eingegebenen Textes getestet. Da das *InkEdit* von *RichTextBox* erbt, einer Textbox für die Eingabe von Text im Rich Text Format, war dies auch kein größeres Problem.

5 Entwicklung des Modells

In diesem Kapitel wird der Weg von dem gegebenen Umständen zum Modell erläutert. Zu Beginn wird der Aufbau des darzustellenden Use Case beschreiben. In den beiden folgenden Kapiteln werden Klassen im Modell und deren Zusammenspiel erläutert. Zum Schluss dieses Kapitels wird auf die Grenzen des Modells eingegangen.

5.1 Aufbau des darzustellenden Use Case

Als Vorlage für den Modellierung des Use Case diene folgende Tabelle:

Use Case Nr. <nr>	<Name: kurzer, aktiver Satz>	
Umfeld	Umfeld	
Systemgrenzen	<Scope: was gehört noch dazu>	
Ebene	<Überblick, Aufgabe oder Teilfunktion>	
Hauptakteur	<Kurzbeschreibung dessen, der Ziel hat>	
Stakeholder u. Interessen	Stakeholder <erste> <zweiter>	Interesse <ihre Interessen> <seine Interessen>
Voraussetz.	<wovon kann man bei Beginn ausgehen?>	
Garantien	<was in <u>jedem</u> Fall gewährleistet ist>	
Auslöser	<wann der Use Case startet>	
Beschreibung	Schritt 1 2	Aktion <von Auslöser bis Aufräumen> <z.B.: "..zurück in Ausgangsz.">
Erweiterung	1a	WENN ... DANN <and. Use Case>
Technologie	<Variationen durch Technik>	

[1]

Diese Modellierungen wurde zur Anpassung an die von Thao Nguyen verwendeten Use Cases noch modifiziert. Da das zu entwickelnde Programm *MockUp Fast* hauptsächlich in Thao Nguyen Aufgabenbereich eingesetzt werden sollte, sollte der Unterschied zu den bisher verwendeten Use Cases nicht zu groß sein.

So kamen zusätzliche Felder für Quelle und Erstellungsdatum hinzu. In den von Nguyen verwendeten Use Cases gab es zu jedem Feld noch die Möglichkeit eine Bemerkung festzuhalten. Um die Übersichtlichkeit nicht zu sehr einzuschränken, wurde nur ein zusätzliches Feld für den gesamten Use Case eingeführt, in dem diese Bemerkungen gespeichert werden können.

Nguyen unterteilte das Feld "Beschreibung" in einen handelnden Akteur und in die von ihm ausgeführte Handlung. Erweiterungen wurden in eine Wenn-Bedingung und in die daraus folgende Aktion unterteilt. Beschreibungen und Erweiterungen konnten zu dem auf einen anderen Use Case verweisen. Diese drei Konzepte wurden mit in das Modell aufgenommen.

Nguyens Vorlage verfügt über weitere Felder, die nicht in das Modell übernommen wurden. Zum einem stehen im Kopf zusätzlich zu Quelle und Datum noch das Projekt, der Ort und der Status des Use Cases. Ort und Status wurden als nicht relevant genug betrachtet. Um eine Überladung des Use Cases und somit zu eine geringere Übersichtlichkeit zu verhindern wurden diese Felder nicht aufgenommen. Da zusammengehörige Use Cases in einem Projekt gespeichert werden, war das Projektfeld im Use Case überflüssig.

Die Erläuterung beschreibt den Use Case textuell, bietet somit keine zusätzlichen Informationen. Der Aufwand für eine Zusammenfassung des Use Cases wäre im Hinblick auf den Nutzen zu groß, vor allem unter dem Aspekt dass das Werkzeug ja genutzt werden soll um, während des Gesprächs den Use Case vollständig zu erfassen. Somit wurde die Erläuterung nicht übernommen.

Das Feld "Nutzen des Hauptakteurs" wird schon durch den Namen des Use Cases beschrieben. Folglich bestand auch keine größere Notwendigkeit dieses Feld zu übernehmen.

Das Feld "Offene Fragen" wurde durch den eigentlichen Zweck des Werkzeuges überflüssig. Ziel ist es ja, offene Fragen während des Kundengesprächs zu erkennen und zu klären.

Durch die Analyse einiger von Nguyen gefertigten Use Cases, wurde festgestellt, dass deren Beschreibungen teilweise auf andere Use Cases verwiesen. Innerhalb dieser wurde die Beschreibung näher erläutert. Dies sollte auch im Programm *MockUp Fast* möglich sein. Zudem sollten bei der Simulation eines Use Cases auf eventuelle Use Case Verweise in den Beschreibungen eingegangen werden. Statt der zugehörigen Oberflächenskizze sollte in einem solchen Fall der Use Case ausgeführt werden auf den verwiesen wird. Dieses Feature sollte jedoch bei Bedarf deaktiviert werden können.

5.2 Klassen des Modells

Aus dem gegebenen Aufbau der Use Cases wurden die folgende Klassen abgeleitet.

5.2.1 UseCase

Die *UseCase*-Klasse repräsentiert den eigentlichen Use Case und ist damit eine der zentralen Klassen des Modells. Sie speichert alle Felder des Use Cases mit Ausnahme der Erweiterungen. Diese werden in der Klasse *Beschreibung* gespeichert.

Stakeholder, *Beschreibung* und Oberflächenskizzen (*MockUps*) werden in einer Liste gespeichert. Der Index der *Beschreibung* legt die Beschreibungsnummer fest. Um die Eindeutigkeit der Namen der *MockUps* zu gewährleisten, werden diese in einem Dictionary (spezielle, assoziative Liste) gespeichert. Der Name der Oberflächenskizze dient als Schlüssel.

Die restlichen Felder des Use Cases lassen sich als String speichern. Für das Erstellungsdatum wurde natürlich ein Datumstyp verwendet. Auch die Nummer des Use Case wird als String gespeichert. So sind hierarchische Nummern möglich.

5.2.2 UseCaseProjekt

Während eines Kundengesprächs werden mehrere Use Cases bearbeitet. Deshalb war eine Klasse notwendig, die alle zusammengehörigen Use Cases speichert und verwaltet. Diese Aufgabe übernimmt die Klasse *UseCaseProjekt*. Um die Eindeutigkeit der Use Case Nummern zu gewährleisten wurde zur Speicherung der *UseCases* ein Dictionary mit der Use Case Nummer als Schlüssel gewählt.

5.2.3 Action

Verweise auf Oberflächenbereiche oder andere Use Cases sollten sowohl von einer Beschreibung als auch von einer Erweiterung aus möglich sein. Da Beschreibung und Erweiterung auch fachlich ähnlich sind zusammengehören (beide erläutern den eigentlichen Ablauf des Use Cases) bot sich die Implementierung in einer Elternklasse an.

Action ist eine abstrakte Klasse und implementiert die Zuweisung der Verweise auf Oberflächenbereiche und Use Cases, sowie einige andere gemeinsame Funktionen.

5.2.3.1 Beschreibung

Beschreibung ist eine konkrete Unterklasse von *Action*. Zu dem Beschreibungstext und der Nummer der Beschreibung kommen noch Verweise auf Oberflächenbereiche und andere Use Cases. Deshalb wurde für die Darstellung der Beschreibungen eine eigene Klasse benötigt. *Beschreibungen* werden ihrem *UseCase* untergeordnet und in ihm gespeichert.

Die Nummer der Beschreibung wird nicht in der *Beschreibung* selbst gespeichert, sondern über den zugehörigen *UseCase* ermittelt. Wird eine Beschreibung umnummeriert, so muss lediglich die Reihenfolge in der Liste geändert werden. Andernfalls müssten alle betroffenen Beschreibungen geändert werden.

Innerhalb der *Beschreibung* werden die zugehörigen *Erweiterungen* gespeichert. Dies sollte das Löschen von Beschreibungen oder das Ändern ihrer Nummern vereinfachen. In beiden Fällen wirkt sich dies auch auf die Erweiterung aus. Da die Erweiterung eine Behandlung der Ausnahmefälle einer Beschreibung erläutert, gehört die *Erweiterung* auch thematisch unter die *Beschreibung*.

5.2.3.2 Erweiterung

Erweiterung ist eine konkrete Unterklasse von *Action*. Auch bei den Erweiterungen war eine eigene Klasse notwendig. Die Vergabe der

Erweiterungsnummer wird wie die der Beschreibungsnummer gehandhabt. In diesem Fall ist jedoch die *Beschreibung* statt des *UseCases* zuständig.

5.2.4 Stakeholder

Bei einem Stakeholder soll seine Rollenbezeichnung und das Interesse an der beschriebenen Funktion gespeichert werden. Somit bot sich ebenfalls eine eigene Klasse an.

Da innerhalb eines Projektes immer wieder die gleichen Stakeholder auftauchen, wäre es denkbar gewesen, die einzelnen *Stakeholder* innerhalb des *UseCaseProjekts* zu speichern. Somit hätte man eine Übersicht über alle betroffenen Stakeholder erhalten. Da sich die Interessen der Stakeholder je nach Use Case ändern, hätte aber eine Trennung der Stakeholder von ihren Interessen stattfinden müssen. Aus diesem Grund werden die *Stakeholder* im *UseCase* gespeichert. Für eine solche Übersicht müsste nur über alle *UseCase*-Objekte iteriert werden.

5.2.5 MockUp

Die *MockUp*-Klasse repräsentiert die Oberflächenskizze und ist neben der *UseCase*-Klasse die zweite zentrale Klasse in dem Modell. Sie enthält den Namen der Oberflächenskizze, das Datum der letzten Änderung, alle zugehörigen *MockUpBeschriftungen* und *MockUpBereiche* sowie die *Strokes*, die die Oberflächenskizze darstellen.

Bei den *Strokes* wurde das Observer-Pattern außer Kraft gesetzt. Das *InkOverlay* schreibt bei Änderung lediglich die Daten in das *MockUp*. Das *MockUp* selbst informiert nicht über stattgefundenen Änderungen. Dies geschieht aus zweierlei Gründen. Zum einen lassen sich diese Daten (1) nicht ohne weiteres in ein bestehendes *InkOverlay* einfügen. Der wesentlich Punkt war jedoch (2) die fehlende Möglichkeit des *MockUps* diese Daten zu interpretieren und somit Änderungen vorzunehmen. Die Daten konnten also nur blind übernommen werden. Dieser Umstand diente als Rechtfertigung, das Observer-Pattern auf dieses Attribut des *MockUps* nicht anzuwenden. Im Nachhinein stellte sich diese Überlegung als nicht schlüssig heraus. So müssten eventuelle andere View-Komponenten doch von der Änderung benachrichtigt werden.

5.2.6 MockUpBereich

Der *MockUpBereich* ist der Bereich eines *MockUps* die mit einer *Action* verknüpft ist. Die Nummer des Bereichs wird über die *Action* bestimmt. So tragen Änderung an der Nummer der *Action* zu Änderungen im *MockUpBereich* bei. Änderungen am *MockUpBereich* sind generell nicht möglich. Denkbar wären nur Änderungen an (a) Größe und Position des Bereiches, (b) am *MockUp* auf dem der Bereich liegt, oder (c) an der *Action* die auf diesen Bereich verweist. Alle diese Änderungen wären jedoch umständlicher durchzuführen, als den bestehenden *MockUpBereich* zu löschen und anschließend einen neuen anzulegen.

5.2.7 MockUpBeschriftung

Eine *MockUpBeschriftung* repräsentiert Maschinentext auf dem *MockUp*. Zusätzlich zum eigentlichen Text, wird die Position auf dem *MockUp* gespeichert.

5.3 Zusammenspiel des Modells

Alle Klassen des Modells tragen eine Referenz auf das Objekt, das ihnen übergeordnet ist (*UseCase* speichert das übergeordnete *UseCaseProjekt*, die *Erweiterung* die übergeordnete *Beschreibung* usw.). Ausnahmen hiervon sind das *UseCaseProjekt*, das keine übergeordnete Klasse hat und der *MockUpBereich*, der zwei Referenzen trägt. Nämlich auf *MockUp* und *Action* untergeordnet.

Um die Modell-Objekte richtig zu verknüpfen, werden die übergeordneten Objekt dem Konstruktor des zu erstellenden Objektes übergeben. Dieses trägt sich dann bei dem übergeordneten Objekt ein. Dadurch wird die Konsistenz des Modells bezüglich der zyklischen Abhängigkeiten gewährleistet. Die Methoden zum Hinzufügen von untergeordneten Objekten dürfen dann nur im *Namespace* sichtbar sein, um eine Fehlverwendung von außen auszuschließen.

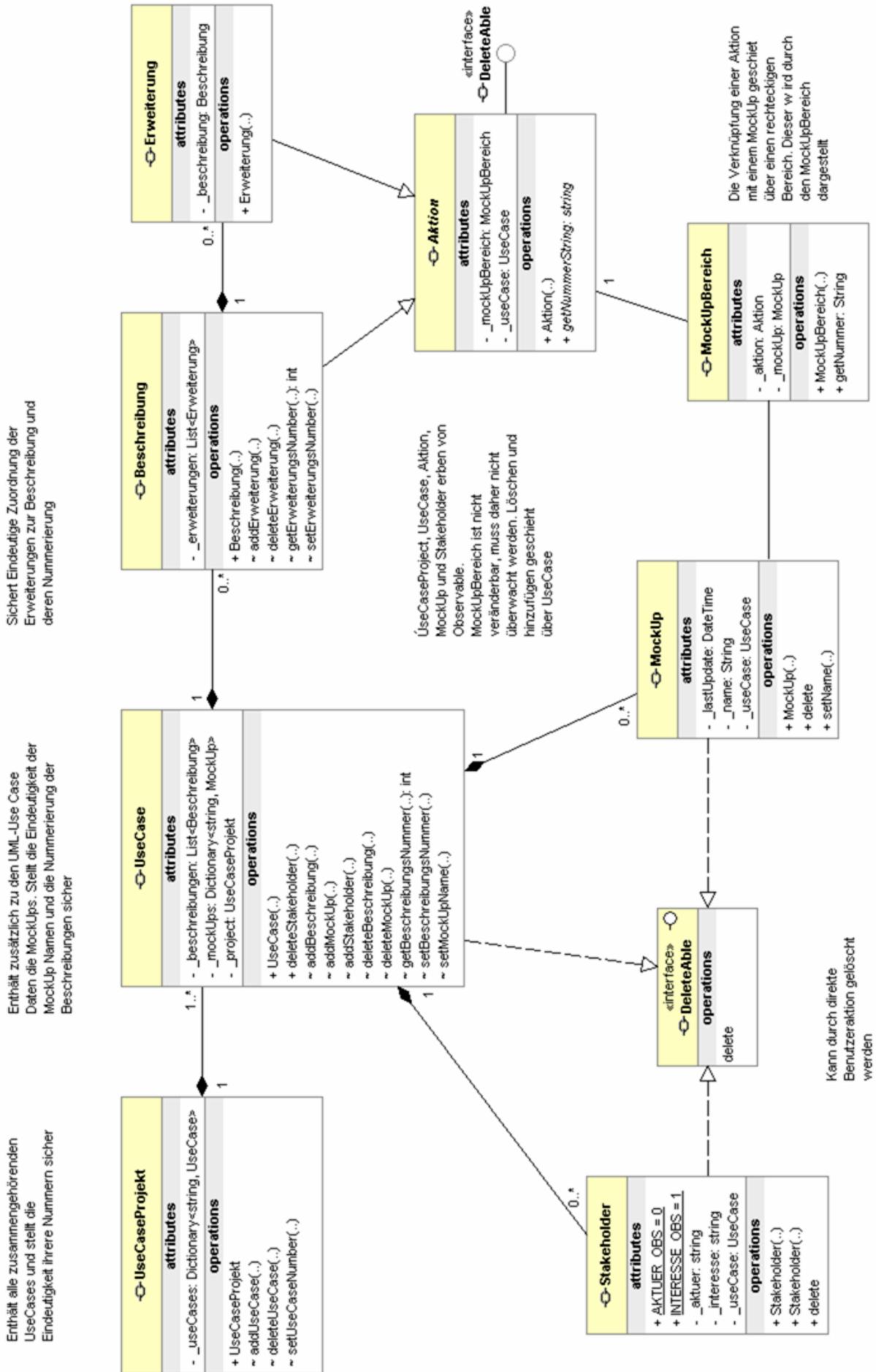
Beim Löschen von Modell-Objekten muss ebenfalls auf die zyklischen Abhängigkeiten geachtet werden. So führt beispielsweise jedes *UseCase*-Objekt eine Liste in der alle *Action*-Objekte eingetragen sind, die auf das *UseCase*-Objekt verweisen. Beim Löschen eines *UseCase*-Objekts werden die Verweise in diesen *Action*-Objekten auf null gesetzt.

5.3.1 Design By Contract

Um sicherzustellen, dass das Modell richtig verwendet wird, wurde mit Design By Contract gearbeitet. Dabei handelt es sich um ein Konzept in der Softwareentwicklung, dass das Zusammenspiel zwischen unterschiedlichen Komponenten regelt. Dies geschieht über so genannte pre- und postconditions. Die preconditions legen die Bedingungen fest, die für den Aufruf einer Methode erfüllt sein müssen. Die postconditions beschreiben die Bedingungen, die nach dem Ausführen der Methode gelten.

Wenn die Bedingungen nicht erfüllt sind, wird eine spezielle Exception geworfen. Diese zeigt an, dass bei der Verwendung der Komponente Fehler gemacht wurden und darf natürlich nicht gefangen werden.

Zum Beispiel wird beim Ändern des Namen des *MockUps* überprüft, ob dieser Name innerhalb des *UseCases* schon vergeben wurde. Ist dies der Fall, wurde das Modell falsch bedient, da der Aufruf der Methode zur Namensänderung einen gültigen Namen voraussetzt. Nach außen hin, sprich zum Entwickler, der diese Komponente verwendet, wird dies durch die spezielle Exception deutlich.



Sichert Eindeutige Zuordnung der Erweiterungen zur Beschreibung und deren Nummerierung

Enthält zusätzlich zu den UML-Use Case Daten die MockUps. Stellt die Eindeutigkeit der MockUp Namen und die Nummerierung der Beschreibungen sicher

Enthält alle zusammengehörenden UseCases und stellt die Eindeutigkeit ihrer Nummern sicher

UseCaseProjekt, UseCase, Aktion, MockUp und Stakeholder erben von Observable.
 MockUpBereich ist nicht veränderbar, muss daher nicht überwacht werden. Löschen und hinzufügen geschieht über UseCase

Die Verknüpfung einer Aktion mit einem MockUp geschieht über einen rechteckigen Bereich. Dieser wird durch den MockUpBereich dargestellt

Kann durch direkte Benutzeraktion gelöscht werden

5.4 Grenzen des Models

Im Folgenden wird auf die Einschränkungen eingegangen, die dieses Modelldesign mit sich bringt. Diese resultieren im Wesentlichen aus der Unterordnung der *Erweiterungen* unter die *Beschreibungen* und der Unterordnung der *MockUps* unter die *UseCases*.

5.4.1 Globale Erweiterungen

Es kann Fälle geben, in denen Erweiterungen nicht genau zu einer Beschreibung zuordnenbar sind. So kann beispielsweise das Schließen des Login-Fensters während der Eingabe des Benutzernamens oder des Benutzerpasswortes geschehen. Beide Fälle erfordern die gleiche Reaktion. Da der bisherige Modellentwurf keine Erweiterung zulässt, die nicht zu genau einer Beschreibung gehört, müsste für jede Beschreibung eine Erweiterung angelegt werden, die diese Abweichung vom Normalfall behandelt.

Eine Möglichkeit, dies zu umgehen, wäre die Speicherung der *Erweiterungen* im *UseCase* statt – wie bisher – in den *Beschreibungen*. Dies hat den Vorteil, dass alle *Erweiterungen* eines *UseCases* in einem Objekt gespeichert sind. Allerdings lässt sich dadurch nicht mehr leicht unterscheiden, welche *Erweiterungen* zu welcher *Beschreibung* gehören und auch welche *Erweiterungen* zu keiner *Beschreibung* gehören.

Das bisherige Konzept zur Nummerierung der *Erweiterungen* müsste geändert werden. Bisher ließ sich die Nummer der *Erweiterung* über ihren Index in der *Beschreibung* bestimmen. Auch beim Löschen einer *Beschreibung* müsste erst bestimmt werden, welche *Erweiterungen* zu ihr gehören, damit auch diese gelöscht werden können. Dies war bisher nicht nötig, da alle *Erweiterungen* innerhalb ihrer *Beschreibung* gespeichert waren.

Besser wäre es also, zwischen *Erweiterungen* die einer *Beschreibung* gehören, und solchen die "global" sind zu unterscheiden. Die globalen Erweiterungen würden direkt im *UseCase* gespeichert werden, während die anderen – wie gehabt – innerhalb ihrer *Beschreibung* gespeichert wären. Dies würde auch Unterscheidung bei der Darstellung in der View erleichtern.

Globale Erweiterungen konnten aufgrund von Zeitmangel nicht mehr implementiert werden.

5.4.2 Zuordnung der MockUps zu einem UseCase

Aus der vorgegebenen Zuordnung der *MockUps* zu genau einem *UseCase* entsteht ein Problem (nicht zu verwechseln mit der in Kapitel 3 Verknüpfung von Use Cases und Oberflächenskizzen erläuterten Problematik bei der Verknüpfung zwischen Aktion und Oberflächenskizze). Es ist nicht möglich, ein *MockUp* für verschiedene *UseCases* zu verwenden. Dies ist kann jedoch gewünscht sein, da die gleiche Oberfläche Funktionen in unterschiedlichen Use Cases erfüllen könnte.

Dies könnte umgangen werden, indem für jeden Use Case die benötigte Oberfläche neu gezeichnet wird. Dies erhöht den Aufwand für den Benutzer jedoch erheblich.

Alternativ könnte man es ermöglichen, dass ein *MockUp* von weiteren *UseCases* verwendet wird. Dies würde jedoch zu Unübersichtlichkeit im Modell führen. So müsste beim Löschen eines *UseCases* überprüft werden, ob andere *UseCases* *MockUps* des zu löschenden *UseCases* verwenden. Wenn dies der Fall ist, müsste immer noch entschieden werden, was mit diesen *MockUps* geschieht. Werden diese einem anderen *UseCase* zugeordnet, und wenn ja welchem? Dies kann man im Zweifelsfall nicht dem Programm überlassen und damit würde das Löschen eines *UseCases* vom Benutzer weitere Aufmerksamkeit verlangen.

Ein weiteres Problem wäre die Eindeutigkeit der *MockUp*-Namen. Soll sie sich nur auf den *UseCase* beziehen, in dem der *MockUp* tatsächlich gespeichert ist? Wenn nicht müsste der Name eventuell geändert werden, sobald ein weiterer *UseCase* darauf verweist.

Sinnvoller wäre es, die *MockUps* direkt im *UseCaseProjekt* zu speichern und keine direkte Verbindung zwischen den *UseCases* und den *MockUps* herzustellen. Über die Verknüpfung zwischen den *MockUps* und den *Beschreibungen* wäre weiterhin feststellbar, welche *MockUps* für welche *UseCases* benötigt werden und umgekehrt. Auch mit dieser Lösung kann dann keine Eindeutigkeit der *MockUp*-Namen innerhalb eines *UseCases* gewährleistet werden, sondern nur noch eine Eindeutigkeit innerhalb des gesamten Projektes. Dies ist eine unzulässige Einschränkung des Namensraums für die *MockUps*.

Die Übersicht im Programm würde darunter leiden, dass sämtliche *MockUps* angezeigt werden müssten, und nicht mehr nur die, die zu dem aktuell betrachteten *UseCase* gehören. Dies könnte abgemildert werden, indem man eine Option zum Ausblenden der *MockUps* anbietet, die nicht mit dem *UseCase* verknüpft sind.

6 Entwicklung der Oberfläche

In dem folgenden Kapitel werden die Entwicklung der Oberfläche und die dabei aufgetretenen Probleme beschrieben. Dabei geht es in erster Linie um die Bedienbarkeit der Oberfläche.

6.1 Bedienbarkeit

Eine wesentliche Anforderung an das Programm war eine Bedienbarkeit, die es ermöglicht während des Gespräches Use Cases und Oberflächenskizze in das Programm einzugeben.

Eine intuitive Oberfläche kann die Bedienung beschleunigen. Die Intuitivität einer Oberfläche drückt sich im wesentlichen dadurch aus, dass die Bedienelemente der Oberfläche erwartungsgemäß platziert sind und reagieren. Es wird keine Zeit zum Auffinden der gewünschten Funktion verschwendet und Fehlbedienungen müssen nicht zeitaufwändig korrigiert werden.

Die Geschwindigkeitseinbußen beim Bedienen einer weniger intuitiven Oberfläche verringern sich mit der Dauer der Programmnutzung, weil der Benutzer Syntax und Semantik der Funktionen kennt. So ist Eclipse sicherlich nicht sonderlich intuitiv – bei einem Programm mit solchem Funktionsumfang auch nicht verwunderlich. Allerdings können versierte Benutzer eine beachtliche Geschwindigkeit entwickeln. Folglich gibt es auch andere Faktoren für die Geschwindigkeit der Bedienung.

Short-Cuts ermöglichen in der Regel einen schnellen Zugriff auf häufig benötigte Funktionen eines Programms. Sie ersparen den Weg, der mit der Maus zurückgelegt werden muss, um das Oberflächenelement, zu erreichen das die Funktion auslöst. Die Hauptersparnis ist jedoch, dass über die Tastatur Funktionen direkt zugänglich sind, die per Maus nur über mehrere Menüebenen erreichbar sind.

Vorraussetzung für Short-Cuts ist jedoch eine Tastatur, die jedoch im Falle eines Tablet-PC nicht vorhanden ist. Wie also kann man die Funktionalität der Short-Cuts anders umsetzen?

Zunächst geht die Bedienung mit dem *Digitizer* schneller als mit der Maus, da die Hand Bedienelemente direkt auswählt und der Umweg über die Mausbewegung wegfällt.

Bei einer überschaubaren Funktionsanzahl kann man die Funktionen direkt zugänglich machen. So muss man sich nicht erst durch mehrere Menüebenen klicken. Während des Erstellens häufig gebrauchte Funktionen sind direkt von der Oberfläche aufrufbar. Öffnen oder Exportieren werden während des Kundengespräches seltener benutzt. Für diese wurden deshalb Menüs verwendet, um die Oberfläche nicht mit Bedienelementen zu überladen.

Ein Use Case enthält in der Regel mehrere Beschreibungen, Erweiterungen und Stakeholder. Für diese existieren immer die gleichen Funktionen (Löschen, Verweis auf Use Case/Oberflächenskizze hinzufügen, Position ändern, Erweiterung zur Beschreibung hinzufügen). Eine Beschreibung verfügt über fast die gleichen Funktionen wie eine Erweiterung.

Daher wäre es nahe liegend, für jede dieser Funktionen einen Button zur Verfügung zu stellen. Die ausgelöste Funktion würde sich dann auf die aktuell ausgewählte Beschreibung oder Erweiterung oder auf den aktuell ausgewählten Stakeholder beziehen. Dazu müsste jedoch erst das entsprechende Element ausgewählt werden, bevor ein weiterer Klick die Funktion auslöst.

Hinzu kommt, dass diese Buttons zentral platziert werden müssten, beispielsweise in der Toolbar für die Use Cases. Dadurch sind sie räumlich von den Elementen getrennt, die sie bearbeiten. Der Benutzer müsste dann große, zeitaufwändige Handbewegungen ausführen, um zur Toolbar zu gelangen.

Aus diesen Gründen wurden für jeden Stakeholder und für jede Beschreibung und Erweiterung eigene Buttons zur Verfügung gestellt. Diese wurden direkt neben dem jeweiligen Element platziert. Soll ein Element geändert werden, muss dieses nicht mehr vorher ausgewählt und der entsprechenden wurde ein Klick gespart.

Die Dicke des Zeichen- und Radierwerkzeuges wurde ursprünglich durch *Splitbuttons* ausgewählt. Dieser *Splitbutton* öffnet ein Untermenü, aus dem dann die gewünschte Dicke ausgewählt werden kann. Dies erwies sich als zu umständlich. Zum einem konnte die Dicke nicht direkt ausgewählt werden, zum anderen schloss sich das Menü, sobald der Stift nicht mehr in Nähe des Monitors war. So das man unter Umständen mehrere Anläufe brauchte um die gewünschte Dicke einzustellen.

Stattdessen steht jetzt für jede Dicke ein zusätzlicher Buttons bereit. Diese werden angezeigt, sobald Radierer oder Zeichenstift ausgewählt wurden.

Auf Dialoge, die das Löschen bestätigen wurde weitestgehend verzichtet. Diese Funktion kann nicht per Tastatur ausgelöst werden und es ist unwahrscheinlich, dass der Benutzer mit dem Stift ungewollt den Löschbutton betätigt. Die Gefahr des versehentlichen Löschens ist also gering. Außerdem behindern die Bestätigungsdialoge den Bedienungsablauf. Sie bringen also mehr Nachteile als Vorteile.

Beim Löschen eines Use Cases wird eine Ausnahme gemacht. In diesem Fall wäre der Verlust zu hoch, da mit dem Use Case auch sämtliche Oberflächenskizzen gelöscht werden. Die wünschenswerte Undo-Funktionalität wurde aus Zeitgründen nicht implementiert. Sie hätte dieses Problem beseitigt.

Eine weitere Möglichkeit die Bedienungsgeschwindigkeit zu erhöhen, ist es, häufig wiederkehrende Abläufe zu automatisieren oder zumindest so zu unterstützen, dass sie schneller von der Hand gehen.

Beim Zeichnen der Oberflächenskizzen boten sich dafür jedoch wenige Möglichkeiten. Da gewünscht war, dass die Oberflächenentwürfe einen Skizzencharakter haben, verboten sich vorgefertigte Elemente (siehe Kapitel 2.2). Die meisten Zeichentools der gängigen Graphikprogrammen widersprachen ebenfalls dem Skizzencharakter. Da der Großteil der

Oberflächenelemente (vom Anwendungsfenster über Eingabefelder bis hin zu den Buttons) rechteckig sind, könnte ein Tool, das beim Zeichnen von Rechtecken hilft, der Geschwindigkeit förderlich sein. Eine Implementierung ähnlich, dem Aufziehen des *MockUpBereichs* wäre denkbar. Ein solches Tool war als zusätzliches Feature gedacht, konnte jedoch mangels Zeit nicht implementiert werden.

Bei den Use Cases boten sich mehrere mögliche Ansätze, von denen jedoch keiner in das Programm implementiert werden konnte. So wäre es zum einem möglich, die Umwandlung des handgeschriebenen Textes durch eine so genannte *WordList* zu unterstützen. Wenn der erkannte Text Ähnlichkeiten mit einem Wort aus dieser Liste hat, würde dieses verwendet werden. Dadurch würde die Erkennung sicherer werden und es wäre weniger Korrekturen nötig. Einzelne Rollen können in dem Projekt sowohl als Stakeholder, wie auch als Akteur in den Beschreibungen auftauchen. Hier wäre ein Einsatzgebiet für die *WordList* gewesen.

6.1.1 Übersichtlichkeit und Aufbau

Beim Aufbau der Oberfläche wurde auf in erster Linie auf die Übersichtlichkeit Wert gelegt. Dadurch sollte die Zeit verringert werden, in der nach bestimmten Elementen (Funktionen, Use Case, Oberflächenskizze) gesucht wird. Ein weiterer Aspekt war die Kompaktheit der Oberfläche um unnötiges Scrollen vermeiden.

Für die Darstellung der Use Cases und Oberflächenskizzen wurde eine Anzeige als Tabs gewählt. Im Gegensatz zur Darstellung in Fenstern erleichtern Tabs das Auffinden des gewünschten Elements, da nicht erst über ein Menü für die unterschiedlichen Fenster oder über die Taskleiste gesucht werden muss. Zusätzlich ließ sich die Verknüpfung von Beschreibungen mit anderen Use Cases leichter visualisieren. So kann der Tab des Use Cases, mit dem die aktuell ausgewählte Beschreibung verknüpft ist, durch ein spezielles Symbol hervorgehoben werden.

Mittels eines *SplitPanels* lässt sich die Größe des Anzeigebereichs für Use Case und Oberflächenskizze variieren. Dadurch kann das gerade bearbeitete Element maximiert werden. Dass lästige Scrollen entfällt.

Die Eingabefelder für den Use Case passen sich dem zur Verfügung stehenden Platz an und es muss nicht horizontal gescrollt werden. Beides führt zu einer besseren Übersicht auf der Oberfläche.

Tablet-PCs bieten die Möglichkeit, vom Querformat der Anzeige ins Hochformat zu wechseln. Aufgrund des vorherrschenden Querformat bei Displays bietet sich das Hochformat zum Zeichnen von Oberflächenskizzen nur bedingt an. Für das Ausfüllen des Use Cases eignet sich das Hochformat aufgrund der Länge des Use Cases jedoch besser. Beim Wechsel ins Hochformat wird daher der Bereich zum Zeichnen der Oberfläche ausgeblendet, so dass ausschließlich der Use Case angezeigt wird.

Das Display eines Tablet-PCs ist nicht ganz so hoch wie ein normales Display. Um nicht zusätzliche Höhe durch Menü- und Toolbars zu verschenden, wurde die Toolbar für die Use Cases in die Menübar integriert. Dort wäre zwar noch Platz für die Toolbar der Oberflächenskizzen gewesen. Um aber deren unterschiedlichen Anwendungsbereich deutlicher zu machen, wurde diese neben der Anzeige für die Oberflächenskizzen platziert.

6.2 Intelligente Oberfläche

Um zu verhindern, dass fehlerhafte Benutzereingaben zu Exceptions im Modell führen, sollte eine intelligente Oberfläche implementiert werden. Die sorgt dafür, dass ungültige Eingaben nicht an das Modell weitergereicht werden. Dies ist wesentlich bei der Nummerierung der *UseCases*, *Beschreibungen* und *Erweiterungen* sowie der Benennung der *MockUps*.

Das Modell stellt zur Validierung der Nummer oder des Namens eine Methode bereit. Erst wenn festgestellt wurde, dass Name und Nummer gültig sind, wird die Eingabe von der View akzeptiert und an das Modell weitergeleitet.

6.3 Umsetzung auf Tablet-PC

Die Umsetzung für den Tablet-PC gestaltete sich erstaunlich einfach. Der *Digitizer* verhält sich fast wie eine Maus. Wird beispielsweise der *Digitizer* kurz über dem Display gehalten, erscheint ein Mauszeiger der sich mit dem *Digitizer* bewegt. So lassen sich auch auf dem Tablet-PC Hovereffekte verwirklichen, wie beispielsweise das Anzeigen von Tooltips.

Durch die fehlende Tastatur erforderte einiges an Umdenken bezüglich des schnellen Zugreifens auf Funktionen. Der nahe liegende Einsatz von Tastenkombinationen fiel daher weg.

6.4 MVC

Die Umsetzung des MVC-Patterns bereitete einige Schwierigkeiten, speziell bei der Trennung zwischen View- und Controller-Klassen. Dies lag größtenteils an der Vorstellung, dass die Controller-Klasse von der View-Klasse getrennt ist und gegebenenfalls ausgetauscht werden kann. View und Controller sind jedoch so stark von einander abhängig, dass dies nicht ohne großen Aufwand möglich wäre. Zudem müssten für den Controller die meisten Attribute der View zugänglich gemacht werden. Dies würde dem Kapselungsprinzip beim objekt-orientierten programmieren widersprechen. Durch private Controller-Klassen innerhalb der jeweiligen View-Klasse würde sich dieses natürlich umgehen lassen.

C# bietet die Möglichkeit, mit `#region <Bezeichner>` und `#endregion` bestimmte Codebereiche zu strukturieren. Mit entsprechenden IDEs lassen sich diese Bereiche auf Wunsch aus- oder einblenden. Außer der Strukturierung des Codes waren keine weiteren Vorteile bei der Verwendung von inneren privaten Klassen sichtbar. Deshalb wurden die Controller-Methoden durch `#region`

strukturiert. So konnte direkt auf die Attribute der View zugegriffen werden; der Code wurde lesbarer (statt `this._view._privateField`, konnte direkt per `this._privateField` zugegriffen werden). Zusätzlich blieb die Anzahl der Klassen überschaubar.

6.5 BeschreibungsTable und ErweiterungsTable

In diesem Kapitel wird auf eine besonders gelungene Umsetzung innerhalb der View-Komponenten eingegangen. Dabei geht es um die Darstellung der Beschreibungen und Erweiterungen. Die Implementierung wird an der BeschreibungsTable-Klasse erläutert. Die Implementierung der ErweiterungsTable-Klasse ist weitestgehend identisch.

Wie bereits in Kapitel 6.1 Bedienbarkeit erläutert, sollten für jede Beschreibung eigene *Controls* zum Manipulieren zur Verfügung gestellt werden. Beschreibungen können gelöscht werden, ihre Position ändern, Verweise auf Use Cases oder Oberflächenskizzen erhalten und Erweiterungen hinzugefügt werden. Diese Funktionen sollten durch einen Klick auf einen Button ausgelöst werden. Außerdem lassen sich selbstverständlich der Akteur und die Aktion einer Beschreibung ändern. Für diese beiden sollten *InkInputs* verwendet werden. Für die Darstellung aller Beschreibungen bot sich eine Tabelle an. Somit erbt *BeschreibungsTable* von der *TableLayoutPanel*-Klasse des *.NET-Frameworks*. Mit dieser lassen sich *Controls* in einer Tabelle anzeigen.

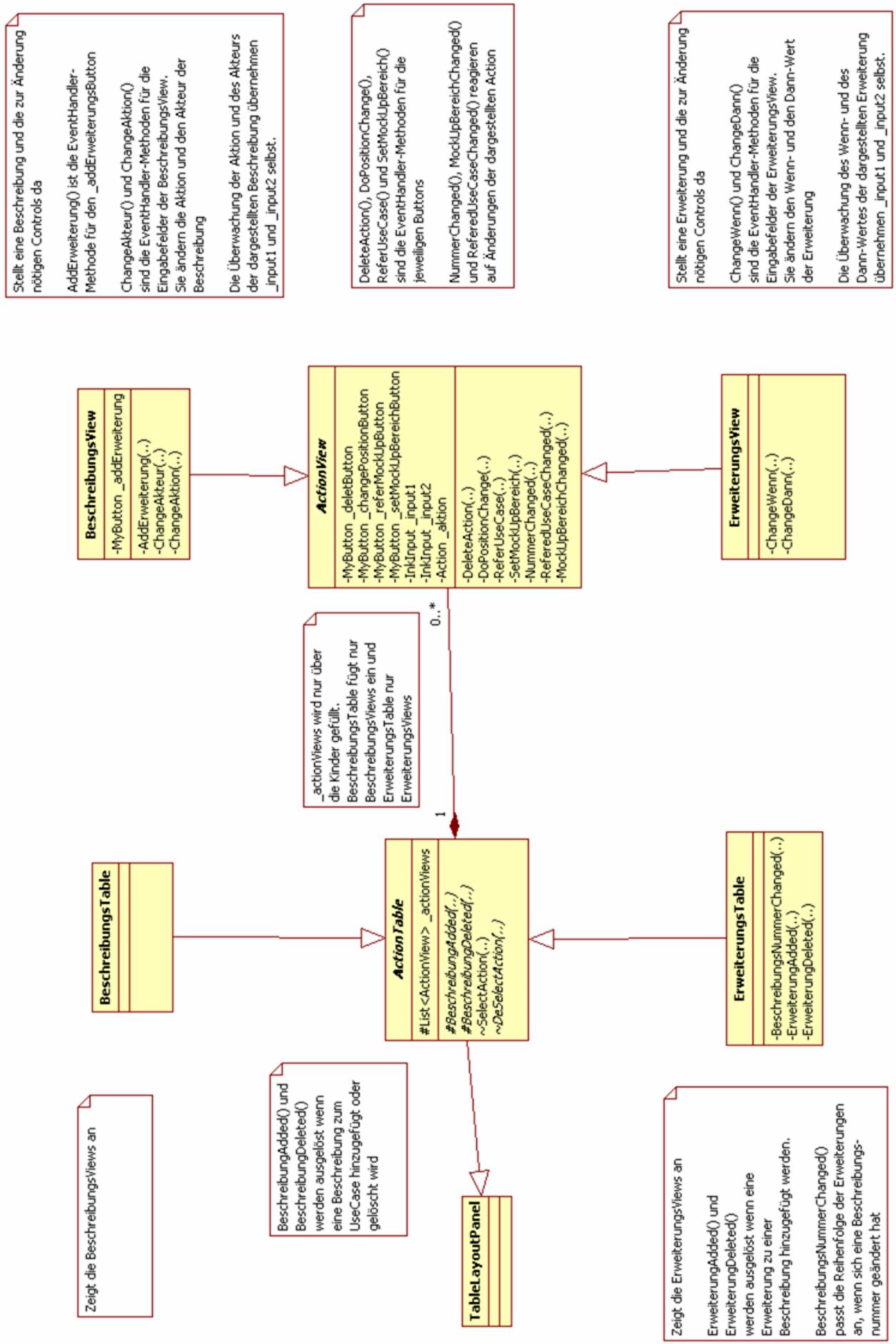
In einem ersten Versuch wurden alle *Controls*, die zur Manipulation einer Beschreibung notwendig waren, direkt in der *BeschreibungsTable* (im Folgenden *Table* genannt) erzeugt und gespeichert. Die *Table* stellte zudem die Methoden zur Verfügung, die zur Änderung einer *Beschreibung* benötigt wurden. Diese Methoden wurden durch einen Klick auf die jeweiligen Buttons ausgelöst. Dies führte zu dem Problem, dass anhand des auslösenden Buttons festgestellt werden musste, welche Beschreibung geändert werden sollte. Dazu wurde innerhalb der Buttons eine Referenz auf die Beschreibung gesetzt, zu der der Button gehörte. Beim Löschen einer Beschreibung musste nun jedoch die gesamte Tabelle neu aufgebaut werden und alle *Controls* für die Beschreibungen neu erstellt werden. Dies kostete zuviel Zeit. Zudem waren viele der Methoden unnötig kompliziert.

Aus den oben genannten Gründen wurde dann eine Klasse *BeschreibungsView* (im folgenden *View* genannt) eingeführt. Diese erzeugt und speichert die *Controls*, die zur Darstellung und Manipulation einer Beschreibung notwendig sind. Zudem wird eine Referenz auf die durch diese *View* dargestellte Beschreibung gespeichert. Die *Table* erzeugt eine *View*, sobald dem *UseCase* eine *Beschreibung* hinzugefügt wird. Anschließend fügt die *View* ihre *Controls* zu der *Table* hinzu.

Da eine eigene Klasse zur Darstellung einer Beschreibung existierte, konnte der Teil der Funktionen, die nur eine Beschreibung betreffen, innerhalb der *View* deklariert werden. Diese Funktionen konnten nur durch einen Klick auf einen der Buttons der dargestellten Beschreibung ausgelöst werden. Somit

musste innerhalb der Funktionen nicht mehr geprüft werden, welche Beschreibung gemeint war.

Lediglich das Auswählen der Beschreibung und die Änderung der Position musste über die *Table* laufen. Da bereits eine andere Beschreibung ausgewählt sein könnte, musste diese deselektiert werden. Da sich die *Views* gegenseitig nicht kannten, war der Umweg über die *Table* nötig. Das Ändern der Position einer Beschreibung beginnt mit dem Anklicken des entsprechenden Buttons der Beschreibung. Danach ändern sich die Buttons für die Positionierung der restlichen Beschreibung. Statt des Icons wird die Nummer der jeweiligen Beschreibung angezeigt. Durch einen Klick auf einen der Buttons wird die Beschreibung auf diese Position gesetzt, die restlichen Beschreibungen rutschen gegebenenfalls auf. Für die Änderung der Buttons und den durch den weiteren Klick ausgelösten Positionstausch musste ebenfalls über die *Table* gegangen werden.



7 Betrachtung des fertigen Programms

In diesem Kapitel wird betrachtet, in wie weit das fertig gestellte Programm den Anforderungen genügt. In erster Linie wird auf die Funktionen eingegangen, die Erwartungen nicht entsprechen. Außerdem wird auf mögliche Verbesserungen eingegangen

7.1 Bedienung mit Tablet-PC

7.1.1 Zeichnen der Oberflächenskizzen

Das Zeichnen der Oberflächenskizzen konnte wie erwartet umgesetzt werden. Die Zeichenfläche lässt sich einfach benutzen, und das Zeichnen nimmt nicht wesentlich mehr Zeit in Anspruch als das Zeichnen auf Papier.

Beim Zeichnen auf Papier existieren jedoch mehr gestalterische Möglichkeiten. So lässt sich beispielsweise die Farbintensität nicht durch unterschiedlich starken Druck variieren. Beim Schraffieren werden Elemente unlesbar. Dies ließe sich durch ein zusätzliches Zeichentool implementieren.

Trotzdem lassen sich brauchbare Oberflächenskizzen anfertigen.

7.1.2 Ausfüllen der Use Cases

Das Ausfüllen der Use Cases mit dem *Digitizer* funktioniert nicht so gut, wie Anfangs erwartet. Das Problem liegt in erster Linie beim Korrigieren von falschen Eingaben. Mit dem *Digitizer* lässt sich der zu korrigierende Text nur sehr schlecht auswählen. Die einzige Möglichkeit ist es, ein gesamtes Wort oder einen gesamten Absatz durch einen Doppelklick auszuwählen. Sollen mehrere Buchstaben am Stück korrigiert werden, müssen diese einzeln mit einer *Application-Gesture* gelöscht werden. Dies kostet viel Zeit. Da die Eingaben in den Eingabefeldern oft nicht richtig erkannt werden, ist die Eingabe über eine Tastatur schneller.

Eine bessere Auswahlmöglichkeit würde das Korrigieren beschleunigen. Dies wäre eventuell durch eigene *Application-Gestures* möglich. So ist beispielsweise eine Geste denkbar, mit der der zu korrigierenden Bereich durch einkreisen der gewünschten Zeichen selektiert wird. Um sicherzustellen, dass *Application-Gestures* richtig erkannt werden, müssen diese jedoch ausführlich getestet werden. Dafür fehlte zum Schluss die Zeit. Aus diesem Grund konnte dieses Feature nicht implementiert werden.

Die häufige Fehl-Erkennung liegt zum einem an der beschränkten Größe der Eingabefelder. Schrift außerhalb der Eingabefelder wird nicht berücksichtigt . daher muss häufig sehr klein geschrieben werden. Das erschwert die Erkennung. Größere Eingabefelder könnten also das Erkennen leichter machen und somit Korrekturen vermeiden. Dadurch nimmt aber die Use Case Tabelle mehr Platz ein und es muss mehr gescrollt werden. Die Möglichkeit, das

Eingabefeld zu vergrößern in das gerade geschrieben wird, tauchte leider zu spät auf um noch implementiert zu werden.

7.2 Export der Use Cases

Das Programm *MockUp Fast* erlaubt Use Cases lediglich als csv-Datei exportiert werden. Dabei gehen bestimmte Informationen verloren. So lassen sich beispielsweise keine Zeilenumbrüche in Eingabefeldern exportieren. Auch eine (noch nicht implementierte) Hervorhebung bestimmter Wörter könnte nicht übernommen werden. Mit einer Druckfunktion könnten diese Informationen exportiert werden, wenn auch nur auf Papier oder PDF. Auch ein Export in eine Excel-Datei wäre möglich sein, denn unterstützt das *.NET-Framework* diverse Office Anwendungen.

Ursprünglich war ein Export in XML-Format gefordert, so dass die erstellten Use Cases in anderen Programmen weiterverwendet werden können. Auch dieses Feature ist mangels Zeit nicht implementiert worden. Interessant wäre auch die Option, Use Cases zu importieren. Dadurch könnten Use Cases die bereits mit anderen Programmen erstellt worden sind, durch eine Verknüpfung mit Oberflächenskizzen erweitert werden.

8 Bedienungsanleitung

8.1 Voraussetzungen

Voraussetzung für die Verwendung des *MockUp Fast* Programms ist ein Tablet-PC mit "Windows XP Tablet PC Edition" als Betriebssystem. Zudem muss das "Microsoft .NET Framework 2.0 Redistributable" installiert sein.

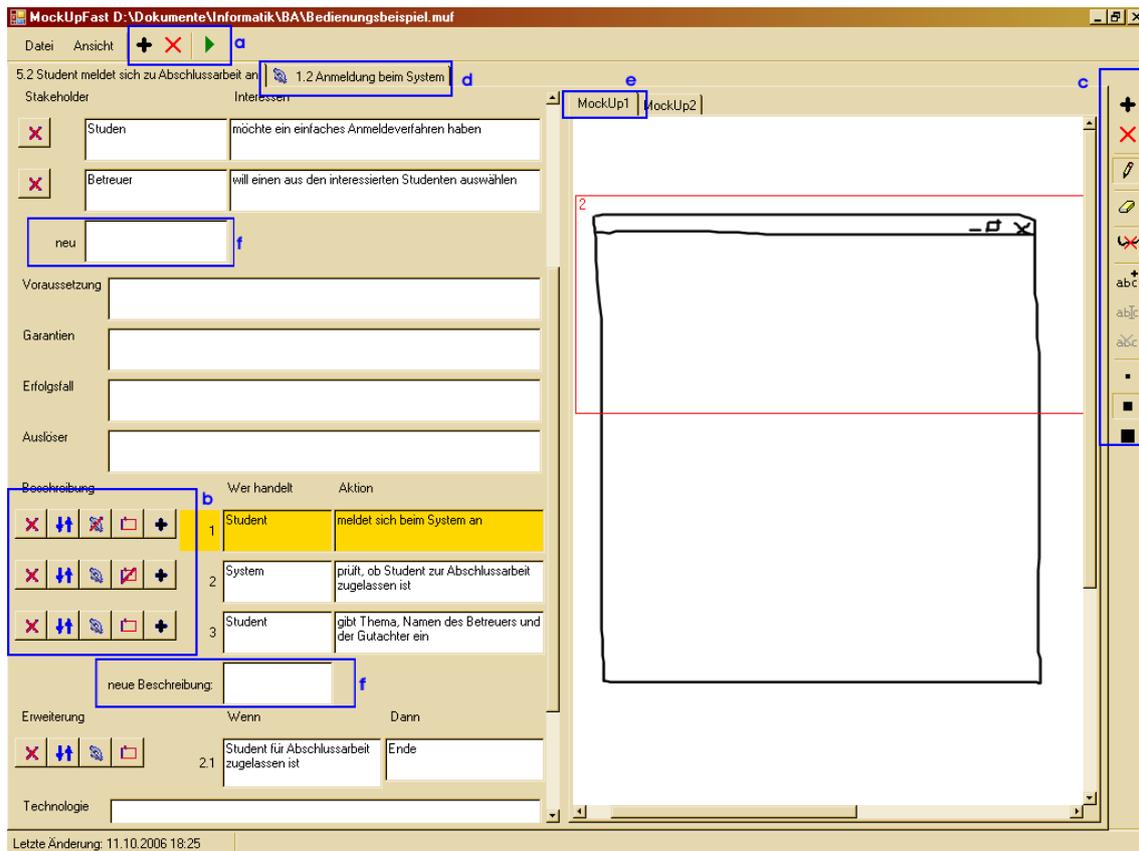
Eine Verwendung ohne Tablet-PC ist ebenfalls möglich. Dazu muss jedoch das "Microsoft .NET Framework 2.0 SDK", das "Microsoft Windows XP Tablet PC Edition 2005 Recognizer Pack" und das "Microsoft Windows XP Tablet PC Edition Development Kit Version 1.7" installiert sein.

8.2 Installation

Für die Installation von *MockUp Fast* muss gegebenenfalls die vorherige Version deinstalliert werden. Dazu muss die Setup-Version des installierten *MockUp Fast* vorliegen.

Durch das Aufrufen der *setup.exe* wird ein Install-Wizard gestartet. Nach Ausführung des Wizards, kann das Programm aus dem Ordner, in das es installiert wurde gestartet werden.

8.3 Benutzung



a: Tools für Use Cases

-  Neuen Use Case anlegen.
-  Ausgewählten Use Case löschen.
-  Ausgewählten Use Case simulieren.

b: Tools für Beschreibung und Erweiterung

-  Beschreibung/Erweiterung löschen
-  Position der Beschreibung/Erweiterung ändern.
Nach dem Drücken des Buttons werden die Pfeilsymbole auf den anderen Beschreibungen/Erweiterungen durch die Positionsnummer ausgetauscht. Dann die entsprechende Position wählen.
-  Einen Verweis auf einen Use Case setzen.
Nach dem Drücken erscheint eine Liste mit allen Use Cases des Projektes, daraus gewünschten auswählen. Liste kann durch einen Klick irgendwo auf der Oberfläche geschlossen werden.
-  Löscht den Verweis auf einen Use Case.
-  MockUp Bereich definieren, auf den sich die Beschreibung / Erweiterung bezieht.
Nach dem Klicken bleibt der Button gedrückt. Auf der Oberflächenskizze kann durch Aufziehen eines Rechteckes der Bereich definiert werden. Abheben des *Digitizers* beendet den Vorgang.



Löschen des MockUp Bereich.



Erzeugt eine Erweiterung zu dieser Beschreibung.

c: Tools zum Zeichnen der Oberfläche



Neue Oberflächenskizze anlegen.



Ausgewählte Oberflächenskizze löschen.



Linien zeichnen.



Radieren.



Komplette Linie löschen (alles was ohne Absetzen des *Digitizers* gezeichnet wurde).



Text schreiben.

Nach Auswahl des Tools kann auf der Zeichenfläche geschrieben werden. Erkannter Text wird in dem eingblendeten Textfeld angezeigt und kann dort korrigiert werden (erst nach Abschluss des Schreibens, da korrigierter Text sonst ersetzt wird).

Nach dem Drücken des OK-Buttons wird der erkannte Text dort auf der Zeichenfläche platziert, wo mit dem Schreiben begonnen wurde.



Ausgewählten Text bearbeiten.

Korrektur muss in dem eingblendeten Textfeld erfolgen. Änderungen werden durch Drücken des OK-Buttons übernommen.



Löscht ausgewählten Text.



Stärkeauswahl fürs Zeichnen und Radieren

d: Use Case Verweis

Wird  in einem Use Case Tab angezeigt, ist die ausgewählte Beschreibung / Erweiterung mit diesem Use Case verknüpft.

e: Namen der Oberflächenskizzen ändern

Durch einen Doppelklick auf den Tab kann der Name der Oberflächenskizze geändert werden. Im folgenden Dialog muss ein gültiger Name (kein Leerstring und kein bereits für diesen Use Case vergebener Name) eingegeben werden.

f: Stakeholder oder Beschreibung anlegen

Durch das Ausfüllen des einzelnen Eingabefeldes wird ein neuer Stakeholder oder eine neue Beschreibung erzeugt.

Ausfüllen der Eingabefelder mit einem *Digitizer*

Es existieren zwei Eingabemöglichkeiten mit dem *Digitizer*. Zum einem kann direkt in das Eingabefeld geschrieben werden. Nach einer kurzen Wartezeit wird die Eingabe in Text umgewandelt.

Der Cursor kann durch einen Klick mit dem *Digitizer* an der entsprechenden Position platziert werden. Durch einen Doppelklick wird das gesamte Wort ausgewählt, durch einen weiteren Doppelklick, der gesamte Absatz. Mit einem Rechtsklick in das Eingabefeld wird der gesamte Inhalt ausgewählt. Ausgewählter Text wird wie gewohnt bei einer Eingabe ersetzt.

Eine schnelle Bewegung von rechts nach links löscht das Zeichen hinter dem Cursor. Eine schnelle Bewegung nach unten gefolgt von einer schnellen Bewegung nach links (ohne Absetzen) fügt einen Zeilenumbruch ein.

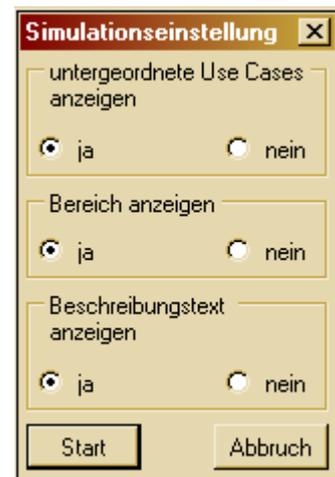
Optional kann das in die Windows XP Tablet Version integrierte Tool zur Eingabe verwendet werden. Dazu muss nach einem Klick in das Eingabefeld auf den erscheinenden Button geklickt werden. In dem daraufhin eingeblendeten Fenster kann dann per *Digitizer* geschrieben werden. Mit diesem Tool lassen sich Fehler leichter korrigieren, als wenn direkt ins Eingabefeld geschrieben wurde.

Simulationseinstellung (▶)

Bei "untergeordneten Use Case anzeigen" kann eingestellt werden, ob bei Beschreibungen, die einen Verweis auf einen Use Case enthalten, dieser mit simuliert werden soll. In diesem Fall wird die Oberflächenskizze zu dieser Beschreibung nicht angezeigt.

Bei "Bereich anzeigen" kann eingestellt werden, ob der Rahmen um den MockUp Bereich der Beschreibungen mit angezeigt werden soll.

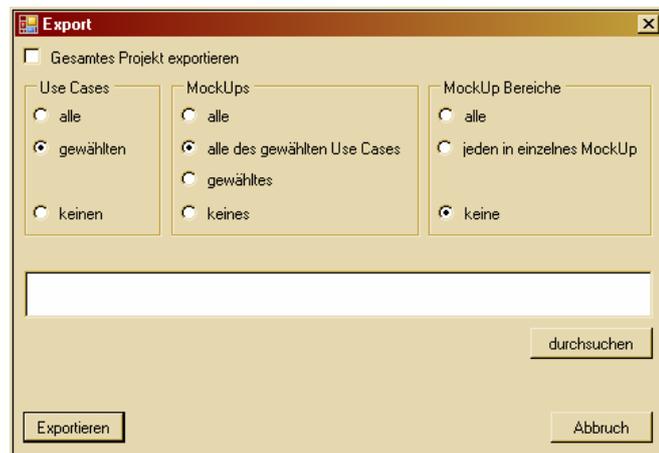
Bei "Beschreibungstext anzeigen" kann eingestellt werden, ob der Akteur und die Aktion der Beschreibungen mit angezeigt werden sollen.



Exporteinstellungen

Ist der Haken bei "Gesamtes Projekt exportieren" gesetzt, werden alle Use Cases und alle MockUps exportiert.

Use Cases werden als csv-Datei exportiert. Sie werden direkt im angegebenen Ordner gespeichert. Der Name der Datei setzt sich aus Nummer und Namen des jeweiligen Use Cases zusammen. Es kann gewählt werden, ob nur der ausgewählte, alle oder gar kein Use Case exportiert werden soll.



MockUps werden als png-Datei exportiert. Sie werden in einem Unterordner des angegebenen Ordners gespeichert. Der Unterordner wird nach dem Use Case benannt zu dem der MockUp gehört. Der Name der png-Datei entspricht dem des MockUps. Bei "allen des gewählten Use Cases" werden nur die MockUps des ausgewählten Use Case exportiert. Bei "gewähltes" wird nur das ausgewählte MockUp exportiert.

Bei "alle" unter MockUp Bereiche werden alle MockUp Bereiche in einer Datei dargestellt.

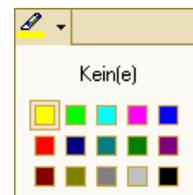
Bei "jeden in einzelnes MockUp" wird für jede Beschreibung und Erweiterung, die ein MockUp Bereich hat, eine eigene Datei erstellt. Diese enthält dann den gesamten MockUp und den MockUp Bereich der jeweiligen Beschreibung. Die Nummer der Beschreibung oder Erweiterung wird an den Dateinamen angehängt.

Bei "keine" wird ausschließlich der MockUp exportiert. Es werden keine MockUp Bereiche dargestellt.

9 Glossar

Im Glossar aufgeführte Begriffe sind im Text *kursiv* hervorgehoben. Bei kursiven Begriffen, die nicht im Glossar auftauchen, handelt es sich um Klassen des Programms *MockUp Fast*. Eine Erläuterung zu diesen Klassen findet sich in Kapitel **5 Entwicklung des Modells** und **6 Entwicklung der Oberfläche**.

.NET-Framework	Das .NET-Framework ist ein sprachübergreifendes <i>API</i> für C++, C#, Visual Basic und J#. Es wurde von Microsoft für die Anwenderplattform .NET entwickelt.
API	Ein Application Programming Interface ist eine Sammlung von Methoden zur Softwareentwicklung.
Application-Gesture	Application-Gestures sind spezielle Bewegungen mit dem <i>Digitizer</i> . Diese lösen bestimmte Funktionen aus. In einem Eingabefeld löscht beispielsweise eine schnelle Bewegung von rechts nach links das Zeichen hinter dem Cursor.
Control	Basisklasse aller Klassen für die graphische Darstellung (ähnlich der Component-Klasse in Java)
Digitizer	elektromagnetischer Stift zur Eingabe auf Tablet PCs
InkEdit	Ein InkEdit ist ein spezielles Eingabefeld aus der "Microsoft Tablet PC Platform SDK" für die Eingabe mittels eines <i>Digitizers</i> .
InkInput	Eingabefeld für Eingabe per Tastatur und <i>Digitizer</i> (Ersatzweise auch Maus), das Text im Rich-Text-Format darstellen kann und die eigene Höhe an den Inhalt anpasst.
MockUp Fast	Name des Programms, das in dieser Bachelorarbeit entwickelt werden sollte.
Namespace	Ein Namespace gruppiert logisch zusammenhängende Klassen in C# (ähnlich dem Package in Java)
Splitbutton	Ein Splitbutton ist ein Button mit einem zusätzlichen Ausklappmenü. In dem Ausklappmenü kann die genaue Funktion festgelegt werden, die beim Drücken des Buttons ausgelöst wird. Splitbuttons werden verwendet um ähnliche Funktionen zu gruppieren und um Platz auf der Oberfläche zu sparen.
Stroke	Repräsentation eines Linienzugs im "Microsoft Tablet PC Platform SDK". Sie wird mittels <i>Digitizers</i> erzeugt. Alle Linienabschnitte die ohne Absetzen des <i>Digitizers</i> gezeichnet wurden, gehören zu einer Stroke.



10 Quellenangaben

[1] Kurt Schneider (2004/2005-3), *Grundlagen der Softwaretechnik, kommentiert*, Seite 116

Andreas Kühnel (2006), *Visual C# 2005 – Das umfassende Handbuch*, Galileo Computing, Bonn, (ISBN 3-89842-586-X)

Martin Fowler, Kendall Scott (1998), *UML – konzentriert*, Addison Wesley, München, (ISBN 3827313295)

Microsoft, *The C# Programming Language for Java Developers* [online], < <http://msdn.microsoft.com/vstudio/java/gettingstarted/csharpforjava/> > (12.10.2006)