Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Bastian Karstaedt

WPF Toolkit für interaktive, raumbezogene Anwendungen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Bastian Karstaedt

WPF Toolkit für interaktive, raumbezogene Anwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Birgit Wendholt
Zweitgutachter : Prof. Dr. rer. nat. Kai von Luck

Abgegeben am 14. September 2009

Bastian Karstaedt

Thema der Bachelorarbeit

WPF Toolkit für interaktive, raumbezogene Anwendungen

Stichworte

Interaktives Rauminformationssystem, Gebäudeinformationssystem, Eventmodell, Toolkit, CityGML, WPF, CAD to XAML, SVG

Kurzzusammenfassung

In dieser Arbeit wird ein Toolkit entwickelt, das den Anwendungsentwickler bei der effizienten Realisierung interaktiver, raumbezogener Applikationen unterstützt. Dazu wurden drei Komponenten entworfen: 1. das *AnnotationTool* zum Annotieren von Raumentitäten mit statischen und dynamischen Inhalten, 2. die *ToolkitLib* – eine Bibliothek für die semi-automatische Transformation von 2D CAD Stockwerksdaten in ein semantisches Gebäudemodell und 3. der *WelcomeScreen*, ein Evaluierungsprototyp, der die Verwendung des AnnotationTools und die Unterstützung bei der Entwicklung raumbezogener Anwendungen zeigt.

Die geometrische Repräsentation des an CityGML angelehnten, semantischen Gebäudemodells erfolgt im weit verbreiteten SVG-Standard.

Bastian Karstaedt

Title of the paper

WPF Toolkit for Interactive, Space-Oriented Applications

Keywords

interactive informationsystem, geographical information system, event model, toolkit, CityGML, WPF, CAD to XAML, SVG

Abstract

This work deals with the design and implementation of a toolkit that supports the development of interactive, geospatial applications. For this purpose three components have been developed: 1. the *AnnotationTool* to annotate spatial entities with static and dynamic content, 2. the *ToolkitLib* – a library for semi-automatic conversion of 2D CAD storey data into a semantic building model, and 3. the *WelcomeScreen*, an evaluationprototype that shows the usage of the AnnotationTool and its benefits in developing spatial applications. The geometric representation of the semantic building model, which is based on concepts of CityGML, is realised by using the widely spread SVG standard.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1 Einführung

Die folgenden Kapitel geben dem Leser einen ersten Eindruck über die Thematik und klären grundsätzliche Fragen.

1.1 Motivation

Die Gesellschaft des 21. Jahrhunderts definiert sich weitestgehend als eine Informationsgesellschaft. Im Zuge der rasanten technischen Entwicklungen stehen uns Informationsbestände zu vielen Bereichen des Lebens zur Verfügung – geradezu jederzeit und überall. Sei es der Liedtext für einen Song, die Biografie eines Schauspielers oder die Route zum indischen Restaurant – in wenigen Augenblicken können wir Datenbestände in riesigem Ausmaß nach relevanten Informationen durchsuchen, oder Datenbestände vereinen, um neue Informationen zu gewinnen.

Die Grundlage dieser Datenbestände sind der “Informationsrohstoff” unserer Gesellschaft. Dieser “Rohstoff” muss zunächst erfasst und anschließend aufbereitet werden, um der digitalen Welt in angemessener Form zur Verfügung zu stehen. Der Informationsrohstoff dieser Arbeit sind digitale Karten über Räume und Informationen, die mit diesen Räumen korrelieren.

Bereits frühzeitig haben kommerzielle Anbieter von Straßenkarten damit begonnen, die Erdoberfläche durch Sattelitenbilder zu kartographieren. Uns steht damit ein “digitaler Globus” zur Verfügung (z. B. Google Earth oder Microsoft Virtual Earth), der überlagert mit Geodaten Zugriff auf Informationen wie z. B. Straßenkarten, Ländergrenzen, Points of Interests (POI), 3D Gebäudemodellen uvm. erlaubt. Die *Geoinformationen* wurden erfasst, bearbeitet, geographisch verortet und sind nun weltweit über das Internet verfügbar.

Grundlage hierfür waren *Geographische Informationssysteme* (GIS), die als Anwendungen mit Ortsbezug unter den *Location Based Services* (LBS) eingeordnet werden. Die *Geoinformatik* ist eine vergleichsweise junge wissenschaftliche Disziplin. Sie befasst sich mit dem Wesen und der Funktion der *Geoinformation* (GI), sowie ihrer Verarbeitung in Form von *Geodaten*. Die Anfänge informationstechnischer Nutzung von Raumdaten finden sich bereits in den 50er Jahren, wobei die *Pioniere* der Geoinformatik weder einen umfangreichen Fundus

an digitalen Daten, noch entsprechende Hardware zur Verfügung hatten. Ab Mitte der Siebziger Jahre waren es vorrangig Behörden, die GIS nutzten um *Geobasisdaten* digitalisiert zu verwalten. Ab 1979 spricht Bartelme vom *GIS-Markt* – die Hardware wird leistungsfähiger und Firmen sehen großes Potenzial in GIS. Die Entwicklung der GIS wurde insbesondere auch von speziellen GIS Anwendungen im Bereich “Facility-Management” und der Gebäudeplanung (z. B. bei der Planung zur Verlegung von Leitungen) vorangetrieben.

GIS bieten heutzutage eine Vielzahl an raumbezogenen Anwendungsmöglichkeiten, sei es die Routenberechnung, die Infrastrukturplanung von Stromanbietern oder die Erfassung und Auswertung meteorologischer Phänomene [habil. Jörg Roth, 2009].Diese *Zeit der Nutzer* wurde laut Bartelme [2005] S.10 um 1988 eingeleitet und findet mit den erwähnten internetgestützten GIS ihren derzeitigen Höhepunkt.

Mittlerweile gibt es viele Applikationen für den privaten Gebrauch mit Navigationslösungen für den Außenbereich. Zu den bekanntesten zählen Anwendungen wie Google Maps und mobile Navigationssysteme für Kraftfahrzeuge und Fußgänger wie z. B. von TomTom N.V., die auf mobilen Endgeräten mit GPS Empfänger betrieben werden. Navigationslösungen finden sich für viele Plattformen wie dem Google Android oder Apple iPhone. Außerdem gibt es eine Vielzahl wissenschaftlicher Arbeiten über ortsabhängige Dienste wie “Deep Map” in Malaka und Zipf [2000]. An der Hochschule für Angewandte Wissenschaften Hamburg wurden mitunter GPS gestützte Systeme von Buchholz [2007] oder auch Parlowski [2009] entwickelt.

Das Wechselspiel von leistungsfähigerer Hardware und unternehmerischem Ehrgeiz bringt stets neue Entwicklungen im GIS Bereich hervor. Anhand aktueller Forschungsarbeiten lässt sich ein Trend erkennen, der zum einen auf die Nutzung mobiler Endgeräte für GIS zum anderen auf eine Fokussierung der GIS auf den Innenbereich abzielt.

Zu LBS und Navigationsanwendung für den Innenbereich sind hauptsächlich wissenschaftliche Arbeiten erschienen. Als Beispiele sind hier “Navio” der Technischen Universität Wien (Gartner u. a. [2004]), “Irreal” von Butz und Krüger [2001], SAIMotion von Bieber u. a. [2002], “Open-Spirit” von Rehrl u. a. [2007] und auch Pfaff [2007] der Hochschule für Angewandte Wissenschaften Hamburg zu nennen.

Parallel entwickelte sich ein offener Standard für die semantische Modellierung von raumbezogenen Daten. Zunächst wurde mit der Modellierung von Straßenkarten begonnen, die auf unterschiedlichen Ebenen verfeinert wurde und letztlich zur Beschreibung des Innenraumes führte. Mit den IFC[1] (Industry Foundation Classes; neuerdings BIM [Building Information Modeling]) und CityGML gibt es zwei Ansätze der semantischen Modellierung. CityGML wird im Folgenden genauer betrachtet.

[1] http://www.iai-tech.org

Wie bereits erwähnt spiegelt sich der Trend zur semantischen Innenraummodellierung auch in der Entwicklung von CityGML[2] (*City Geography Markup Language*) wieder. Mit CityGML[3] wurde ein offenes 3D Stadtmodell entworfen, dessen Objekte in einer *Ontologie* formal modelliert werden (Kolbe [2008a]). Interessenten dieser Modelle werden dazu befähigt, die unterschiedlichsten Fragestellungen zu beantworten – so sind Stadtmodelle z. B. bei der Funknetzplanung für Mobilfunkanbieter, bei der Stadtmodellplanung für Kommunen oder für Umweltsimulationen höchst interessant.

Für die Visualisierung von Stadtmodellen eignen sich diverse Computergrafikformate wie KML (Keyhole Markup Language) – einem Standard der Google Inc.[5]. Doch was unterscheidet CityGML von anderen 3D Stadtmodellen? Welche Vorteile bietet CityGML? Während die meisten anderen Formate auf die *Präsentation* von Raummodellen spezialisiert sind, transportiert CityGML vor allem *Sach- und Strukturinformationen* (vgl. [Rech, 2008] S.13).

Durch Attributierung weiß ein Gebäude sogar *wozu* es gebraucht wird und ob die Benutzung von der ursprünglich vorgesehenen Verwendung abweicht (Gröger u. a. [2008]).

Die Vermutung liegt nahe, dass KML und CityGML konkurrierende Formate sind. Beide verhielten sich jedoch komplementär und zwar in dem Sinne, dass CityGML als Austauschformat semantischer Informationen über virtuelle 3D Stadtmodelle Verwendung findet und KML dem Austausch graphischer Ausprägungen von 3D-Stadtmodellen als Ergebnis eines Gestaltungsprozesses[6] dient (Kolbe [2008b]). KML wird also zur Repräsentation von 3D-Raumdaten verwendet, CityGML hingegen verfügt zusätzlich über eine *semantische Ebene*, die räumliche Informationen in Topologien, Ontologien und Taxonomien verknüpft.

Mit dem Detaillierungsgrad 4 (dem *Level of Detail 4 [lod4]*) steht in CityGML auch ein Innenraummodell zur Verfügung, das man bei Formaten wie KML vergeblich sucht. Es formuliert u.a. feste Rauminstallationen (z. B. Steckdosen, Lampen, Klimaanlage etc.) und Mobiliar (z. B. Aquarien, Möbel, Computer etc.).

Der Nutzen der *semantischen Ebene* in CityGML liegt in der Vielfalt möglicher Abfragen, die bei alleiniger Existenz der Geometrie und Textuierung unbeantwortet blieben. Es werden Entitäten modelliert, die sich in Abfragen gezielt ansprechen lassen, was bei den reinen visuellen Formaten, wie KML nicht möglich ist. Ebenso bestimmen Stadtobjekte in CityGML durch ihre räumlichen und nicht-räumlichen Eigenschaften, sowie ihre Klassifikation und Funktion ihre Bedeutung und die Einordnung in den städtischen Raum[7].

[2] http://www.citygml.org

[3] CityGML wurde von der *Special Interest Group 3D* (SIG 3D[4]) entworfen.

[5] KML ist (wie CityGML auch) ein Standard des *Open Geospatial Consortium* (OGC)

[6] "Dabei ist der Gestaltungsprozess keine einfache Konvertierung eines Datenformats in ein anderes, sondern die Generierung einer computergraphischen 3D-Repräsentation aus 3D-Geoinformationen unter Berücksichtigung kartographischer Prinzipien." vgl. Kolbe [2008b] S.7

[7] Kolbe [2008b] S.3

Die Anwendungen, die von den semantischen Informationen Gebrauch machen, kommen z. B. aus den Bereichen Stadtplanung, Katastrophenmanagement oder auch Umweltsimulation.

Die Anwendungsmöglichkeiten eines Gebäudemodells mit Innenraummodellierung sind vielfältig. Der Fokus dieser Arbeit liegt auf digitalen Gebäudeleitsystemen.

Worin besteht der Vorteil von computergestützten Leitsystemen gegenüber einfachen, auf Papier gedruckten Raumplänen? Zunächst einmal besticht die Papierversion durch eine einfache Bedienung – schließlich stecken mehrere Jahrtausende Erfahrung in der Erstellung von derartigem Kartenmaterial[8]. Jedoch ist der Informationsgehalt solcher Karten oftmals gering und entspricht nicht den Anforderungen der (meist orts*un*kundigen) Zielgruppen, die sich mit unterschiedlichem Informationsbedarf diesen Karten widmen und zudem Zusatzdienste fordern. Ihre Suchkriterien lassen sich grob in zwei Kategorien einteilen: die Suche nach etwas Speziellem (z. B. einem Raum, einer Person) und der Suche nach einer Klasse bzw. eines Typs (z. B. *irgendeinem* Drucker, *irgendeinem* freien Arbeitsplatz).

Folgendes Szenario soll den Vorteil eines digitalen GLS verdeutlichen: Der Weg von zu Hause zum Zielgebäude ist gemeistert, aber im unbekannten Gebäude fällt die Orientierung schwer. Wo finde ich meinen Ansprechpartner? Im welchen Teil des Gebäudes befindet sich Raum 5.02c? Welchen Aufzug muss ich nehmen?[9] Zunächst stehe ich vor *einem* Plan – dem Plan des Stockwerks in dem ich mich gerade befinde. Es ist wünschenswert eine ungefähre Vorstellung vom Zielstockwerk zu haben. Man stelle sich aber vor, alle Stockwerkspläne eines Gebäudes hingen nebeneinander an der Wand! Zudem ist in einigen Gebäuden unklar, welcher Aufzug, oder welche Treppe ohne Umwege zum Ziel führt. Gebäudeleitsysteme bieten hier die Möglichkeit, mir den Weg von meinem aktuellen Standort zum Ziel graphisch dreidimensional hervorzuheben. So kann ein Pfad, ähnlich dem Verlauf einer Arterie durch den Körper, in ein halbtransparentes 3D Gebäudemodell individuell eingezeichnet werden (siehe hierzu Abb. 1.1).

Im Folgenden werden die wesentlichen Vorteile digitaler GIS aufgeführt:

Suchfunktionen Mit digitalen GIS kann gezielt nach *Points of Interest* gesucht werden – z. B. nach Personen, Räumlichkeiten mit bestimmter Raumnutzung (z. B. die Mensa, Arbeitsräume etc.) oder nach konkretem Mobiliar (z. B. Drucker, Erste Hilfe Kasten etc.). Diese stockwerksübergreifenden Informationen bietet ein statischer Stockwerksplan nicht.

[8] siehe http://de.wikipedia.org/wiki/Kartografie#Geschichte_der_Kartographie

[9] Interessante Vorschläge zur Lösung genannter Probleme sind z. B. eine erweiterte Realität (auch: *Augmented Reality*) durch *Headmounted Displays* (vgl. Herglotz [2006]) oder mobile Endgeräte (vgl. Witt [2008]), die der Besucher während seines Aufenthalts bei sich trägt und ihm ortsgebundene Informationen bieten.

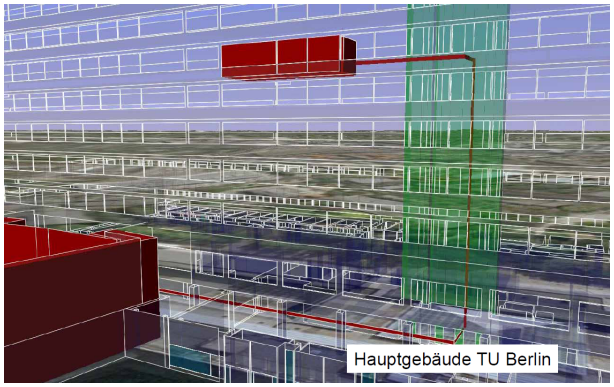

Abbildung 1.1: Navigationspfad (rot) im Hauptgebäude der TU Berlin vom 5.OG zum Audimax (Fahrstühle grün) (aus Kolbe [2008a])

Dynamische Inhalte Die Administration digitaler Raumpläne wird erleichtert, da sich viele Eigenschaften über die Zeit ändern (die Nutzung als auch die Raumgeometrie) – bereits kleine Änderungen sind sofort wirksam, es müssen keine Pläne ausgedruckt und ausgehängt werden. Außerdem verfallen zeitlich begrenzt gültige Informationen in digitalen Systemen ohne manuelles Eingreifen.

Personalisierte Informationen Nachdem der Benutzer eines GIS sich authentisiert hat, können ihm individuelle Informationen geboten werden. Im Bachelorprojekt “Knavi”[10] der Hochschule für Angewandte Wissenschaften Hamburg wurde eine Anwendungen entwickelt, die die RFID Technologie, die in modernen Studentenausweisen integriert ist, ausnutzt, um individuell angepasste Informationen darzustellen.

Interaktive Medien Der Benutzer kann durch interaktive Medien, wie mobile Endgeräte und Touchscreens, mit digitalen Gebäudeleitsystemen interagieren. Die gestenbasierte Interaktion mit Touchscreens wurde in der Arbeit von Rahimi und Vogt [2008] eingehend untersucht. Digitale Beschilderungen mit der Funktion von Landmarken, deren psychologische Aspekte in Schumann [2008] behandelt werden, stellen zusätzliche Hilfsmittel zur Navigation dar.

Befindet man sich vor einem Touchscreen, so möchte man sich manchmal nicht nur einen Weg von A nach B anzeigen lassen, sondern auch mit den dargestellten Objekten interagieren. Folgendes Szenario soll den Nutzen der Interaktion mit virtuellen Raumentitäten verdeutlichen: Studentin Schmidt sucht eines der hochschuleigenen Informationssysteme mit Touchscreen auf. Sie möchte ein Skript ausdrucken, um während der Vorlesung Randnotizen hinzufügen zu können. Sie hält ihren RFID Studentenausweis an das Lesegerät unter

[10] http://code.google.com/p/knavi/wiki/Systemvision

dem Screen und authentisiert[11] sich so beim System. Auf dem Touchscreen erscheint eine Begrüßung, sowie eine Liste der Veranstaltungen, die Studenten ihres Semesters typischerweise in den kommenden Tagen besuchen. Sie tippt auf die Veranstaltung, dessen Skript sie ausdrucken möchte. Unter den nun erscheinenden Informationen rund um die Veranstaltung findet sich eine Liste der Kapitel des Skripts, verdeutlicht durch Datei-Symbole. Während sie mit einem Finger der linken Hand das Symbol auf dem Touchscreen berührt, wählt sie mit einem Finger der rechten Hand ein Stockwerk aus. Das Stockwerk erscheint auf dem Touchscreen. Sie zieht nun das Datei-Symbol auf einen Drucker des Stockwerks. Ein Dialog fragt sie, ob der anfallende Betrag von ihrem Hochschulkonto abgebucht werden soll. Nach der Bestätigung macht sie sich auf den Weg um ihren Ausdruck abzuholen.

Das letzte Szenario veranschaulicht den besonderen, futuristischen Charme der Kombination aus GIS und Touchscreens. Die Touchscreen Technologie erreicht dieser Tage eine vermehrte Verbreitung, was die Akzeptanz bei den Endbenutzern stetig erhöht. In naher Zukunft werden uns mehr und mehr ausgereifte, großformatige Touchscreens im Alltag begegnen – sei es auf Messegeländen, in Warenhäusern oder in Freizeitparks. Es ist zu erwarten, dass zukünftig die Suche nach Informationen mittels Touchscreens an öffentlichen Orten eine ebenso alltägliche Rolle einnehmen wird, wie die Suche nach Informationen im Internet mit Suchmaschinen heutzutage.
Nach wie vor müssen größte Anstrengungen in die *Ergonomie* solcher Systeme investiert werden, damit die Zielgruppe, die kaum bis gar nicht technik-affin ist, mit ihnen umgehen kann. Erkenntnisse über *Seamless Interaction*, *mentale Modelle* und gestenbasierte Interaktion[12] stellen hierbei eine wertvolle Grundlage bereit. Das Ziel sind selbsterklärende, ansprechende Interaktionsmedien für heterogene Zielgruppen.

Für die genannten Szenarien finden sich bereits eine Vielzahl an Gebäudeleitsystemen – oftmals hochspezialisierte Individuallösungen, die auf 2D/3D Computerzeichnungen[13] aufbauen und von Grund auf neu entwickelt werden. Dies hat zur Folge, dass stets Anstrengungen unternommen werden, diese Systeme teilweise von Grund auf neu zu entwickeln.

Es ist an der Zeit, dass verbesserte Interaktionsmedien für die Orientierung in Gebäuden das Leben der Menschen bereichern. Diese Arbeit soll die Grundlagen schaffen, dass Entwickler interaktiver, raumbezogener Anwendungen ihre Ideen schnell in ansprechende Applikationen umsetzen können und sich nicht langwierig mit Grundlagen auseinandersetzen müssen.

[11] Auf die Authentisierung des Nachweises der eigenen Identität folgt die Authentifikation der Bestätigung des Nachweises durch das System.
[12] Die Lektüre von Rahimi und Vogt [2008] wird hierzu empfohlen
[13] Diese Computerzeichnungen werden meist in CAD Dateien gespeichert.

1.2 Ziele

Ziel ist die Entwicklung eines Toolkits, das zukünftigen Entwicklerteams die Grundlage für interaktive, raumbezogene Anwendungen liefert. Nach wie vor ist die Fertigungstiefe[14] in der Informatik verglichen mit anderen Fachgebieten viel zu hoch – im Bereich der Gebäudeleitsysteme soll es sich mit dieser Arbeit ändern.

Um das zu realisieren soll ein semantisches *Innenraummodell* entwickelt werden. Redet man über die Innenraummodellierung, so fallen Begriffe wie “Stockwerk”, “Raum”, “Tür”, “Feste Installationen” (z. B. Heizungen, Steckdosen) oder “Mobiliar” (z. B. Drucker, Tische), die über reine geometrische Konzepte wie Polygone hinausgehen. Diesen Ansatz verfolgt auch der offene Standard CityGML.

Der “Informationsrohstoff”, der dem Toolkit als beispielhafte Datenbasis dient, sind 2D CAD Daten der Stockwerke eines Gebäudes der Hochschule für Angewandte Wissenschaften Hamburg. Sie sind so gesehen “Rohdiamanten”, die erst durch weitere Verarbeitung - der *Extraktion* und Verknüpfung von Informationen – ihren verborgenen Nutzen zum Vorschein bringen. Aus ihnen können geometrische Informationen z. B. über Räume, Treppen oder Aufzüge, als auch topologische Beziehungen, wie die an Räumen angrenzenden Türen gewonnen werden.

Heutige Stadtmodelle sind *statisch* – sie sind “schön anzuschauen”, aber als Bestandteil von Anwendungen fehlt ihnen die Interaktionslogik. Darum soll ein Eventhandling-Mechanismus entwickelt werden, der es ermöglicht, Entitäten mit beliebigem Verhalten (z. B. Reaktion auf Benutzerinteraktionen) und beliebigen kontextsensitive Eigenschaften annotieren zu können. Die Modellierung dieses *Eventmodells* mit einem dazugehörigen Eventmanager, der Trigger und dazugehörige Aktionen verwaltet, ist ein wichtiger Aspekt dieser Arbeit. Der Autor sieht dieses Interaktionsmodell als fehlendes Bindeglied zwischen Benutzer und Raummodell an. Es soll helfen, interaktive Raummodelle effizient realisieren und erweitern zu können.

Schließlich sollen die Daten aus der Extraktion und Annotation in einem *visuellen Prototypen* Verwendung finden. Diese Applikation soll zukünftigen Entwicklern als Grundlage für weitere Anwendungen dienen und die Konzepte des Toolkits validieren.

Die Zielsetzungen lassen sich wie folgt zusammenfassen:

- **Extraktion** aller sinnvoll verwendbarer Informationen aus 2D CAD Stockwerksdaten.
- **Semantisches Gebäudemodell** aller entscheidender Entitäten eines Gebäudes (Stockwerk, Raum, Mobiliar, Personen etc.).

[14] Ad *Fertigungstiefe*: Keiner käme auf die Idee beim Hausbau an das Schmieden von Nägeln oder Abholzen von Bäumen zu denken – in der Informatik ist eine solche Herangehensweise noch immer alltäglich.

- **Annotation** von Räumen. Es soll ein Tool entwickelt werden, das es ermöglicht, das Modell mit weiteren Informationen und interaktiven Entitäten anzureichern. Das Tool wird im Folgenden *AnnotationTool* genannt.
- **Visueller Prototyp** zur Verdeutlichung des Gebrauchs des Toolkits. Der Prototyp wird im Folgenden *WelcomeScreen* genannt.

1.3 Gliederung der Arbeit

Im zweiten Kapitel werden Arbeiten vorgestellt, die vergleichbare Inhalte behandeln. Da es sich hier um ein Toolkit handelt, das diverse Themengebiete umfasst, sind die Themen der vergleichbaren Inhalte weit gefächert. Zunächst wird auf ein Bachelorprojekt eingegangen, das als Ausgangspunkt dieser Arbeit angesehen werden kann. Die Einschränkungen, die dort offensichtlich wurden, werden benannt und Lösungswege aufgezeigt. Darauf wird die Extraktion von Geometrien und semantischen Informationen aus CAD Daten mit Hilfe vorhandener Klassenbibliotheken untersucht. Die extrahierten Daten sollen in ein semantisches Gebäudemodell einfließen. Hierzu wurden CityGML und der IFC Standard untersucht. Nachfolgend wird das Framework EMIC zur Erstellung von Navigationslösungen betrachtet. Zuletzt wird das Eventmodell des JSF Frameworks daraufhin untersucht, ob die zugrunde liegenden Konzepte von Interesse für das Toolkit sind.

Im dritten Kapitel folgt die Anforderungsanalyse. Die funktionalen Anforderungen an das AnnotationTool und den WelcomeScreen werden separat betrachtet. Schließlich werden die nicht-funktionalen Anforderungen an das Toolkit formuliert.

Das vierte Kapitel befasst sich mit dem Design und der Realisierung des Toolkits. Zunächst wird eine Systemidee entwickelt und grundlegende Begriffe definiert. Die Vorgehensweise beim Design ist Top-Down – es wird bei abstrakten Architektursichten begonnen und endet in relevanten Realisierungsdetails. Dabei werden die drei Komponenten – die *ToolkitLib*, das *AnnotationTool* und der *WelcomeScreen* – separat betrachtet. Es werden Möglichkeiten aufgezeigt, das Toolkit mit eigenen Implementierungen zu erweitern.

Im Schlussteil wird zunächst die Arbeit vom Autor rückblickend beurteilt. Es wird aufgezeigt, welche Vorhaben umgesetzt wurden und welche Anforderungen nicht vollends erfüllt werden konnten. Außerdem werden mögliche Erweiterungen erörtert (u.a. das Hinzufügen zusätzlicher Exportformate, personalisierter Informationen und Einsatzmöglichkeiten des Eventmodells).

Um die Lesbarkeit zu erhöhen, wurde darauf verzichtet bei Bezeichnungen von Personen oder Personengruppen auch die weibliche Form zu nennen. Wird z. B. “der Entwickler”, “der

Benutzer", "der Administrator" geschrieben, so ist ausnahmslos auch die weibliche Form gemeint.

2 Vergleichbare Arbeiten

In diesem Kapitel werden Arbeiten untersucht, deren Ergebnisse als Grundlage dieser Thesis dienen können. Zunächst wird auf eine Projektarbeit eingegangen, aus der diese Arbeit hervorgegangen ist. Die Projektergebnisse waren mit Einschränkungen verbunden, die mit dieser Arbeit überwunden werden sollen. Daraufhin werden Bibliotheken und weitere Möglichkeiten zur Extraktion von geometrischen und semantischen Informationen aus CAD Daten untersucht. Es wird ein Framework vorgestellt, das die Entwickelung von Navigationsanwendungen unterstützt. Für ein ausgereiftes Innenraummodell werden verschiedene Modellierungsstandards betrachtet. Letztlich wird auf die Eventhandling-Konzepte verschiedener Frameworks eingegangen.

2.1 Projektarbeit im Wintersemester 08/09 an der Hochschule für Angewandte Wissenschaften Hamburg

Im Wintersemester 2008/09 fand an der Hochschule für Angewandte Wissenschaften Hamburg ein Bachelorprojekt mit dem Titel “Interaktives Informationssystem” statt. Mehrere Studenten, darunter der Autor dieser Arbeit, entwickelten eine Anwendung für die Präsentation von größtenteils hochschulrelevanten Informationen. Die entwickelte Anwendung lief als Web-Applikation auf einem Touchscreen.

Eine Zielsetzung war es, vorhandene CAD Stockwerkspläne zu verwenden, um PC-Arbeitsplatzbelegungen zu visualisieren.

Abb. 2.1 zeigt den Ausschnitt eines Screenshots der entwickelten Anwendung. Der Stockwerksplan wird in der Anwendung als Bitmap Hintergrundgrafik dargestellt. Pro Raum werden die PC-Arbeitsplätze, die durch farbige Rechtecke symbolisiert werden, in einem nicht sichtbaren Raster gruppiert.

Es gibt diverse Kritikpunkte an dieser Umsetzung. Die Konvertierung einer vektorbasierten Grafik in eine Bitmap Grafik geht immer mit Informationsverlusten einher – eine Vergrößerung der Grafik bei angemesser Qualität ist somit nur begrenzt möglich. Außerdem müssen

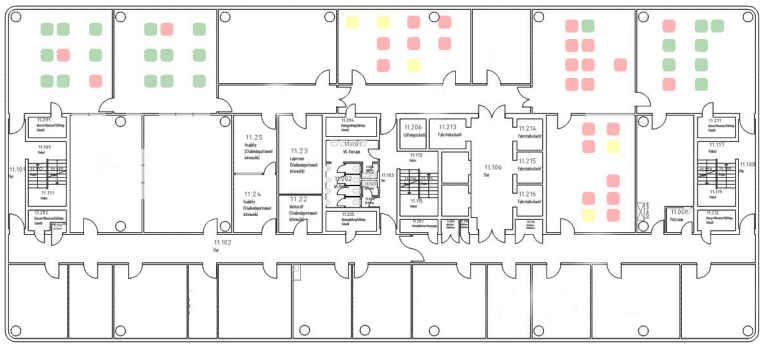

Abbildung 2.1: Visualisierung von PC-Arbeitsplatzbelegungen im 11. Stock an der Hochschule für Angewandte Wissenschaften Hamburg

Änderungen (z. B. am Ort oder der Bezeichnung) der PC-Arbeitsplätze von einem mit der Anwendung vertrauten Entwickler eingepflegt werden. Das macht das System sehr schlecht wartbar.

Beim Umgang mit den CAD Daten wurde ersichtlich, dass die Vektorinformationen in sog. Layern untergebracht sind. Es existieren u.a. Layer über Raumpolygone, Türen und diverse Ausstattungsgegenstände. Der Gedanke drängte sich auf, dass diese Informationen für ein komplexeres Modell verwendet werden können. Damit beschäftigt sich diese Arbeit.

2.2 Extraktion von Informationen aus CAD Daten

Die Extraktion von geometrischen und semantischen Informationen aus CAD Stockwerksdaten ist eines der Ziele des Toolkits, das mit dieser Arbeit entwickelt wird. Es gibt diverse Anforderungen an eine Klassenbibliothek, die auf CAD DWG Daten zugreifen kann. Zum einen sollten einzelne Layer der CAD Daten selektierbar sein, damit auf Geometrien, die z. B. von Räumen stammen, gezielt zugegriffen werden kann. Des Weiteren ist es wünschenswert, dass die Geometrie jeder Entität nach der Konvertierung in der Pfadbeschreibung “path data” des SVG Standards vorliegt, oder auf einfache Weise in diese konvertiert werden kann. Die in dieser Arbeit beispielhaft verwendeten CAD Dateien der Hochschule für Angewandte Wissenschaften Hamburg enthalten in einem Layer Raumnummern. Die Klassenbibliothek sollte in der Lage sein, diese Raumnummern als Text (und nicht als Pfade) zu extrahieren, so dass diese den entsprechenden Räumen zugewiesen werden kann.

Die verwendeten CAD Dateien liegen im proprietären DWG Format vor. Es existieren einige Konvertierungsprogramme[1], die es ermöglichen die Dateien in das DXF Format umzuwan-

[1] Zum Konvertieren von DGW nach DXF eignet sich zum Beispiel das Freeware Tool “A9Converter” (http://www.a9tech.com/a9converter).

deln. Das DXF Format basiert auf dem ASCII-Zeichensatz und die Dokumentation ist frei verfügbar[2]. Daher werden auch Bibliotheken untersucht, die mit diesem Format umgehen können.

Im Folgenden werden mehrere Klassenbibliotheken und weitere Vorgehensweisen beschrieben.

D2X

In Anbetracht der Tatsache, dass als Grafikframework WPF zum Einsatz kommt, welches XAML als deklarative Oberflächenbeschreibungssprache verwendet, scheint das Konversions-Tool *D2X* von Vandervelde [2006] für eine nähere Betrachtung geeignet.

D2X wurde im August 2006 im Rahmen des von Microsoft durchgeführten “The WPF/XAML Conversion Tool Contest”[3] eingereicht und wurde zur besten Anwendung gekürt. D2X konvertiert DWG und DXF CAD Dateien mit 2D oder 3D Inhalten in das XML basierte XAML Format.

D2X behält alle geometrischen Informationen, die in den CAD Daten enthalten sind, bei, ist aber nicht in der Lage, aus den Daten semantische Informationen zu gewinnen. Bei der Konvertierung in XAML werden die Geometrien größtenteils im SVG Format gespeichert, jedoch werden einige Linien auf andere Weise modelliert, was zusätzlichen Konvertierungsaufwand mit sich bringt.

Es fehlt die Möglichkeit einzelne Layer der CAD Dateien zu selektieren, wodurch es ausgeschlossen ist, nach dem Konvertierungsprozess z. B. einzelne Raumgeometrien zu verwenden. Damit kann die Klassenbibliothek nur verwendet werden, wenn die Funktionalität implementiert wird.

netDxf und DxfReader

Die Opensource C# .NET Klassenbibliothek *netDxf*[4] von Daniel Carvajal liegt in der Version 0.1 vor und bietet die Möglichkeit auch auf einzelne Layer der DXF Dateien zuzugreifen. Es unterstützt die meisten Entitäten des DXF Formates (u.a. Arc, Circle, Ellipse, Polyline und Text), aber keine *Splines*. Diese sind jedoch Bestandteil der verwendeten DXF Dateien.

[2] http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=12272454
[3] http://blogs.msdn.com/mswanson/archive/2006/08/29/729501.aspx
[4] http://netdxf.codeplex.com

Die Klassenbibliothek *DxfReader*[5] von Evren Daglioglu bietet einen ähnlichen Funktionsumfang, wie die von netDxf. Leider fehlt auch hier die Unterstützung für *Splines*.

Die Fähigkeit mit Splines umzugehen müsste also zusätzlich implementiert werden, damit eines der Tools in dieser Arbeit verwendet werden kann.

Parsen von DXF Dateien

Eine weitere Möglichkeit bestünde darin, einen eigenen Parser (z. B. mit dem ANTLR Parser Generator[6]) für die DXF Dateien zu generieren. Damit wäre die Entwicklung einer Klassenbibliothek möglich, die genau die oben beschriebenen Anforderungen umsetzen würde. Jedoch ist die Extraktion von Informationen aus CAD Dateien nur ein Teilaspekt dieser Arbeit und daher eine Umsetzung aus Zeitgründen nicht möglich. Das Design des in dieser Arbeit zu entwickelnden Toolkits sollte deswegen so ausgerichtet werden, dass eine nachträgliche Implementation einfach ist.

Fazit

Keine der genannten Klassenbibliotheken wird den Anforderungen vollständig gerecht. Zwar können einzelne Konzepte der Bibliotheken von Nutzen für Teilbereiche dieser Arbeit sein, jedoch ist zusätzlicher Aufwand nötig, um die gewünschten Fähigkeiten hinzuzufügen. Eine komplett eigenständige Lösung, die genau auf die Bedürfnisse zugeschnitten ist, ist in Anbetracht der verfügbaren Zeit nicht in dieser Arbeit umzusetzen.

Daher wurden in dieser Arbeit die CAD Daten in einem mehrstufigen Konvertierungsprozess in ein Dateiformat gebracht, das sich für die Extraktion der relevanten Informationen eignet. Die einzelnen Konvertierungsschritte sind im Anhang unter 6.1 dokumentiert. Zunächst werden in dem Prozess einzelne Layer einer CAD Stockwerksdatei in separate PDF Dateien konvertiert. Darauf wurden diese mit dem Tool *Expression Blend* importiert, auf eine einheitliche Breite normiert und in einer Datei in separat zugänglichen Datenstrukturen zusammengefasst. Dieser Prozess ist zwar mit größerem manuellem Aufwand verbunden, jedoch werden die Informationen dadurch so aufbereitet, dass eine Weiterverarbeitung unkompliziert möglich ist. Eine vollständige Automatisierung des Prozesses bleibt eine Aufgabe für weiterführende Arbeiten.

[5] http://www.codeproject.com/KB/cs/dxfreader.aspx
[6] http://www.antlr.org

2.3 Stadt- und Gebäudemodelle

Die aus dem Extraktionsprozess gewonnenen Informationen und Entitäten sollten in ein Innenraummodell überführt werden, das eine angemessene Repräsentation der geometrischen und semantischen Informationen bietet. Hierzu werden die Modelle IFC und CityGML untersucht.

Borys Kogan stellt in Kogan [2009] diverse Stadtmodelle vor, die es ermöglichen, reale, physische Städte in digitaler Form zu repräsentieren.

Drei Schichten der Datenrepräsentation seien bei den Stadtmodellen besonders wichtig: 1. die *geometrische Beschreibung* der Stadtobjekte, 2. die *Taxonomie* bzw. Kategorisierung und Klassifikation der Objekte und 3. *Darstellungsschicht* zur Visualisierung der digitalen Beschreibung von Stadtobjekten durch Applikationen.

Kogan stellt weiterhin einige Technologien zur Datenrepräsentation räumlicher Objekte vor. Da diese ebenso für diese Arbeit von Relevanz sein können, werden sie im Folgenden kurz erläutert.

2.3.1 IFC – Industry Foundation Classes

Der Standard *IFC* der International Alliance for Interoperability (IAI) ist eine ISO-Norm zur Definition von Komponenten im Bauwesen und Facility-Management. Das Datenmodell beschreibt nicht nur physische Gebäudekomponenten, wie Wände und Türen, sondern auch abstrakte Konzepte, wie Zeitpläne, Organisationen und Kosten, die während eines Bauprojektes benötigt werden. Das Datenmodell wird als Austauschformat im Sinne hoher Interoperabilität zwischen Applikationen im Bauwesen gesehen. Dadurch steht allen Beteiligten der verschiedenen Fachgebiete (Architektur, Bauausführung, Haustechnik und Facility Management) eine gemeinsame Datengrundlage zur Verfügung (siehe Abb. 2.2). “Die Definition der IFC erfasst den gesamten Lebenszyklus einer baulichen Anlage und damit alle Leistungsphasen (Entwurf, Planung, Konstruktion, Vergabe, etc.)“ (Ester [2008] S.4).

Das Datenmodell ist objektorientiert und erlaubt das Hinzufügen zusätzlicher Information z. B. aus dem Bereich der Verwaltung oder dem Gebäudemanagement (z. B. Kosten, Heizlasten). Zudem können externe Dokumente wie Zeichnungsdaten oder Bilder referenziert werden.

Die Notation des IFC Schemas erfolgt im *EXPRESS* Format – dies ist eine Beschreibungsmethode und Bestandteil des ISO-Standards *STEP* (IAI [2009]). IFC Dateien können auch

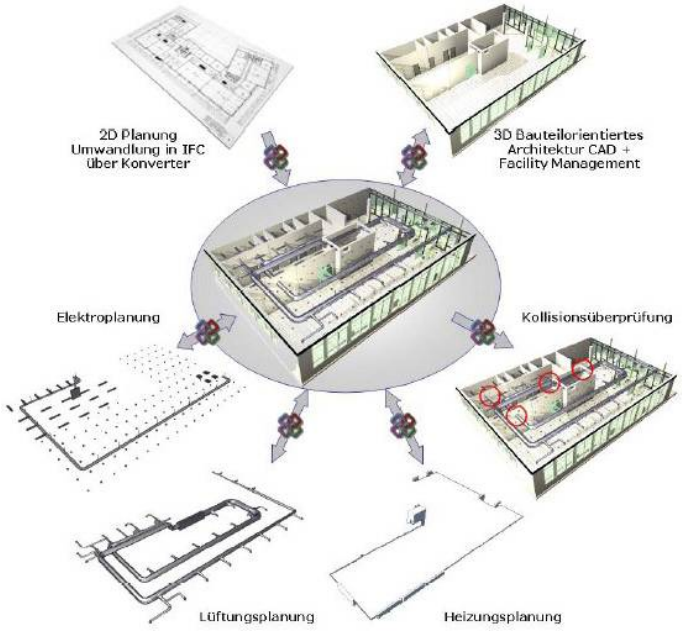

Abbildung 2.2: Datenaustausch der unterschiedlichen Fachbereiche über die IFC-Schnittstelle (Quelle: IAI [2008])

in eine XML Struktur überführt werden. Zur Validierung steht die ifcXML[7] XSD Schema Datei zur Verfügung.

Gebäudekomponenten werden in IFC mit Volumengeometrien dargestellt, wobei Öffnungen wie Fenster und Türen durch boolesche Operationen beschrieben werden (Kogan [2009]).

2.3.2 CityGML – City Geography Markup Language

CityGML ist ein offenes Datenmodell, das von der Special Interest Group 3D (SIG 3D) entwickelt und vom OGC (Open Geospatial Consortium) zum weltweit akzeptierten Standard erhoben wurde. Die ersten Anfänge der Entwicklung des Standards finden sich im Jahr 2002, wo sich Vertreter von Firmen, Kommunen und Universitäten zu einem Planungstreffen zusammenfanden (Rech [2008]).

Der Grundgedanke war ein Stadtmodell zu entwickeln, das nicht nur zur graphischen Visualisierung von 3D Modellen geeignet ist, sondern darüber hinaus eine Strukturierung der enthaltenen Objekte nach inhaltlichen, thematischen und funktionalen Aspekten bietet. Diese Informationen werden für automatisierte Auswertungen verwendet (z. B. für Analysen von Umweltsimulationen), die bei rein visuellen Modellen nicht durchgeführt werden können. Denn

[7] http://www.iai-tech.org/products/ifc_specification/ifcxml-releases/summary

die Geometrie und Erscheinung eines Objektes sagt unmittelbar nichts über die Bedeutung aus (Kolbe [2008a]).

“Rechnergestützte Analysen und Simulationen auf der Basis von 3D-Stadtmodellen erfordern vom Rechner unterscheidbare Entitäten mit räumlichen *und* thematischen Eigenschaften” (Kolbe [2008a] S.4). Dieser semantische Modellierungsaspekt spiegelt sich in CityGML in sofern wieder, als dass Entitäten klassifiziert und ihre Eigenschaften beschrieben werden, und ihre Bedeutung und Funktion in einer Ontologie formal modelliert werden.

Einzelnen Objekten eines Stadtmodells (und sogar eines Gebäudes) wird in CityGML eine weltweit eindeutige Id zugewiesen, wodurch sie beispielsweise für das Management von Immobilien nutzbar gemacht werden. CAD-Systeme können einen derartigen räumlichen Bezug meist nicht herstellen. Die Objekte besitzen thematische Attribute, wie Name, Klasse, Funktion und Nutzung.

CityGML beschreibt eine Vielzahl an städtischen Objekten, wie Vegetation, Flüsse, Bänke und Gebäude. Der Standard beinhaltet Generalisierungshierarchien zwischen Klassen, Aggregationen, Relationen zwischen Objekten sowie räumliche Eigenschaften. Diese thematischen Informationen “erlauben es, die virtuellen 3D(ggf. auch 2D) Stadtmodelle für die anspruchsvolle Analyse in verschiedenen Applikationen im Bereichen der Simulation, Urban Data Mining und Facility-Management anzuwenden” (Kogan [2009] S.22).

CityGML kann durch *Application Domain Extensions* (ADE) fachspezifische erweitert werden. Bestehende Objektarten können um zusätzliche räumliche und nicht-räumliche Attribute, sowie Beziehungen und Relationen ergänzt werden. Ebenfalls können neue, fachspezifische Objektarten eingeführt werden.

In CityGML findet eine konzeptuelle Trennung der Repräsentation von Geodaten und ihrer Visualisierung statt. Die Geometrie von CityGML wird durch eine Teilmenge des GML3 Geometrie-Modells beschrieben Kolbe [2009].

Räumliche und semantische Eigenschaften werden in CityGML in fünf aufeinander folgenden *Level of Detail* (LoD) beschrieben (Kolbe u. a. [2005]). Der kleinste Detaillierungsgrad (LoD0) ist das sog. “Regionalmodell”, das aus einem 2.5D Digitalen Geländemodell (DGM) besteht. Der derzeit größte Detaillierungsgrad findet sich in LoD4, in dem auch das Innere von Gebäuden modelliert wird.

IFC und CityGML haben einige Gemeinsamkeiten und Unterschiede, die im Folgenden auf Basis von Kolbe [2009] betrachtet werden. IFC ist wie CityGML ein semantisches Gebäudemodell, in dem thematische Objekte durch ihre räumlichen, dreidimensionalen Eigenschaften und Beziehungen untereinander repräsentiert werden. IFC bietet ein sehr flexibles Geometrie-Model, bietet aber keine Unterstützung für das CRS (Coordinate Reference System). Da sich die Fokussierung in IFC auf Gebäude richtet, sind keine topographischen Entitäten wie Terrain, Vegetation und Gewässer modelliert. Der Modellierungsschwerpunkt IFC

liegt also im Bereich der Gebäude, wobei CityGML auch die Strukturen von Städten berücksichtigt. IFC Modelle können unter Beibehaltung der meisten semantischen Informationen zu CityGML konvertiert werden (siehe Abb. 2.3.

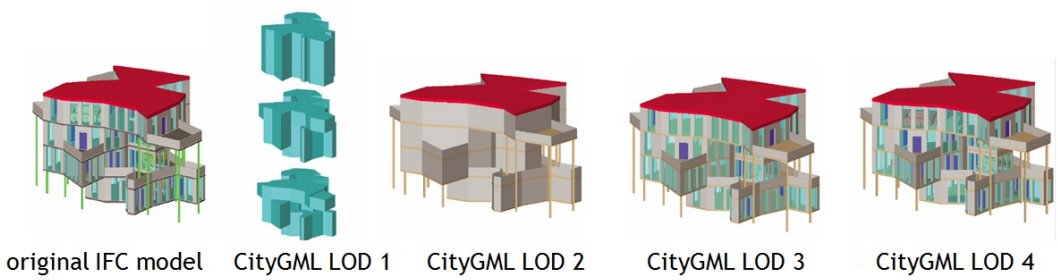

Abbildung 2.3: Konvertierung von IFC nach CityGML (Quelle: Kolbe und Stadler [2008])

Fazit

CityGML und IFC sind Standards mit ausgereiften semantischen Modellierungsmöglichkeiten. IFC legt den Schwerpunkt auf die Modellierung von Gebäuden und berücksichtigt auch Prozesse und die Interessen einzelner Fachgebiete im Bauwesen. CityGML betrachtet hingegen die Topologie ganzer Städte und ist mit dem LoD4 auch in der Lage, Innenräume zu modellieren.

Daher sind zunächst die Konzepte beider Ansätze von Interesse für diese Arbeit. Auf Grund der Skalierbarkeit von CityGML durch die LoDs und des umfassenderen Gesamtkonzepts werden die Modellierungen in dieser Arbeit an CityGML ausgerichtet.

2.4 EMIC – Ein Framework für Navigationsanwendungen

Die Suche nach Frameworks, die die Entwicklung von Navigationsanwendungen unterstützen, gestaltet sich, insbesondere wenn es um den Innenbereich geht, sehr mühsam. Microsoft bietet mit *EMIC* (EMIC Location and Mapping) ein Framework, das im Folgenden genauer betrachtet wird.

Das EMIC Framework bietet Entwicklern eine “Programmierungsplattform” für das Entwerfen von Navigations- und Tracking-Applikationen. Es enthält Tools, um Karten aus verschiedensten Quellen zu integrieren. Außerdem stehen GPS Funktionen genutzt zur Verfügung. Das EMIC-Team versuche mit ihrem Framework, den Prozess der Navigations-Anwendungsentwicklung zu vereinfachen (Kogan [2009]). Nachteil des Frameworks sei, dass

Karten als statische Information als Rastergrafiken abgelegt werden – vektorielle Informationen werden also nicht berücksichtigt. Außerdem ist die Taxonomie oder eine andersartige logische Strukturierung nicht Bestandteil des Frameworks.
Laut Napitupulu [2007] biete das Framework viele Möglichkeiten zur Interaktion mit zweidimensionalen Karten und zur Darstellung von Points-of-Interest (bzw. Areas-of-Interest). Es grundsätzlich für die Nutzung in Outdoor-Szenarien unter Verwendung von GPS gedacht.

Es sprechen mehrere Gründe gegen die Verwendung des Frameworks in dieser Arbeit: Kartenmaterial wird als Rastergrafik verwendet – Ziel ist es jedoch, dass der vektorielle Charakter der CAD Dateien erhalten bleibt. Es wird kein semantisches Gebäudemodell geboten. Der Fokus des Frameworks liegt im Außenbereich – eine Verwendung für den Innenbereich bedeutet somit Entwicklungsaufwand. Außerdem wird Framework von Microsoft nicht weiterentwickelt.

EMIC ist hinsichtlich dieser Kritikpunkte für den Einsatz in dieser Arbeit ungeeignet.

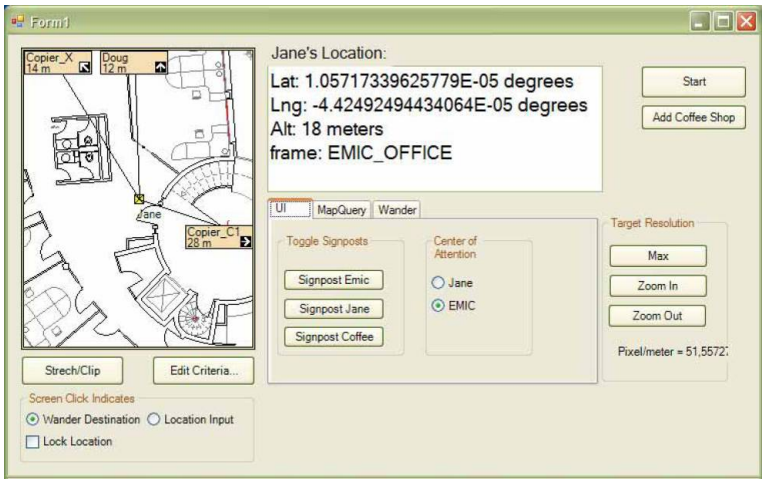

Abbildung 2.4: EMIC: “Demoapplikaton Indoor” (Quelle: Fesl [2006])

2.5 Eventmodell am Beispiel von JSF (JavaServer Faces)

Wie bereits in den Zielen erwähnt wurde, sollen Räume nicht nur mit statischen Informationen annotiert werden können, sondern auch mit dynamischen Inhalten (z. B. multimediale Medien) und multimodalen Interaktionsformen. Darunter fallen Videokonferenzen, “digitale PostIts” oder auch gestenbasierte Interaktionen.

Daher ist ein Konzept zu entwickeln, Verhalten eventbasiert zu modellieren und diese Events mit den Mitteln der Serialisierung vom AnnotationTool zum WelcomeScreen zu transportie-

ren. Dabei geht das Serialisieren über die bekannten Konzepte von Web-Frameworks hinaus.

Ein Framework, das auf ähnliche Weise vorgeht ist JavaServer Faces[8], dessen Eventmodell im Folgenden kurz vorgestellt wird.

JSF ist ein Framework zum Aufbau webbasierter Anwendungen. Die Architektur erlaubt es Anwendungen nach dem Model-View-Controller Entwurfsmuster zu strukturieren. Dabei ähnelt die Vorgehensweise der von Template-Engines wie sie u.a. in PHP verwendet werden (z. B. Smarty[9], HTML_Template_Flexy[10]). Die *View* einer einfachen JSF Anwendung ist eine *JavaServer Pages* Datei, die HTML Code mit eingebetteten Tags enthält.

Im Eventmodell von JSF gibt es Komponenten, die Events auslösen (z. B. ein Button) und sog. *EventListener*, die Events empfangen. Auch *Java Managed Beans* können als *Listener* auftreten. Nicht nur UI Komponenten lösen Events aus. Auch zu Anfang und am Ende jeder der sechs Phasen des *Request Processing Life-Cycle* werden Events ausgelöst (Bosch [2004]).

In folgendem Code-Listing ist ein *Tag* eingebettet, das einen Button und einen dazugehörigen Listener definiert. Die “Bean” ist eine Klasse, die das Event in der Methode `submitButtonClicked` entgegennimmt und bearbeitet.

```
<html:inputText
    identifier = "submitButton"
    value = "Hier klicken"
    actionListener = "#{Bean.submitButtonClicked}"/>
```

Das Konzept der Deklaration von Events in einer XML basierten Struktur ist für das in dieser Arbeit zu entwickelnde Eventmodell evtl. von Nutzen. Da im AnnotationTool Entitäten mit Verhalten annotiert werden sollen, könnte eine solche XML Struktur beim Serialisieren mit diesen Annotationen angereichert werden. Nach der Deserialisierung im WelcomeScreen müssen die Entitäten mit dem Verhalten zu versehen werden. Die Ausarbeitung eines Konzeptes zur Umsetzung dieser Anforderungen ist eines der Ziele dieser Arbeit.

[8] http://java.sun.com/javaee/javaserverfaces/index.jsp
[9] http://www.smarty.net/
[10] http://pear.php.net/package/HTML_Template_Flexy

3 Analyse

In diesem Kapitel soll ein möglichst vollständiges Modell der Anforderungen, die das Toolkit zu erbringen hat, formuliert werden. Hierzu werden Anwendungsfälle und deren Szenarios beschrieben, sowie die funktionalen und nicht-funktionalen Anforderungen konkretisiert. Auf den Ergebnissen aufbauend, wird in Kapitel Design (ab S. 47) die Systemidee und -architektur entwickelt.

3.1 Grundlegende Aufteilung – AnnotationTool und WelcomeScreen

Im Kapitel *Ziele* wurden zwei Hauptapplikationen vorgestellt: Das *AnnotationTool* zum Im- und Exportieren von Raumdaten und dem Annotieren derselbigen; Der Touchscreen-Applikation *WelcomeScreen*, die diese Raumdaten visualisiert und Interaktionsfähigkeiten der Raumentitäten demonstriert.

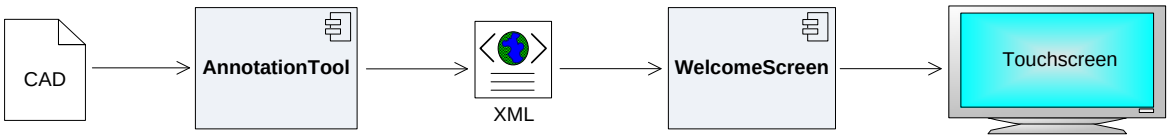

Abbildung 3.1: Zusammenspiel von AnnotationTool und WelcomeScreen

In Abb. 3.1 wird das Zusammenspiel von AnnotationTool und WelcomeScreen deutlich. Das AnnotationTool dient dem WelcomeScreen als Datenlieferant – dieser interpretiert schließlich die Daten und verwendet sie in einer interaktiven Anwendung, die auf einem Touchscreen läuft. Eine detaillierte Beschreibung der Aufgaben beider Systeme findet sich in den folgenden Kapiteln.

3.1.1 Akteure des Toolkits

Die Anforderungen an das Toolkit werden bestimmt von den unterschiedlichen Anforderungen der Hauptakteure: der *Administrator* des GIS, die *Softwareentwickler* und *Benutzer*. Besonderes Gewicht bekommen die Ansprüche der *Softwareentwickler*, die auf Basis dieses Toolkits Erweiterungen implementieren. Die Anforderungen des Benutzer des WelcomeScreens, für den das finale System schließlich gedacht ist, werden zwar im Folgenden knapp beschrieben, sind aber nicht Thema dieser Arbeit[1].

Die Anforderungen dieser Akteure sollen zunächst grob umschrieben werden. In den folgenden Kapiteln finden sich Anwendungsfälle, die die funktionalen Anforderungen dann präzisieren.

Benutzer: Der Benutzer des WelcomeScreens (meist orts*un*kundiger Besucher eines Gebäudes) hat vor allem ein Ziel: schnell das zu finden was er oder sie sucht (eine ansprechende Benutzeroberfläche ist hierbei hilfreich). Zu den Anforderungen des Benutzers zählt bekanntlich insbesondere die Ergonomie eines solchen Systems[2].

Administrator: Der fachkundige Administrator benötigt ein Tool, das ihn so umfassend wie möglich bei seiner Arbeit unterstützt. D.h. insbesondere, dass monotone, sich wiederholdende Tätigkeiten möglichst unterbunden werden. Das Tool soll einfach zu bedienen sein und auch größere Datenmengen handhaben können. Die Ausgabeformate sollen nicht ausschließlich proprietär sein – der Export sollte für eine Weiterverarbeitung in anderen Programmen auch in standardisierte Formate erfolgen.

Softwareentwickler: Der Softwareentwickler, der sich mit dem Toolkit auseinandersetzt, um darauf aufbauende Anwendungen oder Erweiterungen zu implementieren, hat ganz konkrete Anforderungen. Zunächst sollte das System gut dokumentiert sein. Es sollte möglich sein, eine schnelle Übersicht zu bekommen, um eigene Erweiterungen zu implementieren. Detailliert beschriebene Beispielimplementationen vereinfachen diesen Prozess sehr.

3.1.2 Entitäten des Toolkits

Das Toolkit befasst sich mit einem Innenraummodell, dessen Entitäten denen eines realen Gebäudes entsprechen. Der Aufbau eines Gebäudes ist primär hierarchisch organisiert, wie folgende Auflistung zeigt.

[1] Dass die Benutzeranforderungen in dieser Arbeit vernachlässigt werden, heißt nicht, dass zukünftige Entwicklerteams sich darüber keine Gedanken machen müssen – vielmehr sollten sie ihre Anwendungen in umfangreichen Usabilitytests evaluieren.

[2] Da hierzu hinreichende Untersuchungen erfolgt sind, wird darauf nicht näher eingegangen.

- **Gebäude:** Ein Gebäude enthält mindestens ein Stockwerk.
- **Stockwerk:** Ein Stockwerk enthält mehrere Räume, die verschiedene Funktionen aufweisen können (z. B. Konferenz- oder Arbeitsraum, Mensa, Korridor, Büro usw.).
- **Raum:** Ein Raum enthält 0..N Ausstattungsgegenstände (Mobiliar).
- **Tür:** Eine Tür ist immer genau zwei Räumen zugewiesen und verbindet diese somit.
- **Person:** Eine Person *kann* einem festen Raum zugewiesen sein (z. B. ein Professor einem Büro). Der aktuelle Aufenthaltsort kann jedoch davon abweichen und sich dynamisch ändern. Das Modell sollte dies berücksichtigen.
- **Mobiliar:** Mobiliar kann z. B. ein PC-Arbeitsplatz, Tisch, Projektor, Drucker, Erste-Hilfe-Kasten usw. sein.

Die angestrebte virtuelle Repräsentation soll diesen realen Aufbau widerspiegeln.

3.2 Anwendungsfälle und Szenarios

Im Kapitel *Ziele* (Kap. 1.2 S. 15) wurden grundlegende Erwartungshaltungen an das Zielsystem formuliert. Die nun folgenden Anwendungsfälle und deren Szenarios sollen funktionale und nicht-funktionale Anforderungen spezifizieren (Kahlbrandt [2002]) und Qualitätsmerkmale konkretisieren (Starke [2008]).

3.2.1 Anwendungsfälle des AnnotationTools

Das *AnnotationTool* ist eine Anwendung, die Administratoren eines GIS dabei unterstützen soll Gebäudedaten zu verwalten. Diese Daten umfassen die einzelnen Stockwerke, darin befindliche Räume und Mobiliar, sowie Personen und einige weitere Entitäten.
Es soll möglich sein, zunächst ein Gebäudemodell anlegen zu können und dann die einzelnen Stockwerke durch Importieren von Stockwerks-Computerzeichnungen hinzuzufügen. Daraufhin müssen Fehler, die beim Importprozess entstanden sind, durch manuelle Nachbearbeitung behoben werden können.
Schließlich sollen die Entitäten annotiert werden können. Letztlich ist das Modell in geeigneter Weise zu exportieren. Ob die Daten in einer Datei oder Datenbank gespeichert werden sollen, ist noch zu erörtern.

Das AnnotationTool hat also folgende grundlegende Aufgaben zu erfüllen:

Importieren von Computerzeichnungen: Ziel ist die *Extraktion* aller verwendbarer Informationen aus 2D CAD Stockwerksdaten. Auf einzelnen Layern stehen geometrische Informationen zu Räumen, Aufzügen, Korridoren, Liften, Türen und Säulen, sowie weiterer weniger relevanter Details zur Verfügung, die herausgefiltert werden müssen. Die bei der vollautomatischen Extraktion etwaig auftretenden Fehler müssen durch *manuelle Nachdigitalisierung* behoben werden können. Z.B. kann es vorkommen, dass die den Räumentitäten zugewiesenen Geometrien nicht vollständig sind. Bei einer manuellen Kontrolle müssen solche Fehler erkannt (z. B. durch das farbliche Hervorheben ("Highlighten") einzelner/aller Raumentitäten) und schließlich behoben werden können (z. B. durch manuelles Zeichnen der Geometrie).
Der vektorielle Charakter soll beim Extrahieren erhalten bleiben. Außerdem finden sich auf einem Layer identifizierende Eigenschaften, die (soweit sinnvoll) erhalten bleiben sollen.

Annotieren von Entitäten: Das Tool soll das Anlegen, Löschen und Editieren der wesentlichen Entitäten unterstützen. Hierzu zählen u.a. Räume, Türen und Mobiliar. Letzteres soll in Räumen platziert werden können. Außerdem muss ein Verschieben (à la Drag'n'Drop) dieser Entitäten möglich sein (z.B. in einen anderen Raum). Insbesondere soll auch ihr Erscheinungsbild geändert werden können. Eine bereits im Kapitel "Ziele" angesprochene Funktionalität ist die Interaktionsmöglichkeit mit Entitäten. In einer Art Baukasten-System sollen den Entitäten Trigger ("Auslöser") und darauf folgende Aktionen zugewiesen werden können, so dass in der eigentlichen Touchscreen Anwendung ein Anwender mit ihnen interagieren kann. Ein solches Event könnte durch einen Sprachbefehl, Doppelklick, Timer o.ä. ausgelöst werden. Die darauf folgende Aktion könnte ein Voice-Over-IP Anruf bei einer Person sein.

Export als 3D-Gebäudemodell: Ein dreidimensionales Gebäudemodell soll aus den 2D-CAD Daten mit manueller Unterstützung abgeleitet werden. Es ist ist hierbei davon auszugehen, dass die Stockwerke in den CAD Daten planparallel sind, d.h., die hinzuzufügende dritte Dimension ist stets von gleichem Ausmaß.

Eine Übersicht der sich daraus ergebenden Anwendungsfälle findet sich in Abbildung 3.2.2. Wie dort zu erkennen ist, ist der **Akteur** aller folgender Anwendungsfälle ein Administrator des Informationssystems.

Allgemeine Verwaltungsfunktionen

Die folgenden Anwendungsfälle formulieren allgemeine Verwaltungsfunktionen, die von einem Tool zu erwarten sind.

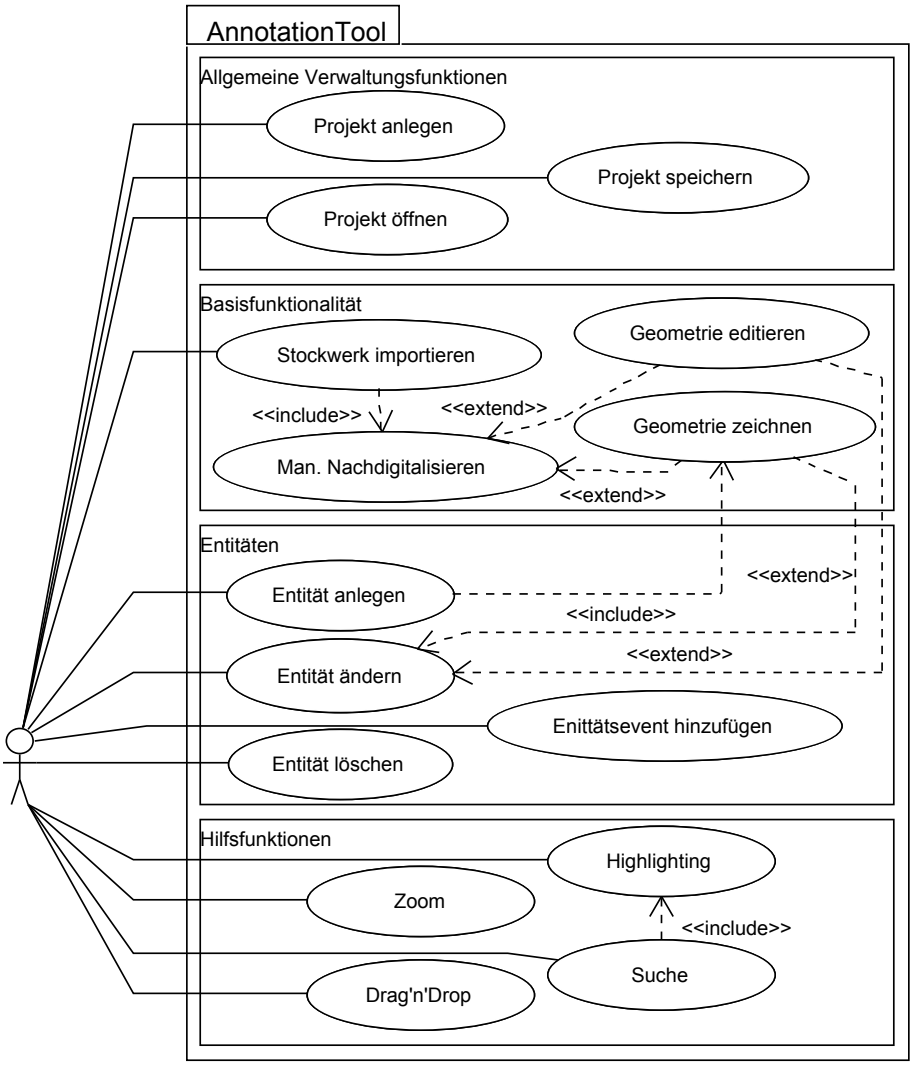

Abbildung 3.2: Anwendungsfalldiagramm des *AnnotationTool*

Anwendungsfall “Projekt anlegen”

Kurzbeschreibung Dieser Anwendungsfall beschreibt das Erstellen eines neuen Projektes.

Vorbedingung Die Applikation *AnnotationTool* ist gestartet und bereit für Eingaben. Es wurden noch keine Eingaben seitens des Akteurs gemacht. Die von einem aktiven Projekt[3] abhängigen Aktionen sind nicht ausführbar, d.h. zum Beispiel, dass das Importieren von Stockwerken (durch einen Klick im entsprechenden Menüpunkt oder eine Tastenkombination) nicht durchgeführt werden kann.

Ablauf Der Akteur klickt auf “Datei” in der Menüleiste. In dem sich öffnenden Menü klickt er auf “Projekt anlegen”. Für das Projekt muss im nun erscheinenden Fenster eine gültige Bezeichnung bzw. Identifikator vergeben werden. Optional kann eine Beschreibung im Feld “Beschreibung” eingegeben werden. Mit einem Klick auf den Button “Anlegen” wird das Projekt erstellt. Die Anwendung ist nun bereit für weitere Eingaben.

Nachbedingung Das System ist nun bereit für das Importieren von Stockwerken.

Anwendungsfall “Projekt speichern”

Kurzbeschreibung Dieser Anwendungsfall beschreibt das Speichern eines Projektes.

Vorbedingung Die Applikation *AnnotationTool* ist gestartet und es existiert ein *aktives Projekt*.

Ablauf Der Akteur klickt auf “Datei” in der Menüleiste. In dem sich öffnenden Menü klickt er auf “Projekt speichern”. Ein Dialog öffnet sich, der den Akteur dazu auffordert das Projekt als Datei im lokalen Dateisystem zu speichern. Hierzu muss ein Name für die Datei in das entsprechende Feld eingegeben werden. Wurde die Datei bereits gespeichert soll das Feld mit dem zuvor verwendeten Namen belegt werden. Existiert bereits eine Datei gleichen Namens muss ein Hinweisdialog erfolgen, der abfragt, ob diese Datei überschrieben werden soll.

Nachbedingung Die gespeicherte Datei kann mit dem AnnotationTool verlustfrei geöffnet werden.

Hinweis Um Arbeitsprozesse zu verkürzen, erscheint es sinnvoll, den erstellten Projekt-Dateityp im Betriebssystem mit dem AnnotationTool zu verknüpfen.

Anwendungsfall “Projekt öffnen”

[3] Aktives Projekt: Ein Projekt wurde zuvor erstellt, oder geöffnet.

Kurzbeschreibung Im Folgenden wird beschrieben wie ein Projekt geöffnet werden kann.

Vorbedingung Die Anwendung ist gestartet und bereit für Benutzereingaben.

Ablauf Der Akteur klickt auf “Datei” in der Menüleiste. In dem sich öffnenden Menü klickt er auf “Projekt öffnen”. Ein Dialog öffnet sich, der den Akteur dazu auffordert eine Projektdatei im lokalen Dateisystem auszuwählen. Ein Klick auf den Button “Öffnen” schließt den Dialog und veranlasst das Tool den in der Datei gesicherten Projektzustand wiederherzustellen.

Nachbedingung Der Zustand des Tools nach dem Öffnen eines Projektes muss dem Zustand nach der Speicherung des Projektes entsprechen.

Basisfunktionalität

Die Basisfunktionen beziehen sich auf den Import von Daten und deren Nachbearbeitung.

Anwendungsfall “Stockwerk importieren”

Kurzbeschreibung Im Folgenden wird beschrieben wie Stockwerksdaten, die in Form von 2D CAD Dateien vorliegen, importiert werden.

Verwendete Anwendungsfälle Manuelles Nachdigitalisieren

Vorbedingung Es liegt ein *aktives Projekt* in der Anwendung vor.

Ablauf Der Akteur klickt auf “Datei” in der Menüleiste und anschließend auf “Importieren”. Ein Dialog öffnet sich, der den Akteur dazu auffordert eine CAD-Datei im lokalen Dateisystem auszuwählen. Ein Klick auf den Button “Importieren” schließt den Dialog und veranlasst das System die Dateiinhalte zu importieren.
Beim Importprozess müssen Daten, die sinnvoll für das Gebäudemodell genutzt werden können, extrahiert, und in Objekten bzw. Entitäten des Toolkits gekapselt werden (z. B. Raum- und Türentitäten). In den CAD-Daten finden sich u.a. sog. *Layer*, die geometrische Informationen über die im Layer befindlichen Elemente enthalten. Die unterschiedlichen Layerbezeichnungen geben einen Hinweis auf die *Funktionen* der enthaltenen Geometrien.
Es existiert z. B. ein Layer mit der Bezeichnung “Raumpolygone”, der zweidimensionale Geometrien aller Räume eines Stockwerkes enthält. Zudem gibt es einen separaten Layer, der die Raumbezeichnungen enthält – diese gilt es schließlich den entsprechenden Raumentitäten zuzuordnen. Die CAD Dateien beinhalten ebenfalls einzelne Layer, die Geometrien von *festen Rauminstallationen* wie z. B. von Feuerlöschern, Rauchmeldern, Wandhydranten oder Lampen, sowie Layer über bewegliches Mobiliar wie z. B.

Fernseher (TV) und Telefone enthalten.
Der Layer “Türe [sic]” beinhaltet Türgeometrien. Eine Tür stellt eine Beziehung zwischen zwei Räumen her (hier sind stets genau zwei Räume durch eine Tür verbunden). D.h., dass es nur möglich ist von Raum A nach Raum B zu gehen, wenn beide die gemeinsame Tür C haben. Der Importprozess sollte diese semantische Begebenheit berücksichtigen und die gewonnenen Türentitäten den entsprechenden Raumentitäten zuweisen.
Ein Hinweisfenster quittiert den Import. Verlief alles fehlerfrei wird “Import erfolgreich!” im Hinweisfenster angezeigt, andernfalls “Import war nicht erfolgreich!” (falls eine Datei nicht dem erwartetem Format entspricht oder Strukturfehler o.ä. aufweist). Diese Schritte können beliebig oft mit weiteren Stockwerksdaten wiederholt werden. Wird ein Stockwerk wiederholt importiert, muss ein Warndialog darauf hinweisen und die Möglichkeit bieten den Import abzubrechen. Andernfalls wird das vorhandene Stockwerk mit den Importdaten ersetzt.

Nachbedingung Das neu importierte Stockwerk wird angezeigt und kann bearbeitet werden. Im Menüpunkt “Bearbeiten”, im Untermenü “Stockwerk auswählen”, ist das neu hinzugefügte Stockwerk eingetragen. Es werden hier alle importierten Stockwerke aufgelistet, wobei sie nach Stockwerksnummer aufsteigend sortiert sind.

Anwendungsfall “Manuelles Nachdigitalisieren”

Kurzbeschreibung Der Importprozess (wie in *Stockwerk importieren* beschrieben) verläuft meist nicht fehlerlos. Dieser Anwendungsfall beschreibt wie Fehler entdeckt und behoben werden können.

Vorbedingung Die Anwendung ist gestartet und es wurde ein Stockwerk importiert.

Verwendete Anwendungsfälle Geometrie editieren, Geometrie zeichnen, Highlighting

Ablauf Zu Beginn müssen Fehler, die während des Importprozesses entstanden sind, durch eine manuelle Überprüfung *erkannt* werden. Fehler treten vor allem beim Import der *Geometrien* auf. Oftmals fehlen sie, sind doppelt eingetragen, gespalten oder deformiert.
Um diese Fehler zu erkennen eignet sich die Hilfsfunktion “Highlighting” (siehe *Highlighting)*. Zunächst ist es sinnvoll alle Entitäten eines Typs zu “highlighten”. Werden somit Fehler in den Geometrien gefunden, können diese bearbeitet werden (siehe *Geometrie editieren*), oder komplett neu gezeichnet werden (siehe *Geometrie zeichnen*).

Nachbedingung Der Mangel sollte behoben werden können, ohne zusätzliche Mängel hinzuzufügen.

Anwendungsfall “Geometrie zeichnen”

Kurzbeschreibung Dieser Anwendungsfall beschreibt das Zeichnen eines Pfades als Grundlage für die Geometrie einer Entität.

Ablauf Beim Zeichnen einer Geometrie wird mit jedem Mausklick auf das Stockwerk ein Eckpunkt des Pfades angelegt, wobei eine Linie von Punkt zu Punkt die Begrenzungen veranschaulicht. Wird während des Anlegens eines neuen Punktes eine Modifiziertaste gedrückt, wird nur eine waagerechte, oder senkrechte Linie gezeichnet. Dies unterstützt das manuelle Zeichnen. Wird nach mindestens drei Punkten der Ursprungspunkt wieder angeklickt, wird der Pfad geschlossen. Das Zeichnen des Pfades bzw. der Geometrie ist beendet.

Nachbedingung Wurde eine Raumgeometrie gezeichnet, so darf diese sich nicht mit anderen Geometrien anderer Räume überschneiden. Wurde eine Türgeometrie angelegt, so muss diese genau zwei Raumentitäten verbinden.

Hinweise Genaues Zeichnen wird durch Heranzoomen erleichtert (siehe hierzu *Zoom*).

Anwendungsfall “Geometrie editieren”

Kurzbeschreibung Dieser Anwendungsfall beschreibt das Editieren einer Geometrie.

Vorbedingung Es muss ein aktives Projekt vorhanden sein. Eine Entität wurde ausgewählt, dessen Geometrie editiert werden soll.

Ablauf Die Umrandung der Geometrie, die editiert werden soll, wird durch Linien von Eckpunkt zu Eckpunkt hervorgehoben. Die Eckpunkte sind durch Kreise o.ä. deutlich sichtbar. Mittels Drag’n’Drop kann ein solcher Eckpunkt nun verschoben werden – dabei werden die verbindenden Linien ebenfalls entsprechend verschoben und in ihrer Länge angepasst. Wurde das gewünschte Ergebnis erreicht, wird der Vorgang durch eine Tastenkombination oder eine Kontextmenüoption beendet. Die geänderte Geometrie wird bei der entsprechenden Entität aktualisiert.

Nachbedingung Die geänderte Geometrie wird der Entität zugewiesen, so dass die ursprüngliche Geometrie verworfen wird.

Hinweise Genaues Editieren wird durch Heranzoomen erleichtert (siehe hierzu *Zoom*).

Annotationen

In diesem Kapitel werden die grundlegenden funktionalen Anforderungen an die Annotation von Entitäten formuliert. Zu den Entitäten des Modells gehören: Ein Gebäude, Stockwerke, Räume, Türen, Mobiliar und Personen.

Generalisierter Anwendungsfall “Entität anlegen”

Kurzbeschreibung Die Entitäten des Modells haben gemeinsame, sowie einige individuelle Attribute. Das Anlegen von Entitäten soll also einem einheitlichem Schema folgen, das in diesem generischen Anwendungsfall[4] beschrieben wird.

Vorbedingung Ein aktives Projekt ist in der Anwendung vorhanden.

Verwendete Anwendungsfälle Geometrie zeichnen

Ablauf Der Aufbau eines realen Gebäudes ist primär hierarchisch organisiert: das Gebäude enthält mehrere Stockwerke, die wiederum Räume enthalten. Ein Raum kann diverses Mobiliar enthalten. Die angestrebte virtuelle Repräsentation soll diesen Aufbau widerspiegeln.
Demzufolge bietet das Kontextmenü einer Entität je nach Entitätstyp unterschiedliche Möglichkeiten andere Entitäten anzulegen. Ein Stockwerk bietet z. B. die Möglichkeit einen Raum anzulegen. Der Raum wiederum bietet die Möglichkeit Türen oder Mobiliar anzulegen, sowie eine Person diesem Raum (z. B. als Büro) zuzuweisen bzw. zu registrieren.
Wählt der Akteur nun den entsprechenden Eintrag im Kontextmenü erscheint ggf. zunächst die Aufforderung eine Geometrie für die zu erstellende Entität händisch zu zeichnen (siehe *Geometrie zeichnen*) oder eine Geometrie aus einer Liste vorgefertigter Geometrien auszuwählen.
Daraufhin erscheint ein Dialog – im Folgenden EntitätsDetail-Dialog genannt. Dieser enthält *mindestens* folgende Felder: Id, Klassifikation, Funktionen, 2D-Geometrie und Beschreibung – diese Attribute haben alle Entitäten des Modells gemeinsam. Das Feld “Funktionen” ist eine Liste, in der mehrere Werte ausgewählt werden können. In den Feldern “Id” und “Geometrie” findet nach jeder Änderung des Inhaltes eine Überprüfung der eingegebenen Werte statt. Im Fehlerfall wird das Feld rot umrandet und eine Fehlernachricht darüber angezeigt. Das Feld “Beschreibung” kann optional ausgefüllt werden. Mit einem Klick auf den Button “Speichern” wird die Entität mit den eingegebenen Werten erstellt und dem Gebäudemodell hinzugefügt. Ansonsten weist ein Dialog auf die Fehler hin.

[4] Anwendungsfälle sind in der UML eine Unterklasse von Classifier, so dass sie (wie Klassen) generalisiert und spezialisiert werden können.

Generalisierter Anwendungsfall "Entität editieren"

Kurzbeschreibung In diesem generischen Anwendungsfall soll beschrieben werden, wie eine Entität mit Hilfe des EntitätsDetail-Dialogs editiert werden kann.

Vorbedingung Es ist ein aktives Projekt vorhanden sowie mindestens eine zu editierende Entität.

Ablauf Das Kontextmenü einer Entität wird aufgerufen. Nach Klick auf den Menüpunkt "Ändern" erscheint der EntitätsDetail-Dialog. Die Felder des Dialogs sind mit den Werten der Entität belegt und können geändert werden. Mit einem Klick auf den Button "Speichern" werden die Änderungen übernommen, wenn keine Fehler mehr angezeigt werden. Ansonsten weist ein Dialog auf die Fehler hin.

Nachbedingung Die vorgenommenen Änderungen an der Entität müssen permanent wirksam sein und dürfen die Integritätsbedingungen (z. B. nur einmalige Vergabe eines Identifikators) des Modells nicht verletzen.

Anwendungsfall "Entität löschen"

Kurzbeschreibung Beschreibt, wie einzelne Entitäten gelöscht werden können.

Vorbedingung Ein Projekt ist aktiv und mindestens eine Entität vorhanden.

Ablauf Das Kontextmenü einer Entität wird geöffnet. Nun wird der Menüpunkt "Löschen" ausgewählt. Die Entität wird gelöscht.

Nachbedingung Beim Löschen einer Entität, die andere Entitäten einschließt (z. B. kann ein Raum mehrere Gegenstände/Mobiliar beinhalten), werden die eingeschlossenen Entitäten ebenfalls gelöscht. Ggf. sollte vor dem Löschen ein Dialog abfragen, ob das Löschen wirklich durchgeführt werden soll.

Anwendungsfall "Event→Aktion hinzufügen"

Kurzbeschreibung Damit ein Benutzer mit einer Entität im *WelcomeScreen* interagieren kann, soll das *AnnotationTool* das Hinzufügen einer Event-Aktion Abfolge[5] zu einer Entität unterstützen. Hierzu ist ein *Eventeditor* angedacht, der alle Events einer Entität verwaltet. Dies wird im Folgenden näher beschrieben.

Vorbedingung siehe Vorbedingungen "Entitäten anlegen" bzw. "Entitäten editieren"

[5] Ein Event besteht aus einem Trigger und einer darauf folgenden Aktion.

Ablauf Im Kontextmenü einer Entität befindet sich der Menüpunkt “Eventeditor öffnen”, der per Klick geöffnet wird. Ein neues Fenster erscheint, in dem alle bereits vorhandenen Events aufgelistet sind. Jeder Eintrag in der Liste kann mit einem entsprechenden Button gelöscht oder geändert werden. Außerdem gibt es einen Button “Event-Action-Pair hinzufügen” – beim Klick auf diesen Button öffnet sich das gleiche Fenster, wie beim Editieren eines Eintrages in der Liste (beim Editieren sind allerdings die Felder mit den zuvor gespeicherten Werten vorbelegt). Dieses Fenster hat zunächst ein Feld “Bezeichnung”, mit dem das Event benannt werden kann. Außerdem beinhaltet es eine Liste mit Triggern und eine Liste mit Aktionen. Aus der Liste “Trigger” kann *ein* Eintrag ausgewählt werden. Aus der Liste “Aktionen” können *mehrere* Einträge ausgewählt werden. Neben dem jeweiligen Eintrag ist ein Button “Einstellungen”, der individuelle Einstellungen eines Triggers oder einer Aktion erlaubt (werden keine Einstellungen vorgenommen, werden die Standardeinstellungen gewählt). Nachdem der Button “Speichern” gedrückt wurde, erscheint das neue Event in der Liste (falls es nicht editiert wurde).
Wird der Button “Schließen” im Eventeditor gedrückt, schließt sich das Fenster.

Hinweis Ausnahmslos jeder Entität steht der Eventeditor zur Verfügung – dem Stockwerk, Räumen, Türen, Mobiliar und Personen.
Sollte sich herausstellen, dass gewisse Events immer wieder (und mit denselben Einstellungen) gebraucht werden, erscheint es aus Gründen der Zeitersparnis sinnvoll, zusätzlich eine Liste mit Standard-Events einzuführen.

Hilfsfunktionen

Hilfsfunktionen unterstützen den Anwender u.a. beim Erkennen und Beheben von Fehlern, beim Auffinden von Entitäten und beim Verschieben derselbigen.

Anwendungsfall “Hightlighting”

Kurzbeschreibung Alle Entitäten eines Typs (z. B. Raum, Tür und/oder Mobiliar) sollen optisch – z. B. durch auffällige Farben – hervorgehoben werden können (engl.: “highlighting”). Entitäten sollen auch einzeln hervorgehoben werden können, z. B. wenn sich der Mauszeiger darüber befindet.

Vorbedingung Die Vorbedingungen entsprechen denen von *Entitäten editieren*.

Ablauf Beim Berühren mit dem Mauszeiger, sollen Entitäten visuell hervorgehoben werden. Verlässt der Mauszeiger das Objekt soll es in den Ursprungszustand zurückkehren. Über den Menüpunkt “Ansicht” und die Unterpunkte “Alle Räume hervorheben”, “Alle

Türen hervorheben” und “Alle Ausstattungsgegenstände hervorheben” können Objekte gleichen Typs hervorgehoben werden.
So können z. B. Mängel einer Geometrie, die beim Importieren auftraten, erkannt werden. Das Beheben der Mängel kann wie in *Geometrie editieren* beschrieben geschehen.

Hinweise Es könnte sinnvoll sein, Tastenkombinationen für die einzelnen Optionen festzulegen.

Anwendungsfall “Drag’n’Drop”

Kurzbeschreibung Entitäten sollen mittels Drag’n’Drop (zu deutsch: “ziehen und fallen lassen”) verschoben werden können. Dies bietet sich insbesondere für Mobiliar, aber auch für den Aufenthaltsort von Personen innerhalb eines Stockwerkes an.

Vorbedingung Die Vorbedingungen entsprechen denen von *Entitäten editieren*.

Ablauf Befindet sich der Mauszeiger über einer Entität und wird die linke Maustaste gedrückt[6], so kann durch Verschieben des Mauszeigers auf dem Stockwerk die Position der Entität verändert werden. Hierbei befindet sich die Entität stets unter dem Mauszeiger. Die Position, an der die Maustaste losgelassen wird, ist die neue Position der Entität. Es erfolgt eine Fehlermeldung, wenn die Entität auf einer Wand, einem anderen Gegenstand oder in einer anderen nicht mit der Realität zu vereinbarenden Position losgelassen wird. Ist dies der Fall wird nach Bestätigung der Fehlermeldung die Entität an den Ursprungsort zurückversetzt.

Nachbedingung Die finale Position der verschobenen Entität muss logisch einwandfrei sein (z. B. darf der Gegenstand nicht in einer Wand *hängen*). Die Geometrie muss entsprechend des neuen Ortes angepasst werden.

Anwendungsfall “Zoom”

Kurzbeschreibung Die Anwendung sollte mit einem Zoom[7] ausgestattet sein. Der Zoom ist u.a. nützlich zum präzisen Zeichnen von Räumen und Türen.

Vorbedingung Es ist sinnvoll, dass im Anwendungsfenster ein Stockwerk eingeblendet ist. Die Computermaus sollte mit einem Mausrad ausgestattet sein.

[6] Bei Systemen mit nur einer Maustaste wird diese gedrückt.
[7] Zoom: Vergrößern/Verkleinern des Bildes oder eines Bildauschnitts.

Ablauf Der Mauszeiger wird an die Position bewegt, die vergrößert oder verkleinert werden soll. Je nach Richtung soll das Drehen des Mausrades das Stockwerk an diesem Punkt heran- oder herauszoomen – alternativ kann eine Tastenkombination gedrückt werden (z. B. bei fehlendem Mausrad).

Hinweis Es ist sinnvoll, dass der Bildausschnitt geändert werden kann. Wird hierfür eine noch festzulegende Taste[8] gedrückt, verschiebt sich der Bildausschnitt entsprechend der Mausbewegung.

Anwendungsfall “Suche”

Kurzbeschreibung Die Suche soll beim Auffinden von Entitäten des Stockwerkes helfen.

Vorbedingung Ein aktives Projekt ist in der Anwendung vorhanden.

Verwendete Anwendungsfälle Highlighting

Ablauf Der Akteur klickt auf “Bearbeiten” in der Menüleiste. In dem sich öffnenden Menü klickt er auf “Suchen”. Ein Dialog öffnet sich. Nach dem Ausfüllen des Suchfeldes wird auf den Button “Suche” geklickt. Befindet sich das gesuchte Objekt im aktuell angezeigten Stockwerk, so wird dieses hervorgehoben (siehe Anwendungsfall “Hightlighting – Hervorheben von Entitäten”). Befindet es sich auf einem anderen Stockwerk, so erscheint ein Dialog, der abfragt, ob in das entsprechende Stockwerk gewechselt werden soll. Falls ja, findet der Wechsel statt und das gesuchte Objekt wird auch dort hervorgehoben.

Ausnahmen und Varianten Ist die Suche ergebnislos, so erscheint ein Hinweisdialog. Hat die Suche mehr als ein Ergebnis, erscheint eine Liste der Ergebnisse, aus der ein Objekt ausgewählt werden kann – daraufhin wird wie in *Ablauf* beschrieben weiter verfahren.

3.2.2 Anwendungsfälle des WelcomeScreens

Der **Akteur** aller folgender Anwendungsfälle ist ein (zumeist orts*un*kundiger) Besucher des Gebäudes. Im Vordergrund steht das Auffinden von Personen, Räumen oder auch Gegenständen. Die zu Grunde liegende Suche lässt sich unterteilen in Suchanfragen, die ein konkretes, individuelles Objekt ausfindig machen wollen (z. B. Raum 10.1) und in Anfragen, die ein Objekt eines bestimmten Typs bzw. eine Klasse von Objekten suchen (z. B. einen Arbeitsraum, Toiletten, Erste-Hilfe-Kasten). Bei Ersteren kann es auch Objekte geben, die

[8] Häufig wird dazu eine sog. Hilfs- oder Modifiziertaste gewählt (Strg, Alt, Umschalt etc.).

dynamisch ihre Position verändern. Die folgenden Anwendungsfälle sind beispielhafte Einsatzmöglichkeiten des Modells.

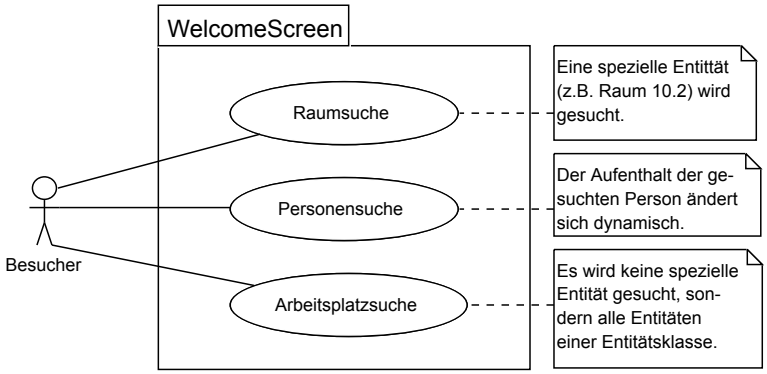

Abbildung 3.3: Anwendungsfalldiagramm des *WelcomeScreens*

Anwendungsfall "Raumsuche – Suche nach einer einzelnen Entität"

Kurzbeschreibung In diesem Anwendungsfall wird beschrieben, wie eine Person, die evtl. zum ersten Mal das Zielgebäude betritt, mit Hilfe des WelcomeScreens den Zielraum auffinden kann.

Vorbedingung Der WelcomeScreen ist bereit für Benutzereingaben.

Szenario Besucher B erreicht das Zielgebäude und sucht Raum 6.60a. B findet den WelcomeScreen schnell (innerhalb von zwei Minuten), da dieser prominent im Eingangsbereich angebracht ist. B wird durch Hinweise auf dem Bildschirm schnell klar, dass dieser durch Berührungen gesteuert wird. B tippt auf das Feld "Suchen nach...". In einer nun erscheinendem Liste erhält B mehrere Suchoptionen zur Auswahl. Da B einen Raum sucht, tippt er auf die Option "Raum". B erhält nun die Möglichkeit ein Stockwerk auszuwählen. Das würde wiederum ein Untermenü mit allen Räumen des Stockwerks öffnen. B entscheidet sich für die Option, die Raumnummer über eine eingeblendete Tastatur direkt einzugeben. B gibt die Raumnummer 6.60a ein (dabei ist es egal, ob in der Raumnummer der '.' enthalten ist, oder nicht) und drückt die 'Enter' Taste. Als Suchergebnis wird zunächst der Weg mittels eines farblich hervorgehobenen Pfades auf einem Stockwerksplan vom WelcomeScreen zum nächsten Fahrstuhl aufgezeigt. Da in diesem Gebäude einige Fahrstühle nur die geraden und einige nur die ungeraden Stockwerke anfahren[9] ist die Fahrstuhlwahl nicht beliebig. Daraufhin wird der Plan

[9] Es gibt auch Fahrstühle, die alle Stockwerke anfahren. Diese Option steht nur für authorisiertes Personal mit speziellem Schlüssel offen.

des Zielstockwerks angezeigt und der Weg vom Fahrstuhl zum Zielraum (wieder durch einen farblich hervorgehobenen Pfad) verdeutlicht.

Nachbedingung Nach einiger Zeit versetzt sich das System wieder in den Ausgangszustand. Zuvor läuft ein visualisierter Timer ab, der bei Bedarf zurückgesetzt werden kann, so dass die Ansicht weiterhin erhalten bleibt.

Hinweise Die Dauer von der Ankunft beim WelcomeScreen bis zur erfolgreichen Suche des Raumes in der Anwendung sollte zwei Minuten nicht überschreiten.

Anwendungsfall "Personensuche – Suche nach einer dynamischen Entität"

Kurzbeschreibung In diesem Anwendungsfall wird eine Person (hier Prof. X) von Person B mit Hilfe des WelcomeScreen*s* gesucht, deren Aufenthaltsort mit einem Trackingsystem ermittelt werden kann. Diese Person wird also ihre Position dynamisch ändern (sofern ein Ortswechseln vollzogen wird).

Vorbedingung Der WelcomeScreen ist bereit für Benutzereingaben. Der Akteur B hat sich mit dem System vertraut gemacht und wird nun seine Abfrage tätigen.

Ablauf B will Professor X, der sich selten in seinem Büro aufhält, mit Hilfe des WelcomeScreens suchen. B tippt nun auf das Feld "Suchen nach...". In einer nun erscheinendem Liste erhält B mehrere Suchoptionen zur Auswahl. Da B eine Person sucht, tippt er auf die Option "Person". B erhält nun die Möglichkeit Personen nach mehreren Kriterien zu suchen (u.a. "ProfessorInnen", "MitarbeiterInnen", "StudentInnen", "Nachnamensuche" etc.). B entscheidet sich für die Option "ProfessorInnen" und tippt auf den Eintrag. Eine alphabetisch geordnete Liste der Professoren erscheint. Es kann ein Buchstabe aus dem Alphabet in einer Tabelle ausgewählt werden um an die entsprechende Stelle in der Liste zu springen oder direkt die Liste durchsucht werden. B wird fündig und tippt auf den Eintrag "Prof. Herbert X.". Die Ansicht des Bildschirms wechselt zur Stockwerksansicht. Zu sehen ist eine Grafik, die eine Person symbolisiert. Die Person bewegt sich auf dem Stockwerksplan von Stock 10. B tippt auf die Option "Weg zur aktuellen Position anzeigen". Zunächst wird der Weg mittels eines farblich hervorgehobenen Pfades auf einem Stockwerksplan vom WelcomeScreen zum nächsten Fahrstuhl aufgezeigt (dazu wird ggf. das Stockwerk gewechselt, sollte sich Prof. X auf einem anderen Stockwerk befinden als der WelcomeScreen). Daraufhin wird der Plan des Zielstockwerks angezeigt und der Weg vom Fahrstuhl zur letzten Position des Professors hervorgehoben.

Anwendungsfall "Arbeitsplatzsuche – Suche nach einer Klasse von Entitäten"

Kurzbeschreibung Gesucht wird ein (freier) Computerarbeitsplatz von Studentin A. Hierbei handelt es sich um eine Suche nach einer Klasse von Entitäten.

Vorbedingung siehe Vorbedingung von *Personensuche – Suche nach einer dynamischen Entität*

Ablauf A sucht einen freien Computerarbeitsplatz – vorzugweise in Stockwerk 11, denn dort befinden sich die Computer ihres Fachbereichs. Momentan befindet sie sich vor dem Touchscreen im Erdgeschoss und tippt dort auf das Feld "Suchen nach...". In einer nun erscheinenden Liste erhält B mehrere Suchoptionen zur Auswahl. Da A einen Arbeitsplatz sucht, tippt sie auf die Option "Computerarbeitsplatz". Eine Liste erscheint mit den Stockwerksnummern und den dazugehörigen freien Arbeitsplätzen. Außerdem erscheint ein Feld "Weg zum nächsten freien Computerarbeitsplatz anzeigen"[10]. Da im 11. Stock noch fünf Plätze frei sind, tippt sie auf den entsprechenden Eintrag des Stockwerks. Das Stockwerk erscheint. Die freien Arbeitsplätze sind leicht zu erkennen, da sie grün hervorgehoben sind. Besetzte Arbeitsplätze sind rot, freie Arbeitsplätz in einem Raum, in dem eine Veranstaltung stattfindet, gelb. Da sie jetzt weiß, wo im 11. Stock freie Arbeitsplätze sind, macht sie sich auf den Weg dorthin.

Nachbedingung A findet einen freien Arbeitsplatz.

3.3 Nicht-funktionale Anforderungen

Zu Beginn dieses Kapitels wurden die drei Hauptakteure, die mit dem System umgehen, genannt: *Benutzer*, *Administratoren* und *Softwareentwickler*. Die Anforderungen der Akteure ist höchst unterschiedlich, denn sie bewegen sich auf unterschiedlichen Schichten des Systems und haben verschiedene Sichten darauf. Der Benutzer bedient das System – er sieht es als Dienstleister an. Der Administrator verwaltet das System, indem er in der Art einer Inventarliste Entitäten hinzufügt, entfernt und ändert. Der Softwareentwickler steigt tief in das System ein, um Änderungen und Erweiterungen vorzunehmen.

In dieser Arbeit stehen vor allem die Anforderungen des Softwareentwicklers im Vordergrund. Dennoch gibt es grundsätzliche Basisanforderungen, die alle drei an das System stellen (wenn auch teils unbewusst). Einige Anforderungen stehen oftmals im Widerspruch zu anderen – so können die Anforderungen an hohe Flexibilität und Portierbarkeit in den meisten Fällen nicht mit der Anforderung an hohe Effizienz bzw. Performanz der Anwendung einhergehen harmonieren. Da jedoch erstere in dieser Arbeit in hohem Maße angestrebt

[10] freier Arbeitsplatz mit der kürzesten Entfernung

werden und bislang keine zeitkritischen Anforderungen formuliert wurden, wird von dem Ziel eine hohe Performanz zu erreichen abgesehen werden[11].

Zuverlässigkeit

Nach ISO [1991] bewertet die Zuverlässigkeit die *Robustheit*, *Verfügbarkeit* und *Korrektheit* eines Systems. Insbesondere die Korrektheit eines Systems ist unabdingbar – ein System, das allen anderen Qualitätsanforderungen genügt, aber nicht korrekt arbeitet, ist unbrauchbar (Sieck [2008] S.6). Sollte das AnnotationTool beispielweise fehlerhafte Daten exportieren, so sind diese Daten letztlich wertlos. Im schlimmsten Fall ist eine Wiederherstellung der Daten unmöglich und es wurde sehr viel Zeit in die Pflege dieser Daten investiert.
Eine Anwendung ist *robust*, wenn sie trotz auftretender Fehler in einem konsistenten Zustand bleibt (Schumann [2008]). Bspw. sollten Fehler in einer Datei beim Importieren in das System nicht zu einem Inkonsistentem Datenmodell oder gar zu einem Absturz führen.

Flexibilität

Da das System als Grundlage für weitere Entwicklungen dienen soll, sind beim Architekturentwurf Strategien für hohe Flexibilität und Anpassbarkeit zu entwickeln. Folgende Qualitätsmerkmale hoher Bedeutung werden hier von [Starke, 2008] (S.71 ff.) genannt: *Analysierbarkeit* (änderungsbedürftige Teile identifizieren), *Modifizierbarkeit* und *Stabilität* (keine unerwarteten Auswirkungen nach Änderungen). Flexibilität muss vor allem hinsichtlich des Hinzufügens neuer Funktionen bzw. Modifizierens vorhandener gegeben sein.
Dabei ist darauf zu achten, nicht *zu viel* Flexibilität mit dem Achitekturentwurf bereitzustellen. Laut [Starke, 2008] S.72 führe dies zu den berüchtigten *Kaugummiarchitakturen*: “In jeder Richtung flexibel, in keiner Richtung stabil”.
Wie bereits erwähnt ist Modifizierbarkeit ein Hauptkriterium. Da das eingesetzte Grafikframework – WPF (Windows Presentation Foundation) - zum Zeitpunkt dieser Arbeit noch als recht neu (und vergleichsweise wenig verbreitet) angesehen werden kann, soll es auch WPF *Neulingen* leicht gemacht werden, Erweiterungen für das Toolkit zu schreiben. Eine gute Dokumentation ist also hier in besonderem Maße erforderlich.

[11] Sollte in einer späteren Phase dennoch die Notwendigkeit aufkommen, finden sich für WPF Anwendungen einige hilfreiche Anleitungen im Internet, die sich explizit mit dem Thema auseinandersetzen (siehe z. B. Optimizing WPF Application Performance im Microsoft Developer Network).

Übertragbarkeit

Weiterhin ist die *Übertragbarkeit* im Qualitätsmodell der ISO/IEC9126 ein auch für diese Arbeit hervorzuhebendes Qualitätsmerkmal. Sie beschreibt die “Eignung der Software, von einer Umgebung in eine andere übertragen zu werden” (Quix [2003]). Wünschenswert ist eine weitestgehend[12] betriebssystemunabhängige Anwendung, die ohne softwarespezifische Anpassungen übertragen werden kann.

[12] Eine Unterstützung der verbreitesten Betriebssysteme wie Windows, Mac OS und unixoide Betriebssysteme sollte angestrebt werden.

4 Design und Realisierung

Das Kapitel *Design und Realisierung* befasst sich mit der Umsetzung der in der *Analyse* herausgearbeiteten Anforderungen.

Hierfür wird zunächst eine *Systemidee* entwickelt. Daraufhin werden Architekturfaktoren erarbeitet, die für die folgenden *Architektursichten* relevant sind. Die Sichten befassen sich mit der Systemumgebung und -infrastruktur (Kontextsicht), den statischen Strukturen der Architekturbausteine (Bausteinsicht), den dynamischen Prozessen zur Laufzeit (Laufzeitsicht) und der physischen Systemumgebung (Verteilungssicht). Die Bausteinsicht ist die umfangreichste Sicht, da mit wichtige Designentscheidungen getroffen werden.

Anders als sonst üblich, wird der Autor in dieser Arbeit in der Bausteinsicht auch auf Details der Realisierung eingehen. Diese Vorgehensweise fügt sich nahtlos in die Top-Down Herangehensweise der Bausteinsicht ein.

4.1 Begriffsdefinitionen

Einige Begrifflichkeiten, die oftmals verwendet werden und im Kontext dieser Arbeit eine veränderte oder erweiterte Bedeutung im Vergleich zum allgemeinen Wortgebrauch bekommen, werden im Folgenden näher erläutert.

Entität und Entitätstyp

Laut GmbH [1986] ist die *Entität* “die bestimmte Seinsverfassung, (Wesen) des einzelnen Seienden, auch dieses selbst”. Überspitzt formuliert ist die Entität alles (materiell oder immateriell) Seiende.

In der Informatik ist eine Entität ein *einzelnes, individuelles Objekt*, das einem Entitätstypen zugeordnet werden kann. Eine Entität entspricht somit einer Instanz einer Klasse, die den Entitätstyp darstellt. Gelegentlich wird in dieser Arbeit der Begriff *Entität* synonym zum Begriff *Entitätstyp* verwendet, was dem allgemeinen Sprachgebrauch entspricht. Wird dieser

Umstand nicht aus dem Kontext ersichtlich, so wird explizit der Begriff *Entitätstyp* verwendet.

Wenn in dieser Arbeit von einer *Gebäudeentität* die Rede ist, sind damit folgende Entitätstypen gemeint: Stockwerk (engl.: *storey*), Raum, Tür, Mobiliar (engl.: *furniture*; im Plural auch *Ausstattungsgegenstände*), aber auch Personen und das Gebäude selbst. Der Begriff "Raumentität" umfasst wiederum alle Entitäten, die sich in einem Raum befinden können, sowie den Raum selbst.

Trigger, Event, Action, Behandlungsroutine

In Kap. 4.4.2 S. 69 werden die Begriffe *Event, Trigger, Behandlungsroutine* detaillierter beschrieben und in einem größeren Kontext betrachtet. Da die Begriffe aber bereits vorher des Öfteren vorkommen, werden sie hier zunächst oberflächlich beschrieben.

Ein *Trigger* löst ein *Event* aus. Eine Entität, die dieses Event empfängt (zuvor wurde sie darauf "registriert"), wählt in einer *Behandlungsroutine* (auch: *Event-Handler*) eine zuvor festgelegt Aktion (im Folgenden *Action*) aus.

Annotation

Im Lateinischen findet sich die "annotatio" – übersetzt "schriftliche Anmerkung" (Stowasser u. a. [1994]). Im Sinne einer "Anmerkung" oder "Beifügung" wird dieser Begriff in der Informatik auch verwendet, um Quelltext mit Zusatz- bzw. Metainformationen zu versehen.

In dieser Arbeit wird unter der *Annotation* bzw. dem *Annotieren* ein Vorgang verstanden, bei dem eine (Gebäude-)Entität mit Informationen über sich selbst (und ihre Umwelt) versehen wird. Dadurch wird ein reflexives Verständnis einer Entität über sich selbst erst möglich (vgl. Zitat v. Prof. Kolbe im Kapitel *Motivation*: "Ein Gebäude weiß, dass es ein Gebäude ist."). Des Weiteren wird darunter verstanden, dass einer Entität weitere Entitäten und auch Verhalten in Form von *EventActions* hinzugefügt werden können.

4.2 Systemidee

In den vorherigen Kapiteln wurden die beiden Hauptkomponenten des Toolkits benannt und ihre Anforderungen und Eigenschaften erörtert:

1. **AnnotationTool**: Ein Tool zum Importieren von CAD-Daten, zur Annotation von Räumen und Export der Daten.

2. **WelcomeScreen**: Ein Prototyp, der die in 1. gesammelten Informationen auf einem Touchscreen visualisiert und das Toolkit evaluiert.

Das Hauptaugenmerk im Kapitel “Design” wird sich auf das AnnotationTool richten. Der WelcomeScreen dient lediglich als Referenzimplementation, um die Verwendung des Toolkits zu verdeutlichen.

Die Gemeinsamkeit beider Komponenten liegt im zu Grunde liegenden Gebäudemodell. Es gilt hierfür ein Modell zu entwickeln, das die Entitäten Stockwerk, Raum, Tür und Mobiliar in einen sinnvollen Zusammenhang stellt. Ein Anwendungsschema, das diese Begrifflichkeiten in einer Ontologie formuliert ist das bereits in Kap.1.1 erwähnte *CityGML*. Dieses Modell sieht eine klare Trennung von Semantik und Präsentation vor, indem es die Realität auf zwei Ebenen abbildet: einer semantischen Ebene, die Sach- und Strukturinformationen kapselt und einer visuellen Ebene, die mit Geometrien und Texturen das Erscheinungsbild von Stadtobjekten beschreibt. Beide Ebenen bzw. Hierarchien können beliebig durchlaufen werden, um thematische und/oder geometrische Abfragen zu stellen (vgl. Gröger u. a. [2008], S.10). Mit dem *Level of Detail 4* werden die für das Anwendungsgebiet dieser Arbeit genannten Innenraum-Entitäten von CityGML bereitgestellt. Somit eignet sich dieser Standard, um ein daran angelehntes Modell zu entwerfen.

Neben der Abbildung realer Objekte ist auch die Modellierung von Personen erforderlich. Diese sollten in dem Modell ihre Position ändern können, haben Kommunikationsschnittstellen und stellen ggf. Dienste bereit (Studienfachberatung, Hausmeisterei, Ansprechpartner d. Usability Labors etc.).

4.2.1 Semantik, Präsentation, Interaktion – die drei Ebenen des Toolkits

Die Ebenen der *Semantik* und *Präsentation* des zu entwickelnden Modells wurden bereits erörtert. Ein weiterer Schwerpunkt liegt, wie bereits in den *Zielen* erwähnt, im Entwurf eines *Eventmodells*. Dies soll es ermöglichen mit Entitäten des Gebäudemodells zu interagieren. Die dritte Ebene der *Interaktion* mit Entitäten wird in Abb. 4.1 dargestellt. Die Ebene setzt auf den Ebenen der *Semantik* und *Präsentation*, die an CityGML angelehnt sind, auf. In Kapitel 4.4.2 wird das Eventmodell ausführlich erörtert.

4.2.2 Gemeinsamer Anwendungskern – ToolkitLib

Aufgrund der Gemeinsamkeit von AnnotationTool und WelcomeScreen bietet es sich an, einen *gemeinsamen Anwendungskern* zu modellieren. Dieser stellt beiden Komponenten ein

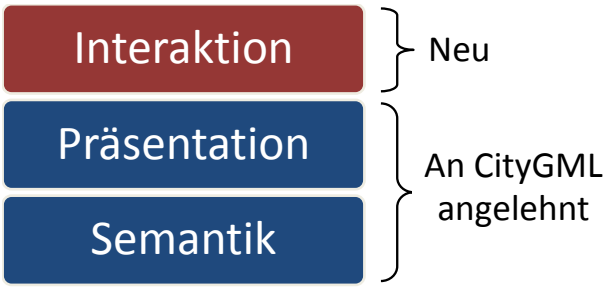

Abbildung 4.1: Die Ebenen des Modells

Gebäudemodell sowie ggf. weitere Dienste bereit. Somit fungiert er zum einen als Objektfabrik, indem die Entitäten des Gebäudes zur Verfügung gestellt werden und zum anderen als Dienstleister für Import-, Export- sowie weiterer Dienste. Dieser Anwendungskern wird im Folgenden *ToolkitLib* genannt werden.

4.2.3 Architekturfaktoren

Es folgt eine Übersicht relevanter Faktoren, die für den Entwurf der System-Architektur erforderlich sind. Sie liefern laut Starke [2008] wichtige Aspekte der Präsentation und Infrastruktur des Systems.

Die Architekturfaktoren beschreiben knapp die *Kernaufgaben*, *Nutzung*, *Benutzeroberfläche*, *Datenverwaltung*, *Steuerung* und *Schnittstellen* zu anderen Systemen. Sie werden im Folgenden für das AnnotationTool, den WelcomeScreen und die ToolkitLib aufgeführt.

ToolkitLib

Kernaufgaben Die *ToolkitLib* ist eine Klassenbibliothek, deren Kern ein an CityGML angelehntes, jedoch stark vereinfachtes und teilweise erweitertes Klassenmodell zur Beschreibung von Gebäuden und darin enthaltenen Entitäten bereitstellt. Es beinhaltet außerdem das zu Grunde liegende Eventmodell für die Interaktion mit den Entitäten und einen Eventmanager, der die auftretenden Events und darauf folgenden Aktionen verwaltet. Die wichtigsten Begriffe der Fachdomäne sind CityGML, Import, Extraktion, Export, Konvertierung, Stockwerk, Raum, Tür, Mobiliar, Person, Eventmodell und Eventmanager.

Nutzung Die Applikationen *WelcomeScreen* und *AnnotationTool* verwenden diese Bibliothek.

Benutzeroberfläche Die Klassenbibliothek hat keine Benutzeroberfläche.

Datenverwaltung Sie bietet Module und Schnittstellen für den Import und Export diverser Formate.

Schnittstellen zu anderen Systemen Die Klassenbibliothek stellt ein Klassenmodell sowie diverse Dienste über Schnittstellen bereit.

AnnotationTool

Kernaufgaben Das System liest CAD Daten ein. Raum-Entitäten können mit Annotationen und EventActions versehen werden. Für Letztere wird ein Eventeditor bereitgestellt, der es dem Benutzer ermöglicht, die einzelnen EventActions einer Entität zu verwalten. Die Informationen lassen sich als XML-Datei speichern und wieder auslesen. Die wichtigsten Aspekte der Fachdomäne sind CAD, Annotation, De-/Serialisierung, Eventmodell und -editor.

Nutzung Das AnnotationTool wird von Administratoren offline verwendet, um Gebäudeinformationen zu verwalten.

Benutzeroberfläche Die Interaktion mit dem Tool erfolgt über eine graphische, objektorientierte Benutzeroberfläche.

Datenverwaltung Die Daten werden in XML-Dateien gespeichert.

Steuerung Die Steuerung der Applikation ist ereignisgetrieben.

Schnittstellen zu anderen Systemen Informationen werden lediglich über XML-Dateien ausgetauscht, wobei dieses System als “Informationslieferant” fungiert. Es handelt sich also im weitesten Sinne um eine Datenschnittstelle. Es wird zudem die Schnittstelle der *ToolkitLib* verwendet.

WelcomeScreen

Kernaufgaben Der visuelle Prototyp liest die Informationen der vom *AnnotationTool* erstellten Daten ein und bereitet diese graphisch auf. Die wichtigsten Aspekte der Fachdomäne sind Touchscreen, GUI, visueller Prototyp.

Nutzung Die Applikation läuft auf einem Touchscreen. Da diese Applikation als Prototyp gedacht ist, wird kein Fokus auf die Ergonomie gelegt.

Benutzeroberfläche Die Interaktion mit der Anwendung erfolgt per Berührung des Touchscreens über eine graphische, objektorientierte Benutzeroberfläche.

Datenverwaltung Die Daten werden aus XML-Dateien gelesen.

Steuerung Die Steuerung der Applikation ist ereignisgetrieben.

Schnittstellen zu anderen Systemen Es wird die Schnittstelle der *ToolkitLib* verwendet. Konkrete Schnittstellen nach außen werden in dieser Arbeit nicht behandelt, könnten aber zukünftig von Interesse sein.

4.3 Architektur

Das MVC-Pattern gehört zum Standard Entwurfsmuster, wenn es um die Trennung von Präsentation, Geschäftslogik und Datenmodell geht. Im folgenden Kapitel wird ein alternatives Entwurfsmuster vorgestellt, das die Architektur moderner Grafikframeworks berücksichtigt. Anschließend wird erörtert, wie das Entwurfsmuster für das Toolkit verwendet werden kann.

4.3.1 Model View Controller – der Standard

Bei Anwendungen mit graphischen Oberflächen ist es im Sinne der Verständlichkeit, Testbarkeit und Wiederverwendbarkeit wichtig, das Design der Oberfläche von der darunter liegenden Logik zu trennen. Hierfür wurde in den 80er Jahren das Model-View-Controller Pattern (MVC) entworfen.

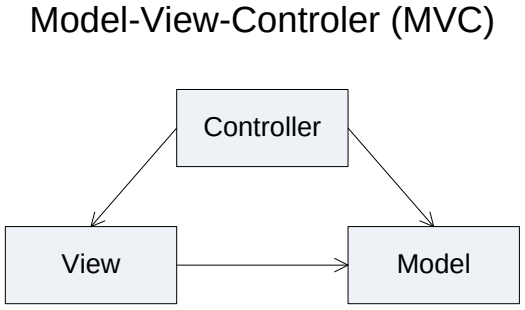

Abbildung 4.2: Model View Controller Pattern

Im MVC-Paradigma enthält das *Model* die Daten, die durch den *Controller* modifiziert werden. Ein Benutzer interagiert über die Präsentationsschicht, der *View*, mit dem Programm. Eingaben werden vom *Controller* entgegengenommen, der darauf ggf. Änderungen am *Model* vornimmt und/oder Informationen an die *View* zurückschickt.

Somit kennt der *Controller* das Model und die *View* – wie in Abb. 4.2 zu sehen. Allerdings gibt es unterschiedliche Auffassungen über die genaue Aufgabe des Controllers (vgl. Orbifold.Net [2009] S.2), die aber auch hier nicht abschließend geklärt werden.

4.3.2 Model View ViewModel – das neue MVC

Das *Model View ViewModel Pattern* (MVVM) ist eine neuere Variante des MVC Patterns und wurde von John Gossman in Gossman [2005] zuerst beschrieben. Er ist einer der Architekten des Design-Tools "Microsoft Expression Blend", das vollständig mit WPF entwickelt wurde. Dabei wurde laut Gossman das MVVM-Pattern intensiv verwendet (vgl. Huber und Pletz [2007] S.2). Die Diskussionen um das *junge* Pattern sind vielfältig[1]. Der Autor dieser Arbeit ist dennoch der Meinung, dass das Architekturmuster für die Strukturierung von WPF Anwendungen Vorteile gegenüber dem herkömmlichen MVC-Pattern hat. Der Einsatz des Patterns und die Vorteile gegenüber dem MVC werden im Folgenden erörtert, indem auf die einzelnen Komponenten *View*, *ViewModel* und *Model* näher eingegangen wird.

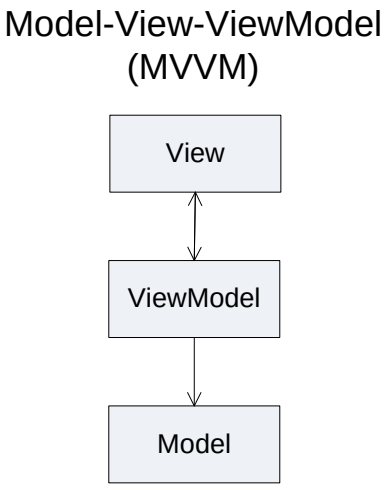

Abbildung 4.3: ModelView-ViewModel Pattern

Die Art und Weise wie graphische Oberflächen – die *View* – in modernen Frameworks erstell werden, gleicht einem Paradigmenwechsel. Zuvor wurden graphische Oberflächen meist in derselben Programmiersprache geschrieben wie der Rest der Anwendung.

[1] siehe u.a. http://www.codecomplete.de/blogs/xamlblog/archive/2007/09/20/mvc-model-view-controller-reloaded-as-mvvm-model-view-viewmodel.aspx, http://dotnet-gui.com/forums/t/216.aspx und http://joshsmithonwpf.wordpress.com/2008/12/01/the-philosophies-of-mvvm/

Das Grafikframework WPF, das zur Realisierung des Toolkits herangezogen wird, bietet mit XAML eine XML basierte Beschreibungssprache für graphische Benutzeroberflächen. Somit können alle Objekte (und sogar dynamisches Verhalten) einer Oberfläche deklarativ beschrieben werden. Diese Herangehensweise findet man auch in Frameworks für Webanwendungen wie Struts und Java Server Faces.

Die Aufgabenverteilung von Designer, der die graphische Oberfläche kreiert und Programmierer ist somit klar abgegrenzt. Eine Durchmischung von *View* und Anwendungslogik ist zwar immer noch möglich, aber durch die Struktur des Grafikframeworks einfacher zu unterbinden, als in früheren Programmierparadigmen.

Die Kommunikation mit der graphischen Oberfläche übernimmt nun das *ViewModel*. Es wird vom Entwickler implementiert und "hat die Aufgabe, alle Informationen bereitzustellen, die für die Aufbereitung der *View* benötigt werden." (s. Huber und Pletz [2007] S. 3). Zu den Aufgaben des ViewModels gehört nicht nur die Darstellung von Informationen des Models, sondern z. B. auch die Verwaltung spezifischer Details der graphischen Oberfläche (z. B. Sortieren der Inhalte einer Liste; Ausgrauen eines Buttons). Trotz der Nähe zum *View* enthält das *ViewModel* aber keine graphischen Komponenten. Wie beim *Controller* des MVC werden Benutzereingaben auch vom *ViewModel* entgegengenommen und weiter behandelt.

Das ViewModel hat eine stärkere Bindung an das View, als es beim MVC der Fall war. Zugriffe der *View* auf die Daten einer Anwendung bzw. das Model erfolgen im MVVM nur über das ViewModel und nicht direkt vom *View* zum *Model*.

In Abb. 4.3 ist das MVVM-Pattern abgebildet. Wird *Databinding* verwendet, so besteht zwischen *View* und *ViewModel* eine Beziehung in beide Richtungen. Databinding ermöglicht das Aktualisieren der View auf sehr einfache Weise (Details siehe Stropek und Huber [2008] S.342 ff.) und ist ein Grund für die Einführung des MVVM-Pattern.

Alle ViewModel Klassen sollten so implementiert sein, dass sie auch ohne die View instanziiert werden können, wodurch sich die Testbarkeit der Anwendung stark erhöht.

4.3.3 Verwendung im Toolkit

Design-Pattern lassen sich in der Praxis oftmals nicht direkt übernehmen, sondern werden den Umständen entsprechend adaptiert. Auf Grund der Unterteilung des Toolkits in die drei bereits erwähnten Komponenten, kann das MVVM-Pattern in einer leicht geänderten Variante Verwendung finden.

In Abb. 4.4 (in Anlehnung an Orbifold.Net [2009]) ist die angestrebte Architektur des Toolkits zu erkennen. In der Abbildung sind auch weitere Details enthalten, die im Folgenden erörtert werden.

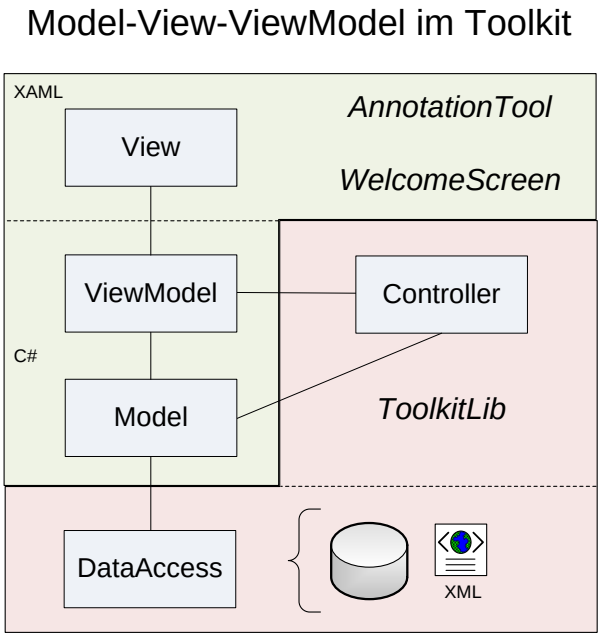

Abbildung 4.4: Architektur des Toolkits

In der Abbildung ist die klassische Drei-Schichten-Architektur zu erkennen. Die Aufteilung der MVVM Komponenten auf die Schichten gestaltet sich einfach. Das AnnotationTool und der WelcomeScreen greifen jeweils auf Dienste der ToolkitLib zurück, die sich in den unteren Schichten – der Businesslogik und Datenhaltung – befinden. Die ToolkitLib bietet also keinerlei graphische Elemente an (mehr zur ToolkitLib findet sich in der Bausteinsicht ab S. 4.4.2).

In der Abbildung sind zu den bereits gezeigten Komponenten des MVVM der *Controller* und die *DataAccess* Komponente hinzugekommen. Letztere stellt lediglich die Persistenzschicht dar und bietet Dienste zum Zugriff auf die Daten, die der Applikation zu Grunde liegen, an. Der Controller fungiert wie der des MVC – somit ist die Architektur eine Mischung aus MVC und MVVM.

4.4 Architektursichten

Die im vorherigen Kapitel *Systemidee* gewonnenen Erkenntnisse fließen nun in die Entwicklung der Architektursichten ein, wie sie von Starke [2008] beschrieben werden. Diese Sichten zeigen das Toolkit jeweils aus einer spezifischen Perspektive, wobei von weniger relevanten Details abstrahiert wird.

Zu den Sichten zählen die Kontextsicht, Bausteinsicht, Laufzeitsicht und Verteilungssicht. Für gewöhnlich werden alle Sichten separat betrachtet und erläutert – an vielen Stellen hat es sich aber angeboten diese Sichten zu vermischen, so dass die beiden zuletzt genannten Sichten in die Bausteinsicht einfließen und nicht zusätzlich erwähnt werden.

4.4.1 Kontextsicht

In der *Kontextsicht* sollen wesentliche Zusammenhänge der Teilsysteme mit ihrem Umfeld konkretisiert werden. Das System wird von außen als Black-Box betrachtet und es werden Schnittstellen sowie Verbindungen mit der Umwelt aufgezeigt.

In Abb. 4.5 werden die grundlegenden Akteure und Systeme der umgebenden Infrastruktur berücksichtigt. Es ist zunächst zu erkennen, dass die Verarbeitungskette bei einem **Fremdsystem** beginnt. Die oftmals in der Raumplanung oder dem Facilitymanagement verwendeten Systeme stellen ihre Daten meist in Form von CAD Dateien zur Verfügung. Die in dieser Arbeit beispielhaft verwendeten CAD Daten stammen aus dem Facility-Management-System der Hochschule für Angewandte Wissenschaften Hamburg und beinhalten jeweils ein Stockwerk.

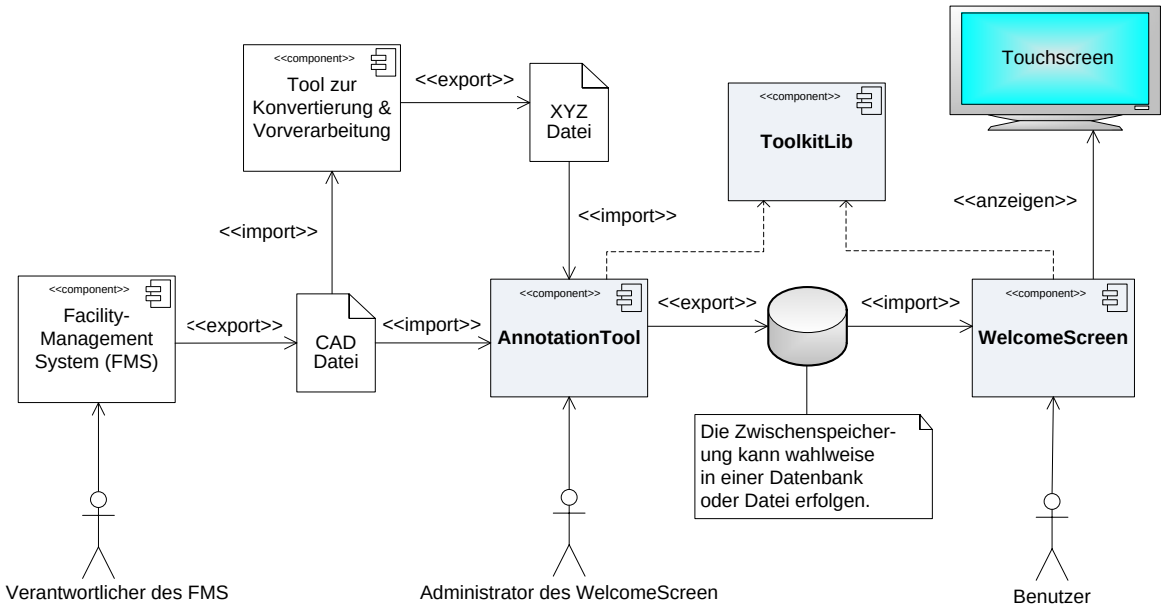

Abbildung 4.5: Infrastruktur der grundlegenden Systeme und Akteure

Oftmals ist kein direkter Zugriff auf die Applikationen oder Datenbanken dieser Fremdsysteme möglich – wie in diesem Fall wird eine für das Fremdsystem zuständige Person (in Abb. 4.5 der “Verantwortliche des FMS”) die Daten in Form mehrerer Dateien bereitstellen. Die

Gefahr besteht, dass notwendige Einstellungen beim Export der Daten nicht vorgenommen wurden und somit die Weiterverarbeitung erschwert wird oder gänzlich unmöglich ist.

Im nächsten Schritt werden die einzelnen CAD Dateien, die *jeweils ein* Stockwerk beschreiben, im **AnnotationTool** importiert. Je nach Umfang der Fähigkeiten des jeweiligen Importmoduls müssen ggf. zuvor einige toolgestütze Bearbeitungs- und Konvertierungschritte vorgenommen werden. Ein solcher Konvertierungsprozess ist im Anhang unter 6.1 genau beschrieben. Mit dem AnnotationTool werden diese Daten nun manuell nachdigitalisiert, annotiert und daraufhin exportiert. Denkbar ist ein Export in eine Datei oder die Verwaltung in einer Datenbank. Der Einfachheit halber wird in dieser Arbeit die Variante des Exports einer Datei bevorzugt.

Die **WelcomeScreen** Touchscreen-Applikation verwendet nun wiederum diese Daten um die enthaltenen Entitäten zu visualisieren und für den Benutzer interaktiv nutzbar zu machen.

Die drei genannten Komponenten *AnnotationTool*, *WelcomeScreen* und *ToolkitLib* bilden das in dieser Arbeit entwickelte Toolkit.

4.4.2 Bausteinsicht

Die *Bausteinsicht* zeigt statische Aspekte des Systems auf und geht detaillierter auf die Implementierungskomponenten und Designentscheidungen ein. Sie verdeutlicht die Struktur und Zusammenhänge von Komponenten und beschreibt deren Schnittstellen. Die in der Analyse herausgearbeiteten Funktionalitäten werden auf Architekturbausteine abgebildet (Starke [2008] S.89).

Im Folgenden werden die Bausteinsichten der ToolkitLib, des AnnotationTools und des WelcomeScreens Top-Down herausgearbeitet, d.h. dass beginnend bei den Komponenten als Blackboxes eine weitere Detaillierung in Whiteboxes folgen wird. Der Schwerpunkt liegt hier auf der ToolkitLib und dem AnnotationTool.

Abbildung 4.6 zeigt die Abhängigkeiten der genannten drei Hauptkomponenten. Wie bereits in “Gemeinsamer Anwendungskern – ToolkitLib” (Kap. 4.2.2 S. 49) beschrieben, stellt die ToolkitLib dem AnnotationTool und dem WelcomeScreen gemeinsam verwendete Objekte, Interfaces und Dienstleistungen bereit.

ToolkitLib

In diesem Kapitel wird die Bausteinsicht der ToolkitLib beschrieben.

Abbildung 4.7 zeigt eine Whitebox-Sicht auf die ToolkitLib. Die Subsysteme ergeben sich aus den funktionalen Anforderungen, wie sie in der Analyse beschrieben wurden:

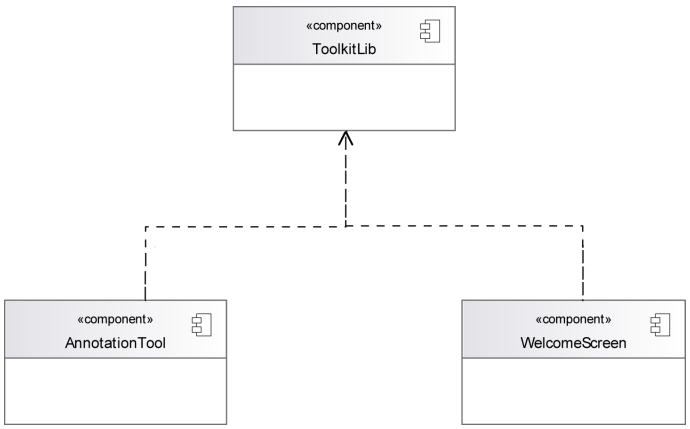

Abbildung 4.6: Komponentendiagramm der ToolkitLib, des AnnotationTools und des WelcomeScreens

BuildingModel (S. 58), *Import* (S. 64), *Export* (S. 68), *EventModel* (S. 69).

Im Folgenden wird auf die einzelnen Subsysteme bzw. Pakete genauer eingegangen.

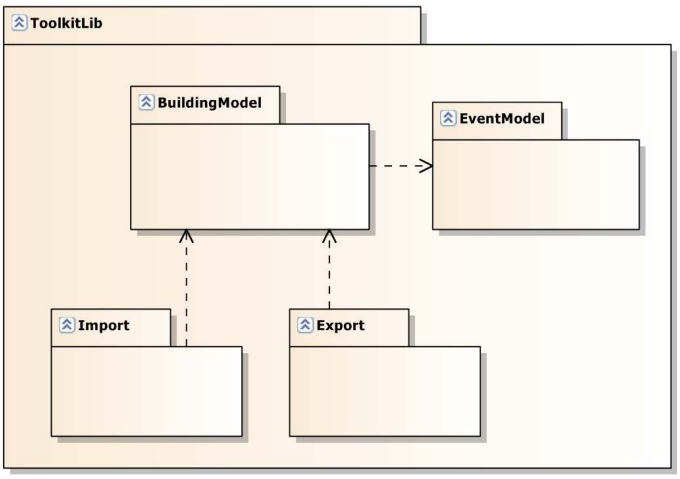

Abbildung 4.7: Whitebox-Sicht ToolkitLib

Paket: `BuildingModel`

Das *BuildingModel* stellt die Entitäten des Gebäudes in Form einer Klassenhierarchie zur Verfügung. Dieses Klassenmodell ist in Abb. 4.8 zu sehen. Es gibt drei Ebenen auf dieses Klassenmodell, die nun erörtert werden.

Gebäudemodell, Persistenz und Verhalten

Die drei Ebenen Gebäudemodell, Persistenz und Verhalten spiegeln sich in den verwendeten Schnittstellen der einzelnen Entitätstypen bzw. Klassen wieder (vgl. Abb. 4.8) und werden im Folgenden genauer betrachtet.

Ebene: Gebäudemodell

Zum einen gibt es die *Ebene der Gebäudeentitäten* – jede Entität wird durch ein Interface beschrieben, das seine Eigenschaften als Gebäudeentität widerspiegelt (z. B. `ISimpleRoom`). Der Nutzen hieraus wird u.a. im AnnotationTool und WelcomeScreen deutlich, wo die Interfaces verwendet werden, um Objekte, die in der GUI dargestellt werden, mit dem Interface einer entsprechenden Gebäudeentität zu dekorieren. Nähere Details hierzu finden sich in "Paket: `ToolBuildingModel`" Kap. 4.4.2 S. 83.

Ebene: Persistenz

Die *Persistenzebene* behandelt das Speichern der Informationen des Gebäudemodells. Um die Daten in einer XML-Datei (benutzerdefiniert) zu serialisieren, implementiert jede Klasse das Interface `IXmlSerializable`. Da bei der Realisierung des Toolkits dem Prinzip des "Programmieren gegen Interfaces" gefolgt wird, ist es für einen `XmlSerializer`, wenn er auf ein Interface trifft, nicht möglich zu entscheiden, welche konkrete Implementierung zur Serialisierung herangezogen wird. Mit Hilfe des Interfaces `IXmlSerializable` kann der Ablauf der Serialisierung und Deserialisieurng im Detail festgelegt werden.

Dies betrifft u.a. die Wahl von Attributen und Elementen, die Art der Serialisierung von Listen bzw. Collections und die Wahl konkreter Implementierungen bei der Serialisierung von Interfacetypen. Insbesondere kommt dies im Gebäudemodell zum Tragen, wenn Entitäten eines Raumes, z. B. Mobiliar oder Medienelemente, die in einer generischen Liste verwaltet werden, serialisiert werden müssen. Hinter *einem* gemeinsamen Interface (`IBuildingFurniture` für alle Entitäten, die ein Raum aufnehmen kann) verbergen sich unterschiedliche Implementierungen, dessen Serialisierungsmechanismus entsprechend ihres konkreten Klassentyps ausgewählt werden muss.

Ebene: Eventmodell

Die dritte Ebene beschreibt das *Verhalten* von Entitäten. Hierfür wurde ein Eventmodell erarbeitet (s. Kap. 4.4.2 S. 69), das es – vereinfacht gesagt – dem Administrator erlaubt, einzelnen Entitäten über die Benutzeroberfläche des AnnotationTools bestimmte Reaktionen auf ein Ereignis hinzuzufügen. In einer konkreten Applikation wie dem WelcomeScreen werden die Zuweisungen dann aktiv und die Entität zeigt das gewünschte Verhalten.

Die Schnittstellen, die diese Sicht begründen, geben Entitäten die Behandlung bestimmter Ereignis- bzw. Eventtypen vor. Deshalb gibt es pro Eventtyp *eine* mögliche Schnittstelle (z. B. `IPositioningEvent`), die eine Gebäudeentität implementieren kann. Der Übersicht halber wurden diese Schnittstellen in Abb. 4.8 entfernt.

Klassenmodell der Gebäudeentitäten

Das vereinfachte Klassenmodell, basierend auf Interfaces, ist das Resultat einer Designentscheidung. Es wäre auch möglich direkt Klassen eines CityGML Klassenmodells zu verwenden, das aus den XSD Schema Dateien von CityGML generiert wird. Durch das Ableiten kann in Unterklassen zusätzliches Verhalten hinzugefügt werden. Auch hier müsste die Serialisierung durch das Implementieren des `IXmlSerializable` Interfaces vorgenommen werden. Dies wiederum hängt stark von den Interna des CityGML Modells ab und wäre bei einer Änderungen des Modells durch die SIG 3D aller Wahrscheinlichkeit nach hinfällig gewesen. Außerdem wäre die Verwendung der Klassen in einem Userinterface z. B. durch Dekorieren von UI-Elementen nicht möglich, ohne in den Quelltext des generierten Klassenmodells einzugreifen. Dies ist aber nicht wünschenswert, da – wie bereits erwähnt – der CityGML Standard sich ändern wird, die die direkten Manipulationen im generierten Quellcode hinfällig machten. Eine weitere Variante wäre das Verwenden von Proxys, die vor die eigentlichen Objekte geschaltet würden und so ihren Zugriff zu steuern und zu erleichtern (die CityGML Klassen sind mitunter sehr komplex – ein Proxy könnte z. B. das Hinzufügen einer Tür zu einem Raum stark vereinfachen). Dies würde aber wiederum bedeuten im generierten Code Änderungen vornehmen zu müssen und wäre so bei Änderungen des CityGML Modells auch hinfällig.

Somit fiel die Entscheidung auf ein stark vereinfachtes Modell mit CityGML als Vorbild. Dieses Modell beschränkt sich auf die für den Zweck des Toolkits notwendigen Entitäten. Diese “Weltsicht” ist zwar stark eingeschränkt, kann aber leicht erweitert werden und ist durch ihre Einfachheit unkompliziert zu verwenden. Außerdem ist es losgelöst von einem konkreten Standard und dessen Implementation. Durch die Nähe zu CityGML ist es nicht allzu aufwendig ein Exportmodul zu implementieren, das korrekte CityGML XML-Dateien generiert.

Das `CodesDictionary` – Realisierungsdetails

Die abstrakte Oberklasse `SimpleAbstractCityObject` faktorisiert Gemeinsamkeiten aller Entitäten des Gebäudemodells heraus. Dies sind vor allem Felder, die eine Entität *umschreiben*. Die Felder `Class`, `Function`, `Description` und `Id` finden sich auch in CityGML Entitäten wieder.

Die Klassifikation (`Class`) und Funktionen (`Function`) werden zur Beschreibung der *Taxonomie* von Stadtobjekten verwendet. Dadurch können die Objekte “nach ihren visuellen, kulturellen oder funktionalen Eigenschaften sortiert werden” (Kogan [2009] S.20).

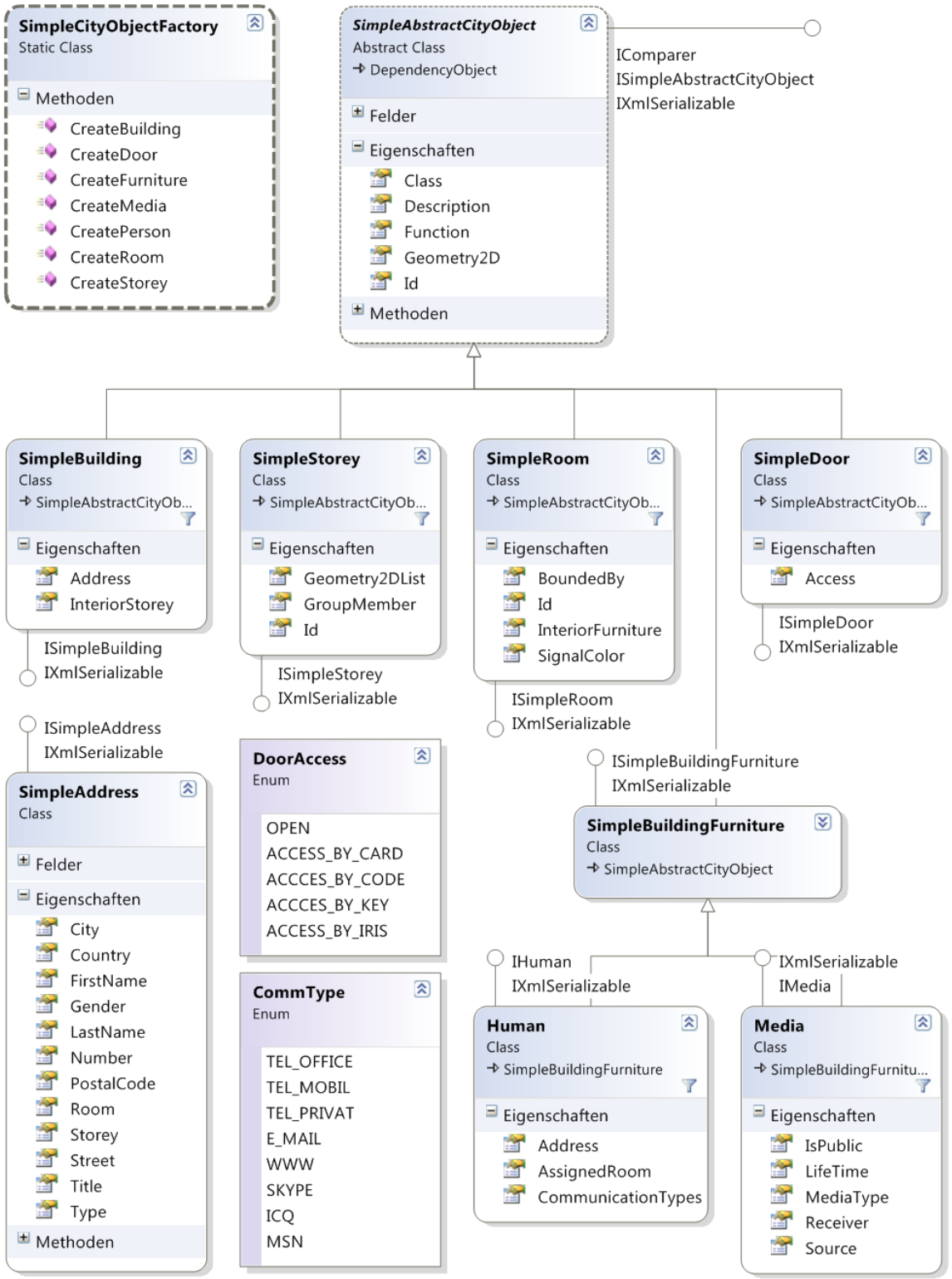

Abbildung 4.8: Klassendiagramm des Gebäudemodells – Whitebox-Sicht “BuildingModel”

Zu den Feldern `Class` und `Function` gibt es ein *Dictionary* in Form einer XML Datei, das vom *Open Geospatial Consortium* bereitgestellt wird (siehe Schlueter u. a. [2007] und Gröger u. a. [2008] S. 159 ff.). Unter einem *Dictionary* wird eine Speicherstruktur bestehend aus einmaligen Schlüsseln und dazugehörigen Werten verstanden. Im CityGML Dictionary sind die Schlüssel eine vierstellige Zahl, der jeweils eine Zeichenkette zugeordnet ist. Für jeden Function- oder Classtype gibt es ein Dictionary (z. B. `RoomClassType`, `BuildingFurnitureFunctionType`, `BuildingClassType` etc.). Es findet sich z. B. für die Entität "Raum" und dem Dictionary `RoomFunctionType` unter dem Schlüssel `1600` eine Zeichenkette "computer room" (Computerraum). Der Zugriff auf diese CityGML "ClassTypes" und "FunctionTypes" erfolgt mit Hilfe der Klasse `CodesDictionary`. Diese bekommt in einer statischen Methode den "CodeIdentifier" einer Entität (z. B. `RoomFunctionType`) übergeben und liefert ein Dictionary, das aus der genannten XML Datei mittels XPath Abfragen generiert wird, zurück. Dieses wird inbesondere im AnnotationTool verwendet, um einer neu anzulegenden Entität ihre Funktionen und Klassifikation zuzuweisen.

Das Dictionary der OpenGIS ist zwar sehr umfangreich, aber dennoch nicht immer ausreichend. Daher wird in Gröger u. a. [2008] S.12 darauf hingewiesen, dass die sog. *External-CodeLists* erweitert oder neu definiert werden sollen. Ebenso können auch andere Dictionarys referenziert werden. Dem Autor dieser Arbeit ist diesbezüglich aufgefallen, dass bspw. ein `RoomType` für behindertengerechte WCs sinnvoll ist – erst dadurch wird es möglich einen Raum (hier ein WC), ohne seine Geometrie zu kennen, als behindertengerecht einzustufen. Es werden auch keine Dictionarys für "Personen" angeboten, was aber in Anbetracht möglicher Suchabfragen (z. B. Suche nach einem Hausmeister) ebenso sinnvoll erscheint.

Adressen mittels `SimpleAddress`

In Abb. 4.8 ist die Klasse `SimpleAddress` abgebildet. Sie wird z. B. von der Klasse `SimplePerson` und `SimpleBuilding` verwendet. Diese Klasse ist an die *eXtensible Address Language* (OASIS [2002], OASIS [2009]) zur Beschreibung von Adressen angelehnt, die auch in CityGML verwendet wird. Die xAL ist für über 200 länderspezifische Adressangaben ausgelegt und daher sehr umfangreich. Deshalb definiert `SimpleAddress` nur die für die Fachdomäne wichtigsten Felder (siehe hierzu Abb. 4.8).

Modellierung von Personen

Abseits moralisch, ethischer Bedenken wird die Klasse `SimplePerson` von *SimpleBuildingFurniture* abgeleitet. Dies mag zunächst befremdlich wirken, dennoch gibt es hierfür gute Gründe.

Auf Grund der bewusst gewählten Nähe zu CityGML ist es ratsam das Modellierungsschema beizubehalten. Dadurch ist es möglich einen CityGML-Exporter zu implementieren, der

das Gebäudemodell inklusive neu hinzugefügter Entitätstypen exportiert, ohne den CityGML Standard zu verletzen. Dafür müssen vorhandene Entitätstypen (z. B. `SimplePerson`) durch sog. Fachschalen (*Application Domain Extension [ADE]*[2]) erweitert werden[3]. Hierfür ist es vorgeschrieben, dass zusätzliche Entitätstypen (z. B. `SimplePerson`) des Modells von *vorhandenen* Typen abgeleitet werden sollen (vgl. Nagel [2009]).

Eine `SimplePerson` besitzt mehrere Kommunikationsschnittstellen (vgl. Enumerator `CommType` in Abb. 4.8) – bspw. E-Mail, Telefon, Skype. Das Feld `Id` kann bei Professoren/Mitarbeitern mit dem hochschulweiten Kürzel versehen werden. Die Klassifikation (Feld `Class`) entspricht dem Einsatzgebiet (Lehre, Verwaltung, Hausmeisterei, Lehrbeauftragte, Student etc.). Die Funktionen (Feld `function`) entsprechen den "Services", die eine Person bereitstellt (z.B. Studienfachberatung, Fundsachenverwaltung, BAföG-Nachweis etc.). Die Verwaltung der Werte dieser Felder kann, wiederum an CityGML angelehnt, mit einem "Dictionary" geschehen. Im Feld `AssignedRoom` kann einer Person ein Raum zugewiesen werden. In der Regel ist es das Büro der Person (oder ggf. ein Abschlussarbeitsraum eines Studenten). Eine Person kann sich dennoch im Gebäudemodell zu einem bestimmten Zeitpunkt in einem anderen Raum befinden, d.h. ihren Aufenthaltsort *dynamisch* ändern.

Geometrie von Entitäten im SVG Standard

Im *BuildingModel* hat jede Entität eine Geometrie (`Geometry2D`), die ihre zweidimensionale Ausdehnung vektoriell beschreibt. Zur Repräsentation der zweidimensionalen Geometrie wird die Pfadbeschreibung "path data" des SVG Standards (*Scalable Vector Graphics*) verwendet (Consortium [2003]). Im folgenden Beispiel wird durch einen solchen SVG-Pfad ein Dreieck beschrieben:

`M 100 100 L 300 100 L 200 300 z` (Beispiel für eine SVG-Pfadbeschreibung)

Dieser Pfad lässt sich als `String` speichern und problemlos serialisieren. Die Entscheidung, die Geometry per SVG zu beschreiben, lag auf Grund des verwendeten Grafikframework WPF nahe[4]. In WPF kann der Eigenschaft "data" einer Instanz der Klasse `Path` ein solcher SVG-String zugewiesen werden. Somit ist es einfach die Form einer Entität des Gebäudemodells zu beschreiben und in der Benutzerschnittstelle einer WPF Applikation zu verwenden. Außerdem ist der Aufbau eines SVG-Pfades sehr schlicht, so dass eine Konvertierung in andere Formate möglich ist.

[2] Ein Tutorial zum Hinzufügen eigener ADEs in CityGML findet sich in Nagel [2009].
[3] Mit entsprechenden XML Schema Definitionen wird der dadurch erweiterte CityGML Standard nicht verletzt.
[4] In WPF wird diese vektorielle Beschreibung "Pfadmarkupsyntax" genannt (siehe http://msdn.microsoft.com/de-de/library/cc189041(VS.95).aspx).

Instanzen des Gebäudemodells – `SimpleCityObjectFactory`

Um Instanzen des Gebäudemodells zu erstellen, bietet die ToolkitLib eine vereinfachte "Factory Klasse" (`SimpleCityObjectFactory`) an (siehe Abb. 4.8). Da der Zugriff bzw. das Erzeugen von Gebäudeentitäten *nur* über diese Klasse möglich ist, werden weitere Zugriffe in das Modell vermieden und somit die Kopplung verringert. Der Aufrufer erhält von der vereinfachten Fabrik eine Instanz vom Typ eines Interfaces (z. B. `ISimpleRoom`). So wird die Implementation vor dem Aufrufer verborgen.

Werden in Zukunft Änderungen und Erweiterungen am bestehenden Klassenmodell vorgenommen, kann in der Factory-Klasse entschieden werden, welches konkrete Objekt eines Typs erzeugt werden soll (dies kommt dann einer Factory [im eigentlichen Sinne] nahe).

Paket: `Import`

Das *Import Paket* (siehe Abb. 4.9) bietet Klassen, die CAD (und Projekt-) Dateien einlesen und als Resultat eines Extraktionsprozesses Entitäten des Gebäudemodells zur Verfügung stellen. Weiterhin sind Schnittstellen vorgegeben, die, möchte man zusätzliche Import-Klassen für weitere Formate entwickeln, implementiert werden müssen. Diese Importer müssen nicht auf Dateien beschränkt sein – vielmehr ist auch der Im- und Export in eine Datenbank denkbar.

Die wichtigsten Interfaces des Import Paketes sind die Schnittstellen `IStoreyImporter` und `IProjectImporter`. Erstere wird von Klassen implementiert, die Stockwerke (z. B. aus DWG oder XAML Dateien) importieren, letztere liest Projektdaten ein. Beide Interfaces erben vom Interface `IBasicImporter`. Dieses Interface gibt allen Importern elementare Methoden, Eigenschaften und Events vor:

`GetFileExtension()` Der Dateityp bzw. die Dateiendung, den die Anwendung durch den Importer importieren kann, wird hier abgefragt (z. B. bin, xml etc.).

`GetFileDialogFilter()` Diese Zeichenkette dient zum Filtern der angezeigten Dateien in einem `OpenFileDialog`. Dadurch werden dem Benutzer nur Dateien des entsprechenden Typs angezeigt.

`IsCorrectFormat(String file)` Eine Datei, die importiert werden soll, kann beschädigt sein, nicht das erwartete Format haben oder eine abweichende interne Datenstruktur aufweisen. Diese Methode sollte *vor* dem Parsen aufgerufen werden und gibt einen booleschen Wert über die Korrektheit der Datei zurück.

`ParseFile()` Der Importer wird veranlasst, die Datei einzulesen und die Entitäten zu generieren. Dieser Prozess ist in einem Thread ausgelagert. Die im Folgenden erklärten Events geben während des Prozesses Auskunft über den Bearbeitungsstatus.

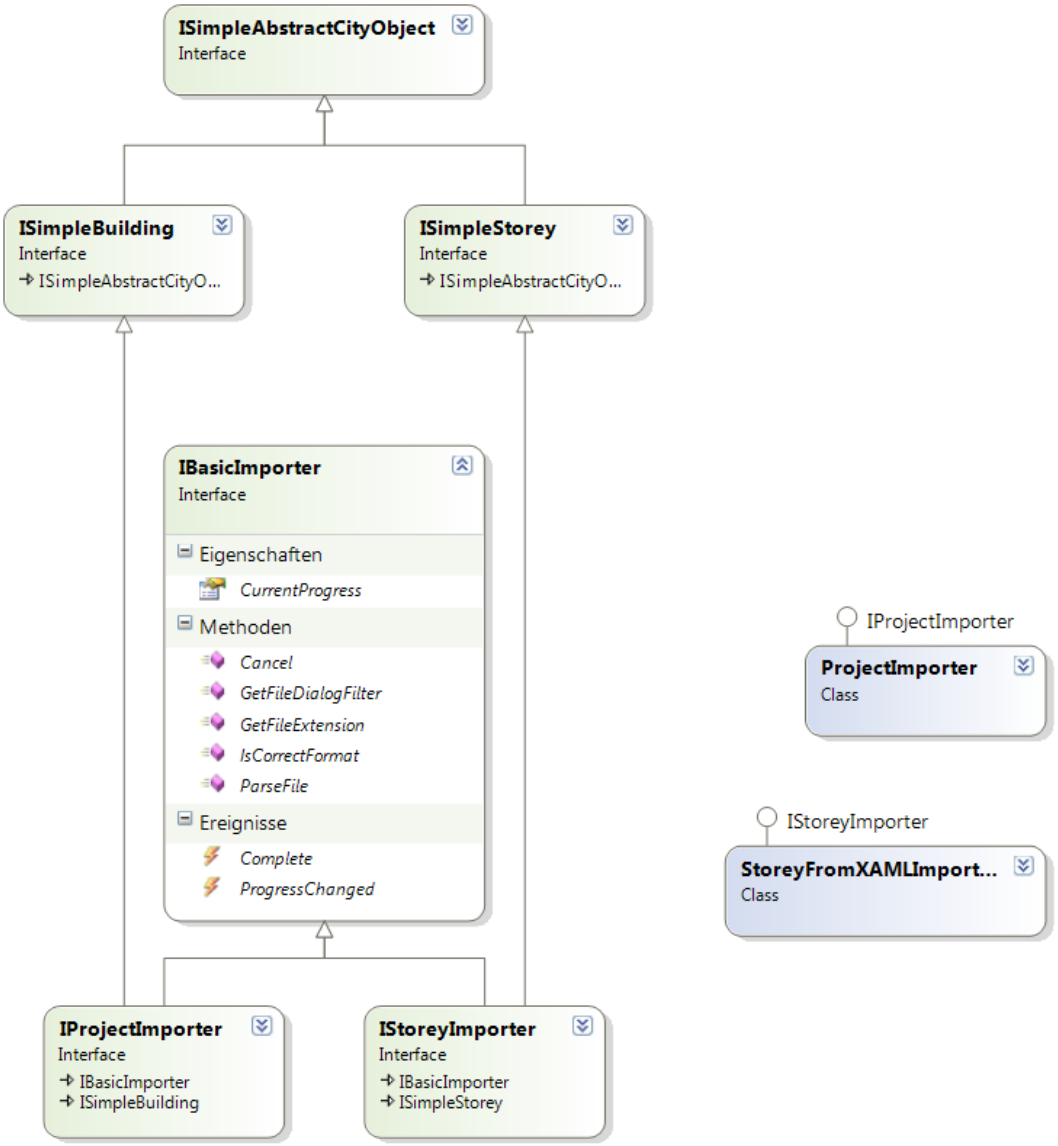

Abbildung 4.9: Whitebox-Sicht Import-Paket mit Zugriff auf Interfaces des BuildingModel-Pakets

`Cancel()` Während des Parsens kann ein Abbruch (z. B. durch den Benutzer) (asynchron) herbeigeführt werden.

Event `ProgressChanged` Das Event `ProgressChanged` informiert darauf registrierte Instanzen über den aktuellen Bearbeitungsstatus. Im AnnotationTool dient es z. B. dazu, die Fortschrittsanzeige zu aktualisieren.

Event `Completed` Das Event `Completed` signalisiert das Ende der Bearbeitung.

Das Interface `IStoreyImporter` fasst die Interfaces `IBasicImporter` und `ISimpleStorey` zusammen, indem es sie implementiert. Durch letzteren Typ kann der "StoreyImporter" wie ein "Storey" – zu deutsch Stockwerk – behandelt werden. Dies verdeutlicht auch das Ziel dieses Importers, nämlich ein Stockwerk aus einer Datei zu importieren und sich darauf wie ein solches zu verhalten. Ebenso erleichtert es dem Entwickler einer neuen Implementation des Importers, durch die vorgegebenen Methoden von `ISimpleStorey` die wichtigsten Schritte des Importprozesses klar zu strukturieren. Das Interface `IProjectImporter` verhält sich analog zum Interface `IStoreyImporter`.

Realisierung des `IStoreyImporter` Interfaces

In der Motivation sprach der Autor von der "Veredelung" eines "Informationsrohstoffes" – hier sind es die DWG Vektordaten der einzelnen Stockwerke eines Gebäudes der Hochschule für Angewandte Wissenschaften Hamburg, die "veredelt" werden sollen. Die folgende Ausführung bezieht sich stets auf den Aufbau dieser DWG Daten.

Abbildung 4.10 zeigt den Inhalt einer solchen CAD DWG Datei. Der Übersicht halber wurden einige Details für die Darstellung entfernt. Vektordaten werden in DWG Dateien auf "Layern" hinterlegt – jeder Layer enthält die ihm zugewiesenen Vektordaten (z. B. enthält der Layer "Fenster" die entsprechenden Vektordaten des Stockwerkes).

Die aufwendigsten Schritte im Importprozess werden durchgeführt, um die `GroupMember` – Räume eines Stockwerkes – zugreifenden Instanzen bereitzustellen. Die Extraktion des Raumnamens (`Id`) und der Funktion (`Function` [Treppe, Aufzug, "normaler Raum"]) ist aus den gegebenen Daten möglich, wenn auch sehr aufwendig. Zumindest ist jedoch die Geometrie der Räume zu extrahieren, wie im `StoreyFromXAMLImporter` geschehen. Die anderen Felder eines Raumes sollten mit einem Standardwert initialisiert werden.

Zudem finden sich auf dem Layer "Türe" [sic] Vektordaten der Türen (`SimpleDoor`). Eine Tür besteht hier meist aus zwei Vektoren, die zusammengefügt werden müssen, um eine komplette Türgeometrie zu ergeben. Daraufhin sollte erkannt werden, welche zwei Räume an eine Tür angrenzen und diesen – entsprechend des `BuildingModels` – zugewiesen werden.

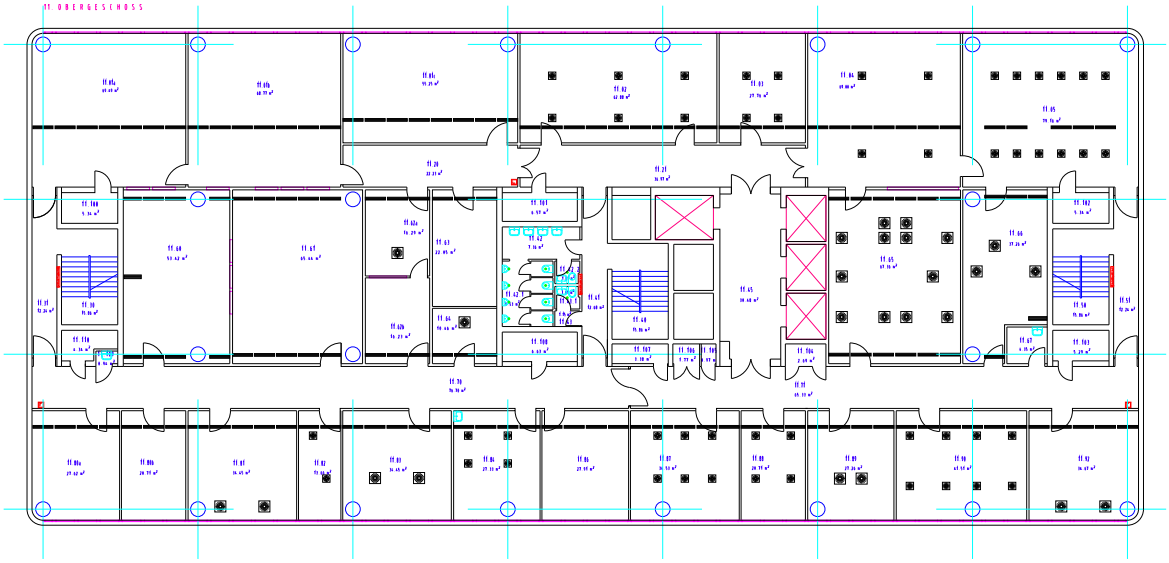

Abbildung 4.10: Vektordaten der CAD Datei *OG11.DWG*

Die Räume (`SimpleRoom`) können wiederum Mobiliar (`SimpleBuildingFurniture`) enthalten. Aus den gegebenen CAD Daten wäre es möglich (wenngleich nach dem angestrebten Einsatzziel wenig sinnvoll) Mobiliar und feste Installationen wie Wandhydranten, Fluchtwegschilder, Heizungskörper zu extrahieren.

Die Realisierung `StoreyFromXAMLImporter` des Interfaces `IStoreyImporter` erfordert zum Import eine vorverarbeitete Datei (Details Kap. 6.1 S. 107 im Anhang) – die DWG Datei ist hierbei in eine XAML Datei umzuwandeln, wobei die Stockwerksbreite normiert und der Zugriff auf einzelne Layer (z. B. Raumpolygone, Türpolygone) ermöglicht wird. Eine **Normierung** der Stockwerksausmaße ist unbedingt auch von weiteren Implementationen des `IStoreyImporter` Interfaces vorzunehmen[5].

Abschließend lässt sich sagen, dass der auf den Import folgende manuelle Nachbearbeitungsaufwand vom Umfang und der Güte der Implementierung eines Importers abhängt. Je umfangreicher ein Importer, desto mehr Details bis hin zu Mobiliar und Türen werden extrahiert. Je höher die Güte, desto weniger Fehler müssen manuell korrigiert werden. Da Fehler aber nie ausgeschlossen werden können, wurden bereits in der Analyse Möglichkeiten zur manuellen Nachbearbeitung gefordert. Diese werden im Kapitel *AnnotationTool* (S. 83) näher erläutert.

[5] Wenngleich mit Vektordaten gearbeitet wird, wird dennoch eine Ausgangsgröße von `1100px` als Stockwerksbreite vorgeschlagen

Paket: `Export`

Das *Export Paket* (siehe Abb. 4.11) stellt Klassen zur Verfügung, die den Export des Gebäudemodells ermöglichen, sowie eine Schnittstelle für weitere Export-Klassen (`IProjectExporter`).

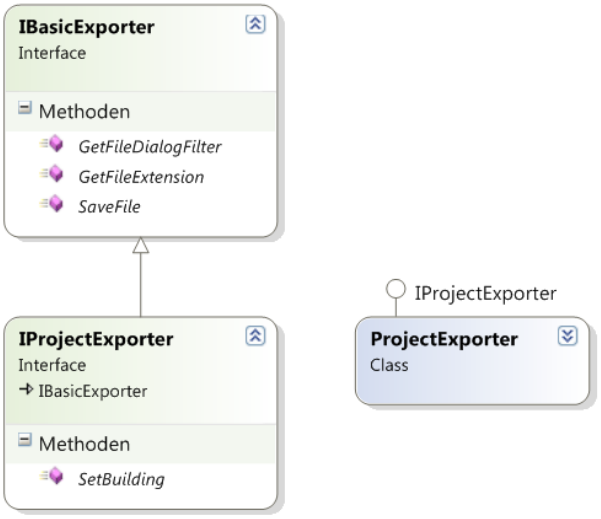

Abbildung 4.11: Whitebox-Sicht Export-Paket mit Zugriff auf Interfaces des BuildingModel-Pakets

Zum Export des gesamten Projektes ist die Übergabe einer Instanz vom Typ `ISimpleBuilding`, sowie des Zielspeicherortes erforderlich. Da ein Gebäude hierarchisch aufgebaut ist, genügt die Übergabe einer Gebäudeentität als “Wurzelelement”, um alle in ihr enthaltenen Entitäten in einem rekursiven Abstieg mit zu serialisieren.

Denkbar wären auch Exporter, die nur Teile eines Gebäudes exportieren. Da das Gebäudemodell gleichsam eine Inventarliste ist, könnte bspw. ein Drucker, Raum, eine Person zu Dokumentationszwecken o.ä. exportiert werden.

XML als Exportformat

Als Projekt-Dateiformat wird die *eXtensible Markup Language* (XML) verwendet. Der hierarchische Aufbau einer XML-Datei eignet sich gut zum Speichern des ebenfalls hierarchisch aufgebauten Gebäudemodells[6]. XML Dokumente ermöglichen den Austausch der Daten zwischen unterschiedlichen Anwendungen – unabhängig vom eingesetzten Framework und der Programmiersprache. Da XML-Dokumente, im Gegensatz zu Datenbanken, nur aus Text bestehen, wird eine höhere Portabilität erreicht.

[6] Ein Gebäude enthält Stockwere, die wiederum Räume enthalten, die wiederum Mobiliar enthalten usw.

Die Beschreibung der erlaubten Elemente und des möglichen Aufbaus eines XML-Dokumentes erfolgt durch eine *XML Schema Definition* (XSD). In anderen Programmiersprachen wird es dadurch möglich, Klassen aus einer XSD zu generieren und auf ihnen aufbauend das entsprechende XML-Dokument einzulesen und weiterzuverwenden. Das Toolkit stellt eine solche XSD für die Projektdatei zur Verfügung.

Paket: `EventModel`

Zukünftigen Entwicklern soll mit dem Eventmodell eine Struktur zur Verfügung stehen, die es erlaubt, dem Modell eigene, konkrete Trigger, Events und Aktionen auf einfache Weise hinzuzufügen.

In diesem Kapitel werden zunächst die *Grundlagen des Eventmodells* (S. 69) erarbeitet. Danach findet eine *Erläuterung der wichtigsten Begriffe* (S. 70) statt. Im Kapitel *Trigger – Auslöser von Events* (S. 72) wird näher auf Trigger eingegangen. Darauf folgen die Kapitel *Grundlegende Eventtypen* (S. 73), *Realisierung des Eventmodells* (S. 74) und *Der* `ToolKitEventManager` (S. 76), der wiederum in kleinere Abschnitte gegliedert ist.

Grundlagen des Eventmodells

Im Kapitel *Analyse* ist deutlich geworden, dass der Benutzer im AnnotationTool einzelnen Entitäten in einem Eventeditor beliebiges *Verhalten* zuweisen können soll. Dieses Verhalten kommt schließlich in einer Endbenutzerapplikation wie dem WelcomeScreen zum Tragen, macht sich aber im AnnotationTool selbst nicht direkt bemerkbar – diese Anforderung ist zugleich auch die größte Herausforderung an das Eventmodell[7].

Abb. 4.12 stellt den Sachverhalt graphisch dar und hebt zur Verdeutlichung die einzelnen Akteure hervor. Der *Entwickler* implementiert mögliches Verhalten von Entitäten – wie dies genau geschehen soll ist Teil des Eventmodells und wird in den folgenden Kapiteln beschrieben. Der *Administrator* fügt mit dem EventEditor (s. Kap. 4.4.2 S. 88) des AnnotationTools den Entitäten das gewünschte Verhalten zu. Im WelcomeScreen werden die entsprechenden Entitäten mit dem zugewiesenen Verhalten versehen – sie werden sozusagen “aktiviert” und reagieren damit z. B. auf Interaktionen eines Benutzers.

Es ergeben sich für das AnnotationTool, den WelcomeScreen und die ToolkitLib bzgl. des Eventmodells also unterschiedliche Aufgaben:

[7] Einfacher wäre es sicherlich das Eventhandling direkt in den Quelltext zu codieren – das entspräche aber nicht dem Gedanken eines Toolkits, wie ein Baukastensystem zu agieren und möglichst viele Freiheitsgrade offen zu lassen.

AnnotationTool: Das AnnotationTool stellt einen Eventeditor bereit, der es dem Benutzer erlaubt, Entitäten mit Verhalten zu annotieren. Die Verwaltung des Verhaltens einzelner Entitäten übernimmt der `ToolkitEventManager` (Kap. 4.4.2 S. 76). In einer Projektdatei werden die Gebäudedaten und Annotationen serialisiert.

WelcomeScreen: Der WelcomeScreen versieht beim Deserialisieren der Projektdatei die entsprechenden Entitäten mit Hilfe des *EventManagers* mit dem festgelegten Verhalten und stellt sie dar.

ToolkitLib: Die ToolkitLib vermittelt zwischen beiden Applikationen. Sie stellt das zu Grunde liegende Eventmodell bereit, sowie den *EventManager*, der das zugewiesene Verhalten der Entitäten in Listen und Dictionarys verwaltet.

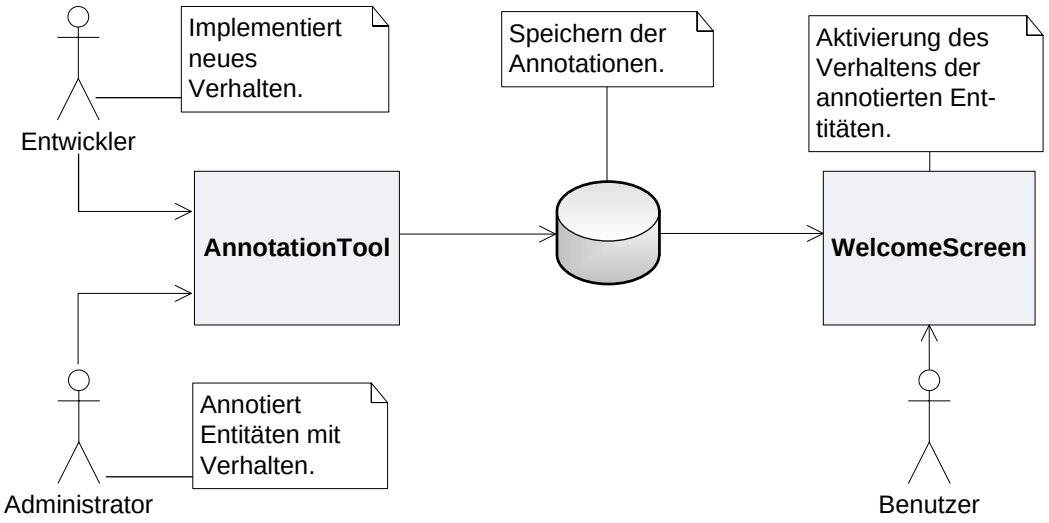

Abbildung 4.12: Annotation einer Entität mit Verhalten

Erläuterung der wichtigsten Begriffe

Trigger werden als Auslöser von *Events* betrachtet. Die Ursache für das Auslösen eines Events durch einen Trigger kann vielfältig sein und wird im Kapitel *Trigger – Auslöser von Events* (S. 72) genauer erörtert.

Ein **Event** kann als “digitales Trägermedium” von Informationen angesehen werden. Die Art dieser Informationen bestimmen den Typ eines Events und setzen sich aus der Quelle eines Events und weiterer Eventargumente zusammen. Die Eventargumente haben eine Schlüsselfunktion bzgl. des Eventtyps und werden ebenfalls im oben genannten Kapitel näher erläutert.

Damit eine Entität mit **Verhalten** versehen werden kann, d.h. auf Events reagieren kann, muss sie zunächst auf ein Event eines Triggers registriert werden. Wird dieses Event ausgelöst, bearbeitet eine **Behandlungsroutine** (*Event-Handler*) das Event – diese Behandlungsroutinen werden im Folgenden auch **Aktionen** bzw. *Actions* genannt. Das Besondere des Eventmodells ist es u.a., dass einem Trigger, bzw. dem Event eines Triggers, (im AnnotationTool) *mehrere* Actions zugewiesen werden können.

Der **`ToolkitEventManager`** verwaltet die Zuordnung eines Triggers und der dazugehörigen Aktion(en). Dieser Event-Manager sollte nicht mit dem Event-Manager der Laufzeitumgebung verwechselt werden (Details siehe Kapitel Kap. 4.4.2 S. 76) – um Verwechslungen auszuschließen wurde das Präfix “Toolkit” gewählt.

Grundverständnis des Eventablaufs

Die Fähigkeit einer Entität auf asynchrone, äußere Stimuli zu reagieren wird durch eine Ereignisbehandlungsroutine bzw. einen *Event-Handler* erworben. Ein *Event* hat einen Auslöser bzw. *Trigger* und eine darauf folgende *Aktion*. In Abb. 4.13 wurde der Ablauf schematisch stark vereinfacht dargestellt.

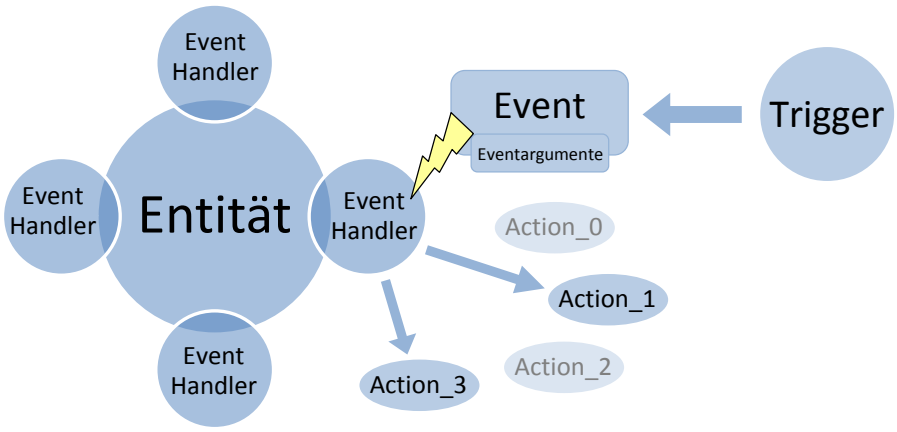

Abbildung 4.13: Schematischer Ablauf beim Auftreten eines Events: Eine Entität reagiert auf ein Event mit zwei Aktionen.

Zunächst löst ein Trigger ein Event aus. Dieses Event transportiert als Eventargumente die Quelle (`source`) des Events, als auch eine Instanz einer EventArgument-Klasse (abgeleitet von der Klasse `EventArgs`), die weitere spezifische Informationen kapselt (z. B. eine Klick-Position, ein Zielobjekt, die Anzahl Klicks etc. pp.). Die “Actions”, mit denen eine Entität auf ein Event reagiert, wurden im AnnotationTool durch den Administrator festgelegt.

Trigger – Auslöser von Events

Als Trigger können nicht nur Ereignisse durch die Interaktion eines Benutzers mit dem System in Betracht gezogen werden, sondern auch Ereignisse systeminterner oder -externer Komponenten (z. B. Balanceboard, Ablaufen eines Timers, ein RPC Aufruf, ein bestimmter Systemzustand etc.). Das Interagieren des Benutzers ist hierbei nicht auf die Berührung des Touchscreens durch den Nutzer beschränkt, sie kann durch beliebige multimodale Interaktionen erfolgen (z.B. Sprache oder Gesten).

Die möglichst generische Beschreibung und Struktur solcher Trigger soll zum einen eine große Vielfalt möglicher Interaktionsmodi abbilden, zum anderen zu klar strukturierten, wiederverwendbaren, einfach zu handhabenden und leicht zu erweiternden Anwendungen führen.

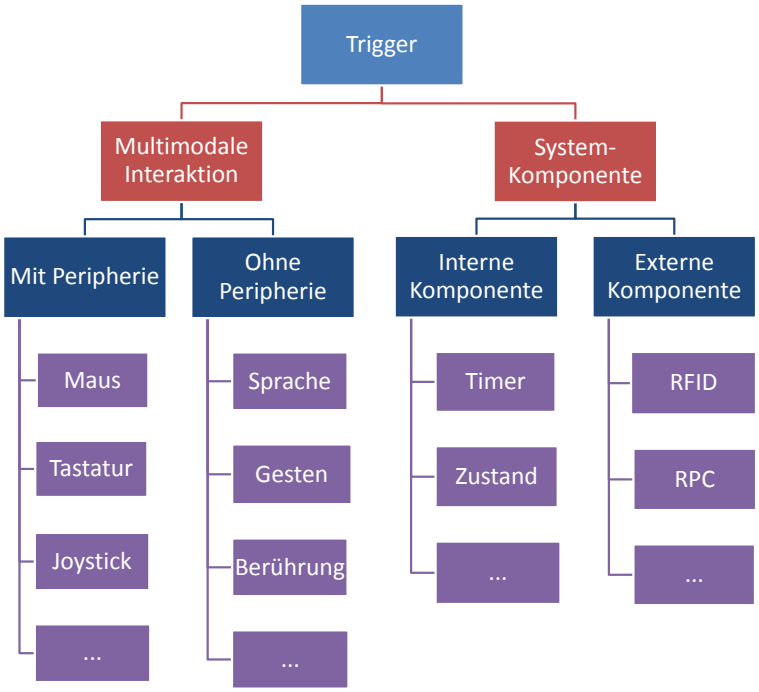

Abbildung 4.14: Kategorisierung möglicher Trigger

In Abb. 4.14 werden Beispiele möglicher Trigger aufgezeigt und kategorisiert. Die Beispiele sollen nur einen Denkanstoß geben, denn ein Trigger an sich ist relativ uninteressant – wichtiger ist der Typ des Events, den ein Trigger auslöst. Darauf wird im folgenden Kapitel eingegangen.

Grundlegende Eventtypen

Events bzw. Eventtypen unterscheiden sich in den Informationen, die sie in Form von Eventargumenten an den Empfänger übermitteln. Der Typ des Eventarguments bestimmt somit den Typ des Events. Als direkten oder indirekten Basistyp haben alle Eventargumentklassen im verwendeten Framework die Klasse `EventArgs` – wie diese Gegebenheit im Eventmodell ausgenutzt wird, wird im nächsten Kapitel beschrieben. In den Unterklassen werden dann diejenigen Felder hinzugefügt, die die Informationen eines Triggers an einen Empfänger übermitteln sollen.

In dieser Arbeit wurden einige grundlegende Eventtypen erarbeitet, die im Folgenden aufgelistet werden. Einige davon wurden von Rahimi und Vogt [2008] – einer Arbeit über gestenbasierte Computerinteraktion – inspiriert.

SelectionEvent: "Selektions-Event", das eine Entität selektiert bzw. auswählt. Dieses Event wird z. B. durch einen Sprachbefehl ("Wähle XYZ aus!"), einen Mausklick oder eine Touchscreen-Berührung ausgelöst.

PositioningEvent: Das "Positionierungs-Event" liefert Informationen über den aktuellen Aufenthaltsort von Entitäten (z. B. Personen). Diese Informationen werden meist in bestimmten Zeitintervallen aktualisiert. Diese Zeitintervalle können ziemlich groß (im Sekundenbereich) oder auch sehr klein sein. Je nach Anforderungen wirken die Bewegungen der Entität dann stetig oder stockend. Letzteres kann durch Interpolation von Zwischenpositionen ausgeglichen werden. Wird die Geometrie und Semantik des Gebäudemodells berücksichtigt, könnten zudem "unlogische" Bewegungen durch Wände vermieden werden.

DropEvent: Befindet sich eine Entität A über einer anderen Entität B und wird dort fallen gelassen ("drop"), so kann B ein *DropEvent* erhalten mit A als Quelle. Dieses Event kann beispielsweise auf ein *PositioningEvent* folgen.

ScaleEvent: Ein "Skalierungs-Event" beinhaltet Informationen über den Faktor der Größenänderung einer Entität.

RotateEvent: Ein "Rotations-Event" veranlasst eine Entität sich um einen bestimmten Winkel um die eigene Achse zu drehen. Die zugehörigen Eventargumentparameter enthalten Informationen über den Grad und die Richtung der Rotation.

PrintEvent: Wie die Bezeichnung des Events bereits andeutet, löst ein *PrintEvent* einen Druckauftrag aus. Die Eventargumentparameter beinhalten spezifische Einstellungen des Druckauftrages.

Dies waren nur einige mögliche Beispiele – die Liste erhebt also keinen Anspruch auf Vollständigkeit.

Details des Eventmodells

In den vorangegangenen Kapiteln wurden die Grundlagen für das nun Folgende erarbeitet, in dem auf das `EventModel` Paket der ToolkitLib im Detail eingegangen wird.

An Hand eines Beispiels werden die Akteure bzw. Klassen, die ein Event erzeugen und behandeln, den Ablauf und Aufbau verdeutlicht. Außerdem dienen die *Sample*-Klassen künftigen Entwicklern als konkrete Implementierungsbeispiele.

Die Abbildung 4.15 zeigt nur einen kleinen Ausschnitt aller Klassen des `EventModel` Paketes und beschreibt am Beispiel des `SampleTrigger`s den grundsätzlichen Aufbau des Eventmodells.

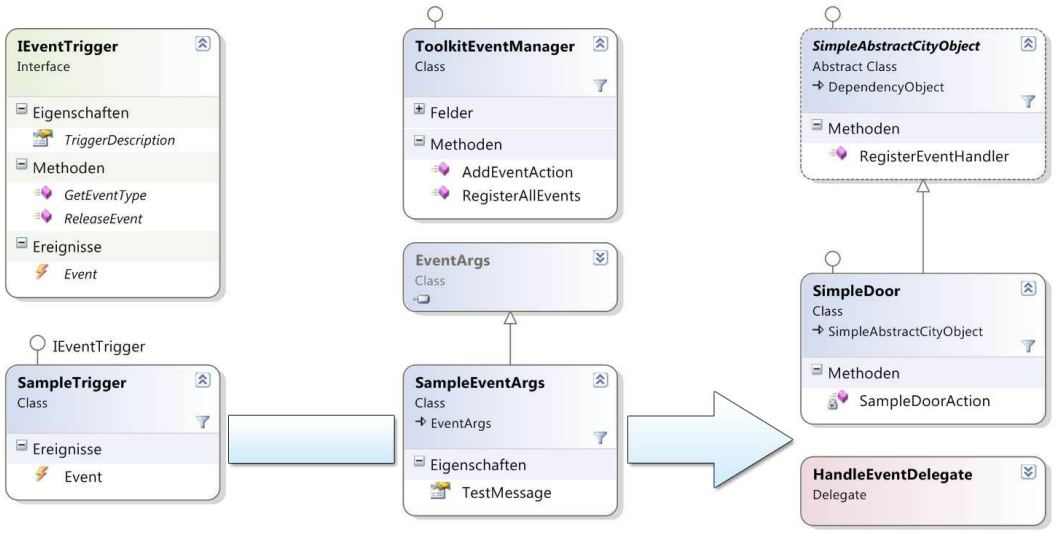

Abbildung 4.15: Ausschnitt des Klassendiagramms des Eventmodells

In Abb. 4.15 sind die Klassen und Interfaces dreispaltig angeordnet: 1. Spalte: Trigger mit Interface `IEventTrigger`, 2. Spalte: Eventargumenthierarchie und ToolkitEventManager, 3. Spalte: Empfänger des Events. Der chronologische Ablauf beginnt in Spalte 1 und endet in Spalte 3 und wird im Folgenden beschrieben.

Der `SampleTrigger` löst ein Event mit Eventargumenten vom Typ `SampleEventArgs` aus. Eine Instanz der Entität `SimpleDoor`, die auf das Event registriert wurde, reagiert darauf mit der Action `SampleDoorAction()`.

Jeder Trigger implementiert das Interface `IEventTrigger` (s. Listing 4.1). Die Vorgaben des Trigger werden im Folgenden erklärt:

Event Jeder Trigger deklariert einen Eventhandler mit Namen `Event`. Die Eventargumente sind vom Typ `EventArgs` – somit können auch Unterklassen (z. B. `SampleEventArgs`) mit dem Event verschickt werden. Beim Empfang des Events wird dann zum konkreten Typ gecastet. Details hierzu werden im Folgenden Kapitel erörtert.

GetEventType() Gibt den tatsächlichen Eventargument-Typ zurück (z. B. `SampleEventArgs`).

TriggerDescription Enthält eine Kurzbeschreibung des Triggers.

```
1 public interface IEventTrigger
2 {
3     event EventHandler<EventArgs> Event;
4     Type GetEventType();
5     String TriggerDescription { get; }
6 }
```

Listing 4.1: Interface `IEventTrigger`

Am Beispiel des `SampleTrigger`s wurde die grundlegende Struktur des Eventmodells exemplarisch erörtert und die wesentlichen Komponenten genannt: Ein Trigger löst ein Event eines bestimmten Typs aus, wodurch eine Entität, die dieses Event empfangen kann, darauf eine Aktion startet.

Das Sequenzdiagramm in Abb. 4.16 zeigt beispielhaft, wie die Hauptakteure (Trigger, Entität usw.) bei der Interaktion eines Benutzers mit dem WelcomeScreen agieren. In diesem Beispiel löst ein Benutzer durch einen Sprachbefehl bei einem “SelectionTrigger” ein Event aus, welches das `SelectionEventArgs` Eventargument transportiert. Mehrere, auf das Event des Triggers registrierte Entitäten empfangen anschließend das Event. Im Eventhandler prüfen die Entitäten, ob sie darauf reagieren müssen, falls dem so ist, wird die entsprechende Aktion gestartet. In diesem Beispiel ist “Entität XYZ” das Ziel der Selektion und reagiert darauf, indem es sich “highlighted” und weitere Aktionen ausführt (“`DoSomethingElse(...)`” im Beispiel).

Denkbar wäre auch ein Sprachbefehl, der alle Entitäten eines Entitättyps selektiert. Der Sprachbefehl “alle Drucker selektieren” würde also dazu führen, dass alle Drucker-Entitäten beim Empfang des Events die entsprechende Aktion ausführen.

Die im Kapitel *Grundlegende Eventtypen* (S. 73) erarbeiteten Typen (u.a. `SelectionEvent`, `DropEvent`, `ScaleEvent` etc.) erweitern das Klassendiagramm in Abb. 4.15 entsprechend. Da es pro Eventtyp einen oder mehrere Trigger und Eventargumenttypen gibt, wurde darauf verzichtet ein vollständiges Klassendiagramm abzubilden. Das dahinterliegende Konzept wurde am Beispiel erörtert und wurde bei den anderen Eventtypen beibehalten.

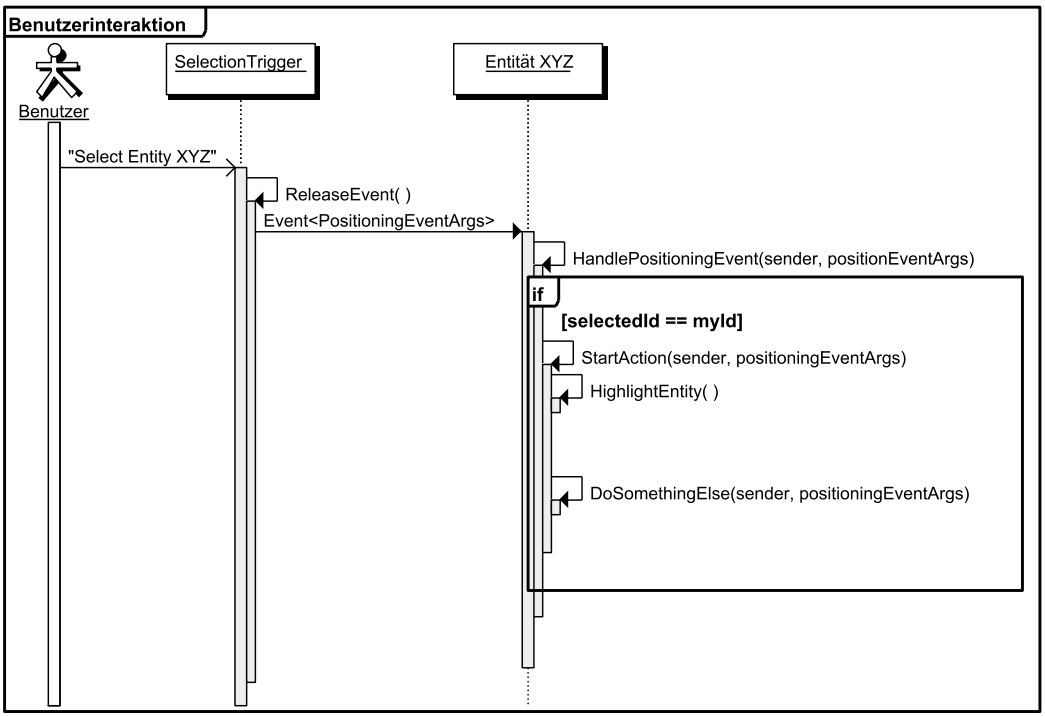

Abbildung 4.16: Sequenzdiagramm – Selektion einer Entität durch den Benutzer

Das Verhalten der Entitäten kann mit dem Eventmodell bereits im Code fest “verankert” werden. Da aber die Zuordnung von Trigger und Aktion erst im AnnotationTool stattfinden soll, müssen weitere Maßnahmen ergriffen werden, die im folgenden Kapitel *Der `ToolKitEventManager`* (S. 76) erörtert werden. Nichtsdestotrotz bleibt die Möglichkeit offen, *ohne* Berücksichtigung des `ToolKitEventManager`s Events und Aktionen zu implementieren und bei einem Entitätstyp (fest im Code) zu registrieren.

Der `ToolKitEventManager`

Zunächst einmal ist klarzustellen, dass der hier beschriebene `ToolKitEventManager` nicht mit dem Eventmanager der Laufzeitumgebung verwechselt werden sollte. Letzterer kümmert sich darum, dass diejenigen Instanzen, die sich auf ein Event angemeldet haben (zumeist durch den Zuweisungsoperator “+=” in div. Sprachen) beim Auslösen dieses Events “informiert” werden, indem ihr Event-Handler mit den entsprechenden Eventargumenten aufgerufen wird.

Der in dieser Arbeit entwickelte `ToolKitEventManager` verwaltet grundsätzlich die Zuordnung des Triggers eines Events zu einer darauf folgenden Behandlungsroutine einer Entität. In der Regel werden bei der Entwicklung einer Anwendung diese Zuordnungen bereits

vom Programmierer fest implementiert – das Ziel des Eventmodells ist es aber, dem Benutzer des AnnotationTools zukünftig eine Reihe an Triggern und mögliche, darauf folgende Aktionen zur Verfügung zu stellen, so dass dieser einer Entität *beliebige Kombinationen* an Triggern und Aktionen zuordnen kann – zur Laufzeit werden darauf in einer Applikation wie dem WelcomeScreen diese Events bei den Entitäten registriert.

Der `ToolKitEventManager` aus Sicht des AnnotationTools

Im Folgenden wird der Gebrauch des Eventmodells im AnnotationTool betrachtet (vgl. hierzu rückblickend Abb. 4.12 S. 70). Hier werden Entitäten vom Administrator mit Verhalten annotiert. Der EventEditor und ToolKitEventManager stehen in diesem Kapitel im Vordergrund.

Im vorherigen Kapitel wurde der Zusammenhang von Trigger, Event und Action dargestellt. Es ist klar geworden, dass eine direkte Beziehung zwischen Trigger → Event sowie Event → Action besteht. Somit gibt es die transitive Beziehung Trigger → Action – diese spiegelt sich auch im `EventEditor` wieder. Er gibt dem Benutzer die Möglichkeit, in der graphischen Oberfläche des AnnotationTools einer Entität Paare an Trigger → Actions zuzuweisen. Schaut man rückblickend auf Abb. 4.12, wird nun deutlich, was mit dem Annotieren einer Entität mit Verhalten gemeint ist.

In Abb. 4.17 sind die Klassen `ToolKitEventManager` und `EventAction` zu sehen, die im Folgenden erläutert werden.

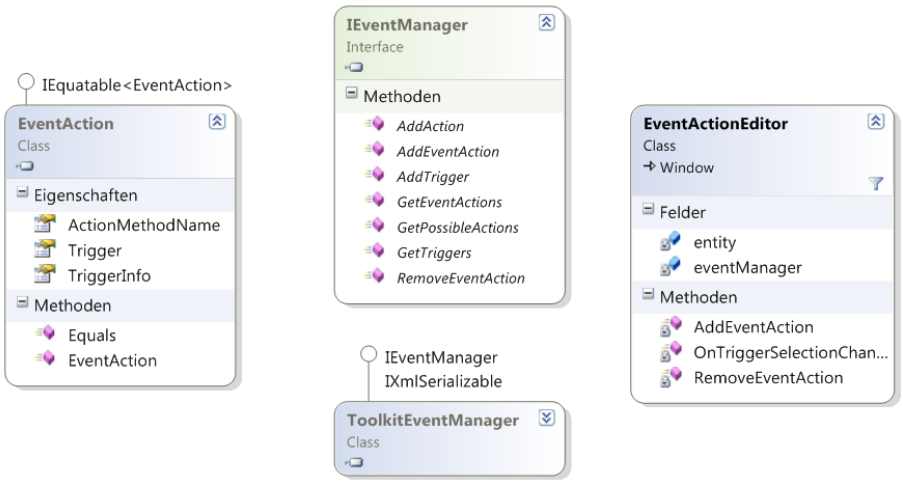

Abbildung 4.17: `ToolkitEventManager` und `EventAction` Klasse

Ein Trigger-Action Paar wird durch die Klasse `EventAction` abgebildet[8]. Sie kapselt, wie die Bezeichnung vermuten lässt, den ausgewählten Trigger und die zugewie-

[8]Korrekter wäre die Bezeichnung “TriggerAction”; da “EventAction” aber der umgangssprachlichen Verwen-

sene Action in sich und bietet diverse Methoden für den Zugriff an. Das generische Interface `IEquatable<EventAction>` bewirkt, dass `EventAction` Instanzen per `Equals()` Methode verglichen werden können und so z. B. überprüft werden kann, ob die *gleiche* Instanz bereits in einer Liste vorhanden ist.

Der `ToolkitEventManager` verwaltet eine *Trigger-Liste* mit konkreten Triggern (`triggers` in Abb. 4.17), ein *EventAction-Dictionary* mit den `EventAction`s, die einer Entität zugewiesen wurden und ein *Meta-Dictionary* für die Zuordnung von Eventtyp und Action (`metaDictionary`). Das Einbringen einer Meta-Ebene ist deshalb erforderlich, weil die Zuordnungen von Event und Aktion, wie oben angedeutet, erst zur Laufzeit (des WelcomeScreens) hergestellt wird und nicht bereits vor Applikationsstart im Programmcode festgelegt ist.

Im Folgenden werden die Listen bzw. Dictionarys genauer beschrieben:

Trigger-Liste: Die Trigger-Liste verwaltet alle implementierten Trigger. Diese werden im Konstruktor des `ToolkitEventManager` mit der Methode `AddTrigger(IEventTrigger trigger)` hinzugefügt. Ein Trigger implementiert das Interface *IEventTrigger*, das die Methode `GetEventType()` vorsieht. Hierdurch kann abgefragt werden, von welchen Typ das Event ist, das der Trigger auslöst. Außerdem überschreibt ein Trigger die `ToString()` Methode, so dass im `EventEditor` des AnnotationTools eine sinnvolle Kurzbeschreibung des Triggers angezeigt wird.

EventAction-Dictionary: Im EventAction-Dictionary werden die `EventAction`s, die einer konkreten Entität im `EventEditor` zugewiesen wurden, gespeichert. Die `EventAction`s werden in einer generischen Collection verwaltet (in einer sog. `ObservableCollection`), die die Änderungen an ihr, ohne das Zutun des Programmierers, an die Benutzeroberfläche weiterleitet. Fügt ein Benutzer eine `EventAction` hinzu, oder entfernt diese, muss die graphische Oberfläche also nicht zusätzlich aktualisiert werden, da dies vom Grafikframework übernommen wird.

Meta-Dictionary: Das Meta-Dictionary speichert die Zuordnung von Eventtypen und die implementierten Actions eines Entitättyps. Folgende Methodensignatur veranschaulicht die verwalteten Informationen:

```
void AddAction(Type targetEntityType,
               Type actionEventType,
               String actionMethodName, String actionDescription)
```

Der Parameter `targetEntityType` ist der Entitätstyp (z. B. `ISimpleRoom`), der eine Action anbietet (ein `Type` kann mittels Reflection ermittelt werden [`typeof`

dung eher entspricht und die erste Assoziation bei der Bezeichnung “EventAction” dem eigentlichen Sinn näher kommt, wurde diese Bezeichnung gewählt.

Operator]). Über den Parameter `actionEventType` wird der Typ des Events angegeben, der von der folgenden Methode `actionMethodName` empfangen werden kann. Damit im `EventEditor` eine sinnvolle Beschreibung angezeigt werden kann, wird eine ebensolche über den Parameter `actionDescription` an das Meta-Dictionary übergeben. Bis auf die `actionDescription` könnten alle anderen Informationen per Reflection aus der `BuildingModel` Klassenhierarchie automatisch extrahiert werden. Auf Grund der notwendigen textuellen Beschreibung einer Aktion hat sich der Autor aber dafür entschieden die Actions in einer Datenstruktur, die vom Entwickler mit den implementierten Actions versehen wird, zu verwalten.

Die Informationen in den genannten Datencontainern stammen also einerseits vom Administrator des AnnotationTools (EventAction-Dictionary) und andererseits vom Entwickler, der eigene Trigger, Eventtypen und Actions implementiert (Trigger-Liste und Meta-Dictionary). Werden die Daten aus der Trigger-Liste und dem Meta-Dictionary verknüpft, stehen alle notwendigen Informationen für das AnnotationTool und den WelcomeScreen zur Verfügung. Hierbei fungiert der Eventtyp – ähnlich einer Fremdschlüsselbeziehung in Datenbanken – als Bindeglied zwischen den beiden Datencontainern. Dadurch kann beispielsweise eine Abfrage formuliert werden, wie in Listing 4.2 zu sehen ist.

```
1 public Dictionary<String, String> GetPossibleActions(IEventTrigger
    trigger, Type targetEntityType)
2     {
3         /* (...) */
4         return metaDictionary[targetEntityType][trigger.GetEventType()];
5     }
```

Listing 4.2: Methode `GetPossibleActions` des `ToolkitEventManager`s

Die Methode `GetPossibleActions(...)` in Listing 4.2 gibt alle implementierten Actions zurück, mit denen ein bestimmter Entitätstyp (z. B. `ISimpleStorey`) auf das Event eines bestimmten Triggers reagieren kann. Von dieser Methode macht der *EventEditor* z. B. Gebrauch, wenn ein Benutzer aus der Liste möglicher Trigger einen auswählt – das Ergebnis der Abfrage wird in einer weiteren Liste angezeigt, in der die möglichen Actions, die auf den Trigger folgen können, angezeigt werden.

Der `ToolkitEventManager` implementiert die Interfaces `IEventManager` und `IXmlSerializable` (s. Abb. 4.17). Das Interface `IEventManager` faktorisiert die Add-, Get- und Remove-Methoden des EventManagers heraus, um ihn ggf. durch eine andere Realisierung, die bspw. die Daten in einer Datenbank verwaltet, ersetzen zu können. Durch das Interface `IXmlSerializable` kann, wie bereits erwähnt, der (De-) Serialisierungsprozess genau gesteuert werden. Hier gilt es das EventAction-Dictionary

beim Speichern der AnnotationTool Projektdatei zu serialisieren und im WelcomeScreen zu deserialisieren.

Nun stellt sich die Frage: Wie kann das den Entitäten im AnnotationTool annotierte Verhalten beim bzw. nach dem Deserialisierungsprozess im WelcomeScreen aktiviert werden? Die Antwort liefert das folgende Kapitel.

Der `ToolKitEventManager` aus Sicht des WelcomeScreens

Dieses Kapitel befasst sich mit dem WelcomeScreen auf der rechten Seite der Abb. 4.12 (S. 70). Die Ausgangssituation ist folgende: Im AnnotationTool wurden Entitäten mit EventActions versehen und darauf das Projekt serialisiert. Jetzt wird das Projekt im WelcomeScreen deserialisiert. Daraufhin werden die Entitäten entsprechend ihrer zugewiesenen EventActions auf die jeweiligen Events registriert. Wie das genau geschieht, wird im Folgenden erörtert.

Der ToolKitEventManager wird mit der Methode `RegisterAllEvents()` dazu veranlasst, die Entitäten auf die zugewiesenen Events zu registrieren. Zunächst werden hierzu alle Eintragungen des EventAction-Dictionary durchlaufen. Pro Entität steht nun eine Liste mit EventActions zur Verfügung. In Listing 4.3 ist zu sehen, wie die Events bei der Entität registriert werden.

```
1  public void RegisterAllEvents()
2  {
3      foreach (KeyValuePair<ISimpleAbstractCityObject,
           ObservableCollection<EventAction>> keyVal in entities)
4      {
5          ISimpleAbstractCityObject entity = keyVal.Key;
6          ObservableCollection<EventAction> eventActionList = keyVal.Value
               ;
7
8          foreach (EventAction eventAction in eventActionList)
9          {
10             // "Register"-Methode für Invoke vorbereiten
11             String registerMethodName = "RegisterEventHandler";
12             MethodInfo registerMethod = entity.GetType().GetMethod(
                   registerMethodName);
13             object[] parameters = new object[] { eventAction.Trigger,
                   eventAction.ActionMethodName };
14
15             // Methode per "Invoke" ausführen
16             registerMethod.Invoke(entity, parameters);
17         }
18     }
19 }
```

Listing 4.3: Methode `RegisterAllEvents` des `ToolkitEventManager`s

Mittels Reflection wird die Methode `RegisterEventHandler()`, die den Trigger und den Namen der Action-Methode übergeben bekommt, aufgerufen. Im Folgenden wird erklärt, was daraufhin geschieht.

Die Methode `RegisterEventHandler`

Die Methode `RegisterEventHandler`, die im ToolkitEventManager aufgerufen wurde, ist in der Basisklasse aller Entitäten implementiert (SimpleAbstractCityObject). Der Methodenrumpf ist in Listing 4.4 zu sehen.

```
107 public void RegisterEventHandler(IEventTrigger trigger, String
        actionMethodName)
108 {
109     HandleEventDelegate del = (HandleEventDelegate)Delegate.
            CreateDelegate(
110         typeof(HandleEventDelegate), this, actionMethodName);
111
112     trigger.Event += new EventHandler<EventArgs>(del);
113 }
```

Listing 4.4: Methode `RegisterSampleEventHandler`

Die übergebenen Parameter `trigger` und `actionMethodName` stammen aus der einer Entität zugewiesenen EventAction-Instanz. Um die Entität zur Laufzeit auf ein Event des `trigger`s zu registrieren, und darauf eine bestimmte Action – z. B. die `SampleAction` in Listing 4.5 – folgen zu lassen, werden *Reflection* und *Delegaten* verwendet. Ohne diese Sprachmittel würde man zu *jeder* neu hinzugefügten Aktion eine if- bzw. switch-Abfrage in den Rumpf der Methode hinzufügen müssen – siehe dazu folgendes Beispiel:

```
if(actionMethodName.Equals("SampleAction"))
    trigger.SampleEvent += new EventHandler<EventArgs>(SampleAction);
```

Das Problem ist zudem, dass ohne Delegaten der Name einer Action in drei Duplikaten im System niedergeschrieben werden müsste – diese Art der Kopplung soll auf Grund der Fehleranfälligkeit bei Änderungen durch den Einsatz von Delegaten vermindert werden.

Ein *Delegat* ist ein “Prototyp einer Funktion” und kann mit dem Konzept des *Funktionszeigers* in C/C++ verglichen werden. Delegaten haben eine Signatur ähnlich der von Methoden. Die Methoden, die auf einen Delegaten zeigen sollen, müssen dieser Signatur entsprechen.

Ein solcher Delegat wird in Listing 4.4 durch die statische Methode `CreateDelegate` erzeugt. U.a. wird der `String` Parameter `actionMethodName` an die Methode übergeben

– dies ist auch der Grund warum der ToolKitEventManager den Methodennamen als String verwaltet. Auf diese Weise können weitere Actions leicht implementiert werden – sie müssen lediglich noch im ToolKitEventManager angemeldet werden und können dann ohne weiteres Zutun im AnnotationTool vom Benutzer verwendet werden!

Nun wird auch klar, worin der Vorteil besteht, dass im Interface `IEventTrigger` die Eventargumente vom Basis-Typ `EventArgs` sind: *Jedes* Event, dessen Eventargumente von `EventArgs` ableiten (das tun alle), kann mit dieser Methode auf eine Entität registriert werden! Andernfalls gäbe es für *jeden* Eventtyp eine eigene "Register"-Methode – dies bedeutete (neben wesentlich mehr Code) auf Grund der sehr ähnlichen Methodenrümpfe eine starke Kopplung und somit großen Aufwand und hohe Fehleranfälligkeit bei Änderungen der "Register"-Methoden.

Die Actions

Wie bereits erwähnt sind die Informationen eines Events in den Eventargumenten gekapselt. Um an diese Informationen zu gelangen, wird zunächst vom Typ der Oberklasse `EventArgs` zum erwarteten Typ gecastet – in Listing 4.5 beispielsweise zum Typ `SampleEventArgs`.

```
107 void SampleAction(object sender, EventArgs e)
108 {
109     SampleEventArgs eventParam = (SampleEventArgs)e;
110     Console.WriteLine("DOOR: Message Received: \"{0}\"", eventParam.
            TestMessage);
111 }
```

Listing 4.5: Eine Beispiel-Implementierung einer *Action*

Dies verdeutlicht nochmal den Sinn des Meta-Dictionarys – es verhindert, dass einem Event eine Action zugewiesen wird, die unpassende Eventargumente enthält, wodurch das Casting fehlschlagen würde.

Eine neu implementierte *Action* muss also im ToolKitEventManager beim Meta-Dictionary angemeldet werden, damit sie im AnnotationTool verfügbar ist.

An Hand eines realitätsnahen Anwendungsbeispiels wird nun der Einsatz konkreter Actions gezeigt. Hierfür wird das Szenario im Kapitel *Motivation* herangezogen. Dort wird ein Dateisymbol auf eine Drucker-Entität fallen gelassen. Dies bewirkt letztlich den Ausdruck des Dateiinhaltes. Es findet folgendes statt: Zunächst erhält das Dateisymbol (eine Medien-Entität) durch die manipulative Geste der Bewegungsverfolgung (vgl. Rahimi und Vogt [2008] S. 48) mehrere *PositioningEvents* und ändert daraufhin seine Position. Wird das Dateisymbol über der Druckerentität fallen gelassen, empfängt die Druckerentität ein *DropEvent* – u.a.

mit der Quelle (die zu druckende Datei) als Eventargument. Zuletzt wird das *PrintEvent* an die Drucker-Entität gesandt und der Ausdruck erfolgt.

Damit ist das Eventmodell vollständig umschrieben worden[9]. Dem Leser sollte nun klar sein, welche Mechanismen hinter Abb. 4.12 stecken, wie diese funktionieren und wie weitere Trigger, Events und Actions hinzugefügt werden können.

AnnotationTool

In diesem Kapitel wird die Bausteinsicht des AnnotationTools beschrieben.

Abbildung 4.18 zeigt eine Whitebox-Sicht auf das AnnotationTool – der Übersicht halber wurden nur die wichtigsten Pakete in die Grafik aufgenommen. Die Subsysteme ergeben sich aus den funktionalen Anforderungen, wie sie in der Analyse beschrieben wurden:

ToolBuildingModel (S. 83), *ContextMenu* (S. 86), *Dialogs* (S. 87), *Helpers* (S. 88).

Im Folgenden wird auf die einzelnen Subsysteme bzw. Pakete genauer eingegangen. Screenshots des AnnotationTools finden sich im Anhang unter Kap. 6.2 S. 109.

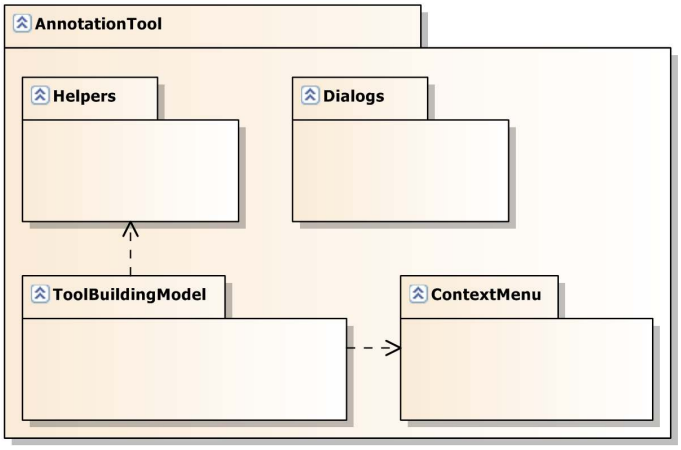

Abbildung 4.18: Whitebox-Sicht AnnotationTool

Paket: `ToolBuildingModel`

Das `ToolBuildingModel` Paket des AnnotationTools dient der graphischen Repräsentation der `BuildingModel` Entitäten aus der ToolkitLib. Wie die Entitäten in die Lage versetzt werden, Objekte der graphischen Oberfläche zu werden, wird im Folgenden erklärt.

[9] Der geschilderte Mechanismus mit Reflection als Basis wurde prototypisch implementiert.

Abb. 4.19 zeigt das Klassendiagramm des Paketes. Um eine Entität in der graphischen Oberfläche verwenden zu können, müsste diese von der abstrakten WPF Klasse `Shape` ableiten[10].

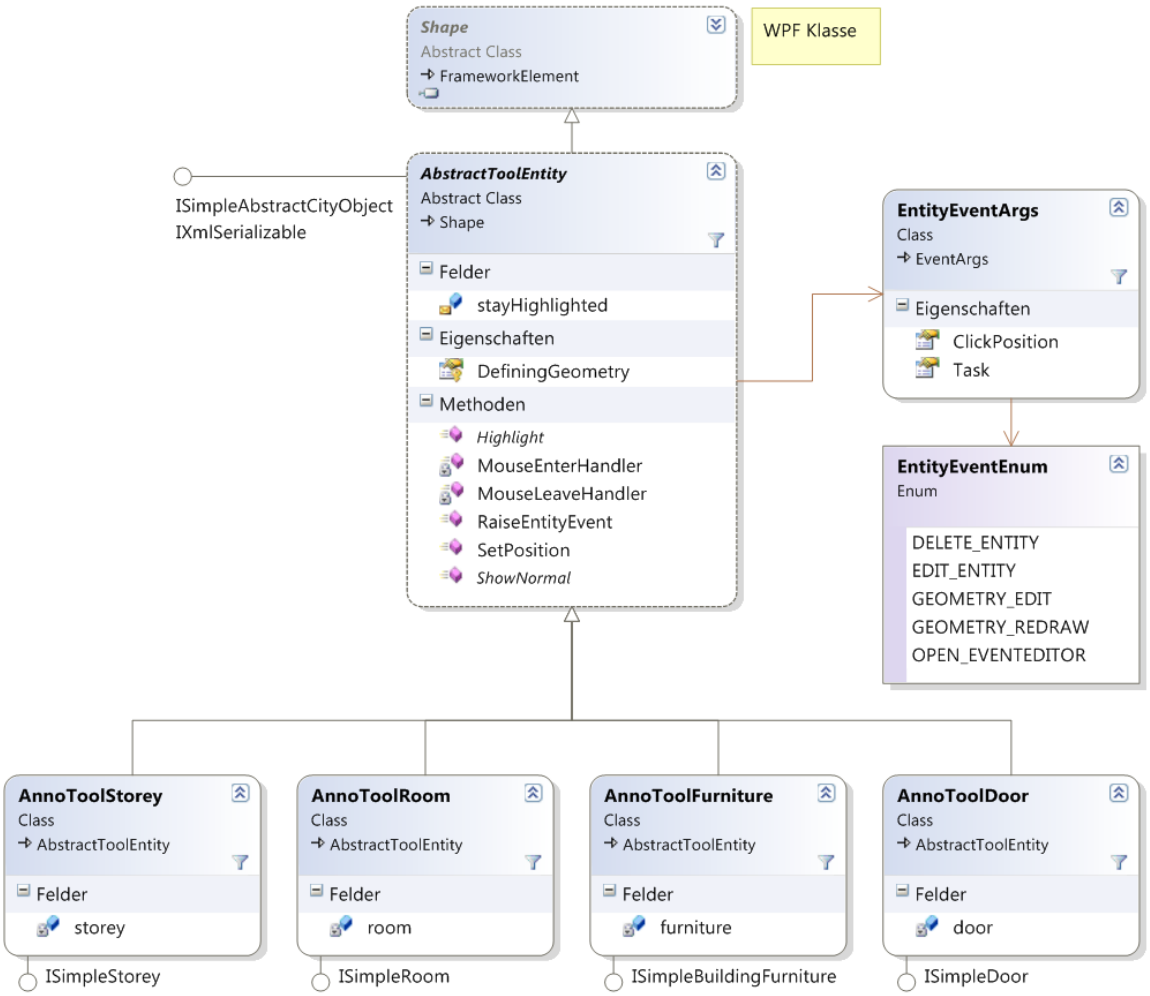

Abbildung 4.19: Whitebox-Sicht `ToolBuildingModel` Paket

Da die Entitäten des `BuildingModel` (z. B. `SimpleRoom`) bereits von einer gemeinsamen, abstrakten Basisklasse ableiten, ist eine zusätzliche Unterklassenbildung von `Shape` nicht mehr möglich. Eine Mehrfachvererbung ist u.a. aus Gründen der Übersichtlichkeit von Vererbungshierachien und Namenskonflikten bei den meisten modernen Programmiersprachen nicht möglich (Beneckenstein [2006]). Dennoch sollen die AnnotationTool Entitäten die

[10] Eine andere, performantere Variante wäre der Einsatz von sog. *Drawing Objects* in WPF – diese sind von der abstrakten Klasse `Freezable` abgeleitet, bieten aber keinerlei Interaktionsmöglichkeiten für den Benutzer (Stropek und Huber [2008])

gleichen Eigenschaften und das gleiche Verhalten haben wie die Gebäudeentitäten der ToolkitLib. Um das zu erreichen, bietet sich der Einsatz des *Decorator-Pattern* an.

Hierzu implementiert das visuelle Pendant der ToolkitLib im AnnotationTool dasselbe Interface – z. B. implementiert die Klasse `AnnoToolRoom` das Interface `ISimpleRoom`, ebenso wie der `SimpleRoom` in der ToolkitLib. Außerdem hält in diesem Beispiel der `AnnoToolRoom` eine Instanz vom Typ `ISimpleRoom`, so dass Methodenaufrufe des `ISimpleRoom` Interfaces an diese Instanz delegiert werden.

Somit können diese AnnotationTool-Entitäten in der graphischen Oberfläche verwendet werden und bieten zudem die Semantik der ToolkitLib-Entitäten. Das Verwenden des Decorator-Patterns bietet hierbei einige Annehmlichkeiten:

- Änderungen an den AnnotationTool-Entitäten durch Interaktion und Manipulation über die Benutzeroberfläche betreffen die Instanzen *direkt*
- Zustände und Verhalten werden dynamisch zugewiesen (z. B. wichtig bei der XML-Serialisierung)
- Die AnnotationTool-Entitäten können in Listen/Collections der ToolkitLib-Entitäten verwendet werden (z. B. `List<ISimpleRoom> GroupMember get; set;` des `ISimpleStorey`)

Die abstrakte Basisklasse `AbstractToolEntity`

In Abb. 4.19 ist die gemeinsame Oberklasse `AbstractToolEntity` aller AnnotationTool-Entitäten zu sehen. Oberklassen bieten die Möglichkeit, Gemeinsamkeiten von ableitenden Klassen zu bündeln. Tiefe Klassenhierarchien vergrößern jedoch die Kopplung, da Änderungen in Oberklassen sich auf alle Unterklassen auswirken. Lilienthal [2008] schreibt hierzu:

> *Die Abhängigkeit von Abstraktionen anstelle von konkreten Implementierungen ist der Schlüssel zu flexiblen und erweiterbaren Architekturen.*
> *Gernot Starke*
> *Starke [2008] S.162*

Die abstrakte Basisklasse gibt die zu überschreibenden Methoden `Highlight` und `ShowNormal` vor. Dies ist eine direkte Umsetzung aus den Anforderungsbeschreibungen der Analyse. Entitäten werden *gehighlightet*, wenn sich der Mauszeiger über ihnen befindet, das KontextMenü (siehe nächstes Kapitel) geöffnet wird oder ein externes Ereignis eintritt. Dieses Ereignis kann z. B. eine Tastenkombination oder Menu-Klick sein, die *alle* Räume eines Stockwerkes *highlighted*.

Paket: `ContextMenu`

Im Kontextmenu einer Entität spiegeln sich viele funktionale Anforderungen der Analyse wieder. Die gemeinsamen Anforderungen aller Entitäten finden sich in der abstrakten Klasse `AbstractContextMenu` (siehe Abb. 4.20). Hierzu zählt das Editieren und Löschen, das Öffnen des Eventeditors und das Ändern oder neu Zeichnen der Geometrie der betreffenden Entität. Die Unterklassen fügen zusätzliche Menüpunkte hinzu, die im Kontext der jeweiligen Entität sinnvoll sind.

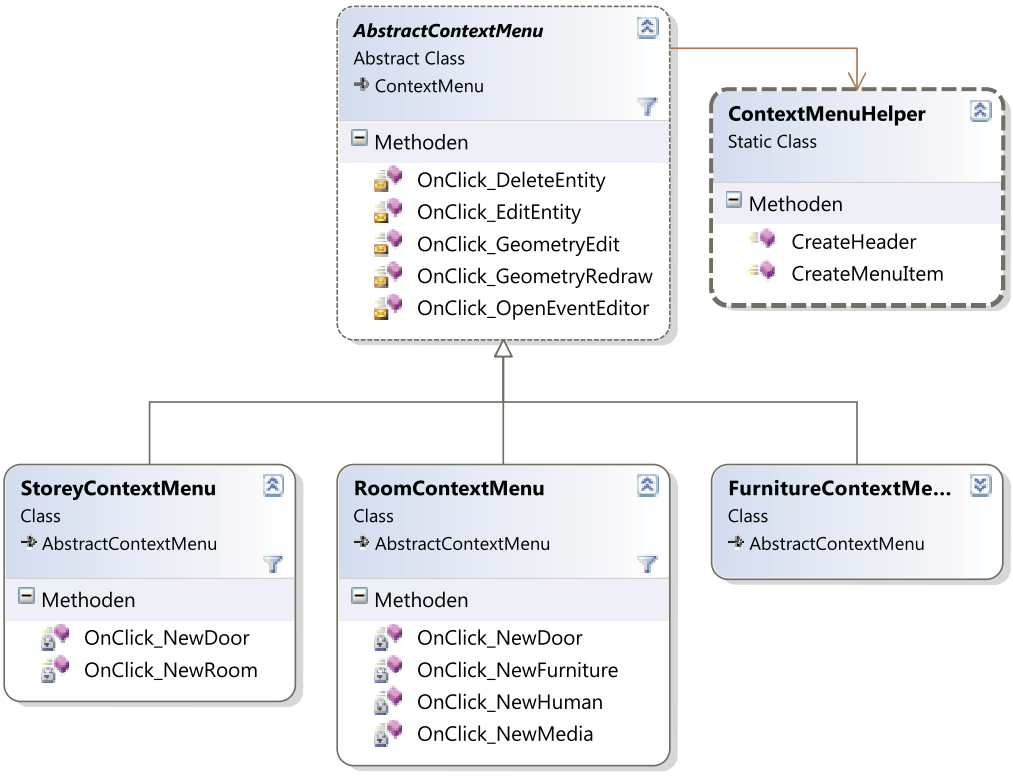

Abbildung 4.20: Whitebox-Sicht `ContexMenu` Paket

Das durch die Auswahl eines Menüpunktes ausgelöste *Event* wird an die entsprechende Entität weitergeleitet. Die Entität selber löst nun wiederum ein Event aus (`EntityEvent`; s. Abb. 4.19) – als Eventargumente (`EntityEventArgs`) werden die Entität (`source`), der Auftrag (`task` vom Typ `EntityEventEnum`) und die Position (`clickPosition`) der Maus beim Öffnen des Kontextmenüs überreicht. Das Senden dieser Events hat nichts mit dem Eventmodell zu tun, das im Kapitel *EventModel* (S. 69) beschrieben wurde. Es ist lediglich ein Weg zur entkoppelten Kommunikation zwischen Instanzen im AnnotationTool.

Der Empfänger des Events ist in diesem Fall der `ModelController`, der die dem Event zu Grunde liegende Anfrage bearbeitet. Näheres über den `ModelController` ist in Kap. 4.4.2 S. 90 zu finden. Der "Umweg" über ein Event hat den Vorteil, dass die Entität den Empfänger nicht kennen muss, was den Grad der Kopplung im System verringert.

Paket: `Dialogs` – Realisierung der graphischen Oberflächen

Wie bereits erwähnt, setzt sich eine graphische Oberfläche bzw. ein Fenster im verwendeten Framework aus einer Datei, die in einer deklarativen Sprache die Oberfläche beschreibt (XAML) und einer "Code-Behind" Datei zusammen. In Abb. 4.21 sind die Klassen des `Dialog`-Paketes aufgeführt, die ebendiesen Aufbau verfolgen.

Die Aufgaben des `EventEditor`s wurden bereits im Kapitel *Paket: `EventModel`* ausführlich beschrieben, weshalb hier nicht nochmals darauf eingegangen wird. Da den Klassen `Search` und `Progress` keine interessanten Designentscheidungen zu Grunde liegen, wird im Folgenden nur auf die Klasse `BaseProperties` eingegangen.

Die Klasse `BaseProperties`

Die Klasse `BaseProperties` bietet dem Benutzer die Möglichkeit über eine Eingabemaske die Eigenschaften einer Gebäudeentität festzulegen – hierzu zählen z. B. die Funktionen (`Function`), Beschreibung (`Description`) oder Geometrie (`Geometry2D`).

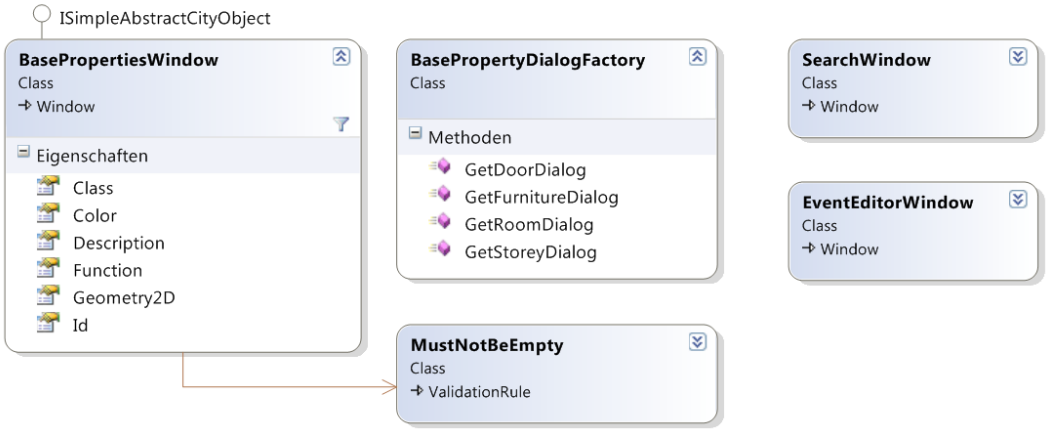

Abbildung 4.21: Whitebox-Sicht `Dialogs` Paket

Die Klasse `BaseProperties` beschreibt in ihrer XAML Ausführung alle in der Eingabemaske *möglichen* Felder (z. B. eine Textbox für die `Id`, eine `ComboBox` für die Farbauswahl etc.). Die gleichnamige "Code-Behind" Datei wählt zur Laufzeit die

für die jeweilige Entität zugehörigen Felder aus. Zur Vereinfachung wurde hierfür die Klasse `BasePropertyDialogFactory` implementiert, die eine Instanz des `BaseProperties`-Dialoges mit entsprechenden Einstellungen (Auswahl der Felder etc.) und vorbelegten Feldern bereitstellt. Nach Abschluss der Bearbeitung durch den Benutzer stellt die Klasse `BaseProperties` die eingegebenen Werte zur weiteren Verwendung zur Verfügung.

Das "Befüllen" einer Liste in der Benutzeroberfläche wird durch sog. *DataBinding* an ein Dictionary erreicht. *DataBinding* ist ein Feature des WPF Frameworks, das es ermöglicht den Inhalt von geeigneten Objekten an Listen bzw. Dictionarys zu koppeln. Deutlich wird dies am Beispiel einer Raumentität: Zunächst wird ein Dictionary für den "CodeIdentifier" `RoomFunctionType` erstellt, welcher die Klasse `CodesDictionary` aus einer XML Datei generiert. Dieses Dictionary wird dann an eine `ComboBox` der Benutzeroberfläche gebunden. Dort (in XAML) wird beschrieben wie `Key` und `Value` als *ein* Eintrag in der ComboBox angeordnet sind (z. B. `1320 : conference room`).

Die *Verifikation* der eingegebenen Daten erfolgt durch zwei Mechanismen. Zum einen werden Änderungen eines Feldes direkt an die Entität weitergeleitet – dies löst ggf. eine Exception aus, die als Fehlermeldung in der Benutzeroberfläche angezeigt wird. Beispielsweise wird im Feld `Id` eines `SimpleRoom` der Wert durch einen regulären Ausdruck untersucht – entspricht der eingegebene Raumname nicht dem regulären Ausdruck wird eine Exception ausgelöst, die als Argument einen Hinweistext enthält (z.B.: `throw new ArgumentException("Raumbezeichnung ist nicht korrekt! Korrektes Bsp.: R_11.01a")`). Dies hat den Vorteil, dass die Entität selbst über die Korrektheit von Werten entscheidet und die grundlegende Validierung nicht in externem Code vorgenommen wird, was die Verständlichkeit des Codes erhöht. Außerdem erfolgt die Rückmeldung an den Benutzer instantan bei jedem eingegebenem Zeichen. Aus Sicht der *Usability* sind solch frühe Rückmeldungen an den Benutzer von Vorteil.

Paket: `Helpers`

Viele der Klassen dieses Paketes können im MVC-Pattern zum Controller gezählt werden, da sie direkt oder indirekt Einfluss auf das Model nehmen. In Abb. 4.22 sind die wesentlichsten Klassen des Paketes zu sehen. Die Verwendung einiger Klassen lässt sich aus dem Namen ableiten. Die Klassen, bei denen einige interessante Realisierungsdetails oder Designentscheidungen getroffen wurden, werden im Folgenden genauer erläutert.

Importer und Exporter

Wie in der Bausteinsicht der ToolkitLib bereits erwähnt wurde, können sich hinter einem Exporter bzw. Importer Interface (z. B. `IStoreyImporter`) mehrere unterschiedliche Rea-

Abbildung 4.22: Whitebox-Sicht `Helpers` Paket

lisierungen bzw. Algorithmen verbergen. Die Auswahl einer konkreten Realisierung erfolgt dann durch den Benutzer des AnnotationTools (z. B. im Datei-Öffnen-Dialog). Beim darauf folgenden Ex- bzw. Importvorgang wird nur auf die Methoden des Interfaces zugegriffen, so dass die konkrete Realisierung in den Hintergrund tritt.

Für diesen Einsatzzweck ist das *Strategie-Pattern* sehr gut geeignet. Der Quelltext wird übersichtlicher, da mehrfache Bedingungsanweisungen zur Auswahl eines Algorithmus vermindern werden. Außerdem macht die Kapselung der einzelnen Algorithmen sie austauschbar und ermöglicht es einen Algorithmus unabhängig von ihn nutzenden Klassen zu variieren (s. S. 373 Gamma u. a. [1996]).

Events zur Kommunikation zwischen Klassen

Unabhängig vom Eventmodell (s. Kap. 4.4.2 S. 69) werden im AnnotationTool Events zur Kommunikation zwischen den Akteuren eingesetzt. Der Vorteil liegt oftmals in einer geringeren Kopplung, da die Klasse, die ein Event auslöst den Empfänger nicht kennen muss. Demzufolge beschränkt sie sich auf ihre klar definierte Rolle innerhalb des Toolkits, was zu einer stärkeren Kohäsion führt.

Als Beispiel für den Einsatz von Events wird der *HighlightingTrigger* herangezogen. Als eine Anforderung wurde u.a. das gleichzeitige “Highlighten” aller Entitäten eines Typs gefordert, um z. B. Fehler bei Raumgeometrien der Raumentitäten für eine digitale Nachbearbeitung ersichtlich zu machen.

Die Anforderung wurde mit dem *Observer-Pattern* umgesetzt. Bei diesem Pattern werden Beobachter, die sich bei einem Subjekt angemeldet haben, bei Änderungen am Subjekt darüber informiert. In diesem Beispiel ist die eintretende Änderung entweder eine Tastenkombination oder die Auswahl eines Menüpunktes (z. B. “Alle Räume hervorheben”). Der *HighlightingTrigger* benachrichtigt daraufhin alle angemeldeten Entitäten über ein Event. Im Eventhandler der Entitäten wird schließlich das Highlighten eingeleitet bzw. rückgängig gemacht – der Zustand wird über eine boolesche Variable im `HighlightingEventArgs` Eventparameter übergeben.

Der `ModelController`

Der `ModelController` verwaltet wie der Name vermuten lässt das *Model* – in diesem Fall also die Gebäudeentitäten des `ToolBuildingModel` Paketes. Das Einfügen, Entfernen und der Zugriff auf die Entitäten sind die Hauptaufgaben des Controllers.

Der Prozess des Einfügens bzw. vielmehr der Generierung neuer Entitäten ist mitunter recht umfangreich. Als Beispiel soll das Sequenzdiagramm in Abb. 4.23 dienen. Hier erzeugt der Administrator des AnnotationTools mit Hilfe des Kontextmenüs eines Raumes eine neue Tür. Das Sequenzdiagramm stellt somit einen Abschnitt der Laufzeitsicht des Programms dar.

Vor der eigentlichen Erzeugung der Tür wird deren Geometrie vom Benutzer eingezeichnet. Die von der Geometrie überdeckten Entitäten werden ermittelt und wenn *genau* zwei Räume überdeckt werden, wird die Tür erzeugt. Da eine Tür zwei Räume verbindet, wird diesen Räumen nun die erzeugte Tür zugewiesen. Zum Schluss wird die Tür mit Hilfe des `DrawUIElementHelper` in die graphische Oberfläche eingefügt.

Weitere Details der Realisierung finden sich in der Abbildung 4.23. Der Übersicht halber wurden einige Akteure weggelassen und Methodensignaturen gekürzt oder gänzlich neu hinzugefügt. Das Prinzip der Erzeugung einer Entität gilt ähnlich auch für Räume und Mobiliar.

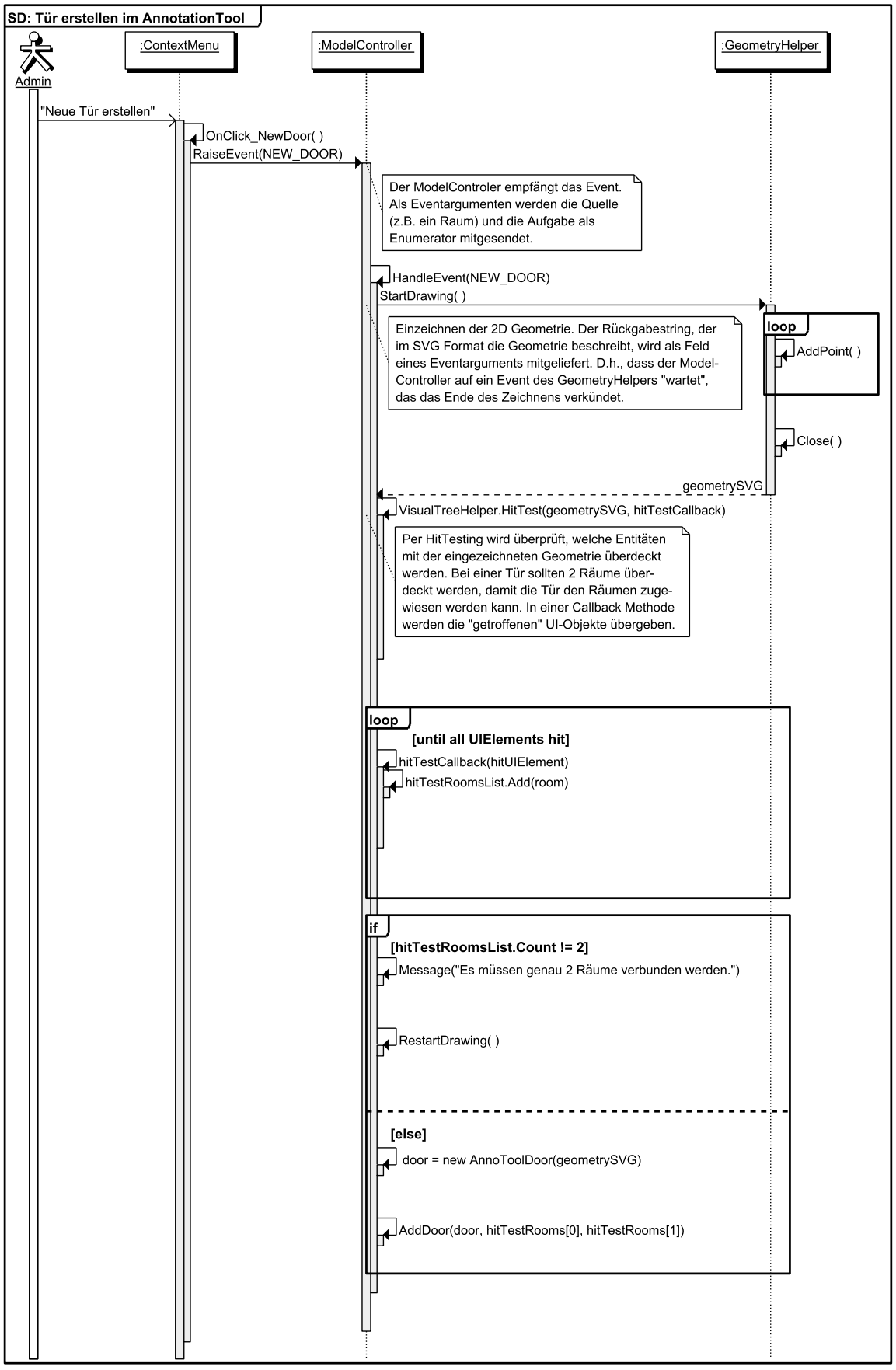

Abbildung 4.23: Sequenzdiagramm: Tür erstellen im AnnotationTool

WelcomeScreen

Der WelcomeScreen ist ein visueller Prototyp, der aufzeigt, wie die im AnnotationTool erstellten Daten in einem raumbezogenen Informationssystem Verwendung finden können. Da der Schwerpunkt dieser Arbeit in der Entwicklung des AnnotationTools und den Konzepten dahinter liegt, wird der WelcomeScreen vergleichsweise kurz beschrieben.

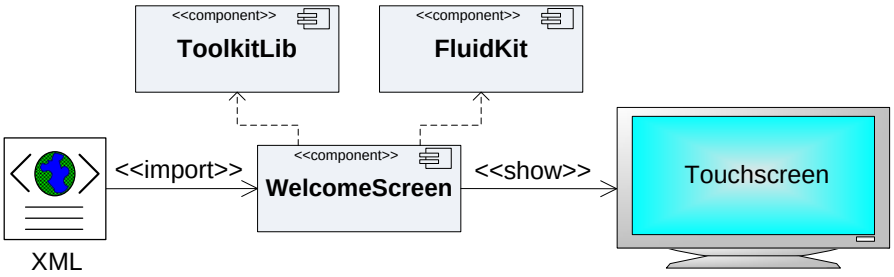

Abbildung 4.24: Komponenten des WelcomeScreens

Abb. 4.4.2 zeigt schematisch, welche Komponenten im Prototypen verwendet werden. Eine dynamische Sicht in Form eines Sequenzdiagrammes ist in Abb. 4.4.2 zu sehen, wo der Ablauf beim Systemstart (stark abstrahiert) dargestellt wird.

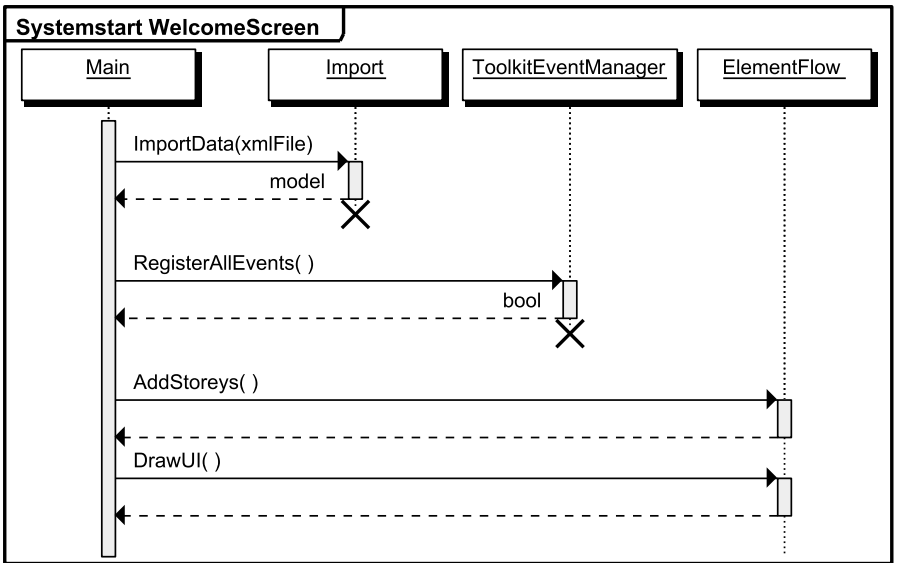

Abbildung 4.25: Systemstart des WelcomeScreens

Zunächst liegt eine vom AnnotationTool erstellte Projektdatei im XML Format vor, die die Stockwerke und darin befindliche Entitäten, die EventActions und ggf. einige Konfigurationseinstellungen enthält. Mit Hilfe des Import-Moduls der ToolitLib werden die Daten zunächst

extrahiert. Nach der Deserialisierung werden die Gebäudeentitäten mit Hilfe des ToolkitEventManagers auf die zugewiesenen EventActions registriert (wie in Kap. 4.4.2 S. 69 beschrieben). Daraufhin werden die Stockwerke einer Instanz der Klasse `ElementFlow` überreicht. Sie stammt aus einer externen Bibliothek (Podila [2009]) und stellt sozusagen die *View* dar, da sie die Stockwerke in Form von sog. `UserControls` entgegen nimmt und visualisiert. Der damit erzielte "Flow-Effekt", mit dem die 2D-Stockwerke durchlaufen werden können, ähnelt dem von der Firma Apple bekanntem "Coverflow". Denkbar wären jedoch auch andere Arten der Visualisierung. Im Anhang unter 6.2 befindet sich ein Screenshot des WelcomeScreens.

4.5 Entwicklungsumgebung, Laufzeitumgebung und Grafikframework

Für die Realisierung des Toolkits wird die *Windows Presentation Foundation* (WPF) von Microsoft herangezogen. Sie ist Teil des .NET Frameworks, das in der Version 3.5 in dieser Arbeit verwendet wird.

Wenngleich andere Frameworks[11] ähnliche Möglichkeiten bieten, ist WPF dennoch in besonderer Weise geeignet die beschriebenen Ziele umzusetzen. Das Framework wurde auf der Grundlage von *Direct3D* aufgebaut, wodurch die Grafik-Hardware zur performanten Visualisierung der graphischen Oberfläche beiträgt (Stropek und Huber [2008]). Mit der deklarativen Oberflächenbeschreibungssprache XAML kann eine klare Trennung von Design und Logik erreicht werden.

Zur Implementierung des Toolkits wurden Visual Studio 2008 Professional Edition und Visual Studio Team System 2010 Team Suite Beta 1 verwendet.

[11] z. B. Adobe Flex, Java 3D usw.

5 Schluss

Erklärtes Ziel des erarbeiteten *Toolkits* ist es, als Grundlage interaktiver, raumbezogener Applikationen dienen zu können. Inwiefern der Autor dieses Ziel erreicht sieht, wird im folgenden Kapitel *Fazit* erörtert. Im Kapitel *Ausblick* (S. 96) werden sinnvolle Erweiterungen des Toolkits aufgezeigt.

5.1 Fazit

Im Folgenden wird auf die Ergebnisse dieser Arbeit eingegangen. Es wird aufgezeigt, welche Ziele erreicht wurden und welche Themen ausbaufähig sind.

Perfektion ist nicht dann erreicht, wenn es nichts mehr hinzuzufügen gibt, sondern wenn man nichts mehr weglassen kann.
Antoine de Saint-Exupéry

Das obige Zitat von Antoine de Saint-Exupéry kann auf alle Lebensbereiche – diese Arbeit mit eingeschlossen – angewandt werden. In der Informatik wird diese Aussage oftmals auf den Kerngedanken “Keep It Simple” konzentriert und fordert ein ebenso einfaches, wie überschaubares Anwendungsdesign – beginnend bei der Architektur bis hinunter zum Quelltext.

Dies wird insbesondere bei Software, die von mehr als einer Person (weiter)entwickelt werden soll, gefordert und spiegelt sich größtenteils in der Umsetzung nicht-funktionaler Anforderungen wieder. Hierzu zählen in erste Linie die Analysierbarkeit, Modifizierbarkeit und generell die Verständlichkeit einer Anwendung.
Es wurden vielerlei Techniken eingesetzt, um die Anforderungen umzusetzen. Beim Anwendungsdesign wurde stets auf Prinzipen wie lose Kopplung, hohe Kohäsion, Zyklenfreiheit und der Trennung von Design und Logik geachtet.

Schnittstellen gestatten es, die Abhängigkeiten zwischen Architekturelementen in der Schnittstelle zu konzentrieren und jede Abhängigkeit von der Implementierung zu vermeiden.
Dr. Carola Lilienthal
Lilienthal [2008] S.81

Außerdem wurden Interfaces genutzt, um die Kopplung zu verringern und eine hohe Anpassbarkeit zu gewährleisten. Einige Entwurfsmuster, u.a. Decorator, Strategy und Observer tragen zur Wiederverwendbarkeit und Entkopplung von Modulen bei, sowie in hohem Maße auch zur Verständlichkeit der Anwendung. Jedoch bleibt im Hinblick auf *WPF Neulinge*, die sich schnell in das Toolkit einarbeiten sollen (siehe S. 45), die Skepsis, ob die Verständlichkeit der Anwendungen damit in ausreichendem Maße gegeben ist.

Die Forderung nach hoher Portierbarkeit ist auf Grund der eingeschränkten bzw. nicht vorhandenen Verfügbarkeit der .NET Laufzeitumgebung für Nicht-Windows Betriebssysteme nicht einzuhalten.

Zu den aus Zeitgründen nicht erfüllten funktionalen Anforderungen des AnnotationTools zählt eine dreidimensionale Darstellung des Gebäudemodells. Außerdem wurde die *Suche* und das *Editieren von Geometrien* nicht umgesetzt. Da jedoch kein neues CAD Tool für die Stadt- oder Gebäudeplanung entwickelt werden sollte, sind diese Teilaspekte keinesfalls schwerwiegend. Der WelcomeScreen (visueller Prototyp) wurde aus den gleichen Gründen nur sehr rudimentär umgesetzt, so dass die Raum- und Arbeitsplatzsuche fehlen.

Ein wesentliches Ziel war die Extraktion geometrischer und semantischer Informationen aus CAD Daten in ein semantisches Gebäudemodell. Zwar ist die Extraktion semantischer Informationen aus den Daten ausbaufähig (Gründe dafür wurden in Kap. 4.4.2 S. 64 hinreichend beschrieben). Hingegen konnten die geometrischen Daten nahezu fehlerfrei extrahiert und in eine standardisierte Vektorbeschreibung umgesetzt werden. Durch manuelle Nachdigitalisierung wird das *Erkennen* und *Beheben* von Fehlern des Extraktionsprozesses ermöglicht. Dem Entwickler raumbezogener Anwendungen steht damit eine Klassenbibliothek zur Verfügung, das 2D-CAD Daten in ein semantisches Gebäudemodell importiert und außerdem die Möglichkeit bietet, Fehler des Importprozesses zu korrigieren. Zudem stehen Module zur Serialisierung und Deserialisierung der Gebäudedaten bereit.

Das erarbeitete Innenraummodell baut auf den Erfahrungen von CityGML auf. Durch die Trennung von Präsentation und Semantik ist es gelungen, zukünftigen GIS eine Plattform zu bieten, die die Konzepte aktueller Gebäudemodelle umsetzt. Die semantischen Modelleigenschaften ermöglichen thematische Anfragen, die über die Möglichkeiten rein geometrischer Modelle hinausgehen.

Mit einem sehr flexiblen Eventmodell ist es gelungen Entitäten toolgestützt mit Verhalten und kontextsensitiven Eigenschaften zu versehen. Heutige Stadtmodelle – meist rein statische Repräsentationen einer dynamischen Realität – werden dadurch um dynamische Inhalte und Prozesse erweitert. Dadurch kann der Benutzer des WelcomeScreens mit Gebäudeentitäten interagieren, oder Personen des Modells können durch Positionierungssysteme dynamisch

ihren Aufenthaltsort ändern. Für das Eventmodell sind viele weitere Einsatzmöglichkeiten denkbar – einige werden im nächsten Kapitel erörtert.

Das Toolkit bildet also mit den drei Komponenten *ToolkitLib*, *AnnotationTool* und *WelcomeScreen* eine umfangreiche Basis für interaktive, raumbezogene Informationssysteme. Die unterschiedlichen Anforderungen von Administratoren, die ein graphisches Tool zur Pflege der Gebäudedaten benötigen, sowie von Anwendungsentwicklern, die ein einfach zu erweiterndes System für die Entwicklung individueller, raumbezogener Anwendungen fordern, wurden beim Design des Toolkits berücksichtigt.

In einer Machbarkeitsstudie wurde gezeigt, wie eine komplette, interaktive Anwendung mit Raumbezug auf Basis des Toolkits entwickelt werden kann. Dazu zählt die bereits erwähnte Extraktion von Informationen aus CAD Daten in ein semantisches, an CityGML angelehntes Gebäudemodell, sowie die Möglichkeit diese Daten nachzubearbeiten und mit weiteren Informationen und sogar mit Verhalten zu annotieren und auf Basis dessen, eine interaktive, raumbezogene Anwendung zu entwickeln.

Das Ziel, ein Toolkit zu entwickeln, das zukünftigen Entwicklerteams die Grundlage für interaktive, raumbezogene Anwendungen liefert, wurde mit dieser Arbeit erreicht.

5.2 Ausblick

In diesem Kapitel werden zukünftige Erweiterungen und Einsatzmöglichkeiten des Toolkits erörtert.

5.2.1 Zusätzliche Exportformate

Die mit dem AnnotationTool gesammelten Daten können nicht nur auf Informationssystemen wie dem WelcomeScreen zum Einsatz kommen, sondern auch z. B. für Fluchtpläne, in Printmedien oder einer Internetpräsenz genutzt werden. Hierfür ist der Export in diverse Dateiformate – z. B. PDF, SVG, JPG etc. – sinnvoll. Die Erhaltung des vektoriellen Charakters der Ausgabeformate ermöglicht einen hochwertigen Druck, der auch für Poster geeignet ist. Hierfür sind Erweiterungen des AnnotationTools sinnvoll, wie bspw. ein Masseneditiermodus für Gebäudeentitäten und eine Druckvorschau mit weiteren Anpassungsmöglichkeiten.

5.2.2 Routengenerierung

Werden mit dem WelcomeScreen z. B. bestimmte Gegenstände, Räume oder Personen gesucht, ist es sinnvoll dem Benutzer den Weg von A nach B mit Hilfe einer Route aufzuzeigen (und nicht nur die gesuchte Entität im Stockwerk zu *highlighten*).

In Koychev [2008] beschreibt Milen Koychev u.a. verschiedene Möglichkeiten zur Generierung von Routen innerhalb von Gebäuden, die für die Raumsuche verwendet werden können.

Mit Hilfe dieser Routen, die in Form eines kantengewichteten Graphen im zwei- oder dreidimensionalen Raum vorliegen, können Fußgänger von einem Startpunk zu einem Zielpunkt geführt werden. Start- und Zielpunkte werden durch die Knoten des Graphen repräsentiert und liegen stets innerhalb eines Raumes. Die Kanten des Graphen repräsentieren die Strecken, die ein Fußgänger im Gebäude zurücklegen *kann* – ohne z. B. durch Wände zu führen.

Es gibt zwei grundlegende Ansätze einen solchen Graphen zu generieren: manuell und automatisch. Außerdem gibt es Mischformen mit halb-automatischer Generierung. Insbesondere bei einer sich oft ändernden Nutzung einer umfangreichen Gebäudestruktur ist die manuelle Pflege kaum zu leisten. Deshalb wurden Anstrengungen unternommen den Prozess zu automatisieren. In Drexl [2003] wird ein Verfahren vorgestellt, dass diese Graphen automatisch aus CAD Daten generiert. Jedoch sind die generierten Graphen meist sehr groß und unregelmäßig Narasimhan [2007]. Eine Verbesserung der Methode wird in Narasimhan [2007] beschrieben.

Der kürzeste Weg zwischen zwei Knoten eines Graphen lässt sich z. B. mit dem Dijkstra-Algorithmus herausfinden. Der zu Grunde liegende Graph kann – wie erwähnt – automatisch generiert werden, jedoch ist das Resultat selten zufriedenstellend.

Im Vorfeld arbeitete der Autor an einer separaten Lösung, die das manuelle Einzeichnen eines Graphen auf einem Stockwerksplan unterstützt (siehe Abb. 5.1). Die Anwendung diente u.a. dazu, sich mit dem Framework vertraut zu machen und an Algorithmen zur Extraktion von Informationen aus CAD Daten zu arbeiten.

Der Graph und weitere Implementierungsdetails sollten in das AnnotationTool integriert werden. Eine Umsetzung des in Drexl [2003] vorgestellten Verfahrens wäre nützlich. Es erscheint sinnvoll, dass Türen des Gebäudemodells Knoten des Graphen werden können. Über die Türen ist es dann möglich, die angrenzenden Räume zu ermitteln. Eckpunkte des Graphen, die keiner Entität zugeordnet werden können und lediglich für die Form des Graphen sorgen, könnten als “unsichtbares Mobiliar” im Gebäudemodell untergebracht werden.

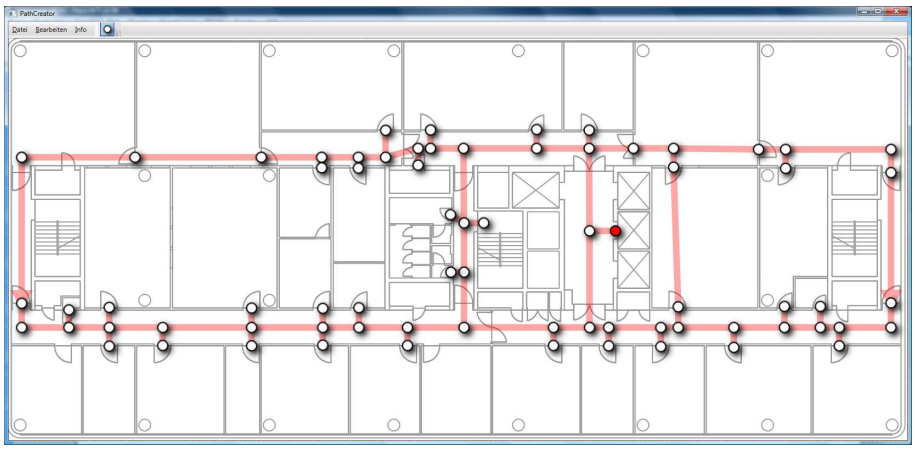

Abbildung 5.1: Manuelles Zeichnen eines Graphen

5.2.3 Personalisierte Informationen

Das Informationssystem sollte mit personalisierten Informationen angereichert werden können. Hierzu sind Schnittstellen zu externer Hardware, z. B. zu mobilen Endgeräten (per Bluetooth o.ä.) oder RFID Lesegeräten zu integrieren. Da Professoren, Studenten und Mitarbeiter moderner Hochschulen über RFID Ausweise verfügen, können Inhalte mit benutzerspezifischen Informationen aufbereitet werden.

Es gibt bereits erste Ideen und Umsetzungen in diesem Bereich, die sich in den WelcomeScreen integrieren lassen. Beispielsweise werden individuelle Stundenpläne angezeigt, nachdem der RFID-Ausweis vor das Lesegerät gehalten wird. Mit Hilfe der beschriebenen Navigationspfade können die Wege von Vorlesungssaal zu Vorlesungssaal angezeigt werden. In Bettzüche u. a. [2009] wurde ein System entwickelt, das u.a. persönliche Routen zu verschiedenen Points Of Interests erstellt und visualisiert.

5.2.4 CityGML als Zielformat

Diverse Designentscheidungen des Gebäudemodells wurden mit Hinblick auf eine zukünftige Verwendung in CityGML getroffen. Dazu zählt insbesondere der Gebrauch von Entitäten, die nicht in CityGML 1.0.0 enthalten sind.

Insbesondere war die Modellierung von Personen erforderlich. Sie werden zwar nicht in CityGML modelliert, aber durch sog. Fachschalen (*Application Domain Extension [ADE]*[1]) können vorhandene Entitäten erweitert werden[2]. Von CityGML Erweiterungen wird gefordert,

[1] Ein Tutorial zum Hinzufügen eigener ADEs in CityGML findet sich in Nagel [2009].
[2] Mit entsprechenden XML Schema Definitionen wird der dadurch erweiterte CityGML Standard nicht verletzt.

dass neue Entitäten von vorhandenen abgeleitet werden. Dieser Umstand wurde im Design berücksichtigt, so dass eine Konvertierung in den CityGML Standard strukturell realisierbar ist.

Die Geometrie von Entitäten – insbesondere von Räumen – wird bei 2D CAD Ausgangsdaten ebenfalls zweidimensional verwaltet. Für die Verwendung in 3D Stadtmodellen müssen diese Daten umgewandelt bzw. ergänzt werden. Bei planparallelen Stockwerken kann ein 3D Modell automatisch abgeleitet werden – dies gestaltet sich jedoch meistens wesentlich problematischer.

Die in dieser Arbeit beispielhaft verwendeten Stockwerksdaten der Hochschule für Angewandte Wissenschaften Hamburg stammen von einem Gebäude, dessen Stockwerke planparallel sind, so dass die dritte Dimension stets von gleichem Ausmaß ist und somit einfach abgeleitet werden kann. Es ist bekannt, dass CAD Daten auch von vielen weiteren Gebäuden in ähnlicher Form vorliegen, so dass an einer Möglichkeit gearbeitet werden sollte, aus den vorhanden Daten ein 3D CityGML Modell (ggf. mit manueller Unterstützung) zu generieren.

5.2.5 Einsatzmöglichkeiten des Eventmodells

Das in dieser Arbeit entwickelte Eventmodell bietet vielfältige Einsatzmöglichkeiten – einige werden im Folgenden vorgestellt.

Die Interaktion sehbehinderter Menschen mit Touchscreens steckt noch in den Kinderschuhen. Dezentrale Gesten[3] sind nur für geübte Anwender von Nutzen und lediglich ein kleiner Schritt in Richtung Barrierefreiheit. Die akustische Steuerung einer Applikation ist dagegen weiter fortgeschritten. Im Eventmodell würde ein Trigger (ggf. als Hintergrundprozess) auf Sprachkommandos warten. Im Erfolgsfall wird ein entsprechendes Event ausgelöst und beim Empfänger bearbeitet.

Ein weiterer Anwendungsfall sind dynamische Prozesse. Jeder Zustand, der sich regelmäßig, zeitlich ändert, wird hier als *dynamisch* verstanden.

Der Aufenthaltsort von Personen ändert sich ständig. Zwar haben einige Personen im betreffenden Gebäude ein Büro, jedoch verteilt sich die Aufenthaltswahrscheinlichkeit ungleichmäßig auf mehrere Räume. Diesem Umstand kann mit Systemen zur Positionsbestimmung begegnet werden. Zum Beispiel kann mit *MagicMap* (“ein System zur kooperativen Positionsbestimmung über WLAN & RFID” [4]) der Aufenthaltsort einer Person bestimmt werden. Die Abweichungen sind teilweise gravierend. Zu genaueren Ergebnissen können neuronale Netze führen (vgl. Oblonczek und Zahn [2009]). In Jensen u. a. [2009] werden die Probleme der

[3] vgl. http://www.blindnews.eu/hilfsmittel/10399.html
[4] MagicMap: http://wiki.informatik.hu-berlin.de/nomads/index.php/MagicMap

Positionsbestimmung genauer betrachtet und Positioning- Engines evaluiert. Die im Gebäudemodell dieser Arbeit enthaltenen Informationen über Räume, Türen und Wände können ebenfalls zur heuristischen Eingrenzung der Ungenauigkeiten von Positionsdaten verwendet werden.

Des Weiteren sind Annotationen angedacht, deren grundlegende Natur insofern dynamisch ist, als dass sie nur für eine gewisse Zeitspanne existieren. Sie können ggf. mit einem oder mehreren Adressaten versehen und öffentlich sichtbar, oder nur authorisierten Personen zugänglich sein. Ein Beispiel hierfür ist das “elektronisches PostIt” – das z. B. als Videostream einer Medienentität des Modells an eine Person “gepinnt” werden kann.

Ein weiteres Einsatzgebiet ist die Umsetzung multimodaler Interaktionen mit Hilfe des Eventmodells. In Rahimi und Vogt [2008] wird (wie bereits in Kap. 4.4.2 S. 73 beschrieben) die “Gestenbasierte Computerinteraktion auf Basis von Multitouch-Technologie” untersucht. Gesten wie “Verschieben und Rotieren” lassen sich mit dem Eventmodell abbilden und tragen zu einer intuitiveren Mensch-Maschine-Interaktion bei.

Literaturverzeichnis

[Bartelme 2005] BARTELME, Norbert: *Geoinformatik: Modelle, Strukturen, Funktionen*. Ausgabe: 4. Springer, 2005. – ISBN 3-540-20254-4

[Beneckenstein 2006] BENECKENSTEIN, Daniel: Mehrfache Vererbung / Deutsches Elektronen-Synchrotron DESY. URL http://www.desy.de/~danielb/talks/, 2006. – Vortrag

[Bettzüche u. a. 2009] BETTZÜCHE, Björn ; JOCHHEIM, Banjamin ; RUHNKE, Jan ; SCHNELL, David: *Knavi*. 2009. – URL http://code.google.com/p/knavi/. – Bachelorprojekt an der Hochschule für Angewandte Wissenschaften Hamburg

[Bieber u. a. 2002] BIEBER, Gerald ; BLIESZE, Marcus ; KIRSTE, Thomas ; OPPERMANN, Reinhard: Aufgabenorientierte und situationsgesteuerte Computerunterstützung für mobile Anwendungen in Indoor-Umgebungen. In: *Mensch & Computer*, 2002

[Bosch 2004] BOSCH, Andy: *Java Server Faces – Das Standard -Framework zum Aufbau webbasierter Anwendungen*. Addison-Wesley Verlag, 2004

[Buchholz 2007] BUCHHOLZ, Clemens: *Entwicklung einer standortabhängigen, GPS gestützten Mitfahrbörse auf Smartphones für eine mobile Peer-to-Peer-Community*, HAW Hamburg, Bachelorthesis, 2007

[Butz und Krüger 2001] BUTZ, A. ; KRÜGER, A.: Orts- und richtungsabhängige Informationspräsentation auf mobilen Geräten. In: *IT+TI* 2 (2001), S. 90–96

[Consortium 2003] CONSORTIUM, World Wide W.: *Paths - SVG 1.1 - 20030114*. 2003. – URL http://www.w3.org/TR/SVG11/paths.html. – Letzter Aufruf am 29.06.2009

[Drexl 2003] DREXL, Thomas: *Entwicklung intelligenter Pfadsuchsysteme für Architekturmodelle am Beispiel eines Kiosksystems (Info-Point) für die FMI in Garching*, Technische Universität München, Diplomarbeit, 2003

[Ester 2008] ESTER, Alexandra: *Konzept für die Nutzung von IFC für die Aufgaben des Facility Managements*, Bauhaus-Universität Weimar, Masterarbeit, 2008. – URL http://www.buildingsmart.de/pdf/bachelor_01.pdf

[Fesl 2006] FESL, Robert: Microsoft Location Framework – Ein Framework zur dynamischen Karten- und Positionsdarstellung / European Microsoft Innovation Centre (EMIC) Aachen. URL http://www.geomv.de/geoforum/2006/beitraege/3_03_LocationFramework.pdf, April 2006. – Präsentation

[Gamma u. a. 1996] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster - Elemente wiederverwendbarer Software*. 4., korrigierter Nachdruck. Addison-Wesley Publishing Company, 1996. – ISBN 3893199500

[Gartner u. a. 2004] GARTNER, G. ; FRANK, A. ; RETSCHER, G.: Pedestrian Navigation System in Mixed Indoor-Outdoor Environments: The NAVIO Project. In: *Proceedings of the CORP 2004 and Geomultimedia04 Symposium*. Vienna, Austria, 2004, S. 165–171

[GmbH 1986] GMBH, Brockhaus: *Brockhaus. Die Enzyklopädie in 24 Bänden.* 19. völlig neubearb. Aufl. F.A. Brockhaus GmbH, Mannheim, 1986. – ISBN 3765311006

[Gossman 2005] GOSSMAN, John: *Introduction to Model/View/ViewModel pattern for building WPF apps*. Blog. 2005. – Letzter Aufruf am 21.08.2009

[Gröger u. a. 2008] GRÖGER, Gerhard ; KOLBE, Thomas H. ; CZERWINSKI, Angela ; NAGEL, Claus: OpenGIS®City Geography Markup Language (CityGML) Encoding Standard / Open Geospatial Consortium Inc. URL http://portal.opengeospatial.org/files/?artifact_id=28802, 2008. – International OGC Standard Version 1.0.0

[Herglotz 2006] HERGLOTZ, Andreas: *Lokalisierung und Orientierung in Gebäuden - IMAPS und Headmounted Display im Einsatz als Museumsführer den Einsatz in mobilen Anwendungen*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2006. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/herglotz.pdf

[Huber und Pletz 2007] HUBER, Thomas ; PLETZ, Christoph: Das Model View View Model-Pattern für WPF-Anwendungen. In: *dot.net magazin* 10 (2007), S. 2–11

[IAI 2008] IAI: *Anwenderhandbuch Datenaustausch BIM/IFC*. IAI - Industrieallianz für Interoperabilität e.V. (Veranst.), 9 2008. – URL http://www.buildingsmart.de/handbuch/files/buildingSMART_Anwenderhandbuch_Version2.0.pdf

[IAI 2009] IAI: *General understanding of the IFC Specification*. IAI - Industrieallianz für Interoperabilität e.V. (Veranst.), September 2009. – URL http://www.iai-tech.org/services/faq/fag-general-ifc-spec

[ISO 1991] ISO: ISO/IEC 9126 / International Organization for Standardization. URL http://www.cse.dcu.ie/essiscope/sm2/9126ref.html, 1991. – Norm

[Jensen u. a. 2009] JENSEN, B. ; KRUSE, R. ; WENDHOLT, B.: Application of indoor navigation technologies under practical conditions. In: *Positioning, Navigation and Communication, 2009. WPNC 2009. 6th Workshop on*, March 2009, S. 267–273

[habil. Jörg Roth 2009] JÖRG ROTH, Prof. D. habil.: Ortsbezogene Anwendungen und Dienste / Georg-Simon-Ohm-Hochschule Nürnberg. URL http://www.wireless-earth.de/teaching/LBS.pdf, 2009. – Skript

[Kahlbrandt 2002] KAHLBRANDT, Bernd: *Software Engineering mit der Unified Modeling Language*. 2. Auflage. Springer-Verlag GmbH, 2002. – ISBN 3-540-41600-5

[Kogan 2009] KOGAN, Borys: *Indoor Navigationssystem mit dynamischer Beschilderung – Entwicklung und Simulation in einer virtuellen 3D-Umgebung*, Hochschule für Angewandte Wissenschaften Hamburg, Masterarbeit, 2009. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/kogan.pdf

[Kolbe 2008a] KOLBE, Prof. Dr. Thomas H.: 3D Stadtmodellierung mit CityGML / Technische Universität Berlin. URL www.geomv.de/geoforum/2008/praesentationen/A3_Kolbe_CityGML.pdf, 2008. – Skript

[Kolbe 2008b] KOLBE, Prof. Dr. Thomas H.: CityGML, KML und das Open Geospatial Consortium / Institut für Geodäsie und Geoinformationstechnik. URL http://www.igg.tu-berlin.de/uploads/tx_ikgpublication/CityGML_und_KML_Kolbe2008.pdf, 2008. – Tagungsband zum 13. Münchener Fortbildungsseminar Geoinformationssysteme an der Technischen Universität München

[Kolbe 2009] KOLBE, T. H.: Representing and Exchanging 3D City Models with CityGML / Institute for Geodesy and Geoinformation Science. 2009. – Forschungsbericht. – 20 S

[Kolbe u. a. 2005] KOLBE, Thomas H. ; GRÖGER, Gerhard ; PLÜMER, Lutz: CityGML – Interoperable Access to 3D City Models / Institute for Cartography and Geoinformation University of Bonn. URL http://www.citygml.org/fileadmin/count.php?f=fileadmin/citygml/docs/Gi4Dm_2005_Kolbe_Groeger.pdf, 2005. – Report. Letzter Aufruf am 05.09.2009

[Kolbe und Stadler 2008] KOLBE, Thomas H. ; STADLER, Alexandra: International standardisation of 3d city models / Institute for Geodesy and Geoinformation Science. 2008. – Forschungsbericht

[Koychev 2008] KOYCHEV, Milen: *Software-Agenten-Paradigma am Beispiel einer Simulationsumgebung für Indoornavigation im Flughafenkontext*, Hochschule für Angewandte Wissenschaften Hamburg, Masterarbeit, 2008. – URL

http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/koychev.pdf

[Lilienthal 2008] LILIENTHAL, Dr. C.: *Komplexität von Softwarearchitekturen – Stile und Strategien*, Universität Hamburg, Dissertation, 2008. – URL http://www.sub.uni-hamburg.de/opus/volltexte/2008/3725/

[Malaka und Zipf 2000] MALAKA, R. ; ZIPF, A.: DEEP MAP - Challenging IT research in the framework of a tourist information system. In: *Information and Communication Technologies in Tourism 2000*, 2000, S. 15–27

[Nagel 2009] NAGEL, Claus: *Adding your own Application Domain Extensions (ADE) to citygml4j*. 2009. – URL http://opportunity.bv.tu-berlin.de/software/wiki/citygml4j/Adding_your_own_Application_Domain_Extensions_(ADE)_to_citygml4j. – Letzter Aufruf am 01.08.2009

[Napitupulu 2007] NAPITUPULU, Jan: Indoor Map Server in einem Flughafenszenario / Hochschule für Angewandte Wissenschaften Hamburg. URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master06-07-proj/napitupulu/report.pdf, März 2007. – Projektbericht

[Narasimhan 2007] NARASIMHAN, Srihari: *Simulation and Optimized Scheduling of Pedestrian Traffic*, Universität Stuttgart, Dissertation, 2007. – URL http://elib.uni-stuttgart.de/opus/volltexte/2007/3023/pdf/DissNarasimhan.pdf

[OASIS 2002] OASIS: Extensible Address Language (xAL) Standard Description Document for W3C DTD/Schema Version 2.0 / Organization for the Advancement of Structured Information Standards. URL https://www.seegrid.csiro.au/subversion/xmml/oasis/OASIS-CIQ/DOCUMENTS/xAL%20Standard%20V2.pdf, 2002. – Standard Description Document. Letzter Abruf am 20.06.2009

[OASIS 2009] OASIS: OASIS xAL Standard v2.0 / Organization for the Advancement of Structured Information Standards. URL http://www.oasis-open.org/committees/ciq/ciq.html#6, 2009. – XML Standard

[Oblonczek und Zahn 2009] OBLONCZEK, Pascal ; ZAHN, Sebastian: *Indoor-Lokalisierung mit Hilfe neuronaler Netze*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2009

[Orbifold.Net 2009] ORBIFOLD.NET: *WPF patterns: MVC, MVP or MVVM or ...?* 2009. – URL http://www.orbifold.net/default/?p=550. – Letzter Aufruf am 10.08.2009

[Parlowski 2009] PARLOWSKI, Dirk: *Entwicklung eines mobilen Navigationssystems für den öffentlichen Nahverkehr*, HAW Hamburg, Bachelorthesis, 2009

[Pfaff 2007] PFAFF, Thomas: *Entwicklung eines PDA-basierten Indoor-Navigationssystems*, HAW Hamburg, Bachelorthesis, 2007

[Podila 2009] PODILA, Pavan: *FluidKit*. 2009. – URL http://www.codeplex.com/fluidkit. – Letzter Aufruf am 01.09.2009

[Quix 2003] QUIX, Christoph J.: *Metadatenverwaltung zur qualitätsorientierten Informationslogistik in Data- Warehouse-Systemen*, Rheinisch-Westfälischen Technischen Hochschule Aachen, Dissertation, 2003. – URL http://darwin.bth.rwth-aachen.de/opus3/volltexte/2003/733/pdf/03_263.pdf

[Rahimi und Vogt 2008] RAHIMI, Mohammadali ; VOGT, Matthias: *Gestenbasierte Computerinteraktion auf Basis von Multitouch-Technologie*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2008. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/rahimi_vogt.pdf

[Rech 2008] RECH, Monika: CityGML, ein Standard made in Germany. In: *gis-Business* 5 (2008), Nr. 5, S. 12–14

[Rehrl u. a. 2007] REHRL, K. ; GÖLL, N. ; LEITINGER, S. ; BRUNTSCH, St. ; MENTZ, H.-J.: Smartphone-based information and navigation aids for public transport travellers. In: *Location Based Services and TeleCartography*, Springer Berlin Heidelberg, 2007, S. 525–544

[Samaschke 2006] SAMASCHKE, Karsten: *XML.NET – XML und Web Services mit dem .NET-Framework*. Auflage: 1. Entwickler.Press, 2006. – ISBN 978-3935042680

[Schlueter u. a. 2007] SCHLUETER, Sandra ; BENNER, Joachim ; BILDSTEIN, Frank ; DREES, Ruediger ; KOHLHAAS, Andreas ; PENDLINGTON, Mark: External Code Lists for CityGML® / Open Geospatial Consortium Inc. URL http://bp.schemas.opengis.net/07-062/Codelists/CityGML_ExternalCodeLists.xml, 2007. – XML Dokument. Letzter Aufruf am 19.06.2009

[Schumann 2008] SCHUMANN, Alewtina: *Ein einfach benutzbares mobiles Navigationssystem für Fußgänger*. GRIN Verlag, 2008. – ISBN 978-3-640227-76-1

[Sieck 2008] SIECK, Diplom-Sozialökonomin I.: *Systematisches Testen als analytische Qualitätssicherungsmaßsnahme im Software-Entwicklungsprozess*. GRIN Verlag, 2008. – ISBN 978-3-638955-87-4

[Starke 2008] STARKE, Gernot: *Effektive Software-Architekturen. Ein praktischer Leitfaden*. 3., aktualis. u. erw. A. Hanser Fachbuchverlag, 2008. – ISBN 3-446-41215-6

[Stowasser u. a. 1994] STOWASSER, Joseph M. ; PETSCHENING, M. ; SKUTSCH, F.: *Stowasser*. neu bear. und erw. Aufl. Verlag Hölder-Pichler-Tempsky, 1994. – ISBN 3-209-01495-7

[Stropek und Huber 2008] STROPEK, Rainer ; HUBER, Karin: *WPF und XAML : Programmierhandbuch*. 1. Aufl. entwickler.press, 2008. – ISBN 978-3-939084-60-0

[Vandervelde 2006] VANDERVELDE, Fred: *D2X*. 2006. – URL http://fritzenhammer.wordpress.com/2009/07/13/looking-for-d2x/. – Letzter Aufruf am 08.09.2009

[Witt 2008] WITT, Kristoffer A.: *Eine verteilte Anwendung zur Nutzung von Ortsbasierten Diensten mit mobilen Endgeräten*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2008. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/witt.pdf

6 Anhang

6.1 Bearbeitungs- und Konvertierungschritte von DWG Daten zur Vorbereitung für das AnnotationTool

Die folgenden Schritte sind als Vorbereitung für das Importieren eines Stockwerkes – vorliegend als DWG Datei aus dem Facility-Management-System der Hochschule für Angewandte Wissenschaften Hamburg – in das AnnotationTool mittels der Import-Klasse `StoreyFromXAMLImporter` durchzuführen. **Für andere Implementierungen des Interfaces `IStoreyImporter` sind diese Schritte ggf. nicht notwendig!**

1. DWG zu PDF

Zunächst wird die DWG Datei in *mehrere* PDF Dokumente konvertiert – entsprechend der Zielentität (z. B. "Raum") wird aus einem oder mehreren Layern (z. B. "Raumpolygone") jeweils ein PDF-Dokument erzeugt. Zur Konvertierung eignet sich insbesondere *AutoDWG DWG2PDF Converter*[1] oder der *DWG TrueView 2008*[2] (Export per PDF-Plotter über den Druckertreiber). Jedes einzelne PDF-Dokument eines Stockwerks gibt letztlich Auskunft über die vektorielle Beschreibung *eines* Entitätstyps, da ansonsten keine Unterscheidung der Pfade auf den einzelnen Layern mehr möglich wäre!

Die Stockwerksdetails werden in einem erstem PDF-Dokument untergebracht. Folgende Layer sollten hierzu ausgewählt werden:

- Fenster
- Treppe
- Flur
- Lift
- Wand_.* (alle "Wand" Layer)

[1] http://www.autodwg.com/PDF/

[2] http://www.autodesk.de/adsk/servlet/index?siteID=403786&id=10686166

- Säulen

→ `OGxx_Details.pdf` (z. B. `OG11_Details.pdf`)

Die Raumpolygone befinden sich auf dem gleichnamigen Layer. Dieser wird separat exportiert.

→ `OGxx_Raumpolygone.pdf`

Die Pfade für die einzelnen Türen, Aufzüge usw. *können* ebenfalls einzeln exportiert werden. Andernfalls müssen diese im AnnotationTool manuell eingetragen werden.

2. PDF zu XAML

In diesem Arbeitsschritt werden die PDF Dateien eines Stockwerks in Expression Design 2[3] importiert und daraufhin als jeweils eigenes Canvas in *einer* XAML Datei exportiert.

Um PDF Dateien in Expression Design 2 importieren zu können, muss zuerst die Dateiendung von `.pdf` zu `.ai` geändert werden.

Bsp.: `OG11_Details.pdf` → `OG11_Details.ai`

Nun sind folgende Schritte zu bewerkstelligen:

- Datei → Neues Projekt
- Import OGxx_xyz.ai (jeweils für alle PDF-Dokumente eines Stockwerks)
- Breite auf 1100px setzen (zur Normierung)
- Position pixelgenau in die linke obere Ecke (Importierte Ebenen müssen sich korrekt überlappen)
- Datei → Exportieren (Canvas-XAML)

→ `OGxx.xaml`

Nun kann das Stockwerk als `OGxx.xaml` im AnnotationTool mit Hilfe des `IStoreyImporters` importiert werden.

[3] http://www.microsoft.com/germany/expression/products/Overview.aspx?key=design

6.2 Screenshots

Es folgen Screenshots des AnnotationTools und WelcomeScreens.

Abbildung 6.1: Screenshot des AnnotationTool (OG11)

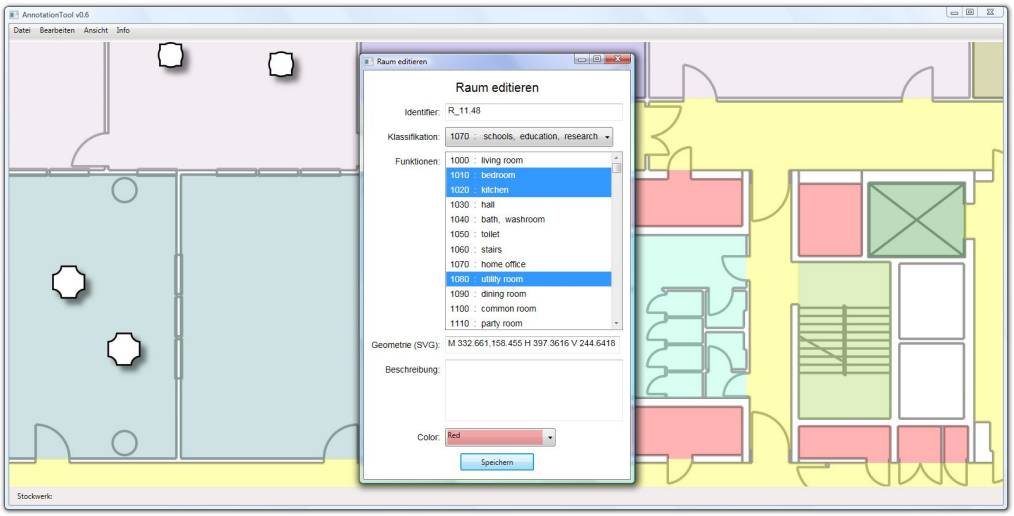

Abbildung 6.2: AnnotationTool mit `BasePropertyDialog` eines Raumes

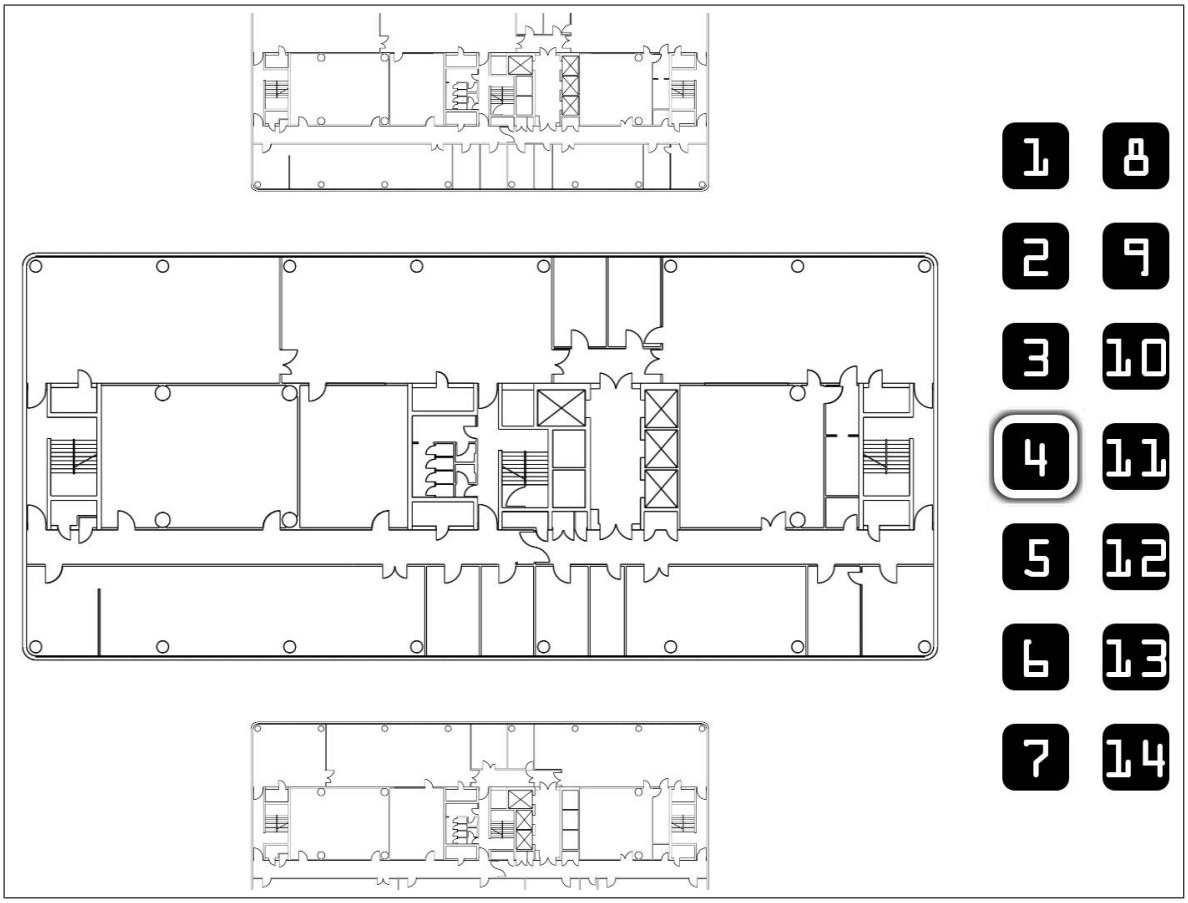

Abbildung 6.3: Screenshot des WelcomeScreens (invertierte Farben)