Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Lukas Grundmann

Entwicklung und Bewertung eines vielseitig einsetzbaren Diagrammeditors für den Microsoft Pixelsense Tabletop PC

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lukas Grundmann

Entwicklung und Bewertung eines vielseitig einsetzbaren Diagrammeditors für den Microsoft Pixelsense Tabletop PC

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Thomas Thiel-Clemen

Abgegeben am 13.02.2014

Lukas Grundmann

Thema der Arbeit
Entwicklung und Bewertung eines vielseitig einsetzbaren Diagrammeditors für den Microsoft Pixelsense Tabletop PC

Stichworte
Kollaboration, Tabletop PC, graphenbasierte Diagramme, Microsoft Pixelsense, NShape Framework

Kurzzusammenfassung
Diese Arbeit entwickelt und bewertet einen Diagrammeditor für die Microsoft Tabletop PC Plattform. Der Editor ist vielseitig einsetzbar und erlaubt das kollaborative Bearbeiten auf Graphen basierter Diagramme. Zur Entwicklung gehören die Recherche und Dokumentation von Grundlagen, eine Anforderungsanalyse und die Erstellung eines Entwurfs. Dieser besteht aus einer Beschreibung der Benutzerschnittstelle, einer fachlichen Architektur und einem technischen Entwurf. Teile des Entwurfs werden realisiert.

Lukas Grundmann

Title of the paper
Development and evaluation of a multi-purpose diagram editor for the Microsoft Pixelsense tabletop PC

Keywords
Collaboration, tabletop PC, graph based diagrams, Microsoft Pixelsense, NShape framework

Abstract
This thesis develops and evaluates a diagram editor for the Microsoft Pixelsense tabletop PC platform. The multi-purpose editor allows the collaborative modification of graph based diagrams. The development process involves collecting and writing down basic knowledge, gathering of requirements and creating a design. The design consists of an user interface description, a subject-specific architecture and a technical draft. The design was implemented partly.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Tische dienen schon seit Jahrhunderten als Arbeitsplatz für Einzelpersonen und besonders aber auch Gruppen, die zusammen gemeinsame Aufgaben bearbeiten. Dabei bleibt die Arbeitsfläche meist passiv und ist sehr flexibel verwendbar. So können ihre Benutzer frei entscheiden, wie sie benötigte Objekte wie zum Beispiel Werkzeuge oder Materialien auf dem Tisch platzieren.

Das Arbeiten an Tischen bringt jedoch auch Herausforderungen mit sich. So gilt es zum Beispiel Konflikte um Objekte zu lösen und wenn ein gewünschtes Objekt außer Reichweite befindet, muss sich die betroffene Person um den Tisch bewegen. Oder sie fragt einen anderen Tischnutzer um Hilfe, der daraufhin eventuell seine gerade ausgeführte Tätigkeit unterbrechen muss.

Seit den letzten beiden Jahrzehnten [MüFj10] gibt es interaktive Tische (Tabletop PCs), welche als Tischfläche einen Bildschirm, der Fingerberührungen erfassen kann, aufweisen. Diese Tabletop PCs können durch Virtualisierung von Objekten ihre Nutzer aktiv bei der Lösung der zuvor beschriebenen Probleme unterstützen, indem sie zum Beispiel bei Konflikten moderieren oder bei Bedarf die virtuellen Objekte bewegen.

Ein weiteres Werkzeug für gemeinsames Arbeiten stellt das Diagramm dar. Dieses gibt allgemein Informationen und die Zusammenhänge zwischen diesen in visueller Form wieder. Die Bandbreite an Diagrammen reicht von Tortendiagrammen, welche die Sitzverteilung nach einer Wahl wiedergeben (siehe zum Beispiel Abbildung 1.1), bis zu Diagrammen, welche die einzelnen Komponenten einer Anwendung aufzeigen (zum Beispiel Abbildung 1.2). Letztere können zum Beispiel bei der Koordination von Teams aus Informatikern, bei denen bestimmte Teile des Teams für bestimmte Komponenten verantwortlich sind, helfen. Denn wo welche Unterteams zusammenarbeiten müssen und voneinander abhängen, lässt sich aus den Verbindungen zwischen den Komponenten ableiten.

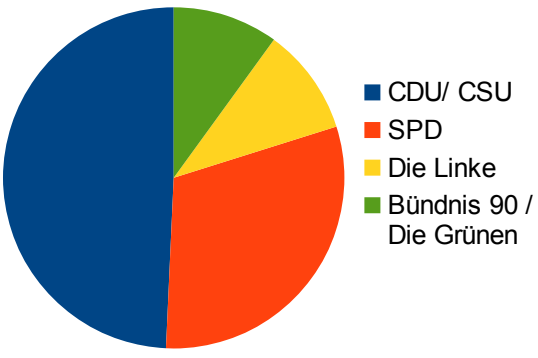

Abbildung 1.1: Sitzverhältnisse nach der Bundestagswahl von 2013 [Bund13]

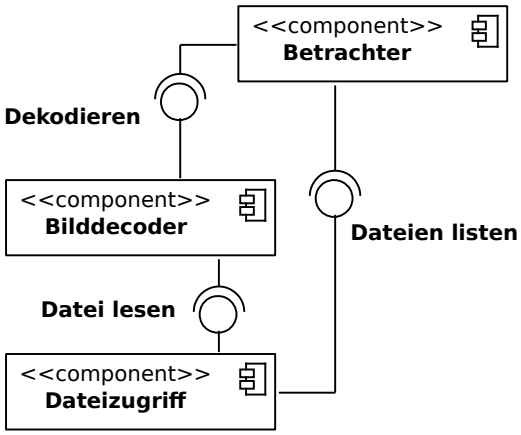

Abbildung 1.2: Beispiel eines Komponentendiagramms

1.2 Zielsetzung

Ziel der Arbeit ist es, die beiden Werkzeuge zum gemeinsamen Arbeiten, Tabletop PCs und Diagramme zusammenzuführen. Dazu soll ein vielseitig einsetzbarer Diagrammeditor für den Microsoft Pixelsense Tabletop PC entwickelt werden, der die Bearbeitung des Inhalts durch mehrere Nutzer gleichzeitig erlaubt. Der Schwerpunkt soll dabei bei den Möglichkeiten des Tabletop PCs zur Unterstützung der Zusammenarbeit mehrerer Personen als auf der Bearbeitung von Diagrammen liegen. Bezüglich Letzterem muss der Editor folgendes unterstützen:

- Bearbeiten von auf Graphen basierten Diagrammen
- Speichern und Laden der Diagramme in proprietären Dateiformat
- Export der Diagramme in ein verbreitetes Dateiformat für Vektorgrafiken

Auf mehrere gleichzeitige Benutzereingaben sollte der Editor robust reagieren und das gemeinsame Bearbeiten von Diagrammen aktiv unterstützen.

1.3 Aufbau der Arbeit

Nach der Einleitung erläutert ein Kapitel zunächst die Grundlagen der Arbeit. Diese stellen die Basis für Entwurf und Implementierung der Anwendung da und erläutern dazu besonders die Erkenntnisse anderer Arbeiten zur Zusammenarbeit mehrerer Personen an einem Tabletop PCs sowie zur Gesten basierten Nutzung berührungsempfindlicher Bildschirme. Außerdem wird kurz der Microsoft Pixelsense sowie das NShape Diagrammframework vorgestellt.

Das folgende Kapitel dokumentiert die Anforderungsanalyse und entsprechend die funktionalen, nicht funktionalen sowie technischen Anforderungen.

Auf die Dokumentation der Anforderungen folgt ein Kapitel, welches den Entwurfsprozess beschreibt. Dieser führt über ein grobes Konzept hin zum detaillierten Entwurf.

Anschließend beschreibt ein Kapitel die Realisierung des Entwurfs und die an der Implementierung durchgeführten Tests.

Das vorletzte Kapitel setzt sich kritisch mit der vorangegangenen Arbeit auseinander und gibt Hinweise darauf, wie die in der Arbeit erworbenen Erkenntnisse auf andere Problemstellungen übertragen werden könnten.

Abschließend fasst das letzte Kapitel die Bachelorarbeit nochmal kurz zusammen und rundet diese ab.

2 Grundlagen

Dieses Kapitel beschreibt technische und methodische Grundlagen, welche aus fremden Arbeiten abgeleitet werden und die eine Basis für die Anforderungsanalyse, den Entwurf sowie dessen Realisierung bieten sollen.

Zunächst wird auf die Ergebnisse von Arbeiten zu den Themen „kollaboratives Arbeiten auf Tabletop PCs“ und „Gesten“ eingegangen. Danach wird der Microsoft Pixelsense Tabletop PC und die zugehörigen SDKs beschrieben. Den Abschluss bildet ein Unterkapitel über das verwendete Diagramm Framework NShape.

2.1 Kollaboratives Arbeiten auf Tabletop PCs

Der Begriff *Kollaboration* steht historisch für eine Zusammenarbeit mit dem Feind in Kriegs- und Besatzungszeiten. [Wiki14] Im Rahmen dieser Arbeit bedeutet Kollaboration dagegen, dass mehrere Personen in räumlich enger Zusammenarbeit, die sich durch Interaktion und Kommunikation zwischen Personen auszeichnet, eine gemeinsame Aufgabe bearbeiten. Dieses Unterkapitel erläutert die Ergebnisse von Forschungsarbeiten über kollaboratives Arbeiten an Tabletop PCs.

2.1.1 Unterteilung der Arbeitsfläche

Bei kollaborativen Arbeiten an Tischen spielt die räumliche Einteilung der Arbeitsfläche eine große Rolle. So bilden sich verschiedene Arten an Arbeitsbereichen [SSCI04]:

- *Persönliche Arbeitsräume* sind einem Tischnutzer zugeordnet. Dieser hält meist die Kontrolle über den Bereich, was bedeutet, dass Ressourcen in seinem Bereich exklusiv von ihm verwendet oder verwaltet werden. Persönliche Arbeitsräume befinden sich meist vor der zugehörigen Person.
- *Gemeinsame Arbeitsräume* befinden sich zwischen den persönlichen Arbeitsräumen und werden oft von vielen Tischnutzern gleichzeitig verwendet. Dies bedeutet, dass auf diesen Flächen Konflikte um Ressourcen auftreten können, die dann zum Beispiel im Gespräch aufgelöst werden müssen.
- *Lagerflächen* enthalten Ressourcen und werden von den Nutzern je nach Bedarf völlig frei auf dem Tisch platziert und bewegt.

Bei passiven Tischen werden die Grenzen zwischen den Bereichen durch erlernte soziale Verhaltensregeln der Benutzer und physikalischen Gegebenheiten bestimmt. So ist die Größe der einzelnen Bereiche von der Anzahl der Tischnutzer und der Tischgröße abhängig. [MRSF04] An Tischen in der Größenordnung eines Microsoft Pixelsense kommt es oft nicht zur Bildung von persönlichen Arbeitsräumen. In diesem Falle kann jeder Zugriff auf eine Ressource zu einem Konflikt führen.

Des Weiteren kann die Konfliktanzahl auf Tabletop PCs dadurch erhöht werden, dass eine Anwendung einem Benutzer erlaubt mehrere, beliebig auf der virtuellen Arbeitsfläche verteilte Ressourcen gleichzeitig oder sogar die gesamte virtuelle Arbeitsfläche (z.B. ihre Ausrichtung) zu verändern. Dadurch kann ein Benutzer versehentlich die Arbeit eines anderen Tischnutzers beeinträchtigen. [MRSF04]

2.1.2 Lösen von Konflikten

Kapitel 2.1.1 erläutert, inwieweit es beim kollaborativen Arbeiten auf Tabletop PCs zu Konflikten kommen kann und dass diese bei ausschließlicher Anwendung von sozialen Verhaltensregeln häufiger als bei konventionellen Tischen auftreten.

Eine kollaborative Tabletop PC Anwendung kann allerdings ihre Benutzer bei der Konfliktvermeidung sowie -auflösung unterstützen, indem sie die Einhaltung von Bearbeitungsrichtlinien ermöglicht und durchsetzt. Diese legen fest, was passiert, wenn mehrere Benutzer gleichzeitig Aktionen ausführen wollen und es dabei zu einem Konflikt kommt. Morris u. a. haben in einem ihrer Artikel ein paar Richtlinien vorgeschlagen. [MRSF04]

Generell können sie zum einen danach unterteilt werden, in welchem Kontext sie angewendet werden:

- beim Zugriff auf ein *Objekt*
- bei Aktionen welche die *ganze Anwendung* betreffen (z.B. Schließen des geöffneten Dokuments)

Des Weiteren können die Richtlinien danach unterteilt werden, wer (aktiv oder automatisch unter Verwendung von Informationen über den Benutzer) darüber entscheidet welche der im Konflikt stehenden Aktionen den Vorzug bekommt:

- *Eigentümer* eines Objekts oder *Initiator* einer globalen Einstellung
- Auf das Objekt eines anderen *Zugreifender* oder von globaler Einstellung *betroffene Nutzer*
- *beide Konfliktparteien*

Tabelle 2.1 fasst die von Morris u.a. vorgeschlagenen Richtlinien zusammen.

	Objekt	Ganze Anwendung
Eigentümer oder Initiator	- *Teilen*: Nutzer können für das Objekt dynamisch zwischen Privat und Öffentlich wählen. - *Explizit*: Eigentümer legt explizit fest, welche Nutzer auf das Objekt zugreifen können. - *Dialog*: Eigentümer kann über einen Dialog bei Bedarf einen Zugriff autorisieren.	- *Privilegierte Objekte*: Globale Aktionen können nur über bestimmte Objekte (zum Beispiel Menüs) ausgelöst werden. - *Immer*: Globale Aktion wird einfach ausgeführt. Somit finden nur die sozialen Verhaltensregeln Anwendung.
Zugreifender oder betroffene Nutzer	- *Öffentlich*: Jeder kann auf das Objekt zugreifen. - *Privat*: Nur der Eigentümer kann auf das Objekt zugreifen. - *Kopie*: Das vom Konflikt betroffene Objekt wird kopiert. - *Persönliche Sichten*: Beim Zugriff auf ein Objekt (zum Beispiel Menü) eines	- *Keine Auswahl, keine Berührungen oder keine berührten Objekte*: Eine globale Aktion wird nur ausgeführt wenn entweder, keine Objekte aktiv angewählt, irgendwo das Display des Tabletop PCs berührt oder ein Objekt berührt ist.

	anderen Nutzer wird dessen Inhalt an den zugreifenden Nutzer angepasst. - *Patt*: Bei einem Konflikt wird betroffenes Objekt deaktiviert. - *Tear*: Von zwei Nutzer in verschiedene Richtungen gezogenes Objekt zerreißt.	- *Voting*: Es wird unter den betroffenen Nutzern abgestimmt.
Beide Konflikt-parteien	- *Rang:* siehe andere Spalte - *Geschwindigkeit, Kraft*: der Zugriff der mit der schnelleren Geste oder größerem Druck auf den Tisch erfolgte wird ausgeführt.	- *Rang*: Jeder Benutzer hat einen eigenen Rang. Bei einem Konflikt gewinnt der höchste Rang.

Tabelle 2.1: Übersicht über die Konfliktlösungsrichtlinien nach [MRSF04]

2.1.3 Auswirkungen des Blickwinkels

Wenn bei einer kollaborativen Arbeit sich die Beteiligten an unterschiedliche Tischseiten verteilen, sieht jeder von ihnen den zu bearbeitenden Inhalt aus einem anderen Blickwinkel. Dies kann zur Beeinträchtigung der Arbeit führen.

So ist Text mit einem Blickwinkel von über ±45° schwer lesbar, was jedoch nur bei größeren Mengen an Text zu einer erheblichen Beeinträchtigung führt. [WiBa05] Ein möglicher Lösungsansatz dieses Problems auf Tabletop PCs besteht darin, dass die Anwendung Textfelder mehrmals in verschiedenen Ausrichtungen anzeigt. [WiBa05]

Manchmal beeinflusst die relative Positionierung der Elemente eines Diagramms, wie schnell dieses zu verstehen ist. [Witt11] Zum Beispiel werden Familienstammbäume oft so visualisiert, dass die Vorfahren in der Zeichnung über den Nachfahren stehen. Separat drehbare Diagrammelemente oder eine drehbare Arbeitsfläche können hier helfen [HCVW06]. Beide Lösungsansätze sind auch auf das Textproblem anwendbar.

2.1.4 Arten der Zusammenarbeit

Viele durch kollaboratives Arbeiten lösbare Aufgaben lassen sich in Unteraufgaben unterteilen. Jeder Nutzer kann sich dann seine eigene Unteraufgabe vornehmen und diese individuell für sich bearbeiten. Die Einzelergebnisse müssen irgendwann zusammen geführt werden, was wieder Austausch innerhalb der Gruppe erfordert. Geschieht das Zusammenführen im Laufe einer kollaborativen Arbeit häufig, findet ein reger Wechsel zwischen loser und enger Zusammenarbeit statt, was auch Mixed Focus genannt wird. [GuGr98]

Tang, Tory u. a. haben mit einer Studie die Koppelung innerhalb einer Gruppe bei der Verwendung eines Tabletop PCs näher untersucht und jeweils drei Arten des individuellen Arbeitens und des gemeinsamen Arbeitens ermittelt. Tabelle 2.2 gibt einen Überblick über diese. Der Wechsel zwischen den einzelnen Arten erfolgt flüssig. Außerdem zeigt die Studie, dass für verschiedene Arten des Zusammenarbeitens verschiedene Werkzeuge geeignet sind. [TTPN06] So werden Werkzeuge, welche einen großen Teil der Arbeitsfläche betreffen, eher beim engen Zusammenarbeiten eingesetzt. Dagegen werden

für das individuelle Arbeiten Werkzeuge benutzt, deren Einsatz einen so kleinen Bereich beeinflusst, dass der Anwender nicht die Arbeit eines anderen beeinträchtigt.

Individuelles Arbeiten	Gemeinsames Arbeiten
- Eine Person erledigt eine Aufgabe und eine andere schaut so genau zu, dass sie Metaereignisse wie zum Beispiel das Freiwerden an Ressourcen mitbekommt. - Während eine Person ein Problem bearbeitet, ist eine andere in keiner Weise involviert. - Zwei Personen bearbeiten zwei verschiedene Unteraufgaben. Oft positionieren sie sich dabei so, dass sie aus den Augenwinkeln die Tätigkeiten des jeweils anderen verfolgen.	- Zwei Personen bearbeiten oft miteinander sprechend das gleiche Probleme am selben Ort - Eine Person kümmert sich um eine Aufgabe, während eine andere es so genau verfolgt, dass sie über alle Details der Lösung informiert ist. Beide kommunizieren oft. - Das gleiche Problem wird von zwei Personen an unterschiedlichen Orten bearbeitet, um zum Beispiel Alternativen zu erarbeiten. Es findet keine Kommunikation statt.

Tabelle 2.2: Unterschiedliche Arten des individuellen / gemeinsamen Arbeitens [TTPN06]

Die Palette an virtuellen Werkzeuge die eine Anwendung zur Verfügung stellt, sollte den regen Wechsel zwischen den einzelnen Arten der Zusammenarbeit unterstützen und keinesfalls hemmen.

2.2 Gesten

Im allgemeinen Sprachgebrauch bedeutet der Begriff *Geste* die Körperbewegung eines Menschen sowie einiger Tiere, die zur Kommunikation mit anderen Lebewesen dient.

Im Kontext dieser Arbeit dagegen ist eine Geste der definierte Bewegungsablauf eines oder mehrerer Finger auf der berührungsempfindlichen Oberfläche des Tabletop PCs. Eine Geste beginnt mit dem Berühren des Tisches und endet spätestens beim los lassen. Wenn die laufende Anwendung eine bestimmte Geste erkennt, löst sie eine bestimmte Aktion aus.

2.2.1 Gesten als direkte Eingabemethode

Ein wesentlicher Vorteil der Verwendung von Gesten gegenüber konventionellen externen Eingabegeräten wie Maus, Tastatur, Joystick und ähnlichem ist, dass sie zur direkten Eingabe verwendet werden können.

Dies bedeutet, dass der sichtbare Effekt einer Geste auf dem Bildschirm dort sichtbar ist, wo sie ausgeführt wird. [Witt11] Zum Beispiel kann eine Anwendung ihren Nutzern ermöglichen virtuelle Objekte mit einem Finger über die Arbeitsfläche zu ziehen (als Drag und Drop Geste bekannt). Dabei befindet sich das Objekt stets unter der Fingerspitze.

Bei Einsatz einer Maus für Drag und Drop bedarf es dagegen mit dem Mauszeiger eines virtuellen Fingers, der durch die Hand des Nutzers ferngesteuert wird. Wenn wie in diesem Falle der Eingabefokus (Mauszeiger) örtlich von seiner Steuerung (Maus) getrennt ist, handelt es sich um eine indirekte Eingabemethode.

Direkte Eingabe ist effizienter als indirekte, da letztere durch ihre Parallelität sowie das Umsetzen einer Bewegung auf eine andere mehr Denkleistung erfordert. [ScBG09]

Bei kollaborativen Tabletop PC Anwendungen hat direkte Eingabe noch den Vorteil gegenüber indirekter, dass sie bei der Vermeidung ungewollter Konflikte unterstützen kann. Denn die Armbewegungen der Nutzer, die gerade Gesten ausführen, können anderen Personen am Tisch klarer zeigen, auf welchen Teilen der Arbeitsfläche aktuell gearbeitet wird, als zum Beispiel ein Mauszeiger oder ein Textcursor.

2.2.2 Genauigkeit Finger basierter Eingabe

Die Spitze eines Mauszeigers ist nur einen Pixel groß. Deshalb und weil eine Anwendung eine schnelle Mausbewegung auf eine langsamere Bewegung des Zeigers umsetzen kann, ist es möglich letzteren Pixel genau zu platzieren. Eine Fingerspitze dagegen berührt gleich mehrere Bildpunkte des Tabletop PCs auf einmal und der Finger verdeckt den Zielpunkt. Dadurch ist eine Pixel genaue direkte Eingabe durch einen Finger nicht möglich. [Witt11]

Eine einfache Lösung ist die Benutzerschnittstelle einer Tabletop PC Anwendung so zu entwerfen, dass berührbare virtuelle Objekte mindestens so groß wie eine Fingerspitze sind. [Micr11]

Dieser Ansatz ist jedoch bei einigen Anwendungsgebieten nicht einsetzbar wie zum Beispiel dem konventionellen Textcursor oder bei der Erstellung von Zeichnungen auf einer Karte. Die Vergrößerung des zu bearbeitenden Inhalts birgt die Gefahr, dass wichtige Kontextinformationen nicht mehr auf der verfügbaren Bildschirmfläche angezeigt werden können. [RWZB10] Außerdem kann es zu Konflikten mit anderen Benutzern (siehe 2.1.1) kommen. Unter anderem folgende Techniken begegnen den beschriebenen Problemen, indem sie eine hoch präzise Eingabe gewährleisten:

- Der Inhalt um das Ziel herum, wird nur in einem lokal begrenzten Bereich vergrößert wie zum Beispiel bei der Verwendung einer Lupe. Hierbei kann der Benutzer den Vergrößerungsfaktor unter anderem durch den Einsatz eines zweiten Fingers [BeWB06] stetig oder eine spezielle Rubbelgeste diskret [OlFH08] steuern.
- Die Fingerspitze steuert einen virtuellen Zeiger, welcher leicht versetzt zum Finger angezeigt wird. Dabei ist der Abstand zwischen Finger und Zeiger möglichst klein zu halten, damit die negativen Effekte der indirekten Eingabe klein bleiben. Ein zweiter Finger kann den Primärfinger unterstützen, indem er zum Beispiel die Geschwindigkeitsumsetzung von Finger auf Zeiger anpasst. [BeWB06]

2.2.3 Klassifikation und Entwurf von Gesten

Gesten können unter anderem nach folgenden Kategorien klassifiziert werden [WoMW09]:

- Die *Form* bestimmt die Anzahl der involvierten Finger sowie Hände und ob letztere an einer Stelle verbleiben.
- Die *Natur* einer Geste gibt an, ob sie mit der ihr zugeordneten Aktion symbolisch, physikalisch, bildlich oder abstrakt in Verbindung steht.
- Der *Kontext* einer Geste kann Objekt bezogen, Arbeitsflächen bezogen oder eine Mischung aus beidem sein. Oder die Geste bedeutet in jedem Kontext das Gleiche.
- Der *Fluss* einer Geste gibt an, ob diese stetig oder diskret ist. Bei stetigen Gesten wird der Effekt sofort beim Ausführen der Geste sichtbar (zum Beispiel beim

Ziehen eines virtuellen Objekts), während bei diskreten Gesten der Effekt erst beim Beenden der Geste ausgelöst wird (zum Beispiel Malen eines „X“ zum Schließen eines Dialogs).

Damit ein Nutzer den Einsatz von Gesten als angenehm empfindet, sollten diese für ihn schnell erlernbar sein. Am Besten sollte er sich auf Grund von Alltagserfahrungen die Bedeutung einer Geste erschließen können. Die Geste sollte also möglichst intuitiv sein.

Studien haben untersucht, was solche Gesten ausmacht und für übliche (über viele Anwendungen hinweg angebotene) Aktionen passende Gesten gesucht. Es folgt eine grobe Zusammenfassung einiger Ergebnisse:

- Die Wahl oder die genaue Ausführung (zum Beispiel Ziehen mit einem oder zwei Fingern) einer Geste hängt von vorher ausgeführten Gesten und sozialen Umständen wie zum Beispiel den anderen Tischbenutzern ab. [HiCa11]
- Bei zweihändig ausgeführten Gesten verwenden Nutzer in mehr als der Hälfte der Fälle ihre beiden Hände symmetrisch bezüglich Haltung, Fingerspitzen auf der Oberfläche und Bewegungsrichtung. [HiCa11]
- Oft führen Tischnutzer mehrere einfache Gesten direkt hintereinander aus, um ein bestimmtes Ziel zu erreichen. [HiCa11] Ein Beispiel wäre, wenn ein Nutzer ein virtuelles Objekt erst zieht und dann dreht, um es vor sich auszurichten. Daher ist es wichtig, einfache Gesten so zu gestalten, dass jederzeit ein fließender Übergang von der einen in die andere möglich ist.
- Ein Hand Gesten werden Gesten mit zwei Händen vorgezogen. [WoMW09]
- Wenn nicht vom System vorgegeben, achten die meisten Nutzer nicht darauf, ob sie eine Geste mit ein, zwei oder drei Fingern ausführen. [WoMW09]

2.2.4 Gesten und Kollaboration

Bei kollaborativer Verwendung von auf Gesten basierten Eingabegeräten, kann es passieren, dass zwei Nutzer gleichzeitig eine Geste auf einem virtuellen Objekt oder der Arbeitsfläche ausführen wollen. Zusammen können die beiden Gesten eventuell eine andere Geste darstellen. Um dadurch entstehende und ungewollte Aktionen verhindern zu können, muss eine Anwendung erkennen, dass verschiedene Berührungen des Bildschirms im gleichen Kontext von verschiedenen Benutzern kommen. Der erkannte Konflikt kann dann zum Beispiel nach den in 2.1.2 beschriebenen Richtlinien gelöst werden.

Abseits von dieser Problematik gibt es jedoch noch die Idee von kollaborativen Gesten, bei denen mehrere Benutzer ausdrücklich eine Geste gemeinsam ausführen. Ein Beispiel dafür wäre das Bewegen eines virtuellen Objekts über große Distanzen hinweg, indem ein Nutzer den Start und ein anderer das Ziel markiert. Morris u. a. haben sich in einer Studie mit kollaborativen Gesten näher auseinander gesetzt [MHPW06] und Folgendes erarbeitet:

- Der Sinn der für eine Aktion benötigte Kollaboration sollte für die Nutzer klar sein.
- In einem großflächigem Kontext wie zum Beispiel dem gesamten Bildschirm sollten nicht mehrere Einbenutzergesten zusammen einer kollaborativen Gesten

gleichen, da diese sonst versehentlich ausgelöst werden könnte.

- Wenn eine häufig bei einer individuellen Arbeitsphase (siehe 2.1.4) benötigte Aktion nur kollaborativ ausgelöst werden kann, müssen andere Benutzer bei ihren individuellen Arbeiten entsprechend häufig unterbrochen werden. Dies kann zu Ineffizienz in der Gruppenarbeit auf Grund von hohem Koordinationsaufwand führen.

Der Einsatz von kollaborativen Gesten sollte also wohl überlegt erfolgen.

2.2.5 Gestenerkennung

Dieses Kapitel geht davon aus, dass das Betriebssystem des Tabletop PCs in regelmäßigen Zeitabständen Schnappschüsse mit folgenden Daten liefert: Zeitstempel und Fingerpositionen der sich auf der Oberfläche befindlichen Finger. Jeder Finger lässt sich dabei, solange er nicht abgehoben wird, über mehrere Datenpakete hinweg eindeutig identifizieren.

Einfache Gesten lassen sich dann durch Vergleichen von aufeinander folgender Schnappschüsse und Vektorrechnung realisieren. Zum Beispiel bei der „Ziehen eines virtuellen Objekts“ Geste berechnet man mit jedem neuen Schnappschuss die relative Positionsveränderung des Fingers auf dem Objekt als Vektor und addiert diesen zur absoluten Position des Objekts. Abbildung 2.1 stellt das Vorgehen bildlich dar. F_{neu} ist dabei die neue Fingerposition, F_{alt} die Alte und P_{alt} die alte Position der linken oberen Ecke des Objekts. Der schwarze Pfeil stellt den Bewegungsvektor dar.

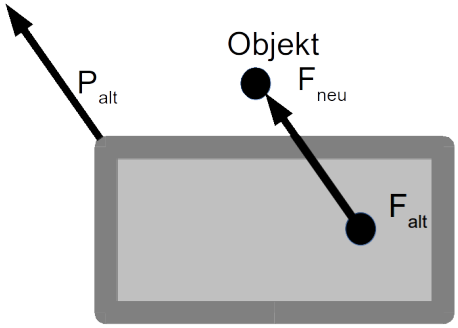

Abbildung 2.1: Umsetzung einer Ziehen Geste

Der Zeitstempel macht es auch möglich noch weitere Eigenschaften der Geste wie Geschwindigkeit und Beschleunigung zu berechnen.

Zur Erkennung bestimmter Formen (zum Beispiel Kreise oder Linien) in der für die Geste ausgeführten Bewegung können die Bewegungsvektoren der einzelnen Finger seit erstmaligem Berühren der Oberfläche hintereinander verkettet werden. Auf diese Vektorketten kann man dann Algorithmen anwenden, die zum Beispiel in der Lage sind Ecken an Hand des Winkels zwischen zwei aufeinander folgenden Vektoren zu erkennen. [Kila12]

2.3 Microsoft Pixelsense[1]

Dieses Kapitel erläutert die wichtigsten Eigenschaften des Microsoft Pixelsense. Bei diesem handelt es sich um die Tabletop PC Plattform, auf welcher der Diagramm Editor läuft.

2.3.1 Hardware

Als Tabletop PC besitzt der Pixelsense ein Gehäuse, dessen Form einem Tisch ähnelt. Als Tischplatte dient ein Bildschirm.

Unter der Verwendung von Infrarot Technik ist der Pixelsense in der Lage Berührungen auf dem Bildschirm zu erkennen. Diese können Fingern oder Objekten zugeordnet werden. Objekte lassen sich dabei entweder über ihre Form oder über spezielle Tags in Form von Aufklebern identifizieren. [Micr00a] [Micr00b] [Micr00c]

Angesteuert wird die virtuelle Arbeitsfläche von konventionellen PC Komponenten.

Den Pixelsense gibt es in zwei Versionen. (Stand 2013) Abbildung 2.2 zeigt die Ältere Version, während Abbildung 2.3 die Neuere abbildet. Ein wesentlicher Unterschied ist, dass die neuere Version durch die Tischbeine sitzenden Benutzern eine größere Beinfreiheit

Abbildung 2.2: Hardware der Pixelsense Plattform Version 1.0 (Surface) [Micr00i]

Abbildung 2.3: Hardware von Pixelsense Version 2.0 [Zdne11]

gewährt. Bildfläche, Auflösung und Leistung sind bei der neueren Version größer als bei der Alten. Tabelle 2.3 gibt eine Übersicht über die Kenndaten der beiden Geräteversionen.

	Version 1.0	Version 2.0
Bildfläche, Auflösung	30 Zoll, 1024 x 768 Pixel	40 Zoll, 1920 x 1080 Pixel
Größe (Breite x Tiefe x Höhe)	108cm x 68,6cm x 55,9cm	109,5cm x 70,7cm x 72,8cm
Prozessor, Arbeitsspeicher, Grafikkarte	2,13 Ghz Intel Core 2 Duo, 2 GB, ATI X1650	Athlon X2 Dual-Core 245e, 4 GB DDR3, AMD HD6750M

Tabelle 2.3: Übersicht über Pixelsenses Hardware Eigenschaften [Micr00h] [Sams13]

Die unterschiedlichen Pixelsense Versionen verwenden unterschiedliche Hardware

1 Der ursprüngliche Name der Plattform war Surface, bevor sie umbenannt worden ist. [Thev00] Diese Arbeit verwendet den neuen Namen für die Hardware und den alten Namen für die SDKs.

Komponenten zur Erfassung von Berührungen.

2.3.2 Software

Auf dem Pixelsense läuft Windows als Betriebssystem. Anwendungsentwickler können das Microsoft Pixelsense SDK verwenden. Dieses setzt unter anderem auf das .net Framework auf und erweitert dieses um Unterstützung für den berührungsempfindlichen Bildschirm des Pixelsense. So wird zum Beispiel das GUI System Windows Presentation Foundation (WPF) um Eingabeelemente erweitert [Micr00d], welche für eine Tabletop PC Anwendung nützlich sind.

Abbildung 2.4: Scatter View [Micr13]

Ein Beispiel für diese ist das ScatterView GUI Element (siehe Abbildung 2.4). Dieses kann mit ScatterViewItems befüllt werden, welche außer Bildelementen wie auf der Abbildung dargestellt beliebige andere GUI Elemente wie zum Beispiel Knöpfe oder Menüs beinhalten können. Desweiteren können die ScatterViewItems vom Benutzer im ScatterView verschoben, gedreht und vergrößert werden. [Micr13] Ein paar einfache Gesten und ihre Erkennung sind also bereits durch das SDK gegeben.

Wie bei der Hardware des Pixelsense gibt es auch Versionsunterschiede bei der Software. Auf der älteren Version der Plattform läuft Windows Vista und .net 3.5 und auf der Neueren Windows 7 und .net 4.0 [Micr00e]. Das Microsoft Surface SDK gibt es der Plattform entsprechend in den Versionen 1.0 und 2.0. Zwar laufen die meisten Software Komponenten der neueren Pixelsense Version auch auf der Alten. Jedoch ist der Treiber für die ältere Berührungserfassung inkompatibel zum neueren SDK. Anwendungen, die auf beiden Plattform Versionen laufen sollen, können mit Adaptern arbeiten. [Witt11]

2.3.3 Model View ViewModel Pattern

Entwickler können .net und WPF basierte Anwendungen nach dem Model View ViewModel (MVVM, siehe Abbildung 2.5, [Smit09]) Pattern entwerfen und implementieren.

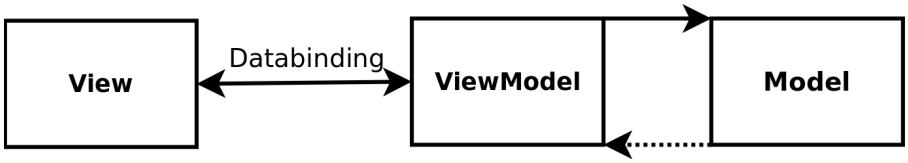

Abbildung 2.5: Model View ViewModel Pattern

Bei diesem Pattern verwaltet ein Model Daten und stellt über Schnittstellen Funktionen zum Ändern dieser bereit. Es realisiert also neben der reinen Datenverwaltung auch die Geschäftslogik einer Anwendung. Änderungen der Daten teilt ein Model über das Beobachter Pattern (siehe [Gamm95]) mit. Dabei weiß das Model über seine Beobachter, die ViewModel, nicht mehr als den Beobachterstatus.

Die ViewModel setzen Aktionen des Nutzers auf der Benutzerschnittstelle, welche durch die Views definiert wird, auf Funktionen der Modelle um und bereiten die Daten zur Anzeige durch die Views auf. Dabei stellen sie Datenattribute bereit, an welche sich die Views binden.

Die Koppelung (Databinding) zwischen View und ModelView ist so lose definiert, dass View und ViewModel in verschiedenen Programmiersprachen implementiert sein können. WPF basierte Views werden zum Beispiel in der Sprache XAML definiert, während die ViewModels zum Beispiel in C# oder VB.net implementiert werden können.

2.3.4 Texteingabe

Das Surface SDK unterstützt nur einen Texteingabefokus zur gleichen Zeit. Das heißt, dass immer nur ein Benutzer Text eingeben kann. Als Eingabemethode kann eine Tastatur dienen. Die Texteingabe ist in diesem Falle eine indirekte Eingabemethode (siehe 2.2.1), da das Steuergerät die Tastatur lokal vom Zeiger dem Textcursor, wo der Effekt statt findet, getrennt ist. [Witt11]

Die Nutzer des Tabletop PCs können entweder eine virtuelle Tastatur verwenden oder auf eine konventionelle physikalische zurückgreifen. Erstere bietet den Vorteil, dass die Anwendung sie nahe des zu bearbeitenden Textfelds mit entsprechender Ausrichtung bei Bedarf erscheinen lassen und nach Verwendung wieder verschwinden lassen kann. Allerdings hat eine virtuelle Tastatur den Nachteil des fehlenden Feedbacks. Dies erschwert das fehlerfreie Tippen. [HoBJ08]

Eine direkte Texteingabemethode wäre das Schreiben direkt auf den Tabletop PC mit dem Finger. Jedoch gleicht ein Finger einem dicken Filzstift, mit dem Text größer wird als zum Beispiel unter der Verwendung eines feinen Kugelschreibers.

2.3.5 Benutzererkennung / -Unterscheidung

In Kapitel 2.2.4 wurde die Notwendigkeit erwähnt, Berührungen bestimmten Nutzern zuordnen zu können oder zumindest bestimmen zu können, dass zwei oder mehr Berührungen von unterschiedlichen Tischnutzern kommen. Auch einige der in Kapitel 2.1.2 beschriebenen Konfliktlösungsrichtlinien haben diese technische Voraussetzung. Besonders die Richtlinien, welche mit Zugriffsrechten für bestimmte Nutzer auf bestimmte Objekte oder mit Eigenschaften des zugreifenden Nutzer arbeiten, benötigen eine eindeutige Zuordnung einer Berührung zu einem Nutzer.

Der Pixelsense Tabletop PC bietet die Möglichkeit Infrarot Tags zu identifizieren und deren Position und Drehung zu bestimmen. Es gibt kleinere Byte Tags (1,9cm x 1,9cm) und größere Identitätstags. Erstere können 8 Bit und Letztere 128 Bits in ihrer Musterung codieren. Das heißt, dass es bis zu 256 unterschiedliche Byte Tags geben kann. [Micr00b] Die Finger eines Handschuhes können mit unterschiedlichen Tags beklebt werden. [MaKG10] Dadurch ist sogar eine eindeutige Bestimmung des Fingertyps (zum Beispiel Zeigefinger) bei einer Berührung möglich. Der Pixelsense kann zwar auf Grund der feinen Musterung auf dem Tag die Position dessen genauer bestimmen als die eines Fingers. Aber der Tag verdeckt auch eine größere Fläche als ein Finger, wodurch das in Kapitel 2.2.2 beschriebene Problem verstärkt wird. Letztendlich trägt sich ein Handschuh im Sommer im warmen Büro nicht komfortabel.

Jedoch kann die Idee mit den Tags eine Benutzerbestimmung vorzunehmen zum Beispiel teilweise weiterverfolgt werden, indem jeder Benutzer eine eigene getagte „Spielfigur“ bekommt.

Eine weitere technische Möglichkeit, um zumindest eine Benutzerunterscheidung bei Berührungen durchzuführen, ist dadurch gegeben, dass der Pixelsense die Ausrichtung einer Fingerspitze bestimmen kann. Mit dieser Information lassen sich zum Beispiel am Tisch gegenüber positionierte Personen bei Berührungen auseinander halten, da es physisch schwer ist, seine Hand 180° zu sich selber zu drehen.

2.4 NShape

Die Implementierung der Anwendung baut auf dem Diagramm Verwaltungs Framework NShape auf. Dieses Kapitel erläutert jenes.

2.4.1 Eigenschaften

NShape [Data14] ist in C# geschrieben und baut auf dem .net Framework auf. Es weist folgende Eigenschaften auf, welche für den entwickelten Editor von Nutzen sind [Pohm09]:

- **Templates:** Die einzelnen Formen (z.B. Klassen bei UML Klassendiagrammen) lassen sich von Templates ableiten. Mit diesen teilen sich jene dann bestimmte Attribute (z.B. Farbe). Wird der Wert eines Attributs im Template geändert, übernehmen alle abgeleiteten Formen den neuen Wert.
- **Designvorlagen:** Attribute können auch mit Stilen aus einer Designvorlage verknüpft werden. Damit übernimmt das Attribut den Wert des Stils des aktiven Designs. Dadurch können die Werte eines Attributs für viele Diagrammelemente gleichzeitig an zentraler Stelle angepasst werden.
- **Abspeichern von Diagrammen** in einer XML Datei oder in einer SQL Datenbank.
- **Trennung von Daten, Bearbeitungslogik, Repräsentationslogik und Windows Anzeigeelementen:** siehe 2.4.2 Aufbau des Frameworks
- **Erweiterbarkeit:** NShape bietet bereits eine Menge an Implementierungen universell einsetzbarer Diagrammformen an, die beliebig erweitert werden kann.
- **Schnittstelle zum Anbinden an eigene Modelle** erlaubt die Erweiterung des Frameworks, so dass es bei der Diagrammbearbeitung die Regeln eines bestimmten Modells beachtet.

2.4.2 Aufbau des Frameworks

Die Komponenten von NShape können in die Gruppen Core, Controller, Presenter und Controls unterteilt werden. [Data12] Abbildung 2.6 gibt eine Übersicht inwieweit die einzelnen Komponenten zusammenhängen. Es gibt Ähnlichkeiten wie Unterschiede zum MVVM Pattern (siehe 2.3.3). Der Core bei NShape übernimmt die gleiche Aufgabe wie das Model und verwaltet die Datenstrukturen. Die ViewModels des MVVM Pattern sind bei NShape in zwei Komponentengruppen aufgeteilt. Während die Controller die Bearbeitungslogik für die Daten enthalten, ist es Aufgabe der Presenter Komponenten die

Anzeigelogik der Daten umzusetzen. Dazu greifen sie auf Elemente der WinForm GUI (Vorgänger von WPF) über eine C# API zu. Wie bei MVVM kommt das Beobachterpattern zum Einsatz. Die Presenter sind Beobachter der Controller und diese wiederum beobachten den Core.

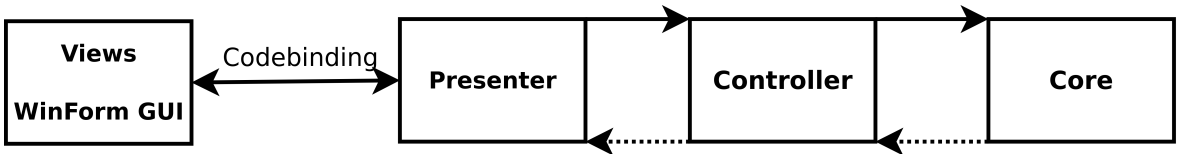

Abbildung 2.6: NShape Komponentenübersicht

3 Anforderungen

Dieses Kapitel erläutert die Anforderungsanalyse und ihre Ergebnisse. Zunächst erklärt ein Unterkapitel die in der Arbeit verwendeten formellen Bezeichner und Arten der Anforderungen. Anschließend dokumentiert je ein Unterkapitel die Analyse der funktionalen, nicht funktionalen sowie technischen Anforderungen.

3.1 Bezeichner und Arten der Anforderungen

Jede Anforderung besitzt einen eindeutigen Bezeichner, damit folgende Kapitel leicht auf eine bestimmte Anforderung verweisen können. Der Bezeichner besteht aus einem A (für Anforderung), gefolgt von einer eindeutigen dezimalen Nummern und einem abschließenden Kürzel, welches die Art der Anforderung bestimmt (siehe Tabelle 3.1).

Kürzel	Art	Bedeutung
F	funktional	Die funktionalen Anforderungen bestimmen die Funktionalitäten, welche die Anwendung erfüllen muss.
NF	nicht funktional	Nicht funktionale Anforderungen beschreiben nach welchen Qualitätsmaßstäben (zum Beispiel Geschwindigkeit der Ausführung) die funktionalen Anforderungen erfüllt werden sollen.
T	technisch	Eine technische Anforderung legt Techniken (zum Beispiel Software Bibliotheken, Hardware) für die Realisierung der Anwendung fest.

Tabelle 3.1: Arten der Anforderungen

3.2 Funktionale Anforderungen

Dieses Unterkapitel hält die funktionalen Anforderungen bezüglich der drei Aspekte Graphen basierte Diagramme, Tabletop PCs und kollaboratives Arbeiten fest.

Die Anwendung richtet sich an Benutzergruppen von mehreren Personen, die gemeinsam unter Verwendung von auf Graphen basierten Diagrammen Zusammenhänge zwischen Informationen zum Beispiel für Planungszwecke visualisieren.

3.2.1 Graphen basierte Diagramme

Bei den Diagrammen kann es sich unter anderem um Visualisierungen eines bestimmten Ablaufs oder Aufbaus handeln. Konkrete Beispiele aus der Informatik für Ablaufpläne sind Programmablaufpläne, welche das Verhalten einer Anwendung darstellen (Abbildung 3.1) und für Aufbaudiagramme Blockschaltbilder, die unter anderem dazu verwendet werden die logischen Bausteine eines Schaltnetzes zu visualisieren (Abbildung 3.4). Auch in anderen Fachgebieten werden Diagramme verwendet. In der Wirtschaft gibt es zum Beispiel nach der BPMN Notation erstellte Diagramme zur Darstellung von Geschäftsprozessen (Abbildung 3.3) [Witt11] [Wiki13] oder Organigramme, die Unternehmensstrukturen aufzeigen (Abbildung 3.2).

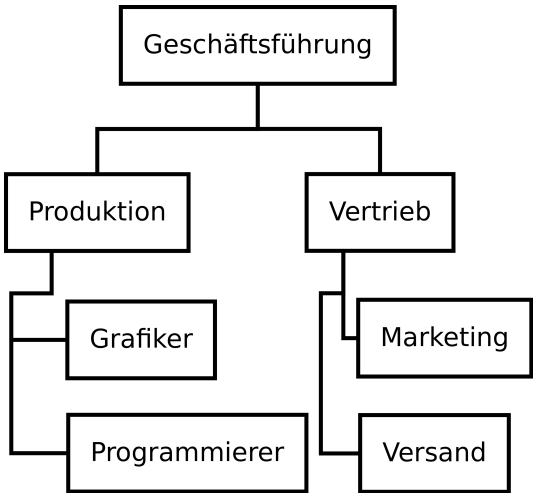

Abbildung 3.2: vereinfachtes Organigramm für ein IT Unternehmen

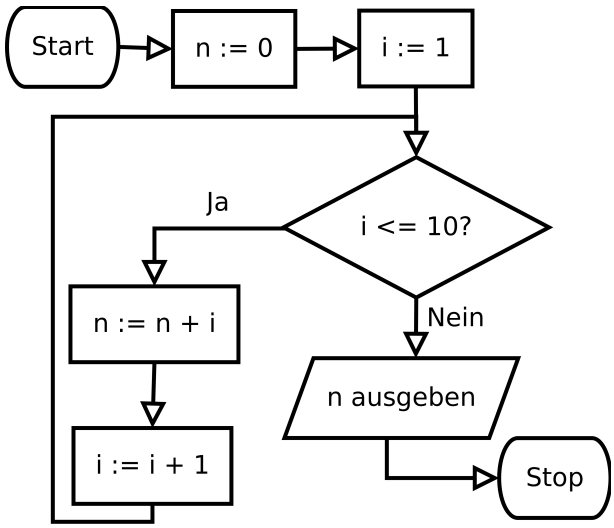

Abbildung 3.1: Programmablaufplan einer Anwendung, welche 1 bis 10 aufsummiert

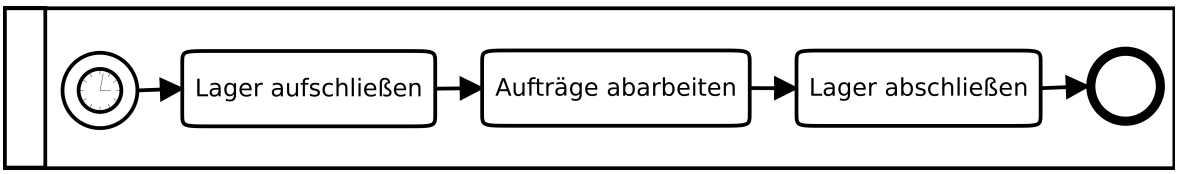

Abbildung 3.3: Beispiel BPMN, Tagesablauf des Versands aus Abbildung 3.2

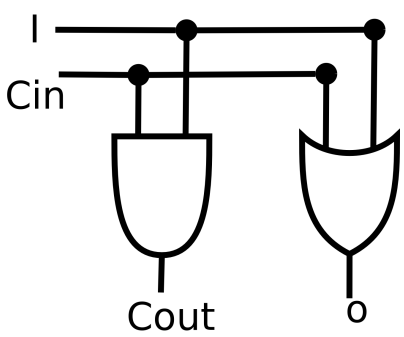

Abbildung 3.4: Halbaddierer, Symbole nach ANSI 91

Alle genannten Diagrammbeispiele haben gemeinsam, dass ihnen ein Graph zu Grunde liegt. Das bedeutet, dass es Knoten und Verbindungen zwischen diesen gibt. Oft werden die Knoten durch beschriftete geometrische Grundformen wie Rechtecke, Kreise oder Trapeze dargestellt. Doch auch beliebige Bilder (◎○⛉⛉•) können Knoten repräsentieren. Die Verbindungen basieren alle auf Linien zwischen den Knoten, die aus ein oder mehreren nacheinander folgenden Geraden bestehen. Pfeilspitzen können auf gerichtete Linien hinweisen. Manche Verbindungen sind beschriftet. Es ist nicht bei jeder Diagrammart egal, an welcher Stelle die Verbindungslinien auf den Rand der Knoten treffen. Auch erlaubt nicht jeder Knoten beliebig viele Verbindungen.

Daraus folgt, dass der Diagrammeditor, um eine große Vielfalt an Graphen basierten Diagrammnotationen unterstützten zu können, folgende Anforderungen erfüllen muss:

- **A1F** Der Editor muss den Nutzern ermöglichen, Knoten zu erstellen, zu verändern, zu verschieben, zu vergrößern, zu drehen und zu löschen.
- **A2F** Die Menge der erstellbaren Knoten muss auf geometrische Grundformen basierte Knoten enthalten und sollte über Bilder erweiterbar sein.
- **A3F** Die Nutzer können auf Geraden basierte Verbindungspfeile zwischen den Knoten ziehen. Dabei sind die Pfeilspitzen optional.

- **A4F** Die Knoten können Verbindungspunkte mit einer zu ihnen relativen Position aufweisen. Diese Punkte legen dann fest, wo eine Verbindungsgerade auf den Knoten treffen kann.

Oft sind Diagramme die Darstellung eines Modells, welches mehr Eigenschaften besitzen kann als dargestellt werden. Eventuell wollen Nutzer des Diagrammeditors neben dem Anzeigen noch weitere Funktionen wie Simulation auf das Modell aus dem Editor heraus anwenden können. Der Diagrammeditor soll die Bearbeitung einer großen Vielfalt an Diagrammen unterstützen und damit die Visualisierung sehr unterschiedlicher Modelle ermöglichen. All jene Funktionen wie Simulation zu realisieren, die ein tiefes Verständnis des Modells erfordern, ist für eine Bachelorarbeit ein zu großer Aufwand und ohne Einschränkung auf eine bestimmte Menge an Modellen generell nicht möglich. Allerdings kann der Diagrammeditor Schnittstellen zur Verfügung stellen, die dem Nutzer erlauben, ihn bei Bedarf über Plugins um die Modell spezifischen Funktionen zu erweitern. Daraus ergeben sich zwei weitere Anforderungen:

- **A5F** Der Editor muss ein Plugin System aufweisen, wobei Plugins dem Editor Funktionalitäten hinzufügen.
- **A6F** Außer den zur Visualisierung der Diagramme notwendigen Daten muss der Editor beliebige weitere verwalten und sie den Plugins zur Verfügung stellen können.

3.2.2 Tabletop PCs

Da der Diagrammeditor Gesten gestützt und damit über direkte Eingabe bedient werden soll, überprüft dieses Unterkapitel, inwieweit die in Kapitel 2.2 beschriebenen möglichen Probleme auf die Bearbeitung von Diagrammen zutreffen.

Es kommen bei Diagrammen häufig Elemente vor, die komplett oder in einer räumlichen Dimension kleiner als eine Fingerspitze sind:

- kleine Knoten wie zum Beispiel die ● Knoten im Blockschaltbild aus Kapitel 3.2.1
- feine Verbindungslinien zwischen den Knoten
- die Verbindungspunkte an den Knoten, welche festlegen wo eine Verbindungslinie genau auf den Knoten treffen kann

Daher wird folgende Anforderung definiert:

A7F Der Editor muss seine Nutzer aktiv bei dem Bearbeiten kleiner Elemente unterstützen.

Diagramme weisen meistens keinen festen Größenmaßstab auf. Das heißt, man kann sie nach Belieben verkleinern und vergrößern, um zum Beispiel große Diagramme als Ganzes auf dem Bildschirm des Tabletop PCs ansehen zu können. Ist die Verkleinerung des Diagramms zu stark, kann es passieren, dass zum Beispiel Text nicht mehr lesbar ist. Daher sollte der Editor neben dem Skalieren ganzer Diagramme eine virtuelle Arbeitsfläche anbieten, die größer als der Bildschirm ist. Bei der Größenänderung der

Diagramme ist zu beachten, dass die Verbindungslinien immer sichtbar sein sollten, da sie ein wesentlicher Bestandteil Graphen basierter Diagramme sind. Folgend werden weitere Anforderungen festgelegt:

- **A8F** Die Anwendung erlaubt die Vergrößerung beziehungsweise Verkleinerung von ganzen Diagrammen. Dabei wird die Dicke der Verbindungslinien nicht verändert.
- **A9F** Der Diagrammeditor weist eine virtuelle Arbeitsfläche auf, die großflächiger sein kann als der Bildschirm. Dieser zeigt dabei einen verschiebbaren Ausschnitt der Arbeitsfläche an.

Die Bedienung des Editors sollte leicht zu erlernen sein, so dass sich die Nutzer auf die Inhalte und das gemeinsame Arbeiten konzentrieren können. Entsprechend gilt für die gewählten Gesten folgende Anforderung:

A10F Die vom Editor unterstützten Gesten müssen möglichst natürlich sein und folglich die dazu in Kapitel 2.2.3 gelisteten Richtlinien erfüllen.

3.2.3 Kollaboratives Arbeiten

Der Editor soll die Bearbeitung der Diagramme durch mehrere Personen gleichzeitig erlauben. Kapitel 2.1.1 hat erläutert, dass die virtuelle Flächenaufteilung des Tisches für die Koordination zwischen den Gruppenmitgliedern nach erlernten Verhaltensregeln eine wichtige Rolle spielt. Deshalb sollte die Benutzerschnittstelle des Editors so gestaltet werden, dass sie der Aufteilung des Tisches nicht im Wege steht.

Indirekte Eingabe (siehe Kapitel 2.2.1) erleichtert Nutzern die sozialen Regeln versehentlich oder mit Vorsatz zu verletzen. Dies führt zu folgender Anforderung:

A11F Auf den Einsatz von indirekten Eingabemethoden sollte beim Editor wenn möglich verzichtet werden.

Lagerflächen dagegen unterstützten die Koordination beim gemeinsamen Arbeiten, was in eine weitere Anforderung mündet:

A12F Der Editor muss virtuelle Lagerflächen für Knoten und Verbindungen anbieten. Diese Lagerflächen können beliebig viele Elemente aufnehmen und die Ansicht auf diese lässt sich vergrößern sowie verkleinern und beliebig auf dem Bildschirm platzieren.

Die Arbeitsfläche des Tabletop PCs, für welchen der Editor realisiert werden soll (Kapitel 2.3), ist so klein, dass sich keine persönlichen Arbeitsräume auf der Tischfläche bilden. Dies bedeutet, dass die sozialen Verhaltensregeln Konflikte beim Zugriff auf einzelne Elemente des Diagramms nicht verhindern. Auch können Operationen, die das gesamte Diagramm oder die Arbeitsfläche betreffen, zu Konflikten führen. Zum Beispiel kann es vorkommen, dass ein Nutzer bei einem großen Diagramm (siehe A9F) die virtuelle Ansicht auf der Arbeitsfläche verschiebt, während ein anderer Nutzer noch in diesem Diagrammausschnitt ein Element bearbeitet. Diese Möglichkeiten von Konflikten führt zu einer weiteren Anforderung:

A13F Soweit technisch möglich muss der Diagrammeditor Konflikte zwischen verschiedenen Nutzereingaben erkennen und bei Erkennung aktiv zu der Lösung von diesen beitragen.

Kapitel 2.1.3 beschreibt die Problematik des Blickwinkels bei Tabletop PC Anwendungen. Dies führt zu folgenden Anforderungen:

- **A14F** Der Diagrammeditor muss Funktionen anbieten, um das gesamte Diagramm in 90° Schritten drehen zu können.
- **A15F** Einzelne Knoten sowie Beschriftungen im Allgemeinen muss der Editor auf Wunsch des Benutzers temporär drehen.

3.3 Nicht funktionale Anforderungen

Um für Tätigkeiten wie längerfristige Planung von Unternehmensprozessen geeignet zu sein, sollte die fertige Realisierung für den täglichen Gebrauch gerüstet sein. Daher werden folgende Anforderungen bezüglich der Zuverlässigkeit aufgestellt:

- **A16NF** Die Fehlbedienung durch einen Nutzer darf nicht zu einem inkonsistenten Zustand der Anwendung oder der Daten führen.
- **A17NF** Das Abspeichern eines Diagramms sollte möglichst robust gestaltet werden, so dass auch bei hoher Systembelastung (zum Beispiel durch andere Anwendungen) die Daten stets konsistent gesichert werden können.

Um eine Alternative zu mächtigen Diagrammeditoren auf konventionellen PCs mit viel mehr Funktionen als der in dieser Arbeit entwickelte Editor zu bieten, sollte letzterer eine leichte Benutzbarkeit aufweisen. Diesbezüglich werden folgende Anforderungen gestellt:

- **A18NF** Aus der Sicht des Benutzers atomare Funktionen wie das Speichern des Diagramms dürfen maximal zwei Eingabeschritte benötigen. Als Eingabeschritt gilt dabei eine Geste (siehe Kapitel 2.2) oder die Bedienung eines virtuellen Knopfs in einem Menü.
- **A19NF** Abstrakte Gesten (siehe Kapitel 2.2.3) dürfen nur zum Aufruf von Funktionen eingesetzt werden, die in der Regel selten (weniger als einmal pro Minute) eingesetzt werden.
- **A20NF** Die Benutzer sollten im Schnitt in weniger als einer Stunde alle Funktionen der Anwendung sicher beherrschen. Dies ist so definiert, dass der Benutzer nach der Lernphase maximal 5 Sekunden überlegen darf, welche Eingaben nötig sind, um eine Funktion auszuführen.

Damit die Umsetzung von *A18NF* ihre Wirkung gut entfalten kann, darf der Benutzer nach erfolgten Eingabeschritten nicht lange auf eine Reaktion der Anwendung warten. Das führt zu folgender Anforderung an die Leistung und Effizienz der Anwendung:

A21NF Auf eine erfolgte Eingabe muss spätestens nach einer halben Sekunde eine Reaktion des Systems erfolgen. Bei Operationen deren Ausführung länger als diese Zeitspanne dauert, sind Warteanimationen einzusetzen.

Die Anwendung wird in wenigen Monaten von nur einem Entwickler (dem Autor dieser Bachelorarbeit) realisiert. Daher müssen Programmierschnittstellen nicht stabil sein und können sich während der Entwicklung verändern, da keine anderen Entwickler zur gleichen Zeit gegen sie entwickeln. Falls die Anwendung nach Abschluss dieser Arbeit weiterentwickelt wird, geschieht dies wahrscheinlich in weiteren zeitlich begrenzten Studienarbeiten oder als Open Source Projekt. Daher werden sich Entwickler oft nur kurz der Anwendung zuwenden, um sie zu warten oder Änderungen durchzuführen. Daher sollte die Anwendung eine leichte Wartbarkeit und Änderbarkeit aufweisen, was zu folgenden Anforderungen führt:

- **A22NF** Die Anwendung muss aus verschiedenen Komponenten bestehen, die einen hohen Zusammenhalt bei geringer Koppelung aufweisen. Die Klassen einer Komponente sollen alle einen LCOM4 Wert zwischen 1 und 2 aufweisen.
- **A23NF** Alle Methoden und Attribute einer Programmierschnittstelle müssen in Englisch dokumentiert werden. Bei den Methoden muss die Dokumentation Vorbedingungen, Nachbedingungen sowie alle Parameter umfassen.
- **A24NF** Methoden dürfen nur in Ausnahmefällen eine zyklomatische Zahl größer als 10 besitzen.

Ein Schwerpunkt der Arbeit ist es sich mit den Möglichkeiten eines Tabletop PCs zur Unterstützung kollaborativen Arbeitens auseinander zu setzen (siehe Kapitel 1.2). Die daraus entstehenden Anwendungsteile können auch für andere Problemstellungen als das Bearbeiten von Diagrammen interessant sein. Daher wird folgende Anforderung bezüglich der Portierbarkeit aufgestellt:

A25NF Die Nutzung von Tabletop PC Techniken sowie die reine Verwaltung von Graphen basierten Diagrammen muss möglichst auf verschiedene Anwendungskomponenten verteilt werden.

Bereits die Ergebnisse früher Planungen (zum Beispiel von Geschäftsprozessen) können vertraulich sein. Auch sind Tabletop PCs durchaus mit dem Internet verbunden und daher die Datensicherheit ein ernst zu nehmendes Thema. Allerdings schätzt der Autor der Arbeit das Sicherheitsrisiko so ein, dass es ausreicht dem Betriebssystem des Tabletop PCs sowie dem Anwender die Verantwortung für die Datensicherheit zu überlassen. Auch steuert ein Diagrammeditor keine Prozesse, welche die Sicherheit der Anwender oder ihres Eigentums beeinflussen. Daher werden keine Anforderungen bezüglich der Sicherheit aufgestellt.

Die Anwendung sollte auf jedem Tabletop PC System, auf der sie läuft, alle verfügbaren Systemressourcen wie Prozessorleistung oder Arbeitsspeicher bei Bedarf nutzen. Da es Tabletop PCs mit mehreren logischen Prozessorkernen gibt (siehe zum Beispiel Kapitel 2.3.1), wird folgende Anforderung an die Skalierbarkeit der Anwendung gestellt:

A26NF Die Anwendung muss Funktionalitäten parallel ausführen, wenn die Parallelisierung zu einer Effizienzsteigerung führt.

3.4 Technische Anforderungen

Für die Arbeit stehen sowohl ein Microsoft Pixelsense der ersten als auch der zweiten Generation zur Verfügung. Da das auf diesen laufende, auf dem .net Framework und der Sprache C# basierte Surface SDK schon einige für Tabletop PCs interessante Funktionalitäten realisiert (siehe Kapitel 2.3.2 und 2.3.5), verwendet die Anwendung dieses. Folgende zwei technische Anforderungen werden aufgestellt:

- **A27T** Die Anwendung läuft auf der Microsoft Pixelsense Plattform der ersten und zweiten Generation.
- **A28T** Die Anwendung verwendet das Microsoft Surface SDK und wird in der Programmiersprache C# sowie der Beschreibungssprache XAML realisiert.

Damit sich der Realisierungsteil der Arbeit in Grenzen hält, verwendet die Anwendung bezüglich der Verwaltung Graphen basierter Diagramme das NShape Framework (siehe Kapitel 2.4.2). Dieses bietet sich an, da es wie die Anwendung in C# geschrieben ist und auf dem .net Framework aufbaut. Außerdem kann es unter einer Open Source Lizenz völlig frei verwendet sowie modifiziert werden. Daher wird eine weitere Anforderungen aufgestellt:

A29T Die Anwendung baut auf dem NShape Framework auf.

4 Entwurf

Dieses Kapitel beschreibt den Entwurfsprozess des Diagrammeditors. Zunächst gibt das Konzept einen Überblick über die Ideen und Vorstellungen, die hinter dem Entwurf stecken. Anschließend folgt eine detaillierte Beschreibung der Benutzerschnittstelle. Darunter fällt eine Erläuterung über die umgesetzten Gesten, verwendeten Menüs und andere visuelle Kontrollelemente. Abschließend hält das Entwurfskapitel aus fachlicher wie technischer Sicht fest, welche Softwarekomponenten zum Umsetzen der Anforderungen realisiert werden müssen. Außerdem werden die Schnittstellen zwischen den Komponenten beschrieben, so dass ersichtlich ist, inwieweit alles miteinander zusammen hängt.

4.1 Konzept

Dieses Unterkapitel erklärt die Ideen, die dem danach folgendem detaillierten Anwendungsentwurf zu Grunde liegen:

- Jeder Benutzer hat seine eigene **Spielfigur**, die mit einem im Anwendungskontext eindeutigen Tag (siehe Kapitel 2.3.5) versehen ist. Sie ermöglicht dem Editor in bestimmten Situationen eine eindeutige Identifikation des Nutzers. Mit der Spielfigur können Benutzer dem Editor gegenüber betonen, was ihr aktueller Eingabefokus auf dem Tisch ist. Auch kann die Spielfigur dem Editor helfen zur Lösung kritischer Konflikte Abstimmungen unter den Nutzern abzuhalten.
- Für ein flüssiges und individuelles Arbeiten erlaubt der Editor beliebig viele frei platzierbarer Ablageflächen (nach A12F) für Diagrammelemente und virtueller Werkzeuge, die Funktionen mit eingeschränkter lokaler Auswirkung (zum Beispiel das Löschen eines Elements) haben. Funktionen mit globaler Wirkung dagegen können aus einer **Singleton Menübox** heraus ausgerufen werden. Diese gibt es nur einmal. Es ist denkbar, dass sie generell deaktiviert ist und nur mit einer bestimmten Spielfigur aktiviert werden kann. Diese würde dann zu einem Nutzer in der Rolle eines Moderators gehören.
- Um der Problematik mit dem Blickwinkel (Kapitel 3.2.3) zu begegnen, nutzt der Editor, dass der Microsoft Pixelsense die Ausrichtung der Fingerspitze auf dem Bildschirm beeinflussen kann. Wenn der Benutzer einen Finger auf einen Knoten oder Textfeld legt, findet eine **temporäre Ausrichtung nach dem Finger** statt, wie es auch *A15F* einfordert. Nach Abheben des Fingers nimmt das Element dann wieder in die Ursprungsausrichtung ein. Da von dem Finger verdeckte Fläche natürlich für den Nutzer nicht sichtbar ist, wird das Element neben der Ausrichtung leicht verschoben. Abbildung 4.1 stellt den Vorgang dar.

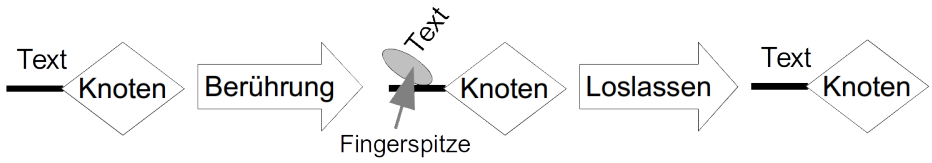

Abbildung 4.1: Darstellung der automatischen Ausrichtung nach dem Finger

- Bei jeder Funktion, welche die Arbeit anderer beeinflussen könnte, ist es nicht immer nötig gleich den Einsatz einer Spielfigur zu verlangen. Zum Beispiel bei Gesten wie dem Verschieben des angezeigten Bildausschnitts (nach A9F), kann der Editor auch mit dem Abstand von erfassten Fingerspitzen und ihrem Bewegungsmuster arbeiten. So ist dann das Verschieben der Arbeitsfläche zum Beispiel nur möglich, wenn zwei Fingerspitzen außerhalb von Elementen und in großem Abstand zueinander, in die gleiche Richtung bewegt werden. Wenn dies so gestaltet wird, dass ein Benutzer es nur unter Einsatz beider Arme durchführen kann, ist seine Aktion deutlicher anderen Personen am Tisch ersichtlich. Der Editor unterstützt also die Anwendung der sozialen Verhaltensregeln durch das Einfordern **großer Gesten** für globale Funktionen.
- Kapitel 2.3.4 hat aufgezeigt, dass der Einsatz einer Tastatur die derzeit praktikabelste Lösung zur Texteingabe auf dem Microsoft Pixelsense Tabletop PC ist. Um die negativen Effekte der indirekten Eingabe (Kapitel 2.2.1) zu verringern, setzt der Editor auf **Textsnippets**. Dabei handelt es sich um frei bewegbare Textbehälter, die irgendwo auf dem Bildschirm unter Verwendung der Tastatur erstellt und dann über direkte Eingabe in Textplatzhalter an einer anderen Stelle des Diagramms gezogen werden können. Die Platzhalter

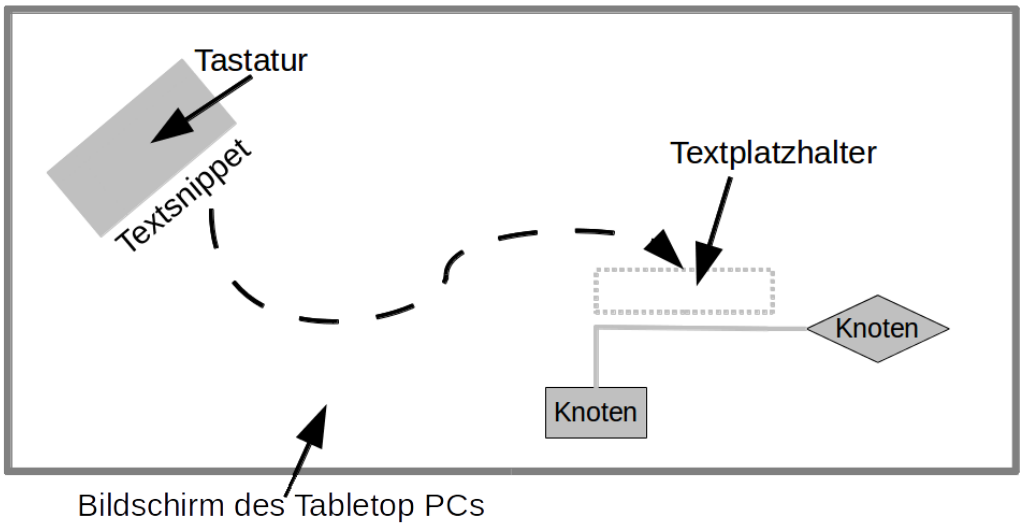

Abbildung 4.2: Funktionalität der Textsnippets

übernehmen dann den Text. Abbildung 4.2 zeigt den Einsatz der Textsnippets an einem einfachem Beispiel. Es veranschaulicht, dass die eigentliche Texterstellung nahe der Tastatur statt finden kann. Entsprechend befindet sich der Textcursor nahe der Tastatur und die Effekte der indirekten Eingabe werden

lokal beschränkt. Sobald ein Snippet erstellt ist, kann ein Nutzer es an seinen Bestimmungsort bewegen. Dies muss natürlich nicht die Person sein, welche die Tastatur bedient. Textsnippets ermöglichen also auch Arbeitsteilung. Ein Nachteil des Verfahrens ist, dass Änderungen an platzierten Textpassagen aufwendiger sind als mit frei bewegbarem Textcursor, da ein neues Textsnippet mit den Änderungen erstellt und erneut platziert werden muss.

- Der Editor unterstützt den Zugriff auf die kleinen Verbindungspunkte und Linien der Diagramme (siehe Kapitel 3.2.1) wie folgt. Sobald eine neue Linie von einem Knoten zu einem anderen Knoten erstellt werden soll, expandiert zunächst der Startknoten. Mit **Expansion eines Knoten** ist gemeint, dass seine Verbindungspunkte vergrößert neben ihm dargestellt werden. Dabei zeigen Linien von den Vergrößerungen zum Knotenrand die eigentlichen Positionen der Verbindungspunkte an (siehe Abbildung 4.3). Dies erlaubt eine zielsichere Auswahl eines dieser. Im nächsten Schritt zieht der Nutzer seinen Finger zum Zielknoten und wählt auch für diesen über Expansion den Verbindungspunkt aus. Daraufhin schlägt der Editor einen Linienzug dem Nutzer vor (**Autorouting**) und erstellt die Verbindung entsprechend.

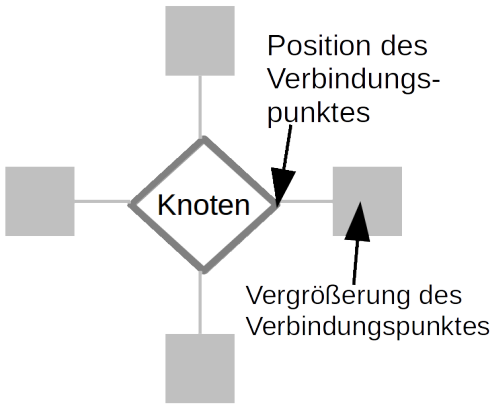

Abbildung 4.3: Expansion der Verbindungspunkte

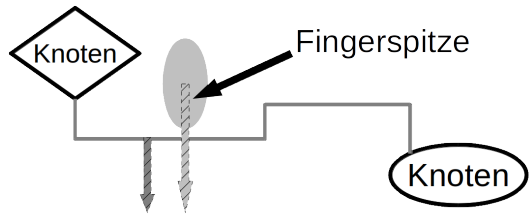

Abbildung 4.4: Ziehgeste über Gerade

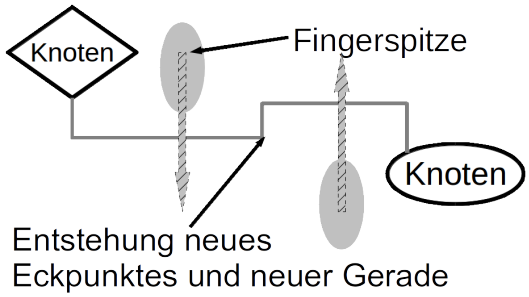

Abbildung 4.5: Erstellung einer Ecke

- Da der Nutzer eventuell mit dem Ergebnis des Autoroutings nicht zufrieden ist, ermöglicht der Editor auch manuelle Anpassungen an der Linienführung der Verbindungen. Dies erfolgt durch Bewegen der Eckpunkte, in denen zwei Geraden im rechten Winkel aufeinander treffen, durch eine **Ziehgeste über die Geraden**. Dabei bewegt der Nutzer eine Fingerspitze von der einen Seite dieser auf die andere. Dadurch verfängt sich die Spitze des Fingers an der Geraden und zieht diese sowie die beiden angrenzenden Eckpunkte mit sich. (siehe Abbildung 4.4) Die Geraden, von denen ein Endpunkt ein Verbindungspunkt ist, können nicht verschoben werden. Ein Nutzer kann einen neuen Eckpunkt mit neuen Geraden erstellen, indem er zwei Finger von beiden Seiten aus über eine bestehende Gerade zieht. (siehe Abbildung 4.5)

- Verschiedene Fingerspitzen, die sich zur gleichen Zeit auf dem Bildschirm befinden (siehe Kapitel 2.2.5), werden jeweils einem Kontext zugeordnet. Dabei bilden Diagrammhintergrund, jeder Knoten, jede Gerade, jedes Textfeld, jede Ablagebox, jedes Menü jeweils einen **Gestenkontext**. Befinden sich mehrere Fingerspitzen in einem Kontext und sind mehr als drei jeweils unnatürlich zueinander ausgerichtet, geht der Editor davon aus, dass mehrere Benutzer gleichzeitig auf dem Kontext arbeiten wollen und ein Konflikt vorliegt. Der Editor wendet daraufhin vom Kontext abhängig eine Richtlinie (siehe Kapitel 2.1.2) zum Lösen an.

4.2 Benutzerschnittstelle

Dieses Unterkapitel dokumentiert die Eigenschaften der Benutzerschnittstelle. Dazu gehören die eingesetzten Gesten, virtuelle Boxen für die Menüs sowie Ressourcen, die visuellen Darstellungen von Knoten sowie Verbindungen und schließlich Werkzeuge zur Beschriftung.

Wenn nicht anders angegeben, geht die Arbeit von dem kartesischen Koordinatensystem und einem zweidimensionalen euklidischen Vektorraum für die Positionsangaben virtueller Objekte aus.

4.2.1 Oft verwendete Gesten

Folgend beschriebene Gesten dienen zum Auslösen oft verwendeter Operationen. Die Gesten können auf verschiedenen Kontexten (zum Beispiel einer bestimmten Box oder einem Knoten) ausgeführt werden. Die Anwendung wendet die verknüpfte Operation dann auf dem entsprechenden Kontext an. Es folgt eine Beschreibung der Gesten:

- Die **Ziehgeste** dient dem Verschieben eines Objekts. Dabei legt der Nutzer mindestens eine Fingerspitze auf das Objekt und bewegt erstere zur Zielposition. Dabei verschiebt der Editor das Objekt analog zur Fingerspitze (siehe Abbildung 4.6).

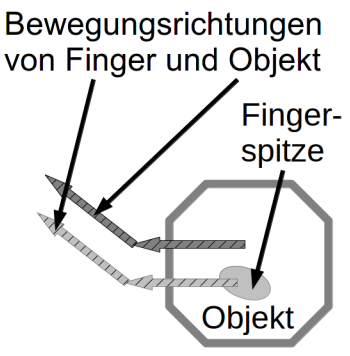

Abbildung 4.6: Ziehgeste

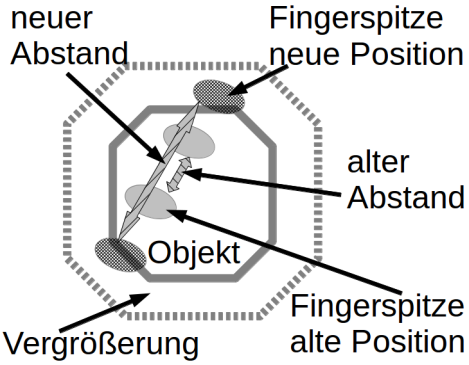

Abbildung 4.7: Zoomgeste

- Die **Zoomgeste** dient zum Vergrößern beziehungsweise Verkleinern eines Objekts. Hierbei legt der Benutzer genau zwei Fingerspitzen auf das Objekt und zieht diese auseinander oder zusammen (siehe Abbildung 4.7). Der genaue Effekt der Geste auf das Objekt hängt davon ab, ob das Objekt ein festes Seitenverhältnis besitzt oder nicht:

- Bei einem festen Seitenverhältnis wird das Objekt relativ zur Distanz der beiden Fingerspitzen vergrößert oder verkleinert.
- Besitzt das Objekt kein festes Seitenverhältnis, entscheidet der Winkel α zwischen der Grundlinie des Objekts und dem Vektor zwischen den beiden Berührungspunkten, ob das Objekt in seiner Breite oder in seiner Höhe relativ zur Vektorlänge verkleinert beziehungsweise vergrößert wird. (siehe Abbildung 4.8) Tabelle 4.1 zeigt welche Winkelgrößen, welchen Effekt auf die Skalierung der Größe des Objekts haben.

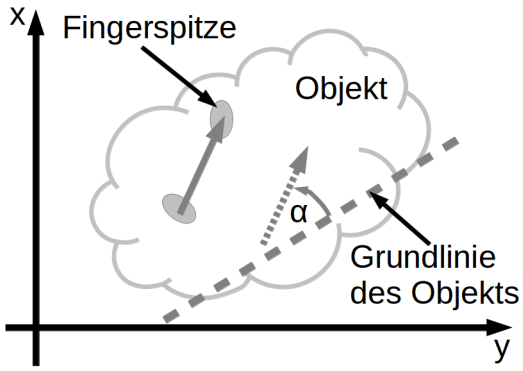

Abbildung 4.8: Zoomgeste ohne festes Seitenverhältnis

Winkel α	Skalierung der
(75° bis 105°) und (255° bis 285°)	Höhe
(345° bis 15°) und (165° bis 195°)	Breite
Sonstige Werte	Höhe und Breite

Tabelle 4.1: Auswirkung des Winkels (siehe Abbildung 4.8) bei Zoomgeste

Ein virtuelles Objekt kann zwei **Griffpunkte** aufweisen. Diese kann der Nutzer jeweils mit der Ziehgeste verschieben. Wird die Position von nur einem Griffpunkt verändert, hat es auch den Effekt der Ziehgeste auf das ganze Objekt. Positionsänderungen beider Griffpunkte bewirken jedoch zusätzlich, dass das entsprechende Objekt so in seiner Größe verändert sowie gedreht wird, dass sich stets beide Griffpunkte unter den Fingerspitzen befinden.

- Bei der **Drehgeste** legt der Benutzer mindestens zwei Fingerspitzen auf das Objekt und führt eine Drehbewegung mit diesen durch. Entscheidend für den Effekt der Geste sind die beiden Fingerspitzen, welche am weitesten voneinander entfernt sind. Der Drehwinkel gleicht dem Winkel zwischen ehemaligen und aktuellem Verbindungsvektor der Fingerspitzen. Dabei ist der Mittelpunkt der Drehung der Schnittpunkt der Strecken zwischen den aktuellen und ehemaligen Positionen der Fingerspitzen. Abbildung 4.9 skizziert die Drehgeste.

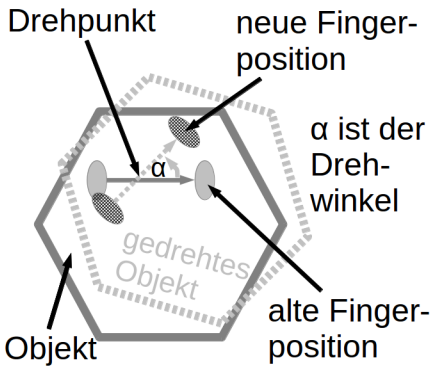

Abbildung 4.9: Drehgeste

- Der **einfache Klick** dient zur Auswahl von Objekten. Hierbei legt der Benutzer mindestens eine Fingerspitze für weniger als eine Viertelsekunde auf das Objekt. Auf dem Bildschirm dürfen sich die Fingerspitzen nicht mehr als einen halben Zentimeter bewegen, da sonst die Geste als Ziehgeste interpretiert wird.
- Das **lange Drücken** dient dazu, um Objekt bezogene Einstellungen aufzurufen. Auch führt die Geste oft zu einem Zustandswechsel des betroffenen Objekts. Um die Geste auszuführen legt der Benutzer mindestens eine Fingerspitze auf das Objekt und lässt sie dort für mehr als zwei Sekunden. Wie beim einfachen Klick dürfen die Fingerspitzen maximal einen halben Zentimeter bewegt werden.
- Der **Doppelklick** ist eigentlich nach der Definition aus Kapitel 2.2 keine Geste, sondern zwei innerhalb einer Viertelsekunde auf demselben Objekt ausgeführte einfache Klicks. Der Doppelklick dient zur Betätigung von virtuellen Schaltern, deren Betätigung eine destruktive oder globale Wirkung haben. Dort dient er als Ersatz für den einfachen Klick, welcher eher versehentlich ausgeführt wird.

4.2.2 Boxen

Boxen enthalten entweder Ressourcen wie Werkzeuge und Diagrammelemente oder Menüelemente wie virtuelle Knöpfe und Auswahllisten. Abbildung 4.10 zeigt den allgemeinen Aufbau einer Box. Jede Box besitzt zwei Griffpunkte (oben links sowie unten rechts), mit der sie wie in Kapitel 4.2.1 beschrieben verschoben, gedreht und vergrößert werden kann. Desweiteren kann die Größe der Box durch das Anwenden der Zoomgeste auf der Titelleiste oder der Scroll- und Zoomleiste verändert werden. Dabei hat die Titelleiste Einfluss auf die Breite der Box, während die andere Leiste die Höhe beeinflusst.

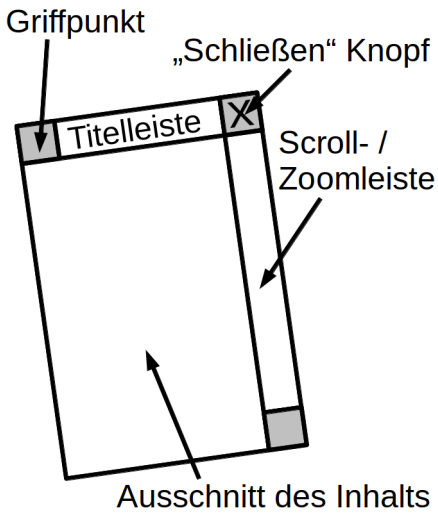

Abbildung 4.10: Allgemeiner Aufbau einer Box

Eine Box kann mehr Elemente enthalten, als in der Inhaltsfläche angezeigt werden können. In diesem Fall wird nur eine Auswahl an Elementen angezeigt. Die Auswahl kann über eine Ziehgeste auf der Zoom- / Scrolleiste verändert werden.

Über einen Doppelklick auf den Schließen Knopf wird die Box mit all ihrem Inhalt vom Editor verworfen. Damit verschwindet die Box auch von der Bildfläche. Nicht jede Box lässt sich entfernen.

Die Titelleiste übernimmt außer dem Anzeigen des Titels der Box noch weitere Aufgaben. So kann der Nutzer über einen einfachen Klick auf die Leiste die Box minimieren. Dabei wird die Größe so stark verkleinert, dass noch gerade der Titel und seine Umrandung angezeigt werden können. Über die Titelleiste lässt sich die Box weiterhin über die Ziehgeste bewegen und über die Drehgeste ausrichten. Der Editor merkt sich die Größe, so dass er nach erneutem einfachen Klick auf die Titelleiste die Box an der aktuellen Position wieder komplett anzeigen kann.

Boxen können gesperrt sein. In diesem Fall muss eine bestimmte Spielfigur (siehe Kapitel 4.1) auf die Box gesetzt werden, um den Inhalt zugänglich zu machen. Auch bewirkt eine Spielfigur auf der Box, dass entnommene Ressourcen einen Hinweis auf die Spielfigur erhalten.

Boxen befinden sich immer im Vordergrund und damit über allen anderen virtuellen Elementen. Daraus folgt, dass Boxen sich nicht überlappen dürfen. Auch darf eine Box nicht über den Bildschirmrand hinaus ragen. Dies stellt der Editor bei Verschiebung, Drehung und Größenänderung der Boxen sicher. Fehlt bei dem Wiederherstellen einer minimierten Box der Platz für die ursprüngliche Größe, so nimmt der Editor die mögliche Größe. Reicht der freie Platz nicht für die Minimalgröße der Box, so bleibt diese minimiert.

Die Minimalgröße der Box ist so definiert, dass sie so breit ist, dass der Titel voll angezeigt werden kann. Außerdem beträgt die Breite wie auch die Höhe mindestens vier Zentimeter.

Im folgenden Teil des Kapitels werden kurz die verschiedenen Arten der Box konkretisiert:

- Eine **Ressourcenbox** enthält Repräsentanten der verfügbaren Arten an Diagrammelementen wie unter anderem Knoten und Verbindungslinien (siehe Abbildung 4.11) und setzt somit A12F um. Dabei wird jeder Repräsentant durch ein Bild dargestellt, welches quadratisch und vier Quadratzentimeter groß ist. Neben dem Bild zeigt die Ressourcenbox an, wie oft sie den Repräsentanten einer bestimmten Art enthält. Die Anzahl kann sowohl begrenzt als auch unbegrenzt sein. Der Nutzer kann mit der Ziehgeste einen Repräsentanten aus der Dialogbox auf die Arbeitsfläche ziehen. Unter eventuell weiteren Voraussetzungen, wird dann auf der Arbeitsfläche ein neues dem Repräsentanten entsprechendes Element erstellt. Nach erfolgreicher Erstellung wird die Anzahl begrenzter Repräsentanten um eins reduziert.

  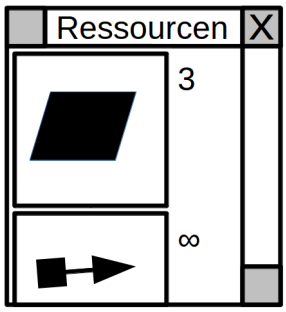

  Abbildung 4.11: Ressourcenbox

  Es können auch einige Elemente von der Arbeitsfläche in die Ressourcenbox gezogen werden. In diesem Fall schaut der Editor, ob das Element dieselben Eigenschaften wie ein bereits vorhandener Repräsentant besitzt. Zu den verglichenen Eingenschaften gehören die Art des Elements (zum Beispiel rechteckiger Knoten), seine Größe und seine Drehung. Findet sich ein passender Repräsentant, so wird dessen Anzahl erhört. Ansonsten erstellt der Editor einen passenden Repräsentanten. Das entsprechende Element wird in beiden Fällen von der Arbeitsfläche entfernt.

- Eine **Löschbox** übernimmt die Funktion eines Mülleimers. Der Benutzer kann Elemente in diese Box ziehen, um sie zu löschen. Wurde das Element über eine Ressourcenbox erstellt, auf der eine Spielfigur stand, muss dieselbe Spielfigur auf der verwendeten Löschbox stehen. Ansonsten ignoriert der Editor das

Löschen und das Element verbleibt an der Stelle auf der Arbeitsfläche, wo es zu Beginn der Ziehgeste gewesen ist.

- Eine **Kopierbox** ist eine leichte Abwandlung der Ressourcenbox. Wenn ein Element auf diese Art von Box gezogen wird, verbleibt eine Kopie des Elements an der Position, an der es sich zu Beginn der Ziehgeste befand.
- Über die **Textbox** können die Nutzer des Editors neue Textsnippets (siehe Kapitel 4.1) erzeugen. Es gibt nur maximal eine Textbox, die entfernt werden kann. Sie enthält einen virtuellen Schalter, mit dem die virtuelle Tastatur angezeigt sowie versteckt werden kann. Beim Anzeigen wird die Tastatur nahe der Textbox positioniert. Zusätzlich enthält die Box ein Eingabefeld. Der Texteingabefokus liegt immer auf diesem. Schließlich bietet die Box einen virtuellen Knopf, mit welchem ein Benutzer auslösen kann, dass der Editor den Inhalt des Eingabefeldes in ein neues Textsnippet kopiert. Dieses wird nahe der Textbox erstellt.
- Die **initiale Box** ist eine sehr wichtige Box. Sie kann daher auch nicht entfernt werden. Es gibt sie nur einmal und sie wird beim Starten des Editors erstellt. Die Box enthält eine Liste an Menüelementen, die aus jeweils einer Zeile Text bestehen. Der Nutzer kann mit einem einfachen Klick auf eines der Menüelemente eine bestimmte Operation ausführen lassen. Konkret weist die Menübox folgende Elemente auf:
  - Neues, leeres Diagramm erstellen
  - Laden eines Diagramms
  - Diagramm speichern
  - Arbeitsfläche im Uhrzeigersinn drehen
  - Arbeitsfläche entgegen Uhrzeigersinn drehen
  - Löschbox erstellen
  - Kopierbox erstellen
  - leere Ressourcenbox erstellen
  - volle Ressourcenbox erstellen
  - Anwendung schließen
  - Textbox erstellen

  Die Drehung der Arbeitsfläche erfolgt in Schritten von 90°. Neue Boxen werden möglichst nahe der initialen Box erstellt. Eine volle Ressourcenbox enthält einen Repräsentanten mit unbegrenzter Anzahl für jede Art an Elementen, welche der Editor zur Verfügung stellt.

4.2.3 Knoten und Verbindungen

Die Erstellung sowie das Entfernen von Knoten erfolgt über die in Kapitel 4.2.2 beschriebenen Boxen. Ein Nutzer des Editors kann über eine Ziehgeste die Position, mit einer Zoomgeste die Größe und mit der Drehgeste die Ausrichtung eines Knotens ändern. Die verwendeten Gesten sind in Kapitel 4.2.1 beschrieben. Anders als Boxen können Knoten überlappen. Dabei ist immer der Knoten im Vordergrund, auf dem zuletzt eine Fingerspitze lag.

Drückt ein Nutzer lange auf einen Knoten, wird eine exakte Kopie von diesem erstellt. Diese kann der Nutzer sofort mit der Ziehgeste an eine andere Position auf der Arbeitsfläche bewegen.

Die Erstellung einer Verbindung startet immer, indem der Repräsentant eines Verbindungselements aus einer Ressourcenbox auf die Arbeitsfläche gezogen wird. Trifft der Repräsentant dabei auf einen zuvor erstellten Knoten, expandiert dieser wie in Kapitel 4.1 beschrieben. Der Anwender kann nun durch einen einfachen Klick auf den expandierten Knoten die Verbindungserstellung abbrechen oder die Ziehgeste von einem der Verbindungspunkte des Knotens auf die Arbeitsfläche ausführen. Dabei schlägt der Editor stets eine Streckenführung vom Verbindungspunkt zur Position der Fingerspitze vor. Bewegt der Nutzer des Editors die Fingerspitze auf einen anderen Knoten, kann er bei diesem den Verbindungspunkt durch Expansion des Knoten wählen.

Wird die Ziehgeste nicht auf einem Knoten beendet, entsteht an dem entsprechenden Ende der Verbindung ein **Dummyknoten**. Anders als normale Knoten kann dieser weder in seiner Größe, noch in seiner Ausrichtung geändert werden. Allerdings lässt er sich ziehen. Wenn dabei das Ziel ein gewöhnlicher Knoten ist, findet für das entsprechende Ende der Verbindungslinie durch Expansion von diesem eine Verbindungspunktwahl statt. Der Dummyknoten weist eine Größe von einem Quadratzentimeter auf. Die Dummyknoten kann ein Benutzer auf eine Löschbox, Ressourcen- oder Kopierbox ziehen. Dann wird die gesamte Verbindungslinie wie das Element einer Box behandelt.

Ein Anwender des Editors kann die Streckenführung einer Verbindungslinie mit den in Kapitel 4.1 beschriebenen Ziehgesten über Geraden verändern.

Das Lösen von Verbindungslinien geschieht über eine Geste, die eine leichte Modifikation zur Ziehgeste darstellt. Dabei wird der Finger mit einer bestimmten Mindestgeschwindigkeit gezogen (schnell ausgeführt). Wird die Ziehgeste schnell über eine an einem Verbindungspunkt hängenden Geraden einer Verbindungslinie ausgeführt, werden der entsprechende Knoten und die Verbindung voneinander getrennt. Führt ein Nutzer die Ziehgeste schnell auf einem Knoten aus, trennt er den Knoten von all seinen Verbindungen. An den getrennten Enden der Verbindungslinien entstehen Dummy-knoten.

4.2.4 Arbeitsfläche

Die Größe des Diagramm kann beim Erstellen eines neuen leeren Diagramms in einer virtuellen Einheit angegeben werden. Die Anzahl Bildpunkte die auf eine virtuelle Einheit fallen, wird durch den **globalen Skalierungsfaktor** bestimmt.

Wenn der Bildschirm nicht die gesamte Arbeitsfläche erfassen kann, zeigt er einen **Ausschnitt** dieser an (siehe Abbildung 4.12). Die Größe des Ausschnitts wird dabei vom globalen Skalierungsfaktor festgelegt. Die Dicke von Verbindungslinien wird A8F entsprechend vom globalen Skalierungsfaktor nicht beeinflusst.

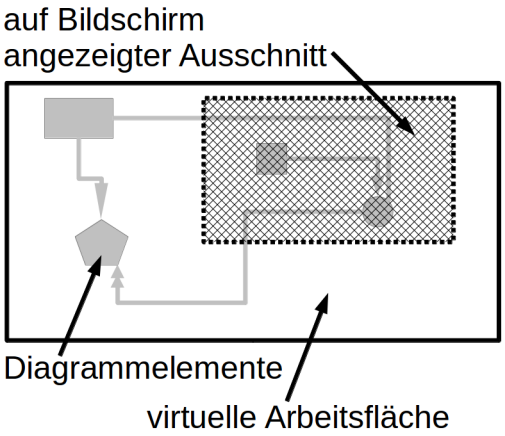

Abbildung 4.12: Virtuelle Arbeitsfläche größer als Bildschirm

Kapitel 4.1 beschreibt, dass der Ausschnitt nur mit einer großflächigen Geste auf der Arbeitsfläche verschoben werden kann. Das gilt auch für das Verändern des globalen Skalierungsfaktor.

Beide Gesten ähneln sich sehr stark und sind große Gesten (siehe Kapitel 4.1). Sie beginnen damit, dass ein oder mehrere Nutzer des Editors insgesamt zwei Fingerspitzen auf den Hintergrund der Arbeitsfläche legen und dann ziehen. Der Abstand zwischen den Fingerspitzen muss anfänglich mindestens 50 Zentimeter betragen. Außerdem darf sich keine andere Fingerspitze zeitgleich auf dem Bildschirm befinden, damit nicht versehentlich durch das Verändern des Bildausschnitts andere Nutzer gestört werden.

Abbildung 4.13 zeigt die große Geste zum Verschieben des Ausschnitts. Der Winkel β zwischen den Bewegungsvektoren beider Fingerspitzen darf höchstens 45° groß sein. Die schneller gezogene Fingerspitze bestimmt, in welche Richtung und wie weit der Ausschnitt auf der Arbeitsfläche gezogen wird. Der Editor stellt sicher, dass der Ausschnitt nicht die Arbeitsfläche verlässt.

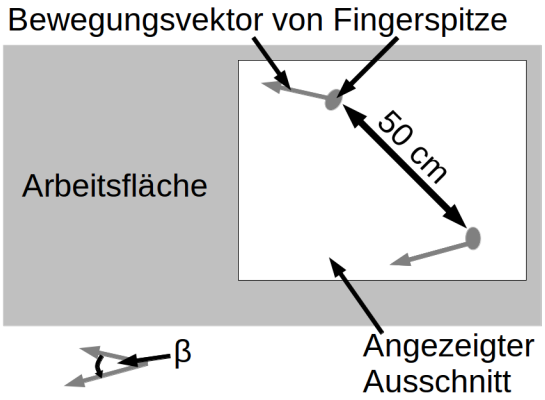

Abbildung 4.13: Geste zum Verschieben des angezeigten Ausschnitts

Bei der Geste zum Verändern des Skalierungsfaktors muss β zwischen 180° ± 15° liegen. Dann wird der Faktor relativ zur Distanz der Fingerspitzen verändert. Wie beim Verschieben der Arbeitsfläche stellt der Editor sicher, dass der Ausschnitt nicht aus der Arbeitsfläche heraus ragt. Dadurch ist der Skalierungsfaktor entsprechend begrenzt.

4.2.5 Text

Kapitel 4.1 beschreibt mit den Textsnippets und den Textplatzhaltern das Konzept, welches hinter den Textfunktionalitäten des Editors steht. In diesem Unterkapitel werden die Eigenschaften der Snippets und Platzhalter noch genauer beschrieben.

Beide Textelemente können mit der Zieh- und der Drehgeste (siehe Kapitel 4.2.1) bewegt sowie ausgerichtet werden. Sie können Knoten wie Verbindungen und sich gegenseitig beliebig überlappen. Wie bei den Knoten und Verbindungen (siehe Kapitel 4.2.3) wird das Element als oberstes angezeigt, welches zuletzt berührt worden ist.

Sowohl Snippets als auch Platzhalter haben ebenfalls mit den Knoten gemeinsam, dass sie sich über eine Kopierbox (siehe Kapitel 4.2.2) und eine Löschbox kopieren beziehungsweise löschen lassen.

Platzhalter kann ein Nutzer über entsprechende Repräsentanten in den Ressourcenboxen erstellen. Sie können vertikal und horizontal über die allgemeine Zoomgeste in ihrer Größe geändert werden. Die Größe, Position und Ausrichtung bestimmt die Anzeigefläche des Textes. Der Text wird dabei in einer bestimmten Schriftart und Größe dargestellt. Bei einem einfachen Klick auf einen Platzhalter wird in seiner Nähe eine bestimmte Menübox angezeigt, über welche der Nutzer die Texteigenschaften bestimmen kann.

Textsnippets kann ein Benutzer des Editors über die Textbox erstellen. Textsnippets dienen wie im Konzept (Kapitel 4.1) beschrieben dem Transport von Text zu einem Platzhalter. Wird ein Textsnippet so über die Ziehgeste gezogen, dass mindestens eine Fingerspitze über einem Platzhalter vom Bildschirm genommen wird, übernimmt dieser den Text vom Snippet. Maximal 64 Buchstaben des Texts eines Snippets werden an seiner aktuellen Position und mit seiner Ausrichtung in der Schriftart Arial bei einer Höhe von 12pt angezeigt.

4.3 Fachliche Architektur

Dieses Kapitel beschreibt den Aufbau des Diagrammeditors aus fachlicher Sicht. Der Editor besteht aus mehreren Komponenten, die über Schnittstellen miteinander verbunden sind. Dabei realisiert eine Komponente jeweils Aufgaben, die logisch zusammen hängen. Abbildung 4.14 gibt einen Überblick über fast alle Komponenten und den Schnittstellen zwischen ihnen.

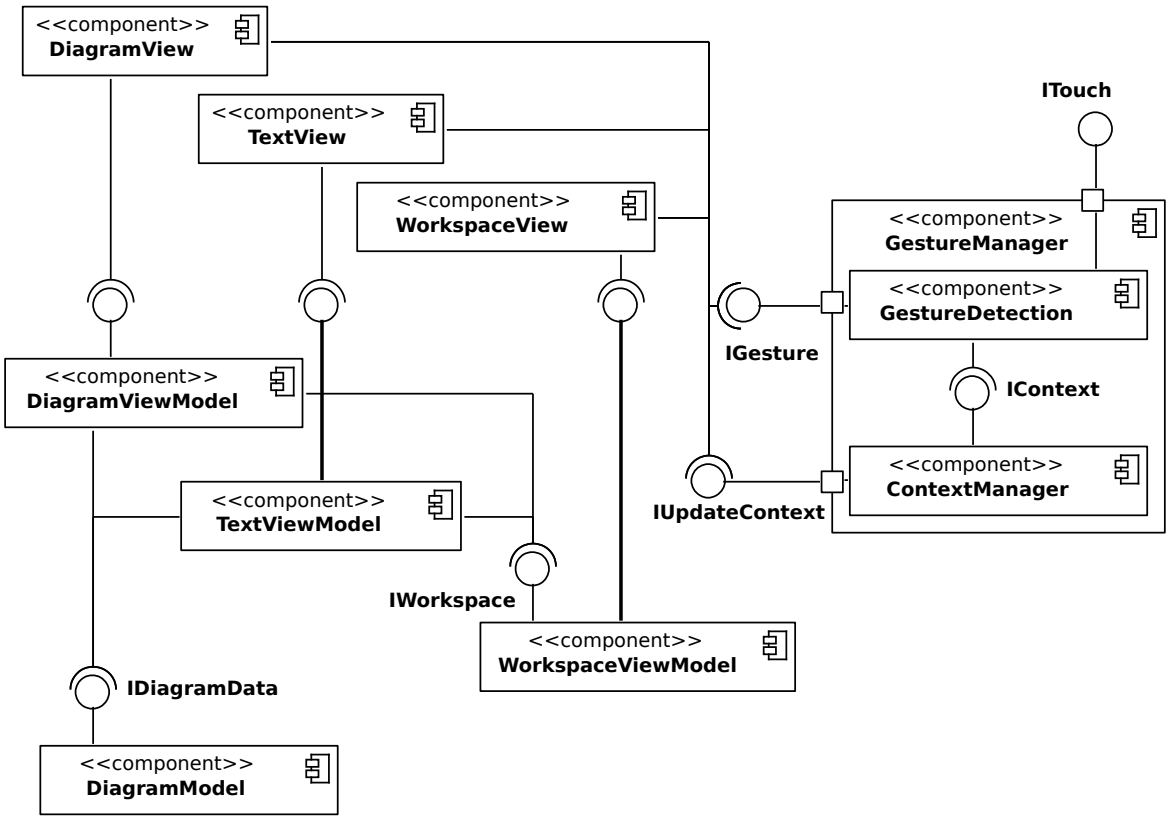

Abbildung 4.14: UML2 Komponentendiagramm der wesentlichen Anwendungsteile

Der Aufbau der Anwendung folgt dem in Kapitel 2.3.3 beschriebenen MVVM Pattern. Die Namensgebung einer Komponente erfolgt so, dass ersichtlich ist, zu welcher der drei Schichten des MVVM Pattern sie gehört. Eine Ausnahme bilden dabei die *Manager*. Diese Komponenten kapseln Funktionalitäten, die von vielen anderen Komponenten verschiedener Schichten verwendet werden beziehungsweise verwendet werden könnten.

In den folgenden Unterkapiteln werden die Komponenten, ihre Aufgaben und die Schnittstellen, welche sie anbieten, genauer beschrieben. Dabei werden teilweise UML2 Klassendiagramme zur Erklärung eingesetzt. Diese enthalten nur die wichtigsten, aber lange nicht alle Methoden und Attribute, welche die Komponenten in ihrer Realisierung aufweisen. Folgende allgemeine Datentypen finden in den Diagrammen Verwendung:

- *Integer* steht für eine Ganzzahl.
- *Float* repräsentiert eine Gleitkommazahl.

- *List* steht für den abstrakten Datentyp Liste.
- *String* steht für eine Zeichenkette.
- *Coord* ist ein Tupel, welches der Angabe von kartesischen Koordinaten dient.
- *Boolean* repräsentiert einen boolschen Wert („wahr“ oder „falsch“).
- *Rectangle* ist die Beschreibung eines beliebig gedrehten Rechtecks.
- *Matrix* repräsentiert eine 3 x 3 Matrix zum Transformieren zwei dimensionaler Koordinaten.

4.3.1 GestureManager

Die Komponenten *GestureManager* übernimmt zu großen Teilen die Aufgabe der Gestenerkennung. Dazu berechnet sie wie in Kapitel 2.2.5 beschrieben die Bewegungsvektoren der einzelnen Fingerspitzen, ordnet diese bestimmten Gestenkontexten (siehe Kapitel 4.1) zu, führt die Konflikterkennung durch und erkennt die oft verwendeten Gesten (siehe 4.2.1).

GestureManager besteht aus den Unterkomponenten *ContextManager* und *GestureDetection.*

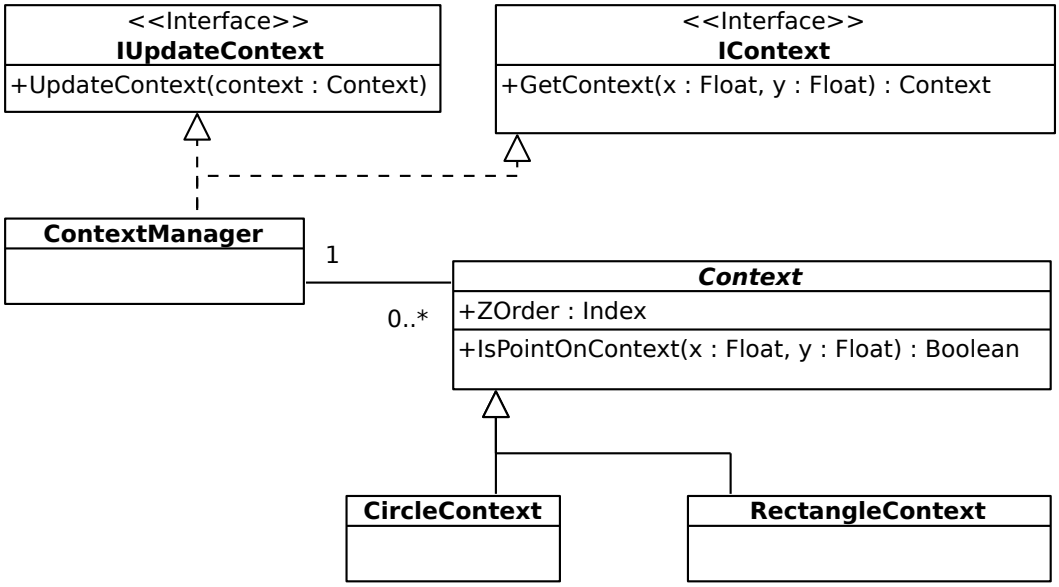

Abbildung 4.15: Aufbau des ContextManager

ContextManager verwaltet die Informationen zu den Gestenkontexten. Abbildung 4.15 zeigt seinen groben Aufbau. Für die Komponente *GestureManager* ist die Zuordnung von Koordinaten (einer Berührung des Bildschirms) zu einem Kontext wichtig. Diese erfragt sie bei *ContextManager* über die Methode *GetContext* der Schnittstelle *IContext.* Die Entscheidung, ob sich eine Koordinate in einem konkreten Gestenkontext befindet, überlässt *ContextManager* dessen Realisierung (Ableitung der Klasse *Context*) und fragt sie über die Methode *IsPointOnContext* ab. Mehrere Gestenkontexte können im zwei dimensionalen Raum überlappen. Um dennoch eine eindeutige Zuordnung von einer Koordinate auf einen Kontext zu erlauben, bestimmt der Wert des *ZOrder* Attributs, welcher Kontext über den anderen liegt und daher der Koordinate zugeordnet wird.

Konkrete Realisierungen der Klasse *Context* sind zum Beispiel *CircleContext* für Kreis förmige und *RectangleContext* für rechteckige Gestenkontexte.

Andere Komponenten können über die Methode *UpdateContext* der Schnittstelle *IUpdateContext* Gestenkontexte registrieren und bei Bedarf aktualisieren.

Die Komponente *GestureDetection (*siehe Abbildung 4.16) führt Gestenkontext bezogen die Konflikterkennung (siehe Kapitel 4.1) sowie die Erkennung der in Kapitel 4.2.1 beschriebenen Gesten durch und hilft anderen Komponenten bei der Gestenerkennung.

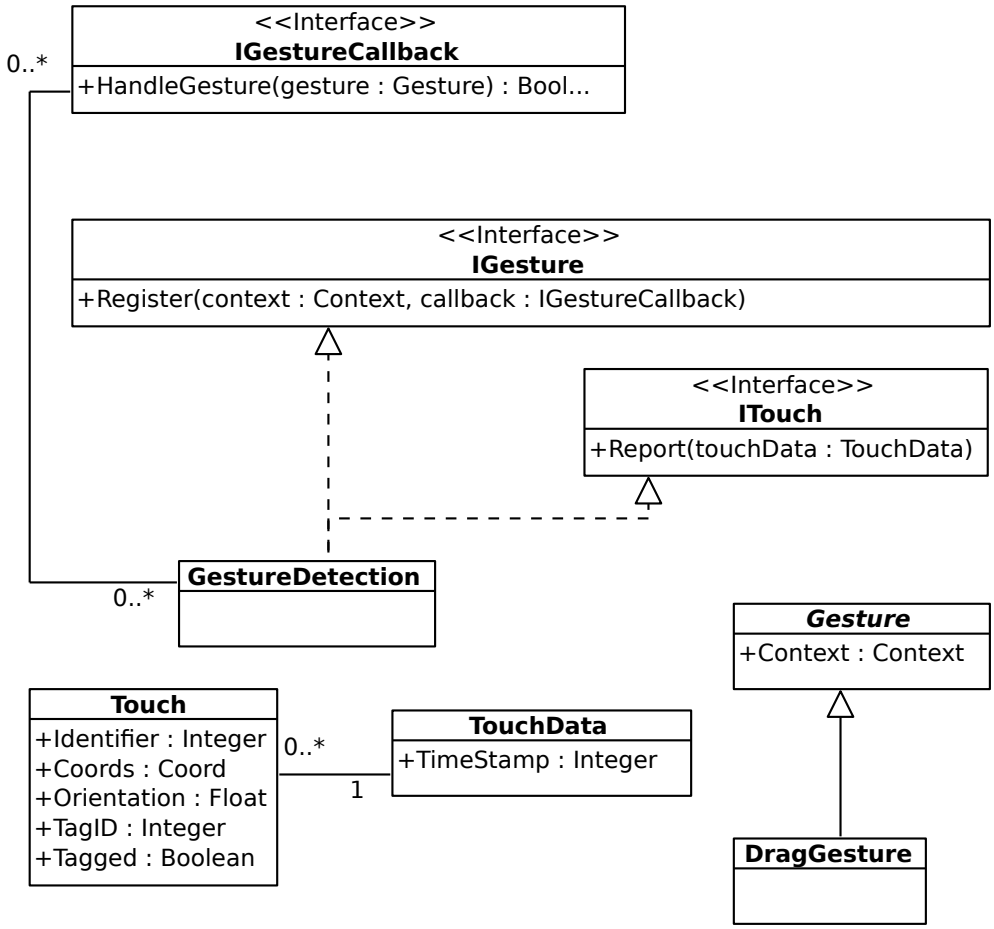

Abbildung 4.16: Aufbau der GestureDetection

Die Gestenerkennung erfolgt wie in Kapitel 2.2.5 beschrieben. Über die Methode *Report* der Schnittstelle *ITouch* erhält *GestureDetection* Schnappschüsse der Informationen über die Berührungen (repräsentiert durch *Touch*) des Bildschirms in Form von *TouchData* Objekten. Das Attribut *Identifier* von *Touch* ermöglicht der Gestenerkennung eine Berührung über mehrere Schnappschüsse hinweg zu verfolgen, um dann unter Verwendung der Koordinaten (Attribut *Coords*) die Bewegungsvektoren zu berechnen. Jeder Schnappschuss enthält einen Zeitstempel. Dies ermöglicht, dass die Ausführungsgeschwindigkeit von Gesten (z.B. in Kapitel 4.2.3 beschrieben) und die Doppelklickgeste ermittelt werden können, selbst wenn die Schnappschüsse in unregelmäßigen Zeitabständen kommen.

Für die Umsetzung der in Kapitel 4.1 beschriebenen Konflikterkennung ist es nötig, dass die Ausrichtung von Fingerspitzen bekannt ist. Dafür sieht die Klasse *Touch* das Attribut *Orientation* vor. Im selben Kapitel sind auch die Spielfiguren mit aufgeklebten Infrarot Tag beschrieben. Damit die Identifikation einer Spielfigur möglich ist, gibt das Attribut *Tagged* an, ob es sich bei einer Berührungen um eine Fingerspitze oder einen Tag handelt. In letzterem Fall dient das Attribut *TagID* der Identifikation eines Tags.

Die Komponente *GestureManager* dient nur der Erkennung von Gesten, das Ausführen der oft anwendungsspezifischen Aktionen übernehmen andere Komponenten. Diese sind über das Beobachterpattern [Gamm95] an *GestureManager* gekoppelt. Dabei meldet die Klientkomponente einen Beobachter, der die Schnittstelle *IGestureCallback* realisiert über die Methode *Register* der Schnittstelle *IGesture* für einen bestimmten Gestenkontext (Attribut *context*), an. Wird nun eine Geste auf diesem erkannt ruft die Komponente *GestureManager* die Methode *HandleGesture* der angemeldeten Beobachter auf und übergibt jeweils eine konkrete Instanz der Klasse *Gesture*, welche die Geste beschreibt.

Jede Instanz der Klasse *Gesture* enthält dabei den Gestenkontext (Attribut *Context*), damit eine Klientkomponente nicht für jeden beobachteten Kontext eine eigenen Beobachter bereitstellen muss, sondern auch einen Beobachter für mehrere Kontexte anmelden kann.

Eine konkrete Realisierung der Klasse *Gesture* ist zum Beispiel *DragGesture* zur Beschreibung einer Ziehgeste.

4.3.2 DiagramModel

Die Komponente *DiagramModel* verwaltet die Daten eines in Kapitel 3.2.1 beschriebenen auf Graphen basierten Diagramms und bietet anderen Komponenten Operationen auf diese Daten an. Abbildung 4.17 gibt einen groben Überblick über die wichtigsten Klassen.

Um Knoten wie in Kapitel 4.2.3 beschrieben zu unterstützen, sieht die abstrakte Klasse *Node* Attribute wie zum Beispiel *Position* für die Position des Knotens im Diagramm vor. Ein Knoten besitzt Verbindungspunkte, die in der Komponente *DiagramModel* durch Instanzen der Klasse *ConnectionPoint* repräsentiert werden. Der Wert des Attributs *RelativePosition* repräsentiert dabei die Koordinaten des Verbindungspunkt relativ zum entsprechenden Knoten. Wie in Kapitel 4.2.3 beschrieben verbindet eine auf Geraden basierte Verbindung (Klasse *Connection*) immer zwei Verbindungspunkte miteinander. Entsprechend referenziert *Connection* zwei *ConnectionPoint* Instanzen. Das Attribut *Corners* enthält dabei eine Liste der Eckpunkte zwischen den einzelnen Geraden, aus denen die Verbindungslinie aufgebaut ist. Klassen wie *ArrowConnection* konkretisieren *Connection*, um die Verbindungslinien zum Beispiel um Pfeilspitzen zu erweitern. Genauso erweitert *ImageBasedNode Node* zum Beispiel um Attribute, die Bildinformationen zu den Knoten enthalten.

Nach Anforderung *A6F* muss der Editor beliebige Daten zu Knoten verwalten können. Die Datensätze werden in Form von *Property* Objekten als generische Schlüssel (Attribut *Key*) - Daten (*Data*) Paare abgelegt. Ein Datensatz kann entweder einem Knoten oder einer Verbindung zugeordnet werden. Entsprechend referenzieren

Instanzen der Klassen *Node* oder *Connection* Instanzen von *Property*.

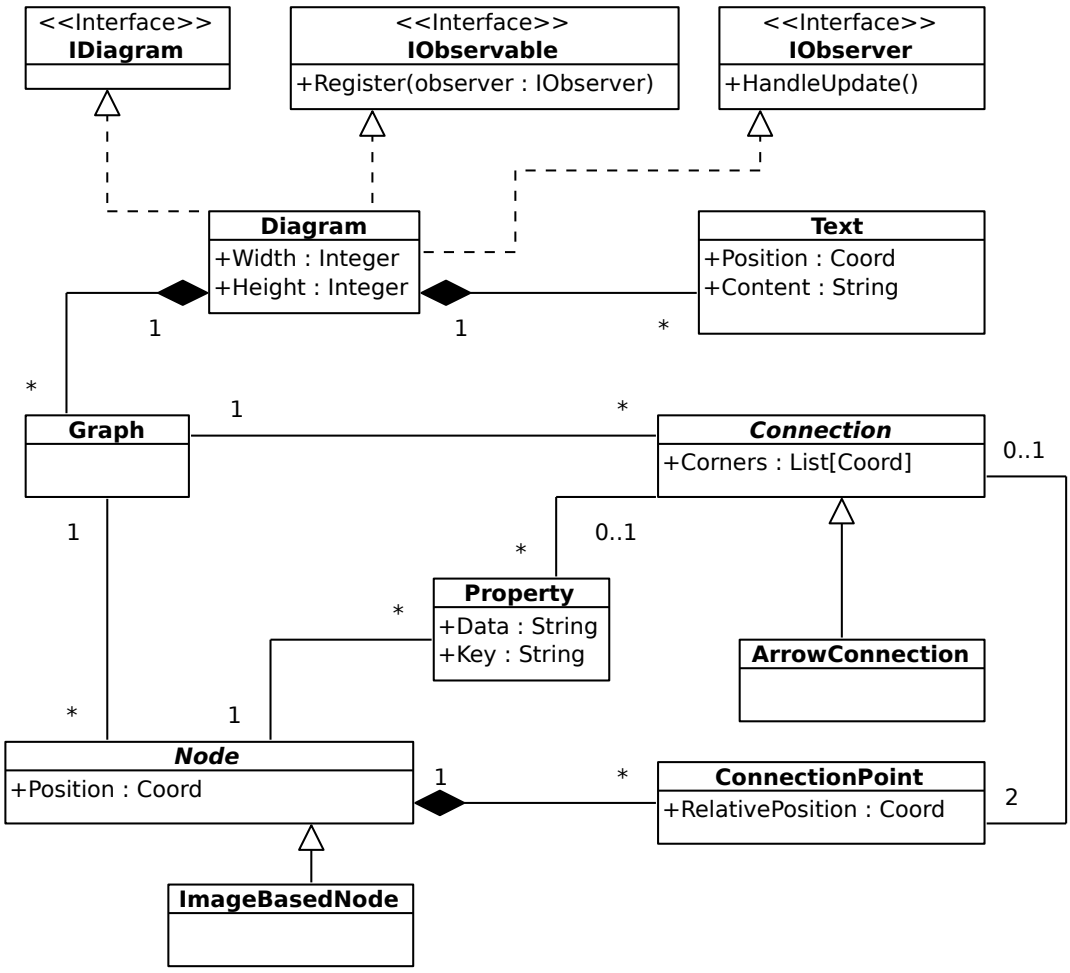

Abbildung 4.17: Aufbau der DiagramModel Komponente

Alle *Node* und *Connection* Objekte eines Graphen, werden von einem *Graph* Objekt referenziert, damit stets ein direkter Zugriff auf alle Knoten sowie Verbindungen möglich ist.

Wie in Kapitel 4.2.5 beschrieben enthalten Diagramme Text, der frei positioniert werden kann. Das *DiagramModel* verwaltet diesen Text über *Text* Objekte, welche über ihre Attribute unter anderem Textinhalt (Attribut *Content*) sowie die Koordinaten (*Position)* einzelner Textobjekte speichern.

Die Klasse *Diagram* stellt schließlich über die Schnittstelle *IDiagram* Operationen auf das referenzierte *Graph*objekt sowie die Textobjekte zur Verfügung. Um andere Komponenten bei Datenänderungen zu informieren, kommt das bereits in Kapitel 2.3.3 beschriebene Beobachterpattern zum Einsatz. Der Beobachter (eine Instanz von *IObserver*) wird über die Methode *Register* der Schnittstelle *IObservable* registriert.

4.3.3 WorkspaceViewModel

Aufgabe der *WorkspaceViewModel* Komponente ist es die Logik hinter den Boxen sowie die virtuelle Arbeitsfläche, welche aus Sicht der Nutzer des Editors in Kapitel 4.2.2 und Kapitel 4.2.4 beschrieben werden, bereitzustellen. Die in den Kapiteln 4.2.3 und 4.2.5 erläuterten Knoten, Textsnippets oder Textcontainer stellen dabei virtuelle Elemente auf der virtuellen Arbeitsfläche dar. Abbildung 4.18 zeigt die wichtigsten Klassen, Methoden und Attribute der Komponente.

Es gibt pro Instanz des Diagrammeditors nur eine Arbeitsfläche, an der alle Nutzer gemeinsam dasselbe Diagramm bearbeiten. Auf fachlicher Ebene wird dieser Umstand durch den Einsatz des Singleton Pattern [Gamm95] durchgesetzt. Die statische Methode *GetDefaultWorkspace* der Klasse *WorkspaceViewModel* gibt dabei immer dieselbe Instanz von *Workspace* (der Arbeitsfläche) zurück.

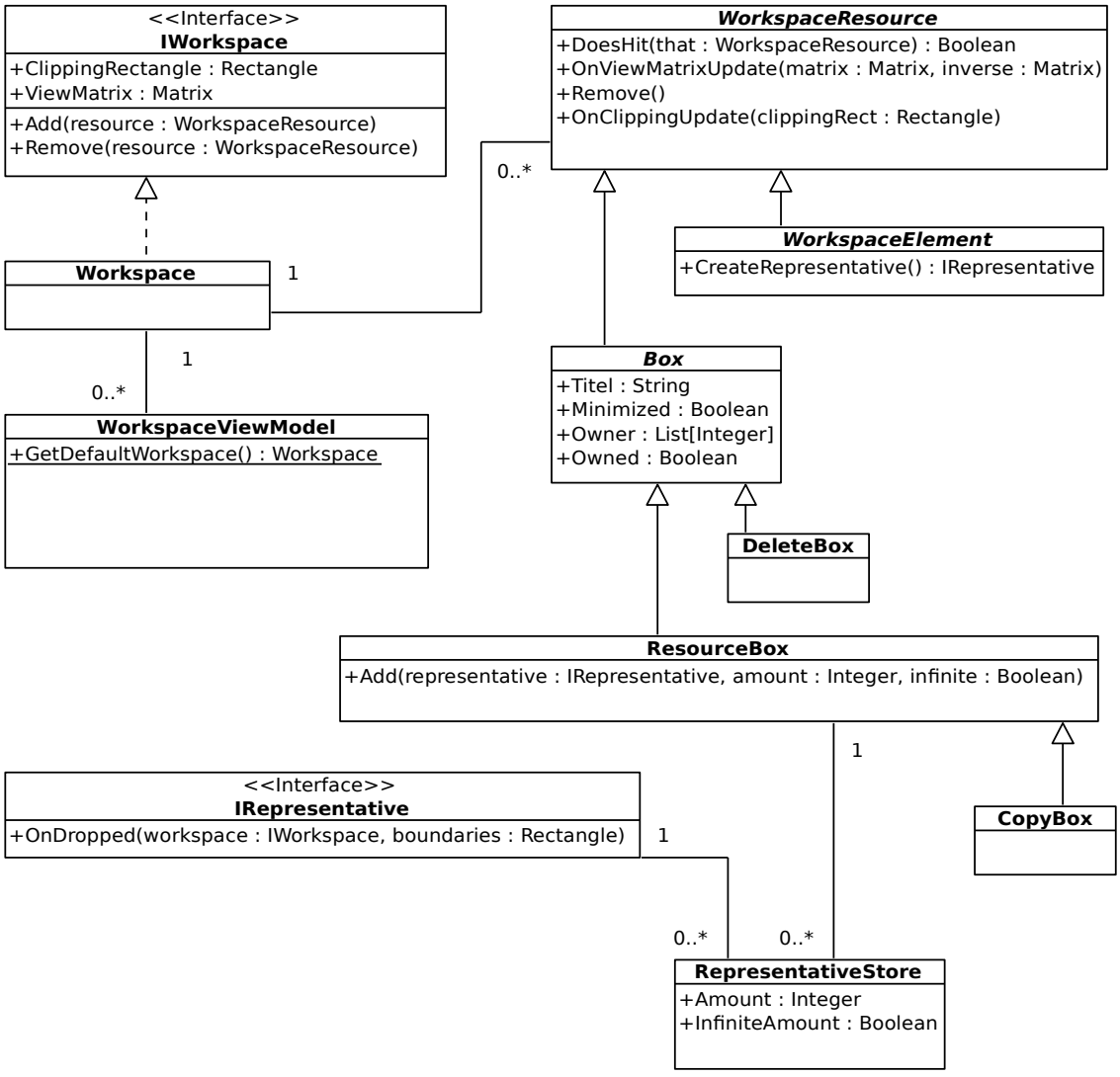

Abbildung 4.18: WorkspaceViewModel Komponente

Nach Anforderung *A25NF* kapselt WorkspaceViewModel die Tabletop PC spezifische Logik und weiß nichts über Graphen beziehungsweise Text spezifische Attribute sowie Methoden der verwalteten virtuellen Elemente. Die Klasse *WorkspaceResource* setzt das Verhalten um, welche alle virtuellen Elemente aufweisen. Da eine Aufgabe der Komponente *WorkspaceViewModel* die Verwaltung der Elemente ist, referenziert *Workspace* beliebig viele Instanzen von *WorkspaceResource*. Virtuelle Elemente können einer Arbeitsfläche über die Methode *Add* der Schnittstelle *IWorkspace* hinzugefügt und über die *Remove* Methode entfernt werden. Beim Aufruf von *Remove* einer *WorkspaceResource* Instanz, kann diese entscheiden, ob das entsprechende virtuelle Element entfernt werden soll oder nicht. Dies benötigt zum Beispiel die Umsetzung der Singleton Menübox aus Kapitel 4.1.

Die Anforderungen *A8F*, *A9F*, *A14F* können alle zusammen erfüllt werden, indem der Bildschirm einen in seiner Größe skalierbaren, relativ zum Diagrammursprung verschiebbaren sowie drehbaren Ausschnitt (Clipping Rechteck) des Diagramms anzeigt. Dadurch gibt es zum einen den Diagrammkoordinatenraum, mit welchem unter anderem das *DiagramModel* (Kapitel 4.3.2) arbeitet und den Bildschirm-koordinatenraum, mit welchem die *View* Komponenten arbeiten. Die Transformationen von *Bildschirm*koordinaten in *Diagramm*koordinaten und in die andere Richtung erfolgt durch lineare Algebra $\vec{v}_{Bild} = M_{Bild} \cdot \vec{v}_{Diagramm}$; $\vec{v}_{Diagramm} = M_{Inverse} \cdot \vec{v}_{Bild}$. M bedeutet dabei Matrix. Um anderen Komponenten die Durchführung dieser Umrechnungen zu erlauben, propagiert *WorkspaceViewModel* die Informationen zum virtuellen Bildausschnitt auf mehrere Arten. Das Attribut *ClippingRectangle* von *IWorkspace* beschreibt das Clipping Rechteck und *ViewMatrix* entspricht M_{Bild}. Darüber hinaus teilt bei entsprechenden Änderungen *Workspace* allen virtuellen Elementen die neuen Matrizen und das neue Clipping Rechteck über Aufruf der Methoden *OnClippingUpdate* und *OnViewMatrixUpdate* der Klasse *WorkspaceResource* mit.

Die Methode *DoesHit* von *WorkspaceResource* überprüft, ob sich zwei Elemente überlappen. Sie ermöglicht somit die Prüfung, ob ein Element von einem Nutzer auf eine Box gezogen worden ist.

Eine Konkretisierung von *WorkspaceResource* ist die Klasse *Box*, die eine allgemeine Box umsetzt. Nach Kapitel 4.2.2 besitzt jede Box einen Titel (Attribut *Titel*) und einen minimierten Zustand, was durch das Attribut *Minimized* festgelegt wird. Des Weiteren kann eine Box Besitzer haben. Diesen Fall drückt, dass Attribut *Owned* aus. Dabei gibt das Attribut *Owner* an, wer die Besitzer sind.

Neben der Löschbox (repräsentiert durch *DeleteBox*) ist eine weitere Konkretisierung einer Box die Ressourcenbox (*ResourceBox*), welche Kapitel 4.2.2 aus Benutzersicht beschreibt und die eine Lagerfläche nach A12F darstellt. Aus Benutzersicht muss eine Ressourcenbox virtuelle Elemente aufnehmen und dabei gleiche virtuelle Elemente gruppieren. Um Speicherplatz zu sparen, nimmt eine Ressourcenbox nicht direkt die virtuellen Elemente auf. Stattdessen speichert sie lediglich pro Elementgruppe einen Repräsentanten (Instanz von *IRepresentative*), der in der Lage ist über seine Methode *OnDropped* auf einer Arbeitsfläche *(*Parameter *workspace*) an einer bestimmten Stelle (Parameter *boundaries*) eine Kopie des repräsentierten virtuellen Elements zu erstellen. Neben einen Verweis auf den Repräsentanten merkt sich die Ressourcenbox in Form

eines *RepresentativeStore* Objekts, wie viele virtuelle Elemente mit Hilfe eines Repräsentanten sie noch erstellen darf (Attribut *Amount*) und ob es eventuell unendlich viele sind (Attribut *InfiniteAmount*).

Neue Elemente nimmt eine Ressourcenbox entweder über ihre *Add* Methode unter Angabe eines Repräsentanten und Menge der Elemente auf oder wenn ein Benutzer ein virtuelles Element, was durch die Konkretisierung *WorkspaceElement* der Klasse *WorkspaceResource* repräsentiert wird auf die Ressourcenbox zieht. In diesem Fall lässt sich diese dann durch Aufruf der *CreateRepresentative* Methode der *WorkspaceElement* Instanz einen passenden Repräsentanten erstellen.

CopyBox setzt die Kopierboxen (siehe Kapitel 4.2.2) um und konkretisiert dazu *ResourceBox*.

4.3.4 DiagramViewModel

Die Aufgabe der *DiagramViewModel* Komponente ist es, die in Kapitel 4.2 beschriebene Benutzerschnittstelle von *DiagramModel* (Kapitel 4.3.2) hinsichtlich von Knoten und Verbindungen zu entkoppeln und damit zur Erfüllung von Anforderung *A25NF* beizutragen. Dafür setzt sie von den Views erkannte Nutzeraktionen auf Operationen der Komponente *DiagramModel* um. Außerdem stellt *DiagramViewModel* Daten von *DiagramModel* zur Anzeige aufbereitet über Datenattribute zur Verfügung. Die Komponente basiert auf der Komponente *WorkspaceViewModel* aus Kapitel 4.3.3. Abbildung 4.19 zeigt den groben Aufbau der Komponente.

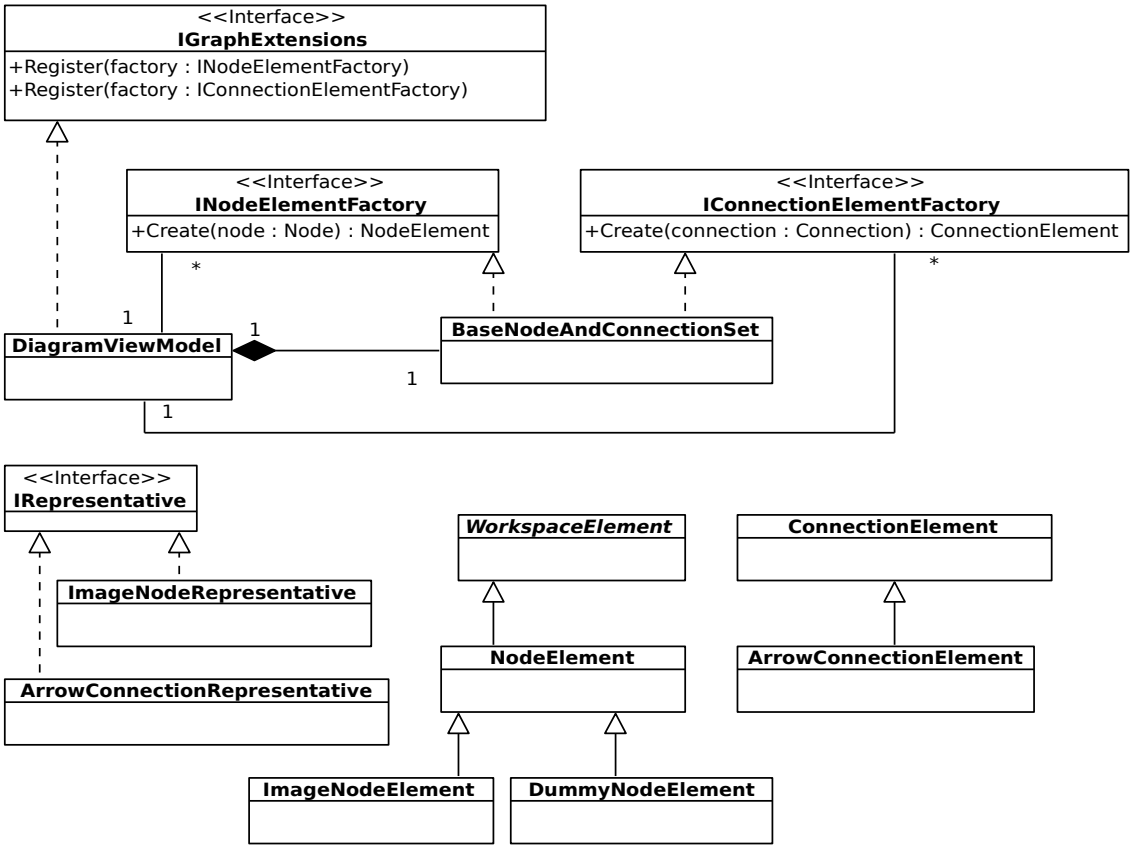

Abbildung 4.19: Aufbau des DiagramViewModels

Die Klasse *NodeElement* repräsentiert die Knoten des Diagramms. Da es sich bei diesen um virtuelle Elemente (siehe Kapitel 4.3.3) handelt, erbt *NodeElement* von der Klasse *WorkspaceElement* aus der Komponente *WorkspaceViewModel*.

ConnectionElement setzt die Verbindungslinien um. Bei diesen handelt es sich aus fachlicher Sicht nicht um virtuelle Elemente, die sich beliebig auf der Arbeitsfläche verschieben, vergrößern und drehen lassen, da ihre Dimensionen von der Position der Knoten an ihren beiden Enden abhängen. Dennoch haben Verbindungslinien wie Knoten Repräsentanten, da ein Benutzer sie mit Hilfe eines Dummyknotens wie in Kapitel 4.2.3 beschrieben in Ressourcenboxen ablegen kann.

ImageNodeElement sowie *ArrowConnectionElement* sind Konkretisierungen von *NodeElement* sowie *ConnectionElement* und repräsentieren Bild basierte Knoten sowie Verbindungslinien mit Pfeilen. Analog dazu gibt es die Repräsentantenklassen *Image-NodeRepresentative* sowie *ArrowConnectionRepresentative*, die wie in Kapitel 4.3.3 beschrieben von den Ressourcenboxen zur Aufnahme virtueller Elemente benötigt werden.

Für die Erstellung einer entsprechenden *NodeElement* oder *ConnectionElement* Instanz aus einem *Node* oder *Connection* Objekt (siehe Kapitel 4.3.2) kommt das Softwarepattern Fabrik Methode [Gamm95] zum Einsatz. Hierbei produzieren konkrete Fabrikklassen wie zum Beispiel *BaseNodeAndConnectionSet*, welche die abstrakten Fabriken *INodeElementFactory* beziehungsweise *IConnectionElementFactory* realisiert, *NodeElement* beziehungsweise *ConnectionElement* Instanzen. Der Klient *Diagram-ViewModel* geht alle ihm bekannten Fabriken durch und ruft jeweils die Methode Methode auf, bis er ein Produkt erhält oder aber alle Fabriken erfolglos probiert hat.

Andere Komponenten können konkrete Fabriken über die Methode *Register* der Schnittstelle *IGraphExtensions* bei der Komponente *DiagramViewModel* anmelden. Dies ist ein Grundpfeiler des Plugin Systems, da Plugins somit die Vielfalt an vom *DiagramViewModel* unterstützten Diagrammelemente erweitern können.

4.3.5 TextViewModel

Die Komponente *TextViewModel* entkoppelt die Benutzerschnittstelle von der Komponente *DiagramModel* (Kapitel 4.3.2) hinsichtlich des Texts im Diagramm. *TextViewModel* ist ähnlich wie *DiagramViewModel* (Kapitel 4.3.4) aufgebaut (siehe Abbildung 4.20).

Es benutzt eine Fabrik (*TextFactory*) um aus den *Text* Objekten der Komponente *DiagramModel TextContainer* Objekte zu erstellen, die Attribute für die Textplatzhalter (siehe Kapitel 4.2.5) besitzen. Instanzen von *TextContainer* referenzieren die entsprechenden Textobjekte, um Änderungen am Text oder seiner Anzeigeeigenschaften wie der Schriftart der Komponente *DiagramModel* mitteilen zu können.

Die Klasse *TextSnippet* repräsentiert die TextSnippets. Diese sind wie die Textplatzhalter virtuelle Elemente und entsprechend erben *TextContainer* und *TextSnippet* von der Klasse *WorkspaceElement* aus der Komponente *WorkspaceView-Model* (siehe Kapitel 4.3.3).

TextSnippetsRepresentative sowie *TextContainerRepresentative* realisieren die Repräsentanten der TextSnippets und Textplatzhalter für die Komponente *Workspace-ViewModel*.

Des Weiteren realisiert das TextViewModel die in Kapitel 4.2.2 beschriebene Box zum Erstellen von TextSnippets (TextCreatorBox) sowie die Box zum Auswählen der Eigenschaften eines Textes (*TextPropertiesBox*).

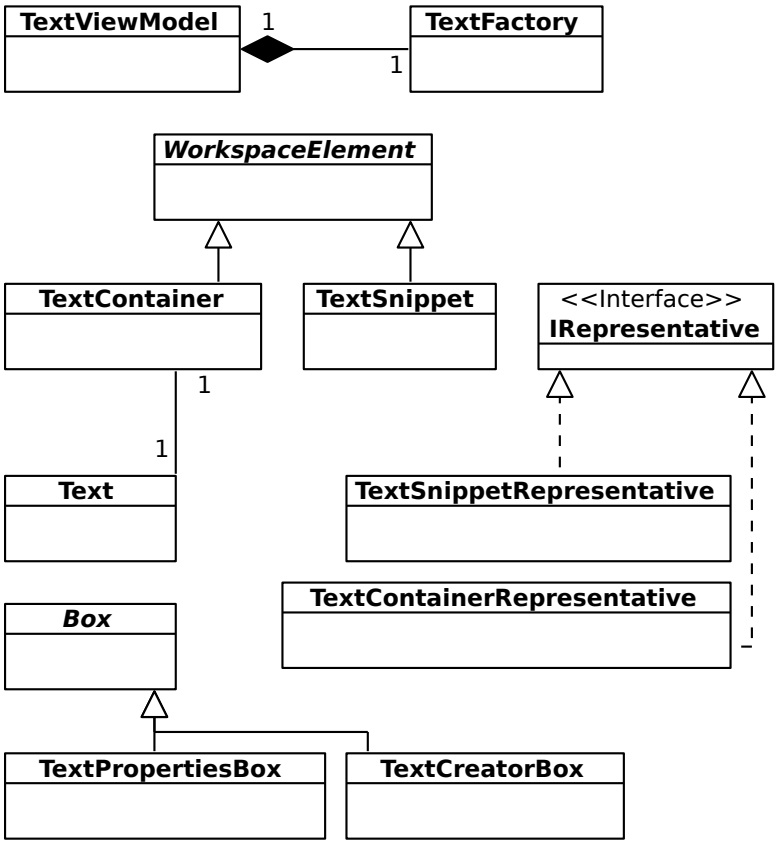

Abbildung 4.20: Aufbau der TextViewModel Komponente

4.3.6 Views

Die View Komponenten setzen zusammen die in Kapitel 4.2 beschriebenen Aspekte der Benutzerschnittstelle um, welche die Nutzer der Anwendung letztendlich auf dem Bildschirm sehen können. Des Weiteren nehmen die Komponenten die Benutzer-eingaben entgegen, führen mit der Komponente *GestureManager* aus Kapitel 4.3.1 die Gestenerkennung durch und leiten die Ergebnisse an die ViewModel Komponenten weiter.

Die View Komponenten bekommen bereits ziemlich genaue Vorgaben von den ViewModel Komponenten wie die einzelnen Elemente der Benutzerschnittstelle positioniert, groß und ausgerichtet sind. Auch das Aussehen von Knoten, Verbindungen und Text ist bezüglich Form, Farbe, Musterung zu großen Teil vorgegeben. Farbe und Musterung der Boxen und anderen Menüs können die Views allerdings frei bestimmen.

Abbildung 4.21 gibt einen groben Überblick über die wichtigen Klassen der View Komponenten. Dabei gehören die dunkelgrau markierten Klassen zur Komponente *DiagramView*, die hellgrau markierten zur Komponente *TextView* und die restlichen zur Komponente *WorkspaceView*. Es lassen sich an Hand der Namen Zuordnungen zu den Klassen in den ViewModel Komponenten erkennen. So repräsentiert zum Beispiel die Klasse *ConnectionElement (*siehe Kapitel 4.3.4) Verbindungen in der ViewModel Schicht, während *ConnectionView* die Verbindungen in der View Schicht umsetzt. Jede *ConnectionView* Instanz erhält folglich Informationen von einer *ConnectionElement* Instanz.

Die Klasse *WorkspaceElementView* ist der Klasse *WorkspaceResource* aus der Komponente *WorkspaceViewModel* (siehe Kapitel 4.3.3) zuzuordnen. Für die Klasse *WorkspaceElement* gibt es in der View Schicht keine entsprechende Klasse, da es in der Schicht irrelevant ist, ob sich ein virtuelles Element in eine Ressourcenbox ziehen lässt.

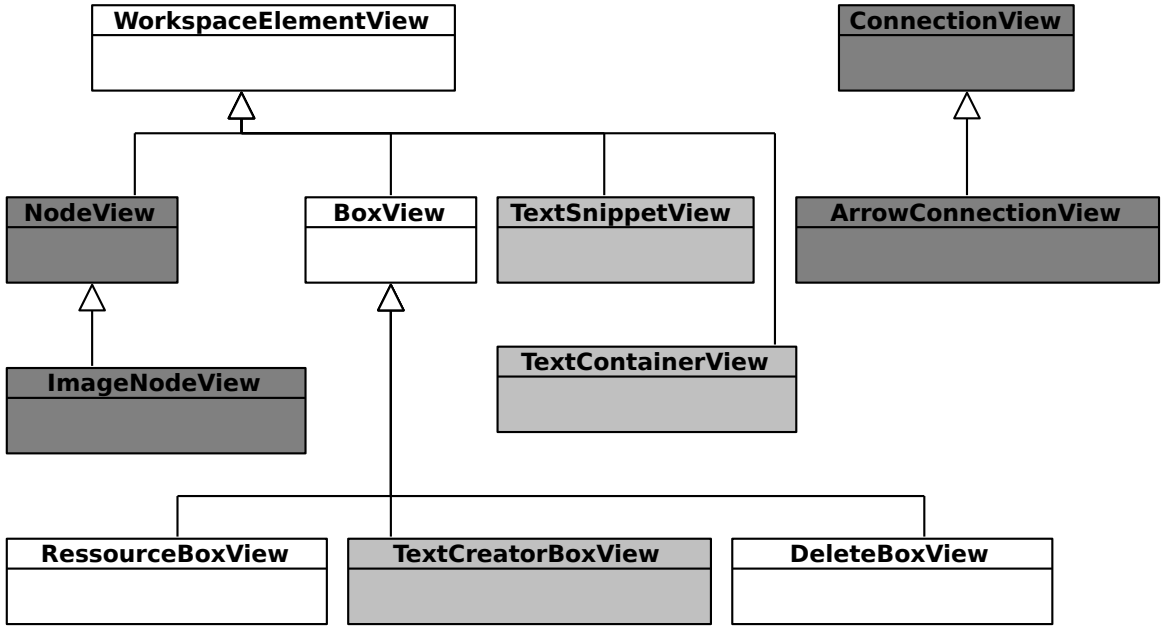

Abbildung 4.21: Überblick über wichtige Klassen der View Komponenten

4.3.7 Pluginsystem

Die Anwendung bietet wie durch Anforderung *A5F* gefordert ein Pluginsystem. Plugins können dabei den dem Diagramm zu Grunde liegenden Graphen um mehr Logik erweitern, in dem sie die generischen Eigenschaften der Knoten und Verbindungen (siehe Kapitel 4.3.2 DiagramModel) nutzen und neue Anzeigeelemente für Knoten und Verbindungen bei der Komponente *DiagramViewModel* anmelden (siehe Kapitel 4.3.4). Abbildung 4.22 zeigt die wichtigsten Bestandteile der Schnittstellen *IHost* und der *IPlugin*. Letztere muss eine Klasse des Plugins realisieren (in der Abbildung *PluginInitializer*). Die Schnittstelle sieht das Attribut *APIVersion* vor, dieses wird mit einem Wert der Anwendung verglichen. Dies dient dazu nach einer wesentlichen Änderungen der Schnittstellen der Anwendung mit dieser inkompatible Plugins zu entdecken. Das bedeutet, dass der interne Anwendungswert nach jeder wesentlichen Schnittstellenänderung inkrementiert wird und nur Plugins vom Plugin System

akzeptiert werden, die einen entsprechenden *APIVersion* Wert aufweisen.

Durch Aufruf der Methode *Register* gibt die Anwendung dem Plugin die Möglichkeit sich zu initialisieren und Anmeldungen bei den Komponenten *DiagramModel* und *DiagramViewModel* durchzuführen. Dazu bekommt das Plugin über den Parameter *host* die Referenz auf ein Objekt dessen Klasse die Schnittstelle *IHost* umsetzt. Über dieses erhält das Plugin Referenzen auf Objekte, welche die Schnittstellen *IDiagram* und das *IGraphExtensions* anbieten.

Um das Plugin wieder zu entfernen, ruft die Anwendung die Methode *Unregister* auf.

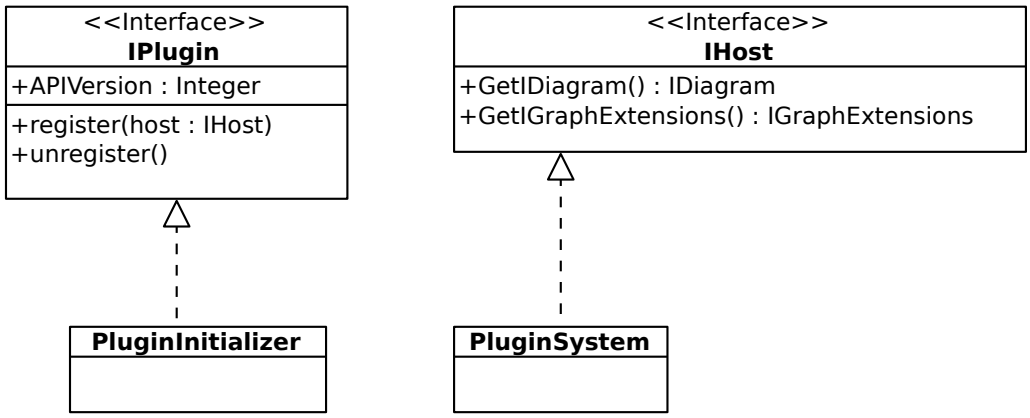

Abbildung 4.22: Schnittstellen vom Plugin System sowie den Plugins

4.4 Technischer Entwurf

Der technische Entwurf beschreibt wie die Techniken zur Umsetzung der fachlichen Architektur zur Umsetzung der Komponenten dieser angewendet werden. Zu den Techniken gehören insbesondere die SDKs, gegen welche die Anwendung entwickelt ist, sowie die eingesetzten Softwarebibliotheken von Drittherstellern.

4.4.1 Verwendung vom ScatterView

Wie in Kapitel 2.3.2 erwähnt stellt das Microsoft Surface SDK mit dem *ScatterView* und den *ScatterViewItems* für die Microsoft Pixelsense Plattform bereits mächtige Erweiterungen des Microsoft WPF Frameworks, welches die Realisierung von Benutzerschnittstellen unterstützt, zur Verfügung.

Die *ScatterViewItems* lassen sich von den Nutzern mit den Fingern verschieben, drehen und vergrößern. Beliebig andere WPF Elemente wie zum Beispiel Buttons oder Listen können in die *ScatterViewItems* eingebettet werden. Ein *ScatterViewItem* fungiert somit als WPF Container.

Ein wesentliches Argument für den Einsatz der *ScatterViewItems* in der Komponente *WorkspaceView* (siehe Kapitel 4.3) ist daher eine Reduzierung des Realisierungsaufwandes, da so das Microsoft Surface SDK einige Aufgaben der Komponente *GestureManagers* übernimmt. Zum Beispiel besitzen WPF Elemente generell ein Kontextmanagement, über das Ereignisse wie Berührungen den einzelnen Kontexten beziehungsweise Elementen zugeordnet werden. Auch muss ein Teil der Erkennung für die Gesten aus Kapitel 4.2.1 nicht durch die zu entwickelnde Anwendung erfolgen.

Zwar unterscheiden sich die auf die *ScatterViewItems* anwendbaren Gesten zum Drehen, Vergrößern und Verschieben in ihrer Form (siehe Kapitel 2.2.3) etwas von den in Kapitel 4.2.1 Beschriebenen. Aber bezüglich der Natur, des Kontexts und des Flusses gleichen sich die Gesten und mit Microsoft Pixelsense erfahrene Nutzer finden bei der Verwendung von *ScatterViewItems* ein gewohntes Verhalten der Anwendung vor. Letztendlich schränkt die Gestenerkennung des Microsoft Surface SDKs die Nutzer bezüglich der Form der Gesten weniger ein als eine exakte Realisierung von Kapitel 4.2.1.

Allerdings ist unter anderem für die Realisierung der Konflikterkennung nach Kapitel 4.1 oder der Erkennung von Ziehgesten (Kapitel 4.2.3) über Geraden sowie großer Gesten (Kapitel 4.2.5) dennoch eine Manipulation des Verhaltens der *ScatterViewItems* notwendig.

Zur Realisierung des modifizierten Verhaltens eignet sich die Verwendung der Routing Ereignisse des WPF Frameworks [Micr00f]. So wird unter anderem bei der Berührung eines WPF Elements (zum Beispiel eines Buttons) zunächst ein Ereignis auf diesem ausgelöst. Wenn das Element das Ereignis nicht als behandelt markiert, löst das Microsoft .net Framework das Ereignis auf dem Container Element (zum Beispiel einem *ScatterViewItem*) aus. Dieser Vorgang wiederholt sich solange, bis entweder ein Element das Ereignis behandelt oder das Wurzelelement erreicht ist, welches keinen übergeordneten Container mehr aufweist.

Die Realisierungen der Erkennung von Konflikten oder Gesten wie der Ziehgeste über Geraden registrieren sich für die berührungsbezogenen Ereignisse bei den entsprechenden *ScatterViewItems*, führen beim Auftreten dieser die Erkennungsalgorithmen aus und markieren die Ereignisse bei Erkennung einer Geste als behandelt.

4.4.2 TouchAdapter

Kapitel 2.3.2 beschreibt das Microsoft Surface SDK und weist daraufhin, dass bestimmte Teile der ersten Version des SDKs nicht auf der neueren Microsoft Pixelsense 2 Plattform laufen. Dazu gehört vor allem auch die Erfassung von Berührungen auf dem Bildschirm.

Da die Anwendung nach Anforderung *A27T* auf beiden Versionen der Plattform laufen muss, passt sich die Anwendung entsprechend beim Start an und setzt je nach Version des Pixelsense andere Adapter [Gamm95] ein. Somit sind die je nach Microsoft Pixelsense Version unterschiedlichen Quellcodeteile nur auf wenige Klassen konzentriert. Nur diese Klassen verwenden die zueinander inkompatiblen Schnittstellen der beiden Microsoft Surface SDK Versionen direkt. Alle restlichen Klassen arbeiten dann nur indirekt über die Adapterklassen mit den entsprechenden Schnittstellen des Microsoft Surface SDK.

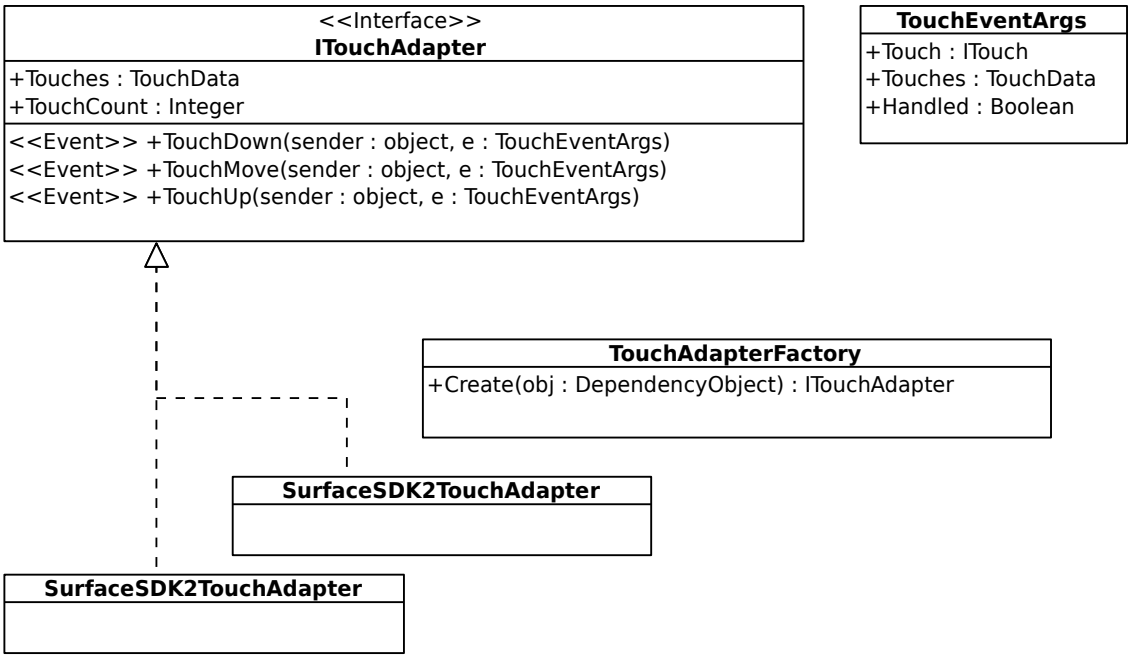

Abbildung 4.23: Überblick über die TouchAdapter

Abbildung 4.23 gibt einen Überblick über die Klassen und Schnittstellen, welche an der Adaption der verschiedenen Versionen des Microsofts Surface SDKs beteiligt sind.

Bei der Klasse *TouchAdapterFactory* handelt es sich um eine Fabrik [Gamm95], deren Aufgabe es ist entsprechend der Microsoft Pixelsense Version entweder eine *SurfaceSDK1TouchAdapter* oder eine *SurfaceSDK2TouchAdapter* Instanz für ein bestimmtes WPF Element (Parameter *obj* der *Create* Methode) zu erstellen. Der TouchAdapter bildet Surface SDK Version spezifische Ereignisse auf die in der Schnittstelle *ITouchAdapter* vorgesehenen Ereignisse und umgekehrt die Methoden und

Eigenschaften der Schnittstelle *ITouchAdapter* auf Methoden und Eigenschaften des WPF Elements ab.

Das Ereignis *TouchDown* wird dabei bei einer neuen Berührung, *TouchMove* bei Positionsänderung einer Berührung und *TouchUp* nach Beenden einer Berührung ausgelöst. Dabei werden über ein *TouchEventArgs* Objekt Informationen über die Berührung (*Touch* Attribut), welche das Ereignis ausgelöst hat, sowie alle Berührungen (*Touches* Attribut), welche zum Zeitpunkt des Ereignis vom WPF Element erfasst sind, übergeben. Der Schnappschuss der Daten über alle Berührungen wird zur Gestenerkennung wie unter anderem in Kapitel 2.2.5 beschrieben verwendet.

Über das Attribut *Handled* können die das Ereignis behandelnden Klassen dieses als fertig behandelt deklarieren. Dies ist zum Beispiel für die Umsetzung der Verhaltensmodifikation der *ScatterViewItems* (siehe Kapitel 4.4.1) notwendig.

4.4.3 Technischer Entwurf der Benutzerschnittstelle

Dieses Kapitel erklärt den technischen Entwurf zur Umsetzung der Benutzerschnittstelle, welche die Kapitel 4.2 aus Sicht der Anwendungsnutzer und 4.3.6 bezüglich der fachlichen Sicht beschreiben.

Anforderung A28T folgend erfolgt die Umsetzung der Benutzerschnittstelle in der Beschreibungssprache XAML sowie in C#. Der XAML Quellcode beschreibt zum einen wie die Benutzerschnittstelle aus durch das Microsoft Surface SDK angebotenen Kontrollelementen wie zum Beispiel Knöpfen, Listen oder Textboxen zusammengesetzt ist. Des Weiteren wird in XAML deklariert, welche Teile der Benutzerschnittstelle (View Komponenten) für entsprechende Teile der ViewModel Komponenten (siehe Kapitel 4.3) erzeugt werden und deklariert dabei die Datenbindungen (siehe Kapitel 2.3.3) zwischen Datenattributen der View Komponenten und Datenattributen der ViewModel Komponenten.

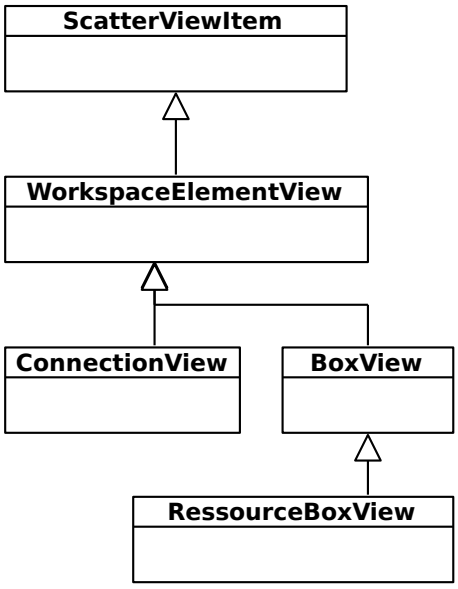

Abbildung 4.24: Überblick über C# Klassen in der View Schicht

Dagegen wird C# dazu benutzt Datenattribute zu definieren, wenn die bereits durch das Microsoft Surface SDK angebotenen nicht ausreichend sind. Darüber hinaus wird C# Quellcode in den View Komponenten dazu verwendet, spezielle Benutzerschnittstellenlogik wie zum Beispiel die Griffpunkte an den Boxen aus Kapitel 4.2.2 umzusetzen.

Abbildung 4.25 zeigt auf, welche View Klassen der fachlichen Architektur (siehe Kapitel 4.3.6) teilweise in Form von C# Quellcode umgesetzt werden. Einige virtuelle Elemente lassen sich wie in den Kapiteln 4.2.1 und 4.2.3 beschrieben horizontal sowie vertikal vergrößern und verkleinern und weisen also kein festes Seitenverhältnis auf. *ScatterViewItem* aus dem Microsoft Surface SDK bietet hierzu keine Geste an. Also muss *WorkspaceElementView* eine solche Geste

umsetzen. Auch die Umsetzung der in Kapitel 4.1 beschriebenen Konflikterkennung obliegt *WorkspaceElementView*. Die Aufgabe von *ConnectionView* ist es die Ziehgeste über Geraden (ebenfalls Kapitel 4.1) zu realisieren. Der Grund für die *BoxView* C# Klasse (Griffpunkte) wurde bereits genannt. Die Aufgabe der Klasse *ResourceBoxView* ist es, die Bildschirmkoordinaten der Auswahl eines Repräsentanten (siehe Kapitel 4.3.3) zu ermitteln und an die Komponente *WorkspaceViewModel* zu melden. Diese rechnet die Bildschirmkoordinaten in Diagrammkoordinaten um und lässt den Repräsentanten an diesen ein entsprechendes neues virtuelles Element erstellen.

Die meisten der View Klassen werden ganz ohne Einsatz von C# Quellcode realisiert und damit ausschließlich in XAML definiert. Dabei kommen die XAML Techniken Templates und Styles [Micr00g] zum Einsatz. Abbildung 4.25 zeigt beispielhaft anhand der Klasse *ImageNodeView* (für bebilderte Knoten) die groben Zusammenhänge.

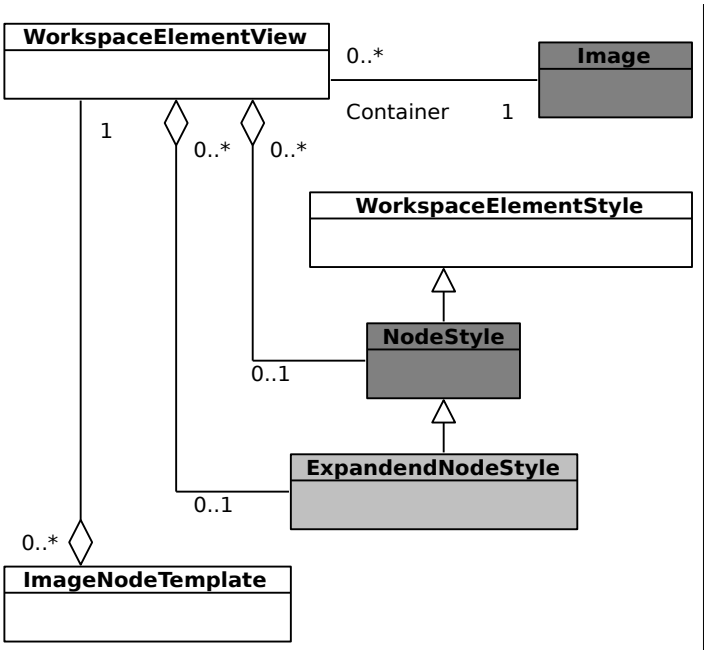

Abbildung 4.25: Übersicht ImageNodeView

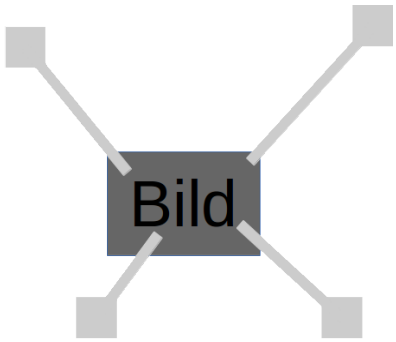

Abbildung 4.26: Expandierter Bildknoten

ImageNodeTemplate deklariert XAML Code, den das Microsoft Surface SDK verwenden soll, um eine Instanz der Klasse *ImageNode* der ViewModel in einem ScatterView Element in der View Schicht umzusetzen. Dieser XAML Code nutzt eine *WorkspaceElementView* Instanz als Container für Microsoft Surface SDKs *Image* Element, welches wiederum das Bild des Knoten (dunkelgrau markiert in Abbildung 4.26) anzeigt. Ein Datenattribut der zu Grunde liegenden Klasse *ImageNode* bestimmt dabei, ob das Microsoft Surface SDK entweder den Style *NodeStyle* oder dessen Konkretisierung *ExpandedNodeStyle* auf die *WorkspaceElementView* Instanz anwendet. Der jeweilige Style bestimmt dabei die Databindings (siehe Kapitel 2.3.3) zwischen den View Elementen (z.B. *Image* und *WorkspaceElementView*) und der Klasse *ImageNode* der Komponente *DiagramViewModel*. Außerdem fügt der ExpandedNodeStyles ein weiteres Microsoft Surface SDK Container Element sowie die Microsoft Surface SDK Elemente für die Verbindungspunkte und Verbindungslinien (in Abbildung 4.26 hellgrau markiert) zwischen *WorkspaceElementView* und *Image* ein.

Für den Einsatz von Templates und Styles spricht eine hohe Flexibilität. So kann das Image WPF Element zum Beispiel in einem neuen Template *TextBasedNodeTemplate* für Knoten mit Textinhalt ausgetauscht werden ohne das die verwendeten Styles angepasst werden müssen. Bei einer Änderung der farblichen Gestaltung der expandierten Verbindungspunkte reicht eine Anpassung der Styles, damit alle Knoten die Änderung des Aussehens erfahren.

4.4.4 Umsetzung von „Drag and Drop“

„Drag and Drop“ bedeutet, dass ein Benutzer der Anwendung virtuelle Elemente ziehen (Drag) und eventuell auf einem bestimmten virtuellen Ziel fallen lassen kann (Drop). Kapitel 4.2.2 beschreibt den konkreten Einsatz des „Drag and Drop“ Konzept im in dieser Arbeit entwickelten Diagrammeditor.

Das Microsoft Surface SDK bietet Klassen an, die bei der Realisierung von „Drag and Drop“ helfen. Allerdings arbeiten alle diese Klassen direkt mit den Benutzerschnittstellenelementen des WPF Frameworks und damit in der View Anwendungsschicht nach dem MVVM Pattern (siehe Kapitel 2.3.3 und 4.3).

Ein genereller Vorteil davon ist, dass in der View Schicht auf jeden Fall Informationen zu den Dimensionen von Elementen sowie die Berührungskoordinaten vorliegen. Diese können dann zur Entscheidung herangezogen werden, ob ein virtuelles Element auf einem anderen fallen gelassen worden ist.

„Drag und Drop“ in der View Schicht umzusetzen bringt jedoch auch Nachteile mit sich. So erhöht sich die Komplexität der Schicht durch Klassen übergreifende Logik. Des Weiteren weist das Microsoft Surface SDK in der Version 1 bezüglich „Drag and Drop“ etwas andere Schnittstellen als in Version 2 auf. Letztendlich müssen für die Entscheidung, ob ein virtuelles Element auf einem anderen fallen gelassen worden ist, benötigte fachliche Informationen über die ViewModel Schicht von der Model in die View Schicht durch gereicht werden.

Daher hat sich der Autor dafür entschieden, die Realisierung von „Drag and Drop“ fast ausschließlich in der ViewModel Schicht zu realisieren. Dazu tauscht sich die in Kapitel 4.3.3 beschriebene Komponente *WorkspaceViewModel* über das Databinding mit der View Schicht darüber aus, ob ein virtuelles Element von einer Fingerspitze berührt wird oder nicht. Des Weiteren entscheidet eine von anderen Komponenten überschreibare Methode der Klasse *WorkspaceResource*, ob das entsprechende virtuelle Element sich über einem anderen befindet. Unter Verwendung dieser Informationen und der Position eines virtuellen Elements erkennt die WorkspaceViewModel Komponente, ob ein virtuelles Element fokussiert, gezogen und entweder auf einem anderen Element oder der Arbeitsfläche fallen gelassen worden ist und informiert die involvierten Elemente entsprechend.

4.4.5 Einbindung des NShape Frameworks

Nach Anforderung *A29T* wird das NShape Framework (siehe Kapitel 2.4) verwendet. Dabei setzt es die Anwendung für die Verwaltung des Diagramms und dessen Modifikation auf der Datenebene ein. NShapes Anzeigelogik verwendet noch den Vorgänger des WPF Frameworks, WinForms für die graphische Benutzeroberfläche. Das das Microsoft Surface SDK jedoch auf WPF aufbaut, greift die Anwendung nicht auf die Anzeigelogik des NShape Frameworks zurück.

Es werden daher nur die Controller und Core Komponenten von NShape durch die Anwendung benutzt. Diese bieten einen sehr viel größeren Funktionsumfang an, als auf Grund der Anforderungen aus Kapitel 3.2 benötigt wird. Daher stellt die Komponente *DiagramModel* (Kapitel 4.3.2) der fachlichen Architektur eine Fassade [Gamm95] des NShape Frameworks dar. Gegen diese ist die Anzeigelogik der Anwendung in Form der DiagramViewModel und TextViewModel Komponenten (Kapitel 4.3.4 und 4.3.5) entwickelt.

4.4.6 Plugins

Ein Plugin wird gegen die Schnittstellen der Anwendung als eine dynamisch gebundene Bibliothek gebaut. Wenn diese dann in das gleiche Verzeichnis wie die auszuführende Datei der Anwendung abgelegt wird, lädt die Anwendung das entsprechende Plugin automatisch beim Start.

5 Realisierung und Test

Dieses Kapitel geht kurz auf die Realisierung des Entwurfs aus Kapitel 4 und auf das Testen ein. Zunächst erörtert ein Unterkapitel inwiefern die Realisierung vom Entwurf abweicht. Ein weiteres Kapitel beschreibt den Status der Realisierung hinsichtlich des Entwurfs. Ein drittes Unterkapitel geht auf das Testen ein.

5.1 Abweichungen der Realisierung vom Entwurf

Dieses Kapitel geht auf die Stellen ein, an denen die Realisierung vom Entwurf abweicht. Dabei handelt es sich um die erfolgte Realisierung und nicht um die Teile des Entwurfs, die nicht umgesetzt worden sind. Auf Letztere geht das Folgekapitel 5.2 ein.

5.1.1 Beziehung zwischen WorkspaceElementView und ScatterViewItem

Bei der Realisierung der Komponente *WorkspaceElementView* als von *ScatterViewItem* abgeleitete Klasse (siehe Kapitel 4.4.1 und 4.4.3) trat das unerwartete Problem auf, dass die *WorkspaceElementView* Instanzen beim Hinzufügen zum *ScatterView* WPF Element vom Microsoft Surface SDK in einen weiteren *ScatterViewItem* Container gepackt werden. Abbildung 5.1 zeigt die somit entstehenden Klassenrelationen, wenn *WorkspaceElementView* von *ScatterViewItem* erbt.

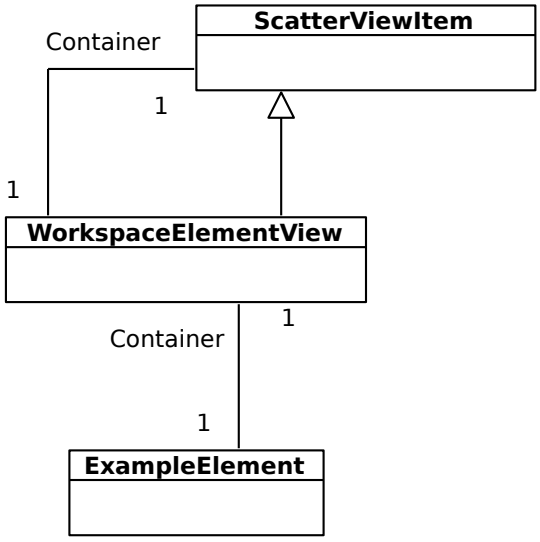

Abbildung 5.1: Weitere ScatterViewItem Instanz als Container

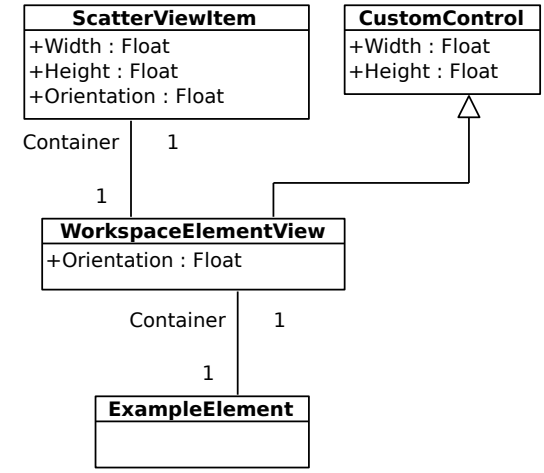

Abbildung 5.2: Realisierung WorkspaceElementView

Es ist in angemessener Zeit nicht gelungen das Microsoft Surface SDK so zu verwenden, dass es die *WorkspaceElementView* Instanzen unmittelbar und ohne zusätzlichen Container dem *ScatterView* hinzufügt. Daher ist *WorkspaceElementView* wie von Abbildung 5.1 veranschaulicht realisiert worden. *WorkspaceElementView* erbt von dem Benutzerschnittstellenelement *CustomControl* aus dem Microsoft Surface SDK, so dass ein *ScatterViewItem* es aufnehmen kann und es gleichzeitig weiterhin selber Container sein kann. Ein paar benötigte Funktionen und Attribute wie zum Beispiel *Width* für die Breite oder *Height* für die Höhe erbt *WorkspaceElementView* auf

diese Weise. Andere Attribute wie zum Beispiel *Orientation* für die Ausrichtung des entsprechenden virtuellen Elements realisiert *WorkspaceElementView* dagegen selber. Falls nötig werden diese Attribute dann mit den entsprechenden Attributen vom *ScatterViewItem* Container Element abgeglichen.

5.1.2 Komponenteneinteilung der Views

Die fachliche Architektur sieht wie in Kapitel 4.3.6 beschrieben eine Trennung der View Schicht in drei Komponenten vor. Die Realisierung weist diese Trennung nicht auf, da dem Autor zur Anfangszeit der Realisierungsphase das diesbezügliche Wissen über XAML fehlte. Da insgesamt andere Aspekte der Realisierung höher priorisiert worden sind, gehört die Trennung der View Schicht in unterschiedliche Komponente zu den Teilen des Entwurfs, die mit den gegebenen Entwicklerressourcen nicht umgesetzt werden konnten.

5.1.3 Unterschiede bei den Lagerflächen (Boxen)

Die Realisierung der Komponente *WorkspaceViewModel* (siehe Kapitel 4.3.3) weist bezüglich der Boxen Unterschiede zum Entwurf auf, da die Erstellung bestimmter Boxen etwas anders als in Kapitel 4.2.2 beschrieben abläuft und Boxen nicht in jedem Falle über Knoten und Textelementen dargestellt werden.

Die Boxerstellung erfolgt nicht über das Drücken von Knöpfen möglichst nah der Position des Knopfes auf dem Bildschirm, sondern wie die Erstellung anderer virtueller Elemente über das Ziehen eines Repräsentanten auf die Arbeitsfläche. Dabei kann sofort nach der Auswahl eines Boxrepräsentanten mit dem Finger über die oft verwendeten Gesten neben der Position auch noch die Größe und die Ausrichtung der Box angepasst werden. Die Vorteile der Boxerstellung auf diese Art und Weise ist, dass sie eine direkte Eingabemethode nach Kapitel 2.2.1 ist und die Erstellung eines jeglichen virtuellen Elements auf der Arbeitsfläche dem selben Schema folgt.

Anders als im Entwurf vorgesehen werden Knoten und Textelemente, auf denen sich ein Finger befindet über Boxen, die nicht zur selben Zeit in ihrer Position, Ausrichtung oder Größe verändert werden dargestellt. Dies hat unter anderem den Vorteil, dass dieses Verhalten mehr der direkten Eingabe entspricht, aber den Nachteil, dass es Konflikte begünstigt. So kann zum Beispiel ein Benutzer einen anderen bei der Auswahl eines Repräsentanten in einer Ressourcenbox behindern, wenn er ein anderes virtuelles Element zur selben Zeit über die Box zieht. Nach Einschätzung des Autors überwiegt jedoch der Vorteil.

5.1.4 Unterschiede bei den Text bezogenen Komponenten

Die Realisierungen der Komponenten *TextView* und der *TextViewModel* weisen geringfügige Unterschiede zur Spezifikation (siehe Kapitel 4.3.5 und 4.3.6) auf. Anstatt das Verändern der Texteigenschaften wie zum Beispiel die Schriftgröße über separate Boxen anzubieten, expandiert sich ein Textplatzhalter ähnlich wie die Knoten (siehe Kapitel 4.1) um die entsprechenden Bedienelemente. Somit sind diese immer sehr direkt an der Stelle ihres Effekts (dem Textplatzhalter), was einer direkten Eingabe näher kommt als eine, Box die weit von dem zugehörigen Textplatzhalter entfernt sein kann.

5.1.5 Zusätzliche Komponenten Tools und AppCore

Die Realisierung weist gegenüber dem Entwurf zwei weitere Komponenten auf.

Die Komponente *Tools* wird von allen anderen Komponenten verwendet und enthält Klassen, deren Funktionalität sich der Autor im Microsoft .net Framework gewünscht hätte. Diese zum Teil von .net abhängigen C# Klassen zeichnet aus, dass sie portabel sind in der Hinsicht, dass sie für Realisierungen zu ganz anderen Problemstellungen interessant sein könnten und unabhängig vom Microsoft Surface SDK und NShape Framework sind. Beispiele für Bestandteile der Komponente sind Methoden, welche auf Listen angewendet werden können oder eine Klasse, welche nicht Achsen paralleles Rechtecke realisiert.

Die Komponente *AppCore* initialisiert die anderen Komponenten und verbindet sie miteinander. Sie ist damit das Gegenteil zur Tools Komponente, da sie alle anderen Komponenten verwendet. Eine konkrete Aufgabe von AppCore ist zum Beispiel das Bereitstellen der Singleton Menübox (siehe Kapitel 4.2.2).

5.2 Status der Realisierung

Dieses Kapitel beschreibt wie weit der Entwurf aus Kapitel 4 umgesetzt worden ist. Dazu geht ein Unterkapitel zunächst nach der fachlichen Komponenteneinteilung aus Kapitel 4.3 vor und erklärt den Realisierungsstatus einer jeden Komponente. Anschließend schlüsselt ein weiteres Unterkapitel auf, welche Anforderungen entsprechend umgesetzt worden sind.

5.2.1 Status nach fachlichen Komponenten

Kapitel 4.4.1 beschreibt, dass das *ScatterViewItem* viele Teilaufgaben der Komponente **GestureManager** erfüllen kann. Die Umsetzung der verbleibenden Teilaufgaben verteilt sich auf die Komponenten der View Schicht. Von den oft verwendeten Gesten aus Kapitel 4.2.1 gibt es Umsetzungen der Ziehgeste, der Zoomgeste, der Drehgeste und des einfachen Klicks. Die horizontale beziehungsweise vertikale Zoomgeste ist nicht umgesetzt worden.

Für die exakte Umsetzung der oft verwendeten Gesten (siehe Kapitel 4.2.1) existiert eine experimentelle Realisierung, deren Verwendung sich im Quellcode der Klasse *WorkspaceElementView* (siehe Kapitel 4.4.3) aktivieren lässt. Diese Realisierung diente dem Autor der Arbeit dazu den Realisierungsaufwand für eine Eigenimplementierung der Gestenerkennung abzuschätzen und funktioniert und erkennt die Gesten nicht zuverlässig.

Der Status der Realisierung der Komponente **DiagramModel** genügt, damit die darauf aufbauenden Komponenten funktionieren. Das bedeutet, dass eine Persistenz der Diagrammdaten im Arbeitsspeicher und fundamentale Operationen auf diesen Daten wie zum Beispiel das in Kapitel 4.2.3 erwähnte Autorouting rudimentär umgesetzt worden sind. Allerdings ist die Anbindung an das NShape Framework nicht sehr weit fortgeschritten und es werden lediglich einige Datenattribute von Knoten gespeichert und geladen. Damit ist eine sitzungsübergreifende Persistenz der Diagramme nicht

gewährleistet.

Einige entworfene Aspekte der Komponente **WorkspaceViewModel** (siehe Kapitel 4.3.3) wurden realisiert. So lassen sich virtuelle Elemente hinzufügen sowie entfernen. Elementposition, -ausrichtung sowie -größe werden mit den entsprechenden Views synchronisiert. Die virtuelle Arbeitsfläche mit in seiner Größe, Drehung und Ausrichtung veränderbaren Bildausschnitt existiert. Ebenso bietet die Realisierung die Umrechnung zwischen Bildschirm- und Diagrammkoordinaten an. Dass Boxen stets vor den nicht durch den Nutzer berührten anderen virtuellen Elementen liegen und des Weiteren zuletzt berührte virtuelle Elemente über anderen angezeigt werden, ist umgesetzt worden.

Boxen im Allgemeinen sind in der ViewModel Schicht realisiert worden. Allerdings fehlt das in den Kapiteln 4.2.2 und 4.1 beschriebene Spielfigurenkonzept komplett. Auch können sich Boxen überlappen und teilweise über den Bildschirmrand hinaus ragen.

Die konkreten Realisierungen der Löschbox, der Ressourcenbox sowie deren Spezialisierung Kopierbox existieren in der ViewModel Schicht.

Die Realisierung der Komponente **DiagramViewModel** bietet analog zur Realisierung der Komponente *DiagramModel* bebilderte Knoten an. Außerdem sind die visuellen Aspekte auf Geraden basierter Verbindungslinien ohne Pfeilspitzen sowie die in Kapitel 4.2.3 beschriebenen Dummyknoten teilweise realisiert worden. Position, Größe sowie Drehung der Knoten werden zwischen Model Schicht und View Schicht synchronisiert. Die Verbindungspunkte existieren auf der ViewModel Schicht, sind allerdings nutzlos, da die Operation zur logischen Verbindungserstellung fehlt.

Unter den beschriebenen allgemeinen Einschränkungen zum Beispiel bei den Gesten oder der Umsetzung der Boxen setzt die Realisierung der Komponente **TextViewModel** alle in ihrem Entwurf (Kapitel 4.2.5 und 4.3.5) vorgesehenen Funktionalitäten um. So gibt es eine Implementierung der Box zur Erstellung der Textsnippets. Die Realisierung der Textplatzhalter erlaubt auf der ViewModel Schicht das Festlegen der Schriftgröße und ob der Text kursiv oder fett ist. Diese Eigenschaften werden sowohl mit der Komponente *DiagramModel* als auch mit der View Schicht abgeglichen.

Für die **WorkspaceView** Komponente ist entsprechend der Komponente *Workspace-ViewModel* der *WorkspaceElementView* Container (siehe auch Kapitel 4.4.3) realisiert worden, so dass die Nutzer virtuelle Elemente auf der Arbeitsfläche verschieben, unter einem festen Seitenverhältnis vergrößern und drehen können. Einen erkannten einfachen Klick melden die virtuellen Elemente an die ViewModel Schicht. Auch der *BoxView* Container für die Boxen existiert in der Realisierung. Er weist die Titelleiste sowie die Griffpunkte auf. Allerdings lassen sich die Boxen anders als im Entwurf spezifiziert (siehe Kapitel 4.2.2) nicht minimieren und entweder vertikal oder horizontal

vergrößern beziehungsweise verkleinern. Analog zur ViewModel Schicht sind die View Klassen für die Löschbox, die Ressourcenbox und die Kopierbox realisiert worden.

Die Gesten für das Verändern des angezeigten Bildausschnitts wurden nicht implementiert. Der Bildausschnitt kann dennoch über eine spezielle Menübox („Clipping Control“ in Abbildung 5.3) verändert werden.

Analog zur Komponente *DiagramViewModel* weist die Realisierung der View Schicht Klassen der Komponente **DiagramView** zur Visualisierung bebilderter Knoten (siehe Abbildung 5.4), Dummyknoten und Geraden basierter Verbindungslinien ohne Pfeilspitzen auf. Zur Ziehgeste über Geraden (siehe Kapitel 4.1) gibt es keine Realisierung.

Entsprechend der Realisierung der Komponente *TextViewModel* sind alle View Klassen der **TextView** Komponente realisiert worden. Abbildung 5.5 zeigt davon die Box zur Texterstellung und die Textplatzhalter.

Es gibt keine Realisierung des **Pluginsystems** und folglich auch keine Plugins.

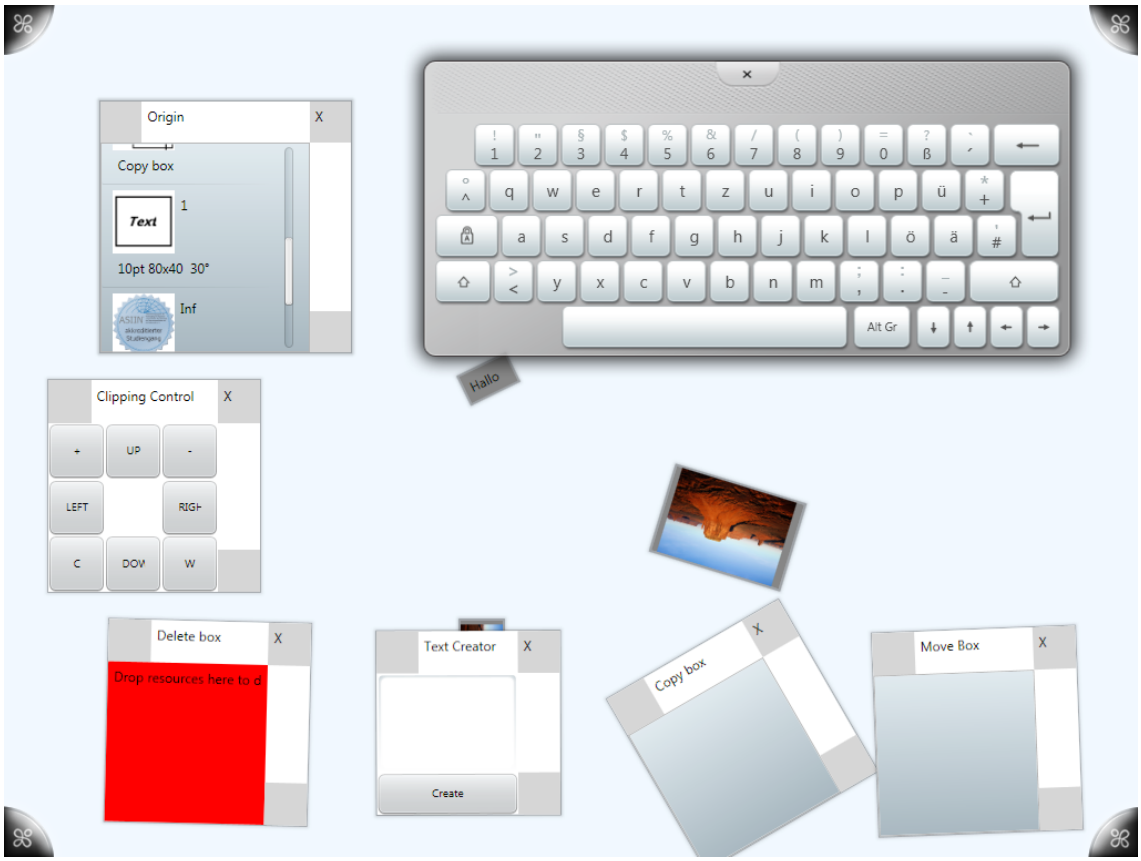

Abbildung 5.3: Screenshot, der alle Boxen zeigt

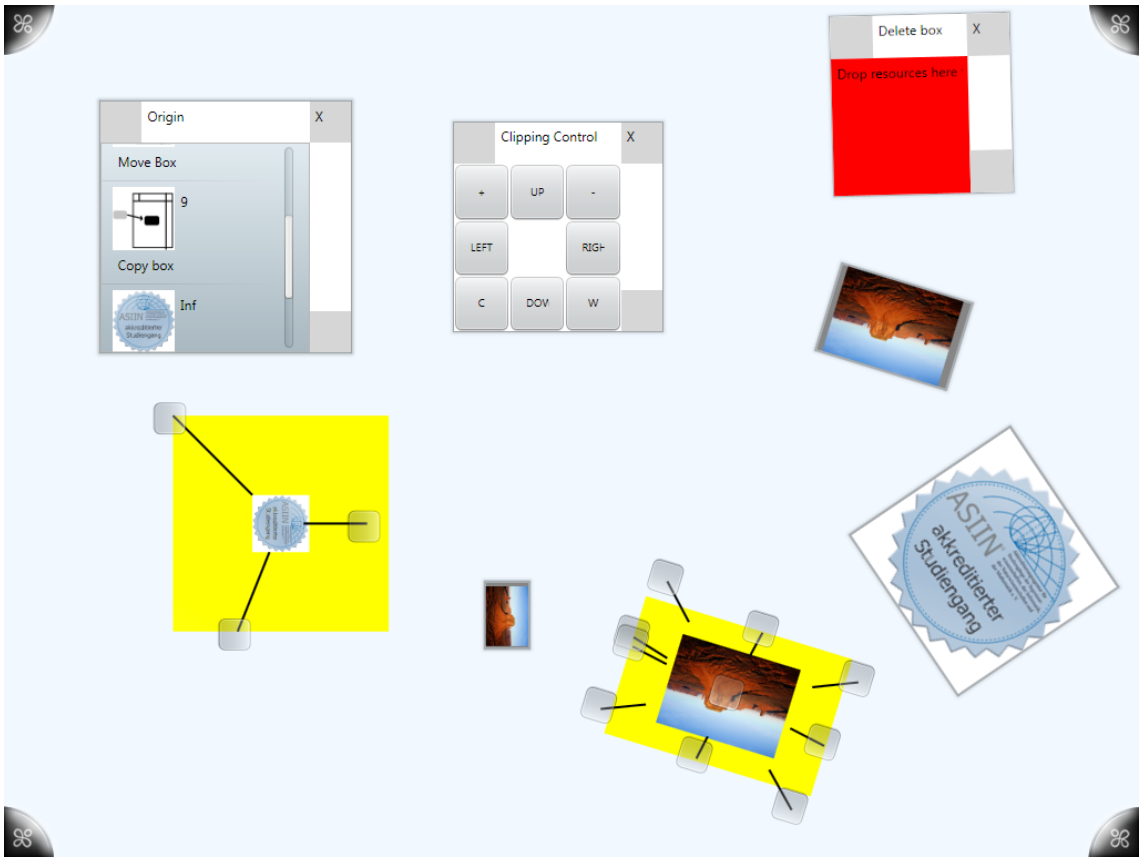

Abbildung 5.4: Knoten in expandiertem und normalen Zustand

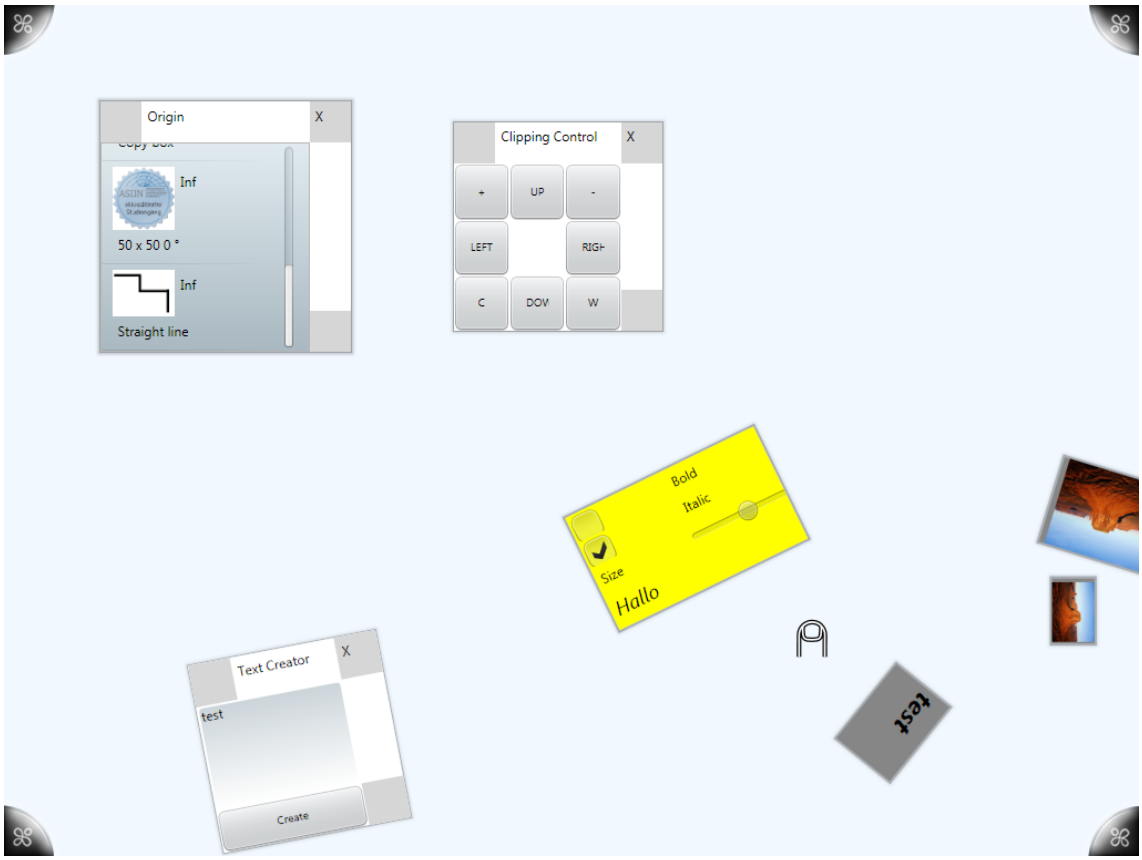

Abbildung 5.5: Box zur Texterstellung und Textplatzhalter (expandiert und normal)

5.2.2 Status nach Anforderungen

Tabelle 5.1 fasst zusammen, welche Anforderungen bis in die Realisierung erfüllt worden sind und wessen Erfüllung weitere konkrete Umsetzungen des Entwurfs erfordert.

	Beschreibung in Stichworten	**erfüllt**
A1F	Erstellung und Veränderung von Knoten	ja
A2F	Knoten basierend auf Grundformen und Bildern	*nur Bilder*
A3F	Verbindungslinien	*nein*
A4F	Verbindungspunkte	ja
A5F	Plugin System	*nein*
A6F	Generische Attribute Knoten und Verbindungen	*keine Persistenz*
A7F	Unterstützung der Nutzer bei kleinen Elementen	ja
A8F	Vergrößerung und Verkleinerung ganzer Diagramme	ja
A9F	Arbeitsfläche mit veränderbaren Bildausschnitt	ja
A10F	Unterstützte Gesten nach Richtlinien aus Kapitel 2.2.3	ja
A11F	Sofern möglich Verzicht auf indirekte Eingabemethoden	ja
A12F	Virtuelle Lagerflächen	ja
A13F	Erkennung und Lösung von Konflikten	*nein*
A14F	Drehen der Diagramme in 90° Schritten	ja
A15F	Temporäre Drehung von Elementen	*nein*
A16NF	Erhaltung der Datenkonsistenz bei Fehlbedienung	*nein*
A17NF	Robustes Speichern auch bei hoher Systembelastung	*nein*
A18NF	Maximal zwei Eingabeschritte für atomare Funktionen	*ja*
A19NF	Seltener Einsatz abstrakter Gesten	*ja*[1]
A20NF	Geringe Einarbeitungszeit der Anwendungsnutzer	*??* [2)]
A21NF	Reaktion auf Eingabe nach spätestens 0,5 Sekunden	*??* [2)]
A22NF	LCOM4 Wert zwischen 1 und 2	*??* [2)]
A23NF	In Code Dokumentation aller Schnittstellen	*nein*
A24NF	Zyklomatische Zahl nie größer als 10	ja
A25NF	Tabletop PC spezifische Techniken Komponenten mäßig von Diagramm bezogenen Funktionen getrennt	ja
A26NF	Parallele Ausführung	*nein*
	Fortsetzung und Beschriftung auf nächster Seite	

	Beschreibung in Stichworten	**erfüllt**
A27T	Anwendung läuft auf Microsoft Pixelsense 1 und 2	*nur 1*
A28T	Realisierung mit Surface SDK, C# und XAML	ja
A29T	Anwendung baut auf NShape Framework auf	*bedingt*

Tabelle 5.1: durch Realisierung (nicht) erfüllt Anforderungen

[1] da keine abstrakten Gesten realisiert worden sind; [2] ?? bedeutet ungetestet

5.3 Test

Dieses Kapitel geht auf das Testen im Rahmen der Arbeit ein. Zunächst motiviert ein Unterkapitel das Testen und schränkt es auf die Verifikation der Einhaltung von Spezifikationen ein. Danach gehen zwei Unterkapitel auf die zur Verifikation durchgeführten Tätigkeiten ein. Abschließend bewertet ein Unterkapitel diese.

5.3.1 Testmotivation und Einschränkung

Eine Anforderung ist in der Tabelle 5.1 aus Kapitel 5.2.2 als erfüllt gekennzeichnet unter der Annahme, dass der Entwurf auch wirklich den Anforderungen folgt und die Realisierung der Spezifikationen durch den Entwurf entspricht. Auf Grund konzeptioneller oder Programmierfehler kann diese Annahme für bestimmte Anforderungen falsch sein. Solche Fehler aufzudecken und zu *verifizieren*, dass alle Spezifikationen erfüllt sind, ist eine Aufgabe von Tests.

Darüber hinaus kann es nötig sein zu zeigen, dass eine entwickelte Anwendung mit den an sie gestellten Anforderungen, ihrem Entwurf und letztendlich ihrer Realisierung die Wünsche der Zielgruppe ihrer Anwender erfüllt. Die *Validierung,* dass eine Anwendung ihrem Einsatzzweck gerecht wird, ist für Produzenten von Anwendungen mit kommerziellen Absichten essentiell. Denn es hat für diese Interessengruppe keinen Nutzen mit viel Aufwand die Entwicklung einer in sich perfekten Anwendung voranzutreiben, mit der die Kunden dennoch unzufrieden sind, da die Anwendung nicht ihre Erwartungen erfüllt. Die Validierung einer Anwendung kann unter anderem durch stetigen Austausch mit dem Auftraggeber erfolgen oder, falls eine Anwendung nicht im Zuge eines Auftrags entwickelt wird, durch Marktforschung oder Feldtests mit potentiellen Kunden bereits in einer frühen Entwicklungsphase.

Auch für Grundlagenforschung kann sich Validierung anbieten, um zum Beispiel den möglichen Nutzen neuer Techniken nachzuweisen.

Eine aussagekräftige Validierung über Marktforschung oder mit Testnutzern bedeutet einen hohen Aufwand, da eine repräsentative und ausreichend große Untermenge an Personen, die der Zielgruppe entsprechen für Befragungen oder Nutzbarkeitstests ausgewählt werden müssen. Nur eine Hand voll von Freunden der Entwickler zu nehmen reicht bei weitem nicht aus, weil dabei die Gefahr besteht, dass die Testpersonen mit ihren Ansichten und Fähigkeiten zu sehr den Entwicklern ähneln und nicht die gesamte Zielgruppe repräsentieren.

Auf Grund des hohen Aufwandes einer Validierung mit Testkunden und da mit dieser Arbeit keine kommerziellen Ziele verfolgt werden, hat der Autor auf eine Validierung mit Testkunden verzichtet.

Die folgenden Unterkapitel von 5.3 beschränken sich daher auf Themen zur durchgeführten Verifikation der Einhaltung von Spezifikationen.

5.3.2 Dynamische Tests

Dieses Kapitel erklärt, welche dynamischen Tests durchgeführt wurden. Einen dynamischen Test zeichnet aus, dass für ihn die getestete Anwendung oder Teile von ihr ausgeführt werden. Dabei bekommt die getestete Realisierung als Eingabeparameter sowohl typische als auch Extremwerte überreicht und es wird überprüft, ob die Reaktion der Realisierung unter anderem in Form von Ausgabewerten ihrer Spezifikation durch den Entwurf entspricht.

Für *WorkspaceViewModel* (siehe Kapitel 4.3.3) gibt es ein paar Komponententests, die essentielle Funktionalität überprüfen. Zum Beispiel verifiziert ein Test, dass Boxen einen höheren Z Wert als andere nicht von einem Nutzer berührte virtuelle Elemente zugewiesen bekommen. Der Z Wert legt fest, welches Element bei sich überlappenden Elementen komplett angezeigt wird. Die anderen Komponenten wurden nicht einzeln, getrennt von der gesamten Realisierung getestet.

Der Autor hat im Rahmen eines Systemtests der in den Kapiteln 5.1 und 5.2 beschriebenen Version der Realisierung deren Verhalten bezüglich folgender Punkte verifiziert:

- Verändern der Größe, Position sowie Ausrichtung von Knoten sowie die Expansion von Knoten
- Erstellen, Löschen und Kopieren von Knoten über die entsprechenden Boxen
- Entfernen und Erstellen von Boxen
- Verändern des angezeigten Bildausschnitts bezüglich Position, Größe und Drehung
- Erstellen, Kopieren und Löschen von Textsnippets sowie von Textplatzhaltern
- Anpassung der Texteigenschaften fett, kursiv sowie der Schriftgröße

Dieser Systemtest wurde vom Autor gegen die Beschreibung der Anwendung aus Nutzersicht in Kapitel 4.2 unter Berücksichtigung von den Kapiteln 5.1 und 5.2 entwickelt. Der Test deckt dabei nur einfache Anwendungsfälle wie zum Beispiel das Arbeiten mit genau einem Knoten und keine dem Alltag entsprechenden Anwendungsfälle mit komplexen Diagrammen, die sehr viele Elemente enthalten, ab. Auch reicht für den Systemtest ein Tester aus. Der Anhang dieser Arbeit enthält eine detaillierte Beschreibung des Systemtests mitsamt eines Protokolls über die Testdurchführung.

5.3.3 Statische Analyse

Neben der Entwicklung und Durchführung der dynamischen Tests hat der Autor eine statische Analyse durchgeführt. Hierbei kam die Realisierung nicht zu Ausführung, sondern Computer gestützte Entwicklungswerkzeuge haben den zur Realisierung gehörenden Binärcode sowie Sourcecode analysiert und zum Beispiel hinsichtlich der Einhaltung allgemein empfohlener Programmierrichtlinien untersucht.

Die Ergebnisse dieser Untersuchungen helfen beim Finden von Fehlern. So kann ein Computerprogramm zum Beispiel zuverlässig ermitteln, ob es Stellen im Sourcecode gibt, die niemals zur Ausführung kommen können. Dies kann zum Beispiel aus fehlerhaften Bedingungen in Fallunterscheidungen resultieren.

Auch kann ein Computerprogramm aus dem vorliegenden Sourcecode bestimmte Metriken berechnen. Diese Werte unterstützen dann die objektive Beurteilung des Sourcecodes zum Beispiel hinsichtlich seiner Komplexität oder des Zusammenhalts. Die Anforderungen *A24NF* und *A22NF* verwenden zum Beispiel in ihren Formulierungen die Metriken LCOM4 sowie zyklomatische Zahl.

Für die statische Analyse in der Arbeit kamen die Programme NDepend [Ndep14] in Version 5.1.0 und das Testframework SonarQube [Sona14] (Version 4.1 mit C# Plugins in Version 2.1) zum Einsatz. Beide haben mit ihren Standardeinstellungen bezüglich des Analyseumfangs und der automatischen Bewertung Regelverletzungen festgestellt. Einige dieser Verletzungen stuften die Programme als kritisch ein. Folgende Auflistung enthält Beispiele der kritischen Regelverletzungen:

- SonarQube fand an fünf Stellen auskommentierten Sourcecode und an zwei Stellen Verwendung der Standardausgabe.
- NDepend entdeckte 18 nicht verwendete Klassen und 24 Methoden, die niemals zur Ausführung kommen.
- Beide Programme fanden Verstöße gegen gängige Namenskonventionen.

Es ist dem Autor nicht gelungen SonarQube so zu konfigurieren, dass es für die Klassen die LCOM4 Werte berechnet. NDepend unterstützt die Metrik grundsätzlich nicht. Laut SonarQube beträgt die durchschnittliche zyklomatische Zahl 1,5 pro Methode und NDepend hat zwei Methoden mit einer höheren zyklomatischen Zahl als 10 aufgespürt. Da die zwei Methoden nur einen kleinen Teil der laut SonarQube insgesamt 665 Methoden ausmachen, wurde A24NF in Tabelle 5.1 auf Seite 58 dennoch als erfüllt gekennzeichnet.

Abbildung 5.6 zeigt einen von NDepend erstellten Abhängigkeitsgraphen. In diesem stehen die Knoten für die einzelnen Anwendungskomponenten. Die Views befinden sich in *DiagramEditor*. Eine Pfeilverbindung drückt aus, dass die Komponente, von welcher der Pfeil weg geht, abhängig von der Komponente ist, auf welche der Pfeil zeigt. Aus dem Graphen ist ersichtlich, dass die Realisierung die durch den fachlichen Entwurf in Kapitel 4.3 vorgesehenen Abhängigkeiten der Komponenten untereinander erfüllt.

So sind keine Komponente aus der ViewModel Schicht von Komponenten aus der View Schicht oder DiagramModel von Komponenten aus der ViewModel Schicht abhängig.

Auch sind in der Realisierung wie im Entwurf vorgesehen WorkspaceViewModel und DiagramModel voneinander entkoppelt.

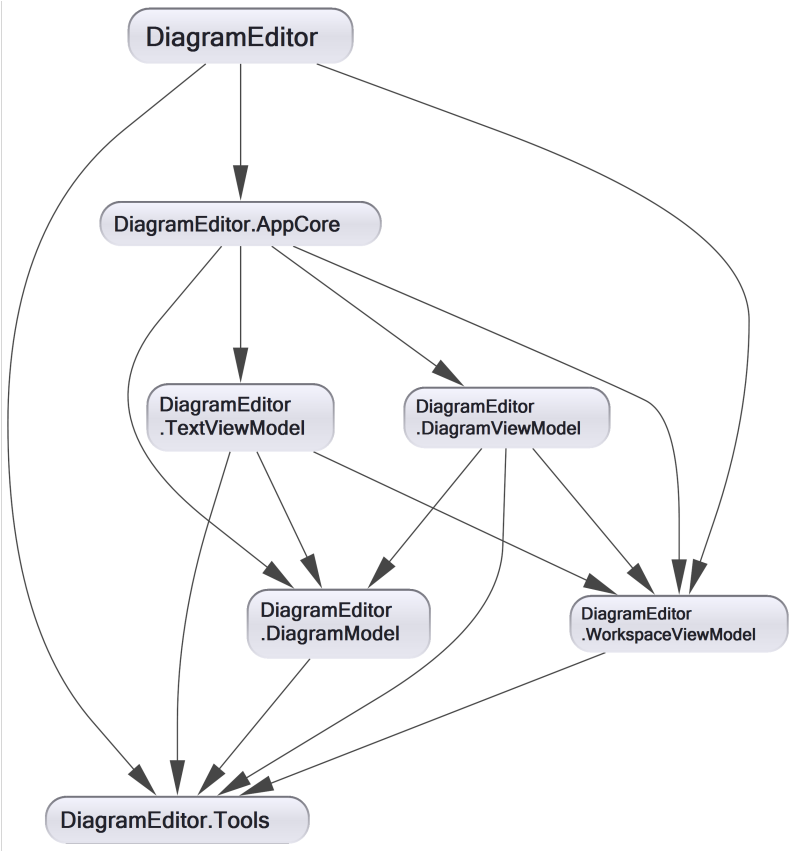

Abbildung 5.6: berechneter Abhängigkeitsgraph

5.3.4 Testbewertung

Dieses Kapitel nimmt eine Bewertung der durchgeführten Tests durch. Ziel ist es eine Beurteilung zu unterstützten, welcher Aufwand noch nötig wäre, um die erfolgte Realisierung oder Teile von ihr in einem Produktivsystem erfolgreich einzusetzen.

Das in den vorangegangenen Kapiteln beschriebene Vorgehen bei den Tests reicht von seinem Umfang her keineswegs aus, um eine ausreichende Qualitätssicherung zu gewährleisten.

Zunächst stellt der Umfang der dokumentierten Spezifikation ein Problem da. Der Theorieteil der Bachelorarbeit spezifiziert den Diagrammeditor nur auf Komponenten-ebene unter Nennung der wichtigsten Klassen und einer kurzen Beschreibung derer Aufgabe. Zwar war eine Anforderung (*A23NF*) alle Schnittstellenmethoden sowie -attribute über Kommentare im Code zu spezifizieren. Aber diese Anforderung ist nach Kapitel 5.2.2 nicht vollständig erfüllt worden. Gegen eine nicht dokumentierte Spezifikation einer Methode oder eines Attributs kann somit nur der Programmierer einen Test entwickeln.

Da der Autor wie bei einer Bachelorarbeit durchaus üblich in Personalunion Programmierer und gleichzeitig der Testentwickler war, relativiert sich dieser Umstand im Rahmen der Arbeit zwar, aber ein grundsätzliches Problem bleibt bestehen. Denn es gibt die Grundregel bezüglich des Testens von Software, dass wenn möglich ein Programmierer nicht seinen eigenen Sourcecode testen sollte, da er ungewollt voreingenommen ist und fehlerhafte Gedankengänge bei der Entwicklung der Tests wiederholen kann.

Zwar bedeutet dies nicht, dass im Rahmen der Entwicklung mit nur einem Programmierer gar nicht getestet werden sollte, aber der Autor hat es vorgezogen, die für die Arbeit zur Verfügung stehenden Entwicklerressourcen eher in die Realisierung zu stecken als in Tests, die kaum Fehler finden und deren Qualität als gering zu bewerten ist.

So decken die Komponententests nur einen Teil der WorkspaceViewModel Komponente ab, auf Integrationstest wurde komplett verzichtet und der Systemtest beschränkt sich nur auf einfache Anwendungsfälle. Letzteres resultiert auch daraus, dass die zum Zeitpunkt des Testens fertige Realisierung keine aufwendigen Diagramme mit einer Vielzahl verschiedener Knoten und vor allem Verbindungen zwischen diesen unterstützt.

Die statische Analyse hat gezeigt, dass die entworfene Grundstruktur durch die erfolgte Realisierung berücksichtigt worden ist. Darüber hinaus konnten die Analysewerkzeuge viele Verstöße gegen allgemeine Programmierrichtlinien feststellen. Einige dieser Mängel wie zum Beispiel auskommentierter Sourcecode oder nicht verwendete Klassen sind auf den Prototypenstatus der Realisierung zurückzuführen. Verstöße gegen XAML und C# hat der Autor oft aus Unerfahrenheit im Umgang mit diesen Sprachen begangen. Letztendlich hängen die Ergebnisse einer statischen Analyse aber auch stark von dem gewählten Werkzeug und seiner Konfiguration ab.

6 Bewertung

In diesem Kapitel bewertet der Autor rückblickend die vorliegende Arbeit zunächst in einem Unterkapitel. Ein weiteres Unterkapitel bewertet dann ein paar im Rahmen der Arbeit angewandte Methodiken bezüglich ihrer allgemeinen Eignung.

6.1 Bewertung der Arbeit bezüglich ihrer Zielsetzung

Kapitel 1.2 hat einige Ziele für diese Arbeit gesetzt. Dieses Kapitel erörtert inwieweit diese erreicht worden sind.

Über die Benutzerschnittstelle der Realisierung können mehrere Nutzer zur gleichen Zeit virtuelle Elemente auf dem Bildschirm eines Tabletop PCs platzieren. Dank des in dieser Arbeit entwickelten Boxensystems (siehe Kapitel 4.2.2 und 5.1.3) erfolgt die Erstellung sowie das Löschen einzelner Elemente meist über direkte Eingabe. Diese wiederum hilft bei der Vermeidung versehentlicher Konflikte (siehe Kapitel 2.2.1).

Bei Aktionen, welche den gesamten Bildschirm betreffen, wie zum Beispiel das Drehen des gesamten Diagramms, lässt sich nicht vermeiden, dass alle Nutzer betroffen sind. Um zur Konfliktvermeidung in diesem Zusammenhang beizutragen, erlaubt der Editor nur jeweils eine Instanz von Boxen, welche Aktionen mit globalen Effekten haben (zum Beispiel die Clipping Box, siehe Kapitel 5.2.1). Die Nutzer können auf Grund einer flexiblen Gestaltung des Boxensystems entscheiden, ob sie die entsprechenden Boxen in der Mitte des Displays und damit gut für alle sichtbar oder etwa in der Nähe einer Person in der Rolle eines Moderators platzieren. Somit trägt der Editor zur Konfliktvermeidung bei, indem er wenige wichtige Regeln zur Organisation der Arbeitsfläche einführt und durchsetzt.

Eine effiziente Vermeidung sowie Lösung ungewollter Konflikte ist ein wesentlicher Aspekt des kollaborativen Arbeitens. Zwar liegt das Auflösen von Konflikten ganz in der Verantwortung der Nutzer, aber die realisierte Benutzerschnittstelle trägt durch das Setzen auf direkte Eingabemethoden und das gezielte Einschränken der Flexibilität der Nutzer zur Vermeidung von Konflikten bei.

Bei dem für die Arbeit eingesetzten Microsoft Pixelsense Tabletop PC führt nicht wirklich ein Weg an der Tastatur (siehe Kapitel 2.3.4), die eine indirekte Eingabemethode darstellt, zur Texterstellung vorbei. Das realisierte Textsnippetsystem (siehe Kapitel 4.2.5) reduziert durch eine Mischung von indirekter und direkter Eingabe bei der Texterstellung die negativen Effekte indirekter Eingabe und fördert den Einsatz von Arbeitsteilung zwischen den Nutzern. Diese besteht aus Phasen des individuellen und Phasen des gemeinsamen Arbeitens, was ein weiterer Aspekt von Kollaboration (siehe Kapitel 2.1.4) ist.

Indem die Realisierung eine Drehung der Ansicht auf die Daten erlaubt, begegnet sie dem Problem, dass auf verschiedene Seiten eines Tabletop PCs verteilte Nutzer einen anderen Blickwinkel auf die Daten haben (siehe Kapitel 2.1.3).

Ebenfalls wichtig für produktives kollaboratives Arbeiten ist die Einteilung der Arbeitsfläche (siehe Kapitel 2.1.1). Diese unterstützt die realisierte Benutzerschnittstelle über die flexibel platzierbaren Boxen, von denen bestimmte Lagerflächen für virtuelle

Elemente darstellen.

Zusammenfassend lässt sich feststellen, dass die Realisierung die gegebene Tabletop PC Technik nutzt, um Kollaboration zu unterstützen. Dies war nach Kapitel 1.2 der Schwerpunkt der Arbeit.

Des Weiteren sollte der zu entwickelnde Editor Diagramme bearbeiten können, denen ein Graph zugrunde liegt. Zwar können die Nutzer bebilderte Knoten und Text erstellen, aber es ist nur ein Knoten realisiert worden und die Möglichkeit der Verbindungserstellung zwischen den Knoten fehlt in der Realisierung. Auch lassen sich die erstellten Daten weder abspeichern und laden noch exportieren. Folglich ist Realisierung kein vollständiger Diagrammeditor und stellt einen frühen Prototypen für die kollaborative Benutzerschnittstelle dar.

Der Entwurf der unfertigen Realisierung dagegen sieht einen vollständigen Diagrammeditor vor und beschreibt die benötigte Benutzerschnittstelle in Kapitel 4.2.3 und die entsprechenden Architekturkomponenten unter anderem in den Kapiteln 4.3.2 sowie 4.3.4. Für das Speichern und Laden der Diagramme enthält der Entwurf die Verwendung des NShape Frameworks (siehe Kapitel 4.4.5).

Abschließend steht das Fazit, dass diese Arbeit bezüglich der Realisierung zwar nur einen Teil, aber bezüglich des Entwurfs alle Ziele erreicht hat. Es wäre falsch, die vorliegende Arbeit nur anhand ihrer Realisierung zu bewerten, da diese nur den Abschluss des Entwicklungsprozesses darstellt. Dieser jedoch als Ganzes veröffentlicht und damit anderen Entwicklern von Nutzen sein kann.

6.2 Methodische Abstraktion

Ein paar Methoden insbesondere der Informatik hatten einen wesentlichen Einfluss auf diese Arbeit. Ziel dieses Kapitel ist es diese Methoden unabhängig von der Arbeit zu beschreiben und eine allgemeingültige Bewertung von ihnen vorzunehmen:

- Bei einer auf **Gesten** basierten Benutzerschnittstelle, erfasst und interpretiert eine Anwendung die Körperbewegungen (Gesten) seiner Nutzer. Entsprechend der Interpretation reagiert es dann. Somit ist es möglich, dass einem Benutzer die Interaktion mit der Anwendung natürlich vorkommt, da er die Kommunikation über Gesten von seinem Alltag her kennt.

  Damit den Nutzern die Bedienung der Anwendung über Gesten aber auch leicht fällt, müssen die Gesten leicht ausführbar sein und insbesondere muss einer Geste folgenden Reaktion der Anwendung dem Benutzer von seiner Lebenserfahrung her logisch erscheinen. Das Finden solcher Gesten ist aufwendig und erfordert viel Forschung.

  Ein weiterer Vorteil von Gesten basierter Benutzereingabe ist, dass sie Zeigegeräte wie zum Beispiel eine Computermaus ersetzen und eine direkte Eingabe darstellen kann, welche weniger Umdenken erfordert. Allerdings können spezielle Eingabegeräte eine höhere Genauigkeit anbieten als die direkte Eingabe über Gesten.

- Hinter den Textsnippets (siehe Kapitel 4.4.1) steckt der Grundgedanke **eine Aufgabe in Unteraufgaben aufzuteilen** wie zum Beispiel in die eigentliche Texterstellung über eine Tastatur irgendwo gefolgt vom Transport zum Zielort.

  Bei einer kollaborativen Anwendung erlaubt dies den Nutzern, sich auf bestimmte Unteraufgaben zu spezialisieren und dabei die für sie vorgesehenen Werkzeuge zu verwenden. Dies kommt den Nutzern besonders entgegen, wenn zum Beispiel aus technischen Gründen bestimmte Werkzeuge nur in einer begrenzten Anzahl zur Verfügung gestellt werden können oder nicht überall auf der Arbeitsfläche einsetzbar sind.

  Um die Gesamtaufgabe beenden zu können, müssen die einzelnen Akteure sich zwischen dem Ausführen ihrer Einzelaufgaben koordinieren. Dadurch kann es zwar zu Konflikten kommen, die aber wiederum die Zusammenarbeit stärken können. Problematisch kann es werden, wenn eine Teilaufgabe deutlich länger als die nachfolgenden Unteraufgaben dauert. Dadurch kann es vorkommen, dass für spätere Unteraufgaben Verantwortliche warten müssen, wenn die Aufgabenaufteilung unter den Benutzern nicht optimal ist.

  Ein weiteres Problem ist, dass möglicherweise durch die zusätzliche benötigte Kommunikation die Erfüllung der Gesamtaufgabe durch ihre Unterteilung mehr Zeit in Anspruch nimmt.

  Aus Entwicklersicht ist zu beachten, dass durch bewusste Unterteilung von Aufgaben sich die Anzahl der unterschiedlichen Werkzeugarten erhöhen kann.

- Bei einer auf **Schichten** basierten Architektur wird die Anwendungslogik auf unterschiedliche Schichten verteilt. Dabei bauen die Schichten aufeinander auf und jede der Schichten hat immer nur Detailwissen über die andere Schicht, von der sie direkt abhängig ist.

  Durch die Schichten ist die klare Strukturierung einer Anwendung möglich, was die Wartbarkeit erhöht. In einem Team mit mehreren Entwicklern, kann eine Aufgabenteilung nach Schichten statt finden.

  Wenn eine Unterteilung in vom System abhängige und unabhängige Schichten stattfindet, führt eine geschichtete Architektur zu einer Verbesserung der Portabilität der gesamten Anwendung, da bei einer Übertragung auf ein anderes System nur die vom System abhängigen Schichten angepasst beziehungsweise ausgetauscht werden müssen.

  Da beim Austausch von Daten zwischen zwei Schichten diese von allen Zwischenschichten durch gereicht werden müssen, wird die Effizienz der Anwendung gerade bei vielen Schichten verringert.

  Das benötigte Durchreichen von Daten erhöht außerdem die Menge an redundantem Sourcecode.

- Bei der **Koppelung über Databinding** stellen zwei miteinander gekoppelte Anwendungskomponenten jeweils öffentliche Attribute bereit und das Databinding als Teil der Laufzeitumgebung gleicht unter der Verwendung von Konvertern die Attribute der einen Komponente mit denen der anderen ab. Somit sind die Komponenten nur lose miteinander gekoppelt.

  Dies hat den Vorteil, dass beide Komponenten mit unterschiedlichen Werkzeugen (zum Beispiel in verschiedenen Programmiersprachen) realisiert werden können, was besonders nützlich bei der Aufgabenverteilung in einem Entwicklerteam ist.

  Weiter erlaubt die lose Koppelung einen einfachen Austausch der Komponenten und das sogar zur Laufzeit.

  Wenn die Koppelung über das Databinding allerdings erst zur Laufzeit statt findet, können Fehler wie zum Beispiel das Fehlen von passenden Konvertern oder Attributen auch erst dann festgestellt werden.

  Um den Vorteil der losen Koppelung zu erhalten, kennt das Databinding System nicht den Einsatzzweck der ausgetauschten Daten und kann die Synchronization nicht dahingehend optimieren.

- Bei der Anbindung eines **Frameworks über eine Fassade** an eine Anwendung, wird die Anwendung gegen die Schnittstellen der Fassade entwickelt und die Fassade realisiert die Schnittstellen dann unter Verwendung des Frameworks. Die Fassade entkoppelt also das Framework von der restlichen Anwendung.

  Ein Vorteil dieses Vorgehens ist, dass die Fassade die Verwendung des Frameworks für die Restanwendung vereinfachen und anwendungsspezifischer gestalten kann.

  Wenn das Framework bestimmte Merkmale wie zum Beispiel threadsicheren Zugriff nicht aufweist, kann die Fassade diese nachrüsten und das Fehlen somit für die Restanwendung transparent machen.

  Falls das Framework durch ein Vergleichbares zum Beispiel im Rahmen einer Portierung der Anwendung ausgetauscht wird, muss nur die Fassade und nicht der Rest der Anwendung angepasst werden. Somit kann ein Projekt die Abhängigkeiten von einem Framework reduzieren. Dies kann zum Beispiel auf Grund Lizenz technischer Gründe von Nutzen sein.

  Falls die Schnittstellen der Fassade zu einfach gestaltet sind, können allerdings bestimmte Fähigkeiten des Frameworks nur unvollständig ausgenutzt werden.

  Weitergehend gestaltet sich die Realisierung der Fassade sehr aufwendig, wenn die von ihr gebotenen Schnittstellen semantisch nicht zu denen des Frameworks passen.

7 Zusammenfassung und Ausblick

Entsprechend der Zielsetzung in der Einleitung fasst das zweite Kapitel Recherche-ergebnisse über Gesten basierte Eingabe, kollaboratives Arbeiten an Tabletop PCs, das Microsoft Pixelsense und das NShape Diagrammframework zusammen.

Das dritte Kapitel formuliert an Hand der Recherergebnisse sowie Grundlagenwissen des Autors funktionale, nicht funktionale und technische Anforderungen,

Das vierte Kapitel dokumentiert einen gegen die Anforderungen und unter Berück-sichtigung des zweiten Kapitels entwickelten Entwurf. Dieser sieht eine Benutzer-schnittstelle vor, die über berührungsbasierte Gesten auf dem Bildschirm bedient wird. Der Bildschirm zeigt einen veränderbaren Ausschnitt einer virtuellen Arbeitsfläche an. Diese sieht neben den virtuellen Diagrammelementen Boxen vor, die Ressourcen sowie Werkzeuge aufnehmen können.

Die fachliche Architektur richtet sich nach dem Model View ViewModel (MVVM) Softwarepattern und trennt die Logik für die virtuelle Arbeitsfläche sowie dem Boxensystem von der Diagrammlogik. Der technische Entwurf sieht schließlich eine Umsetzung der Anwendung auf der Microsoft Pixel Sense Plattform in erster wie zweiter Generation unter Verwendung des Microsoft Surface SDK und des NShape Diagrammframeworks vor.

Das fünfte Kapitel beschreibt den Status der Realisierung sowie ihre Abweichungen vom Entwurf und die an ihr durchgeführten Tests. Es konnten wesentliche Teile der virtuellen Arbeitsfläche, der Boxen sowie Werkzeuge zur Texterstellung realisiert werden. Die Realisierung der Diagrammbearbeitung steht erst am Anfang. Die Tests zur Verifikation der Realisierung fallen rudimentär und knapp aus.

Das sechste Kapitel enthält eine Bewertung der Arbeit bezüglich erfüllter Zielsetzung und nimmt eine methodische Abstraktion vor.

Die bisher verfügbaren Tabletop PC Systeme sind vor allem preislich noch weit davon entfernt wirkliche persönliche Computer zu werden und eine weite Verbreitung zu erfahren. Auf der anderen Seite soll sich der Diagrammeditor durch Einfachheit und einer Unterstützung vielfältiger graphischer Elemente an die Breite Masse richten. Daher hält es der Autor nicht für sinnvoll die Realisierung von ihm zu Ende zu führen.

Allerdings hat die Arbeit einige von der Diagrammbearbeitung unabhängige Erfahrungen und Ideen zu den Themen Tabletop PC und kollaboratives Arbeiten hervorgebracht, bei denen sich eine weitere und genauere Erforschung durchaus lohnen könnte.

A1 Systemtest

Nachfolgende Tabelle beschreibt die einzelnen Schritte des Tests. Die Schritte bauen aufeinander auf und sind daher in der angegebenen Reihenfolge auszuführen.

Die Vorbedingung für den Test ist, dass die Anwendung zusammen mit der Microsoft Surface Shell bereits auf dem Tabletop PC gestartet ist und es sich um ein leeres Diagramm handelt. Ein Tester genügt und dessen Positionierung zum Tisch ist irrelevant.

Nach der Testbeschreibung folgen noch die Testprotokolle.

Schritt	Testanweisungen	Erwarteter Effekt
ST1	Ziehen der initialen Ressourcenbox über die Titelleiste mit mehreren Fingern	Box bewegt sich analog zu den Fingern
ST2	Berühren des linken oberen und des rechten unteren Eckpunkts mit jeweils einem Finger und die Finger bewegen.	Eckpunkte bleiben stets unter den Fingerspitzen. Box verschiebt, vergrößert und dreht sich entsprechend.
ST3	Finger auf den Repräsentanten der Löschbox (*DeleteBox* in der initialen Ressourcenbox) legen und auf die Arbeitsfläche ziehen; Vor dem Loslassen das Bild des Repräsentanten mit zweitem Finger vergrößern sowie drehen	Das Bild des Repräsentant befindet sich stets unter den Fingerspitzen und vergrößert sowie dreht sich analog zu deren Abstand und Ausrichtung zueinander. Nach Loslassen des Bilds wird dieses durch eine Löschbox ersetzt, welche die letzte Position, Drehung und Größe des Bilds aufweist.
ST4	Ziehen eines Knotenrepräsentanten auf die Arbeitsfläche	Nach Loslassen des Repräsentantenbildes entsteht ein Knoten, dessen Mittelpunkt dem Bildmittelpunkt und Größe sowie Ausrichtung der Beschreibung des Repräsentanten in der Ressourcenbox entspricht.
ST5	Mindestens zwei Fingerspitzen auf den zuvor erstellten Knoten legen und auseinander ziehen	Knoten vergrößert und dreht sich entsprechend der Bewegung der Fingerspitzen
ST6	Ziehen des Knotens auf die Löschbox; Dabei muss die Löschbox größer als der Knoten sein!	Knoten verschwindet von der Arbeitsfläche
ST7	Erstellen einer Kopierbox (*CopyBox*) sowie einer Ressourcenbox (*MoveBox*) über die initiale Ressourcenbox	Boxen werden wie zuvor die Löschbox erstellt
ST8	Ziehen eines Knotenrepräsentanten von der initialen Ressourcenbox in	Zielbox nimmt den Repräsentanten in ihre Liste mit der Anzahl 1 auf

	die zuvor erstellte MoveBox	
ST9	Ziehen des zuvor in MoveBox gezogenen Repräsentanten in die Kopierbox	Sowohl MoveBox als auch Kopierbox enthalten den Repräsentanten mit der Anzahl 1
ST10	Ziehen des Repräsentanten aus der Kopierbox auf die Löschbox	Kopierbox ist leer
ST11	Erstellung eines neuen Knotens auf der Arbeitsfläche über den Repräsentanten aus der MoveBox	Knoten entsteht auf der Arbeitsfläche und MoveBox ist leer
ST12	Knoten verändern und auf die Kopierbox ziehen	Knoten bleibt auf der Arbeitsfläche mit der Position, Ausrichtung sowie Größe wie zu Beginn des Ziehens und die Kopierbox enthält einen Repräsentanten, der dem Knoten entspricht.
ST13	Knoten erneut verändern und wieder auf die Kopierbox ziehen	Knoten bleibt erneut auf der Arbeitsfläche und die Kopierbox weist einen neuen dem Knoten entsprechenden Repräsentanten auf.
ST14	Knoten nicht mehr verändern und auf die MoveBox ziehen	Knoten verschwindet von der Arbeitsfläche und MoveBox enthält einen Repräsentanten, der dem Knoten zu Beginn des Ziehens entspricht
ST15	Repräsentanten des Knotens aus der MoveBox auf die Kopierbox ziehen	Die Kopierbox weist einen Repräsentanten mit der Anzahl 2 und einen mit der Anzahl 1 auf und die MoveBox ist leer.
ST16	Ziehen der MoveBox auf die Löschbox; Die Löschbox muss größer als die MoveBox sein	Die MoveBox verschwindet vom Bildschirm.
ST17	Knoten erstellen und angezeigten Bildausschnitt über die „Clipping Control“ Box verschieben	Knoten wird relativ zu den Boxen verschoben. Deren Position und Ausrichtung sowie Größe aller virtueller Elemente bleiben unverändert.
ST18	Bildausschnitt vergrößern	Boxen bleiben unverändert und der Knoten wird kleiner.
ST19	Bildausschnitt verkleinern	Boxen bleiben unverändert und der Knoten wird größer.
ST20	Bildausschnitt im Uhrzeigersinn drehen	Boxen bleiben unverändert und der Knoten wird um 45 Grad um den Bildschirmmittelpunkt gedreht.
ST21	Auf Knoten tippen	Der Knoten expandiert.
ST22	Erneut auf Knoten tippen	Knoten nimmt wieder normalen Zustand an.
ST23	„Text Creator“ Box und ein	Arbeitsfläche enthält eine Box zur

	Textelement über die initiale Ressourcenbox erstellen	Texterstellung und einen Textplatzhalter. Die Erstellung von letzterem gleicht der eines Knotens. Zum Beispiel entspricht seine Ausrichtung der Repräsentantenbeschreibung.
ST24	Tippen auf das Textfeld der „Text Creator“ Box	Virtuelle Tastatur sollte erscheinen
ST25	Tippen des Textes „Dies ist ein Test“ und Betätigen des „Create“ Buttons der „Text Creator“ Box	Am rechten unteren Ende der Box entsteht ein Textsnippet mit dem eingegebenen Text.
ST26	Ziehen des Textsnippets auf den zuvor erstellten Textplatzhalter	Der Platzhalter enthält den Text „Dies ist ein Test“
ST27	Tippen auf den Platzhalter	Der Textplatzhalter expandiert und zeigt Einstellungsmöglichkeiten für die Texteigenschaften.
ST28	Einstellen der Texteigenschaften auf fett, kursiv und Verändern der Schriftgröße	„Dies ist ein Test“ nimmt die eingestellten Eigenschaften an.
ST29	Erneutes Tippen auf Textplatzhalter	Der Textplatzhalter nimmt wieder die normale Größe an und die Texteigenschaften bleiben erhalten.
ST30	Ziehen des Textplatzhalters auf die Kopierbox	Die Kopierbox vermerkt einen zum Platzhalter passenden Repräsentanten.
ST31	Ziehen des Repräsentanten aus der Kopierbox auf die Arbeitsfläche	Kopie des Textplatzhalters mit Standardtext und an neuer Position entsteht und Repräsentant verschwindet aus Kopierbox.
ST32	Erstellen eines neuen Textsnippets mit dem Inhalt „Ein weiterer Text Test“; Ziehen des neuen Textsnippets in die Kopierbox.	Textsnippet verbleibt auf Arbeitsfläche und Kopierbox enthält ein dem Textsnippet entsprechenden Repräsentanten.
ST33	Ziehen des Repräsentanten des Textsnippets auf die Arbeitsfläche	Kopie des Textsnippets entsteht an der Zielposition mit dem Inhalt „Ein weiterer Text Test“
ST34	Aufräumen; Dies bedeutet, dass alle virtuellen Elemente bis auf die virtuelle Tastatur auf die Löschbox gezogen werden; Dabei ist die Löschbox größer als jedes andere Element	Die initiale Ressourcenbox und die „Clipping Control Box“ lassen sich nicht entfernen und verbleiben mit der Löschbox als einzige Elemente auf der Arbeitsfläche.

Testprotokoll vom 20.12.2013

Ausführender Tester: Lukas Grundmann

Durchgeführte Testschritte: ST1 bis ST34

Festgestellte Fehler:

- Textplatzhalter passen sich wohl in ihrer Größe automatisch an den angezeigten Text an. Dieses Verhalten ist aber nicht erwünscht, da die Größe des Platzhalters beschränkend auf den angezeigten Text wirken soll.
- Werden Repräsentanten nach dem Wählen mit dem Finger nicht gezogen und dann Losgelassen werden sie nicht wieder von der Ausgangsbox aufgenommen wie es zu erwarten wäre, sondern Verschwinden einfach unter der Box. Das Wegziehen der Box offenbart dann die entsprechenden Repräsentanten, die sich dann weiter wie spezifiziert verarbeiten lassen.
- Gelegentliche Abstürze der Anwendung auf Grund der Registrierung einer ungültigen Touch ID beim Microsoft Surface SDK (laut Exception Text).

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[BeWB06] BENKO, HRVOJE ; WILSON, ANDREW D. ; BAUDISCH, PATRICK: Precise selection techniques for multi-touch screens. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '06*. New York, NY, USA : ACM, 2006 — ISBN 1-59593-372-7, S. 1263–1272

[Bund13] BUNDESZENTRALE FÜR POLITISCHE BILDUNG: *Viele Verlierer, eine Siegerin | bpb*. URL http://www.bpb.de/politik/hintergrund-aktuell/169417/bundestagswahl-2013. - abgerufen am 2014-02-07

[Data12] DATAWEB: *NShape Handbuch*. URL https://code.google.com/p/nshape/downloads/list. - abgerufen am 2013-06-10

[Data14] DATAWEB: *NShape - .Net Diagramming Framework for Industrial Applications - Home*. URL http://nshape.codeplex.com/. - abgerufen am 2014-02-08

[Gamm95] GAMMA, ERICH; HELM, RICHARD; RALPH, JOHNSON; VLISSIDES, JOHN ; KERNIGHAN, B. W. (Hrsg.): *Design Patterns - Elements of Reusable Object-Oriented Software*, *Addison-Wesley Professional Computing Series*. Reading, Massachusetts : Addison Wesley Publishing Company, 1995

[GuGr98] GUTWIN, CARL ; GREENBERG, SAUL: Design for individuals, design for groups: tradeoffs between power and workspace awareness. In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, *CSCW '98*. New York, NY, USA : ACM, 1998 — ISBN 1-58113-009-0, S. 207–216

[HCVW06] HANCOCK, MARK S. ; CARPENDALE, SHEELAGH ; VERNIER, FREDERIC D. ; WIGDOR, DANIEL ; SHEN, CHIA: Rotation and Translation Mechanisms for Tabletop Interaction. In: *Proceedings of the First IEEE International Workshop on Horizontal Interactive Human-Computer Systems*, *TABLETOP '06*. Washington, DC, USA : IEEE Computer Society, 2006 — ISBN 0-7695-2494-X, S. 79–88

[HiCa11] HINRICHS, UTA ; CARPENDALE, SHEELAGH: Gestures in the wild: studying multi-touch gesture sequences on interactive tabletop exhibits. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '11*. New York, NY, USA : ACM, 2011 — ISBN 978-1-4503-0228-9, S. 3023–3032

[HoBJ08] HOGGAN, EVE ; BREWSTER, STEPHEN A. ; JOHNSTON, JODY: Investigating the effectiveness of tactile feedback for mobile touchscreens. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '08*. New York, NY, USA : ACM, 2008 — ISBN 978-1-60558-011-1, S. 1573–1582

[Kila12] KILAN, DENNIS: *SurfMo - Freihand Modellierung auf Microsofts Surface*.

Berliner Tor 5, 20099 Hamburg, HAW Hamburg, 2012

[MaKG10] MARQUARDT, NICOLAI ; KIEMER, JOHANNES ; GREENBERG, SAUL: What caused that touch?: expressive interaction with a surface through fiduciary-tagged gloves. In: *ACM International Conference on Interactive Tabletops and Surfaces*, *ITS '10*. New York, NY, USA : ACM, 2010 — ISBN 978-1-4503-0399-6, S. 139–142

[MHPW06] MORRIS, MEREDITH RINGEL ; HUANG, ANQI ; PAEPCKE, ANDREAS ; WINOGRAD, TERRY: Cooperative gestures: multi-user gestural interactions for co-located groupware. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '06*. New York, NY, USA : ACM, 2006 — ISBN 1-59593-372-7, S. 1201–1210

[Micr00a] MICROSOFT: *Gestures: Touch and Multitouch*. URL http://technet.microsoft.com/en-us/library/ee692057(v=surface.10).aspx. - abgerufen am 2013-05-12

[Micr00b] MICROSOFT: *Tagged Objects*. URL http://technet.microsoft.com/en-us/library/ee692094(v=surface.10).aspx. - abgerufen am 2013-05-12

[Micr00c] MICROSOFT: *The Power of PixelSense™*. URL http://www.microsoft.com/en-us/pixelsense/pixelsense.aspx. - abgerufen am 2013-05-12

[Micr00d] MICROSOFT CORPORATION: *Microsoft Surface Architecture*. URL http://msdn.microsoft.com/en-us/library/ee804755(v=surface.10).aspx. - abgerufen am 2013-05-13

[Micr00e] MICROSOFT CORPORATION: *Requirements*. URL http://msdn.microsoft.com/en-us/library/gg697158.aspx. - abgerufen am 2013-07-01

[Micr00f] MICROSOFT CORPORATION: *Übersicht über Routingereignisse*. URL http://msdn.microsoft.com/de-de/library/ms742806(v=vs.90).aspx. - abgerufen am 2014-01-07

[Micr00g] MICR: *Styling and Templating*. URL http://msdn.microsoft.com/en-us/library/ms745683(v=vs.110).aspx. - abgerufen am 2014-01-17

[Micr00h] MICROSOFT: *Physical Features of a Microsoft Surface Unit*. URL http://technet.microsoft.com/en-us/library/ee692114.aspx. - abgerufen am 2013-05-12

[Micr00i] MICROSOFT CORPORATION: *IC353813.jpg (JPEG-Grafik, 276 × 176 Pixel)*. URL http://i.technet.microsoft.com/dynimg/IC353813.jpg. - abgerufen am 2013-05-06

[Micr11] MICROSOFT CORPORATION: Microsoft® Surface® 2 Design and Interaction Guide (2011)

[Micr13] MICROSOFT CORPORATION: *ScatterView Control*. URL http://msdn.microsoft.com/en-us/library/ff727729.aspx. - abgerufen am 2013-07-03

[MRSF04] MORRIS, MEREDITH RINGEL ; RYALL, KATHY ; SHEN, CHIA ; FORLINES, CLIFTON ; VERNIER, FREDERIC: Beyond „social protocols“: multi-user coordination policies for co-located groupware. In: *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, *CSCW '04*. New York, NY, USA : ACM, 2004 — ISBN 1-58113-810-5, S. 262–265

[MüFj10] MÜLLER-TOMFELDE, CHRISTIAN ; FJELD, MORTEN: Introduction: A Short History of Tabletop Research, Technologies, and Products. In: MÜLLER-TOMFELDE, C. (Hrsg.): *Tabletops - Horizontal Interactive Displays*, *Human-Computer Interaction Series* : Springer London, 2010 — ISBN 978-1-84996-112-7, 978-1-84996-113-4, S. 1–24

[Ndep14] NDEPEND: *NDepend | Achieve higher .NET code quality with ndepend*. URL http://www.ndepend.com/. - abgerufen am 2014-02-11

[OlFH08] OLWAL, ALEX ; FEINER, STEVEN ; HEYMAN, SUSANNA: Rubbing and tapping for precise and rapid selection on touch-screen displays. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '08*. New York, NY, USA : ACM, 2008 — ISBN 978-1-60558-011-1, S. 295–304

[Pohm09] POHMANN, PETER: *NShape - An open source diagramming framework for .NET WinForms*. URL https://nshape.googlecode.com/files/NShapeWhitePaper02.pdf. - abgerufen am 2013-07-01

[RWZB10] REMY, CHRISTIAN ; WEISS, MALTE ; ZIEFLE, MARTINA ; BORCHERS, JAN: A pattern language for interactive tabletops in collaborative workspaces. In: *Proceedings of the 15th European Conference on Pattern Languages of Programs*, *EuroPLoP '10*. New York, NY, USA : ACM, 2010 — ISBN 978-1-4503-0259-3, S. 9:1–9:48

[Sams13] SAMSUNG: *Samsung LFD Monitors*. URL http://www.samsunglfd.com/product/spec.do?modelCd=SUR40&specType=FULL. - abgerufen am 2013-05-12

[ScBG09] SCHMIDT, DOMINIK ; BLOCK, FLORIAN ; GELLERSEN, HANS: A Comparison of Direct and Indirect Multi-touch Input for Large Surfaces. In: *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part I*, *INTERACT '09*. Berlin, Heidelberg : Springer-Verlag, 2009 — ISBN 978-3-642-03654-5, S. 582–594

[Smit09] SMITH, JOSH: *DAS MODEL-VIEW-VIEWMODEL (MVVM)-ENTWURFSMUSTER FÜR WPF*. URL http://msdn.microsoft.com/de-de/magazine/dd419663.aspx. - abgerufen am 2013-07-03

[Sona14] SONARQUBE: *SonarQube™*. URL http://www.sonarqube.org/. - abgerufen am 2014-02-11

[SSCI04] SCOTT, STACEY D. ; SHEELAGH, M. ; CARPENDALE, T. ; INKPEN, KORI M.: Territoriality in collaborative tabletop workspaces. In: *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, *CSCW '04*. New York, NY, USA : ACM, 2004 — ISBN 1-58113-810-5, S. 294–303

[Thev00] THEVERGE: *Microsoft rebrands original Surface table-based touchscreen as PixelSense*. URL http://www.theverge.com/2012/6/19/3096652/microsoft-surface-table-rebranded-as-pixelsense. - abgerufen am 2013-05-06. — The Verge

[TTPN06] TANG, ANTHONY ; TORY, MELANIE ; PO, BARRY ; NEUMANN, PETRA ; CARPENDALE, SHEELAGH: Collaborative coupling over tabletop displays. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '06*. New York, NY, USA : ACM, 2006 — ISBN 1-59593-372-7, S. 1181–1190

[WiBa05] WIGDOR, DANIEL ; BALAKRISHNAN, RAVIN: Empirical investigation into the effect of orientation on text readability in tabletop displays. In: *Proceedings of the ninth conference on European Conference on Computer Supported Cooperative Work*, *ECSCW'05*. New York, NY, USA : Springer-Verlag New York, Inc., 2005 — ISBN 978-1402040221, S. 205–224

[Wiki13] WIKIPEDIA: Business Process Model and Notation. *Wikipedia*.

[Wiki14] WIKIPEDIA: Kollaboration. *Wikipedia*.

[Witt11] WITTERN, HAUKE: *Konzeption, Entwicklung und Evaluierung einer Geschäftsprozessmodellierungsanwendung für Multitouch Geräte*. [Elektronische Ressource]. Aufl. Hamburg : Hochschule für Angewandte Wissenschaften, 2011

[WoMW09] WOBBROCK, JACOB O. ; MORRIS, MEREDITH RINGEL ; WILSON, ANDREW D.: User-defined gestures for surface computing. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *CHI '09*. New York, NY, USA : ACM, 2009 — ISBN 978-1-60558-246-7, S. 1083–1092

[Zdne11] ZDNET: *microsoft_surface2-v6.jpg (JPEG-Grafik, 580 × 400 Pixel)*. URL http://www.zdnet.de/wp-content/uploads/legacy_images/news/2011/01/microsoft_surface2-v6.jpg. - abgerufen am 2013-05-06

Hiermit versichere ich, dass ich die vorliegende Arbeit nach §16 (5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. Februar 2014 Lukas Grundmann