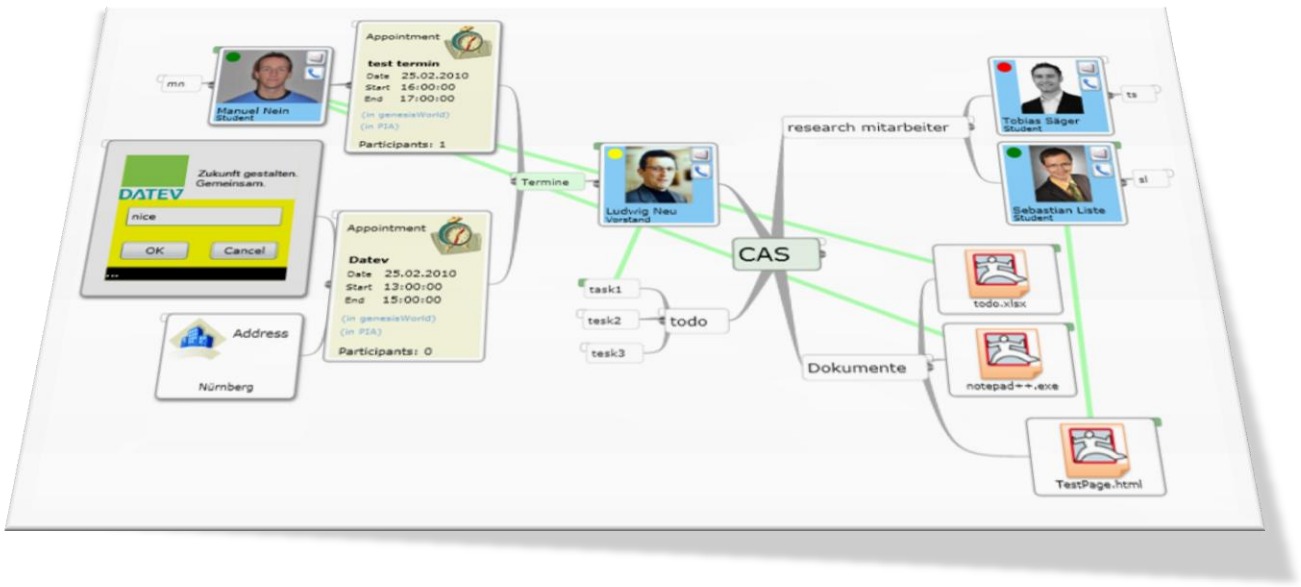
Bachelor-Thesis

Thema:

Konzeptionierung und Entwicklung einer Lösung zur kollaborativen Modellierung einer CRM-Domäne in Mindmaps

Topic:

Design and development of a solution to enable collaborative modeling of a CRM domain within mind maps



Abgabedatum: 12.03.2010

Manuel Naujoks
Referent: Prof. Fuchß
Korreferent: Prof. Philipp

CAS Software AG
Wilhelm-Schickard-Str. 8 - 12
76131 Karlsruhe

Hochschule Karlsruhe
Technik und Wirtschaft
Moltkestraße 30
76133 Karlsruhe

Erklärung

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die den Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

(Ort, Datum) (Unterschrift)

Zusammenfassung

Mindmaps dienen der schnellen Strukturierung von Gedankengängen und sind somit oft Bestandteil von Planungen. Damit spielen sie eine Rolle in Unternehmen die beispielsweise kundennah agieren wollen. Gerade in einem agilen Umfeld ist die effiziente Informationsverwaltung ein entscheidender Faktor. Mehrere Personen wollen mit den gleichen Daten möglichst zeitgleich arbeiten und Zusammenhänge erzeugen, die sie zu einem späteren Zeitpunkt wiederverwenden wollen.

Um dies zu ermöglichen, werden in dieser Thesis zwei Erweiterungen des Mind-Mapping-Konzepts geplant und implementiert. Die erste Erweiterung ist das gemeinschaftliche Modellieren von Informationsstrukturen. Dafür wird ein Mindmap-Editor benötigt, mit dem mehrere Personen an der gleichen Mindmap zur gleichen Zeit arbeiten können. In der Vorarbeit wurde ein einfacher Editor in Silverlight erstellt, der um Funktionalität für Zusammenarbeit erweitert wird. Es werden die grundlegenden Operationen definiert, die für das Arbeiten mit Mindmaps wichtig sind und untersucht, wie diese ausgeführt werden können, sodass gemeinschaftliches Arbeiten ermöglicht werden kann. Die Implementierung dieses Konzepts nutzt WCF-Dienste über das Internet, die von dem Mindmap-Editor verwendet werden.

Das zweite Mind-Mapping-Konzept, das in dieser Thesis behandelt wird, ist die Wiederverwendung von Informationen aus der Mindmap. Dafür wird zunächst eine Anbindung an ein CRM-System geplant, um deren Daten in der Mindmap modellieren zu können. Anschließend wird eine Lösung konzipiert, die es erlaubt, dass sich die modellierten Strukturen auf die CRM-Objekte auswirken. Für die Implementierung wird eine beispielhafte CRM-Domäne verwendet und der Editor aus der Vorarbeit erweitert, damit dieser die Daten des CRM-Systems als Knoten in Mindmaps verwenden kann. Durch diese Integration wird eine Verbindung zwischen CRM-System und Mindmaps hergestellt, die das Wiederverwenden von Daten ausreichend unterstützt.

Die Implementierung dieser Erweiterungen erfolgt im Rahmen eines Prototyps und wird in Silverlight vorgenommen.

Abstract

Mind maps can be really helpful when it comes to organizing thoughts quickly. Therefore they are used in companies that for example want to operate close to the customers. Efficient management of information is of tremendous value particularly in an agile environment. Many people want to work with the same data in a real-time-like fashion to create connections of data, which they intend to reuse at a later point in time.

This thesis covers the design and the implementation of two extensions of the concept of mind mapping. The first extension concerns the collaborative modeling of structures with information. This requires a mind map editor that enables many people to work with the same mind map simultaneously. A simple editor has been implemented durring the preparatory work period. This editor was built using Silverlight and gets extended to provide collaborative functionality. Basic operations that represent the essence of working with mind maps are being defined and analyzed to determine a way to invoke them, so that they enable collaboration. In regard to the implementation of this concept, the mind map editor consumes WCF services over the internet.

The second concept of mind mapping that is covered in this thesis is reuse of information. In order to being able to do that, the integration of a CRM system is designed to enable modeling of that CRM data in mind maps. To allow CRM objects to be effected by the modeled structures, a solution is designed. An exemplary CRM domain is used for the implementation and the extension of the mind map editor. This lets the editor become capable of using data of the CRM system as nodes in the mind map document. Integration like this provides a connection between a CRM system and mind maps to offer an infrastructure to reuse data sufficiently.

All extensions are implemented as a prototype. Silverlight is the technology that is used.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeausschnittverzeichnis

Tabellenverzeichnis

1 Einleitung

Optimales Informationsmanagement kann ein entscheidender Faktor bei der Durchführung eines kundenorientierten Projekts sein. Wenn die beteiligten Parteien nicht über die gleichen Informationen verfügen, kann das Projekt leicht in Zeitverzug geraten und sogar zu Vertragsstrafen führen. Ein Kunde möchte stets wissen, über welche Informationen sein Projektpartner verfügt, um rechtzeitig das notwendige Wissen für eine erfolgreiche Projektabwicklung zur Verfügung stellen zu können. Auf der anderen Seite ist es im Interesse des Auftragsnehmers, die Kunden möglichst früh und sehr eng in den Prozess zu integrieren, um das Ergebnis erzielen zu können, das den Erwartungen aller beteiligten Parteien entspricht. Um die relevanten Informationen projektspezifisch zu speichern, verwenden Firmen heutzutage geeignete Computersysteme.

Schwierigkeiten bestehen dabei aber bei der Informationserzeugung. Zusammen mit Kunden müssen in Besprechungen und Konferenzen Details des zu realisierenden Projekts geklärt werden. Die daraus entstehenden Zusammenhänge müssen als Informationsstrukturen effektiv erstellt werden, wofür häufig Editoren eingesetzt werden, die umfangreiche Mindmaps erzeugen können. In Mindmaps können zusammengehörende Informationen einfach und übersichtlich strukturiert werden und eignen sich damit für die Kommunikation von Zusammenhängen zwischen den einzelnen Projektteilnehmern. Idealerweise unterstützt der Mindmap-Editor die zeitgleiche und gemeinsame Bearbeitung von verschiedenen Personen, sodass Strukturen gemeinschaftlich erstellt werden können. Neben reinen Projekteigenschaften, können in Mindmaps auch Termine geplant und auf Dokumente verwiesen werden. Die entstehende Mindmap kann wichtige Informationen aus verschiedenen Quellen verknüpfen, sodass komplexere Zusammenhänge der projektrelevanten Informationen stets übersichtlich bleiben.

Nachdem ein solches Mindmap-Dokument erstellt wurde, müssen die gesammelten Informationen und deren Strukturen in das geeignete Computersystem übertragen werden. Dies ist ein manueller Vorgang, der abhängig von dem Mindmap-Editor und dem Hintergrundsystem mehr oder weniger zeitaufwendig ist. Wenn die Informationen übernommen wurden, kann es vorkommen, dass Kunden weitere Anforderungen stellen oder sich Bestandteile des Projekts ändern. In einem solchen Fall müssen die ursprünglich erstellten Mindmaps entsprechend geändert werden. Allerdings müssen anschließend auch die Daten im Hintergrundsystem manuell angepasst werden. Diese Anpassungen zwischen den Daten, die in übersichtlichen Strukturen einer Mindmap erstellt werden, und den Daten, die im Hintergrundsystem gespeichert sind, müssen immer wieder durchgeführt werden.

1.1 Umfeld

Die CAS Software AG entwickelt CRM-Software für den Mittelstand [CAS10]. Dabei möchte sie ihren Kunden die Möglichkeit bieten, die Verwaltung von Kundenbeziehungen so erfolgreich wie möglich zu gestalten. Diese Kunden können in Gesprächen mit ihren Auftraggebern Mindmaps erstellen, deren Zusammenhänge sie aber manuell in genesisWorld oder CAS PIA, den CRM-Systemen der CAS Software AG, eintragen müssen. Diese CRM-Systeme erleichtern die Interaktion mit den Kunden, bieten aber noch keine Möglichkeit, die Planung von solchen Interaktionen zu unterstützen. Die erstellten und verwendeten Mindmaps verfügen über keine direkte Anbindung an die CRM-Systeme, sodass Kunden der CAS Software AG zwei separate Systeme einsetzen müssen, um Mind-Mapping in Kombination mit CRM betreiben zu können. Die CAS Software AG ist daran interessiert, ihren Kunden eine optimal integrierte Lösung anbieten zu können, um Kundenbeziehungen sehr effizient verwalten zu können.

Die dafür nötigen Innovationen werden im Rahmen von Forschungsprojekten und Abschlussarbeiten analysiert und prototypisch konzipiert, worum es sich auch bei dieser Bachelor-Thesis handelt.

1.2 Zielsetzung

Im Rahmen dieser Bachelor-Thesis soll eine Möglichkeit konzipiert und beispielhaft entwickelt werden, die es erlaubt Mindmaps in Kombination mit einem CRM-System gemeinschaftlich verwenden zu können. Dafür soll ein spezieller Mindmap-Editor realisiert werden, der das Erstellen von Informationsstrukturen effektiv unterstützt. Da es sich bei dem CRM-System CAS PIA der CAS Software AG um eine webbasierte Lösung handelt [CAS10], soll die Erstellung von Mindmaps auch webbasiert vorgenommen werden können. Um diesen Editor Browser- und Betriebssystemunabhängig zu implementieren, soll Silverlight verwendet werden. Die konkrete Anbindung an ein CRM-System der CAS Software AG muss in dieser Thesis nicht erfolgen, da eine beispielhafte Anbindung im Rahmen des zu entwickelnden Prototyp vollkommen ausreichend ist. Die Konzeptionierung dieser Anbindung, die aus der Sicht des Mindmap-Editors betrachtet werden soll, muss demonstrativ mit beispielhaften Daten aus dem CRM-System erfolgen können. So soll es möglich sein, kundenorientierte Mindmaps zu erstellen, die Verbindungen zu den Strukturen in dem CRM-System haben können. Da Kunden der CAS Software AG zusammen mit ihren Kunden Projekte durchführen, ist eine Lösung notwendig, die das enge Zusammenarbeiten zwischen Projektteilnehmern über das Internet in echtzeitähnlicher Weise unterstützen kann. Damit spielt die gemeinschaftliche Modellierung von Mindmaps in dieser Bachelor-Thesis eine wichtige Rolle.

Mit einem solchen Mindmap-Editor kann der manuelle Schritt der Übernahme von Informationen aus Mindmaps in das CRM-System automatisiert werden, indem der Editor über eine direkte Anbindung an das CRM-System verfügt. Kunden der CAS Software AG können damit Mind-Mapping und CRM als vollständig integriertes System nutzen, um Interaktionen mit ihren Kunden noch effizienter zu gestalten.

1.3 Aufbau der Arbeit

Diese Thesis besteht aus fünf Kapiteln, die das vorgestellte Thema behandeln. Dabei werden in Kapitel 2 Grundlagen zunächst allgemeine Grundlagen beschrieben, sowie Grundlagen vorgestellt, die sich auf Zusammenarbeit und Domänenmodellierung in Mindmaps beziehen. Danach werden in Kapitel 3 Ausgangslage der Mindmap-Editor, der in der Vorarbeit dieser Thesis erstellt wurde, sowie die konkrete Aufgabenstellung und Vorgehensweise beschrieben. Es werden ebenfalls allgemeine Anforderungen an die nötigen Erweiterungen des Editors und daraus resultierende Anwendungsfälle vorgestellt. In Kapitel 4 Gemeinschaftliche Modellierung beginnt der erste Teil dieser Ausarbeitung. Dort werden die speziellen Anforderungen anhand der Anwendungsfälle, die Planung und die Implementierung der gemeinschaftlichen Aspekte in Mindmaps beschrieben. Danach wird in Kapitel 5 Domänenmodellierung der zweite Teil behandelt, indem ebenfalls auf spezielle Anforderungen, Entwurf und Implementierung eingegangen wird. Die Kapitel 4 Gemeinschaftliche Modellierung und 5 Domänenmodellierung werden jeweils mit einem Vergleich des Ergebnisses mit den geplanten und angeforderten Funktionalitäten und Verhaltensweisen beendet. Eine allgemeine Zusammenfassung ist in Kapitel 6 Fazit zu finden. Dort werden abschließend das Gesamtergebnis der beiden Teile dieser Thesis beschrieben und die Kernaussagen zusammengefasst.

2 Grundlagen

In diesem Kapitel werden die Grundlagen vorgestellt, die für das Verständnis dieser Thesis relevant sind. Dazu wird zunächst auf einige allgemeine Grundlagen eingegangen. Da sämtliche Implementierungen, die im Rahmen dieser Ausarbeitung beschrieben werden, in Silverlight vorgenommen werden, wird diese Technologie beschrieben. Anschließend werden die Grundlagen erweiterten Mind-Mappings vorgestellt. Dabei wird geklärt, was Mindmaps sind und wie sie eingesetzt werden können. Danach werden Grundlagen in Bezug auf Zusammenarbeit und Domänenmodellierung vorgestellt.

2.1 Allgemeine Grundlagen

In dieser Thesis werden einige Technologien verwendet, die in diesem Abschnitt beschreiben werden. So wird neben Silverlight auch WCF beschrieben. Zusätzlich wird auf eine spezielle Syntax für anonyme Methoden eingegangen. Abschließend wird die verwendete Syntax bei der Modellierung von Klassendiagrammen in UML in Bezug auf Properties in C#, vorgestellt.

2.1.1 Silverlight

Seit Jahren ist Adobe Flash[1] aus dem Web nicht mehr wegzudenken. Mit Silverlight[2] gibt es eine ähnliche Technologie von Microsoft [Mic10]. Es handelt sich dabei um ein Plug-in für Webbrowser und ermöglicht das Verwenden einer eingeschränkten Version des .NET Frameworks. Somit können Anwendungen unabhängig vom Betriebssystem und in Programmiersprachen wie C# oder VB.net entwickelt werden. Genau wie die grafische Entwicklungsplattform des .NET Framework mit dem Namen Windows Presentation Foundation (WPF), setzt auch Silverlight die Extensible Application Markup Language (XAML) ein um Oberflächen deklarativ zu beschreiben [Nau08]. Das Plug-in liegt zum Zeitpunkt dieser Ausarbeitung in Version 3.0 vor. Eine dem .NET Framework ähnliche Klassenbibliothek, sowie Garbage Collection machen Silverlight zu einer umfangreichen Technologie, die das Erstellen von Anwendungen für Browser ermöglicht. Die Entwicklung wird dabei durch die Entwicklungsumgebung Visual Studio unterstützt. Silverlight verwendet eine Vererbungshierarchie für grafische Elemente, die im Rahmen dieser Thesis relevant ist und daher nachfolgend beschrieben wird.

UIElement

Aus der Beschreibung von Microsoft [Mic101] geht hervor, das `UIElement` die höchste Klasse in dieser Vererbungshierarchie ist, die Eigenschaften und Funktionen besitzt, die für die grafische Darstellung von Komponenten erforderlich ist. Dabei handelt es sich um die indirekte Oberklasse aller Silverlight-Controls, die eine sichtbare Ausprägung besitzen. `UIElement` ist zusätzlich in der Lage, grundlegende Benutzerinteraktionen zu behandeln.

FrameworkElement

Der Beschreibung von Microsoft [Mic] nach, ist `FrameworkElement` eine direkte Unterklasse von `UIElement`. Sie unterstützt umfangreichere Mechanismen, um grafisches Layout zu realisieren. Dafür besitzt es Eigenschaften, die beispielsweise Breite und Höhe des Elements beschreiben und ist in der Lage, per DataBinding mit anderen Silverlight-Controls verbunden zu werden. Microsoft [Mic103] beschreibt DataBinding als einen Mechanismus, mit dem eine Änderung des Werts einer Eigenschaft automatisch an allen per DataBinding verbundenen Eigenschaften vorgenommen wird.

---

[1] Adobe Flash — Adobes Browser Plug-in Platform: http://www.adobe.com/de/flashplatform/
[2] Microsoft Silverlight — Microsofts Browser Plug-in: http://www.silverlight.net/

UserControl

Um eigene Silverlight-Controls zu definieren, existiert laut Microsoft [Mic102] die Klasse `UserControl`, die eine Unterklasse von `FrameworkElement` ist. Von ihr können eigene Klassen ableiten, um eigene Steuerelemente für Silverlight zu gestalten. `UserControl` besitzt dafür eine `Content`-Eigenschaft, die verwendet werden kann, um ein `UIElement` zu beinhalten. Dadurch erlauben `UserControl`-Objekte die Verschachtelung von Silverlight-Controls.

2.1.2 WCF

Juval Löwy schreibt in seinem Buch [Juv08] über die Windows Communication Foundation (WCF), dass es sich dabei um eine Plattform für das Entwickeln und Bereitstellen von Diensten unter Windows handelt. WCF stellt dafür eine Laufzeitumgebung zur Verfügung, die es ermöglicht, Dienste mit dem .NET Framework und dessen Datentypen zu entwickeln und trotzdem Kompatibilität zu anderen System zu gewährleisten. Juval Löwy beschreibt weiter, dass WCF eingesetzt werden kann, um dienstorientierte Systeme über das Internet zu veröffentlichen und somit für die Entwicklung von Web Services genutzt werden kann. Ein Web Service ist laut Marco Kuhrmann und Gerd Beneken [Kuh07] eine Schnittstelle, die über ein Netzwerk plattformneutral genutzt werden kann. Ein entscheidender Vorteil von WCF gegenüber anderen serviceorientierten Möglichkeiten im .NET Framework ist, dass in Silverlight die Clientseite von WCF unterstützt und Web Services somit direkt in Silverlight konsumiert werden können. Diese Möglichkeit beschreibt Microsoft in [Mic106]. Im Rahmen dieser Thesis ist WCF interessant, da so eine Client-Server-Architektur erstellt werden kann, dessen Serverseite mit dem .NET Framework und dessen Clientseite mit Silverlight entwickelt werden kann. Da Silverlight-Clients über einen Webserver an den Browser ausgeliefert werden, entsteht so für den Nutzer das Gefühl, als würde sich das komplette System, das er nutzt, im Internet befinden. Der Vorteil, dass ein Nutzer neben dem Silverlight-Plug-in nichts installieren muss, äußert sich in dieser Beziehung unterstützend.

2.1.3 Lambda-Expressions

In C# können Delegate-Objekte als typisierte Methodenzeiger verwendet werden. Dabei kann ein Delegate-Objekt auch auf eine anonyme Methode zeigen, dessen Methodenrumpf bei der Initialisierung des Delegate-Objekts angegeben wird, wie es Microsoft in [Mic105] darstellt. Wenn ein Delegate-Objekt ausgeführt wird, wird die Methode aufgerufen, auf die es zeigt. Da diese Methode den Methodenkontext hat, in dem sie definiert wurde, kann sie auf Variablen in diesem Kontext zugreifen. Als anonyme Methode kann ein Delegate-Objekt beispielsweise die lokalen Variablen der definierenden Methode benutzen, selbst wenn letztere nicht mehr aktiv ist. Mit Lambda-Expressions können in C# anonyme Methoden, die einen Rückgabewert haben, sehr einfach definiert werden, wie Microsoft es in [Mic091] zeigt. Anonyme Methoden, die keinen Rückgabewert haben, können als Aktion definiert werden.

- Lambda-Expression: `(x) => x * x` // Funktion die x quadriert!
- Action-Expression: `() => { x = 0; }` // Aktion die x auf 0 setzt!

2.1.4 Properties in UML

Silverlight erlaubt die Verwendung von C# als Programmiersprache und eine dem .NET Framework ähnliche Laufzeitumgebung, wie in Unterabschnitt 2.1.1 Silverlight beschrieben. In C# gibt es das Konzept der erweiterten Eigenschaften, auch Properties genannt, die Getter- und Setter-Methoden für Attribute in attributähnlicher Syntax definieren können. Microsoft beschreibt in [Mic104] diese Eigenschaften als spezielle Methoden, die wie Attribute verwendet werden können. Durch diese

Verwendung werden Properties zu abstrakten Attributen, die Assoziationen der Klasse zu einer anderen Klasse darstellen, ohne die Herkunft der konkreten Referenz zu definieren. In dieser Thesis wird eine Property-Eigenschaft in UML-Notation wie ein normales Attribut behandelt, sodass Assoziationen in UML, Properties im Code entsprechen können. Ein Vorteil von Properties ist, dass sie auch in Schnittstellen definiert werden können. Eine Klasse, die das Interface implementiert, muss diese vorgeschriebenen Eigenschaften dann implementieren. In UML kann die Assoziation zu dem Typ des Properties somit auch direkt von einem Interface erfolgen. Gleichbedeutend mit der Assoziation der Schnittstelle, ist das Darstellen des Properties als Attribut der Schnittstelle. Damit können Beziehungen zwischen Klassen auch mithilfe von Interfaces beschrieben werden, was in dieser Thesis genutzt wird.

2.2 Grundlagen erweiterter Mind-Mapping-Konzepte

Das Thema dieser Thesis beschäftigt sich mit dem Konzept des Mind-Mappings und wie in diesem Konzept gemeinschaftlich CRM-Objekte verwendet werden können. Daher werden in diesem Abschnitt die Begriffe "Mind-Mapping", "Zusammenarbeit" und "CRM-Domäne" beschreiben.

2.2.1 Mind-Mapping

Mindmaps sind grafische Darstellungsformen, die Assoziationen von Informationen darstellen. Das menschliche Gehirn verwaltet ebenfalls Informationen und verknüpft diese miteinander, ähnlich einer Mindmap. Es besteht schätzungsweise aus 1 Billion Gehirnzellen die beliebig miteinander verbunden sein können. Um die Erstellung dieser Verknüpfungen im Gehirn zu unterstützen, hat Tony Buzan ein grafisches Mittel eingeführt, das es dem Menschen ermöglichen soll, viele Information übersichtlich und effektiv außerhalb des Gehirns zu strukturieren. Dieses Mittel stellt er in dem Buch vor, dass er zusammen mit Barry Buzan geschrieben hat [Buz02]. Dabei wird eine zentrale Information betrachtet und davon ausgehend weitere relevante Informationen grafisch assoziiert. Jede dieser Unterinformation stellt anschließend eine neue Möglichkeit dar, beliebige Wissensstrukturen zu verknüpfen. Die Ursache, warum dies von Menschen als besonders übersichtlich wahrgenommen und effektiv erinnert werden kann, liegt in der Beanspruchung mehrere Funktionen des menschlichen Gehirns.

2.2.1.1 Unterstützung des Gehirns

Wie in Buzans Buch [Buz02] beschrieben, ist die linke Gehirnhälfte für geistige Arbeiten verantwortlich, wozu unteranderem Linearität, Logik, Analyse, Wörter und Zahlen gehören. Die rechte Hälfte ist für Gestaltung, Räumlichkeit, Bilder, Farben und ähnliches zuständig. Notizen, die alle diese Aspekte adressieren, lassen sich von Menschen leicht verstehen und erinnern.

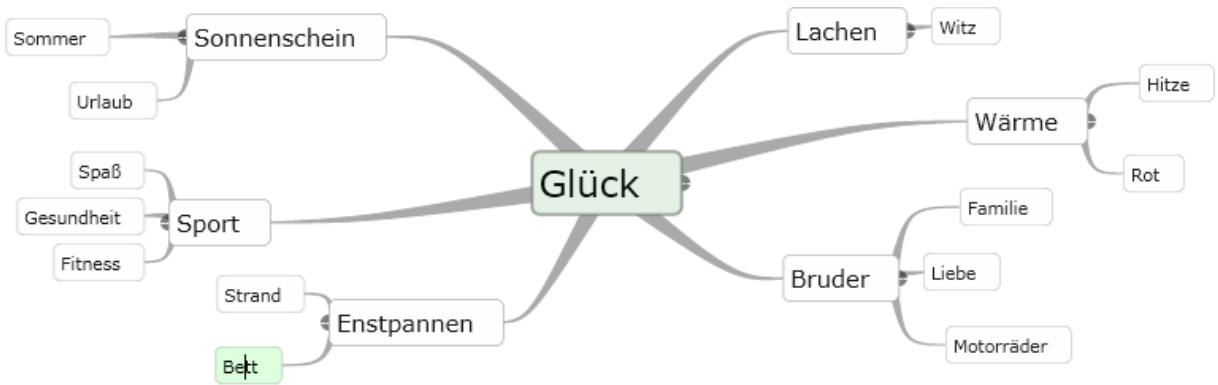


Abbildung 1: Beispiel einer Mindmap

Mindmaps können so eine Notizvariante sein, wenn sie entsprechend eingesetzt werden. Abbildung 1 zeigt eine typische Mindmap zum Thema "Glück". Eine solche Struktur ist ein baumartiger Graph und kann auch als solcher modelliert werden.

2.2.1.2 Abgrenzung zu Graphen

Graphen sind laut Heiko Körner Zweiertupel [Kör07]. Der erste Wert ist eine Menge an Knoten und der zweite ist eine Menge von Kanten zwischen diesen Knoten. Also ist auch eine Mindmap ein Graph, da sie aus Knoten und Verbindungen besteht. Genauer betrachtet entspricht die Struktur der Mindmap einem Baum, da nicht alle möglichen Verbindungen zur Verfügung stehen. Es gilt, dass für alle Möglichkeiten zwei Knoten mit gleicher Entfernung zum Mittelpunkt zu betrachten, diese beiden Knoten keine gemeinsamen Unterknoten haben dürfen, die eine beliebige Entfernung zu den beiden betrachteten Knoten haben. Anders ausgedrückt, kann ein Baum auch als zusammenhängender und azyklischer Graph bezeichnet werden, wie es Heiko Körner beschreibt. Bei einer Mindmap handelt es sich um einen allgemeinen Baum, der besonders grafisch dargestellt wird, nämlich mit einem Mittelpunkt als Wurzel. Diese Struktur wird im weiteren Verlauf der Ausarbeitung mit Topologie bezeichnet. Folglich stellen Mindmaps Untermengen von Graphen dar.

2.2.1.3 Mindmap-Editoren

In diesem Unterabschnitt wird eine Liste an bestehenden Mindmap-Editoren vorgestellt. Da die Implementierung eines eigenen Editors für die Entwicklungen in dieser Thesis unabdingbar ist, werden die bestehenden Programme nur kurz aufgelistet. Tabelle 1 zeigt diese Auflistung und fasst die Informationen von den Webseiten der Hersteller zusammen. Die Zusammenfassung äußert sich dadurch, dass angegeben wird, ob Zusammenarbeit und erweiterte *Knoten* unterstützt werden. Das Programm MindManager unterstützt die gemeinschaftliche Erstellung von Mindmaps, die auch erweiterte *Knoten* beinhalten können. Dabei werden Objekte von Microsoft SharePoint[3] unterstützt. MindPlan, von der Firma Weilgut, unterstützt *Knoten* als Objekte einer eigenen Projektmanagementlösung, sowie Lotus Notes[4].

Programm	Hersteller	Gemeinschaftlich	Mehr als Textknoten
MindManager (Desktop&Browser)	Mindjet http://www.mindjet.com	Ja	Ja (SharePoint Objekte)
Freemind (Desktop)	Open Source http://freemind.sourceforge.net/wiki/index.php/Main_Page	Nein	Nein
MindMeister (Browser)	MeisterLabs http://www.mindmeister.com	Ja	Nein
iMindMap (Desktop)	ThinkBuzan http://www.thinkbuzan.com	Nein	Nein
Xmind (Desktop)	Open Source http://www.xmind.net	Ja	Nein
MindPlan (Desktop)	Weilgut http://www.weilgut.com	Nein	Ja (Weilgut PM & Lotus Notes Objekte)

Tabelle 1: Auflistung vorhandener Mindmap-Editoren

Für alle anderen Mindmap-Editoren konnte nur Unterstützung von *Knoten* mit multimedialen Inhalten, aber keine externen Objekten festgestellt werden.

[3] SharePoint in MindManager: http://www.mindjet.com/products/mindmanager-for-sharepoint/overview
[4] Lotus Notes und Weilgut PM in MindPlan: http://www.weilgut.com/weilgut2_en.nsf/id/pa_mindplan_funktionsuebersicht

2.2.2 Zusammenarbeit

An einem Projekt sind in der Regel mehrere Personen beteiligt. Diese Personen arbeiten zusammen, indem sie beispielsweise Dokumente bearbeiten. Wenn Computer diese Aktivitäten unterstützen, ist nach Peter H. Carstensen und Kjeld Schmidt von *Computer Supported Cooperative Work* (CSCW) die Rede [Car99]. Während mit CSCW das Forschungsgebiet gemeint ist, bezieht sich der Begriff "Groupware", welcher von Louis S. Richman und Julianne Slovak geprägt wurde [Ric87], auf ein System, das Zusammenarbeit unterstützt. Auch wenn diese Artikel vor einigen Jahren geschrieben wurden, hat die Idee von Zusammenarbeit am Computer erst durch die Ankündigung von Google Wave[5] viel Aufmerksamkeit bekommen. Auf der Google Entwicklerkonferenz I/O wurde es im Mai 2009 von Lars Rasmussen vorgestellt [Wen09]. In Google Wave wird Zusammenarbeit fast in Echtzeit über das Internet ermöglicht. Wenn eine Person in der Browseranwendung etwas eintippt, sehen alle beteiligten Personen was gerade getippt wird und können ebenfalls das Dokument editieren. Da Dokumente nicht mehr nur von einer Person bearbeitet werden können, brauchen andere Personen nicht zu warten, bis die erste Person die Bearbeitung beendet hat. Dies führt zu einer gleichzeitigen und damit schnelleren Bearbeitung des Dokuments. Ein weiterer Vorteil besteht darin, dass keine geografische Nähe der Teilnehmer gegeben sein muss, solange sie über eine Internetverbindung verfügen. Damit ist Google Wave ein Beispiel für ein internetbasiertes System das gemeinschaftliches Arbeiten fast in Echtzeit unterstützt.

2.2.3 CRM-Domäne

Um eine CRM-Domäne zu beschreiben, muss zunächst geklärt werden, was CRM und was eine Domäne ist. CRM ist die Abkürzung für Customer Relationship Management und bezeichnet damit ein System, das die Verwaltung von Beziehungen zu Kunden ermöglicht. Wie in Microsofts Whitepaper [Mic09] beschrieben, ist dies möglich- indem das CRM-System die Interaktionen mit den Kunden aufzeichnet und diese Aufzeichnungen nutzt, um beispielsweise Einnahme-Möglichkeiten voll auszuschöpfen und Kundenloyalität zu verbessern. Microsoft schreibt weiterhin, dass CRM-Systeme der Organisation ermöglichen, mehr Wert aus seinen Kundenbeziehungen zu gewinnen, während die Effizienz der operativen Tätigkeit bei der Interaktion mit diesen Kunden gesteigert werden kann. Nachdem CRM als Hilfsmittel für Firmen hiermit vorgestellt wurde, muss noch geklärt werden, was eine Domäne ist. Martin Fowler hat in seinem Buch *Patterns of Enterprise Application Architecture* eine Beschreibung angegeben, die auch in der Übersicht auf seiner Website gefunden werden kann [Mar02]. Genaugenommen, stellt er dort das Domain Model vor, das als eine Abstraktion der Domäne gesehen werden kann. Wenn er also das Domain Model als ein Netz von miteinander verbundenen Objekten beschreibt, die bedeutungsvolle Individuen repräsentieren, dann stellt das, was durch diese Objekte repräsentiert wird, die Domäne dar. Als Beispiele für Individuen, also Domänenobjekte, führt er Einheiten wie "Bestellung auf einem Bestellformular", "Produkt" und "Vertrag" an. Damit ist nun beschrieben was CRM und Domäne sind. Es gilt nun zu klären, was eine CRM-Domäne ist. Die Objekte oder Einheiten, die das CRM-System verwendet, machen die CRM-Domäne aus. In den Produkten der CAS Software AG werden solche Einheiten verwendet, wie aus der Dokumentation für das CRM-System genesisWorld von CAS hervorgeht [CAS09]. Dort werden zum Beispiel Objekte wie "User", "Address", "Calendar", "Note" oder "Document" beschrieben. Die konkrete Beschreibung ist an dieser Stelle nicht relevant. Diese Objekte werden vom CRM-System verwendet um Kundenbeziehungen zu verwalten und stellen somit eine beispielhafte CRM-Domäne dar.

[5] Google Wave Google Wave Preview Website: http://wave.google.com/

3 Ausgangslage

Da sich diese Thesis mit der besonderen Verwendung von Mindmaps beschäftigt, wird ein Editor benötigt, der lizenzfrei, nicht funktional überladen und beliebig anpassbar ist. Um diese Kriterien erfüllen zu können wurde ein entsprechendes Programm in Silverlight entwickelt. Die beispielhafte Mindmap im Unterabschnitt 2.2.1.1 Unterstützung des Gehirns wurde mit diesem proprietären Mindmap-Editor erzeugt, der auch als *MindGraph*-Editor bezeichnet wird. *MindGraph* soll im Rahmen der Vorarbeit die Grundlage für die weiteren Analysen und Entwicklungen in dieser Thesis darstellen. In diesem Kapitel wird zunächst auf das zugrunde liegende Objektmodell, sowie die Funktionalität dieses Editors eingegangen.

3.1 Vorarbeit

Wie im Unter-Unterabschnitt 2.2.1.2 Abgrenzung zu Graphen erläutert, handelt es sich bei der Topologie einer Mindmap um einen Graphen. Sie ist also eine Menge von Knoten, Kanten und deren Zuordnungen. Um solche Strukturen zu erzeugen, werden visuelle und topologische Komponenten der Lösung unterschieden und im Folgenden aus einer technischen Perspektive erläutert. Die beschriebenen Implementierungsdetails sind für die in dieser Thesis behandelten Erweiterungen relevant und werden daher entsprechend vorgestellt.

3.1.1 Visuelle Objekte

Bei einem *Knoten* handelt es sich um ein `UserControl`-Objekt, das verschiedene Punkte enthält, an die beliebige Objekte über Silverlight DataBinding vom Typ `Point` andocken können. Wie in Abbildung 2 dargestellt, verfügt ein *Knoten* über drei dieser Punkte, welche durch die roten Umrandungen angedeutet werden.

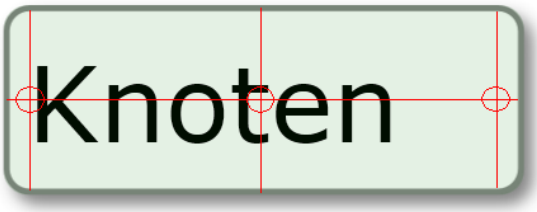


Abbildung 2: Docking Points eines *Knotens* einer Mindmap in *MindGraph*

Diese Docking-Funktionalität wird in der abstrakten Klasse `Dockable` implementiert. Damit ist sichergestellt, dass *Knoten* Verbindungen zu anderen *Knoten* aufbauen können. Knotenspezifische Fähigkeiten werden in der `BasicItem`-Klasse eingeführt. Diese Klasse implementiert auch das Interface `IItem` und besitzt somit eine abstraktere Sicht auf sich selbst. Konkrete *Knoten* lassen sich als Unterklassen von `BasicItem` implementieren, um grafische Aspekte wie zum Beispiel spezielle Darstellungen zu definieren. In Abbildung 3 ist die Vererbungshierarchie der *Knoten* zu sehen. Die Klassendiagramme von den existierenden Klassen der Vorarbeit, die in diesem Kapitel abgebildet sind, wurden von Visual Studio automatisch erstellt und sollen lediglich einen Einblick in die bereits bestehenden Strukturen vermitteln. Erweiterungen dieser Klassen werden in den weiteren Kapiteln näher beschrieben.

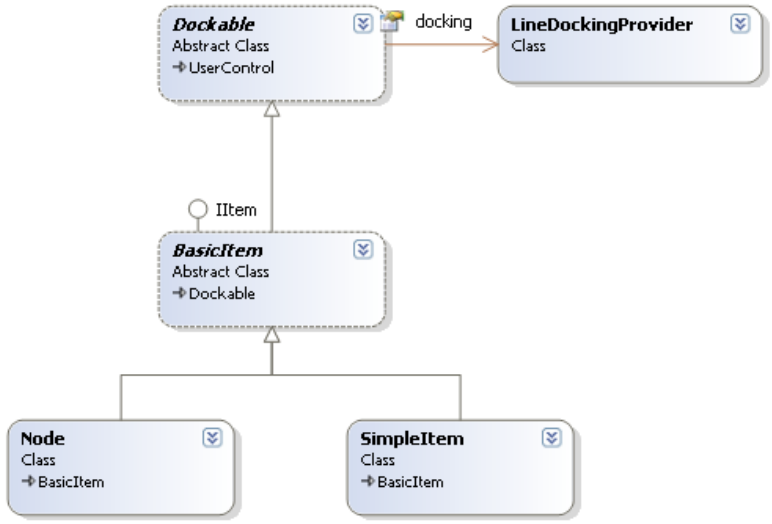


Abbildung 3: Vererbungshierarchie von *Knoten* in *MindGraph*

Um einen Mindmap ähnlichen Graphen erzeugen zu können, dürfen *Knoten* nicht direkt verbunden werden, sondern bedürfen einer Kante. Um dies sicherzustellen existiert für Linien eine andere Oberklasse, nämlich `BaseLine`. `StandardLine` und `CurvedLine` implementieren die abstrakte Schnittstelle von `BaseLine`.

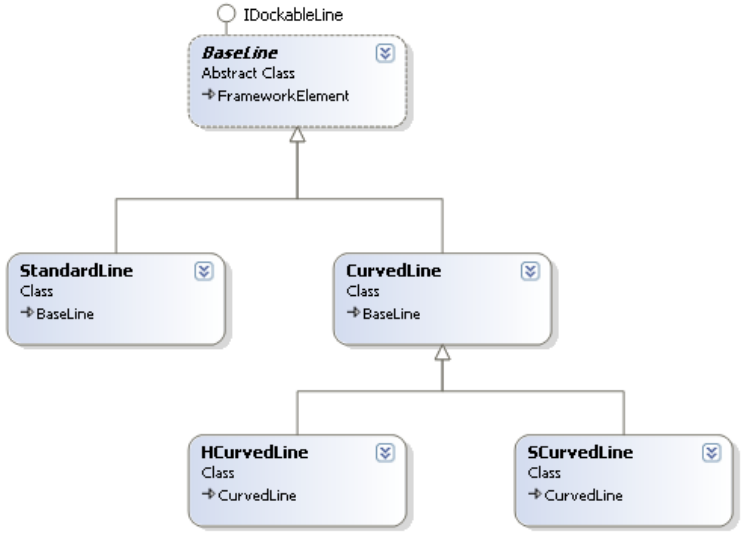


Abbildung 4: Vererbungshierarchie von Kanten in *MindGraph*

Abbildung 4 zeigt die Vererbungshierarchie der Kanten im Mindmap ähnlichen Graphen. `SCurvedLine` und `HCurvedLine` realisieren besondere Krümmungen. Da `BaseLine` die Schnittstelle `IDockableLine` implementiert, können alle konkreten Linien an *Knoten* vom Typ `Dockable` gebunden werden. Dazu müssen der Begin- und der Endpunkt, welche von `IDockableLine` spezifiziert werden, mit dem entsprechenden Andockpunkt eines *Knoten* verknüpft werden. Diese Verknüpfung hat den Vorteil, dass wenn sich die Position eines *Knoten* auf einer Oberfläche ändert, die Linien sich automatisch entsprechend ändern. Folglich muss keine explizite Linienpositionierung stattfinden. Im Weiteren wird der Graph nur noch aus Sicht der *Knoten* gesehen, wozu der Editor eine abstrakte Sicht verwendet, die durch das Interface `IItem` zur Verfügung gestellt wird.

3.1.2 Topologische Objekte

Da ein Graph aus Sicht von `IItem`-Objekten betrachtet werden kann, wird in diesem Unterabschnitt auf topologische Aspekte eingegangen. Zu jedem `IItem`, gehört ein `ItemGroup`-Objekt. Diese Gruppe repräsentiert den *Knoten* selbst, sowie eine Sammlung von dessen Unterknoten. Die Unterknoten sind dabei wieder `IItem`-Objekte. Somit lassen sich Graphen nicht nur visuell sondern auch topologisch abbilden. Um Operationen auf der gesamten Struktur auszuführen, verfügt jede Knotengruppe über eine Referenz auf ein `TopologyControl`-Objekt. Dieses Objekt existiert einmalig für jede Mindmap und kennt den *Knoten*, der den Mittelpunkt darstellt. Abbildung 5 zeigt diesen Zusammenhang der topologischen Objekte.

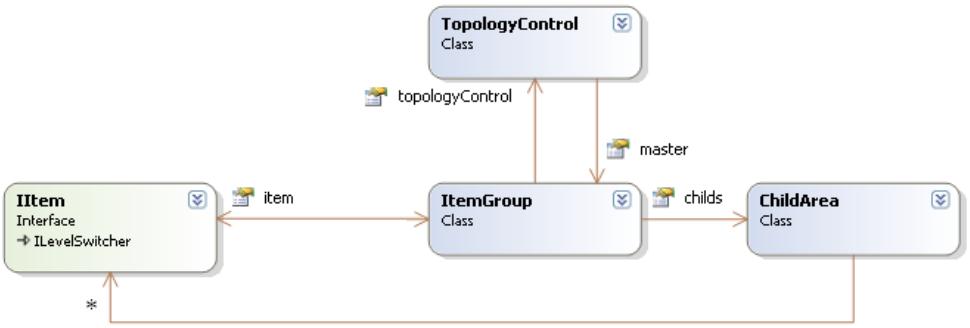


Abbildung 5: Struktur der Topologie einer Mindmap in *MindGraph*

`TopologyControl` verfügt außerdem über Artefakte, die für alle `ItemGroup`-Objekte gelten, wie zum Beispiel Funktionen. Operationen die auf einer Knotengruppe ausgeführt werden können, werden als `IMapFunction`-Objekte implementiert, die über ein `FunctionProvider`-Objekt bereitgestellt werden. Dadurch können sich Funktionen unabhängig von der Topologie der Mindmap ändern und auch getrennt entwickelt werden. `ItemGroup` verfügt über eine `InvokeFunction`-Methode, über die die Funktionen aufgerufen werden können. Wie in der Abbildung 6 zu sehen, existieren momentan zwei Mindmap-Funktionen.

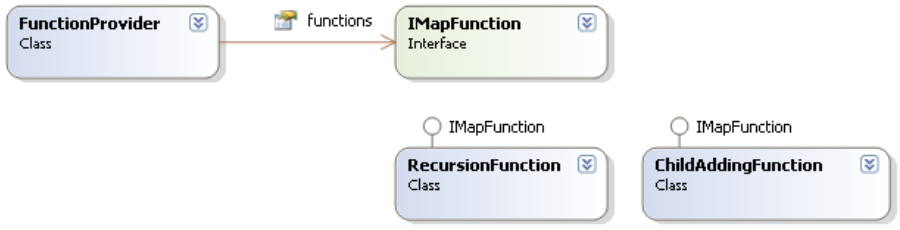


Abbildung 6: Erweiterbare Funktionsliste in *MindGraph*

Die Funktion `RecursionFunction` kann eine Aktion rekursiv auf einer Knotengruppe ausführen. Dabei wird die Aktion zuerst auf dem Hauptknoten der Knotengruppe aufgerufen. Anschließend wird die Aktion auf den Knotengruppen aller Unterknoten ausgeführt. Auf diese Weise können alle Unterknoten, unabhängig von ihrer Entfernung zum betroffenen *Knoten*, von einer Aktion erreicht werden. `ChildAddingFunction` besitzt das Wissen über den Erzeugungsvorgang eines *Knotens* und dessen Integration in die bestehende Topologie. Auch diese Funktion wird auf einer Knotengruppe ausgeführt. Neue *Knoten* können somit von jeder Knotengruppe erzeugt und anschließend an diese angehängt werden.

3.1.3 Mindmap-Editor

Nachdem das grundlegende Objektmodell des Mindmap-Editors beschrieben wurde, wird nachfolgend auf Möglichkeiten der Verwendung des Editors eingegangen. Die Anwendung funktioniert, sofern das Silverlight-Plug-in installiert ist, in einem Webbrowser. Mithilfe der Out-Of-Browser-Funktion von Silverlight, kann die Anwendung auch lokal auf dem Computer installiert werden. Wie in Abbildung 7 zu sehen, besteht die Oberfläche der Anwendung aus einer Arbeitsfläche und aus einer Menüleiste.

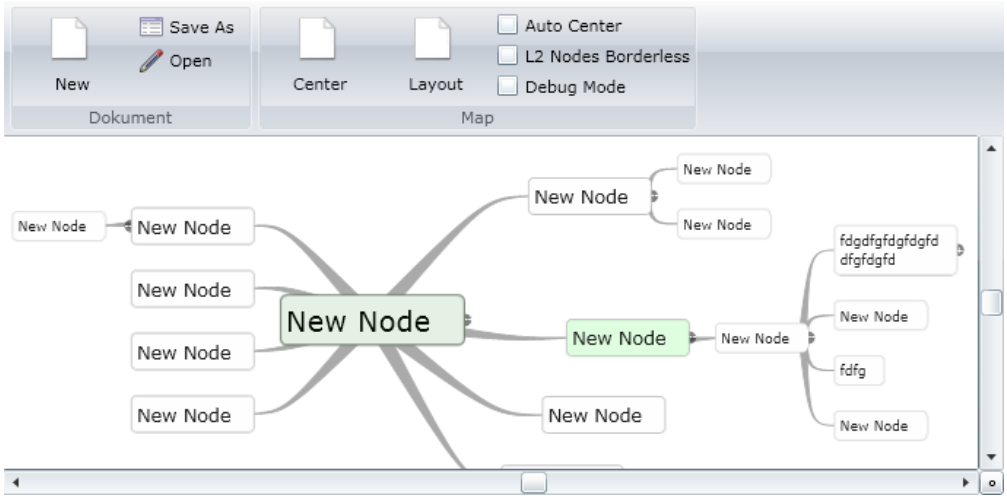


Abbildung 7: Oberfläche des *MindGraph*-Editors

Über das, dem Microsoft Office ähnliche Ribbon[6], können allgemeine Funktionen durchgeführt werden. Dabei gliedert sich das Menü in zwei Bereiche, nämlich den dokumentenspezifischen und den Mindmap-spezifischen Teil. Über den dokumentenspezifischen Teil des Menüs lassen sich Mindmaps erzeugen, speichern und öffnen, wobei zwei Formate zur Auswahl stehen. Das Standardformat entspricht einer proprietären Topologie-Abbildung in XML, auf die hier nicht näher eingegangen wird, da sie für die Funktionalität des Editors nicht relevant ist. Zusätzlich kann der Editor Mindmaps öffnen und speichern, die mit dem Programm Freemind erzeugt wurden. Es ist an dieser Stelle zu erwähnen, dass die Ribbon-Struktur, mit der das erwähnte Menü erstellt wurde, von der bereits bestehenden Anwendung 123teamworks[7] der CAS Software AG übernommen wurde.

Funktionen, die die Topologie der aktuellen Mindmap bearbeiten, werden über Tastenkombinationen und Maus in der Arbeitsfläche vorgenommen. So lassen sich zum Beispiel neue *Knoten* als Unterknoten einfügen, wenn der Benutzer die "Einfügen"-Taste drückt. Mit der "Enter"-Taste lassen sich Nachbarknoten einfügen. Die verfügbaren Tastenkombinationen sind durch bereits vorhandene Editoren inspiriert, die in Unter-Unterabschnitt 2.2.1.3 Mindmap-Editoren vorgestellt wurden. Das topologische Verschieben von *Knoten* erfolgt durch das Verschieben eines *Knoten* in die Nähe eines anderen *Knotens*. Wenn die Maustaste losgelassen wird, wird der *Knoten* mitsamt seinen Unterknoten an den Zielknoten angehängt.

[6] Ribbon Menüstruktur in Microsoft Office 2007: http://office.microsoft.com/en-us/help/ha100898951033.aspx
[7] 123teamworks Teamarbeit-unterstützende Forschungsanwendung der CAS: http://labs.cas.de/123teamworks

3.2 Aufgabenstellung

Nachdem die Struktur der Mindmap und der Editor beschrieben wurden, werden in diesem Abschnitt die Bestandteile dieser Thesis dargestellt. Dabei handelt es sich um zwei Teilaufgaben. Im ersten Teil soll der beschriebene Mindmap-Editor so erweitert werden, dass er gemeinschaftliches Bearbeiten von Mindmaps unterstützt. Der zweite Teil besteht aus der Erweiterung des Editors zur Unterstützung von *Knoten*, die Objekte einer CRM-Domäne repräsentieren können. Da davon ausgegangen wird, dass die Konzeptionierung und Implementierung der Zusammenarbeit in Mindmaps der umfangreichere Teil ist, wird dieser zuerst durchgeführt.

3.2.1 Zusammenarbeit in Mindmaps

Die Zusammenarbeit soll über das Internet und möglichst in Echtzeit ermöglicht werden. Das bedeutet, dass wenn eine Person eine Änderung an einer Mindmap durchführt, beispielsweise einen *Knoten* editiert, dann müssen diese Änderungen bei allen Beteiligten reproduziert werden. Die auf diese Weise entstehenden Konflikte müssen adäquat behandelt werden. Es muss also zunächst eine geeignete Infrastruktur entworfen und implementiert werden, die beliebig viele Personen zur gleichzeitigen Bearbeitung einer Mindmap unterstützt. Die Mindmap-Dokumente sollten dazu in dieser Infrastruktur gespeichert werden können, sodass sie vom Internet aus erreichbar sind. So kann derjenige, der eine Mindmap-Sitzung initiiert, weitere Bearbeiter einladen, unabhängig davon wo diese sich befinden. Es ist auch ein Rechtesystem erforderlich, damit die beteiligten Personen nur Dokumente bearbeiten, die sie besitzen oder zu deren Bearbeitung sie eingeladen wurden. Abschließend muss der Editor an diese Infrastruktur angepasst werden.

In der unteren Auflistung werden die erwähnten High-Level-Anforderungen als Liste zusammengefasst, um im Verlauf des Dokuments darauf verweisen zu können.

(K1) Online Zusammenarbeit
Für Teilnehmer einer Sitzung muss Zusammenarbeit in der Mindmap über das Internet möglich sein.

(K2) Konfliktbehandlung
Auftretende Konflikte müssen so behandelt werden, dass das Dokument konsistent bleibt.

(K3) Multiuserfähigkeit
Mehrere Personen müssen gleichzeitig und zusammen an einer Mindmap arbeiten können.

(K4) Dokumentenspeicher
Mindmaps müssen über das Internet zugänglich sein, aber auch lokal gespeichert werden können.

(K5) Rechtesystem
Das System muss sicherstellen, dass keine Aktionen von Teilnehmern durchgeführt werden können, denen nicht die nötigen Berechtigungen zugewiesen wurden.

3.2.2 CRM-Objekte in Mindmaps

Um Einheiten aus einer CRM-Domäne in Mindmaps modellieren zu können, muss der Mindmap-Editor diese Einheiten kennen. Zusätzlich müssen diese Einheiten sich wie normale *Knoten* verhalten, damit die Modellierung unabhängig von den konkreten *Knoten* vorgenommen werden kann. Dies betrifft hauptsächlich die gemeinschaftliche Modellierung. Trotzdem müssen die *Knoten* ihre eigenen Bedeutungen kennen um sich entsprechend verhalten zu können. Sie müssen also die Verbindung zu den Domänenobjekten, die sie repräsentieren, herstellen können und somit in einem speziellen Kontext verwendet werden können. Da es vorkommen kann, dass Domänenknoten nicht vollständig

sind und nachträglich neue Domänenobjekte erstellt werden können, muss auch der Mindmap-Editor in der Hinsicht erweiterbar sein, sodass neue Knotentypen jederzeit auf einfache Art und Weise definiert werden können.

Die damit erwähnten Anforderungen sind in der nachfolgenden Auflistung zusammengetragen und stellen die allgemeinen Anforderungen an die Domänenmodellierung in einem Mindmap-Editor dar.

(D1) Domänenknoten
Es müssen *Knoten* in Mindmaps verwendet werden können, die Domänen abbilden können.

(D2) Kontextabhängigkeit
Die Domänenknoten müssen über ihre eigene Datenanbindung verfügen und auf ihre Umgebung reagieren können.

(D3) Erweiterbarkeit
Eigene Domänenknoten müssen nachträglich definiert werden können.

3.2.3 Use-Cases

Aus den oben aufgelisteten Anforderungen für Zusammenarbeit und Domänenmodellierung ergeben sich zwei funktionale Sichten auf das zu erstellende System. Die erste Sicht ist aus der Perspektive der Zusammenarbeit zu sehen und enthält vier Use-Cases, die im Diagramm in Abbildung 8 abgebildet sind. Dabei interagieren die Teilnehmer direkt mit den Use-Cases. Von der Infrastruktur, die aus Clients und Server besteht, wird damit abstrahiert.

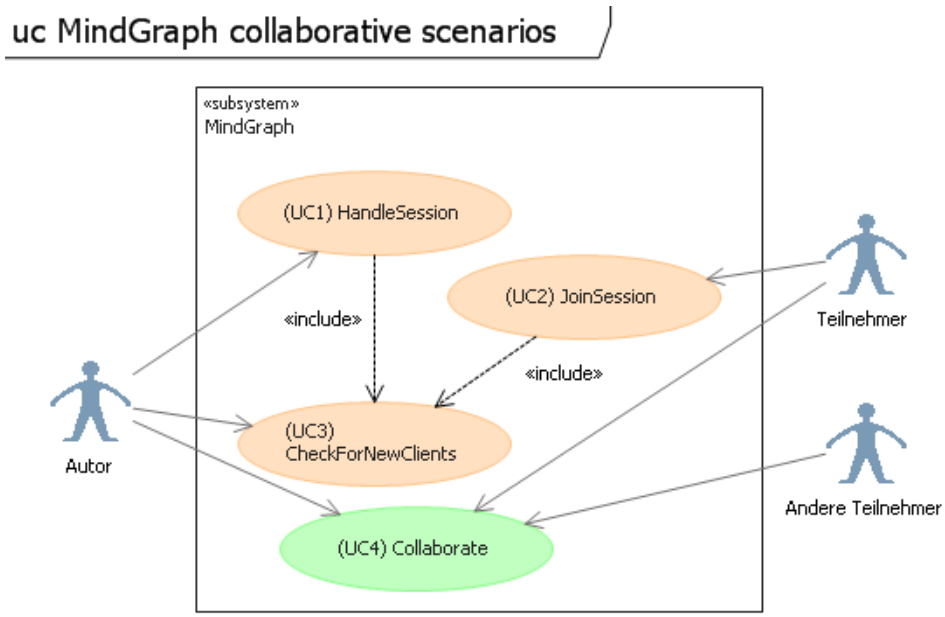


Abbildung 8: Use-Cases für Zusammenarbeit in *MindGraph*

(UC1) Sitzung verwalten
Um Anforderung (K3) Multiuserfähigkeit erfüllen zu können, muss Zusammenarbeit in Sitzungen erfolgen, die von einem speziellen Teilnehmer, einem Autor, verwaltet werden.

(UC2) Sitzung beitreten
Um Anforderung (K3) Multiuserfähigkeit erfüllen zu können, müssen Teilnehmer Sitzungen beitreten können. Der Autor kann dies erlauben oder verweigern (Anforderung (K5) Rechtesystem).

(UC3) Neue Teilnehmer ermitteln
Der Autor einer Sitzung muss neue Teilnehmer ermitteln können, um diese zu zulassen oder zu verweigern.

(UC4) Zusammenarbeiten
Der Autor und alle Teilnehmer einer Sitzung müssen gemeinschaftlich die Mindmap bearbeiten können. Konkrete Mindmap-Operationen sind bereits vorhanden, sodass nur deren gemeinschaftliche Aspekte betrachtet werden müssen und nicht die Operationen selbst.

Die andere Seite des Systems betrifft die Modellierung von Domänenobjekten. Um die Anforderungen aus Unterabschnitt 3.2.2 CRM-Objekte in Mindmaps zu erfüllen, sind zwei Use-Cases nötig, die im Diagramm in Abbildung 9 dargestellt sind.

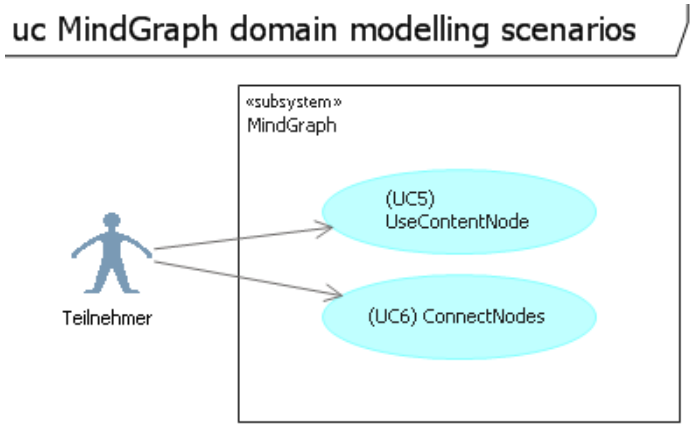


Abbildung 9: Use-Cases für Domänenmodellierung in *MindGraph*

(UC5) Verwenden von speziellen Knoten
Um Anforderung (D1) Domänenknoten erfüllen zu können, müssen Teilnehmer spezielle *Knoten* mit frei definierbarem Inhalt (Anforderung (D3) Erweiterbarkeit) benutzen können.

(UC6) Verbinden von Knoten
Um Anforderung (D2) Kontextabhängigkeit erfüllen zu können, müssen alle *Knoten* in der Mindmap neben ihren eigenen Datenanbindungen auch Verbindung zu anderen *Knoten* haben können. Diese Verbindungen müssen von Teilnehmern hergestellt werden können.

3.2.4 Architektur
Nachdem die insgesamt sechs Use-Cases vorgestellt wurden, wird eine Architektur benötigt, in der diese implementiert werden können. Dabei stellt der in Abschnitt 3.1 Vorarbeit vorgestellte Mindmap-Editor mit dem Namen *MindGraph* die bereits existierende Komponente dar, die um die beschriebenen Use-Cases erweitert werden soll. Dafür wurden alle Use-Cases in einem Untersystem mit der Bezeichnung *MindGraph* definiert, welches im Komponentendiagramm in Abbildung 10 der Komponente "MindGraph" entspricht. Die Unterkomponente "MindMapCore" enthält dabei sämtliche Funktionalität, die in der Vorarbeit entwickelt wurde. Die Erweiterung für Zusammenarbeit in *MindGraph* wird durch eine eigene Komponente mit der Bezeichnung "Client Proxy" erreicht, die einen administrativen ("Administration Service" Komponente) und einen Zusammenarbeit ermöglichenden Dienst ("Collaboration Service" Komponente) nutzt. Beide Dienste sollen auf einem Server laufen und gemäß der Anforderung (K1) Online Zusammenarbeit über das Internet erreichbar sein. Der Zusammenarbeit ermöglichende Dienst behandelt sämtliche Nutzeraktionen in dem

Mindmap-Editor als *Ereignisse* einer Sitzung, die durch den administrativen Dienst verwaltet werden soll. Um diese Dienste nutzen zu können, abstrahiert "Client Proxy" davon und ermöglicht eine Integration in die bestehende Komponente "MindGraph" mit Hilfe einer Unterkomponente mit der Bezeichnung "Collaboration". In dieser Unterkomponente müssen die Use-Cases (UC1) Sitzung verwalten, (UC2) Sitzung beitreten, (UC3) Neue Teilnehmer ermitteln und (UC4) Zusammenarbeiten realisiert werden. "MindMapCore" delegiert dazu Aktionen an die Unterkomponente "Collaboration" und wird über *Ereignisse* von den Diensten informiert. Das Komponentendiagramm in Abbildung 10 zeigt die daraus resultierenden Abhängigkeiten der Komponenten.

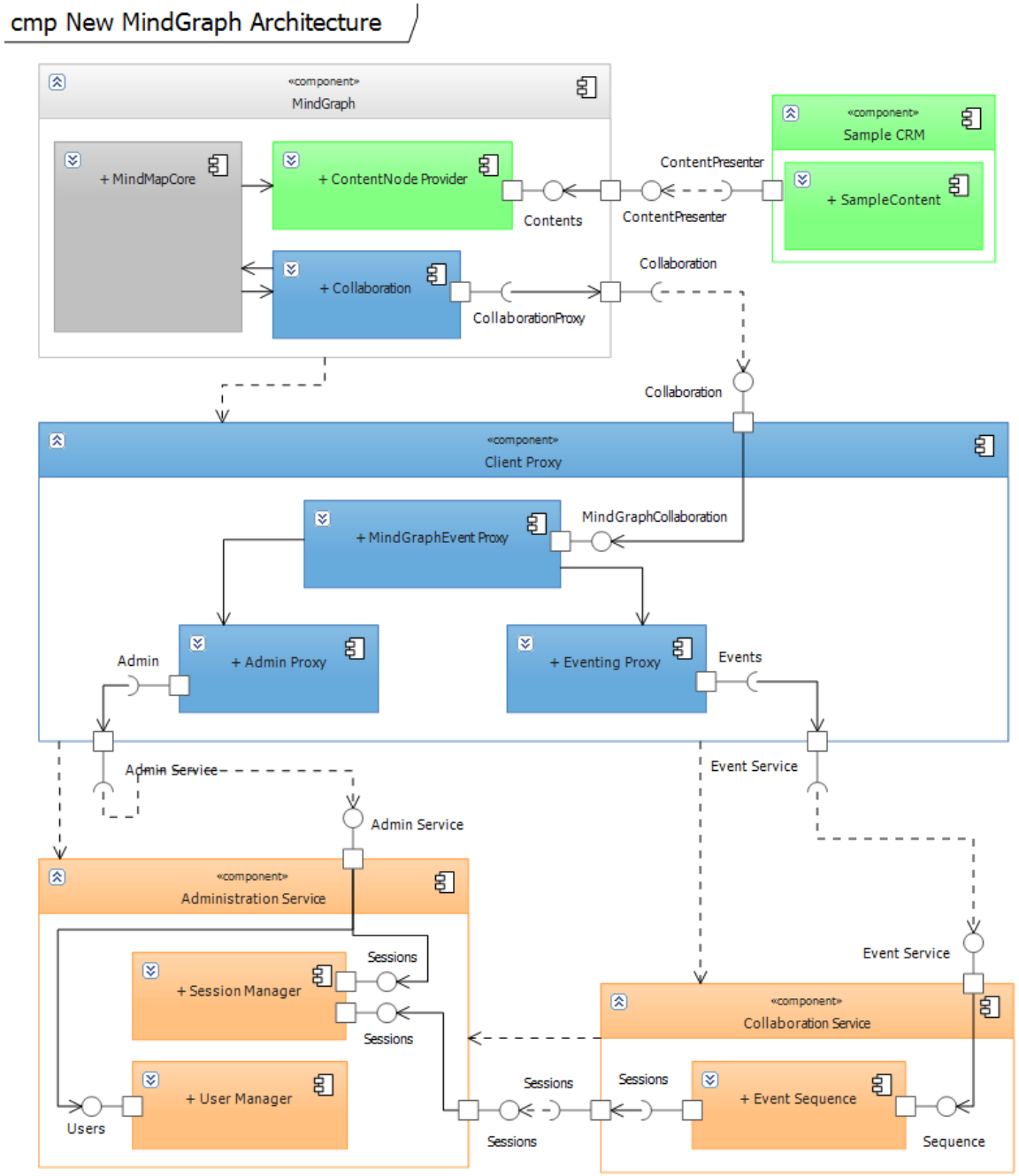


Abbildung 10: Architektur des neuen Systems mit *MindGraph* als Ursprung

Die zweite Erweiterung betrifft die Modellierung von Domänenobjekten. Dafür muss nach Anforderung (D1) Domänenknoten eine Möglichkeit in *MindGraph* geschaffen werden, eigene *Knoten* zu benutzen. Diese eigenen *Knoten* werden nach Anforderung (D3) Erweiterbarkeit in einer

eigenen Komponente definiert, die als "Sample CRM" bezeichnet wird. Durch eine Unterkomponente mit der Bezeichnung "ContentNode Provider" erfolgt die Anbindung, die von dem Nutzer im Rahmen des Use-Case (UC5) Verwenden von speziellen Knoten genutzt werden kann. Für den Use-Case (UC6) Verbinden von Knoten muss "MindMapCore" und "ContentNode Provider" erweitert werden. Nachdem die grobe Architektur hiermit vorgestellt wurde, wird das komplette System, inklusive Dienste im Folgenden mit *piamind* bezeichnet.

3.3 Vorgehen

Wie bereits in Abschnitt 3.2 Aufgabenstellung erwähnt, besteht diese Thesis aus zwei Teilen. Während sich der erste Teil mit Aspekten der Zusammenarbeit in Mindmaps beschäftigt, besteht der zweite aus der Modellierung von Domänenobjekten. In diesem Abschnitt wird beschrieben, wie lange und in welchen Schritten die erwähnten Teile konzipiert und implementiert werden sollen. Dabei wird der erste Teil auch als erste Iteration bezeichnet, die aus vier Arbeitspaketen besteht.

3.3.1 Iteration 1 (Zusammenarbeit)

In der ersten Iteration sollen die vier Use-Cases (UC1) Sitzung verwalten, (UC2) Sitzung beitreten, (UC3) Neue Teilnehmer ermitteln und (UC4) Zusammenarbeiten realisiert werden. Dafür soll in der ersten Woche die Servicelandschaft für die Unterstützung von Zusammenarbeit geplant werden. Anschließend folgt die Implementierung derselben. Wenn die Serverseite funktionsfähig ist, kann die neue Funktionalität in den bestehenden Editor integriert werden. Für diese Planung und Vorbereitung ist eine Woche vorgesehen. In der letzten Woche dieser Iteration soll der Editor endgültig erweitert werden. In Tabelle 2 sind die Arbeitspakete als Tätigkeiten aufgelistet. Für diese vier Tätigkeiten sind insgesamt fünf Wochen eingeplant.

Tätigkeit	Geplant
Infrastruktur für online Zusammenarbeit konzipieren	1 Woche
Infrastruktur für online Zusammenarbeit implementieren	2 Woche
Editorerweiterung für online Zusammenarbeit vorbereiten	1 Woche
Editorerweiterung für online Zusammenarbeit integrieren	1 Woche

Tabelle 2: Arbeitspakete der Iteration 1

3.3.2 Iteration 2 (CRM-Domäne)

In der zweiten Iteration sollen die restlichen zwei Use-Cases (UC5) Verwenden von speziellen Knoten und (UC6) Verbinden von Knoten realisiert werden, wofür ebenfalls fünf Wochen eingeplant sind. In der ersten Woche soll festgelegt werden, wie die Domänenobjekte aufgebaut werden können. In den nachfolgenden zwei Wochen soll der bestehende Mindmap-Editor so erweitert werden, dass Mindmaps mit Domänenknoten modelliert werden können. Anschließend soll die entwickelte Erweiterung verwendet werden, um Objekte aus einer CRM-Domäne als *Knoten* verwenden zu können. In der letzten Woche sollen diese CRM-*Knoten* mit den Objekten verbunden werden, die sie repräsentieren und damit kontextsensitiv werden.

Tätigkeit	Geplant
Domänenobjekte konzipieren	1 Woche
Editorerweiterung für Verwendung von Domänenobjekten	2 Woche
Domänenobjekte identifizieren und implementieren	1 Woche
Verbindung der Domänenobjekte mit Kontext	1 Woche

Tabelle 3: Arbeitspakete der Iteration 2

4 Gemeinschaftliche Modellierung

Der erste Teil dieser Bachelor-Thesis besteht aus der Konzeptionierung und Entwicklung einer Lösung zur gemeinschaftlichen Modellierung in dem bereits in Unterabschnitt 3.1.3 Mindmap-Editor vorgestellten Mindmap-Editor. Dafür werden zunächst die Anforderungen konkretisiert.

4.1 Anforderungen

Die Anforderungen an ein Zusammenarbeit ermöglichendes System wurden in Unterabschnitt 3.2.1 Zusammenarbeit in Mindmaps bereits festgelegt. (K1) Online Zusammenarbeit, (K2) Konfliktbehandlung und (K3) Multiuserfähigkeit beziehen sich dabei auf das Verwalten von asynchronen Aktionen. Tom Gross und Michael Koch beschreiben dazu unter anderen die vier folgenden Punkte [Gro07].

- Awareness

Die Teilnehmer müssen über die Änderungen der anderen möglichst schnell informiert werden. Nur so kann das Gefühl echter Zusammenarbeit entstehen. Dies bezieht sich hauptsächlich auf die Anforderung (K1) Online Zusammenarbeit.

- Fehlertoleranz

Viele Teilnehmer müssen Änderungen durchführen können, ohne einen fehlerhaften Stand des Dokuments zu erzeugen. Damit bezieht sich dieser Punkt auf die Anforderungen (K1) Online Zusammenarbeit und (K3) Multiuserfähigkeit.

- Nebenläufigkeitskontrolle

Während die Teilnehmer das Dokument bearbeiten, müssen ihre Änderungen mit den Änderungen der anderen allgemeingültig bleiben. Kollisionen dürfen keinen inkonsistenten Zustand des Dokuments entstehen lassen. Dieser Punkt betrifft dabei Anforderung (K2) Konfliktbehandlung.

- Benutzbarkeit als Einzelbenutzereditor

Auch wenn kein anderer Teilnehmer Änderungen am Dokument durchführt, muss der Zustand der Mindmap so gespeichert werden, dass immer die aktuelle Version vorliegt und bei Bedarf sofort mit der gemeinschaftlichen Zusammenarbeit begonnen werden kann. Dies bezieht sich ebenfalls auf die Anforderung (K1) Online Zusammenarbeit.

4.1.1 Atomare Operationen

Um Zusammenarbeit zu ermöglichen, müssen die möglichen Aktivitäten der teilnehmenden Benutzer festgelegt werden. Diese werden im weiteren Verlauf atomare Operationen genannt. Um eine Mindmap im Team bearbeiten zu können, werden im Wesentlichen vier Operationen benötigt.

- *Knoten* einfügen
- *Knoten* bearbeiten
- *Knoten* verschieben
- *Knoten* entfernen

Da diese Operationen von unterschiedlichen Teilnehmern einer gemeinschaftlichen Sitzung unabhängig voneinander genutzt werden können, muss ihre Ausführung nebenläufig stattfinden

können. Um das zu gewährleisten, müssen sie auf ihr Konfliktpotenzial untersucht werden. Dabei können Konflikte im Kontext einer Mindmap auf zwei Arten auftreten, nämlich knoten- und hierarchiespezifisch. Knotenspezifische Konflikte betreffen einen bestimmten *Knoten*, der von zwei atomaren Operationen gleichzeitig benutzt wird. In Tabelle 4 werden Konfliktbehandlungen für diesen Fall spezifiziert. Operationen die zu keinem Konflikt führen erfordern keine Spezifikation, was durch ein Minus dargestellt ist.

	***Knoten* einfügen**	***Knoten* bearbeiten**	***Knoten* verschieben**	***Knoten* entfernen**
***Knoten* einfügen**	-	-	-	*Knoten* entfernen
***Knoten* bearbeiten**	-	Letzte Änderung	-	*Knoten* entfernen
***Knoten* verschieben**	-	-	Letzte Position	*Knoten* entfernen
***Knoten* entfernen**	*Knoten* entfernen	*Knoten* entfernen	*Knoten* entfernen	*Knoten* entfernen

Tabelle 4: Konfliktpotenzial von Aktionen an gleichem *Knoten*

Wenn ein *Knoten* entfernt werden soll, wird er einfach entfernt, unabhängig von den anderen Operationen. Konkurrierende Änderungen an *Knoten* werden durch die letzte Änderung abgeschlossen, ohne weitere Behandlung. Wenn ein *Knoten* eingefügt werden soll, dann kann es vorkommen, dass der übergeordnete *Knoten* zeitgleich gelöscht wird. Dies wäre ein hierarchiespezifischer Konflikt. In Tabelle 5 werden Konfliktbehandlungen bezüglich der Anwendung der atomaren Operationen auf über- und untergeordnete *Knoten* beschrieben.

(parent) (sub)	***Knoten* einfügen**	***Knoten* bearbeiten**	***Knoten* verschieben**	***Knoten* entfernen**
***Knoten* einfügen**	-	-	-	*Knoten* entfernen
***Knoten* bearbeiten**	-	-	-	*Knoten* entfernen
***Knoten* verschieben**	-	-	-	*Knoten* entfernen
***Knoten* entfernen**	-	-	-	*Knoten* entfernen

Tabelle 5: Konfliktpotenzial von Aktionen an Knotenhierarchie

Es können also nur Konflikte auftreten, wenn ein *Knoten* gelöscht wird und auf dessen Unterknoten andere atomare Operationen angewendet werden. In diesem Fall werden diese Operationen nicht ausgeführt, und der Oberknoten mit allen seinen Unterknoten entfernt. Alle anderen Situationen stellen im Kontext der Hierarchie der Mindmap keine Probleme dar und können gleichzeitig stattfinden.

Im Folgenden werden die atomaren Operationen näher beschrieben. Dabei wird davon ausgegangen, dass ein Teilnehmer die Operationen selbst ausführt. Wie die anderen Teilnehmer von den Änderungen erfahren, wird im Unterabschnitt 4.1.2 Globaler Zustand beschrieben.

4.1.1.1 Knoten einfügen

Ein *Knoten* kann nur als Unterknoten eines anderen *Knotens* eingefügt werden. Dabei müssen *Knoten* für alle Teilnehmer der Zusammenarbeit eindeutig identifizierbar sein. Folglich spielt die

Knotenidentität eine entscheidende Rolle und wird im Weiteren mit *nodeId* angegeben. Es ergibt sich folgende Signatur der Einfügen-Operation:

addNode :: (parentNodeId, newNodeId, nodePosition) -> ()

Dabei gibt der erste Parameter den *Knoten* an, der den übergeordneten *Knoten* des neuen *Knotens* darstellt. Der zweite Parameter ist die gewünschte Identität des neuen Knoten. Die Position des neuen *Knotens* wird als drittes angegeben. Es wird kein Rückgabewert benötigt.

4.1.1.2 Knoten bearbeiten

Ein *Knoten* kann zunächst nur atomar bearbeitet werden. Das bedeutet, dass seine Daten nur von einem Nutzer zu einem Zeitpunkt erfolgreich bearbeitet werden können. Mit Knotendaten sind alle Inhalte gemeint, die die eine Knoteninstanz ausmachen, abgesehen von der Position. Das führt zu folgender Signatur:

editNode :: (nodeId, nodeData) -> ()

Die Operation gibt keinen Wert zurück.

4.1.1.3 Knoten verschieben

Während die Daten eines *Knotens* mit einer eigenen atomaren Operation bearbeitet werden können, kann ein Positionswechsel unabhängig von der Bearbeitung durchgeführt werden. Dafür besitzt die Operation folgende Signatur:

moveNode :: (nodeId, nodePosition) -> ()

Während der erste Parameter wieder den entsprechenden *Knoten* identifiziert, enthält der zweite Parameter die Position nach dem Verschieben. Dabei gibt die Position die Koordinaten auf der Dokumentfläche an und nicht die topologische Position, wie zum Beispiel die Identität des übergeordneten *Knotens*. Eine relative Positionsangabe würde eine Berechnung der neuen Position erfordern, die die aktuelle Position auf der Zeichenfläche lesen und dann schreiben müsste. Um Konflikte bei nebenläufiger Ausführung zu vermeiden, müssten Mechanismen zur Synchronisierung verwendet werden. Bei der verwendeten absoluten Positionierung wird nur einmal geschrieben, was keine Synchronisierung benötigt.

4.1.1.4 Knoten entfernen

Wenn ein *Knoten* entfernt werden soll, dann wird nur dessen Identität benötigt. Wenn der ausgewählte *Knoten* gelöscht wird, müssen alle *Knoten*, die ihn als übergeordneten *Knoten* haben ebenfalls rekursiv gelöscht werden. Dies führt zu unterer Signatur:

removeNode :: (nodeId) -> ()

Ein Rückgabewert ist nicht erforderlich.

4.1.2 Globaler Zustand

Nachdem in vorherigen Unterabschnitt die atomaren Operationen vorgestellt wurden, die die Benutzer ausführen können um ihre Änderungen zu propagieren, wird in diesem Unterabschnitt beschrieben, wie diese Änderungen an sämtliche Teilnehmer weitergeleitet werden. Dabei muss sichergestellt werden, dass jeder Teilnehmer mit den zur Verfügung gestellten Informationen den gleichen globalen Zustand erzeugen kann. Nur dann kann die Konsistenz des Dokuments bewahrt

werden. Es wird also eine koordinierende Instanz benötigt, die sämtliche atomare Operation aggregiert und den einzelnen Teilnehmern der Zusammenarbeit zukommen lässt.

Die Teilnehmer müssen in regelmäßigen Abständen den globalen Zustand abfragen, um ihr Dokument aktuell zu halten. Ansonsten würden viele Kollisionen auftreten, deren späte Behandlung eventuell ein unerwünschtes Ergebnis darstellt. Dank des globalen Zustands können Benutzer ihre eigenen Transaktionen verwalten. Da die atomaren Operationen eines Teilnehmers den globalen Zustand verändern und dann an alle weitergegeben werden, kann der Verursacher auf die entsprechende Zustandsänderung warten. Erst wenn diese eintritt, gilt der Vorgang, der mit der atomaren Operation durchgeführt wurde, als abgeschlossen. Ist dies nicht der Fall, dann ist die Änderung gegenstandlos.

4.1.3 Zusammenfassung

Nachdem in diesem Abschnitt erörtert wurde, wie Zusammenarbeit ermöglicht werden kann, werden die daraus resultierten Anforderungen zusammengefasst. Dabei werden die High-Level Anforderungen aus Unterabschnitt 3.2.1 Zusammenarbeit in Mindmaps mit konkreten Anforderungen ergänzt.

(K6) Atomare Operationen ausführen

Der Kern der Zusammenarbeit besteht aus den atomaren Operationen, die in Unterabschnitt 4.1.1 Atomare Operationen vorgestellt wurden. Operationen für das Hinzufügen, Editieren, Verschieben und Entfernen von *Knoten* müssen aufrufbar sein.

(K7) Atomare Operationen aggregieren

Damit die atomaren Operationen auch einen Effekt haben können, wird ein globaler Zustand benötigt, wie in Unterabschnitt 4.1.2 Globaler Zustand beschrieben, der sich durch die atomaren Operationen transformieren lässt. Ausgeführte Operationen müssen vom Server als Sequenz aufgefasst und zusammengeführt werden.

(K8) Atomare Operationen anwenden

Die Sequenz der Zustandsänderungen kann dann verwendet werden um einen konsistenten Zustand auf allen Clients zu erzeugen. Zusammengefasste atomare Operationen müssen den eigenen Zustand transformieren können.

Die aktuellen Anforderungen sind in Tabelle 6 zusammen mit Priorität und betreffendem System aufgelistet, wobei A die höchste Priorität darstellt. Das betreffende System ist entweder die Service-Infrastruktur oder der Client.

ID #	Anforderung	Betreffendes System	Priorität
K1	Online Zusammenarbeit	WebService	A
K2	Konfliktbehandlung	WebService	B
K3	Multiuserfähigkeit	WebService	B
K4	Dokumente auf Server speichern	WebService	C
K5	Rechtesystem	WebService	C
K6	Atomare Operationen ausführen	Client	A
K7	Atomare Operationen aggregieren	WebService	A
K8	Atomare Operationen anwenden	Client	A

Tabelle 6: Anforderungen an Zusammenarbeit in *MindGraph*

4.2 Entwurf

Nach dem alle Anforderungen an gemeinschaftliche Verwendung des Mindmap-Editors vorgestellt wurden, wird in diesem Abschnitt auf die konkrete Architektur der Infrastruktur eingegangen. Wie in Unterabschnitt 3.2.3 Use-Cases beschrieben, sollen vier Anwendungsfälle in diesem Abschnitt behandelt werden. Diese lassen sich in einen gemeinschaftlichen ((UC4) Zusammenarbeiten) und einen administrativen ((UC1) Sitzung verwalten, (UC2) Sitzung beitreten und (UC3) Neue Teilnehmer ermitteln) Bereich unterteilen. Um die bereits theoretisch beschriebene Zusammenarbeit zu ermöglichen, muss es eine Sitzung geben, an der sich die Nutzer beteiligen können. Das Verwalten dieser Sitzungen wird von einem administrativen Teilnehmer, dem Autor, übernommen. Im Folgenden wird zunächst die Struktur des administrativen und anschließend des gemeinschaftlichen Bereichs vorgestellt. Dabei werden die Aktionen der Rollen näher beschrieben, die zusammen mit den Use-Cases in Unterabschnitt 3.2.3 Use-Cases vorgestellt wurden.

4.2.1 Administrative Serverseite

Die administrative Seite besitzt einen eigenen Serverdienst, der das Verwalten von Sitzungen und Benutzern unterstützt. Über ein Sitzungsverwalter-Objekt lassen sich die Sitzungen mit einer eindeutigen ID erzeugen. Anschließend können Teilnehmer sich an der Sitzung anmelden. Das Klassendiagramm in Abbildung 11 zeigt die Struktur des Dienstmodells.

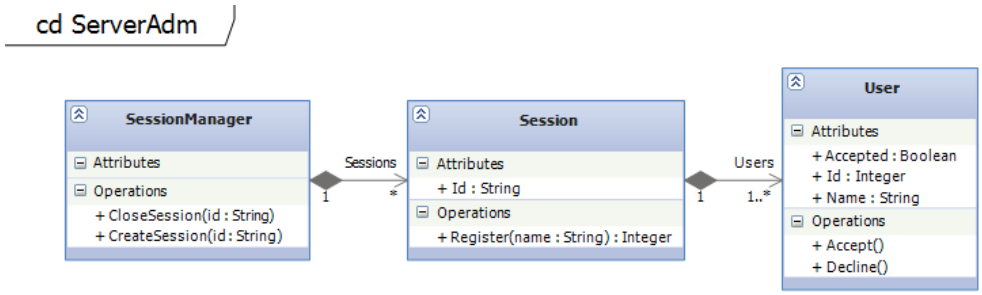


Abbildung 11: Administrative Klassenstruktur auf der Serverseite

Um das Modell zu manipulieren sind Dienstoperationen nötig. Beim Starten des Servers wird der `SessionManager` erzeugt. Anschließend lässt sich der Dienst verwenden. Diese Verwendung wird in Bezug auf die drei Use-Cases beschrieben.

4.2.1.1 Zusammenarbeit verwalten

In diesem Unter-Unterabschnitt wird die Planung des Use-Case (UC1) Sitzung verwalten beschrieben. Die Operationen, die die Sitzungen verwalten, gehören zu den wichtigsten Operationen des Diensts. Über einen Aufruf von `CreateSession` auf dem Server lässt sich ein Sitzungsobjekt anhand einer eindeutigen `Session`-ID erzeugen. Diese `Session`-ID wird im Falle der erfolgreichen Erzeugung als Rückgabewert zurückgeliefert. Sollte der Server nicht funktionieren oder einen Fehler verursacht haben, kann der Aufrufer so über das Problem informiert werden. `Session`-IDs sind als Zeichenketten codiert. Das Sequenzdiagramm in Abbildung 12 zeigt, wie der Autor beim administrativen Dienst die entsprechenden Methoden aufruft.

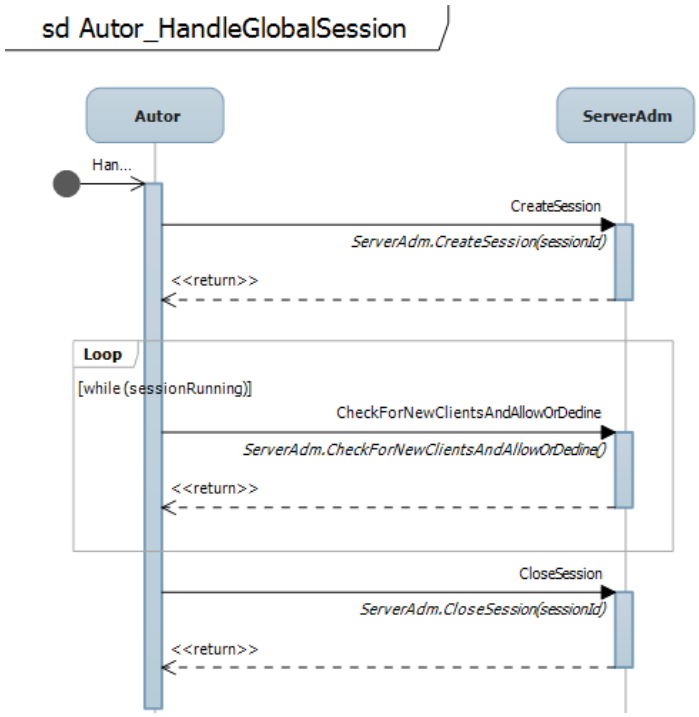


Abbildung 12: Dienstaufrufe zum Verwalten von Sitzungen

Eine erzeugte Sitzung kann über die Sitzungskennung jederzeit geschlossen werden. An einer offenen Sitzung können sich Teilnehmer anmelden, was im Unter-Unterabschnitt 4.2.1.2 Zusammenarbeit beitreten beschrieben wird. Angemeldete Benutzer können von jeder Sitzung abgefragt werden. Obwohl diese Operationen von jedem Teilnehmer aufgerufen werden können, wird die Rolle des Autors als verantwortlich für die Sitzungsverwaltung angesehen. Die Operation `CheckForNewClientsAndAllowOrDecline` ist ein Platzhalter für sämtliche Operationen des Use-Case (UC3) Neue Teilnehmer ermitteln und wird in Unter-Unterabschnitt 4.2.1.3 Teilnehmer der Zusammenarbeit verwalten beschrieben.

4.2.1.2 Zusammenarbeit beitreten

In diesem Unter-Unterabschnitt wird die Planung des Use-Case (UC2) Sitzung beitreten beschrieben. Wenn einem Teilnehmer eine Sitzungs-ID bekannt ist, kann er sich bei dieser Sitzung anmelden. Er erhält seine Nutzerkennung zurück und kann somit von einer erfolgreichen Registrierung ausgehen. Um unerwünschte Nutzer zu behandeln, gelten angemeldete Teilnehmer nicht automatisch als akzeptiert. Sie müssen erst vom Autor der Sitzung aktiviert oder bestätigt werden. Dieser Vorgang wird in Unter-Unterabschnitt 4.2.1.3 Teilnehmer der Zusammenarbeit verwalten beschrieben. Der neue Teilnehmer muss also in regelmäßigen Abständen abfragen, ob er aktiviert wurde. Dazu ruft der die Operation `IsOnline` mit seiner Nutzerkennung auf und erhält eine Antwort. Die Nutzerkennung ist eine sitzungsunabhängige Ganzzahl, die den `User` eindeutig identifiziert. Diese Antwort kann interpretiert und so der Status des Teilnehmers ermittelt werden. Sobald die Aktivierung vorhanden ist, kann der aktuelle Sitzungsstatus aus der erhaltenen Antwort extrahiert werden. Ab diesem Zeitpunkt gilt der Benutzer als vollwertiger Sitzungsteilnehmer. Das Geheimnis der Sitzung, also der globale Sitzungsstatus, kann dann vom Benutzer als anfänglicher Zustand verwendet werden. Abbildung 13 zeigt die Interaktion des neuen Teilnehmers mit dem administrativen Serverdienst in einem Sequenzdiagramm.

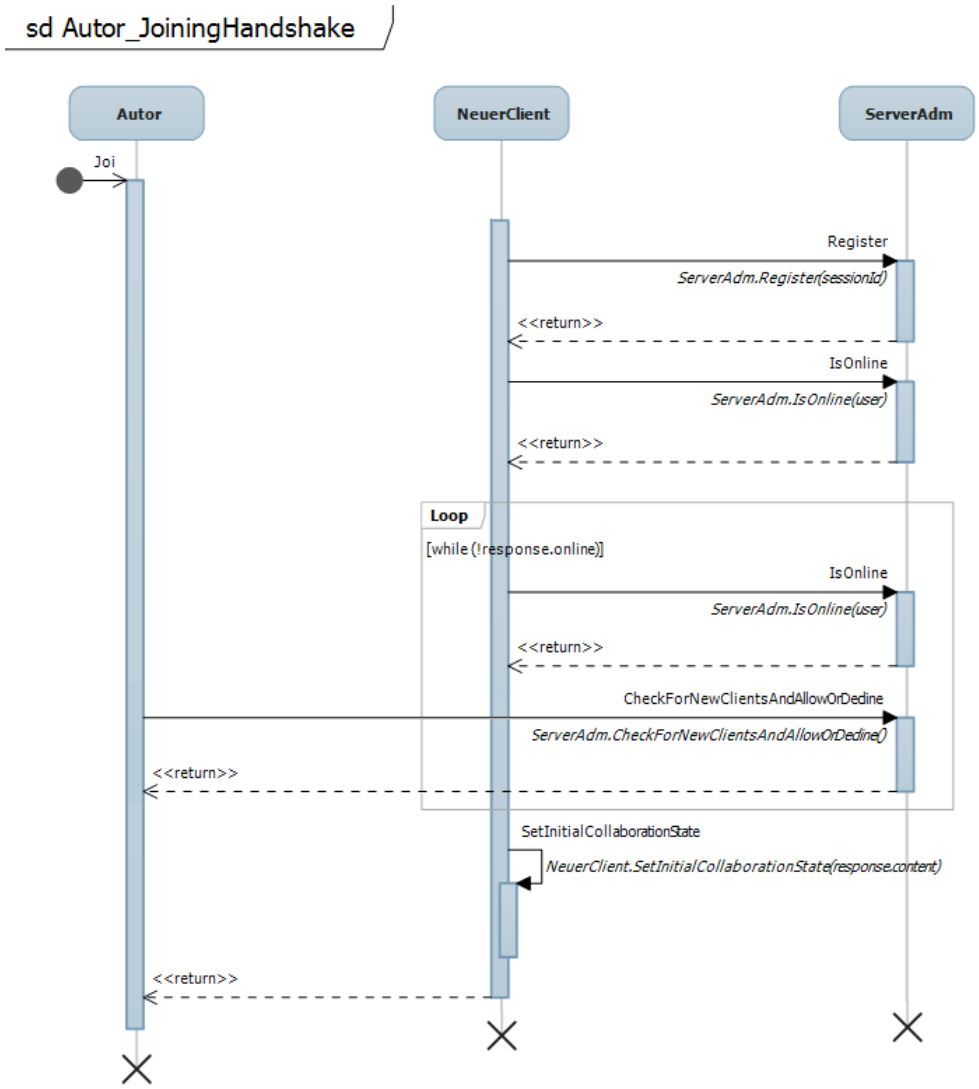


Abbildung 13: Dienstaufrufe zum Beitreten einer Sitzung

Wie das Sitzungsgeheimnis aussieht wird von der Bestätigung des Sitzungsautors festgelegt. Wird der Teilnehmer nicht aktiviert, kann er entweder für unbestimmte Zeit in seiner Abfrageschleife verweilen oder nach einer bestimmten Zeit eine neue Anmeldung vornehmen. Eine serverseitige Frist für die Aktivierung gibt es nicht. Die Verantwortung der Benutzerverwaltung liegt beim Autor der entsprechenden Sitzung.

4.2.1.3 Teilnehmer der Zusammenarbeit verwalten

In diesem Unter-Unterabschnitt wird die Planung des Use-Case (UC3) Neue Teilnehmer ermitteln beschrieben. Existiert eine Sitzung an der sich mehrere Teilnehmer angemeldet haben, ist der Autor der Sitzung dafür zuständig, diese Nutzer zuzulassen oder abzulehnen. Dafür kann er zu jedem Zeitpunkt die `GetAllClients`-Operation mit seiner `Session`-ID aufrufen. Er erhält eine Liste mit allen angemeldeten Benutzern zurück. In dieser Liste können die neuen Nutzer vom Autor identifiziert werden und entsprechend behandelt werden. Für die Aktivierung der Teilnehmer kann die Operation `AcceptClient` aufgerufen werden. Dazu muss die aktuelle Sitzungs-ID, sowie die entsprechende Nutzerkennung sowie das Sitzungsgeheimnis als Parameter übergeben werden. Abbildung 14 zeigt die Möglichkeiten der Benutzerbehandlung durch `Autor` in einem Sequenzdiagramm.

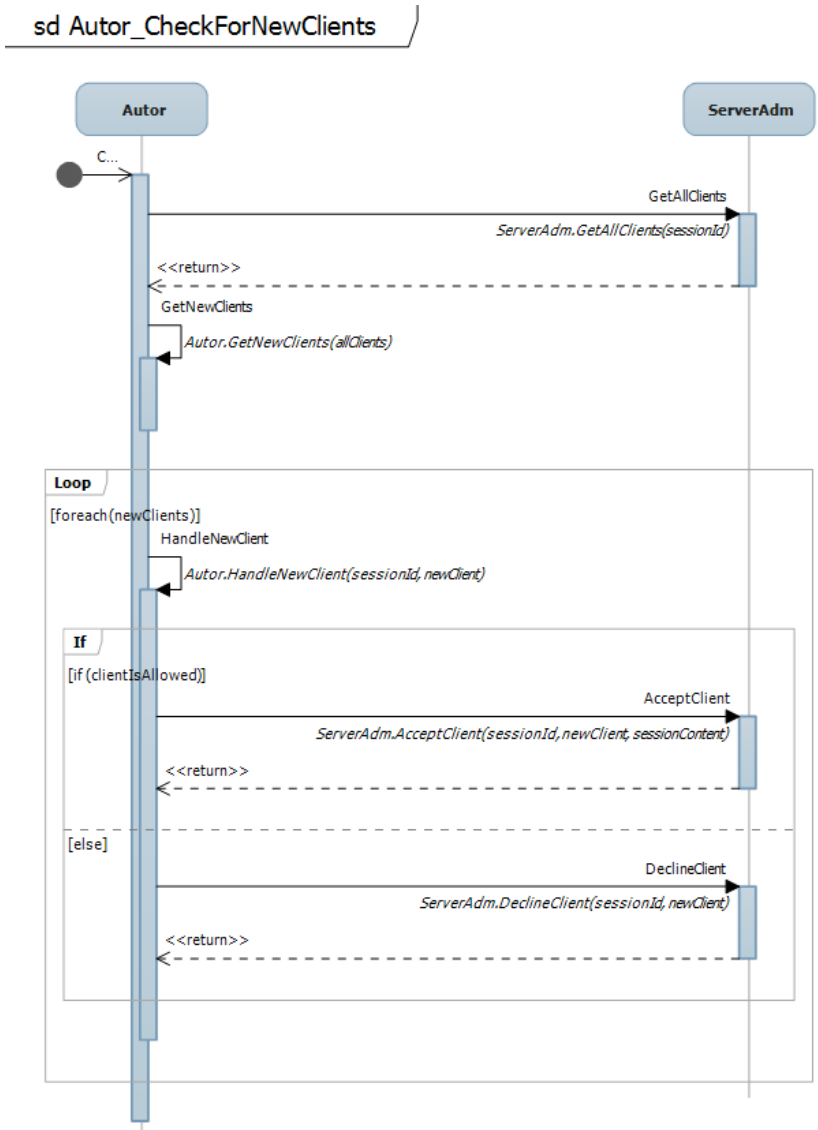


Abbildung 14: Dienstaufrufe zum Verwalten von Teilnehmern einer Sitzung

Um einen Benutzer abzulehnen kann die Operation `DeclineClient` aufgerufen werden. An dieser Stelle ist es nur notwendig die Sitzungs- und die Nutzerkennung zu übergeben.

Authentifizierung

Da alle diese Operationen als Serveroperationen ohne Zustand aufgerufen werden können, muss der Kontext jedes Mal als Parameter übergeben werden. Dadurch könnten theoretisch alle möglichen Dienstnehmer Teilnehmer aktivieren, sogar die Teilnehmer selbst. Um dieses Verhalten zu verhindern, müsste bei der Erstellung einer Sitzung dem Autor ein geheimer Schlüssel mitgeteilt werden, um den Autor am Service authentifizieren zu können. Nur wenn `AcceptClient` und `DeclineClient` mit dem korrekten Schlüssel aufgerufen werden, sind Operationen gültig und werden ausgeführt. Dieser Sicherheitsaspekt wird vorerst aber nicht berücksichtigt, um die Architektur und dessen Implementierung nicht unnötig zu erschweren. Die Autoren-Authentifizierung und Rechteverwaltung können aber nachträglich integriert werden. Diese Entscheidung ist dank der niedrigen Priorität der Anforderung (K5) Rechtesystem möglich.

4.2.2 Zusammenarbeit ermöglichende Serverseite

Nachdem die administrativen Aspekte der Serviceseite vorgestellt wurden, wird in diesem Unterabschnitt die gemeinschaftliche Sicht und damit der Use-Case (UC4) Zusammenarbeiten betrachtet. Gemäß der Anforderung (K7) Atomare Operationen aggregieren existiert eine Eventsequenz, die vom `StateManager` verwaltet wird. Die in Anforderung (K6) Atomare Operationen ausführen definierten Operationen können auf dem Zustandsverwalter aufgerufen werden. Dieser erzeugt daraufhin Zustandsänderungen, welche als Objekte in die Sequenz eingefügt werden.

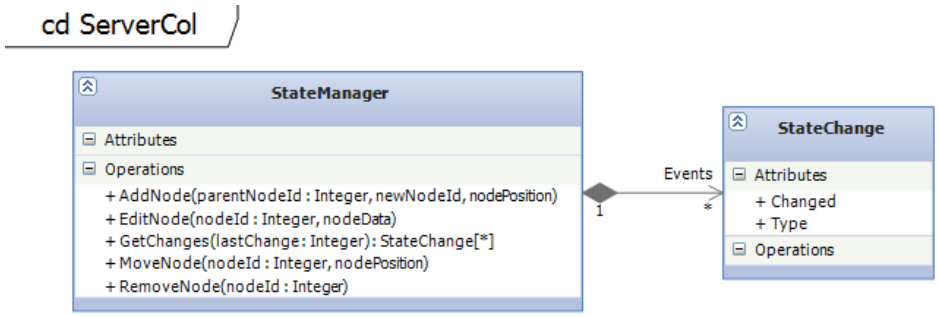


Abbildung 15: Zusammenarbeit ermöglichende Klassenstruktur auf der Serverseite

Das Objektmodell des Dienstes besteht, wie in Abbildung 15 gezeigt, aus einer einfachen Elementstruktur. Die Aggregation wird durch die `GetChanges`-Methode des `StateManagers` abgefragt, welcher die Rückgabe erzeugt. Diese einfache Elementstruktur könnte problematisch werden, wenn die Sitzung viele *Events* speichert. Da in der Regel immer nur die letzten abgefragt werden, wird der Dienst mit der Zeit immer unperformanter, da aus der gesamten Liste immer nur die zuletzt hinzugefügten Elemente abgefragt werden. Um schlechter Performanz auf dem Server entgegenzuwirken, müsste die Struktur die letzten `StateChange`-Objekte immer vorhalten, um diese jederzeit performant zurückgeben zu können. Es wird also ein geeignetes Caching benötigt. Die Cachegröße müsste abhängig von der Anzahl der neuen *Events* gewählt werden. Wenn ein `GetChanges`-Aufruf mehr *Events* abfragen will als der Cache enthält, wird erst dann auf die vollständige Liste verwiesen. Da für den Prototyp zunächst nur von kurzer und nicht sehr aktiver Zusammenarbeit ausgegangen wird, wird Caching vorerst vernachlässigt.

Die Aufrufe der Clients an den Server entsprechen weitestgehend den Methoden des `StateManagers`. Das Sequenzdiagramm in Abbildung 16 zeigt die vorhandenen Operationen zusammen mit ihrer Signatur. `GetActions` entspricht dabei der `GetChanges`-Methode des Servicemodells. `NodeId`-Werte sind als 32-Bit Ganzzahl und `NodePositions` als Gleitkommzahl-Zweiertupel codiert. `NodeData` ist abhängig von der Art des *Knotens*. Da im zweiten Teil dieser Thesis verschiedene Knotentypen eingeführt werden, wird der Typ vorerst auf `object` festgelegt und kann so alle bekannten Datentypen, außer Arrays, beinhalten.

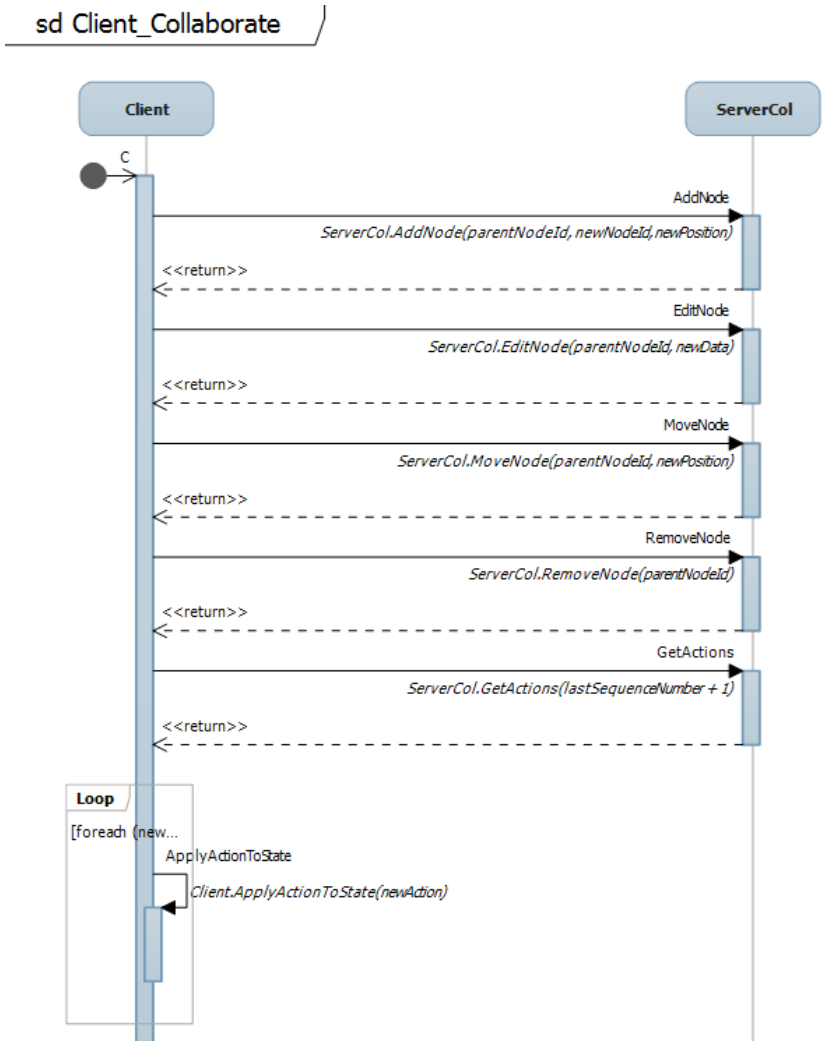


Abbildung 16: Dienstaufrufe für gemeinschaftliche Aktionen in Mindmaps

Der Aufruf der `GetActions`-Methode auf dem Server liefert sämtliche *Ereignisse*, die seit der letzten Aktualisierung eingetreten sind. Diese werden dann vereinzelt auf den aktuellen Status angewendet (`ApplyActionToState`), sodass der globale Zustand auf allen Clients erzeugt werden kann. Aus Sicht des Clients gibt es also nur zwei Ereignisströme, nämlich den der lokalen und den der globalen *Events*. Nur diese beiden müssen synchronisiert werden, egal wie viele Clients am System angemeldet sind. Dabei sind zwei Fälle zu beachten.

Bei der Synchronisierung der lokalen mit den globalen *Events*, die zuvor selbst lokal ausgelöst wurden, handelt es sich um eine Transaktion. Ausgelöst durch das eigene lokale *Event*, entspricht das gleiche *Event* im globalen Ereignisstrom dem eigentlichen Aktionsende. Erst bei Aktionsende, ist die Transaktion als vollständig oder abgeschlossen zu betrachten. Bei der Synchronisierung der lokalen mit den globalen *Events*, die nicht selbst ausgelöst wurden, handelt es sich um atomare Aktionen die angewendet werden können. Dieses Anwenden darf dann aber keine lokalen *Ereignisse* auslösen, da diese sonst kontinuierlich im globalen Strom auftreten würden.

4.2.3 Integration

Es wurden die administrative und die Zusammenarbeit ermöglichende Serverseite vorgestellt. Diese beiden Diensterbringer müssen kombiniert werden, um ein integriertes System zu erzeugen. Es wurde bereits erwähnt, dass atomare Operation nur im Kontext einer Sitzung ausgeführt werden

können. Das Gleiche gilt für das Ermitteln des globalen Zustands. Operationen müssen zusätzlich von einem Benutzer ausgeführt werden. Da beide Dienste alle Operationen allerdings zustandslos zur Verfügung stellen, muss der Kontext beim Aufruf mit übergeben werden. Das bedeutet, dass alle Serviceoperationen, die den atomaren Operationen entsprechen, zwei zusätzliche Parameter erhalten, nämlich eine `Session`-ID und eine Nutzerkennung. `GetChanges` erhält ebenfalls eine `Session`-ID als weiteres Argument. Außerdem muss der `SessionManager` im Dienst `ServerCol` eine `Hashtable` verwenden, die die Eventsequenzen den Sitzungen zuordnet.

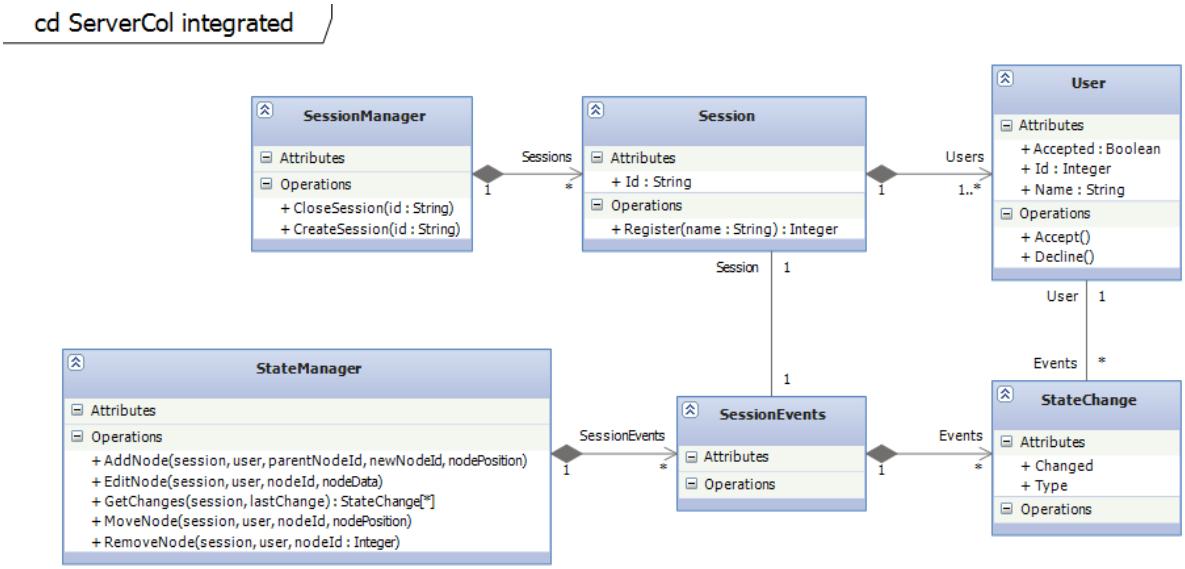


Abbildung 17: Integrierte Klassenstruktur auf der Serverseite

Diese Zuordnung wird durch ein `SessionEvents`-Objekt repräsentiert. Das konkrete *Event* erhält als `StateChange`-Objekt einen Verweis auf den `User`, der es ausgeführt hat. Als kombinierte Struktur des Dienstes ergibt sich das Klassendiagramm in Abbildung 17.

4.2.4 Clientseite

Die Serverseite mit den beiden beschrieben Diensten ist damit vollständig beschrieben. Diese Dienste müssen aus dem Mindmap-Editor automatisch aufgerufen werden, wenn der Nutzer eine Aktion durchführt. Zusätzlich muss in regelmäßigen zeitlichen Abständen der globale Zustand abgefragt und erhaltene Änderungen in das Dokument integriert werden. Eine Erweiterung des bisherigen Editors ist somit erforderlich. Eine Möglichkeit wäre, die Aufrufe der Operationen direkt in die bisherige Struktur zu integrieren. Dadurch würde der bestehende Code eine Menge neuer Abhängigkeiten bekommen, die zu einer schwereren Wartbarkeit führen würden.

Eine andere Möglichkeit besteht in der Einführung einer speziellen Struktur, die es ermöglicht, auf *Events* aus der inneren Struktur zu reagieren. Mit dieser Interception-Struktur besteht die Möglichkeit, Überwacher zu definieren, die die Veränderungen in der Mindmap beobachten können. Die Klasse, die die aktuelle Mindmap-Topology verwaltet, kann dafür eine Schnittstelle auf der Add, Edit, Move und Remove mit entsprechenden Parametern aufgerufen werden kann, verwenden. Wenn in der Struktur solche Operationen auf die Mindmap angewendet werden, wird das entsprechende *Ereignis* ausgelöst. Auf diese Weise können beliebige und selbst definierbare Komponenten genutzt werden, um auf das Benutzerverhalten systematisch und anpassbar zu reagieren. Für die gemeinschaftliche Seite können diese *Events* an den Service-Proxy der Zusammenarbeit ermöglichenden Serverseite weitergeleitet werden.

Nachdem die internen Operationen jetzt propagiert werden können, müssen sie auf den Seiten der anderen Clients angewendet werden. Dazu muss ebenfalls eine Möglichkeit geschaffen werden, die Mindmap ohne Nutzereingaben zu verändern. Dieser Vorgang muss automatisch stattfinden können. Es wird also das Gegenteil eines Interceptors benötigt. Die Lösung ist ein Erzeuger, der genau wie ein Beobachter von der strukturverwaltenden Einheit des Dokuments verwaltet wird. Dieser Erzeuger benötigt eine Referenz auf die aktuelle Struktur, um diese dann verändern zu können. Dazu kann sich die Struktur beim Erzeuger registrieren. Anschließend können auf dem Erzeuger die Add-, Edit-, Move- und Remove-Operationen aufgerufen und dann automatisch in der Mindmap ausgeführt werden.

4.2.4.1 Implizite und explizite Events

Es gilt zu klären, bei welchen Operationen ein *Event* an die Interception-Struktur signalisiert werden soll. Angenommen es gibt einen *Knoten*, der einen Unterknoten hat. Durch eine Verschiebung des Oberknotens werden beide *Knoten* auf der Oberfläche verschoben. Wenn zwei *Events* entstehen, wird das Verhalten im Rahmen dieser Ausarbeitung als rekursiv oder explizit bezeichnet. Bei nur einem *Event* handelt es sich entsprechend um implizites Verhalten. In Abbildung 18 wird ein einfaches Move-*Event* von zwei Clients gleichzeitig ausgeführt. Obwohl die Move-Operationen nach 4.1.1 Atomare Operationen eine sehr deutliche Konfliktbehandlung haben, bleibt folgender Konflikt versteckt und führt zu unterschiedlichen Darstellungen auf beiden Seiten.

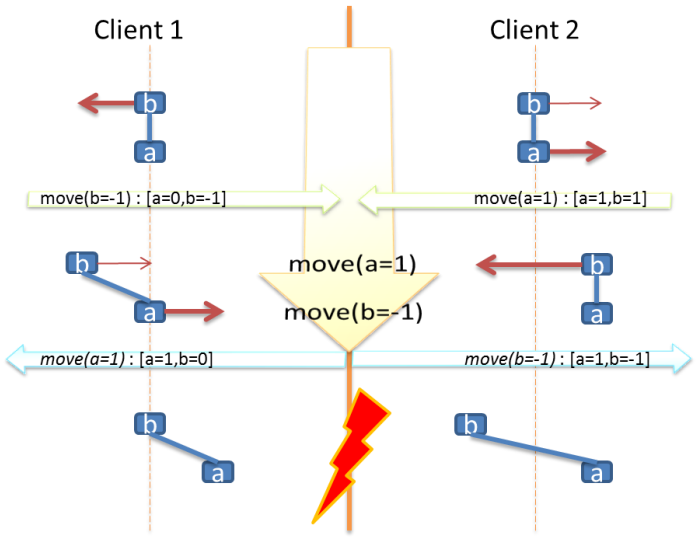


Abbildung 18: Konfliktgefahr bei impliziten *Ereignissen*

Das Problem ist, dass bei der Verschiebung von *Knoten* a, auch der *Knoten* b verschoben wird, aber dies ohne ein explizites *Event*. Ansonsten würde der Fall verlaufen, wie in Abbildung 19 dargestellt. Wenn der *Knoten* a verschoben wird, werden zwei *Events* ausgelöst, nämlich eines für a und eines für b. Daher kann man diese Eventerzeugung aus Sicht des Oberknotens auch als rekursiv bezeichnen. Gleichzeitig wird von der anderen Seite auch ein Move für b vorgenommen. Der Dienst für Zusammenarbeit bildet die Sequenz und die Teilnehmer erhalten zwei *Ereignisse* für b. Wie in den Anforderungen an die Operationen in 4.1.1 Atomare Operationen definiert, wird ein Konflikt bei der Move-Operation (b=1 und b=-1) so behandelt, dass der letzte Vorgang gewinnt. In diesem Fall also b=1. Bei einer anderen Reihenfolge wäre das entsprechend anders. Der Konflikt kann also von jedem Client behandelt werden und führt zu einem konsistenten Zustand der Mindmap-Struktur, da die Mindmap bei allen Clients weiterhin gleich dargestellt wird.

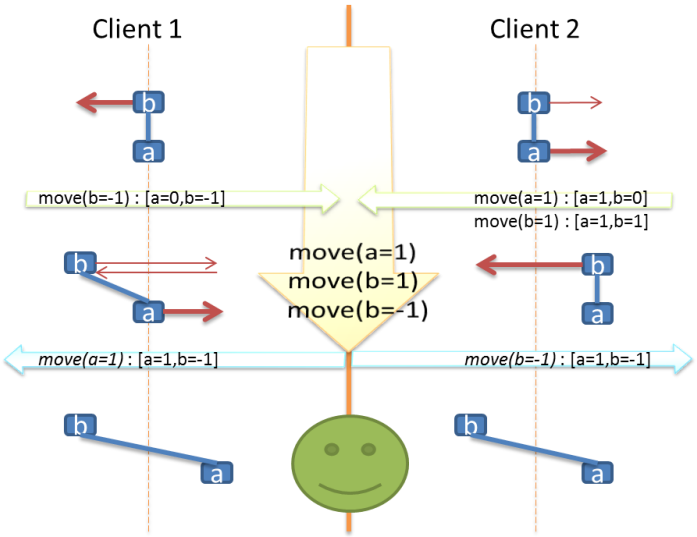


Abbildung 19: Konfliktfreiheit bei expliziten *Ereignissen*

Es ist also wichtig, dass Move-*Events* auf jeden Fall rekursiv auftreten müssen. Sobald sich die Position eines *Knotens* verändert und dies kein *Ereignis* auslöst, kann die Darstellung bei den einzelnen Clients unterschiedlich werden. Es muss auch das Anwenden eines Move-*Events* auf die eigene Struktur so erfolgen, dass nur der *Knoten* verschoben wird, auf den sich das *Event* bezieht. Das Verschieben eines *Knotens* durch einen Benutzer, verschiebt dagegen auch die Unterknoten, erzeugt aber auch die nötigen *Ereignisse*. Das Gleiche gilt auch für die Add- und Edit-Operation, da diese ebenfalls die Positionierung von *Knoten* übernehmen können. Wenn ein *Knoten* durch den Nutzer editiert wird, kann sich dadurch die Größe des *Knoten* ändern. Diese Änderung verschiebt ebenfalls sämtliche Unterknoten. Auch dort darf keine implizite Verschiebung von anderen *Knoten* verursacht werden. Auf der anderen Seite dürfen die *Events* auch erst erzeugt werden, wenn der Verschiebe- oder Editiervorgang abgeschlossen ist. Anderenfalls würden viele *Events* ausgelöst, die andere *Events* einfach überschreiben würden. Die Remove-Operation ist von diesem Problem nicht betroffen. Da die Clients aber alle Knotenkennungen verwalten müssen (siehe 4.2.4.4 Eindeutige Knotenidentität), müssen auch Remove-*Events* rekursiv erzeugt werden. Ansonsten müsste die ID-Verwaltung zusätzliches Wissen über die Baumstruktur haben, was zu einer sehr komplexen Kennungsverwaltung führen würde. Was aus dem oberen Beispiel auch hervorgeht ist, dass eigene *Events* auf keinen Fall leichtfertig ignoriert werden dürfen. Angenommen Client 1 hätte sein b=-1 *Ereignis* ignoriert, dann würde b jetzt wieder an eine andere Stelle verschoben und es gäbe erneut einen versteckten Konflikt.

4.2.4.2 Events ignorieren

Es muss also geklärt werden, wann *Ereignisse* nicht beachtet werden müssen und ob ein solches Verhalten überhaupt sinnvoll ist. Das *Events* nicht leichtfertig ignoriert werden dürfen, wurde bereits dargestellt. Wenn *Events* generell nicht ignoriert werden, dann kann die flüssige Darstellung der Nutzerinteraktion beeinträchtigt sein. Angenommen, ein Teilnehmer verschiebt einen *Knoten* a zweimal: a=1 und a=2. Diese Verschiebungen erfolgen so schnell hinter einander, dass sie einzeln nicht transaktional vollständig ausgeführt werden. Der Begriff der Transaktion wurde im Unterabschnitt 4.2.2 Zusammenarbeit ermöglichende Serverseite vorgestellt. Anschließend treten die globalen *Ereignisse* ein und werden behandelt. Dadurch wird a, momentan an Position a=2, zunächst zurück zu a=1 verschoben, bevor a endgültig an a=2 verschoben wird. Dadurch entsteht ein wahrnehmbares Springen des *Knotens*. Um dieses Verhalten zu vermieden, muss sichergestellt

werden, dass *Events* ignoriert werden können, die sich nicht auf *Knoten* beziehen, die zwischenzeitlich von anderen Nutzern geändert wurden. Es ergeben sich also zwei Bedingungen, die beide erfüllt sein müssen, um das aktuelle *Event* ignorieren zu dürfen.

- Nur wenn ein *Ereignis* durch einen Nutzer mit der Kennung ausgelöst wurde, die der Kennung des empfangenden Teilnehmers entspricht.
- Nur wenn der *Knoten*, auf den sich das empfangene *Ereignis* bezieht, zuletzt vom empfangenden Nutzer geändert wurde.

Ein beispielhaftes Szenario ist in Abbildung 20 abgebildet. Client 1 kann beide Move-*Events* ignorieren, da diese beide Bedingungen erfüllen. Die Nutzerkennung ist den *Events* schematisch angehängt (nach dem Schrägstrich).

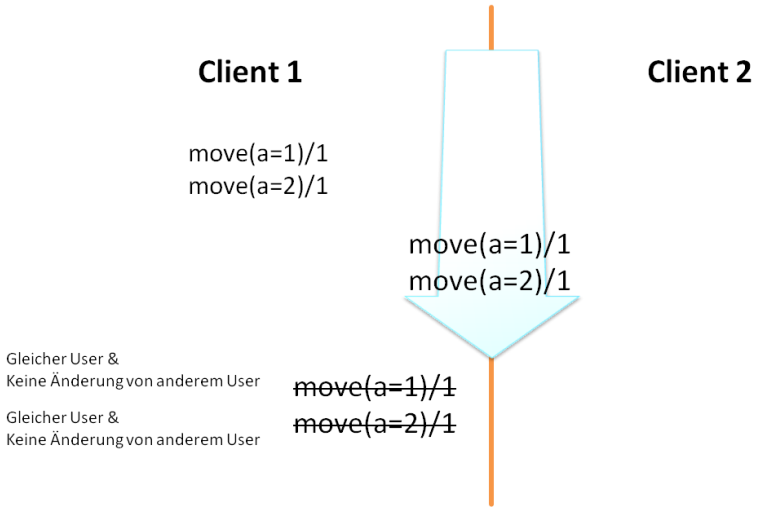


Abbildung 20: *Ereignisse* können ignoriert werden

Ein Beispiel, in dem Client 1 nicht beide *Events* ignorieren kann, ist in Abbildung 21 dargestellt. Lediglich das erste *Ereignis* muss nicht behandelt werden. Es ist ersichtlich, dass der konsistente Zustand trotzdem aufrecht erhalten werden kann.

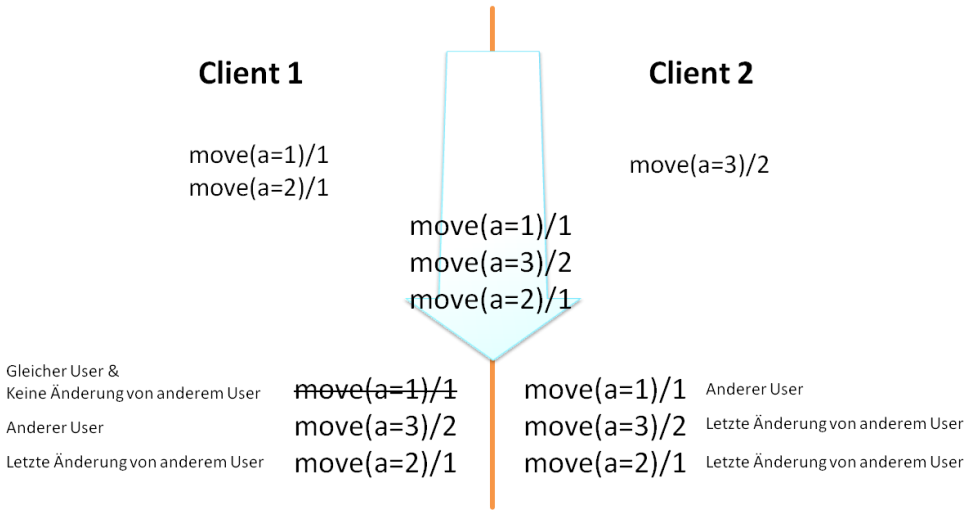


Abbildung 21: *Ereignisse* können nicht immer ignoriert werden

Für das Vermeiden von *Ereignissen* ist es erforderlich, dass jedes *Ereignis* die Nutzerkennung enthält, die den erzeugenden Teilnehmer identifiziert. In Unterabschnitt 4.2.3 Integration ist die

unterstützende Servicestruktur dargestellt. Da sich die Frage nach der Ignorierbarkeit nur auf Basis einzelner *Knoten* stellt, ist es wichtig, dass *Knoten* global eindeutig referenziert werden können. Das Kennungsmanagement von *Knoten* wird in 4.2.4.4 Eindeutige Knotenidentität behandelt.

4.2.4.3 Gleichzeitige Erzeugung

Wenn Teilnehmer einer Zusammenarbeit an verschiedenen Orten an einem zentralen Dokument arbeiten, führt dies zwangsläufig zu Konflikten. In Unterabschnitt 4.1.1 Atomare Operationen wurde dieses Konfliktpotenzial bereits untersucht, um die Atomarität sicherzustellen. Dort wurde das Auftreten von zwei Add-Operationen als unproblematisch dargestellt. Diese Problemlosigkeit bezieht sich aber nur auf das Ergebnis der Add-Operation, nicht auf dessen Seiteneffekte. Das Ergebnis von Add (nicht der Rückgabewert) ist eine neue Struktur, die das neue Element enthält. Der Seiteneffekt ist die Positionierung in eben dieser Struktur. Angenommen zwei oder mehr Teilnehmer wollen gleichzeitig einen *Knoten* erzeugen, der an der gleichen Position erscheinen soll. Der eigene *Knoten* wird bei den jeweiligen Erzeugern an diese Stelle verschoben. Wenn die aktuellen globalen *Events* ankommen, werden die neuen *Knoten* der anderen Teilnehmer an die gleiche Stelle verschoben. Dieses Problem existiert, da die Benutzer selbst entscheiden können wo die *Knoten* erzeugt werden sollen. Eine Lösung wird zunächst nicht benötigt, lediglich eine geeignete Visualisierung.

4.2.4.4 Eindeutige Knotenidentität

Wenn ein Benutzer einen *Knoten* erzeugt, dann dauert es eine kurze Zeit bis dieser *Knoten* bei allen anderen Teilnehmern der Zusammenarbeit erzeugt wird und sichtbar ist. In dieser Zeit soll der erstellende Benutzer bereits mit der Bearbeitung beginnen können. Dass alle Teilnehmer den *Knoten* kennen, weiß der Ersteller, wenn das entsprechende *Event* in der globalen Eventsequenz erscheint. Erst zu diesem Zeitpunkt verfügt der *Knoten* über eine gesicherte Identität. Diese Identität muss aber vorher schon gewährleistet werden. Um dieses Problem zu lösen, müssen IDs so erzeugt werden können, dass sie von Anfang an eindeutig und gesichert sind. Dies geht nur, wenn die ID eine Kennzeichnung des Erzeugers enthält. Damit auch dieser Ansatz erfolgreich sein kann, muss gewährleistet werden, dass die Benutzer keine IDs ihrer Mitbenutzer erzeugen können. In Unterabschnitt 4.2.1 Administrative Serverseite wurde die Nutzerkennung als sitzungsunabhängige eindeutige Ganzzahl festgelegt. Zusätzlich wurde in 4.2.2 Zusammenarbeit ermöglichende Serverseite die Knotenidentität als sitzungsabhängige eindeutige Ganzzahl definiert. Da jeder Benutzer seine Nutzerkennung kennt, könnte er beim Erzeugen eines neuen *Knotens* eine neue Knotenkennung erzeugen, die von keinem anderen Benutzer erzeugt werden kann. Angenommen eine solche Benutzerkennung kann erzeugt werden. Diese kann dann im Kontext der Add-Operation an die anderen Teilnehmer geschickt werden, die einen *Knoten* mit dieser ID in ihrem Dokument erzeugen. Zusätzlich werden Add-*Events* anderer Teilnehmer mit deren IDs lokal erzeugt. Die Benutzer müssen also eine Tabelle führen, die Knotenkennungen ihren *Knoten* zuordnet, sodass bei einem knotenspezifischen *Ereignis* der richtige *Knoten* gefunden werden kann.

Nutzerkennung und Knotenkennung, die beides 32-Bit Ganzzahlen sind, müssten unterschieden werden und gleichzeitig eine neue Zahl ergeben. Beispielsweise Nutzer 4 erzeugt *Knoten* 3. Dann erzeugt Nutzer 3 *Knoten* 4. Eine Möglichkeit diese beiden Fälle zu unterscheiden wäre das unterschiedliche gewichten. Beispielsweise wären die Ergebnisse von *10*4 + 3* und *10*3 + 4* die eindeutigen IDs für die beiden genannten Beispiele. Da aber mehr als 9 *Knoten* existieren können, ist dieses Verfahren nicht ausreichend. Die Gewichtung müsste *(2^{32})*<nutzer> + <knoten>* sein, um maximal 2^{32} *Knoten* und Benutzer in einer gemeinschaftlichen Sitzung zu ermöglichen. Das Ergebnis wäre dann aber keineswegs eine 32-Bit Ganzzahl. Da in jedem Fall also der Typ der Knotenkennung

angepasst werden muss, kann auch ein einfacheres Berechnungsverfahren verwendet werden, das auch die Ergebnisse für einen Betrachter verständlich erscheinen lässt. Es wäre also geeigneter die Nutzerkennung und die Knotenkennung als Zeichenketten mit einem nicht-numerischen Trennzeichen zu verbinden. Ein entsprechender Identitätsgenerator bräuchte dazu nur seine Nutzerkennung und eine Zählervariable verwalten, die bei jeder neuen ID einfach um eins erhöht und mit der Nutzerkennung zusammengefügt wird. Wenn eine Knotenkennung durch einen anderen Benutzer erzeugt wurde, müssen die anderen Nutzer diese dann speichern, da sich weitere Operationen des Nutzers auf eben diesen *Knoten* beziehen können. Wenn ein Nutzer einen *Knoten* entfernt, dann muss die ID des *Knoten* bei allen Nutzern aus deren Kennungsmanagement entfernt werden. Da Knotenkennungen zusammen mit einer Referenz auf den entsprechenden Mindmap-*Knoten* gespeichert werden, würden diese Referenzen, wenn sie weiterhin bei den Clients existieren, zu einem inkonsistenten Zustand des Dokuments führen können.

4.3 Implementierung

Nachdem das Servicemodell im vorherigen Abschnitt beschrieben wurde, wird in diesem Abschnitt die Implementierung der Dienste sowie der clientseitigen Komponenten beschrieben. Die Architektur auf der Serverseite setzt dabei auf ASP.NET[8] auf. Um ASP.NET zu hosten, wird ein virtueller Server von 1&1 verwendet, auf dem Windows Internet Information Services 7 läuft. Der Silverlight-Client wird über .aspx-Dateien an die zugreifenden Browser ausgeliefert. Um die Dienste aus dem vorherigen Abschnitt zur Verfügung stellen zu können, wird der in IIS 7[9] integrierte Windows Communication Foundation (WCF) Container verwendet. Die Konfiguration des Servers sowie IIS wird von Administratoren vorgenommen und ist damit nicht Bestandteil dieser Ausarbeitung. Auf der Seite des Clients, existiert der bereits bestehende Mindmap-Editor der im Rahmen der beschrieben Erweiterung eine Referenz auf eine Bibliothek (client library) enthält, die sämtliche Aspekte, der in Abschnitt 4.2 Entwurf beschrieben gemeinschaftlichen Funktionen bereitstellt. Diese Bibliothek kann von beliebigen Silverlight-Clients verwendet werden. Eine generelle Assembly könnte durch erneutes Kompilieren für das .NET Framework erstellt werden. Abbildung 22 zeigt die Client- und Serverseite, ähnlich wie in Unterabschnitt 3.2.4 Architektur, vor einem technischen Hintergrund.

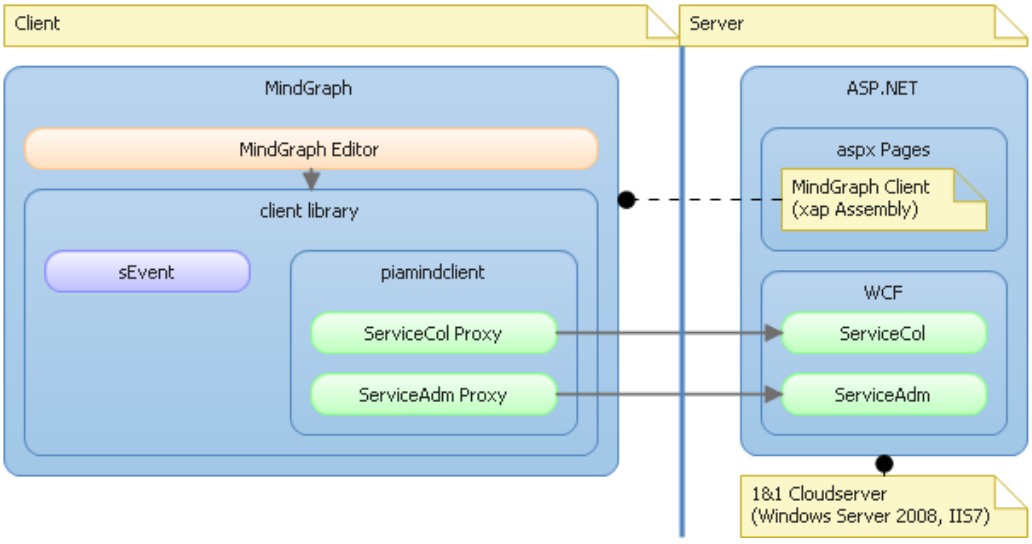


Abbildung 22: Aufbau der Client-Server-Umgebung

[8] ASP.NET Serverseite des .NET Frameworks
[9] IIS 7 Internet Information Services als Webserver-Komponente in Windows Server

Die wiederverwendbare Bibliothek besteht aus einem Framework für das Auslösen und Behandeln von *Events*, entsprechenden Proxy-Classen für die Kommunikation mit den WCF-Diensten und aus einer Integration mit spezifischen Funktionalitäten zur vereinfachten Verwendung der serverseitigen *piamind*-Infrastruktur. In diesem Abschnitt wird zunächst auf die Implementierung der Dienste `ServiceCol` für die gemeinschaftlichen Funktionen und `ServiceAdm` für die administrativen Funktionen eingegangen. Anschließend wird in Unterabschnitt 4.3.2 Ereignis-Framework das verwendete Framework zur Benutzung von *Ereignissen* beschrieben. In Unterabschnitt 4.3.3 Client Bibliothek wird die Client-Bibliothek betrachtet und wie eine Abstraktion von technischen Details der *Events* und Dienstaufrufen erreicht, beziehungsweise wie die Verwendung der Komponente konkret und wiederverwendbar gestaltet ist. Abschließend werden noch Anknüpfungspunkte in der bestehenden *MindGraph*-Anwendung behandelt, bevor auf einen alternativen Client eingegangen wird, der eine administrative Sicht auf die Service-Komponenten `ServiceCol` und `ServiceAdm` bietet und als Dashboard bezeichnet wird.

4.3.1 ASP.NET und WCF-Services

Auf der Serverseite werden die beiden Dienste für Administration und Zusammenarbeit in einem ASP.NET Projekt gehostet. Das ermöglicht die Erstellung der serverseitigen Komponenten mit einer Entwicklungsumgebung wie Visual Studio. Dies bringt vor allem den Vorteil des automatisierten Testens der Service-Klassen mit Unit-Tests mit sich, wovon im Rahmen der Entwicklung gebraucht gemacht wird. In diesem Unterabschnitt wird vorausgesetzt, dass der Leser mit ASP.NET und WCF vertraut ist. Um die Dienste über das Internet zur Verfügung zu stellen, müssen über die web.config des Web Projekts die entsprechenden Endpunkte definiert werden. Dabei ist zu beachten, das Silverlight-Clients nur WCF-Dienste konsumieren können, die ein `basicHttpBinding` als Kommunikationsverbindung verwenden. Wird das standardmäßige WebService-Binding (`wsHttpBinding`) verwendet, kann Visual Studio keinen Silverlight-Proxy erstellen und ein bestehender Proxy verursacht eine entsprechende Ausnahme. Die Dienste werden jeweils durch ein Interface spezifiziert, das als `ServiceContract` verwendet und dazu mit einem `ServiceContract`-Attribut gekennzeichnet werden muss. Sämtliche Operationen des Dienstes müssen in dem Interface mit einem `OperationContract`-Attribut versehen werden. Der eigentliche Dienst wird als Implementierung des Interfaces realisiert und als `BehaviorConfiguration` in der web.config registriert. Zusätzlich kann ein `ServiceBehavior`-Attribut verwendet werden, um das Nebenläufigkeitsverhalten der Dienste zu steuern. Die Dienste `ServiceCol` und `ServiceAdm` sind so spezifiziert, dass sie keine gleichzeitigen Aufrufe zulassen und Aufrufer stattdessen in einer Warteschlange automatisch verwalten. In dem Klassendiagramm in Abbildung 23 ist die Dienst-Schnittstelle von `ServiceAdm` abgebildet.

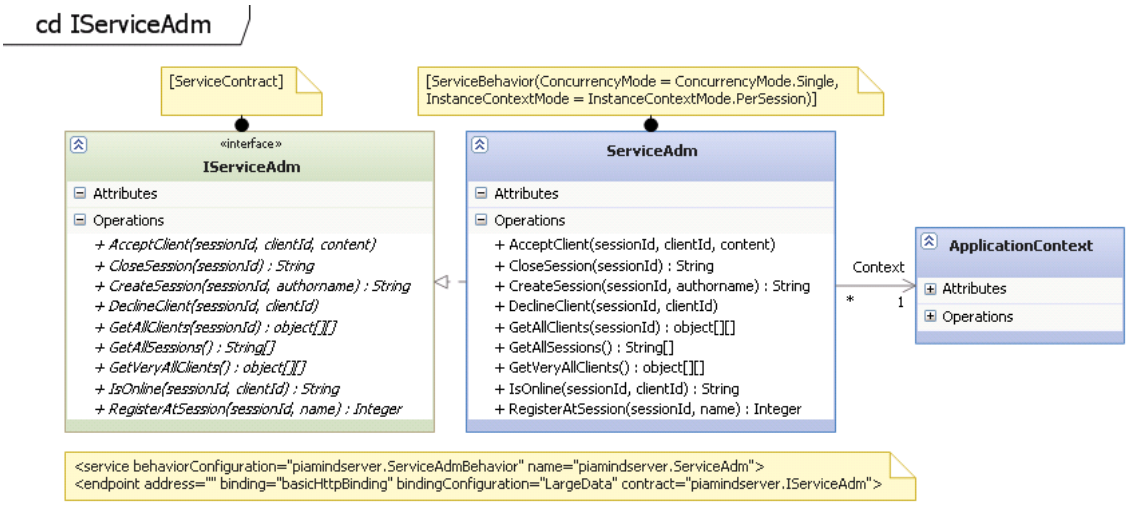


Abbildung 23: WCF-Dienst als administrative Serverseite

Es ist eine Referenz auf den `ApplicationContext` vorhanden, der in 4.3.1.1 Persistente Objekte vorgestellt wird. Darüber werden die Dienst-Aufrufe an den in Unterabschnitt 4.2.3 Integration vorgestellten `SessionManager` weitergeleitet. Der XML-Kommentar zeigt den relevanten Ausschnitt der Dienstkonfiguration aus der web.config-Datei der ASP.NET-Umgebung. Die Implementierung des `ServiceCol`-Interfaces nutzt den ebenfalls im selben Unterabschnitt beschriebenen `StateManager`, um sämtliche Aufrufe dorthin weiterzuleiten. Auch hier wird der `StateManager` über den `ApplicationContext` zur Verfügung gestellt, wie das Klassendiagramm des `ServiceCol`-Dienstes in Abbildung 24 darstellt.

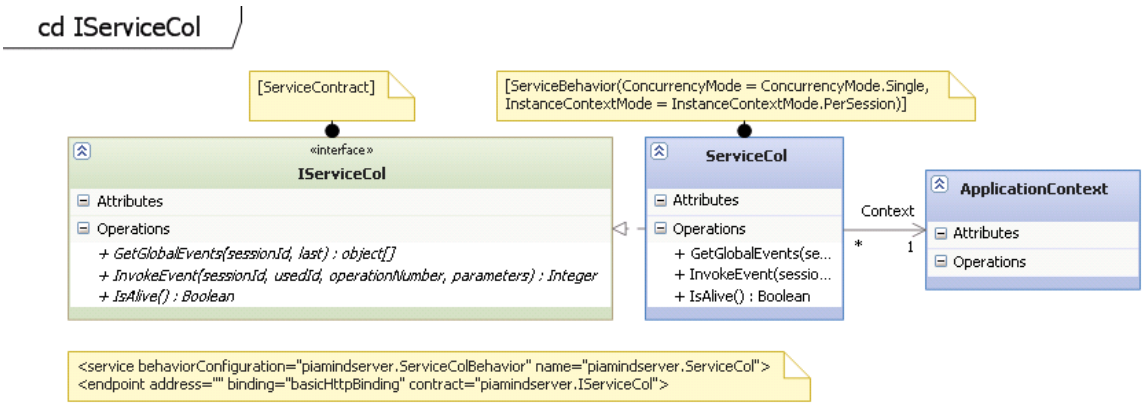


Abbildung 24: WCF-Dienst als Zusammenarbeit ermöglichende Serverseite

Es ist ersichtlich, dass der `ServiceAdm`-Dienst den Sequenzdiagrammen aus 4.2.1 Administrative Serverseite sehr ähnlich ist, während der `ServiceCol`-Dienst eine andere Spezifikation benutzt, als in 4.2.2 Zusammenarbeit ermöglichende Serverseite beschrieben. Die Ursache liegt in der Verwendung des bereits erwähnten *Ereignis*-Frameworks, das im Unterabschnitt 4.3.2 Ereignis-Framework vorgestellt wird.

4.3.1.1 Persistente Objekte

Da der WCF-Dienst eine Referenz auf den globalen `SessionManager` oder den `StateManager` besitzen muss, muss untersucht werden, ob eine Referenz auf ein einfaches Objekt ausreichend ist. Da der Windows Activation Service (WAS)[10] ein Objekt der Dienstimplementierung immer dann

[10] Windows Activation Service (WAS) Dienst im IIS der die Erzeugung von Objekten zur Laufzeit steuert.

erstellt, wenn ein Nutzer eine ASP.NET Sitzung öffnet und eine WCF-Operation aufruft. Eine Referenz auf ein statisches Objekt ist ebenfalls nicht ausreichend, da dieses statische Objekt das automatische Testen der Anwendung erschwert. Außerdem würde es viele Zugriffsprobleme verursachen, wenn das Nebenläufigkeitsverhalten des Dienstes irgendwann einmal geändert werden würde. Statt einer Referenz auf ein direktes Objekt, wird das Objekt im Applikationskontext von ASP.NET gespeichert. Diese Speicherstruktur gilt für die gesamte Laufzeit der ASP.NET-Anwendung und dessen Garbage Collection Einstellung kann über den IIS konfiguriert werden.

Bei jedem Dienstaufruf wird also die Referenz aus diesem Kontext verwendet. Doch standardmäßig haben WCF-Dienste keinen Zugriff auf diesen Speicherbereich, sondern nur Objekte vom Typ `Page`. Der Silverlight-Client (MindGraph.xap) wird in einer solchen `Page` (mit der Dateiendung "aspx") über ASP.NET an den anfragenden Browser ausgeliefert. Die Referenz auf den Applikationskontext, der vom Typ `HttpApplicationState` ist, wird in einem `ApplicationContext`-Objekt gekapselt. Ist der Applikationskontext nicht verfügbar, werden die Objekte in `ApplicationContext` zwischengespeichert. Anschließend kann jeder Dienst darauf zugreifen und den Session- und `StateManager` verwenden. Da die Dienste nur für sich nicht nebenläufig sind, kann es vorkommen, dass `ServicCol` und `ServiceAdm` gleichzeitig auf den `SessionManager` im Kontext zugreifen. Wie in Unterabschnitt 4.2.3 Integration beschrieben, werden `StateChange`−Objekte einer `Session` zugeordnet. Da die .NET Framework Strukturen thread safe sind, kann es vorkommen, dass bei einem Zugriff auf ein nicht mehr vorhandenes Element im `SessionManager` lediglich eine Exception geworfen wird. Diesen Fall muss das StateManager-Objekt berücksichtigen. Die Struktur von `ApplicationContext` ist im Klassendiagramm in Abbildung 25 abgebildet. Dabei ist zu beachten, dass `ApplicationContext` eine eigene Klasse ist und nicht mit dem Applikationskontext von ASP.NET verwechselt werden darf.

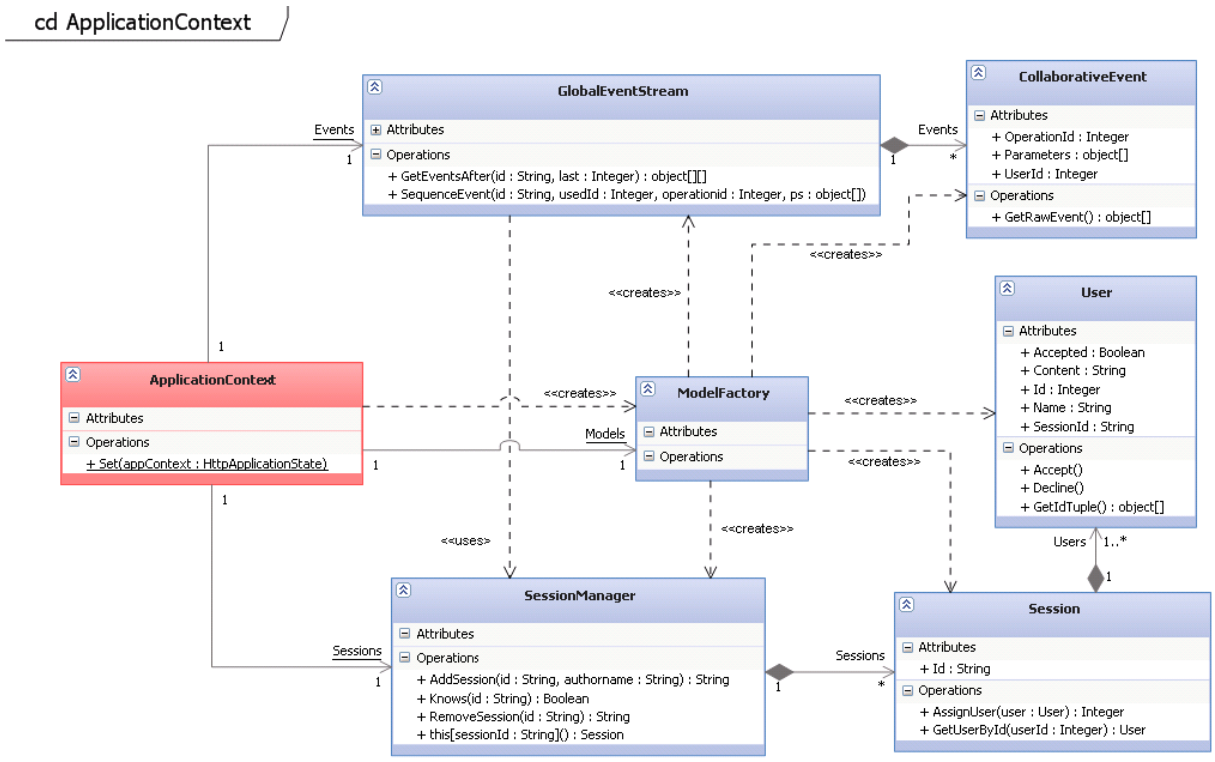


Abbildung 25: Implementierte integrierte Klassenstruktur auf der Serverseite

Der Unterschied zu der Struktur, die in der Entwurfsphase (4.2 Entwurf) in Unterabschnitt 4.2.3 Integration als integrierte Struktur beschrieben wurde, besteht hauptsächlich darin, dass der `StateManager` durch einen `GlobalEventStream` ersetzt wurde, der `CollaborativeEvent`-Objekte statt `StateChange`-Objekte verwaltet. Es wurde bereits angedeutet, dass die ursprünglich beschriebenen atomaren Operationen (4.1.1 Atomare Operationen) eine gemeinsame Struktur haben und als *Ereignis*-Operationen zusammengefasst werden können. Details dazu werden in Unterabschnitt 4.3.2 Ereignis-Framework behandelt. Weiterhin werden die Objekte nicht mehr direkt erzeugt, sondern die Erzeugung wird durch eine `ModelFactory` übernommen. Referenzen auf diese Struktur werden durch den `ApplicationContext` zur Verfügung gestellt und zusätzlich in den Applikationskontext von ASP.NET gespeichert, wenn dieser wie oben beschrieben, durch eine `Page` übergeben wird.

4.3.1.2 Unit-Tests für WCF

Es wurde erwähnt, dass die Entwicklung der Dienste getestet vorgenommen wird. Das bedeutet, dass das Verhalten der Dienste wie im Abschnitt 4.2 Entwurf definiert, zunächst durch Unit-Test-Fälle spezifiziert wird. Anschließend wird der Code geschrieben, der von den entsprechenden Unit-Test-Spezifikationen getestet wird, bis die Testfälle erfolgreich sind. Es werden dabei nicht alle Testfälle auf einmal zuerst geschrieben, sondern immer nur für ein detailliertes Verhalten. Wenn das Verhalten dann implementiert wurde und der Test erfolgreich ist, dann erfolgt die Spezifizierung der nächsten Funktion. Auf diese Weise erfolgt der Test- und Implementierungsvorgang iterativ und somit nach dem Prinzip des Test Driven Development, wie es Robert C. Martin auf seinem ehemaligen Blog beschreibt [Rob05].

Die Testfälle beziehen sich dabei auf die einzelnen Operationen, die die Dienste anbieten. Jeder Testfall erzeugt einen anfänglichen Zustand, führt die entsprechende Operation durch und vergleicht das gewünschte Ergebnis mit dem tatsächlichen Ergebnis. Es wird also nicht direkt das konkrete Verhalten, sondern lediglich das Ergebnis getestet. Wie die Dienste das Verhalten dann implementieren ist den Diensten überlassen. Wenn sich die Implementierung der Dienste ändert, dann müssen die Testfälle nicht geändert werden, da diese lediglich mit den Operationen, die die Dienste anbieten, arbeiten. Dies ist natürlich nur der Fall, wenn sich die Methodensignaturen nicht ändern. WCF-Dienste lassen sich sehr einfach testen, denn sie verhalten sich wie normale Objekte unter Test. Die erfolgreiche Ausführung eines Testlaufs sagt daher nichts über das Funktionieren des Dienstes aus, sondern gibt lediglich Auskunft über das Funktionieren des Dienstverhaltens. Das reicht für die Entwicklung der Dienste aber vollkommen aus, da der WCF-Container in ASP.NET nicht getestet werden muss, sondern lediglich der eigene Code. Eine Auflistung aller Testfälle der Dienste kann dem Anhang entnommen werden.

4.3.2 Ereignis-Framework

Die in Unterabschnitt 4.1.1 Atomare Operationen definierten Operation werden nicht direkt auf Dienstaufrufe abgebildet, wie in Unterabschnitt 4.2.2 Zusammenarbeit ermöglichende Serverseite dargestellt. Dies wäre zu unflexibel, da somit bei jeder Änderung der Struktur der Dienst neu kompiliert und veröffentlich werden müsste. Stattdessen werden alle atomaren Operationen über eine einzige generische Dienstoperation aufgerufen, die von den einzelnen Aufrufen abstrahiert. Die Abstraktion ist möglich, da alle atomaren Operationen die gleiche Struktur haben: sie gehören zu einer Sitzung und zu einem Nutzer, sie sind eindeutig und haben Parameter. Diese Struktur wird in einem `CollaborativeEvent`-Objekt der Servicestruktur abgebildet, wie in Abbildung 25 in Unterabschnitt 4.3.1.1 Persistente Objekte dargestellt. Während das ursprünglich geplante

`StateManager`-Objekt konkrete Operationen für die Verwaltung von atomaren Operationen hatte, kann ein `GlobalEventStream`-Objekt generisch mit *Ereignissen* umgehen. Abstrakte *Ereignisse* behandeln ihre Parameter einfach wie Objekte vom Typ `object`, welcher glücklicherweise die Oberklasse von allen anderen .NET Typen ist. Man kann diese Abstraktion wie in Abbildung 26 als Klassendiagramm darstellen, indem man die Klassen aus der Designphase von abstrakten Klassen der Implementierungsphase ableiten lässt: `StateManager` verhält sich wie `GlobalEventStream`. Also kann `GlobalEventStream` als abstrakte Repräsentation von `StateManager` verwendet werden.

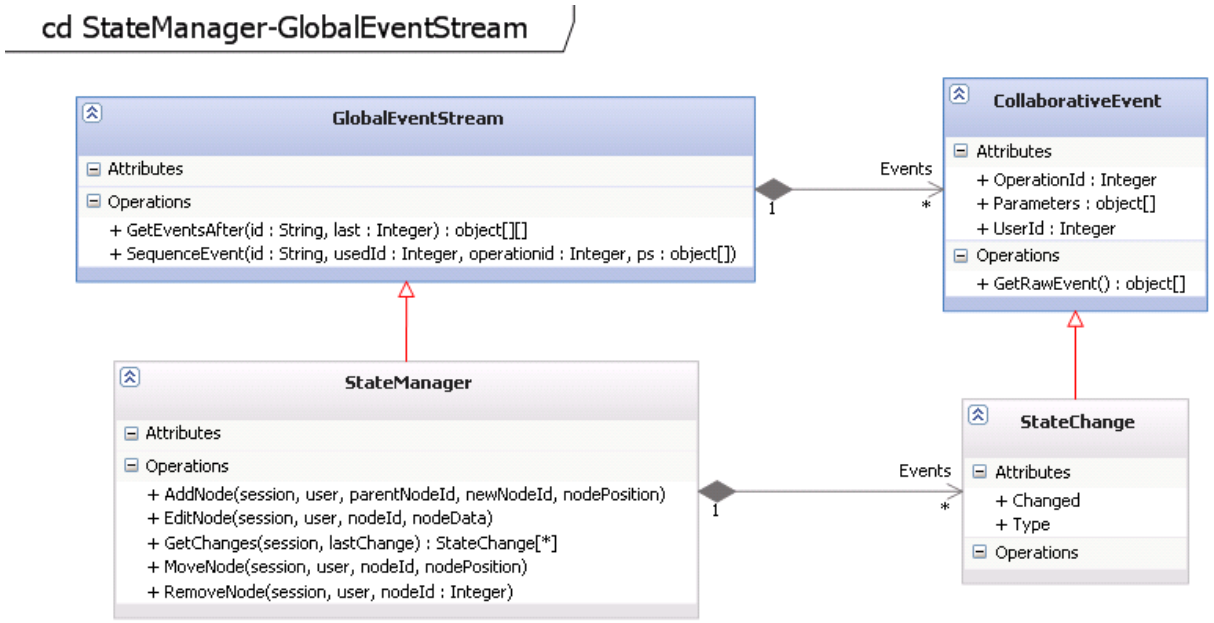


Abbildung 26: StateManager wird durch GlobalEventStream abgelöst

Die generische Operation für das Ausführen eines *Events* besitzt folgende Signatur.

> *SequenceEvent (sessionId, userId, operationId, args) : operationId*
>
> operationId := Integer; sessionId := String; userId := Integer; args := object[];

Welche atomare Operation aufgerufen wird, wird über eine eindeutige Operationskennung festgelegt. Um das Ende des Aufrufs dem Aufruf der richtigen atomaren Operation zuordnen zu können, wird die Operationskennung wieder zurückgegeben. *UserId* und *SessionId* ermöglichen die Zuordnung zu einem Teilnehmer und einer Sitzung, während *args* eine Liste der Parameter darstellt. Die `GetChanges`-Operation bleibt als `GetEventsAfter` erhalten und gibt eine Liste mit *Events* zurück.

> *GetEventsAfter(sessionId, lastSequenceNumber) : events*
>
> events := (sessionId, [event]*); event := (operationId, userId, args);

Das Modell auf der Serverseite ist fähig mit diesem generischen System umzugehen, es muss lediglich ein anderer `StateManager` eingeführt werden. Die `StateChange`-Klasse besitzt bereits ein Typ-Feld, das die Art des *Events* beschreibt. Die administrative Seite ist davon nicht betroffen und bleibt wie beschrieben. Es gilt zu klären, wie der Client diese generische Operation so aufrufen kann, sodass er die Illusion hat, eine typsichere atomare Operation aufzurufen. Eine Verarbeitung des

Ergebnisses der `GetEventsAfter`-Operation muss ebenfalls typsicher erfolgen können. Die Lösung dieser Implementierung liegt in der Benutzung eines einfachen *Ereignis*-Frameworks. Da vorhandenen Frameworks zu viel Funktionalität bieten und sich nur unter hohem Aufwand an einen eigenen Dienst koppeln lassen, wurde ein eigenes Framework namens *sEvent* entwickelt. Dieses Framework lässt sich mit beliebigen Endpunkten verwenden. Der benötigte Client muss nur von der abstrakten Klasse `EventProxy` erben und die Factory-Methode überladen. Diese Methode liefert eine Funktion zurück, die bei Aufruf ein Objekt zurückliefert, das `IEventService` implementiert.

Das Gleiche gilt für die administrative Seite. Dafür bietet das Framework das Interface `IAdministratorService`, durch das sich ein eigener Endpunkt für die administrativen Aspekte implementieren lässt. Entsprechende Implementierungen für die Benutzung der WCF-Dienste `ServiceCol` und `ServiceAdm` werden im Unterabschnitt 4.3.3 Client Bibliothek beschrieben. Die Klassenstruktur des Frameworks ist in dem Klassendiagramm in Abbildung 27 ersichtlich. Dabei versucht das Framework die Serverseite auf dem Client zu spiegeln. `Session`- und `User`-Objekte haben zumindest eine ähnliche Struktur. Dabei ist wichtig zu beachten, dass administrative Objekte nicht direkt erzeugt werden können. Um an ein `Session`-Objekt zu gelangen, muss über den `SessionManager` zunächst eine `Session` erzeugt werden. Anschließend kann eine Liste mit allen `Session`-Objekten abgefragt werden. Dabei gibt das Framework die Objekte nicht direkt zurück, sondern über vorher registrierte Callbacks. Die Antwort auf die Sitzungsanfrage muss also vom Aufrufer erwartet werden. Erst diese Sitzungsliste enthält die `Session`-Objekte. Sämtliche Operationen des `SessionManagers` können nur mit einem `Session`- und/oder `User`-Objekt durchgeführt werden.

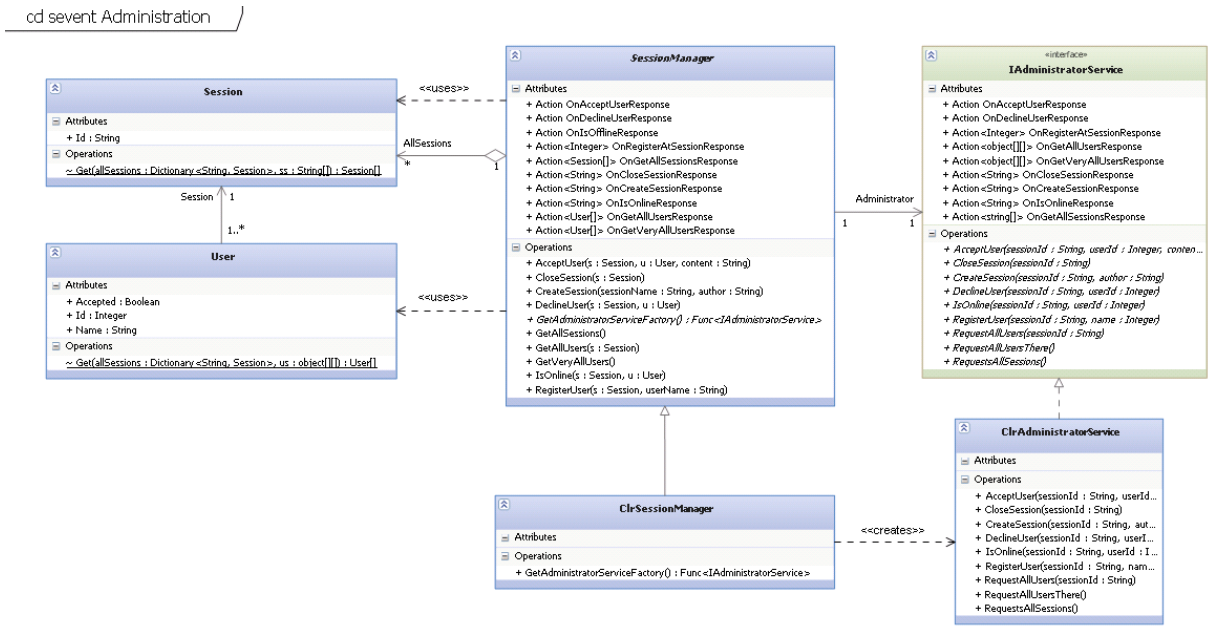


Abbildung 27: Administrative Klassenstruktur in *sEvent*

Das Ereignismanagement von *sEvent* hat die Aufgabe, *Ereignisse* typsicher zu definieren und auch zu behandeln. Diese typsicheren *Events* müssen dann in eine endpunktunabhängige Struktur verpackt werden. Vom Endpunkt erhaltene *Events* müssen entsprechend ausgepackt werden. Dieser Verpackungsvorgang wird durch `EventServiceIdMapper` vorgenommen, während sich `EventProxy` um die Typsicherheit kümmert. Mit Hilfe von `EventRepresentation`-Objekten, lassen sich die *Events* mit den definierten Typen behandeln. Das Klassendiagramm in Abbildung 28

zeigt dabei nur die Teile des Frameworks, die *Events* mit einem generischen Parameter unterstützen. Für mehrere Parameter existieren passende Überladungen der Klassen- und Methodensignaturen.

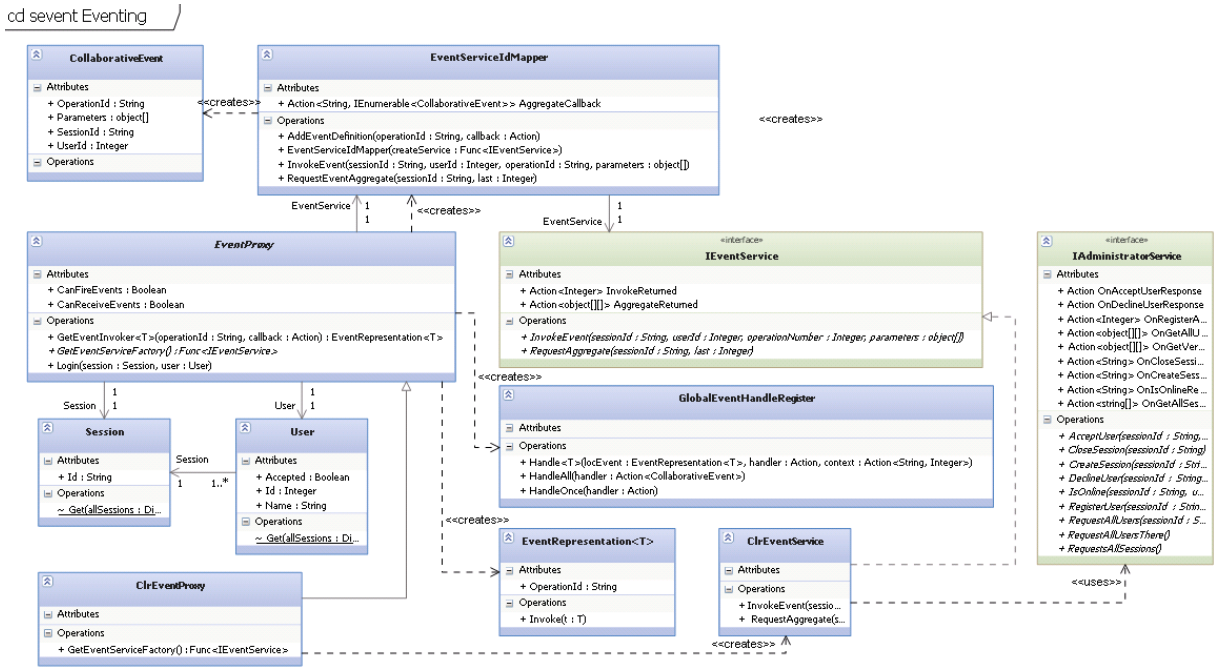


Abbildung 28: Zusammenarbeit ermöglichende Klassenstruktur in *sEvent*

Mit dem *sEvent*-Framework können die atomaren Operationen also als `EventRepresentation`-Objekte erzeugt und somit typsicher behandelt werden. Dabei setzt *sEvent* auf das Registrieren von Callbacks und Eventhandlern, da so ein endpunktabhängiges asynchrones Verhalten sehr einfach unterstützt werden kann, wie das bei WCF-Aufrufen zum Beispiel der Fall ist. In Codeausschnitt 1 wird gezeigt, wie atomare Operationen mit *sEvent* definiert und ausgeführt werden können. Dabei ist zunächst wichtig, dass ein konkreter `EventProxy` erzeugt, mit einem Endpunkt verbunden und ein Benutzer dort angemeldet wurde. Dann kann ein beispielhaftes "add"-*Ereignis*, wie in Codeausschnitt 1 gezeigt, definiert werden.

```
User u = ...//irgendein Nutzer-Objekt
EventProxy ep = …//WCF-Proxy oder CLR-Proxy
ep.Connect();
ep.Login(u);
var add = ep.Get<string, int>("add");
add.Invoke("test", 5);
```

Codeausschnitt 1: Beispielhafte *Event*-Definition mit *sEvent*

Ausgeführt können die definierten *Events* dann über den Aufruf der `Invoke`-Methode. Hier können die definierten Parameter typsicher übergeben werden. Die Aufrufe sind nur erfolgreich, wenn der `EventProxy` als Benutzer an einer gültigen Sitzung angemeldet ist. Ansonsten können die Aufrufe nicht durchgeführt werden. Das Abfragen der Eventsequenz ist auch möglich, wenn keine Anmeldung als Nutzer möglich ist, allerdings muss dann wenigstens eine Anmeldung an eine `Session` vorhanden sein. Ist dies auch nicht der Fall, dann ist der Aufruf der `get`-Funktion nicht durchführbar. Da ein Nutzer immer einer `Session` zugeordnet ist, ermöglicht die Anmeldung als Nutzer das Ausführen und Abfragen von *Ereignissen*.

```
var get = ep.GetRequestor((h) =>
{
      h.Handle(add,
            (s, i) => ConsoleWriteline("string:" + s + ", int:" + i),
            (s, u) => ConsoleWriteline("Session:" + s.ToString() + "," +
                                       "User: " + u.ToString ()));
});
get(0);
```

Codeausschnitt 2: Beispielhafte *Event*-Behandlung mit *sEvent*

In der ersten Codezeile in Codeausschnitt 2 wird ein *Event*-Abfrager erzeugt, der das oben definierte "add"-*Ereignis* behandelt. Die Behandlung wird mit der Registrierung von zwei Delegates zu dem zu behandelnden *Event* definiert. Der erste Callback enthält die typsicheren Parameter, die das zu behandelnde *Ereignis* enthalten kann. Der zweite Callback liefert Informationen zu Sitzung und Nutzer, des zu behandelnden *Events*. Mit der letzten Zeile des oberen Codeausschnitts werden alle *Events* vom `EventService` abgefragt. Der Parameter dieses Aufrufs gibt an, wie viele *Events* bereits abgefragt wurden und bei diesem Aufruf ignoriert werden können. Es ist also sinnvoll, diesen Wert hochzuzählen, um nicht immer alle *Events* zu erhalten. In der Regel sind lediglich neue *Events* von Interesse. Wenn diese vom Service zurückgeliefert werden, kommt der Kontrollfluss bei dem im `GetRequestor` definierten `Handle` an und kann dort zur typsicheren Behandlung der *Ereignisse* beitragen. Das setzt voraus, dass von allen Clients in der Zusammenarbeit die gleichen *Events* durch `EventRepresentation`-Objekte mit identischer Signatur definiert sind.

Neben den Funktionalitäten für *Ereignisse*, verfügt das Framework auch über das bereits beschriebene Sitzungsmanagement. So lassen sich Sitzungen erstellen, an denen sich Benutzer anmelden können. Angemeldete Nutzer einer Sitzung können abgefragt und bei Bedarf akzeptiert werden. Das Beispiel in Codeausschnitt 3 zeigt den dafür nötigen Code.

```
SessionManager sm = …//WCF-Manager oder CLR-Manager
sm.CreateSession("sessionname", "authorname");
sm.OnGetAllSessionsResponse += (sa) => {//sa = Liste aller Sitzungen
      sm.OnRegisterAtSessionResponse += (uid) => {//uid = Nutzerkennung
            sm.OnGetAllUsersResponse += (ua) => {//ua = Liste der Nutzer
                  User u1 = ua.First((u) => u.Id == uid);
                  sm.AcceptUser(sa[0], u1, "hallo");
            };
            sm.GetAllUsers(sa[0]);
      };
      sm.RegisterUser(sa[0], "nutzername1");
};
sm.GetAllSessions();
```

Codeausschnitt 3: Sitzungs- und Nutzermanagement mit *sEvent*

In Codeausschnitt 3 ist zu sehen, dass das Ergebnis der `GetAllSessions`-Anfrage vor dem Aufruf der Abfrage als .NET-Event definiert werden muss. Das gleiche gilt auch für die `RegisterUser`- und `GetAllUsers`-Anfrage. Dies ist der Fall, da das Ergebnis erst später vorhanden sein kann. Wann die eigentliche Aktion mit Nutzer- und Sitzungs-Objekt, im Beispiel das Aktivieren eines Nutzers mit der `AcceptUser`-Anfrage, erfolgt, ist undefiniert, muss aber vor den eigentlichen Aufrufen definiert werden. Dafür setzt das Framework auf ein Programmiermodell für asynchrone Funktionsaufrufe, wie es bereits in der Ereignisbehandlung durch registrierte Handler angedeutet wurde. Dadurch eignet es sich für die Verwendung in Zusammenhang mit Diensten über das Internet, die bekannterweise hochgradig asynchron funktionieren, wie zum Beispiel WCF.

Unit-Tests für sEvent

Um die Funktionsfähigkeit des Frameworks verifizieren zu können, wurde es unter der Verwendung von Unit-Tests entwickelt. Die Testfälle decken dabei das endpunktunabhängige Verhalten ab, indem genau wie bei den Tests für die Service-Komponenten das Ergebnis der API-Aufrufe geprüft wird. Anstatt spezieller Endpunkte enthält das *sEvent*-Framework CLR-Provider, die die Nutzung ohne WCF oder ähnlichen Technologien ermöglichen. Standardmäßig wird also ein `ClrEventService` und ein `ClrAdministratorService` verwendet, der das Verhalten beliebiger Endpunkte simuliert. Da dieses Verhalten durch die Schnittstellen `IEventService` und `IAdministratorService` vorgegeben wird, können so die darauf aufsetzenden Funktionen getestet werden. Wenn spezielle Endpunkt-Provider definiert werden, kann zumindest von der Korrektheit der verwendenden Infrastruktur ausgegangen werden. Eine Liste sämtlicher Testfälle der *sEvent*-Implementierung kann dem Anhang entnommen werden.

4.3.3 Client Bibliothek

Um von dem *sEvent*-Framework und den Proxys für die WCF-Dienste zu abstrahieren, wird eine dedizierte Bibliothek für den Client verwendet. Es wurden bereits die Möglichkeiten der Endpunkterweiterung in *sEvent* erwähnt. Für die Verwendung in Verbindung mit den WCF-Diensten `ServiceCol` und `ServiceAdm` wurden spezielle Wrapper implementiert, die als Adapter für die von Visual Studio generierten Proxy-Klassen verwendet werden. Ein entsprechender `WcfEventService` kapselt alle spezifischen Dienstaufrufe, während `WcfAdministratorService` die Sitzungsverwaltung übernimmt. Das zu implementierende Interface `IEventService` ist dem WCF-`ServiceContract` `IServiceCol` auf der Serverseite sehr ähnlich, sodass die Adaption einfach vorgenommen werden kann.

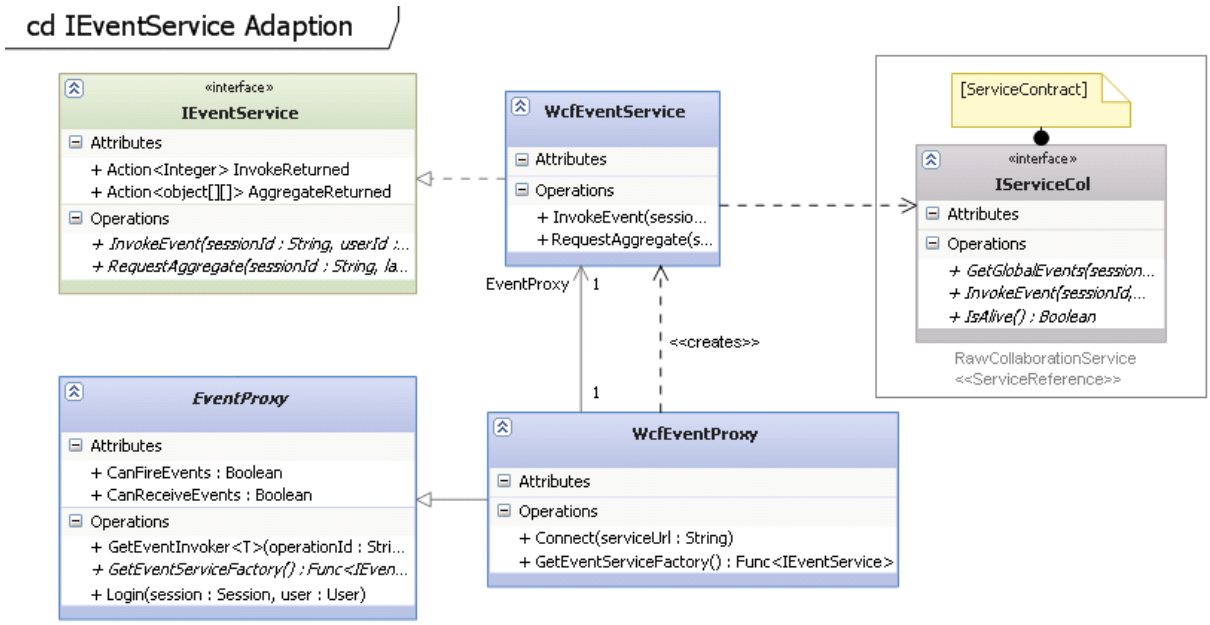


Abbildung 29: Adapter zur Verwendung von WCF-Proxy und IEventService

Der `RawCollaborationService`-Bereich in Abbildung 29 entspricht dem automatisch erzeugten WCF-Proxy. Zum Verständnis wurde dort anstelle des generierten Codes das serverseitige Interface abgebildet. Das `WcfEventService`-Objekt versteckt dabei alle Operationen die nicht benötigt werden. Das `IAdministratorService`-Interface ist dem `ServiceContract`

IServiceAdm ebenfalls sehr ähnlich, was eine einfache Anpassung durch den WcfAdministratorService-Adapter ermöglicht.

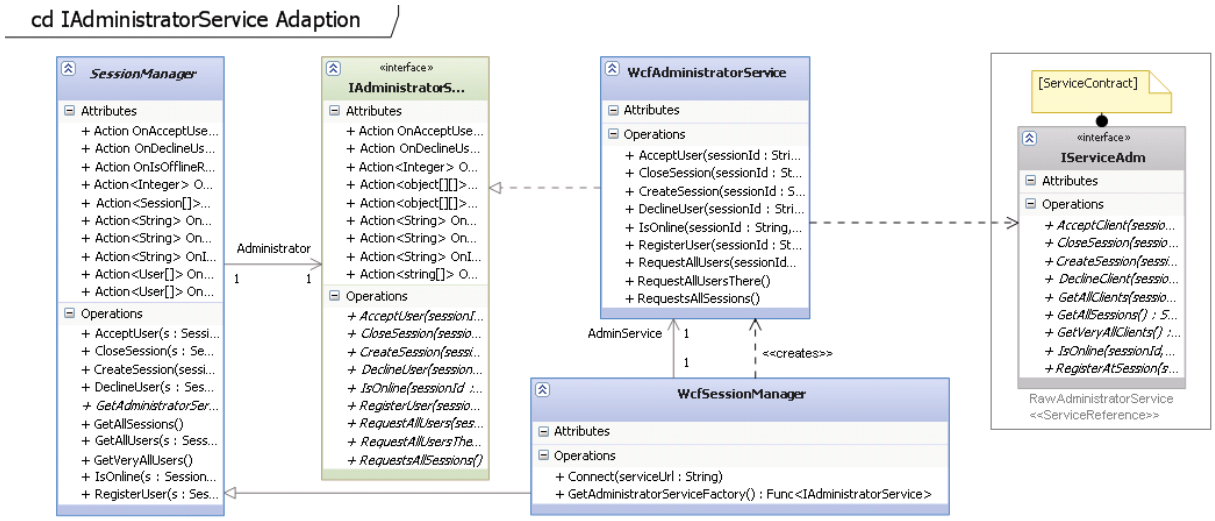


Abbildung 30: Adapter zur Verwendung von WCF-Proxy und IAdministratorService

Der RawAdministratorService-Bereich in Abbildung 30 entspricht dem automatisch erzeugten WCF-Proxy. SessionManager verfügt über eine direkte Referenz auf ein IAdministratorService. Zusätzlich verfügt WcfSessionManager über eine weitere Referenz, aber direkt auf den WcfAdministratorService. Dies ist erforderlich, um spezifische Operationen direkt ausführen zu können, wie zum Beispiel die Connect-Methode. Alle generellen Operationen werden von dem abstrakten SessionManager auf der Schnittstelle ausgeführt. Nachdem das *sEvent*-Framework nun erfolgreich an die WCF-Endpunkte angepasst ist, kann eine domänenspezifische Komponente definiert werden, die das einfache Verwenden durch Clients ermöglichen soll. In dem speziellen Namespace für *MindGraph*, werden die aus den Anforderungen bekannten vier atomaren Operationen, welche in Unterabschnitt 4.1.1 Atomare Operationen beschrieben wurden, mit Hilfe von *sEvent* definiert. Codeausschnitt 4 zeigt diese Definitionen.

```
OperationAdd = this.wcfEventProxy.GetEventInvoker
                  <string, string, double, double>("add",
                        () => andlers.LocalAddHandler());
OperationEdit = this.wcfEventProxy.GetEventInvoker
                  <string, string>("edit",
                        () => handlers.LocalEditHandler());
OperationMove = this.wcfEventProxy.GetEventInvoker
                  <string, double, double>("move",
                        () => handlers.LocalMoveHandler());
OperationRemove = this.wcfEventProxy.GetEventInvoker
                  <string>("remove",
                        () => handlers.LocalRemoveHandler());
```

Codeausschnitt 4: Definition der atomaren Operationen mit *sEvent*

Um auch diese *Events* zu behandeln, ist EventRequestor wie in Codeausschnitt 5 definiert. Das handlers-Objekt enthält dabei die konkreten Behandlungsmethoden, die durch den benutzenden Client zur Verfügung gestellt werden. Wie dieses handlers-Objekt erzeugt und mit Callbacks ausgestattet werden kann, ist in Unter-Unterabschnitt 4.3.4.1 Auslösen von Events abgebildet.

```
OperationEventRequestor = this.wcfEventProxy.GetAggregateRequestor((h) =>
{
      h.HandleAll((ce) => handlers.GlobalAllHandler(
            ce.SessionId, ce.OperationId, ce.UserId, ce.Parameters));
      h.HandleOnce(
            () => handlers.GlobalOnceHandler());
      h.Handle(this.Events.OperationAdd,
            (pn,nn,px,py) => handlers.GlobalAddHandler(pn, nn, px, py),
            (s,u) => { });
      h.Handle(this.Events.OperationEdit,
            (n,s) => handlers.GlobalEditHandler(n, s),
            (s,u) => { });
      h.Handle(this.Events.OperationMove,
            (n,px,py) => handlers.GlobalMoveHandler(n, px, py),
            (s,u) => { });
      h.Handle(this.Events.OperationRemove,
            (n) => handlers.GlobalRemoveHandler(n),
            (s, u) => { });
});
```

Codeausschnitt 5: Behandlung der *Events* der atomaren Operationen mit *sEvent*

Eine allgemeine Zugriffsklasse für Clients ermöglicht die Verwendung der Sitzungsverwaltung und der generellen Funktionalitäten bezüglich *Ereignisse*. Von dieser Klasse können spezialisierte Zugriffspunkte abgeleitet werden, die spezifische *Ereignisse* unterstützen. Anschließend kann der spezialisierte Client die entsprechende Ergänzung verwenden. In dem Klassendiagramm in Abbildung 31 ist `PiamindClient` als allgemeine Zugriffsklasse dargestellt. *MindGraph*-spezifische *Ereignisse* werden in der `PiamindMindGraphClient`-Klasse spezifiziert. `PiamindMindGraphClient` ist damit ein spezieller Client. Dazu verwendet dieser eine Referenz auf ein `PiamindMindGraphClientEvents`-Objekt, welches Methoden zum typsicheren Auslösen und Behandeln von *Ereignissen* zur Verfügung stellt. Die `MindGraphServiceClient`-Klasse stellt eine Möglichkeit dar, die gesamte Funktionalität von der *piamind*-Infrastruktur im Kontext von *MindGraph* über ein zentrales Objekt zu verwenden.

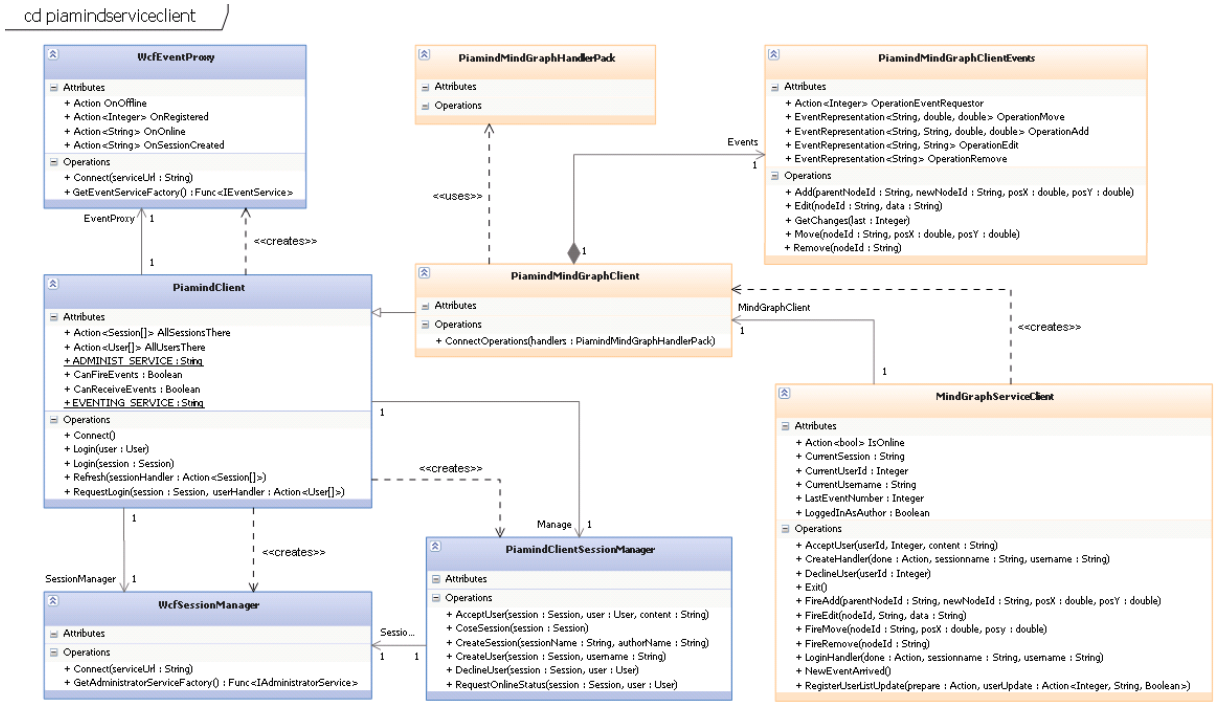


Abbildung 31: Klassenstruktur von Client-Bibliothek "piamindclient"

Die hell eingefärbten Klassen stellen die domänenspezifischen Komponenten dar, während die dunkel eingefärbten Klassen die allgemeingültigen Funktionen kapseln und eine einfache Erstellung weiterer Zugriffspunkte ermöglichen.

In dem speziellen *MindGraph*-Zugriffspukt, also der `MindGraphServiceClient`-Klasse, ist das Konzept der Zusammenarbeit als Polling-Mechanismus implementiert und erfordert das Aufrufen der `GetChanges`-Funktion in `PiamindMindGraphClientEvents` alle zwei Sekunden. Bei Bedarf kann dieses Intervall entsprechend angepasst werden. Die Nutzerliste einer Sitzung, an die der `PiamindClient` angemeldet ist, wird durch `MindGraphServiceClient` alle acht Sekunden aktualisiert. Um diese Aktualisierungen zu behandeln, werden registrierte Callbacks entsprechend aufgerufen.

4.3.4 MindGraph Anbindung

Um die vorgestellte Client-Bibliothek in den bestehenden *MindGraph*-Client zu integrieren, muss die bereits in Abschnitt 3.1 Vorarbeit beschriebene Struktur um eine Interception-Struktur erweitert werden, die bereits in Unterabschnitt 4.2.4 Clientseite erwähnt wurde. Dazu muss die in Unterabschnitt 3.1.2 Topologische Objekte vorgestellte `TopologyControl`-Klasse um zwei Abhängigkeiten erweitert werden. Die erste ist eine Referenz auf ein Interceptor-Objekt, das das Interface `IIterceptor` implementiert. Die zweite Abhängigkeit besteht darin, dass es auf einem übergebenen `TopologyContext`-Objekt die Methode `ApplyToTopology` aufruft und sich dort selbst übergibt. Dadurch kann `TopologyContext` die weitere Konfiguration übernehmen. Dazu gehört zum Beispiel die Erzeugung eines `IdProvider`-Objekts, das die in 4.2.4.4 Eindeutige Knotenidentität beschriebenen Knotenkennungen verwaltet. Zusätzlich wird ein `DefaultReactor` erzeugt, über den *Ereignisse* auf die Mindmap angewendet werden können. Dazu bedient sich das Erzeuger-Objekt direkt der *Knoten*, die im `IdProvider` mit IDs gespeichert werden, und verändert diese entsprechend. Dafür müssen die *Knoten* vorher allerdings registriert worden sein.

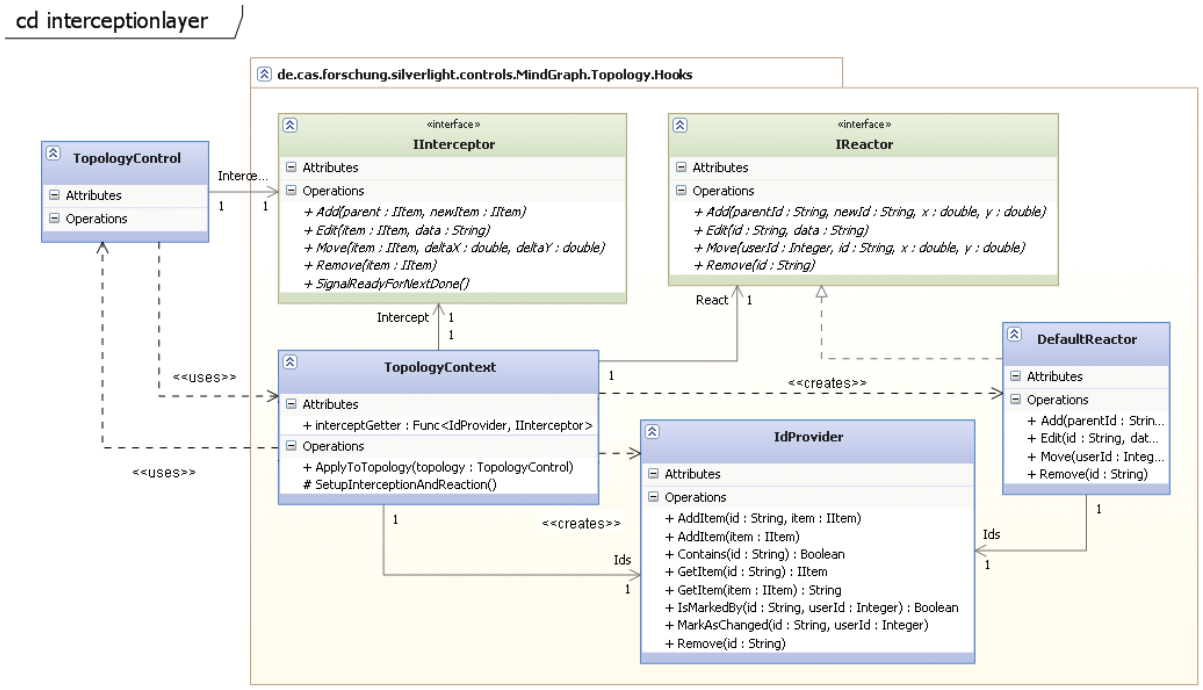


Abbildung 32: Aufbau der Interception-Struktur in *MindGraph*

Dieser Vorgang wird durch einen Interceptor durchgeführt, der von `TopologyControl` über sämtliche benutzergesteuerten Vorgänge in der Mindmap informiert wird. Im dem Klassendiagramm in Abbildung 32 ist die Struktur des Interception-Namespace dargestellt. Wenn ein eigener Interceptor oder Erzeuger benutzt werden soll, dann muss ein spezieller `TopologyContext` vom allgemeinen abgeleitet werden, der die `SetupInterceptionAndReaction`-Methode überschriebt.

4.3.4.1 Auslösen von Events

Wenn der Benutzer eine Aktion in dem Mindmap-Editor ausführt, dann müssen entsprechend *Ereignisse* ausgelöst werden und an die Interception-Komponente weitergegeben werden. Nachdem die dafür nötige Infrastruktur beschrieben wurde, wird nun dessen Benutzung behandelt. Im Klassendiagramm in Abbildung 33 ist als zentrale Komponente die `CollaborationConnector`-Klasse erkennbar.

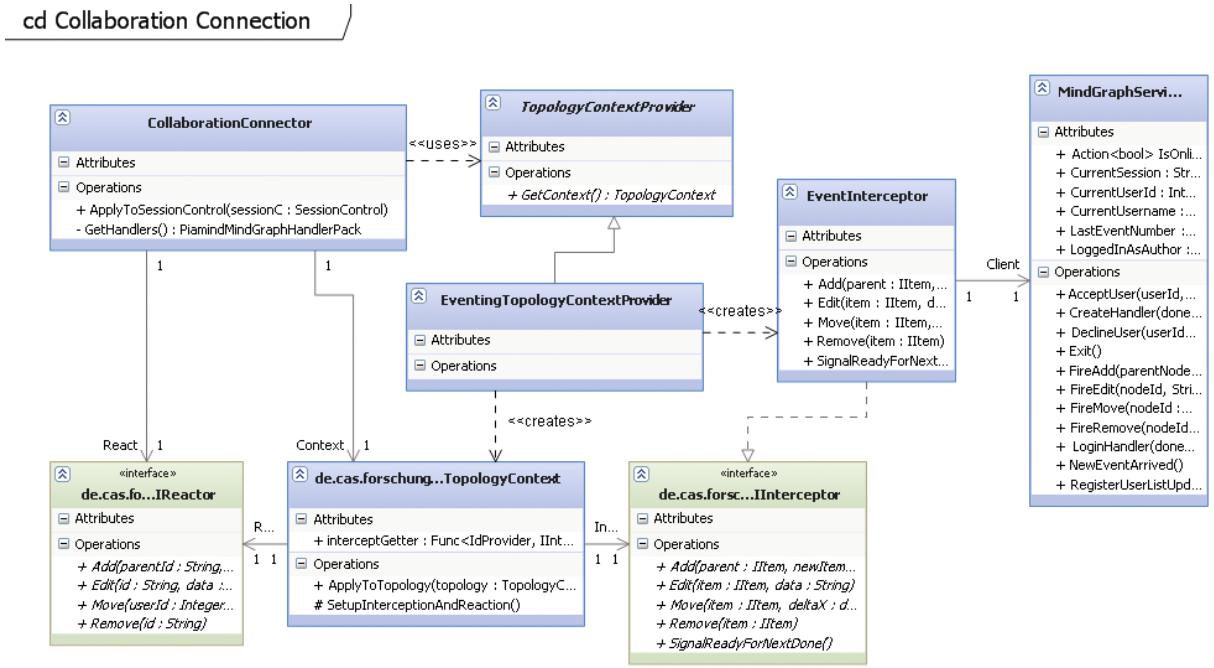


Abbildung 33: Verbindung von Interaktion in *MindGraph* mit *Ereignis*-Service

Diese Klasse verbindet ein `EventInterceptor`-Objekt mit einem `TopologyContext`-Objekt. Dabei handelt es sich um einen eigenen Interceptor, der *Events* direkt an ein `MindGraphServiceClient`-Objekt weitergibt. Die Klasse `MindGraphServiceClient` wurde in Unterabschnitt 4.3.3 Client Bibliothek mit ihrer Funktionalität vorgestellt. So können *Ereignisse*, die in der Mindmap geschehen, direkt an die Client-Bibliothek weitergegeben werden. Damit ist die erste Seite der Integration von Zusammenarbeit abgeschlossen.

Die zweite Seite betrifft das Eintreffen von *Ereignissen* vom *piamind*-Service, wie in Unterabschnitt 4.2.2 Zusammenarbeit ermöglichende Serverseite geplant. Diese *Ereignisse* müssen auf das aktuelle Dokument angewendet werden. Dafür wird bei der Erstellung des `CollaborativeConnector`-Objekts ein `SessionControl` registriert, das in Unter-Unterabschnitt 4.3.4.4 Oberflächenerweiterung beschrieben wird. Wenn bei diesem `SessionControl`, welches auch die Referenz auf den im Abbildung 33 dargestellten `MindGraphServiceClient` hält, ein `PiamindMindGraphHandlerPack`-Objekt registriert wird, müssen die Behandlungen der *Events*

vom Server spezifiziert werden. Diese Spezifizierung erfolgt bei der Erzeugung des Handler-Pakets, wie in Codeausschnitt 6 gezeigt. Anschließend kann über die Verwendung des `CollaborationConntector`-Objekts die Mindmap komplett von außen beobachtet und manipuliert werden. Diese Möglichkeit ist im Sinne der Zusammenarbeit erwünscht und unterstützt die Erweiterbarkeit des Editors. Sollte eine anderes Backend für den *MindGraph*-Editor verwendet werden, existiert somit nur eine Stelle, die angepasst werden muss. Wenn in Unter-Unterabschnitt 4.3.4.4 Oberflächenerweiterung die Oberfläche zur Unterstützung von Sitzungsverwaltung und Zusammenarbeit erweitert wird, stellt das `CollaborationConntector`-Objekt die Schnittstelle zwischen der Mindmap und den in Unterabschnitt 4.3.1 ASP.NET und WCF-Services beschriebenen Diensten dar.

```
return new PiamindMindGraphHandlerPack()
{
      LocalAddHandler = eventDoneSignal,
      LocalEditHandler = eventDoneSignal,
      LocalMoveHandler = eventDoneSignal,
      LocalRemoveHandler = eventDoneSignal,
      GlobalAllHandler = (s, o, u, a) => this.CountNewEvent(u),
      GlobalAddHandler = (pn, nn, dx, dy) =>
            this.context.React.Add(pn, nn, dx, dy),
      GlobalEditHandler = (n, d) =>
            this.context.React.Edit(n, d),
      GlobalMoveHandler = (n, dx, dy) =>
            this.context.React.Move(this.ceUserId, n, dx, dy),
      GlobalRemoveHandler = (n) =>
            this.context.React.Remove(n)
};
```

Codeausschnitt 6: Definition konkreter *Event*-Behandlungen für *MindGraph* in *sEvent*

Die Aufrufe an den `DefaultReactor`, wie in Unterabschnitt 4.3.3 Client Bibliothek bei der Ereignisbehandlung gezeigt, können über das hier erzeugte Objekt vom Typ `PiamindMindGraphHandlerPack` ausgeführt werden.

4.3.4.2 Nichtauslösen von Events

Wenn Aktionen durch den Erzeuger durchgeführt werden sollen, die als *Ereignisse* anderer Nutzer ausgelöst werden, dürfen keine lokalen *Events* entstehen, die an den registrierten Interceptor weitergeleitet werden. Würde der Aufruf einer Methode von einem Erzeuger dazu führen, dass ein Interceptor informiert werden würde, würden diese wieder an den WCF-Dienst geschickt. Solche *Ereignisse* würden nie endgültig behandelt werden können. Um dieses ungewünschte Verhalten zu vermeiden, muss sichergestellt werden, dass die Operationen, die ein Erzeuger aufruft, nicht durch Benutzeraktionen aufgerufen werden können. Nur dann können diese Erzeugeroperationen so implementiert werden, dass keine Benachrichtigungen erzeugt werden. Andererseits müssen alle Operationen, die als Ergebnis einer direkten Nutzerintegration aufgerufen werden, den Interceptor von `TopologyControl` über Add, Edit, Move und Remove informieren. *Events* die nach der Feststellung aus Unter-Unterabschnitt 4.2.4.2 Events ignorieren nicht beachtet werden brauchen, werden ebenfalls nicht ausgeführt. Der in Unterabschnitt 4.3.4 MindGraph Anbindung beschriebene `IdProvider` kümmert sich darum, indem er neben der *Knoten*-Knotenkennungs-Zuordnung auch den letzten Bearbeiter eines *Knotens* speichert. Wenn dann ein eigenes *Event* eintrifft, wird die Ignorierbarkeit geprüft und das *Event* nach erfolgreicher Prüfung nicht beachtet. *Ereignisse* werden ebenfalls nicht beachtet, wenn Sie sich auf gelöschte *Knoten* beziehen, oder *Knoten*, die noch nicht erstellt wurden.

4.3.4.3 Mindmap Verwaltung

Eine Anforderung an die gemeinschaftliche Mindmap-Modellierung besteht laut (K4) Dokumentenspeicher in einer Möglichkeit, Map-Dokumente zu laden, die nicht auf der lokalen Festplatte gespeichert sind. Ein solcher Dokumentenspeicher würde das Bearbeiten von Mindmaps an jedem Ort ermöglichen. Folglich könnten gemeinschaftliche Sitzungen mit bestehenden Dokumenten von überall gestartet werden, wo ein Internetzugang vorhanden ist. Da diese Anforderung nicht für die eigentlichen Aspekte der Zusammenarbeit relevant ist, wurde sie bisher nicht beachtet. Die in Unterabschnitt 4.1.3 Zusammenfassung vergebene Priorität C rechtfertigt dies. Dennoch muss dieses Feature implementiert werden. Da der Mindmap-Editor eventuell in ein bestehendes System integriert wird, steht die endgültige Art und Weise der Integration des Dokumentenspeichers noch nicht fest. Um für diesen Prototyp dennoch eine temporäre Möglichkeit zu haben, Mindmaps online abzuspeichern und zu laden, wurde ein proprietäres Web-basiertes Dateisystem mit dem Namen *piafile* entwickelt. Dieses System besteht aus einem einfachen WCF-Dienst, der mit einer speziellen Berechtigung Verzeichnisse und Dateien auf dem Server erzeugen kann. Eine Bibliothek für den Client ermöglicht den Zugriff auf diesen Dienst und erlaubt im Prinzip zwei wesentliche Operationen. Die erste Operation ist das Speichern von Dokumenten. Der entsprechende Dialog ist in dem Bildschirmausschnitt in Abbildung 34 abgebildet.

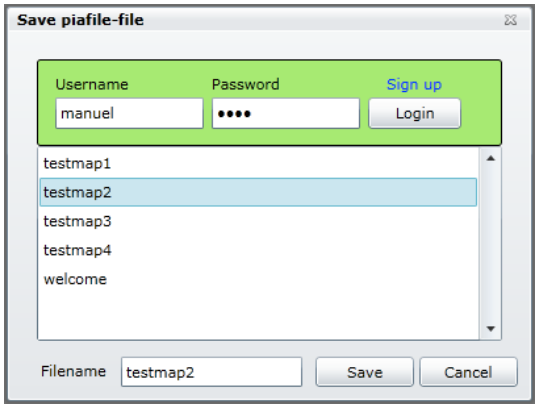


Abbildung 34: Speichern einer Mindmap im *piafile*-Dienst

Um den Dienst nutzen zu können, braucht ein Benutzer ein Benutzerkonto, das er über den Link "Sign Up" erstellen kann. Mit dem Benutzernamen und dem Password wird dann auf dem Server ein Verzeichnis angelegt. Wenn dieser Nutzer sich dann anmeldet und Dokumente speichert, dann werden diese Dokumente in dem Verzeichnis des Nutzers abgelegt. Bei einer erneuten Anmeldung können die gespeicherten Dokumente dann geladen werden, wie in Abbildung 35 gezeigt.

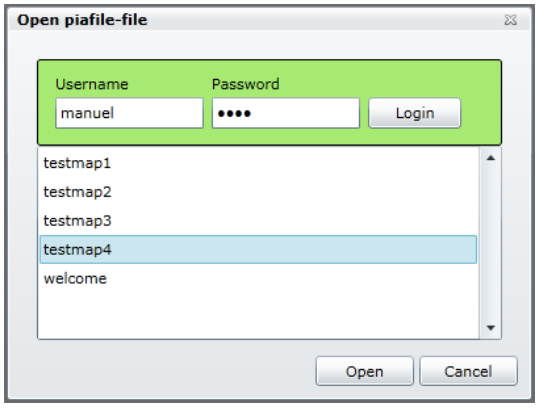


Abbildung 35: Öffnen einer Mindmap aus dem *piafile*-Dienst

Diese beiden Dialoge werden direkt in den *MindGraph*-Editor integriert, sodass dieser die aktuelle Mindmap serialisieren und in *piafile* speichern kann. Das Laden von Dokumenten erfolgt entsprechend. Dazu kann die bereits in Unterabschnitt 3.1.3 Mindmap-Editor erwähnte Funktionalität zum Speichern und Laden verwendet und muss lediglich an die *piafile*-Bibliothek angepasst werden. Als Dateiformat wird die ebenfalls erwähnte proprietäre XML-Abbildung verwendet. Die Integration von *piafile* wird vorübergehend als bevorzugte Art und Weise verwendet um Dokumente zu laden und zu speichern. Die bisherige Funktionalität wird in Import und Export umbenannt und als sekundär behandelt. Das Online-Dateisystem *piafile* kann auch unabhängig von *MindGraph* verwendet werden. Dazu besitzt es eine eigene grafische Oberfläche, die das Verwalten der Dokumente eines angemeldeten Nutzers ermöglicht. Abbildung 36 zeigt diese Oberfläche mit einem geladenen Mindmap-Dokument in XML.

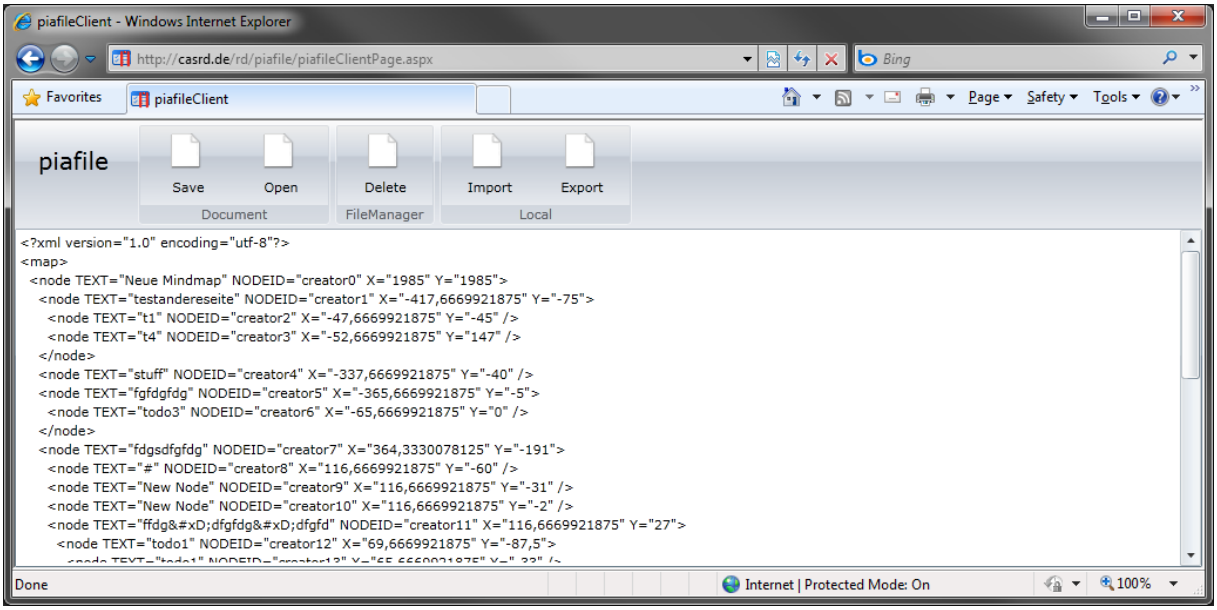


Abbildung 36: Geöffnete Mindmap im *piafile*-Client

Der hier abgebildete *piafile*-Client kann unter http://casrd.de/rd/piafile/ aufgerufen und benutzt werden.

4.3.4.4 Oberflächenerweiterung

Für die Erweiterung der Oberfläche des bestehenden *MindGraph*-Clients wird ein spezielles Silverlight-`UserControl` benötigt, das sowohl den Vorgang der Eröffnung einer gemeinschaftlichen Sitzung sowie das Beitreten zu einer solchen unterstützt. Die anfängliche Darstellung des Controls ist in dem Bildschirmausschnitt in Abbildung 37 dargestellt.

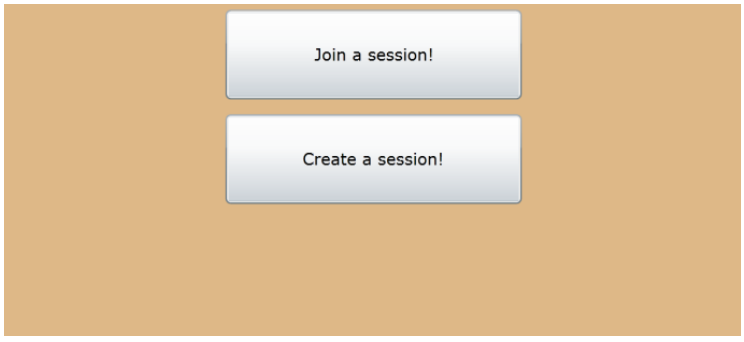


Abbildung 37: Start-Dialog zur Sitzungsverwaltung

Dieses Silverlight-Control wird im Code als `SessionControl`-Objekt verwendet. Um die Integration mit der im *MindGraph*-Editor verwendeten Mindmap-Struktur zu ermöglichen, wird ein `CollaborationConnector`-Objekt verwendet, das in Unter-Unterabschnitt 4.3.4.1 Auslösen von Events vorgestellt wurde.

Für die Integration des Controls in die bestehende Oberfläche muss zunächst festgelegt werden, wann eine Sitzung eröffnet werden kann. Für das Starten einer Zusammenarbeit muss der Autor über ein geöffnetes Mindmap-Dokument verfügen. Dieses hat er beispielsweise über *piafile* geladen, wie in Unter-Unterabschnitt 4.3.4.3 Mindmap Verwaltung beschrieben. Dann kann er über den Button "Create a session" den Sitzungserstellungs-Dialog aus Abbildung 38 sichtbar machen, in dem er eine gewünschte Sitzungskennung sowie seinen eigenen Namen eingeben kann. Anschließend kann er die Sitzung öffnen und erhält damit den rechten Dialog aus Abbildung 38.

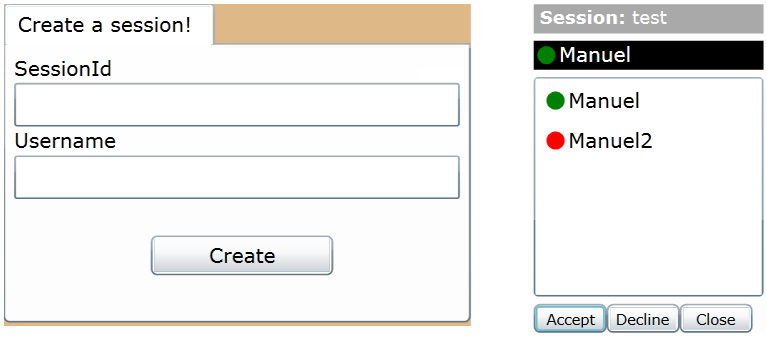


Abbildung 38: Dialog zum Eröffnen einer neuen Sitzung (links) und Benutzerliste (rechts)

Sobald die Sitzung gestartet wurde, werden die Ergebnisse der Benutzerinteraktionen als *Ereignisse* der Mindmap-Manipulation an den Server geschickt. Um andere Nutzer zu einer Sitzung einzuladen, muss der Autor nur seine Sitzungskennung weitergeben. Wenn ein anderer Teilnehmer sich angemeldet hat, kann der Autor diesen über die Kontrollleiste in der Benutzerliste akzeptieren, wie in Abbildung 38 dargestellt. Wenn er dies veranlasst, wird automatisch die komplette Mindmap im *MindGraph*-XML-Format in eine Zeichenkette gespeichert und mit der aktuellen Eventsequenznummer versehen. Der akzeptierte Benutzer erhält diese Zeichenkette, welche automatisch deserialisiert und als aktuelle Mindmap geöffnet wird. Da die aktuelle Eventsequenznummer dem Client des Benutzers bekannt ist, werden automatisch alle *Ereignisse* vom Server abgefragt die seit dem Zeitpunkt der Serialisierung stattgefunden haben. Diese werden dann auf die Mindmap angewendet. Für jedes *Ereignis* wird die erhaltene Sequenznummer um eins erhöhen, um zukünftige *Events*, die der Client in regelmäßigen Abständen versucht abzufragen, nicht mehrfach zu erhalten. Ab diesem Zeitpunkt ist der Benutzer vollständig und aktiv an der Sitzung angemeldet.

Wenn die Sitzung viele *Events* verursacht und lange andauert, dann könnte die Sequenznummer überlaufen, da sie nur eine 32-Bit Ganzzahl ist. Für den Prototyp ist dieses Problem aber nicht wahrscheinlich und kann daher ignoriert werden. Eine Lösung für ein produktives Umfeld müsste dieses Problem lösen. Damit ein Nutzer an einer Sitzung teilnehmen kann, muss seine aktuelle Mindmap geschlossen werden. Nur dann kann die serialisierte Version des Autors geöffnet werden. Diese serialisierte Version wurde in Unter-Unterabschnitt 4.2.1.2 Zusammenarbeit beitreten auch als Sitzungsgeheimnis oder globaler Zustand zum Zeitpunkt der Benutzeraktivierung bezeichnet.

Um aus einer Sitzung auszutreten, muss der Teilnehmer lediglich sein Browserfenster schließen oder zu einer anderen Seite wechseln. Dadurch bleibt die Nutzerkennung weiterhin an der Sitzung angemeldet. Gemäß den Anforderungen aus Unterabschnitt 3.2.1 Zusammenarbeit in Mindmaps in Bezug auf (K3) Multiuserfähigkeit, ist es nicht erforderlich die Lösung so zu gestalten, dass Nutzer sich abmelden können. Sie müssen lediglich zusammenarbeiten können. In diesem Prototyp ist dieses Verhalten tolerierbar. Wenn der Autor seine Sitzung verlassen möchte, kann er diese entweder explizit schließen, oder sie einfach nur verlassen. Verlässt er sie nur, können die anderen Teilnehmer weiterhin gemeinschaftlich die aktuelle Mindmap bearbeiten. Solange die Nutzer an einer gemeinschaftlichen Sitzung teilnehmen, können sie keine anderen Mindmap-Dokumente in dem geöffneten Editor bearbeiten. Die Buttons für das Öffnen und Neu Anlegen eines Dokuments sind dafür deaktiviert. Der in Abbildung 38 abgebildete Dialog für die Sitzungserzeugung ist in Kombination mit der ebenfalls dargestellten Benutzerliste in die bestehende Oberfläche des *MindGraph*-Editors integriert. Der Bildschirmausschnitt in Abbildung 39 zeigt die integrierte Version.

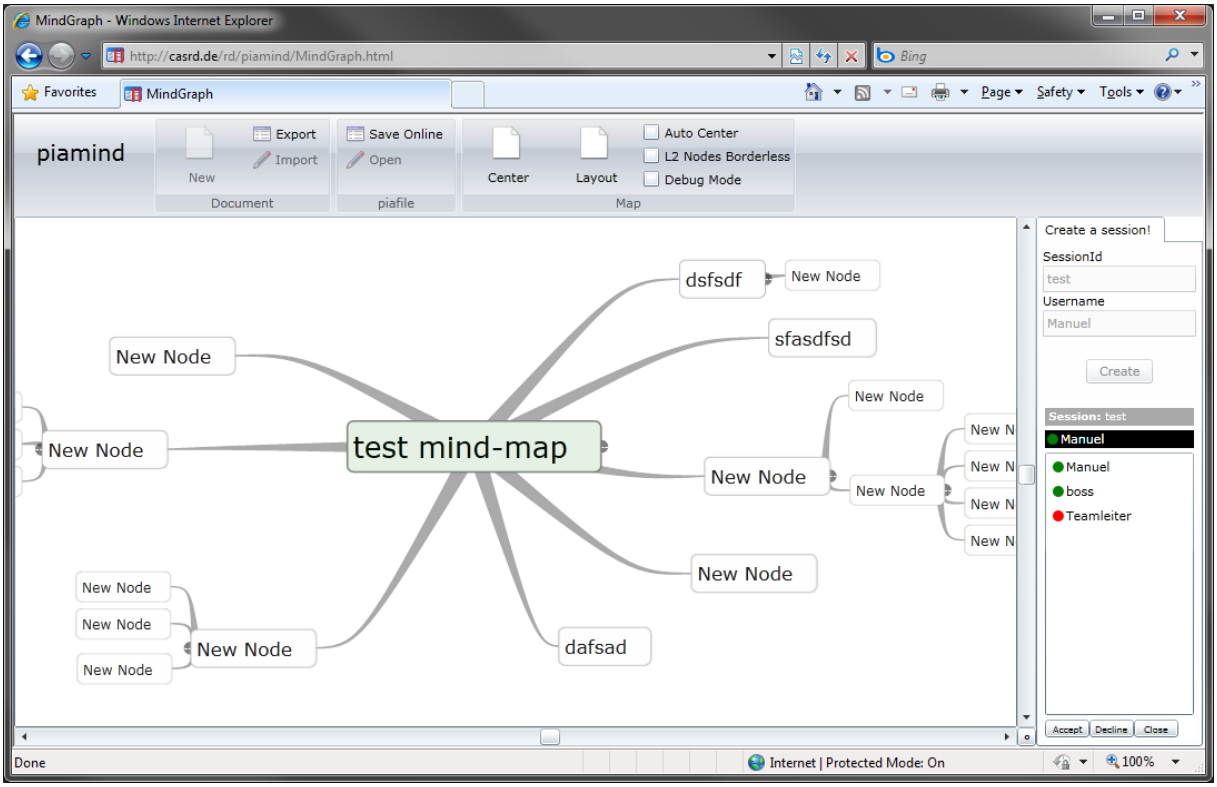


Abbildung 39: *MindGraph*-Oberfläche mit integrierter Sitzungsverwaltung und Nutzerliste

4.3.5 Dashboard Client

Um die Zusammenarbeit verwalten zu können, wurde eine spezielle Sicht auf den Service implementiert. In diesem sogenannten Dashboard wird auf alle *Events* reagiert, indem sie einfach angezeigt werden. Man sieht im Dashboard also sofort die komplette Sequenz und die Nutzer eine Sitzung. Zusätzlich können Sitzungen verwaltet und Nutzer aktiviert oder deaktiviert werden. Selbst wenn ein Nutzer deaktiviert ist, kann man über das Dashboard *Events* im Namen dieses Nutzers verschicken. Dafür nutzt das Dashboard die gleiche Client-Bibliothek, wie sie auch der *MindGraph*-Client verwendet. Es wird lediglich nicht der Domänen-Client verwendet, sondern eine allgemeine Zugriffsklasse, wie im Klassendiagramm in Unterabschnitt 4.3.3 Client Bibliothek dargestellt.

So meldet sich das Dashboard bei dem Auswählen einer Sitzung automatisch als diese Sitzung bei dem `WcfEventProxy` an. Ab diesem Zeitpunkt kann es alle *Events* abrufen, die existieren. Wenn

ein Benutzer in der Sitzung ausgewählt wurde, meldet sich das Dashboard automatisch als dieser Benutzer an und ermöglicht das Erzeugen der *Events*. Da es dafür nicht die *MindGraph*-spezifischen Zugriffsklassen verwendet, brauch es nicht auf den Aktivierungsstatus des Nutzers zu achten. Das macht das Dashboard als administrative Komponente auch für Anwendungen geeignet, die andere Eventtypen als Add, Edit, Move und Remove verwenden. Dafür wird ein typunabhängiger Eventhandler registriert, wie in Codeausschnitt 7 gezeigt wird.

```
OperationEventRequestor = this.wcfEventProxy.GetAggregateRequestor((h) =>
{
      h.HandleAll((ce) =>
      {
            PublishEventToUi(
                  ce, ce.SessionId, ce.OperationId, ce.UserId);
            lastEventSequenzNumber++;
      }
});
Execute.Every(5.Seconds())(() =>
{
      If (selectedUser != null){ ep.Login(selectedUser); }
      else if(selectedSession != null){ ep.Login(selectedSession); }
      else { return; }
      OperationEventRequestor(lastEventSequenzNumber);
});
```

Codeausschnitt 7: Behandlung von undefinierten *Events* mit *sEvent*

Für jedes eintreffende *Ereignis* wird eine direkte Referenz auf ein `CollaborationEvent`-Objekt übergeben, sodass die Behandlung allgemeingültig durchgeführt werden kann. In dem Bildschirmausschnitt in Abbildung 40 ist die Oberfläche des Dashboards dargestellt.

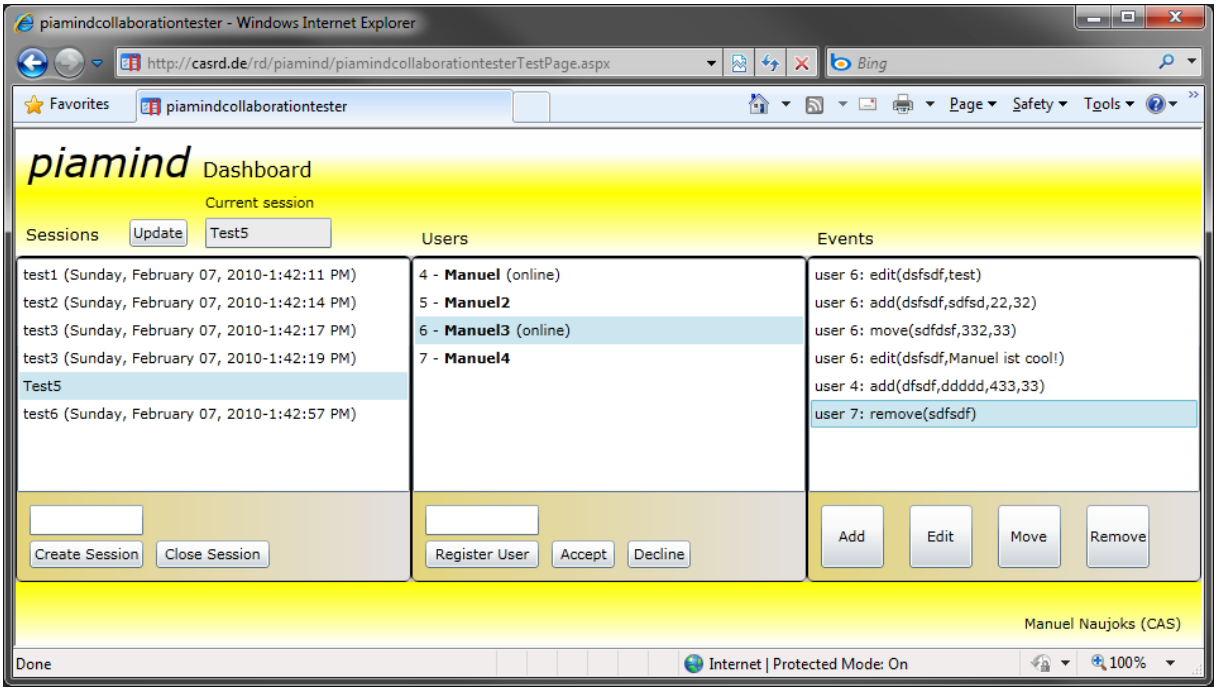


Abbildung 40: Dashboard zur administrativen Verwaltung von Sitzungen

Da in einem *Event*-Objekt nur eine Operationskennung als Nummer enthalten ist, versucht das Dashboard diese Nummer über die *MindGraph*-Klassen der Client-Bibliothek aufzulösen. Der entsprechende Name wird nur angezeigt, wenn dies möglich ist. Ähnliches gilt für die *Event*-Buttons.

Sie ermöglichen das Senden der Add-, Edit-, Move- und Remove-*Events*, indem sie versuchen die Zugriffsklassen für *MindGraph* zu verwenden. Dadurch werden die *Events* entsprechend codiert und können nur von anderen *MindGraph*-Clients als solche dargestellt werden. Sollte eine Sitzung *Ereignisse* verwenden, die anders codiert sind, werden diese durch die spezifischen `EventProxy`-Handler einfach nicht behandelt, sondern nur von `HandleAll`-Handlern, wie sie beispielsweise vom Dashboard benutzt werden. Letztendlich dient das Dashboard zur Übersicht über alle laufenden Sitzungen und welche Nutzer bei einer solchen Sitzung angemeldet sind. Diese administrative Oberfläche dient nur zum Verwalten des Prototyps und ist mit einem Passwortdialog geschützt.

4.4 Ergebnis der ersten Iteration

Nachdem in diesem Kapitel das Thema Zusammenarbeit in Mindmaps analysiert, geplant und implementiert wurde, wird in diesem Abschnitt der erste Teil dieser Ausarbeitung abgeschlossen. Das Ergebnis des ersten Teils ist der neue Mindmap-Editor, der die Bearbeitung von Dokumenten mit mehreren Benutzern unterstützt. Es handelt sich dabei um einen Prototyp, der theoretisch nicht für die produktive Benutzung vorgesehen ist. Dies ist in den Entscheidungen begründet, die die Implementierung vereinfacht haben. So wurde auf ein geeignetes Cachingsystem zur effizienten Bearbeitung von serverseitigen *Events* verzichtet, wie in Unterabschnitt 4.2.2 Zusammenarbeit ermöglichende Serverseite beschrieben. Weiterhin werden *Ereignisse* mit Sequenznummern versehen, um Linearität der atomaren Operationen zu gewährleisten. Diese Nummer ist eine 32-Bit Ganzzahl und könnte irgendwann überlaufen. Diese Möglichkeit wurde in Unter-Unterabschnitt 4.3.4.4 Oberflächenerweiterung festgestellt. Natürlich ist dies erst der Fall, wenn 2147483648 *Events* ausgeführt wurden. Dann würden die Clients abstürzen, da sie ihre lokale Sequenznummer nicht mehr hochzählen können. Serverseitig könnten *Events* zwar noch hinzugefügt aber nicht mehr abgefragt werden. Angenommen, es würden jede Sekunde 10 *Ereignisse* ausgelöst, dann würde eine *piamind*-Sitzung nach 2485,5 Tagen nicht mehr funktionieren. Es ist allerdings wahrscheinlicher, dass vorher der Speicher des Servers verbraucht ist. Auch wenn diese Möglichkeit sehr unwahrscheinlich erscheint, muss sie dennoch in einem maßen tauglichen Produkt beachtet werden. Im Rahmen dieser Bachelorthesis reicht die Feststellung dieser Problematik aus. Eine andere Unbequemlichkeit bei der Nutzung des Editors besteht darin, dass sich angemeldet Benutzer nicht von einer Sitzung abmelden können. Um diese Sitzung zu verlassen, müssen sie, wie in Unter-Unterabschnitt 4.3.4.4 Oberflächenerweiterung beschrieben, einfach den Browser schließen oder eine andere Website besuchen. Dadurch wird der Nutzer aus der Sitzung allerdings nicht entfernt und es entsteht eine Leiche, die erst beim Schließen der Sitzung durch den Autor oder dem automatischen Reset des Applikationskontexts entfernt wird. Für Zusammenarbeit im Rahmen des Prototyps ist dies nicht so tragisch. Für die produktive Benutzung muss die Sitzungsverwaltung aber noch optimiert werden. Da die Entwicklung der gemeinschaftlichen Aspekte in einer testgetriebenen Weise vorgenommen wurde, kann die Auswirkung einer Änderung auf die bestehende Funktionalität jederzeit sofort überprüft werden. Dazu müssen lediglich alle Testfälle ausgeführt werden. Testfälle wurden in der *piamind*-Infrastruktur für die Komponenten der WCF-Dienste und dem *sEvent*-Framework erstellt. Diese zwei Bestandteile definieren die komplette gemeinschaftliche und administrative Funktionalität des Systems. Durch die Client-Bibliothek ist lediglich eine andere Zugriffsschicht mit vereinfachenden Aufrufen und WCF-Proxys dazugekommen. Theoretisch müsste diese Zwischenschicht auch getestet werden. Da es sich dabei aber nicht um Logik der Zusammenarbeit und Administration selbst, sondern nur um Integrationslogik handelt, ist hier das manuelle Testen durch einen Anwender vollkommen ausreichend.

4.4.1 Anforderungstreue

In Unterabschnitt 4.1.3 Zusammenfassung wurden alle Anforderungen an die gemeinschaftliche Mindmap-Modellierung aufgelistet. Dabei zeigt Tabelle 7 welche Anforderungen erfüllt wurden und welche nicht.

ID #	Anforderung	Erfüllt
K1	Online Zusammenarbeit	Ja
K2	Konfliktbehandlung	Ja
K3	Multiuserfähigkeit	Ja
K4	Dokumente auf Server speichern	Temporär
K5	Rechtesystem	Temporär
K6	Atomare Operationen ausführen	Ja
K7	Atomare Operationen aggregieren	Ja
K8	Atomare Operationen anwenden	Ja

Tabelle 7: Anforderungen mit Implementierungsstatus

Die Anforderung (K4) Dokumentenspeicher konnte nur proprietär adressiert werden, da zum Zeitpunkt der Entwicklung noch nicht feststand, ob und in welches System der Mindmap-Client integriert werden soll. Die in Unter-Unterabschnitt 4.3.4.3 Mindmap Verwaltung beschriebene Implementierung eines Dateisystems für die Verwendung im Internet, mit dem Namen *piafile*, kann jederzeit durch eine andere Komponente ersetzt werden. Da sich diese Erweiterung nur auf die Oberfläche und die Verwendung der bestehenden Import- und Export-Funktionalität bezieht, kann ein anderes Dateisystem zu einem späteren Zeitpunkt ohne Probleme integriert werden. Es ist also nicht erforderlich irgendwelche Komponenten der *piamind*-Infrastruktur anzupassen. Die zweite Anforderung, die nicht endgültig realisiert wurde, ist (K5) Rechtesystem. In dem implementierten Konzept der Sitzungsverwaltung besteht die Möglichkeit Benutzern entweder alle oder gar keine Rechte zu geben. Da auch hier zum Zeitpunkt der Entwicklung nicht geklärt werden konnte, welches bereits vorhandene Rechtesystem der CAS-Infrastruktur verwendet werden soll, wurde diesbezüglich ein proprietäres Rechtesystem implementiert. Diese Lösung ist für den Prototyp vollkommen ausreichend. Sollten irgendwann granularer Rechte benötigt werden, müssen diese entsprechend festgelegt und den Teilnehmern propagiert werden können. Durch das in den Unter-Unterabschnitten 4.2.1.2 Zusammenarbeit beitreten und 4.2.1.3 Teilnehmer der Zusammenarbeit verwalten vorgestellten Sitzungsgeheimnis, werden der Status der Sitzung sowie die aktuelle Eventsequenznummer an Benutzer übertragen. Im Falle von besonderen Berechtigungen des Nutzers, könnte der Autor diese entsprechend auch übertragen. Der Client müsste sich dann an diese Vorgaben halten. Eine solche Implementierung würde mit relativ geringem Aufwand durchgeführt werden können, solange sich die Rechte weiterhin nur auf eine Sitzung beziehen. Wenn das Rechtesystem in Bezug auf die WCF-Dienst erweitert werden müsste, dann müsste sichergestellt werden, dass nur der Autor die in Unter-Unterabschnitt 4.2.1.3 Teilnehmer der Zusammenarbeit verwalten vorgestellten Dienstoperationen aufrufen kann. Dazu müsste ein geheimer Schlüssel beim Erzeugen der Sitzung an den Autor übertragen werden, der bei jeder sitzungsverwaltenden Operation mit übergeben werden müsste. Nur wenn der richtige Schlüssel auf dem Server ankommt, würde die Operation als gültig gelten und ausgeführt. Damit könnte verhindert werden, dass Teilnehmer der Zusammenarbeit sich selbst Rechte an einer Sitzung geben, die durch eine andere Person erzeugt wurde. Allerdings müssten diese dazu einen eigenen *piamind*-Client verwenden, der die WCF-Dienste direkt ansprechen kann. Bei der Verwendung des *MindGraph*-Clients ist dies nicht möglich, sodass dieser Sicherheitsaspekt im Rahmen des unproduktiven Einsatzes nicht weiter beachtet werden brauch. Abgesehen von den zwei Anforderungen, die nicht endgültig umgesetzt

wurden, was hiermit begründet wurde, kann das Ergebnis der ersten Iteration in Bezug auf die Realisierung der Anforderungen als erfolgreich angesehen werden.

4.4.2 Zeitlicher Ablauf

In diesem Unterabschnitt wird das Ergebnis des ersten Teils dieser Thesis in Bezug auf die Bearbeitungszeit bewertet. In Unterabschnitt 3.3.1 Iteration 1 (Zusammenarbeit) wurde die erste Iteration zur Entwicklung der gemeinschaftlichen Aspekte zeitlich geplant. In Tabelle 8 wird diese Planung mit dem tatsächlichen Verlauf verglichen.

Tätigkeit	**Geplant**	**Tatsächlich**
Infrastruktur für online Zusammenarbeit konzipieren	1 Woche	1 Woche
Infrastruktur für online Zusammenarbeit implementieren	2 Wochen	1,5 Wochen
Editorerweiterung für online Zusammenarbeit vorbereiten	1 Woche	1 Woche (*sEvent*-Framework)
		0,5 Wochen (Client-Bibliotek)
Editorerweiterung für online Zusammenarbeit integrieren	1 Woche	0,5 Wochen (Oberflächenerweiterung)
		1 Woche (Integration aller Komponenten)
	5 Wochen	5,5 Wochen

Tabelle 8: Arbeitspakete der Iteration 1 verglichen mit tatsächlichem Zeitaufwand

Dabei ist zu erkennen, dass die Implementierung der Infrastruktur eine halbe Woche schneller abgeschlossen werden konnte. Dies ist der Fall, da WCF eine einfache Entwicklungsmöglichkeit für Serverdienste im Rahmen von ASP.NET darstellt. Die Editorerweiterung dagegen, hat zwei halbe Wochen länger gedauert als ursprünglich geplant. Dies ist einerseits auf die Entwicklung einer eigenen *Ereignis*-Bibliothek zurückzuführen. Wäre dagegen eine bereits existierende Bibliothek verwendet worden, dann hätte die Verwendung eventuell schneller abgeschlossen werden können. Allerdings hätte dann eine sicherlich notwendige intensive Beschäftigung mit einem solchen Framework wesentlich mehr Zeit gekostet. Außerdem hätte die Integration an die verfügbare Infrastruktur die Implementierung der Integration erschwert. Diese Integration von Diensten, *Ereignis*-Framework und Oberfläche hat am Ende die Iteration um die andere halbe Woche verlängert. Dies ist durch die aufwendige Verbindung der Eventerzeugung mit der bestehenden Struktur des Mindmap-Editors zu begründen. Da der geplante Zeitraum der ersten Iteration durch genannte Punkte nur um eine halbe Woche überschritten wurde, handelt es sich auch aus zeitlicher Sicht um ein erfolgreiches Ergebnis.

Der Zusammenarbeit ermöglichende *MindGraph*-Editor kann unter http://casrd.de/rd/piamind2/ aufgerufen und benutzt werden.

5 Domänenmodellierung

Der zweite Teil dieser Bachelor-Thesis besteht aus der Konzeptionierung und Entwicklung einer Lösung zur Modellierung von Objekten einer CRM-Domäne mit dem bereits erwähnten Mindmap-Editor. Dafür werden zunächst die Anforderungen konkretisiert.

5.1 Anforderungen

In Unterabschnitt 3.2.2 CRM-Objekte in Mindmaps wurden die allgemeinen Anforderungen an einen Mindmap-Editor beschrieben, der das Modellieren von Domänenobjekten ermöglich soll. Die Anforderungen (D1) Domänenknoten und (D3) Erweiterbarkeit beziehen sich dabei auf die Möglichkeit spezielle *Knoten* erzeugen zu können, die sich zwar wie normale *Knoten* verwenden lassen, aber ein eigenes Verhalten und Aussehen besitzen können. Die Anforderung (D2) Kontextabhängigkeit bezieht sich auf das Verhalten dieser Knoten und wird in Unterabschnitt 5.1.2 Repräsentierende Knoten beschrieben. Zuvor wird die Möglichkeit beschrieben, die das Aussehen dieser *Knoten* betrifft.

5.1.1 Knoten mit beliebigem Inhalt

Das wohl wichtigste Merkmal eines *Knotens* als Objekt einer Domäne ist das Aussehen. Obwohl Domänenknoten sich in diesem Merkmal unterscheiden können, müssen sie sich wie alle anderen *Knoten* verhalten, wenn sie modelliert werden. Damit ist es erforderlich, das Aussehen des *Knotens* und dessen Modellierungsverhalten zu trennen. Da nach Anforderung (D3) Erweiterbarkeit verschiedene *Knoten* verschiedene Oberflächen haben müssen, wird eine Art Container benötigt, der selbst ein *Knoten* ist und sich wie einer verhält, aber eine austauschbare Oberfläche besitzt. Die Erweiterbarkeit könnte auch erreicht werden, indem man für jedes Domänenobjekt einen neuen *Knoten* erstellt. Dies würde bei vielen Knotentypen aber das nachträgliche Verändern von allgemeinem Verhalten deutlich erschweren, da in einem solchen Fall jeder dieser *Knoten* verändert werden müsste, um den neuen Anforderungen gerecht zu werden. Bei einem Container-*Knoten* würden neue Abhängigkeiten alle von dem eigentlichen Aussehen des Domänenobjekts getrennt.

5.1.2 Repräsentierende Knoten

Ein spezieller *Knoten* besitzt unabhängig von seinem Aussehen zwei Arten von Verhalten, die im oberen Unterabschnitt bereits erwähnt wurden. Das erste Verhalten ist allgemeines Verhalten, oder auch Modellierungsverhalten genannt, dass alle *Knoten* gemeinsam haben. Das zweite Verhalten ist spezifisch und wird von dem Domänenobjekt selbst bestimmt. Dafür ist entweder die Oberfläche verantwortlich, oder ein externes Ereignis. Abhängig davon wie die Ereignisse erwartet und behandelt werden, lässt sich ein Kontext verwenden, der durch die Einheit der Domäne bestimmt wird, die von dem entsprechenden *Knoten* repräsentiert wird. Externe Ereignisse können entweder durch die Domäne oder durch die Verwendung des *Knotens* ausgelöst werden. Die Verwendung des *Knotens* in der Mindmap bezieht sich auf die Topologie. Abhängig davon, welche *Knoten* mit dem Domänenknoten verbunden sind, kann sich das Verhalten des *Knotens* anpassen.

5.1.3 Zusammenfassung

Aus den allgemeinen Anforderungen aus Unterabschnitt 3.2.2 CRM-Objekte in Mindmaps ergeben sich drei zusätzliche spezifische Anforderungen, die im Folgenden zusammengefasst werden.

(D4) Knotentypen erzeugen

Der in Unterabschnitt 5.1.1 Knoten mit beliebigem Inhalt beschriebene Container-*Knoten* muss in der Mindmap erstellt werden können. Dabei muss bei der Erstellung die Oberfläche und das spezifische

Verhalten des Domänenobjekts an diesen *Knoten* gebunden werden. Welcher Domänenknoten erzeugt werden soll, muss über die Oberfläche des Mindmap-Editors ausgewählt werden können.

(D5) Knoten beliebig verknüpfen
Da Domänenknoten ihr spezifisches Verhalten von den *Knoten* abhängig machen können, mit denen sie verbunden sind, müssen repräsentierende *Knoten* auf alle Elemente zugreifen können, mit denen sie verbunden sind. In einer Mindmap können keine Netze entstehen, da sie in einer Baumstruktur aufgebaut sind, wie in Unter-Unterabschnitt 2.2.1.2 Abgrenzung zu Graphen beschrieben. Domänenobjekte dagegen können eine theoretisch unbestimmte Menge an Verknüpfungen zu beliebigen anderen Objekten herstellen. Somit müssen Domänenknoten als Graph modelliert werden können. Der Mindmap-Editor muss also um eine Möglichkeit ergänzt werden, beliebige Verknüpfungen zwischen *Knoten* herzustellen.

(D6) Beispielhafte CRM-Knoten verwenden
Der erweiterte Mindmap-Editor muss die Möglichkeit bieten, Domänenobjekte zu definieren und als *Knoten* zu verwenden. In Unterabschnitt 2.2.3 CRM-Domäne wurden beispielhaft Einheiten aus einer CAS-Domäne aufgezählt. Diese Objekte müssen standardmäßig von *MindGraph* unterstützt werden und als *Knoten* in Mindmaps verwendet werden können. Dabei ist es nicht erforderlich, dass diese Objekte über eine externe Datenanbindung verfügen.

Die aktuellen Anforderungen sind in Tabelle 9 zusammen mit Priorität und betreffendem System aufgelistet, wobei A die höchste Priorität darstellt. Das betreffende System ist überall der Client, da das Hintergrundsystem für die Zusammenarbeit nicht betroffen ist.

ID #	Anforderung	Betreffendes System	Priorität
D1	Domänenknoten	Client	A
D2	Kontextabhängigkeit	Client	B
D3	Erweiterbarkeit	Client	A
D4	Knotentypen erzeugen	Client	A
D5	*Knoten* beliebig verknüpfen	Client	B
D6	Beispielhafte CRM-*Knoten* verwenden	Client	C

Tabelle 9: Anforderungen an Unterstützung von Domänenobjekten in *MindGraph*

5.2 Entwurf

Nachdem im vorherigen Abschnitt die Anforderungen an die Modellierung von speziellen *Knoten* in Mindmaps zusammengefasst wurden, wird die Erweiterung des Editors in diesem Abschnitt entsprechend geplant. Wie in Unterabschnitt 3.2.3 Use-Cases beschrieben, werden zwei Anwendungsfälle in diesem Abschnitt beschrieben. Um den Use-Case (UC5) Verwenden von speziellen Knoten realisieren zu können, wird zunächst der Aufbau eines erweiterten *Knotens* behandelt. Anschließend wird untersucht, wie *Knoten* im Rahmen des Use-Case (UC6) Verbinden von Knoten beliebig verbunden werden können, unabhängig von der Topologie einer Mindmap. Abschließend wird in diesem Abschnitt geplant, wie beispielhafte CRM-Objekte als *Knoten* verwendet werden können.

5.2.1 Erweiterte Knoten verwenden

In diesem Unterabschnitt wird die Planung des Use-Case (UC5) Verwenden von speziellen Knoten beschrieben. In Unterabschnitt 5.1.1 Knoten mit beliebigem Inhalt wurde bereits erwähnt, dass es *Knoten* in der Mindmap geben muss, die beliebige Inhalte haben können. Nur so können beliebige Domänenobjekte mit beliebigen Oberflächen verwendet werden. Die *Knoten*, die der Mindmap-

Editor bereits erstellen kann, wurden in Unterabschnitt 3.1.1 Visuelle Objekte vorgestellt. Dort wurde außerdem die Vererbungshierarchie von *Knoten* beschrieben, in die sich neue *Knoten* integrieren müssen, um in einer Mindmap verwendet werden zu können. *MindGraph* unterstützt momentan zwei Knotentypen. Der erste ist ein nicht mehr verwendeter Testknoten (`SimpleItem`) und der zweite ist der standardmäßige Textknoten (`Node`). Beide erben von der abstrakten Klasse `BasicItem`, die grundlegende Funktionalität bietet, die für alle Konten in der Mindmap gelten soll. Neue Knotentypen müssten also ebenfalls von dieser abstrakten Klasse ableiten, wie das Diagramm in Abbildung 41 gezeigt.

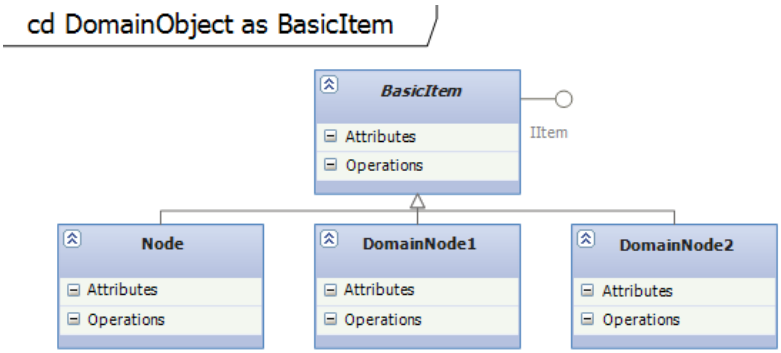


Abbildung 41: Domänenobjekte als *Knoten*

Bei vielen Domänenobjekten würde diese Lösung das allgemeine Verhalten dieser Objekte in allen neuen Klassen spezifizieren. Sollte sich etwas an der Art und Weise ändern, wie der Mindmap-Editor mit *Knoten* umgeht, dann müssten alle neuen Klassen angepasst werden. Langfristig ist dieser Ansatz also mit viel Wartungsaufwand verbunden und steht damit der Anforderung (D3) Erweiterbarkeit entgegen, die eine einfache Erweiterbarkeit voraussetzt. Stattdessen ist ein in Unterabschnitt 5.1.1 Knoten mit beliebigem Inhalt bereits erwähnter Container erforderlich, der auf der einen Seite ein *Knoten* und auf der anderen ein Adapter für beliebigen Inhalt ist. Die daraus resultierende Struktur ist im Klassendiagramm in Abbildung 42 abgebildet.

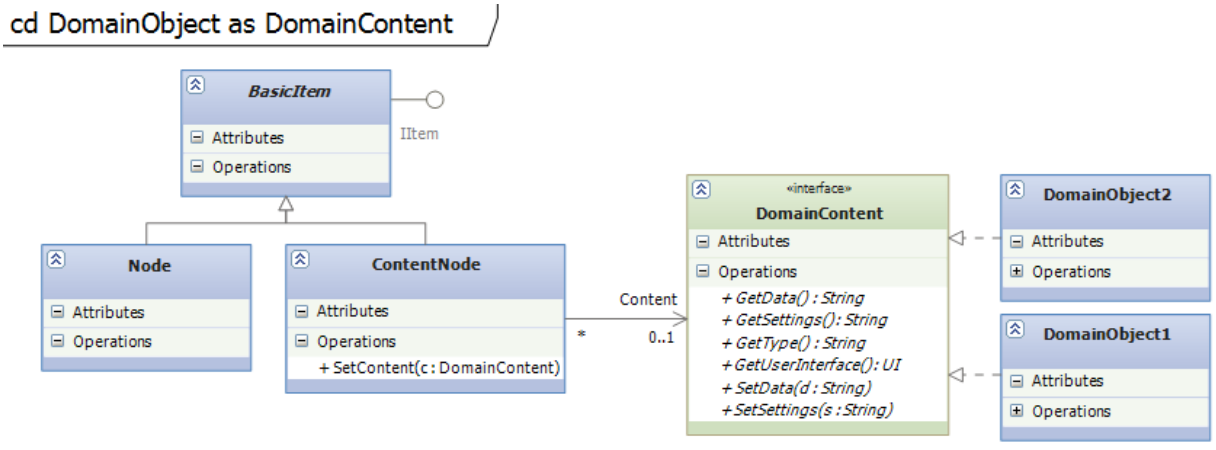


Abbildung 42: Domänenobjekte als *Knoteninhalte*

Mit einem solchen Aufbau kann das allgemeine Verhalten der Domänenknoten von deren Aussehen und dem spezifischen Verhalten getrennt werden. Eine einfache Erweiterbarkeit kann dann unabhängig von konkreten *Knoten* vorgenommen werden, solange Domänenobjekte als *Knoteninhalt* behandelt werden können, was im oberen Diagramm als Schnittstelle `DomainContent` dargestellt ist.

5.2.1.1 Knoteninhalt

Als *Knoteninhalt* werden alle Objekte bezeichnet, die `DomainContent` implementieren. In diesem Unter-Unterabschnitt soll die Schnittstelle `DomainContent` geplant werden. Da ein Domänenobjekt eine eigene Oberfläche haben kann, muss diese über die Schnittstelle erreicht werden können. Dafür erhält diese die Operation `GetUserInterface`, welche eine Referenz auf die Oberfläche zurückliefert. Sämtliches spezifische Verhalten kann über diese Oberfläche ausgelöst werden, unabhängig davon, dass sich diese Oberfläche in einem *Knoten* befinden wird. Zusätzlich besteht jeder Inhalt aus Einstellungen und Daten, die bei der Serialisierung und Deserialisierung von *Knoten* als Zeichenketten verwendet werden, um den Zustand des Domänenobjekts wiederherstellen zu können. `DomainContent` benötigt dafür entsprechende Get- und Set-Methoden. Um die Knoteninhaltstypen unterscheiden zu können, benötigt jeder Inhalt eine eindeutige Typkennung, die ebenfalls über die Schnittstelle abgefragt werden kann. Damit ist ausreichend festgelegt, welche Operationen Knoteninhaltstypen implementieren müssen. Die `DomainContent`-Schnittstelle aus dem Diagramm in Abbildung 42 zeigt diese Methoden. Dabei ist den Knoteninhaltstypen völlig selbst überlassen, wie sie aufgebaut sind und wie sie ihre Oberfläche von ihrem Verhalten trennen. Dadurch kann eine sehr hohe Flexibilität erreicht werden, die bereits bei der Entwicklung von speziellen Inhalten beginnt. Durch die Trennung durch das Interface können spezielle *Knoten* unabhängig voneinander entwickelt und verwendet werden.

5.2.1.2 Knoteninhalt-Knoten

Nachdem der *Knoteninhalt* spezifiziert wurde, muss betrachtet werden, wie diese Inhalte den *Knoten* zugeordnet werden können. Der Container-*Knoten* aus dem oberen Diagramm muss dafür eine spezielle `SetContent`-Methode bereitstellen. Als Parameter erhält diese Methode ein Objekt das `DomainContent` implementiert und stellt dessen Oberfläche dann auf dem *Knoten* dar. Damit könnte sogar der gleiche *Knoteninhalt* auf mehreren *Knoten* dargestellt werden. Wenn ein Container-*Knoten* erstellt wird, dann muss ihm ein Inhalt übergeben werden. Dabei stellt sich die Frage nach der Herkunft dieser Inhalte. Es ist also eine Fabrik erforderlich, die bei Bedarf einen Container-*Knoten* erzeugt und ihm einen *Knoteninhalt* zuweist, der ebenfalls erzeugt wird. Welcher *Knoteninhalt* erzeugt werden soll, kann der Fabrik über die in Unter-Unterabschnitt 5.2.1.1 Knoteninhalt beschriebene Typkennung mitgeteilt werden. Dafür muss die Fabrik alle Inhalte kennen, beziehungsweise muss eine Möglichkeit bereitstellen, wie sich neue Typen registrieren können. Damit diese Fabrik *Knoten* aller Knotentypen erzeugen kann, wird die leere Typkennung für den bereits existierenden Textknoten reserviert. Im Vergleich zu diesem Textknoten, ist der einzige Unterschied, dass der Textknoten eine Eingabemöglichkeit für Text enthält, während der *Knoten* mit beliebigem Inhalt die vom Inhalt definierte Oberfläche enthält.

5.2.2 Knotenverbindungen erstellen

In diesem Unterabschnitt wird die Planung des Use-Case (UC6) Verbinden von Knoten beschrieben. *Knoten* können jetzt beliebigen Inhalt anzeigen und sich auch speziell verhalten. Dieses Verhalten kann abhängig davon sein, wie die Oberfläche des Inhalts verwendet wird. Es besteht aber noch keine Möglichkeit auf das Umfeld, also mit welchen anderen *Knoten* der spezielle *Knoten* verbunden ist, zu reagieren. In diesem Unterabschnitt soll festgelegt werden, wie Knotenverbindung in Mindmaps realisiert werden können. In Unterabschnitt 3.1.2 Topologische Objekte wurde die Struktur von Knotenbeziehungen in *MindGraph* vorgestellt. Wie auch in Unter-Unterabschnitt 2.2.1.2 Abgrenzung zu Graphen beschrieben, sind die Verbindungen von *Knoten* in einer Mindmap als Kanten in einem baumartigen Graphen zu sehen. Dies sind die primären Verbindungen von *Knoten*.

Domänenobjekte dagegen, besitzen nicht näher definierte Abhängigkeiten und können daher nicht pauschal auf Bäume eingeschränkt werden. Diese beliebigen Verbindungen zwischen Objekten müssen auch in Mindmaps modelliert werden können, zusätzlich zu den primären Verbindungen der Mindmap-Topologie. Dies sind die sekundären Verbindungen von *Knoten*.

5.2.2.1 Primäre Knotenverbindungen

Es gilt zu klären, wie primäre Knotenverknüpfungen von Domänenobjekten verwendet werden können. In Unterabschnitt 3.1.1 Visuelle Objekte wurde die abstrakte Oberklasse `BasicItem` eingeführt, die das Interface `IItem` implementiert. Dadurch wird eine abstrakte Sicht auf *Knoten* in der Mindmap ermöglicht. In Unterabschnitt 3.1.2 Topologische Objekte wurden zwei Assoziationen vorgestellt, die die primären Verknüpfungen von *Knoten* ausmachen. Zum einen besteht eine Referenz auf die Knotengruppe. Durch diese Gruppe kann der Oberknoten ermittelt werden. Die zweite Assoziation geht ebenfalls über die Knotengruppe und verbindet alle Unterknoten des betrachteten *Knotens*. Da diese Assoziationen vorhanden sind, müssen sie geeignet zur Verfügung gestellt werden, sodass *Knoteninhalte* darauf zugreifen können. Dies erfolgt über einen speziellen Kontext, der zusätzliche Referenzen auf die primären Verbindungen kapselt und von dem speziellen *Knoten* an dessen Inhalt übergeben werden kann, wie im Klassendiagramm in Abbildung 43 dargestellt. Die Struktur von `IItem`, `ItemGroup` und `ChildArea` entspricht der aus Unterabschnitt 3.1.2 Topologische Objekte.

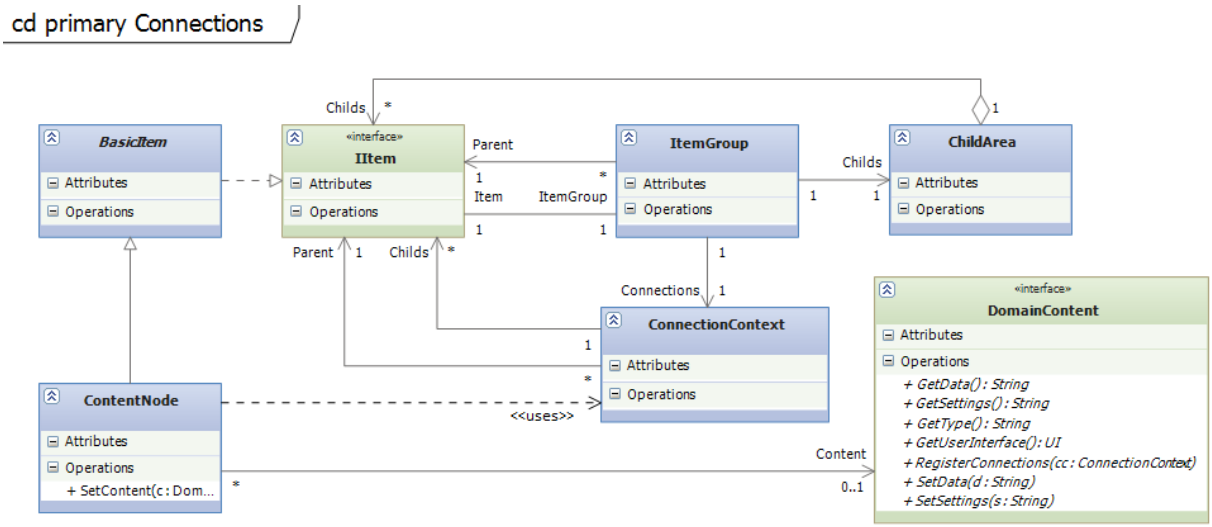


Abbildung 43: Integration von Verbindungs-Objekt und *Knoteninhalt*

Dafür erhält die `DomainContent`-Schnittstelle eine `RegisterConnections`-Methode, über die einmalig eine Referenz auf das `ConnectionContext`-Objekt des Container-*Knotens* übergeben wird. Der *Knoteninhalt* kann diese Referenz speichern und über diese beliebig auf die primären Verbindungen des *Knotens* zugreifen, der ihn als Inhalt enthält.

5.2.2.2 Sekundäre Knotenverbindungen

Domänenknoten haben somit die Möglichkeit, auf ihre Topologie zu reagieren. Wie beliebige Verbindungen zwischen *Knoten* erstellt und verwaltet werden können, wird in diesem Unter-Unterabschnitt behandelt. Angenommen es sind zwei *Knoten* vorhanden, die miteinander verbunden werden sollen. In Unterabschnitt 3.1.1 Visuelle Objekte wurden die Punkte auf einem *Knoten* schematisch dargestellt, die als Andockpunkte verwendet werden können. Das Andockverhalten ist in der abstrakten Klasse `Dockable` definiert und kann somit in allen Objekten, die von `BasicItem`

ableiten, verwendet werden. Da auch die Sammlung an Verbindungslinien vorgestellt wurde, müssen diese Linien nur an die Punkte auf den *Knoten* gebunden werden, genau wie dies bei der topologischen Verknüpfung von *Knoten* der Fall ist. Der Vorgang der Verbindung wird dabei von einem dedizierten Objekt vorgenommen, wie im Sequenzdiagramm in Abbildung 44 abgebildet.

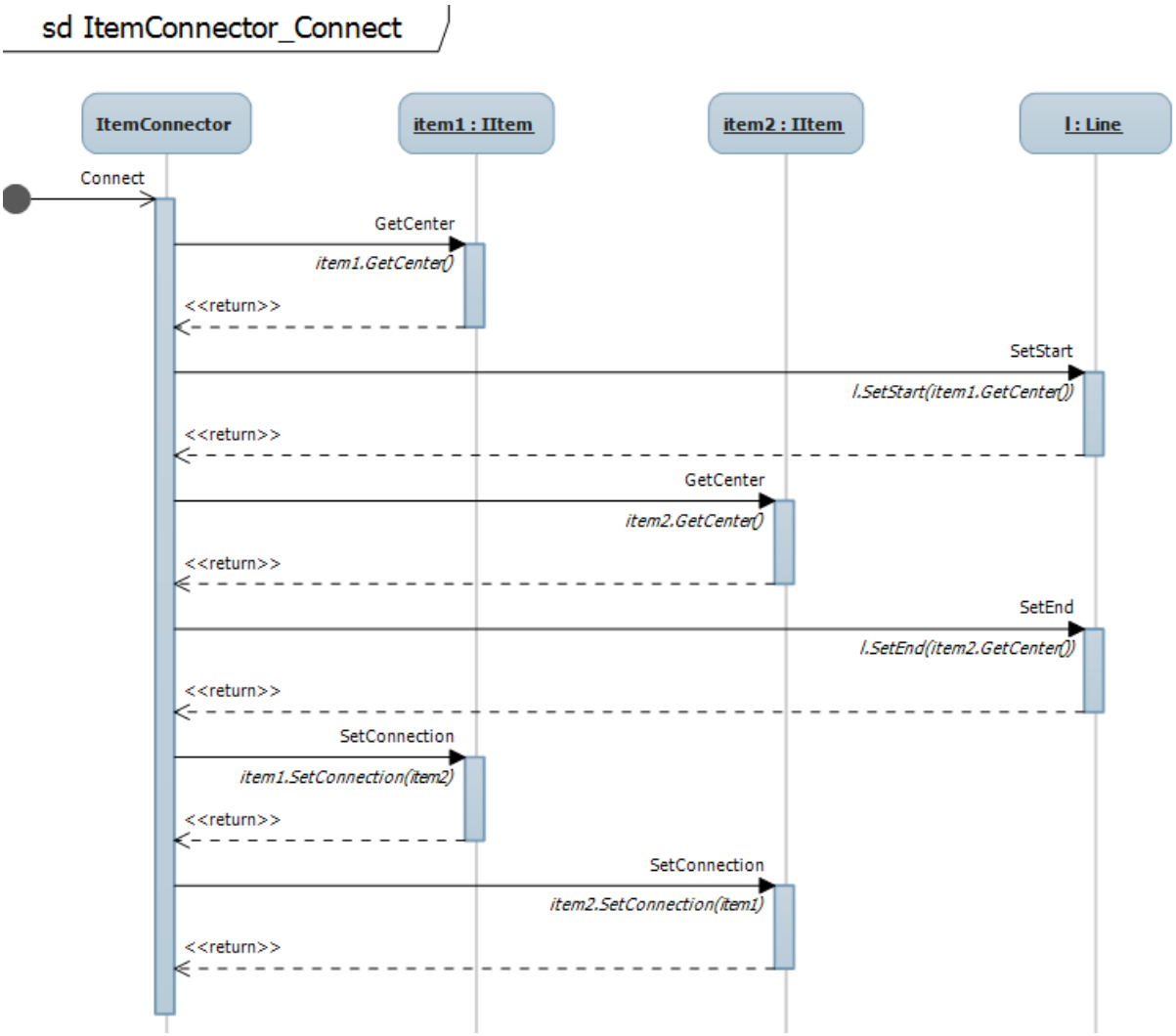


Abbildung 44: Ablauf des Vorgangs der Verbindung zweier *Knoten*

Neben der Verbindung der Linie mit den mittleren Andockpunkten der *Knoten* um die Darstellung zu erzeugen, wird die Verbindung auch logisch hergestellt, indem dem Knotenobjekt selbst die Referenz des *Knotens* übergeben wird, mit dem die Verknüpfung hergestellt wurde. Diese Referenz muss von dem *Knoten* gespeichert werden. Sollte die Verbindung aufgelöst werden, müssen die Linie und die Referenzen entsprechend wieder entfernt werden. Zur Verwaltung von topologischen Verknüpfungen wurde in Unter-Unterabschnitt 5.2.2.1 Primäre Knotenverbindungen bereits ein spezieller Kontext eingeführt. Dieses Objekt eignet sich auch zur Verwendung mit nicht topologischen Verknüpfungen, da seine Integration in die `DomainContent`-Schnittstelle bereits geplant wurde. Da die *Knoten* über die Knotengruppe auch die Referenz auf diesen Kontext erreichen können, müssen sie lediglich eine eigene Art des Zugriffs nutzen, der von den primären Verknüpfungen unterschieden werden kann. Die `ConnectionContext`-Klasse erhält dafür eine Liste mit Referenzen auf alle *Knoten*, zu denen eine Verbindung besteht. Da jeder *Knoten* sein eigenes `ConnectionKontext`-Objekt besitzt, können somit beliebige sekundäre Verbindungen verwaltet und *Knoteninhalten* zur Verfügung gestellt werden.

5.2.3 CRM-Module

Dass *Knoten* in der Mindmap beliebige Inhalte haben und beliebig verknüpft werden können, wurde in den vorherigen Unterabschnitten beschrieben. In diesem Unterabschnitt werden diese neuen Möglichkeiten genutzt, um beispielhafte CRM-Inhalte zu konzipieren, die als *Knoteninhalte* in der Mindmap verwendet werden können. Nach Anforderung (D6) Beispielhafte CRM-Knoten verwenden sollen die CRM-Objekte, die in Unterabschnitt 2.2.3 CRM-Domäne vorgestellt wurde, implementiert werden. Dabei handelt es sich um "User", "Address", "Calendar", "Note" und "Document". Note entspricht dem bereits vorhandenen Textknoten und muss daher nicht weiter beachtet werden, da dessen Verwendung ohne speziellen Knoteninhaltstyp bereits möglich ist. Es ist wichtig zu beachten, dass die Abstraktionsstufen von den Knotentypen selbst festgelegt werden können. Das Konzept des *Knoteninhalts* stellt nur eine Umgebung für beliebige Inhalte bereit. Was diese *Knoten* bedeuten, muss vom Entwickler des entsprechenden *Knoteninhalts* selbst definiert werden.

5.2.3.1 User-Knoten

Als erster Knotentyp soll ein Benutzerknoten betrachtet werden. Benutzer haben hauptsächlich eine Kennung und einen Namen. Da Benutzer auch Personen sind, kann der Benutzerknoten durch einen allgemeinen Personenknoten mit einer Rolle ersetzt werden. Im Grunde können beliebige Attribute verwendet werden, um eine Person darzustellen. Für einen beispielhaften Knoteninhaltstyp werden aber nur Kennung, Name, Rolle und ein Foto verwendet. Die Datenanbindung soll über eine Datenhaltung im Speicher erfolgen.

Zur Demonstration der Interaktionsmöglichkeiten soll der Personenknoten eine kontextabhängige Abbildung seiner selbst erzeugen können, wenn dies von dem modellierenden Benutzer angefordert wird. Da die Idee dieses *Knotens* lediglich ist, als Beispiel für frei definierbare *Knoten* angesehen zu werden, genügt dieser Aufbau völlig. Was die Bedeutung eines Personenknotens betrifft, so repräsentiert ein solcher *Knoten* nicht direkt eine Person. Stattdessen steht ein Personenknoten für eine Repräsentation einer Repräsentation einer Person. Durch diese zusätzliche Indirektion können mehrere Personenknoten die gleiche Personrepräsentation aus dem Speicher vertreten.

5.2.3.2 Address-Knoten

Eine Adresse kann ebenfalls als *Knoten* verwendet werden. Sie besteht aus einer einfachen Anschrift und kann somit zum Beispiel zur Erweiterung von Personenknoten genutzt werden. Die Anschrift enthält Straße, Hausnummer, Stadt mit Postleitzahl und Länderinformation und soll genau wie Personen aus einer Datenbank im Speicher zur Verfügung gestellt werden. Dabei repräsentiert ein Adressknoten eine Repräsentation einer Adresse.

5.2.3.3 Calendar-Knoten

Ein Kalender ist eine Sammlung von Terminen. Daher ist es sinnvoller Kalender als verbundene Terminknoten zu sehen, als einen globalen Kalenderknoten mit einer integrierten Liste. In der Darstellung als Terminknoten können wesentlich granularer Abhängigkeiten ausgedrückt werden, indem Termine gezielt mit Personen- oder Adressknoten verbunden werden. Ein Termin besteht dabei aus einer Bezeichnung und einem Datum. Punkte einer Agenda können als Notizen mit Textknoten hinzugefügt werden. Terminknoten repräsentieren Repräsentationen von Terminen. Soll ein neuer Termin erstellt werden, muss der Terminknoten dies unterstützen und die Repräsentation des Termins sowie den Termin selbst erzeugen können.

5.2.3.4 Document-Knoten

Um ein Dokument aus der Datenbank im Speicher referenzieren zu können, kann ein spezieller Dokumentknoten verwendet werden. Wenn das Dokument im Speicher direkt vorhanden ist, kann der Dokumentknoten eine direkte Repräsentation des Dokuments darstellen. Wenn das Dokument im Speicher selbst nur eine Repräsentation ist, repräsentiert ein Dokumentknoten diese Repräsentation. Dokumentknoten können beliebig mit Personen, Terminen und allen anderen *Knoten* verknüpft werden. Dabei besteht ein Dokumentknoten aus einem Namen und eventuell aus weiterführenden Informationen zum Inhalt.

5.2.3.5 Beispielhafte Datenhaltung

In Unter-Unterabschnitt 5.2.3.3 Calendar-Knoten wurde bereits erwähnt, dass die Datenanbindung der vorgestellten Knotentypen auch veränderbar gestaltet werden kann. So können zum Beispiel neu erstellte *Knoten* neue Repräsentationen in der Datenbank erzeugen. In einer produktiven Umgebung könnten die *Knoten* über direkte Referenzen auf Datenzugriffsschichten verfügen. Um die vorgestellte Beispielsdomäne zu entwickeln, wird eine beispielhafte Datenbank im Speicher gehalten. Diese Datenbank verfügt über keine Assoziationen sondern stellt lediglich eine Sammlung von Personen, Adressen, Terminen und Dokumenten dar. Sämtliche Verbindungen werden auf der Ebene der Mindmap erstellt. Daten aus einer produktiven Datenquelle könnten dagegen durchaus Assoziationen besitzen. Das sich diese Assoziationen auf die Mindmap auswirken können sollen, wurde in keiner Anforderung festgehalten und muss daher in dieser Iteration nicht weiter berücksichtigt werden. Die Anforderung (D2) Kontextabhängigkeit verlangt lediglich das Vorhandensein einer direkten Datenbindung des *Knotens* an die Quelleneinheit. Das Erzeugen einer Mindmap aus einer bestehenden verbundenen Datenquelle kann in einer zukünftigen Version des Mindmap-Editors implementiert werden, wenn ebenfalls Anforderungen an konkrete Domänenobjekte vorhanden sind. Im Rahmen der Anforderung (D6) Beispielhafte CRM-Knoten verwenden genügt das beschriebene Verhalten der Datenhaltung im Speicher völlig. Da es sich bei diesen Beispieldaten nur um Testdaten mit zufälligen Werten handelt, ist eine Beschreibung nicht sinnvoll und wird daher weggelassen.

5.3 Implementierung

In Abschnitt 5.2 Entwurf wurde behandelt, wie Modellierung von Domänenobjekten ermöglicht werden kann. Dazu wurde zunächst geplant, wie eigene *Knoten* in der Mindmap genutzt werden können und wie diese aufgebaut sein müssen. Dabei wurde auf die Trennung von *Knoten* und *Knoteninhalten* eingegangen. Auf der Seite der *Knoten* wurde zusätzlich behandelt, wie *Knoten* unabhängig von ihrer Topologie verbunden werden können. In diesem Abschnitt wird beschrieben, wie diese Strukturen und Funktionen implementiert sind. Dabei wird zunächst auf die Implementierung der neuen Knotenstruktur eingegangen. Anschließend wird die Realisierung von Knotenverbindungen und deren Erstellung dargestellt, bevor die neu eingeführten CRM-*Knoten* aus einer technischen Perspektive vorgestellt werden. Abschließend wird gezeigt, wie die bestehenden Komponenten angepasst werden müssen, um weiterhin Zusammenarbeit zu unterstützen und um die neuen Funktionen über die Oberfläche des Editors verwenden zu können.

5.3.1 Knotenfabrik

Da Knoteninhaltstypen unabhängig von den *Knoten* entwickelt werden können, wie in Unterabschnitt 5.2.1 Erweiterte Knoten verwenden festgestellt wurde, muss auch deren Erzeugung getrennt durchgeführt werden. Durch das gemeinsame Interface `IContentModule` für alle Knoteninhaltstypen, kann die Erzeugung von der Einheit übernommen werden, die diese Inhalte zur

Verfügung stellt. Wenn die Inhalte erzeugt wurden, können sie in einen Container-*Knoten* eingesetzt werden. Diese Container-*Knoten* sind Objekte der Klasse `ContentNode`. Genau wie die bisherigen Textknoten, handelt es sich bei `ContentNode` um eine Implementierung des `IItem`-Interfaces. Das Klassendiagramm in Abbildung 45 zeigt die Umgebung der Container-*Knoten*.

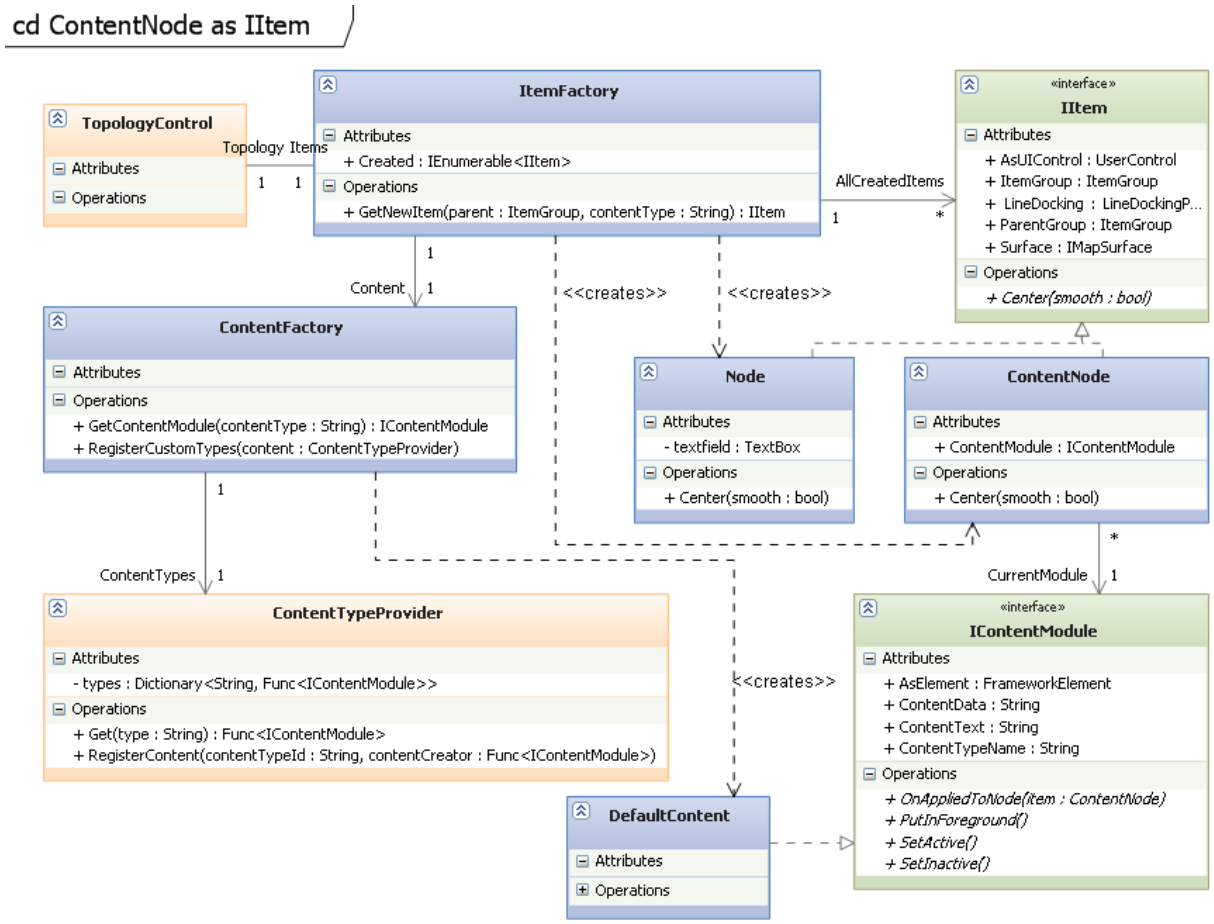


Abbildung 45: Struktur der ItemFactory mit der Erzeugung von ContentNode- und IContentModule-Objekten

`Node` und `ContentNode` implementieren nicht direkt das Interface `IItem`, sondern erben von der abstrakten Klasse `BasicItem`. Diese Oberklasse implementiert `IItem`, wurde aber zur besseren Übersicht weggelassen, ohne der Bedeutung der Vererbungshierarchie zu schaden. Die Erzeugung der beiden Knotentypen `Node` und `ContentNode` wird durch ein Objekt vom Typ `ItemFactory` durchgeführt. Diese Knotenfabrik ist über ein zentrales Objekt vom Typ `TopologyControl` erreichbar. *Knoteninhalte* für `ContentNode` werden von einem Objekt vom Typ `ContentFactory` erzeugt. Die erzeugte Instanz wird als `IContentModule` an `ItemFactory` zurückgegeben und dort in das erzeugte `ContentNode`-Objekt eingesetzt. `ContentFactory` kennt die zu erzeugenden Objekte nicht direkt, sondern besitzt eine Referenz auf ein `ContentTypeProvider`-Objekt. Bei diesem `ContentTypeProvider` können Einheiten, die *Knoteninhalte* zur Verfügung stellen wollen, eine Kombination aus Factory-Methode und eindeutiger Inhaltskennung registrieren. Das Objekt vom Typ `ContentFactory` braucht dann nur noch die Kennung des zu erzeugenden *Knoteninhalts* wissen, um die entsprechende Factory-Methode von `ContentTypeProvider` zu erhalten. Um einen *Knoten* mit der Knotenfabrik `ItemFactory` zu erzeugen, muss spezifiziert werden, welchen Inhalt der neue *Knoten* haben soll. Dadurch kann die Entscheidung über einen konkreten Inhalt von der Erzeugung der Inhalte getrennt werden.

Die Erzeugung geschieht dabei so, wie in Codeausschnitt 8 aus der Klasse `ItemFactory` dargestellt. Dass *Knoten* nur als Unterknoten eines bereits bestehenden *Knoten* erzeugt werden können wird an dieser Stelle nicht weiter beachtet. Handelt es sich bei der übergebenen Inhaltskennung um einen leeren String oder um einen nicht vorhanden String, wird ein normaler Textknoten erzeugt, wie in Unter-Unterabschnitt 5.2.1.2 Knoteninhalt-Knoten definiert. Da der Inhalt eines *Knotens* den Typ des *Knoten* ausmachen, können die Bezeichnungen Knotentyp und Inhaltskennung als Synonyme verwendet werden.

```
public IItem GetNewItem(ItemGroup parent, string contentType)
{
      IItem newNode;
      if (!string.IsNullOrEmpty(contentType))
      {
            newNode = this.GetContentNode(parent, contentType);
      }
      else
      {
            newNode = this.GetStandardNode(parent);
      }
      this.allCreatedItems.Add(newNode);
      return newNode;
}
private Node GetStandardNode(ItemGroup parent)
{
      return new Node(this.topology, parent);
}
private ContentNode GetContentNode(ItemGroup parent, string contentType)
{
      IContentModule content =
            this.Content.GetContentModule(contentType);
      ContentNode newNode =
            new ContentNode(this.topology, parent, content);
      return newNode;
}
```

Codeausschnitt 8: Knotenerzeugung anhand der Inhaltskennung

Wenn die Inhaltskennung eine gültige und nicht leere Zeichenkette ist, wird mit dieser die `GetContentModule`-Methode von `ContentFactory` aufgerufen. Wenn `ContentFactory` keine entsprechende Factory-Methode finden kann, wird ein `DefaultContent`-Objekt erzeugt und zurückgegeben. Ist eine Factory-Methode vorhanden, wird sie ausgeführt und das Ergebnis ist der *Knoteninhalt* zur Kennung. Mit diesem Inhalt kann das neue `ContentNode`-Objekt erzeugt werden. Damit Knoteninhaltstypen registriert werden können, muss auf das Objekt vom Typ `ContentTypeProvider` dort zugegriffen werden können, wo die *MindGraph*-Komponente verwendet wird. Die Referenz auf den *MindGraph*-Editor wird in der XAML-Datei der Hauptseite der Silverlight-Assembly definiert, wie in Codeausschnitt 9 gezeigt.

```
…
<local:MindModuleMenu x:Name="mindmenu" Grid.Row="0"/>
<mindgraph:MindGraphView x:Name="mindgraph" Grid.Row="1"/>
```

Codeausschnitt 9: Deklarative Beschreibung des *MindGraph*-Editors mit grafischem Menü

Es ist dabei erkennbar, das Menü und Editor getrennt voneinander existieren. Über die Code-Behind-Datei des Silverlight-Controls kann dem grafischen Menü eine Instanz auf ein abstraktes Menü, welches die Schnittstelle `IMapMenu` implementiert, übergeben werden. Nach dieser Zuweisung kann die Mindmap durch das grafische Menü bearbeitet werden, indem dieses sämtliche Befehle an

die Menü-Instanz von `MindGraphView` weiterleitet. Dadurch können grafische Menü-Komponenten einfach ausgetauscht und die Funktionalität von *MindGraph* auf gleiche Art und Weise genutzt werden.

Damit Silverlight den *MindGraph*-Editor als Control verwenden kann, muss dieser von der Silverlight-Klasse `UserControl` erben. Um eine vereinfachte Sicht auf die relevanten Methoden von `MindGraphView` zu erhalten, implementiert `MindGraphView` die Schnittstelle `IMindGraph`. Diese Schnittstelle spezifiziert eine Referenz auf die erwähnte Instanz, die `IMapMenu` implementiert, und veröffentlicht somit alle Methoden, die eine grafische Menüstruktur für den *MindGraph*-Editor aufrufen kann. Zur besseren Übersicht wurden alle Methoden dieses Interfaces in Abbildung 46 weggelassen, bis auf die `SelectNodeType`-Methode. Über diese Methode kann ein Menü festlegen, welche Inhalte neu erzeugte *Knoten* haben sollen.

Es wurde bereits vorgestellt, dass die `ItemFactory` bei der Erstellung eines neuen *Knotens* eine Inhaltskennung erwartet. Es existiert eine globale Inhaltskennung, die bei jeder Knotenerzeugung verwendet wird. Über die `SelectNodeType`-Methode kann diese Inhaltskennung zur Laufzeit umgestellt werden, sodass neue *Knoten* mit der neuen Inhaltskennung erzeugt werden. Das bedeutet, dass das grafische Menü wissen muss, welche Knoteninhaltstypen zur Verfügung stehen. Bei der Erzeugung des grafischen Menüs muss dieses Wissen zur Verfügung gestellt werden. Gleichzeitig muss zu den Inhaltskennungen auch die Inhaltserzeugung definiert werden können. Die `IMindGraph`-Schnittstelle spezifiziert dafür eine direkte Referenz auf das `ContentTypeProvider`-Objekt, das bereits vorgestellt wurde. In Abbildung 46 ist die Klassenstruktur so dargestellt, wie sie von dem Verwender der *MindGraph*-Komponente gesehen wird.

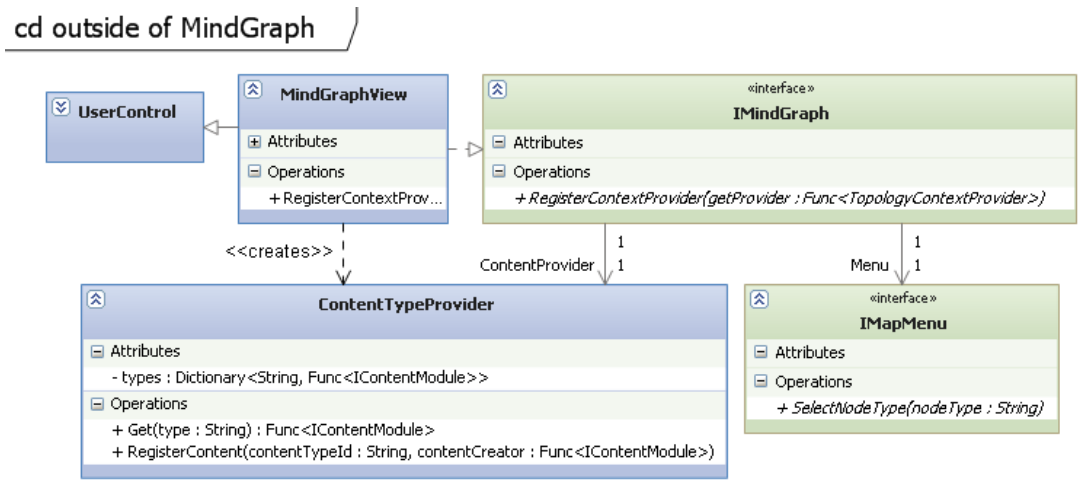


Abbildung 46: Äußere Sicht auf die *MindGraph*-Komponente

`IMindGraph` besitzt die Methode `RegisterContextProvider`, über die eine Factory-Methode registriert werden kann, die ein Objekt erzeugt, dessen Klasse eine Unterklasse von `TopologyContextProvider` ist. Über diesen `TopologyContextProvider` können eigene Interceptor-Objekte in *MindGraph* verwendet werden. Die Interception-Struktur wurde in Unterabschnitt 4.3.4 MindGraph Anbindung behandelt. Für die Verwendung von eigenen Knoteninhaltstypen ist diese Methode nicht relevant und wird daher nicht weiter betrachtet.

Wenn eine MindGraphView-Instanz von der Silverlight-Laufzeitumgebung erzeugt wird, dann wird auch eine ContentTypeProvider-Instanz erstellt, unabhängig von den TopologyControl-Objekten, die sich immer nur auf die aktuell geöffnete Mindmap beziehen. Da ItemFactory und ContentFactory mit jedem neuen TopologyControl-Objekt neu erzeugt werden, besitzt ContentFactory die RegisterCustomTypes-Methode, mit der das globale ContentTypeProvider-Objekt nachträglich registriert werden kann. Die Erzeugung von konkreten *Knoteninhalten* kann dann global definiert werden und gilt somit für die gesamte Lebenszeit des *MindGraph*-Editors. Wenn in der XAML-Datei die MindGraphView-Instanz mit einem Namen versehen wurde, können in der Code-Behind-Datei Knotentypen über die RegisterContent-Methode angemeldet werden, wie in Codeausschnitt 10 gezeigt.

```
IMindGraph mg = this.mindgraph;
mg.ContentProvider.RegisterContent("sample", () =>
{
      return new Sample.SampleContent();
});
```

Codeausschnitt 10: Anmeldung eines Knoteninhalttpys zur Verwendung im *MindGraph*-Editor

Als Inhaltskennung wird im oberen Beispiel "sample" gewählt. Diese Kennung muss über die SelectNode-Methode der IMapMenu-Schnittstelle ausgewählt werden. Wenn anschließend *Knoten* erzeugt werden, wird die ebenfalls im oberen Beispiel definierte Factory-Methode ausgeführt, die ein SampleContent-Objekt erzeugt und zurückgibt. SampleContent muss dafür lediglich das IContentModule-Interface implementieren und wird automatisch durch den *MindGraph*-Editor innerhalb des neu erzeugten *Knotens* dargestellt. IContentModule spezifiziert das Feld ContentTypeName. Durch dieses Feld muss von SampleContent die gleiche Inhaltskennung zurückgegeben werden, mit der das Modul auch bei ContentTypeProvider registriert wurde. Die Kennung, die bei der Registrierung angegeben wird, wird vom grafischen Menü verwendet. Die Kennung, die durch die ContentTypeName-Eigenschaft zurückgegeben wird, dient dazu den Inhaltstyp zu serialisieren und beim Deserialisieren wieder erstellen zu können. Normalerweise sind diese Kennungen identitisch.

5.3.2 Verbindungsfunktion

In Unterabschnitt 5.2.2 Knotenverbindungen erstellen wurde zwischen primären und sekundären Verbindungen zwischen *Knoten* unterschieden und gezeigt, wie diese erstellt und *Knoteninhalten* zugänglich gemacht werden können. In diesem Unterabschnitt soll die Implementierung dieser Funktion beschrieben werden. Da die Erstellung von primären Knotenverbindungen bereits in der Vorarbeit implementiert ist und deren Verwaltung in Unterabschnitt 3.1.2 Topologische Objekte behandelt wurde, wird an dieser Stelle nur auf die sekundären Verbindungen eingegangen. Um zwei *Knoten* miteinander zu verbinden, wird für beide *Knoten* die UserTriggeredConnect-Methode auf dem ItemConnector-Objekt aufgerufen, das von dem aktuellen TopologyContext verwaltet wird. Nach einem Aufruf wird der übergebene *Knoten* gemerkt. Bei einem zweiten Aufruf wird die Verbindung zwischen dem neu übergegebenen und dem gemerkten *Knoten* hergestellt, vorausgesetzt der gemerkte *Knoten* wurde zwischenzeitlich nicht gelöscht. Wenn zwischen diesen beiden *Knoten* bereits eine Verknüpfung besteht, wird diese entfernt. Durch das Löschen eines *Knotens* werden alle seine Verbindungen zu anderen *Knoten* gelöscht. ItemConnector verwendet für diesen Vorgang zwei Funktionen, die über die in Unterabschnitt 3.1.2 Topologische Objekte vorgestellte Funktionsinfrastruktur aufgerufen werden können. Das zuständige FunctionProvider-Objekt wird von TopologyControl verwaltet.

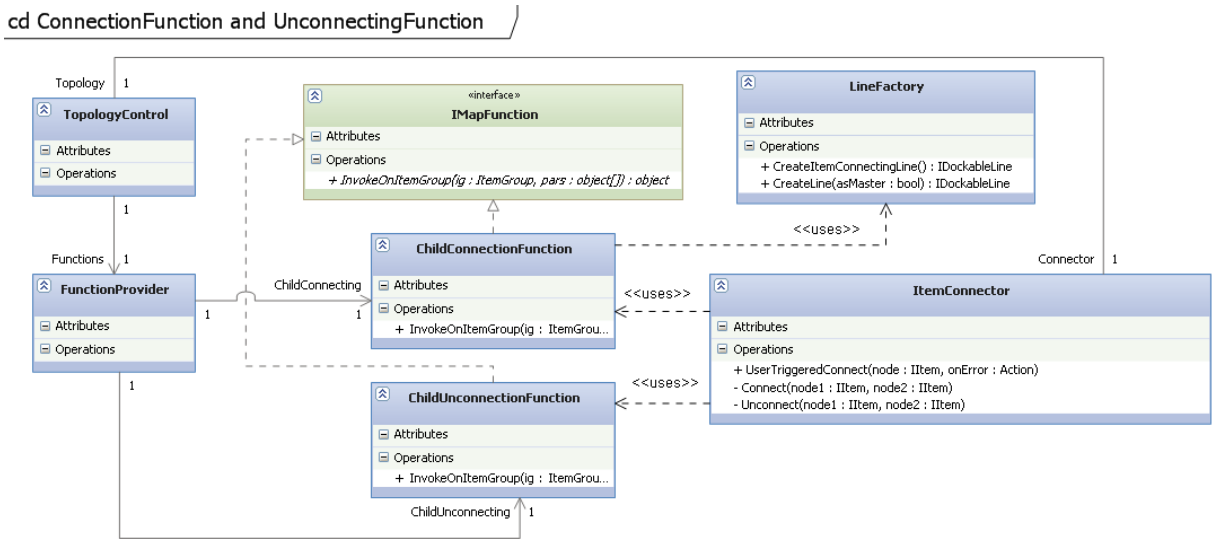


Abbildung 47: FunctionProvider mit Funktionen zum Erstellen und Entfernen von Knotenverbindungen

Wie in dem Klassendiagramm in Abbildung 47 zu sehen, verwendet `ChildConnectionFunction` eine Referenz auf die `LineFactory`, die die Verbindungslinien aus Unterabschnitt 3.1.1 Visuelle Objekte erzeugen kann. Die Referenz auf diese Factory kann über die in `IItem` spezifizierte Referenz auf die Zeichenfläche (`IMapCanvas`) des *MindGraph*-Editors erreicht werden. Nachdem die Verbindungslinie erstellt wurde, wird sie wie in Unter-Unterabschnitt 5.2.2.2 Sekundäre Knotenverbindungen geplant, an die zentralen Andockpunkte der *Knoten* gebunden, zwischen denen sie eine Verknüpfung herstellen soll. Diese Funktion ist in Codeausschnitt 11 der `ChildConnectionFunction`-Klasse dargestellt.

```
line.DockStart(node1.LineDocking.GetCenterBinding());
line.DockEnd(node2.LineDocking.GetCenterBinding());
node1.ConnectionsToOthers.Add(node2, line);
node2.ConnectionsToOthers.Add(node1, line);
node1.Surface.InsertLine(line);
```

Codeausschnitt 11: Herstellung einer grafischen und logischen Verbindungslinie zwischen zwei *Knoten*

Die Verbindung wird bei den zwei betroffenen *Knoten* nicht nur per Silverlight DataBinding angedockt, sondern auch noch logisch registriert. Abschließend wird die neue Linie der Zeichenfläche hinzugefügt. Zum Entfernen der Verbindung muss die logische und die grafische Verbindung der zwei *Knoten* entfernt werden, was von `ChildUnconnectionFunction` wie in Codeausschnitt 12 übernommen wird.

```
var line = node1.ConnectionsToOthers.Get(node2);
node1.ConnectionsToOthers.Remove(node2);
node2.ConnectionsToOthers.Remove(node1);
node1.Surface.RemoveLine(line);
```

Codeausschnitt 12: Entfernung einer grafischen und logischen Verbindungslinie zwischen zwei *Knoten*

Während die grafische Verbindung nur Auswirkungen auf die Zeichenfläche hat, kann die logische Verbindung von den *Knoten* verwendet werden. In Unter-Unterabschnitt 5.2.2.2 Sekundäre Knotenverbindungen wurde dafür ein `ConntectionContext`-Objekt vorgesehen. Die Implementierung besteht allerdings aus zwei Klassen. Das `SecondaryConnectionHub`-Objekt ist das Gegenstück zum `ChildArea`-Objekt, während letzteres primäre und ersteres sekundäre Verknüpfungen in der Topologie der Mindmap verwaltet. Über .NET-Events können Informationen

über Veränderungen in diesen beiden Strukturen auch abonniert werden. Das `ConnectionContext`-Objekt fasst pro *Knoten* in der Mindmap sämtliche Verknüpfungen zusammen, indem es beide Verwaltungsinstanzen und zusätzlich den Vaterknoten des aktuellen *Knotens* an einer zentralen Stelle zusammenführt.

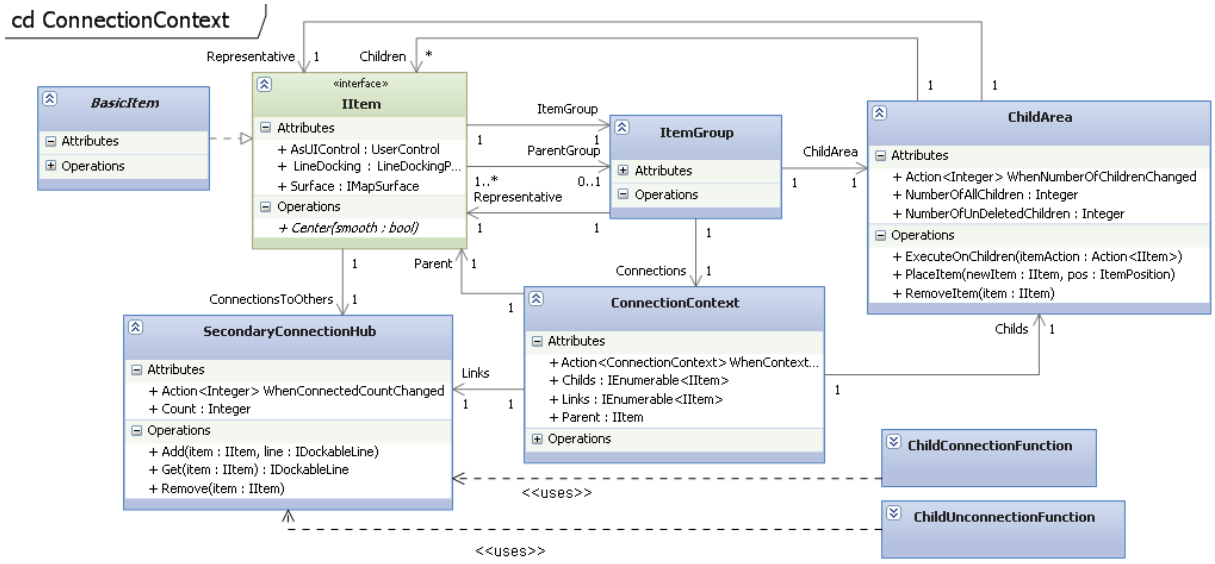


Abbildung 48 Verwaltung von Knotenverbindungen über ConnectionContext

Im Klassendiagramm in Abbildung 48, lässt sich der `ConnectionContext` eines jeden *Knotens* über die Knotengruppe des *Knotens* erreichen. Es ist an dieser Stelle zu berücksichtigen, dass der Mittelpunkt in der Mindmap eine Null-Referenz als `ParentGroup` enthält.

Es wurde angedeutet, dass zwischen zwei *Knoten* nur eine sekundäre Verbindung bestehen kann. Dies hat den Grund, dass bestehende Verbindungen durch den Nutzer leicht wieder entfernt werden können. Da ein *Knoten* nach Unterabschnitt 3.1.1 Visuelle Objekte nur drei Andockpunkte besitzt und nur der zentrale genutzt wird (der linke und rechte Punkt werden in der Mindmap Topologie verwendet), wären mehrere Assoziationen zwischen gleichen Knotenpaaren sehr unübersichtlich. Weiterhin sind nur Verknüpfungen zwischen unterschiedlichen *Knoten* möglich. Prinzipiell könnte eine Kante erstellt werden, die eine entsprechende Krümmung darstellen kann, um *Knoten* mit sich selbst zu verbinden. Es besteht zwar die Anforderung (D5) Knoten beliebig verknüpfen, wobei dort mit "beliebig" die Verknüpfung unabhängig von der Topologie gemeint ist. Die Funktionalität zur Verknüpfung von einem *Knoten* mit sich selbst war nicht direkt gefordert und wird daher in diesem Prototyp nicht implementiert. Eine weitere Einschränkung besteht darin, dass es keine gerichteten Linien gibt. Folglich sind alle Verbindungen ungerichtet und gelten damit in beide Richtungen. Da `ChildConnectingFunction` bei der Erstellung einer Linie nur eine logische Verbindung registrieren könnte, wäre eine spätere Implementierung einer solchen Pfeil-Linie möglich.

5.3.3 CRM-Knoten

In Unterabschnitt 5.2.3 CRM-Module wurden vier Knotentypen vorgestellt, deren Implementierung in diesem Unterabschnitt behandelt werden soll. Laut Anforderung (D2) Kontextabhängigkeit müssen *Knoten* auf ihre Umgebung in der Mindmap reagieren können. Mit der Einführung der `ConnectionContext`-Klasse verfügen *Knoten* über Referenzen auf sämtliche Verknüpfungen zu ihren Nachbarknoten. Wenn sich diese Nachbarknoten ändern, muss das zuständige `ConnectionContext`-Objekt eine Information über diese Änderung zur Verfügung stellen. Dafür

kann bei ihm ein Callback als .NET-Event registriert werden, über das man über Änderungen in den verbundenen *Knoten* informiert werden will. Um die registrierten Callbacks aufzurufen, muss `ConnectionContext` eigene Callbacks bei dem verbundenen `ChildArea`- und `SecondaryConnectionHub`-Objekt anmelden. Wie in Unterabschnitt 5.3.2 Verbindungsfunktion erwähnt, besitzen diese beiden Verbindungsverwalter entsprechende .NET-Events, die ausgeführt werden, sobald die Anzahl der Verbindungen verändert wurde. Das `ConnectionContext`-Objekt muss sicherstellen, dass es bei allen verbunden *Knoten* auf Änderungen reagieren kann. Das Diagramm in Abbildung 49 zeigt, dass `BasicItem` das Interface `IContent` implementiert.

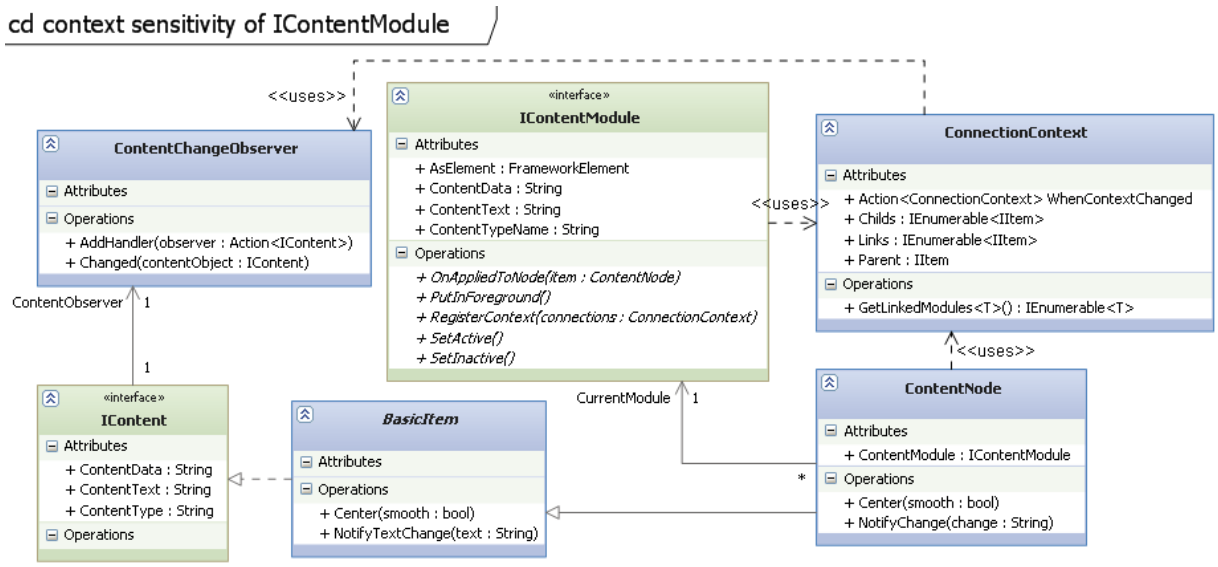


Abbildung 49 Überwachung von Inhaltsveränderungen in *Knoten*

`IContent` spezifiziert drei Eigenschaftsfelder. Durch die Implementierung dieser Schnittstelle muss `BasicItem` diese drei Felder und eine Referenz auf ein `ContentChangeObserver`-Objekt implementieren. Über die `NotifyTextChange`-Methode kann `BasicItem` beliebige Inhaltsänderungen an das Observer-Objekt und damit an weitere Observer propagieren, die sich über Delegates bei `ContentChangeObserver` registriert haben. Sobald zwei *Knoten* verbunden werden, müssen sich ihre `ConnectionContext`-Objekte bei dem Observer-Objekt des jeweils anderen *Knotens* registrieren. Da auf alle neuen Verbindungen reagiert werden kann, ist diese Registrierung möglich.

Wenn also zu beliebiger Zeit die `NotifyTextChange`-Methode von einem `BasicItem`-Objekt aufgerufen wird, werden dessen `ContentChangeObserver` und somit auch die `ConnectionContext`-Objekte der Nachbarelemente informiert. Diese `ConnectionContext`-Objekte können dann alle Eventhandler ihrer `WhenContextChanged`-Events ausführen. Dieses Netz der Änderungsüberwachung wird automatisch erstellt, sobald *Knoten* in der Mindmap eingefügt, verknüpft oder topologisch verschoben werden. `WhenContextChanged` ist somit eine zentrale Stelle, über die automatisch auf Änderungen in den umliegenden *Knoten* und auf das Hinzufügen oder Entfernen von primären und sekundären Knotenverknüpfungen eines beliebigen *Knotens* reagiert werden kann. In Unterabschnitt 5.2.1 Erweiterte Knoten verwenden wurden die Methoden `SetData` und `SetSettings` geplant. In der Implementierung entsprechen die `ContentText`-Eigenschaft der `SetData`-Methode und die `ContentData`-Eigenschaft der `SetSettings`-Methode.

Wenn ein *Knoteninhalt* einem `ContentNode`-Objekt zugeordnet wird, wird bei dem *Knoteninhalt* die Methode `OnAppliedToNode` mit einer Referenz auf den Container-*Knoten* aufgerufen. Diese Referenz kann genutzt werden, um eine Veränderung des *Knoteninhalts* als Veränderung von `ContentNode` zu propagieren. Dafür muss lediglich die `NotifyChange`-Methode des `ContentNode`-Objekts mit dem geänderten Wert aufgerufen werden. Diese ruft die `NotifyTextChange`-Methode der Oberklasse auf und informiert somit das `ContentChangeObserver`-Objekt über die Änderung. Bei der Zuweisung eines *Knoteninhalts* an ein `ContentNode`-Objekt wird zusätzlich die Methode `RegisterContext` aufgerufen, die durch die Schnittstelle `IContentModule` vorgegeben ist. Dieser Methode wird ein `ConnectionContext`-Objekt übergeben. Damit kennt der *Knoteninhalt* sämtliche Verbindungen, die sein Container-*Knoten* besitzt und kann auf deren Änderungen reagieren. Bei der Behandlung dieser Änderungen über das .NET-Event `WhenContextChanged` von `ConnectionContext`, sollte `NotifyChange` des `ContentNode`-Objekts nicht aufgerufen werden, da dies abhängig von der Behandlung zu einem Stack Overflow führen kann. Zusätzlich zu der Information über Änderung kann der *Knoteninhalt* auch typsicher auf die *Knoteninhalte* der Nachbarknoten zugreifen. Dafür kann die Methode `GetLinkedModules` verwendet werden, die alle *Knoteninhalte* der verbundenen *Knoten* zurückgibt, die von dem vom Aufrufer angegebenen Typ sind.

5.3.3.1 PersonModule

Der in Unter-Unterabschnitt 5.2.3.1 User-Knoten vorgestellte Benutzerknoten besteht aus einem `PersonModule`-Objekt, das einem `ContentNode`-Objekt zugewiesen wurde. Dabei besitzt das Modul ein einziges Datenfeld, welches die Personenkennung enthält. Diese Kennung wird als `ContentText`-Eigenschaft des Moduls verwendet. `ContentData` wird nicht verwendet. Die Oberfläche des Personenmoduls besteht aus zwei Ansichten. Die erste Ansicht ist die Erstellungsansicht, in welcher eine Benutzerkennung eingegeben werden kann. Theoretisch kann auch ein neuer Benutzer erstellt werden. Mit einer eingegebenen Benutzerkennung kann die Person aus der CRM-Domäne geladen und mit Name, Rolle und Bild in der zweiten Ansicht angezeigt werden. Abbildung 50 zeigt diese beiden Ansichten.

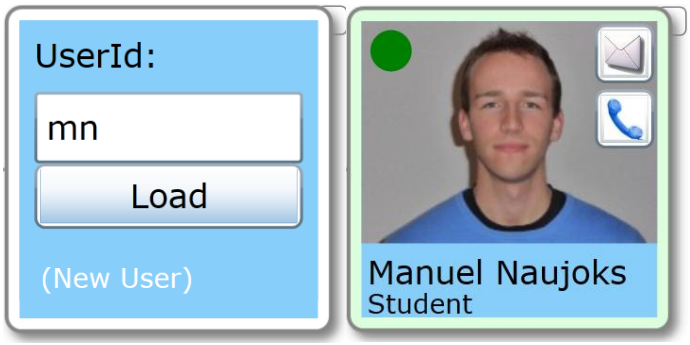


Abbildung 50: Erstellungsansicht (links) und normale Ansicht (rechts) des Personenknotens

Das Foto der Person wurde aus dem CAS Intranet kopiert, während die Icons für E-Mail[11] und Telefon[12] im Internet gefunden und verändert wurden, um Urheberrechtsverletzungen zu vermeiden. Die Funktionsweise des Personenmoduls ist an dieser Stelle nicht bedeutend, da es lediglich ein Beispiel eines CRM-*Knotens* sein soll, der in einer eigenen Silverlight-Assembly mit der Bezeichnung `SampleCRM` definiert wurde. In Codeausschnitt 13 ist die Registrierung des Moduls zu

[11] E-Mail-Icon Original von http://www.pawprintspictures.com/contact2.html
[12] Telefon-Icon Original von http://www.carbonfund.org/blog/news/spams-carbon-footprint-assessed

sehen. Als Inhaltskennung ist die Zeichenkette "person" angegeben, die auch über das in `IContentModule` spezifizierte Feld `ContentTypeName` von `PersonModule` zurückgegeben wird. Die Factory-Methode für die Erstellung wird als anonyme Methode zur Verfügung gestellt.

```
mg.ContentProvider.RegisterContent("person", () =>
{
      var pn = new SampleCRM.PersonModule();
      pn.OnLoadPerson((id) =>
            SampleCRM.Person.GetPersons().FirstOrDefault(
                  (p) => p.Id == id));
      return pn;
});
```

Codeausschnitt 13: Anmeldung des Knoteninhaltstyps für Personenknoten mit externer Datenanbindung

Bei der Erzeugung des `PersonModule`-Objekts wird dessen externe Datenanbindung festgelegt. Wenn der Benutzer des Mindmap-Editors auf den Load-Button des Modules klickt, um eine Person zu laden, wird der Delegate ausgeführt, der im oberen Codeausschnitt mit der `OnLoadPerson`-Methode registriert wurde. Anhand der eingegeben Benutzerkennung wird dann die entsprechende Person aus der Datenbank geladen und zurückgegeben.

In Unter-Unterabschnitt 5.2.3.5 Beispielhafte Datenhaltung wurde eine Datenbank im Speicher geplant. Die `SampleCRM.Person.GetPersons`-Methode liefert eine Liste aller Personen aus der Implementierung dieser beispielhaften Speicherdatenbank. In dieser Auflistung kann die Person mit der gewünschten Kennung gefunden und dem `PersonModule`-Objekt zurückgegeben werden kann. Auf diese Weise erfolgt die beispielhafte Datenanbindung für Personen. `Person`-Objekte repräsentieren eine Person in der Speicherstruktur. `PersonModule`-Objekte und damit Benutzerknoten repräsentieren `Person`-Objekte. Um `Person`-Objekte aus einer anderen Struktur, wie zum Beispiel einer produktiven Datenbank zu laden, müsste lediglich die `OnLoadPerson`-Methode mit einem geeigneten Ladevorgang als Delegate aufgerufen werden.

Bei der Zuweisung des geladenen Objekts an das Modul, ruft das Modul die `NotifyChange`-Methode mit der neuen Benutzerkennung auf. Auf diese Weise können benachbarte *Knoten* über das Vorhandensein des neuen Personenknotens informiert werden. Die Veränderung des Status des `PersonModule`-Objekts erfolgt über die Veränderung der Werte des `Person`-Objekts, die ebenfalls aus der Beispieldatenbank aktualisiert werden können. Datenanbindung ist damit eine alleinige Angelegenheit des CRM-Objekts und kann unabhängig von der Mindmap implementiert werden. Neben dem lesenden Zugriff auf die Speicherdatenbank kann auch ein schreibender Zugriff implementiert werden. Da der Speicher aber nur lokal verändert werden würde, würde eine solche Änderung nicht gemeinschaftlich stattfinden können. Die Erstellung eines `Person`-Objekts aus einem Personenknoten ist also nur in einer produktiven Umgebung mit gemeinsamer Datenbankstruktur sinnvoll und wurde daher nicht implementiert.

5.3.3.2 AppointmentModule

In Unter-Unterabschnitt 5.2.3.3 Calendar-Knoten wurden Terminknoten geplant. Diese *Knoten* bestehen aus einem `ContentNode`-Objekt mit einer Referenz auf ein `AppointmentModule`-Objekt. `AppointmentModule` besitzt dabei genau wie Personenknoten eine Erstellungsansicht und eine normale Ansicht, die in Abbildung 51 gezeigt werden.

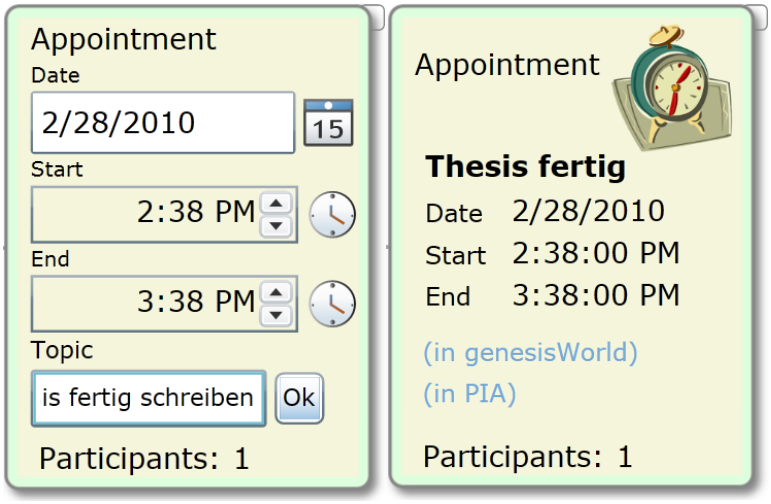


Abbildung 51: Erstellungsansicht (links) und normale Ansicht (rechts) des Terminknotens

Das Wecker-Icon (j0234131.wmf) wurde aus der Clipart-Galerie von Microsoft Office kopiert. Termine werden nur erzeugt und können in der derzeitigen Implementierung nicht geladen werden. Damit können bestehende Termine nur in der Mindmap verwendet werden, in der sie erzeugt wurden. Das Laden von vorhandenen Terminen kann zu einem späteren Zeitpunkt implementiert werden, wenn eine produktive Datenanbindung aus der CAS Infrastruktur genutzt werden kann. Sämtliche Daten des Termins werden in einer speziellen Termin-Zeichenkette codiert und können somit über die `ContentText`-Eigenschaft serialisiert und deserialisiert werden. Das `ContentData`-Feld wird nicht verwendet. Um die Uhrzeiten für den Termin einzugeben, wird das `TimePicker`-Control aus dem Silverlight Toolkit[13] verwendet. Alle anderen Controls sind Bestandteil des Silverlight Frameworks. In der Erstellungsansicht und der normalen Ansicht des `AppointmentModule`-Objekts wird ein Feld mit der Anzahl der Teilnehmer angezeigt. Diese Anzahl ergibt sich aus allen Personenknoten, die nach Unterabschnitt 5.2.2 Knotenverbindungen erstellen, sekundäre oder primäre Verbindungen zu dem Terminknoten haben. Um diese Verbindungen zu ermitteln, wird in der `RegisterContext`-Methode, die durch `IContentModule` vorgeschrieben ist, bei dem übergebenen `ConnectionContext`-Objekt ein Eventhandler für das .NET-Event `WhenContextChanged` registriert. Immer wenn sich dieser Kontext wie in Unterabschnitt 5.3.3 CRM-Knoten beschrieben ändert, wird eine Liste aller `PersonModule`-Objekte der benachbarten *Knoten* ermittelt. Die Anzahl der Elemente in dieser Liste ergibt die Anzahl der Teilnehmer an dem Termin des `AppointmentModule`-Objekts, wie die Zuweisung an das Teilnehmerfeld in Codeausschnitt 14 zeigt.

```
public void RegisterContext(ConnectionContext connections)
{
      connections.WhenContextChanged += (cc) =>
            this.participants.Text =
                  cc.GetLinkedModules<PersonModule>().Count().ToString();
}
```

Codeausschnitt 14: Kontextsensitivität von Terminen über Knotenverbindungen

`AppointmentModule` ist damit ein `IContentModule`-Objekt, das auf seine Umgebung in der Mindmap reagieren kann und somit kontextsensitiv ist. Dafür besitzt es keine direkte externe Datenanbindung, wie in Codeausschnitt 15 aus der Registrierung des Moduls erkennbar ist.

[13] Silverlight Toolkit Zusätzliche Controls für Silverlight 3: http://silverlight.codeplex.com/

```
mg.ContentProvider.RegisterContent("appointment", () =>
{
      return new SampleCRM.AppointmentModule();
});
```

Codeausschnitt 15: Anmeldung des Knoteninhaltstyps für Terminknoten

In der Factory-Methode wird lediglich ein `AppointmentModule`-Objekt erzeugt und zurückgegeben. Um eine externe Datenquelle anzubinden, müssen Lade- und Speichervorgänge bei dem Modul registriert werden. Die `ContentTypeName`-Eigenschaft von `AppointmentModule` ist auf die Zeichenkette "appointment" festgelegt, die auch bei der Registrierung des Moduls angegeben werden muss.

5.3.3.3 DocumentModule

In Unter-Unterabschnitt 5.2.3.4 Document-Knoten wurde ein Knotentyp vorgestellt, der ein Dokument aus der CRM-Domäne repräsentieren kann. Das dafür notwendige Modul ist `DocumentModule`, das genau wie `PersonModule` und `AppointmentModule` in der `SampleCRM`-Assembly definiert ist. Es besitzt eine Erstellungsansicht und eine Ansicht um das geladene Dokument anzuzeigen. Beide Ansichten sind in Abbildung 52 dargestellt.

Abbildung 52: Erstellungsansicht (links) und normale Ansicht (rechts) des Dokumentknotens

Das Dokument-Icon enthält einen Bestandteil des CAS Logos[14] und wurde in Form eines Dateisymbols umgestaltet. Obwohl geplant wurde, Dokumente aus der Speicherdatenbank zu laden, unterstützt die Implementierung des Moduls ausschließlich das Laden von bestehenden Dokumenten von der Festplatte. Dadurch kann ein `DocumentModule`-Objekt im Rahmen des Prototyps beliebige Dateien repräsentieren und ist nicht auf Testdaten beschränkt, was eine sinnvolle Nutzung ermöglicht. Eine Anbindung an eine externe Datenquelle mit bestehenden Dokumenten kann aber genau wie für `PersonModule`-Objekte nachträglich implementiert werden. Da die Datenanbindung durch das Modul vollständig selbst übernommen wird und es Dokumente eigenständig von der Festplatte auswählen kann, wenn der Benutzer auf den Load-Button klickt, muss es in der registrierten Factory-Methode lediglich erzeugt und zurückgegeben werden. Diese Erzeugung wird in Codeausschnitt 16 gezeigt.

```
mg.ContentProvider.RegisterContent("document", () =>
{
      return new SampleCRM.DocumentModule();
});
```

Codeausschnitt 16: Anmeldung des Knoteninhaltstyps für Dokumentknoten

[14] CAS Logo Original von http://www.cas.de/

Die ContentTypeName-Eigenschaft der DocumentModule-Objekte ist auf "document" festgelegt. Welche Datei geladen wurde, wird durch den Dateinamen angegeben, der in der ContentText-Eigenschaft gespeichert wird. Der Inhalt der Datei wird nicht geladen und gespeichert, da die Datei lediglich repräsentiert wird. DocumentModule kann aber zu einem späteren Zeitpunkt so erweitert werden, dass es eine Vorschau des Inhalts des Dokuments anstelle eines beispielhaften Dateisymbols anzeigt. Weiterhin kann das Modul so erweitert werden, dass ein Doppelklick auf den Dokumentknoten die Datei öffnet oder zumindest dessen Inhalt anzeigt. DocumentModule-Objekte sind in Bezug auf Nachbarknoten nicht kontextsensitiv.

5.3.3.4 AddressModule

In Unter-Unterabschnitt 5.2.3.2 Address-Knoten wurde der Adressknoten vorgestellt. Die Implementierung dieses *Knotens* nutzt ein AddressModule-Objekt, das in diesem Unter-Unterabschnitt beschrieben wird. Dieses Modul ist in der SampleCRM-Assembly definiert und besitzt zwei Ansichten. Die erste ist die Erstellungsansicht und die zweite ist die Ansicht der geladenen Adresse, wie in Abbildung 53 zu sehen ist.

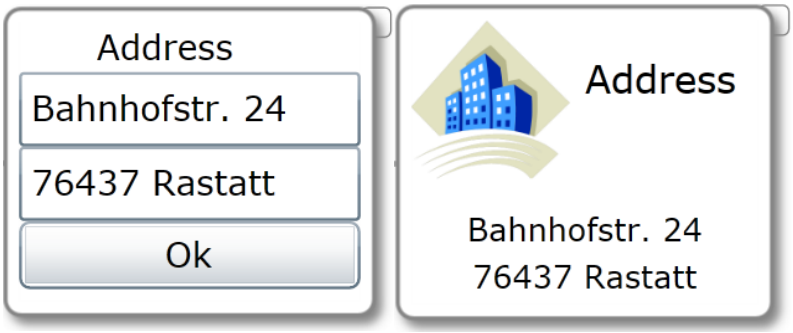


Abbildung 53: Erstellungsansicht (links) und normale Ansicht (rechts) des Adressknotens

Das Adress-Icon (j0205462.wmf) wurde aus der Clipart-Galerie von Microsoft Office kopiert. Die Erstellungsansicht des *Knotens* erlaubt das Eingeben von zwei Adresszeilen. Damit unterstützt AddressModule nicht wie geplant das Laden von vorhandenen Adressen, sondern erlaubt lediglich das Erstellen von neuen Adressen. Genau wie Dokumentknoten erlauben Adressknoten somit eine sinnvolle Nutzung des Prototyps, da vorgegebene Testdaten nicht sinnvoll eingesetzt werden könnten. Eine externe Datenanbindung kann zu einem späteren Zeitpunkt implementiert werden. Durch eine einfache Factory-Methode wird ein AddressModule-Objekt erzeugt, wie Codeausschnitt 17 zeigt.

```
mg.ContentProvider.RegisterContent("address", () =>
{
      return new SampleCRM.AddressModule();
});
```

Codeausschnitt 17: Anmeldung des Knoteninhaltstyps für Adressknoten

Objekte vom Typ AddressModule haben als ContentTypeName-Eigenschaft die Zeichenkette "address", welche auch bei der Registrierung der Factory-Methode angegeben werden muss. Die geladene Adresse wird in einer speziellen Zeichenkette codiert, um so von der ContentText-Eigenschaft verwendet werden zu können. Das Feld ContentData wird nicht verwendet. AddressModule-Objekte sind in Bezug auf Nachbarknoten nicht kontextsensitiv.

5.3.4 Anpassung für Zusammenarbeit

Nachdem die Implementierung der *Knoten* und deren Inhalte beschrieben wurde, wird in diesem Unterabschnitt behandelt, wie die Funktionen zur gemeinschaftlichen Modellierung an die neuen Funktionen angepasst werden. Die Aspekte der Zusammenarbeit wurden in Kapitel 4 Gemeinschaftliche Modellierung vorgestellt. Da *Knoten* mit einem speziellen *Knoteninhalt* erzeugt werden können, muss die Add-Operation angepasst werden. Außerdem können *Knoten* sekundäre Verbindungen mit anderen *Knoten* eingehen. Auch dafür muss eine geeignete gemeinschaftliche Operation genutzt werden. Abschließend wird beschrieben, wie die Edit-Operation genutzt wird, um *Knoteninhalte* gemeinschaftlich zu ändern.

5.3.4.1 Erweiterte Add-Operation

In Unter-Unterabschnitt 4.1.1.1 Knoten einfügen wurde die Operation definiert, die dem Einfügen eines *Knotens* in der Mindmap entspricht. Da zu diesem Zeitpunkt davon ausgegangen wurde, dass *Knoten* nur Text enthalten können, brauchten Knotentypen nicht unterschieden zu werden. In Unterabschnitt 5.2.1 Erweiterte Knoten verwenden wurde aber das Verwenden von erweiterten *Knoten* behandelt, die anhand einer eindeutigen Typkennung des Inhalts unterschieden werden können. In Unterabschnitt 5.3.1 Knotenfabrik wurde gezeigt, dass die Erstellung eines neuen *Knotens* in der Mindmap nur mit einer Inhaltskennung möglich ist. Aus diesem Grund muss die Add-Operation durch einen weiteren Parameter erweitert werden, um allen Teilnehmern einer gemeinschaftlichen Sitzung, wie sie in 4.2.1 Administrative Serverseite beschrieben wurde, den Typ des neuen *Knotens* mitteilen zu können. Die neue Add-Operationen hat also folgende Signatur:

addNode :: (parentNodeId, newNodeId, nodePosition, contentTypeName) -> ()

Die ersten drei Parameter sind von der bisherigen Operation bekannt. Der vierte Parameter ist die eindeutige Inhaltskennung des neuen *Knotens*, die in Unterabschnitt 5.3.1 Knotenfabrik als Feld `ContentTypeName` von `IContentModule` beschrieben wurde. Da es sich dabei um eine Zeichenkette handelt, kann der in Unterabschnitt 4.3.3 Client Bibliothek definierte `EventInvoker` `OperationAdd` um diesen Parameter ergänzt werden, wie Codeausschnitt 18 zeigt.

```
OperationAdd = this.wcfEventProxy.GetEventInvoker
                  <string, string, double, double, string>("add",
                  () => handlers.LocalAddHandler());
```

Codeausschnitt 18: Angepasste Definition der atomaren Add-Operation

Dank des generischen *sEvent*-Frameworks braucht die Serverseite der Zusammenarbeit nicht neu kompiliert, sondern lediglich der entsprechende Eventhandler erweitert werden. Dank der Typsicherheit von *sEvent* würde die Behandlung von `OperationAdd` nicht kompilieren, wenn der neue Parameter nicht auch bei der Behandlung angegeben würde. Codeausschnitt 19 zeigt die neue Behandlungsstruktur aus Unterabschnitt 4.3.4 MindGraph Anbindung.

```
OperationEventRequestor = this.wcfEventProxy.GetAggregateRequestor((h) =>
{     …
      h.Handle(this.Events.OperationAdd,
            (pn,nn,px,py,t) => handlers.GlobalAddHandler(pn,nn,px,py,t),
            (s,u) => { });
});
```

Codeausschnitt 19: Angepasste Behandlung der Add-Operation

In dem `GlobalAddHandler` wird das Link-*Ereignis* behandelt, indem es auf die Mindmap angewendet wird, wie Codeausschnitt 20 zeigt.

```
return new PiamindMindGraphHandlerPack()
{     …
      GlobalAddHandler = (pn, nn, dx, dy, t) =>
            this.context.React.Add(pn, nn, dx, dy, t)
};
```

Codeausschnitt 20: Angepasste *Event*-Behandlung der Add-Operation für *MindGraph*

Damit die neue Behandlungsfunktion sinnvoll genutzt werden kann, müssen auch die in 4.3.4 MindGraph Anbindung vorgestellte `DefaultReactor`-Klasse, das `IReactor`-Interface, sowie die `EventInterceptor`-Klasse und das `IInterceptor`-Interface um den neuen Parameter erweitert werden. Damit kann ein *Knoten* mit einem speziellen Inhalt, den ein Teilnehmer erzeugt hat, von allen anderen Teilnehmern der gleichen Sitzung auch automatisch mit dem gleichen Inhaltstyp erzeugt werden. Die Erzeugung ist dabei nur erfolgreich, wenn die Teilnehmer über die gleichen Inhaltstypen verfügen. Da diese Inhaltstypen Bestandteil des Silverlight-Clients sind, der über den Server an die Browser aller Aufrufer gelangt, kennen alle Teilnehmer die gleichen Knoteninhaltstypen. Die Clients sind also immer identisch. Das in Unterabschnitt 4.1.1 Atomare Operationen untersuchte Konfliktpotenzial der Add-Operation wird durch den neuen Parameter nicht beeinträchtigt.

5.3.4.2 Link-Operation

In Unterabschnitt 5.3.2 Verbindungsfunktion wurde die Implementierung vorgestellt, mit der *Knoten* mit anderen *Knoten* verknüpft werden können. Die Verknüpfungen sind somit Bestandteil der Mindmap und müssen auch gemeinschaftlich modelliert werden können. Dafür ist es nötig eine eigene atomare Operation zum Verwalten der Verbindungsherstellung zu definieren. Der Operation wird für die zwei *Knoten*, die verbunden werden sollen, jeweils eine Knotenkennung als Parameter übergeben. Damit die gleiche Operation auch verwendet werden kann, um eine Verbindung wieder zu lösen, erhält die Link-Operation auch einen Wert der angibt, ob eine Verbindung hergestellt oder entfernt werden soll. Es ergibt sich die folgende Signatur:

linkNodes :: (nodeId1, nodeId2, link) -> ()

Um die neue Operation auch als *Ereignis* auslösen zu können, ist eine Definition des `EventInvoker OperationLink` erforderlich, genau wie für die anderen *Ereignisse* in Unterabschnitt 4.3.3 Client Bibliothek. Codeausschnitt 21 zeigt diese Definition.

```
OperationLink = this.wcfEventProxy.GetEventInvoker
                  <string, string, bool>("link",
                  () => handlers.LocalLinkHandler());
```

Codeausschnitt 21: Definition der atomaren Link-Operation

Die Behandlung des Link-*Ereignisses* ist in Codeausschnitt 22 zu sehen.

```
OperationEventRequestor = this.wcfEventProxy.GetAggregateRequestor((h) =>
{     …
      h.Handle(this.Events.OperationLink,
            (n1,n2,l) => handlers.GlobalLinkHandler(n1, n2, l),
            (s,u) => { });
});
```

Codeausschnitt 22: Behandlung der Link-Operation

Das *Ereignis* wird genau wie alle anderen *Ereignisse* auf die Mindmap angewendet, indem der `DefaultReactor` verwendet wird. Dieser muss, genau wie das `IReactor`-Interface, um eine entsprechende Methode erweitert werden, damit die Anwendung des Link-*Ereignisses* wie in Codeausschnitt 23 möglich ist.

```
return new PiamindMindGraphHandlerPack()
{     …
      GlobalLinkHandler = (n1, n2, l) =>
            this.context.React.Link(n1, n2, l)
};
```

Codeausschnitt 23: *Event*-Behandlung der Link-Operation für *MindGraph*

`DefautReactor` verwendet eine Referenz auf das in Unterabschnitt 5.3.2 Verbindungsfunktion vorgestellte `ItemConnector`-Objekt und ruft eine speziell für den Erzeuger optimierte `Connect`- oder `Unconnect`-Methode auf, abhängig von dem link-Wert des *Ereignisses*. Der link-Wert gibt an, ob die Verbindung hergestellt oder entfernt werden soll. Um das Link-*Event* ausführen zu können, muss die `EventInterceptor`-Klasse und das `IInterceptor`-Interface um die entsprechende Methode erweitert werden. In `ItemConnector` kann dann beim Aufruf der `UserTriggeredConnect`-Methode die Link-Methode des aktuellen `EventInterceptor`-Objekts aufgerufen werden, die die Verknüpfung der beiden *Knoten* an alle Teilnehmer der gleichen Sitzung propagiert. Was das Konfliktpotenzial der Link-Operation angeht, so wird eine Verknüpfung auf der Empfängerseite nur hergestellt, wenn beide *Knoten* vorhanden und noch nicht verbunden sind. In allen anderen Fällen wird das Link-*Event* ignoriert um lokale Konflikte gemäß Unterabschnitt 4.1.1 Atomare Operationen zu vermeiden. Das Entfernen einer Verknüpfung wird ebenfalls nur dann vorgenommen, wenn die *Knoten* existieren und über eine bestehende Verbindung verfügen.

5.3.4.3 Edit-Operation und ContentNode

In Unterabschnitt 5.3.1 Knotenfabrik wurde für alle Knoteninhaltstypen das `IContentModule`-Interface vorgesehen. In diesem Interface wird das Feld `ContentText` spezifiziert, sodass alle Klassen die `IContentModule` implementieren eine solche Eigenschaft besitzen. In Unterabschnitt 5.3.3 CRM-Knoten wurde beschrieben, das `BasicItem` die Schnittstelle `IContent` implementiert. Damit muss `BasicItem` auch die in `IContent` vorgesehene Eigenschaft `ContentText` implementieren. Diese Realisierung von `ContentText` liefert bei Abfrage den Wert der `ContentText`-Eigenschaft des *Knoteninhalts* zurück. Wird das `ContentText`-Feld der `BasicItem`-Klasse gesetzt, wird der Wert ebenfalls an das `ContentText`-Feld des Inhalts weitergegeben. In der Beschreibung der CRM-*Knoten* in Unterabschnitt 5.3.3 CRM-Knoten wurde bei jedem *Knoten* erwähnt, dass die Daten die den *Knoteninhalt* ausmachen, in dem durch `IContentModule` spezifizierten `ContentText`-Feld hinterlegt sind. Über dieses Feld können sie auch geändert werden. Wenn ein *Knoteninhalt* eine Änderung seiner Daten bekannt machen will, kann der die `NotifyChange`-Methode von `ContentNode` aufrufen und den Wert seiner `ContentText`-Eigenschaft übergeben. Dadurch kann das `ContentNode` als `BasicItem`-Objekt eine Knotenänderung nicht nur an den `ContentChangeObserver` weiterreichen, sondern gleichzeitig auch an den in Unter-Unterabschnitt 4.3.4.1 Auslösen von Events vorgestellten `EventInterceptor`. Da normale Textknoten ebenfalls `BasicItem`-Objekte sind, haben sie die gleiche Infrastruktur zum Propagieren von Änderungen an `EventInterceptor`. Während bei den Textknoten der Text geändert wird, ändert sich bei den *Knoteninhalten* die aktuelle Repräsentation der Daten. Beides kann über die gleichen Mechanismen mit einer Edit-Operation, wie in Unter-Unterabschnitt 4.1.1.2 Knoten bearbeiten definiert, den anderen Teilnehmern der Sitzung mitgeteilt

werden. Bei diesen wird das Edit-*Event* behandelt und der `DefaultReactor` setzt die `ContentText`-Eigenschaft des entsprechenden `ContentNode`-Objekts auf den geänderten Wert. Da dieser Wert an den Inhalt des `ContentNode`-Objekts weitergegeben wird, kann der Inhalt die Repräsentation des Inhalts des anderen Teilnehmers interpretieren und seinen eigenen Zustand entsprechend verändern. Auf diese Weise kann das bestehende Edit-*Event* von *Knoteninhalten* benutzt werden, ohne das eine Änderung erforderlich ist. *Knoteninhalte* müssen nur eine geeignete Repräsentation ihres Zustands über die `NotifyChange`-Methode nach außen weitergeben können und beim Setzen der eigenen `ContentText`-Eigenschaft auf eine Repräsentation eines externen Zustands reagieren können. Das in Unterabschnitt 4.1.1 Atomare Operationen untersuchte Konfliktpotenzial der Edit-Operation wird durch die vorgestellte Verwendung des Datenparameters nicht beeinträchtigt. Es wird lediglich die Verantwortung über die Konfliktbehandlung von `ContentNode` auf die Objekte übertragen, die `IContentModule` implementieren.

5.3.5 Oberflächenerweiterung

Nachdem in diesem Abschnitt die Implementierung verschiedener neuer Funktionen des Mindmap-Editors beschrieben wurde, müssen diese über die Oberfläche benutzbar sein. Um *Knoten* mit speziellen *Knoteninhalten* zu verwenden, muss das in Unterabschnitt 3.1.3 Mindmap-Editor vorgestellte Ribbon-Menü erweitert werden. Um *Knoten* miteinander zu verbinden, ist eine weitere Erweiterung der Benutzeroberfläche notwendig.

5.3.5.1 Knotenmenü für Knotentypen

In Unterabschnitt 5.3.1 Knotenfabrik wurde erwähnt, dass das grafische Menü des *MindGraph*-Editors die Inhaltstypen auswählen kann. Dafür muss es die `SelectNodeType`-Methode der `IMapMenu`-Instanz von `MindGraphView` mit der entsprechenden Inhaltskennung aufrufen. Um diese Methode aufzurufen, muss der Benutzer über das grafische Menü den Knotentyp auswählen können. In Unter-Unterabschnitt 4.3.4.4 Oberflächenerweiterung wurde ein Bildschirmausschnitt der Oberfläche gezeigt, die um Zusammenarbeit ermöglichende Komponenten erweitert wurden. Um eine Möglichkeit zu haben, Knotentypen auszuwählen, enthält die Menüleiste eine neue Auflistung aller vorhandener Knotentypen, die ausgewählt werden können. In dem Bildschirmausschnitt in Abbildung 54 ist diese Auflistung dargestellt.

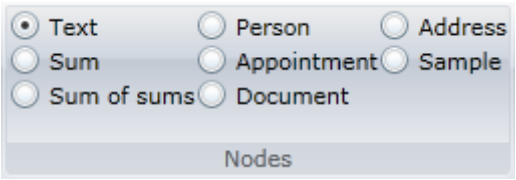


Abbildung 54: Auswahlmöglichkeit für Knotentypen im grafischen Menü des *MindGraph*-Editors

Wenn der Knotentyp "Text" ausgewählt ist, werden die bekannten Textknoten erstellt. Die Auswahl der Knotentypen "Person", "Appointment", "Document" und "Address" erstellt *Knoten*, die die in Unterabschnitt 5.3.3 CRM-Knoten vorgestellten speziellen Knoteninhalte enthalten. Die Knotentypen "Sum", "Sum of sums" und "Sample" sind *Knoten* mit denen die in Unterabschnitt 5.3.3 CRM-Knoten beschriebene `ConnectionContext`-Funktionalität manuell getestet werden kann. Da sich diese Testknoten nicht auf die CRM-Domäne beziehen, werden sie nicht weiter behandelt. In der XAML-Datei des grafischen Menüs ist die Knotenauswahl als Gruppe von Radiobuttons implementiert, wie Codeausschnitt 24 zeigt. In dieser Gruppe kann zu jeder Zeit nur ein Radiobutton ausgewählt sein.

```
<RadioButton x:Name="nodeText" Content="Text" Tag=""
      GroupName="n" />
<RadioButton x:Name="nodeSum" Content="Sum" Tag="sum"
      GroupName="n" RadioButton>
<RadioButton x:Name="nodeSumSum" Content="Sum of sums" Tag="sumsum"
      GroupName="n" />
<RadioButton x:Name="nodePerson" Content="Person" Tag="person"
      GroupName="n" />
<RadioButton x:Name="nodeAppoint" Content="Appointment" Tag="appointment"
      GroupName="n" />
<RadioButton x:Name="nodeDoc" Content="Document" Tag="document"
      GroupName="n" />
<RadioButton x:Name="nodeAddress" Content="Address" Tag="address"
      GroupName="n" />
<RadioButton x:Name="nodeSample" Content="Sample" Tag="sample"
      GroupName="n" />
```

Codeausschnitt 24: Knoteninhaltstypen als RadioButtons im grafischen Menü

Über die `Tag`-Eigenschaft der `RadioButton`-Objekte ist der `ContentTypeString` des zu erzeugenden Knotentyps definiert. Mit diesem `ContentTypeString` können Instanzen der registrierten Knoteninhaltstypen erzeugt werden, wie in Unterabschnitt 5.3.1 Knotenfabrik demonstriert wurde. Dort wurde gezeigt, dass Knoteninhaltstypen mit einer Inhaltskennung über die `RegisterContent`-Methode registriert werden können. Diese Inhaltskennung entspricht dem `ContentTypeString`-Wert. Über diese Zuordnung kann die Erzeugung der *Knoteninhalte* von den Factory-Methoden übernommen werden, wobei die Art des zu erzeugenden *Knoteninhalts* über das grafische Menü festgelegt werden kann. Alle Radiobuttons besitzen den gleichen Eventhandler, der in Codeausschnitt 25 dargestellt ist.

```
private void node_Checked(object sender, RoutedEventArgs e)
{
      this.menu.SelectNodeType((sender as RadioButton).Tag.ToString());
}
```

Codeausschnitt 25: Behandlunhg der Benutzerauswahl eines Knoteninhaltstyps

Wenn ein Radiobutton ausgewählt wird, dann wird der Werte dessen `Tag`-Eigenschaft ermittelt. Da diese Eigenschaft der Knoteninhaltskennung entspricht, kann mit ihr die `SelectNodeType`-Methode bei der Menü-Instanz aufgerufen werden. Alle neuen *Knoten* werden anschließend mit dem Inhalt erzeugt, der durch die Radiobuttons im grafischen Menü festgelegt wurde.

5.3.5.2 Knotenannotation

Um *Knoten* miteinander zu verbinden, wie in Unterabschnitt 5.3.2 Verbindungsfunktion beschrieben, wird eine Benutzerinteraktion benötigt. Diese könnte über das grafische Menü des Mindmap-Editors erfolgen. Eine andere Möglichkeit besteht in der direkten Interaktion mit den *Knoten* selbst, die verbunden werden sollen. In dem *MindGraph*-Editor ist letztere Variante implementiert. Da *Knoten* sich per Drag-and-Drop auf der Zeichenfläche der Mindmap frei bewegen lassen, müssen sie durch eine spezielle Interaktion über das Erstellen einer Verbindung informiert werden. Um diese Interaktion zu ermöglichen, verwendet der *MindGraph*-Editor Annotationen, die auf einem *Knoten* der Mindmap positioniert werden können und eine besondere Funktion besitzen. Der in der Vorarbeit entwickelte Editor (Unterabschnitt 3.1.3 Mindmap-Editor) nutzt mithilfe dieses Konzepts einen Unterknotenzuklapper. Über ein halbkreisförmiges grafisches Element an einer Seite eines *Knotens*, können bei einem Klick auf dieses Element sämtliche Unterknoten des *Knotens*

ausgeblendet werden. In Abbildung 55 sind die verwendeten Annotationen an rechten Rand der Knoten erkennbar.

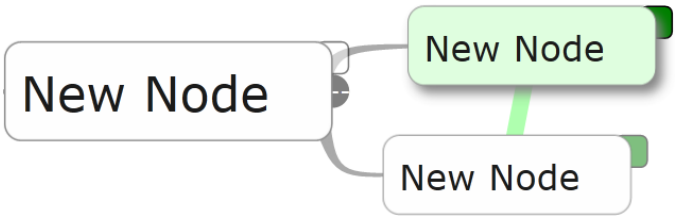


Abbildung 55: Grafische Annotationen als Bestandteil von Mindmap-*Knoten*

Das Rechteck am oberen rechten Rand der *Knoten* ist eine neue Annotation, über die Verknüpfungen zwischen *Knoten* verwaltet werden können. Bei einem Klick in dieses Rechteck wird wie in Unterabschnitt 5.3.2 Verbindungsfunktion beschrieben, die `UserTriggeredConnect`-Methode von `ItemConnector` aufgerufen. Auf diese Weise lassen sich die Verbindungen über die direkte Interaktion mit den *Knoten* herstellen und wieder entfernen. Die Annotation wird als `ConnectAnnotation`-Objekt dem entsprechenden `BasicItem` über die `AddAnnotation`-Methode zugewiesen. Dafür muss es die Schnittstelle `IAnnotation` implementieren. Die erwähnte Unterknotenzuklapper-Annotation ist als `HideAnnotation`-Objekt implementiert. Mithilfe von Annotationen können *Knoten* als `BasicItem`-Objekte um beliebige grafische Elemente erweitert werden und verhindern somit, dass die `BasicItem`-Klasse für jede grafische Erweiterung geändert werden muss. Dabei können diese Erweiterungen unabhängig voneinander entwickelt werden. Trotz der erwähnten Trennung können sie auf den Zustand des `BasicItem`-Objekts zugreifen und diesen bei Bedarf ändern. So färbt sich beispielsweise das Rechteck des `ConnectAnnotation`-Objekts grün, wenn der *Knoten*, zu dem es gehört, über mindestens eine sekundäre Verbindung zu einem anderen *Knoten* verfügt.

5.4 Ergebnis der zweiten Iteration

In diesem Kapitel wurde die Modellierung einer CRM-Domäne behandelt. Dabei wurde in dem Abschnitt 5.1 Anforderungen festgelegt, dass dafür ein neuer Knotentyp benötigt wird, der frei definierbare *Knoteninhalte* darstellen kann. Damit diese *Knoten* auch spezielle Bedeutungen haben können, müssen sie mit anderen *Knoten* in Verbindung gesetzt werden können und zwar unabhängig von der Topologie der Mindmap. In Abschnitt 5.2 Entwurf wurde geplant, wie *Knoteninhalte* in *Knoten* eingesetzt werden können und wie Knoteninhaltstypen und *Knoten* dafür aufgebaut sein müssen. Es wurde gezeigt, dass dafür eine gemeinsame Schnittstelle benötigt wird, die auch die getrennte und austauschbare Implementierung der Inhalte ermöglicht. Der benötigte Container-*Knoten* ist damit genau wie der bereits vorhandene Textknoten ein *Knoten* in der Mindmap und kann daher genauso verwendet werden.

Als Erweiterung wurde geplant, wie die bestehende Funktionalität zum Zeichnen und Andocken von Linien verwendet werden kann, um Knotenpaare gezielt miteinander grafisch zu verbinden. Diese Verbindungen sollen es ermöglichen, *Knoten* gezielt Bedeutung beizumessen, die durch die baumartige Struktur einer Mindmap nicht ausgedrückt werden kann. Als spezielle Knotentypen wurde eine beispielhafte CRM-Domäne geplant, die aus Personen, Terminen, Adressen und Dokumenten besteht. Diese Objekte können in Form von Modulen als Knoteninhaltstypen implementiert werden, wie in Abschnitt 5.3 Implementierung gezeigt wurde. Dabei ist es möglich, dass Knoteninhaltstypen außerhalb des *MindGraph*-Editors entwickelt werden können. Die

Erzeugung von Mindmap-*Knoten* mit diesen Inhalten kann über das grafische Menü eingestellt werden. Die Anpassung des Editors in Bezug auf die getrennte Erzeugung von *Knoten* und *Knoteninhalten* ist ebenfalls in der Implementierung beschrieben. Um in diesen Modulen auf die benachbarten *Knoten* zuzugreifen, wurde eine Variante konzipiert und implementiert, die mithilfe eines Verbindungskontexts Änderungen der verbundenen *Knoten* überwacht und Möglichkeiten zur Behandlung zur Verfügung stellt.

Das grafische und logische Verbinden von zwei *Knoten* ist dafür entsprechend implementiert worden. Dabei können Knotenpaare, wie in Unterabschnitt 5.3.2 Verbindungsfunktion beschrieben, keine Verbindungen mit sich selbst eingehen, noch gerichtet verbunden werden. Eine solche Implementierung wurde nicht direkt gefordert und kann daher zu einem späteren Zeitpunkt nachgeholt werden. Eine außerplanmäßige Implementierung dieser Funktion konnte aufgrund der zeitlich beschränkten Implementierungsphase nicht durchgeführt werden. Des Weiteren wurde die Beispieldomäne bewusst sehr einfach gewählt. Die Implementierung der CRM-*Knoten* in Unterabschnitt 5.3.3 CRM-Knoten ist daher nur als Prototyp mit einer primitiven Datenanbindung vorgenommen worden, die lediglich Anbindungsmöglichkeiten demonstriert. Um den *MindGraph*-Editor produktiv einsetzen zu können, ist eine spezielle Anbindung an eine spezielle CRM-Domäne notwendig, die im Rahmen dieser Thesis nicht untersucht wurde. Es wurde lediglich anhand einer beispielhaften Implementierung gezeigt, dass produktive Knotentypen realisiert werden können. Die dafür nötige Umgebung ist vorhanden, wie der in Bezug auf die Mindmap kontextsensitive Terminknoten aus Unter-Unterabschnitt 5.3.3.2 AppointmentModule zeigt.

Der Personenknoten aus Unter-Unterabschnitt 5.3.3.1 PersonModule nutzt eine externe Datenanbindung, die Informationen über Personen aus dem Speicher lädt. Alle vier Knoteninhaltstypen veranschaulichen die Verwendung von beliebigen Oberflächen, die unabhängig von Mindmap-Funktionalität gestaltet und implementiert werden können und sich trotzdem in Container-*Knoten* integrieren lassen. Dank dieser Integration können die neuen Modellierungsmöglichkeiten angepasst werden, damit sie mit den gemeinschaftlichen Funktionen aus Kapitel 4 Gemeinschaftliche Modellierung genutzt werden können. Das Ergebnis ist ein Mindmap-Editor, der die Modellierung einer demonstrativen CRM-Domäne in einer gemeinschaftlichen Art und Weise ermöglicht und unterstützt.

5.4.1 Anforderungstreue

Ob das Ergebnis der Implementierung der Domänenmodellierung die in Abschnitt 5.1 Anforderungen festgelegten Anforderungen erfüllt, wird in diesem Unterabschnitt behandelt. Dabei zeigt Tabelle 10 die Auflistung der Anforderungen und deren Erfüllt-Status.

ID #	Anforderung	Erfüllt
D1	Domänenknoten	Ja
D2	Kontextabhängigkeit	Ja
D3	Erweiterbarkeit	Ja
D4	Knotentypen erzeugen	Ja
D5	*Knoten* beliebig verknüpfen	Ja
D6	Beispielhafte CRM-*Knoten* verwenden	Ja

Tabelle 10: Anforderungen mit Implementierungsstatus

Es können beliebige *Knoten* in der Mindmap verwendet werden, wenn diese definiert und registriert wurden. Die Erzeugung dieser *Knoten* wird automatisch vorgenommen, wenn die Oberfläche die Inhaltskennung des zu erzeugenden *Knotens* für den *MindGraph*-Editor auswählt, wie in

Unterabschnitt 5.3.5 Oberflächenerweiterung beschrieben. Alle erzeugten *Knoten* können über sekundäre Verbindungen miteinander verknüpft werden. Da die vorgestellten CRM-Objekte beispielhaft als Knoteninhaltstypen implementiert wurden, ist auch ihre Datenanbindung demonstrativ realisiert. *Knoten* können somit auf ihre Umgebung in der Mindmap (wie der Terminknoten in Unter-Unterabschnitt 5.3.3.2 AppointmentModule) und auf ihre Umgebung in der CRM-Domäne (wie der Personenknoten in Unter-Unterabschnitt 5.3.3.1 PersonModule) reagieren. Damit können alle Anforderungen an die Domänenmodellierung als erfüllt angesehen werden.

5.4.2 Zeitlicher Ablauf

In diesem Unterabschnitt wird das Ergebnis des zweiten Teils dieser Thesis in Bezug auf die Bearbeitungszeit bewertet. In Unterabschnitt 3.3.2 Iteration 2 (CRM-Domäne) wurden für die zweite Iteration ein Bearbeitungszeitraum von fünf Wochen vorgesehen. In Tabelle 11 wird diese Planung mit der tatsächlichen Zeitaufwendung verglichen.

Tätigkeit	**Geplant**	**Tatsächlich**
Domänenobjekte konzipieren	1 Woche	1,5 Wochen
Editorerweiterung für Verwendung von Domänenobjekten	2 Woche	2,5 Wochen
Domänenobjekte identifizieren und implementieren	1 Woche	0,5 Wochen (CRM-Objekte)
Verbindung der Domänenobjekte mit Kontext	1 Woche	1 Woche (Mindmap Kontext)
		0,5 Wochen (Beispiel Kontext)
	5 Wochen	6 Wochen

Tabelle 11: Arbeitspakete der Iteration 2 verglichen mit tatsächlichem Zeitaufwand

Um die neue Struktur von speziellen Knotentypen und deren Inhalte zu planen, wurde die geplante Zeit um eine halbe Woche überzogen. Dies ist auf die Planung der umfangreichen Integration in die bestehende Erzeugungsstruktur für *Knoten* zurückzuführen, die ursprünglich nur Textboxen als *Knoteninhalt* unterstützt hat. Die eigentliche Implementierung des neuen Container-*Knoten* hat ebenfalls eine halbe Woche länger gedauert, als vorgesehen. Wie bereits erwähnt, wurde die CRM-Domäne absichtlich sehr einfach gewählt, sodass die entsprechende Realisierung der beispielhaften Knotentypen in der Hälfte der geplanten Zeit durchgeführt werden konnte. Auf diese Weise konnte ein Teil der Projektverspätung wieder aufgeholt werden.

Die letzte Tätigkeit in der zweiten Iteration bestand darin, die *Knoteninhalte* der CRM-*Knoten* mit dessen Kontext zu verbinden. Dafür war die Implementierung der `ConnectionContext`-Klasse nötig, um die Verbindungen der *Knoten* in der Mindmap den Modulen der Inhalte zur Verfügung stellen zu können. Die Terminknoten nutzen diese Anbindung. Eine Anbindung an einen externen Kontext konnte für den Personenknoten nur beispielhaft vorgenommen werden, indem eine Liste vorgegebener Testpersonen zur Laufzeit erstellt wird. Insgesamt hat diese Kontextanbindung eine halbe Woche länger gedauert als geplant. Obwohl die zweite Iteration damit um insgesamt eine Woche verlängert werden musste, konnte eine lauffähige Version des *MindGraph*-Editors erstellt werden. Diese Version ist als Prototyp entsprechend funktionsfähig und daher kann das Ergebnis der zweiten Iteration als erfolgreich betrachtet werden.

6 Fazit

In dieser Bachelor-Thesis wurde eine Lösung konzipiert, die das gemeinschaftliche Modellieren einer Mindmap ermöglicht, in der Daten aus einem beispielhaften CRM-System als *Knoten* verwendet werden können. Dafür wurden zwei Teilaufgaben betrachtet, wobei die erste die Konzeptionierung von Zusammenarbeit in Mindmaps betrifft. Diesbezüglich werden Benutzeraktionen im Mindmap-Editor als Operationen aufgefasst, die als *Ereignisse* an einen zentralen Server geschickt werden. Dieser Server erzeugt eine globale Eventsequenz aus allen *Ereignissen*, die er erhält. Clients können diese Sequenz in regelmäßigen Abständen abfragen und erhalten alle neuen *Events*, die seit der letzten Abfrage aufgetreten sind. Damit diese *Events* bei den Clients auf deren Zustand der Mindmap angewendet werden können, müssen sie atomar sein und über ein behandelbares Konfliktpotenzial verfügen. Die Granularität der atomaren Operationen, die die *Ereignisse* erzeugen, muss so gewählt werden, dass die *Events* nach einer Behandlung bei allen Teilnehmern der Zusammenarbeit zu einem konsistenten Zustand führen. In dieser Ausarbeitung wurde diese Lösung in einem Zusammenarbeit ermöglichenden Prototyp implementiert. Das Ergebnis dieser Implementierung wurde in Abschnitt 4.4 Ergebnis der ersten Iteration beschrieben. Die zweite Teilaufgabe dieser Thesis bestand aus der Konzeptionierung der Integration von Mindmaps mit CRM-Daten. Um Daten generell als *Knoten* verwenden zu können, muss eine Trennung von *Knoten* und *Knoteninhalt* erreicht werden. Knoteninhaltstypen können dann als Objekte der CRM-Domäne definiert werden. Durch primäre und sekundäre Verknüpfungen zwischen Knotenpaaren in der Mindmap-Topologie kann ein Kontext hergestellt werden, der Strukturen mit CRM-Objekten repräsentieren kann. Auf diesen Kontext können die Objekte reagieren und entsprechend ihrer Datenanbindung ihren Zustand verändern. Um die Integration mit einem CRM-System zu veranschaulichen, wurde ein Prototyp entwickelt, der beispielhaft Personen, Termine, Adressen und Dokumente als Knoten in Mindmaps verwenden kann, um komplexe Strukturen zu modellieren. Dabei wurden Kontextsensitivität und Datenanbindung an eine CRM-ähnliche Datenquelle demonstriert. Das Ergebnis dieser Implementierung wurde in Abschnitt 5.4 Ergebnis der zweiten Iteration beschrieben.

In Bezug auf den konzeptionellen Teil dieser Thesis, können die Kernaussagen wiefolgt zusammengefasst werden. Zusammenarbeit in Mindmaps kann durch ein ereignisorientiertes Client-Server-System mit geeignet gewählten *Ereignissen* und deren Ausführung ermöglicht werden. Was die Modellierung der CRM-Domäne betrifft, so können durch die Trennung von *Knoten* und *Knoteninhalt*, CRM-Objekte mit eigener Benutzeroberfläche, eigenem kontextsensitiven Verhalten und eigener Datenanbindung definiert werden.

Mit der prototypischen Implementierung beider Konzepte konnte ein funktionsfähiger Mindmap-Editor erstellt werden, der es dem Benutzer ermöglicht, kundenorientierte Mindmaps in Echtzeit-ähnlicher Zusammenarbeit zu erstellen. Die erstellten Strukturen sind automatisch mit Daten in einem CRM-System verbunden, sodass Informationen eines Kundenprojekts aus der Mindmap direkt im Hintergrundsystem gespeichert werden. Es ist keine manuelle Übertragung der Informationen mehr notwendig. Auch wenn in diesem Prototyp nur eine einseitige Anbindung an eine beispielhafte Datenquelle implementiert wurde, konnte die nötige Infrastruktur für alternative Anbindungen vollständig realisiert werden. Abschließend ist zu erwähnen, dass diese Bachelor-Thesis als Grundlage für die Integration der vorgestellten Konzepte in Anwendungen der CAS Software AG, sowie der Entwicklung von weiterführenden Funktionalitäten und Anbindungen dient. Aus diesem Grund musste die Ausarbeitung ausführlicher ausfallen als ursprünglich geplant und konnte daher nicht auf 60 Seiten reduziert werden.

7 Literaturverzeichnis

[Buz02] Buzan, Tony und Buzan, Barry. 2002. *Das Mind-Map Buch.* s.l. : mvg Verlag, 2002. ISBN 978-3636062437.

[Car99] Carstensen, Peter H. und Schmidt, Kjeld. 1999. CSCW: New Challenges to Systems Design. *Forside - IT-Universitetet i København.* [Online] 1999. [Zitat vom: 16. Dezember 2009.] http://www.itu.dk/~schmidt/papers/cscw_intro.pdf.

[CAS09] CAS Software AG. 2009. *CAS genesisWorld Server-SDK - Schnittstellen am Applikationsserver.* [Dokument aus dem CAS Intranet] Karlsruhe : CAS Software AG, 2009.

[CAS10] —. CRM Standardlösungen - CAS Software AG - deutscher Marktführer im Mittelstand. *CAS Software AG - deutscher Marktführer im Mittelstand.* [Online] [Zitat vom: 8. März 2010.] http://www.cas.de/Produkte/Uebersicht-CRM-Standardloesungen.asp.

[Mar02] Fowler, Martin. 2002. Domain Model. *Catalog of "Patterns of Enterprise Application Architecture" Book.* [Online] Martin Fowler, 15. November 2002. [Zitat vom: 19. Februar 2010.] http://martinfowler.com/eaaCatalog/domainModel.html.

[Gro07] Gross, Tom und Koch, Michael. 2007. *Computer-Supported Cooperative Work.* [Hrsg.] Michael Herczeg. München : Oldenbourg Wissenschaftsverlag GmbH, 2007. ISBN 978-3-486-58000-6.

[Kör07] Körner, Heiko. 2007. *Algorithmen auf Graphen.* [Skript zur Vorlesung Alogrithmen] Karlsruhe : s.n., 2007.

[Kuh07] Kuhrmann, Marco und Beneken, Gerd. 2007. *Windows Communication Foundation; Konzepte, Programmierung, Migration.* München : Spektrum Akademischer Verlag, 2007. IDBN 978-3-8274-1598-1.

[Juv08] Löwy, Juval. 2008. *Programming WCF Services, Second Edition.* s.l. : O'Reilly Media, 2008. ISBN 978-0-596-52130-1.

[Rob05] Martin, Robert C. 2005. The OLD Object Mentor blog site. *The Three Rules Of Tdd.* [Online] Object Mentor, 7. Oktober 2005. [Zitat vom: 6. Februar 2010.] http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd.

[Mic105] Microsoft. Anonymous Methods (C# Programming Guide). *MSDN Visual C# Developer Center.* [Online] [Zitat vom: 4. März 2010.] http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx.

[Mic09] —. 2009. CRM Software for Customer Relationship Management - Microsoft Dynamics CRM. *White Paper: CRM in a Challenging Economy.* [Online] 8. April 2009. [Zitat vom: 19. Februar 2010.] http://crm.dynamics.com/docs/CRM_Investment_in_a_Down_Economy_FINAL.pdf.

[Mic103] —. Data Binding. *MSDN Silverlight Developer Center.* [Online] [Zitat vom: 3. März 2010.] http://msdn.microsoft.com/en-us/library/cc278072(VS.95).aspx.

[Mic106] —. Featurevergleich zwischen Silverlight und WCF. *MSDN Silverlight Developer Center.* [Online] [Zitat vom: 4. März 2010.] http://msdn.microsoft.com/de-de/library/dd470110(VS.95).aspx.

[Mic] —. FrameworkElement-Klasse. *MSDN Silverlight Developer Center.* [Online] [Zitat vom: 3. März 2010.] http://msdn.microsoft.com/de-de/library/system.windows.frameworkelement(VS.95).aspx.

[Mic091] —. 2009. Lambda Expressions (C# Programming Guide). *MSDN Visual C# Developer Center.* [Online] Juli 2009. http://msdn.microsoft.com/en-us/library/bb397687.aspx.

[Mic10] —. 2010. Microsoft Silverlight. *MSDN Silverlight Developer Center.* [Online] 2010. [Zitat vom: 4. März 2010.] http://msdn.microsoft.com/de-de/silverlight/default.aspx.

[Mic104] —. Properties (C# Programming Guide). *MSDN Visual C# Developer Center.* [Online] [Zitat vom: 4. März 2010.] http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx.

[Mic101] —. UIElement-Klasse. *MSDN Silverlight Developer Center.* [Online] [Zitat vom: 3. März 2010.] http://msdn.microsoft.com/de-de/library/system.windows.uielement(VS.95).aspx.

[Mic102] —. UserControl-Klasse. *MSDN Silverlight Developer Center.* [Online] [Zitat vom: 3. März 2010.] http://msdn.microsoft.com/de-de/library/system.windows.controls.usercontrol(VS.95).aspx.

[Nau08] Naujoks, Manuel. 2008. XAML - WPF per XML (Seminararbeit). *Website von Manuel Naujoks.* [Online] 1. Februar 2008. [Zitat vom: 7. Januar 2010.] http://neokc.de/aktuelles/dokumente/XAML-Ausarbeitung.pdf.

[Wen09] Rasmussen, Lars. 2009. Went Walkabout. Brought back Google Wave. *Official Google Blog.* [Online] 28. Mai 2009. [Zitat vom: 6. Januar 2010.] http://googleblog.blogspot.com/2009/05/went-walkabout-brought-back-google-wave.html.

[Ric87] Richman, Louis S. und Slova, Julianne. 1987. SOFTWARE CATCHES THE TEAM SPIRIT New computer programs may soon change the way groups of people work together -- and start delivering the long-awaited payoff from office automation. *Fortune 500 Daily & Breaking Business News - FORTUNE on CNNMoney.com.* [Online] 8. Juni 1987. [Zitat vom: 7. Januar 2010.] http://money.cnn.com/magazines/fortune/fortune_archive/1987/06/08/69109/index.htm.

Anhang

1. Glossar

Event/Ereignis

Das Ausführen einer atomaren Operation verursacht ein *Event*. Dieses *Event* wird an den Zusammenarbeit ermöglichenden *piamind*-Dienst geschickt. *Events* können von diesem Dienst abgefragt werden um über Veränderungen benachrichtigt zu werden. *Event* und *Ereignis* sind im Kontext von *piamind* Synonyme und dürfen nicht mit .NET-Events verwechselt werden.

Knoten

Knoten sind Bestandteile der Mindmap und besitzen einen Inhalt. Bei normalen *Knoten* ist dieser Inhalt Text. Erweiterte *Knoten* in Rahmen von *MindGraph*, können frei definierbare *Knoteninhalte* haben. Der Knotentyp eines *Knoten* gibt Auskunft über dessen Inhalt und kann daher auch verwendet werden um auf Typ des *Knoteninhalts* zu verweisen.

Knoteninhalt

Knoteninhalt stellt den Inhalt eines *Knoten* dar. Spezielle *Knoten* können in *MindGraph* beliebige Inhalte haben, die in Silverlight modelliert werden können. Damit lassen sich beispielsweise Objekte einer Domäne als *Knoteninhalte* definieren. *Knoteninhalte* sind dabei konkrete Objekte eines Knoteninhaltstypen.

Mindmap

Ein Dokument, das Punkte enthält, die in einer radialen Weise angeordnet sind und so die Organisation von Informationen grafisch unterstützt, wird Mindmap genannt.

MindGraph

MindGraph ist der Mindmap-Editor zur einfachen Erstellung und Bearbeitung von Mindmaps. Dieser Editor wurde in der Vorarbeit zur Bachelor-Thesis erstellt und dient als Grundlage der in dieser Thesis betrachteten Mindmap-Erweiterungen in Bezug auf Zusammenarbeit und Domänenmodellierung.

piafile

Das Web-basierte Dateisystem *piafile* von *MindGraph* wird benutzt, um Mindmaps online abzuspeichern und von dort zu laden.

piamind

Die Infrastruktur, die aus den Diensten zur Zusammenarbeit und dem *MindGraph*-Client selbst besteht, wird als *piamind*-System oder *piamind*-Infrastruktur bezeichnet.

sEvent

Das in *MindGraph* eingesetzte *Ereignis*-Framework, das gemeinschaftliche Sitzungen über das Internet ermöglicht, wird mit *sEvent* bezeichnet. Die Endpunkte des Frameworks wurden mit den *piamind*-Diensten verbunden, sodass *Ereignisse* ausgeführt und behandelt werden können.

2. Liste aller Testfälle für piamind-Dienste

Class Name	Test Name
ServiceAdmTest	TestingWorks
ServiceAdmTest_CheckForNewClients	AcceptClient_NoSessionNoUser_NotOnline
ServiceAdmTest_CheckForNewClients	AcceptClient_OneUser_Online
ServiceAdmTest_CheckForNewClients	AcceptClient_OneUser_OnlineAsAccepted
ServiceAdmTest_CheckForNewClients	AcceptClient_OneUser_OnlineNotAsAccepted
ServiceAdmTest_CheckForNewClients	AcceptClient_OneUser_OnlineWithContent
ServiceAdmTest_CheckForNewClients	AcceptClient_OnlyAuthor_AuthorAccepted
ServiceAdmTest_CheckForNewClients	AcceptClient_SessionNoUser_NotOnline
ServiceAdmTest_CheckForNewClients	DeclineClient_NoSessionNoUser_NotOnline
ServiceAdmTest_CheckForNewClients	DeclineClient_OneUserOffline_NotOnline
ServiceAdmTest_CheckForNewClients	DeclineClient_OneUserOnline_NotOnline
ServiceAdmTest_CheckForNewClients	DeclineClient_OneUserOnlineOfflineOnline_LatestContent
ServiceAdmTest_CheckForNewClients	DeclineClient_SessionNoUser_NotOnline
ServiceAdmTest_CheckForNewClients	GetAllClients_ClientWithSessionNameId_ValidParams
ServiceAdmTest_CheckForNewClients	GetAllClients_NoClients_EmptyList
ServiceAdmTest_CheckForNewClients	GetAllClients_NoClientsNoSession_EmptyList
ServiceAdmTest_CheckForNewClients	GetAllClients_OneClient_ListWithUserId
ServiceAdmTest_HandleGlobalSession	CloseSession_SameIdAgain_NewSession
ServiceAdmTest_HandleGlobalSession	CreateSession_AmbiguousId_NoSession
ServiceAdmTest_HandleGlobalSession	CreateSession_UniqueId_NewSession
ServiceAdmTest_HandleGlobalSession	GetAllClients_SessionsWithAuthor_AuthorName
ServiceAdmTest_HandleGlobalSession	GetAllSessions_NoSessions
ServiceAdmTest_HandleGlobalSession	GetAllSessions_OneSession_OneSession
ServiceAdmTest_HandleGlobalSession	GetAllSessions_OneSession_ThatSession
ServiceAdmTest_HandleGlobalSession	GetAllSessions_TwoSessionsDifferenNames_ThatSessions
ServiceAdmTest_HandleGlobalSession	GetAllSessions_TwoSessionsSameName_OnlyOneSession
ServiceAdmTest_JoiningHadshake	IsOnline_NoSessionNoUser_NotOnline
ServiceAdmTest_JoiningHadshake	IsOnline_OneSessionNoUser_NotOnline
ServiceAdmTest_JoiningHadshake	IsOnline_OneUser_NotOnline
ServiceAdmTest_JoiningHadshake	IsOnline_OneUser_Online
ServiceAdmTest_JoiningHadshake	Register_ExistingSession_TwoDifferentUserIds
ServiceAdmTest_JoiningHadshake	Register_ExistingSession_TwoValidUserIds
ServiceAdmTest_JoiningHadshake	Register_ExistingSession_UserId
ServiceAdmTest_JoiningHadshake	Register_NoSession_NoUserId
ServiceAdmTest_JoiningHadshake	Register_TowSessions_OneUserOnEach
ServiceColSessionTest	GetGlobalEvents_TwoEventsByDifferentUsers_EventsContainUserInfo
ServiceColSessionTest	GetGlobalEvents_TwoEventsOnSingleSessions_HeaderContainsSessionId
ServiceColSessionTest	GetGlobalEvents_TwoEventsOnTwoSessions_OneEventOnEach
ServiceColTest	GetGlobalEvents_NoSamples_NoChanges
ServiceColTest	GetGlobalEvents_OneSample_NoChanges
ServiceColTest	GetGlobalEvents_OneSample_OneChanges
ServiceColTest	GetGlobalEvents_ThreeSamples_NoChanges
ServiceColTest	GetGlobalEvents_TwoSamples_OneChange

ServiceColTest	GetGlobalEvents_TwoSamples_TwoChanges
ServiceColTest	InvokeEventP1_Sample_NewNodeEvent
ServiceColTest	InvokeEventP2_Sample_NewNodeEvent
ServiceColTest	InvokeEventP3_Sample_NewNodeEvent
ServiceColTest	InvokeEventP3_SampleArgs_ArgsStored
ServiceColTest	InvokeEventP3_TwoEventsDiffNumOfArgs_ArgsStored

3. Liste aller Testfälle für sEvent

Class Name	Test Name
EventProxySessionTest	Login_LoggedInInvoke1Param_UserAndSessionContext
EventProxySessionTest	Login_LoggedInInvoke2Param_UserAndSessionContext
EventProxySessionTest	Login_LoggedInOnClosedSession_SessionContext
EventProxySessionTest	Login_LoggedInTwoSessions_FirstSessionContextCorrect
EventProxySessionTest	Login_LoggedInTwoSessions_SecondSessionContextCorrect
EventProxySessionTest	Login_LoggedInTwoUsers_UserContextsCorrect
EventProxySessionTest	Login_NotLoggedIn_Exception
EventProxyTest	GetAggregateRequestor_CalledAfterOneEvent_SameEventInAggregate
EventProxyTest	GetAggregateRequestor_CalledAfterTwoDifferentEvents_BothEventsInAggregate
EventProxyTest	GetAggregateRequestor_CalledAfterTwoDifferentEvents_OnlyOneRegisteredHandleCalled
EventProxyTest	GetAggregateRequestor_CalledAfterTwoSameEvents_BothEventsHandled
EventProxyTest	GetAggregateRequestor_CalledAfterTwoSameEvents_BothEventsWithParams
EventProxyTest	GetAggregateRequestor_HandleAll_BeforeSpecificHandlers
EventProxyTest	GetAggregateRequestor_NoEvent_NoHandlerCalled
EventProxyTest	GetAggregateRequestor_OneEventAggregateReq_HandlerCalled
EventProxyTest	GetAggregateRequestor_OneEventNoAggregateReq_NoHandlerCalled
EventProxyTest	GetAggregateRequestor_SkipFourEvents_HandlerOfNoEventCalled
EventProxyTest	GetAggregateRequestor_SkipNoEvent_HandlerOfBothEventsCalled
EventProxyTest	GetAggregateRequestor_SkipOneEvent_HandlerOfSecondEventCalled
EventProxyTest	GetAggregateRequestor_TwoDifferentEvents_BothHandledAtOnce
EventProxyTest	GetAggregateRequestor_TwoDifferentEvents_HandleCalledOnce
EventProxyTest	GetAggregateRequestor_TwoDifferentEvents_HandleOnceAndSpecificCalled
EventProxyTest	GetEventInvoker_EmptyString_HandlerCalledEmptyString
EventProxyTest	GetEventInvoker_NullString_HandlerCalledNullString
EventProxyTest	GetEventInvoker_SameIdTwice_Exception
EventProxyTest	GetEventInvokerInt_CalledWithInt_Callback
EventProxyTest	GetEventInvokerIntString_CalledWithIntString_Callback
SessionManagerTest	AcceptUser_SessionsWithUser_Accepted
SessionManagerTest	AcceptUser_SessionsWithUser_UserInitiallyUnaccepted
SessionManagerTest	AcceptUser_UserInClosedSession_NoUserAnymore
SessionManagerTest	CloseSession_CalledWithNoSessions_NoSessions
SessionManagerTest	CloseSession_CalledWithSession_NoSessionsAnymore
SessionManagerTest	CreateSession_CalledWithAuthorName_Authorname
SessionManagerTest	CreateSession_CalledWithEmptyAuthorName_Exception
SessionManagerTest	CreateSession_CalledWithEmptySessionName_Callback
SessionManagerTest	CreateSession_CalledWithExistingSession_NoSessionNameReturned

SessionManagerTest	CreateSession_CalledWithNull_Callback
SessionManagerTest	CreateSession_CalledWithSample_Callback
SessionManagerTest	CreateSession_CalledWithSampleTwice_OneSession
SessionManagerTest	DeclineUser_AcceptedUser_UserUnaccepted
SessionManagerTest	DeclineUser_UserInClosedSession_NoUserAnymore
SessionManagerTest	GetAllSessions_CalledWithNoSessions_NoSessions
SessionManagerTest	GetAllSessions_CalledWithOneSession_OneSession
SessionManagerTest	GetAllSessions_CalledWithTwoSession_TwoSession
SessionManagerTest	GetAllUsers_ClosedSession_ZeroUsers
SessionManagerTest	GetAllUsers_NoUserRegistered_ZeroUsers
SessionManagerTest	GetAllUsers_OneUserRegistred_OneUser
SessionManagerTest	GetAllUsers_TwoUserRegistered_TwoUser
SessionManagerTest	GetVeryAllUsers_NoSessions_NoUsers
SessionManagerTest	GetVeryAllUsers_NoUserRegistered_ZeroUsers
SessionManagerTest	GetVeryAllUsers_TwoSessionsOneWithUser_OneUser
SessionManagerTest	GetVeryAllUsers_TwoSessionsWithOneUserEach_TwoUser
SessionManagerTest	IsOnline_AcceptedUser_GetsContentOnlyOnce
SessionManagerTest	IsOnline_AcceptedUser_GetsContentOnlyOnceUntilNextAcception
SessionManagerTest	IsOnline_AcceptedUserInClosedSession_UserContent
SessionManagerTest	IsOnline_DeclinedAcceptedUser_OfflineAndNoUserContent
SessionManagerTest	IsOnline_DeclinedUser_GetsOfflineResponse
SessionManagerTest	IsOnline_UnacceptedUser_NotOnline
SessionManagerTest	IsOnline_UnacceptedUserInClosedSession_NoUserAnymore
SessionManagerTest	RegisterUser_OnClosedSession_InvalidId
SessionManagerTest	RegisterUser_OnSession_NewUserWithId
SessionManagerTest	RegisterUser_TwiceOnSession_TwoUsersWithIds
SessionManagerTest	RegisterUser_TwiceWithSameUsername_TwoUsersWithSameUsername
SessionManagerTest	RegisterUser_TwoUsersOnTwoSessions_EachSessionOneUser
SessionManagerTest	RegisterUser_WithEmptyUsername_Exception
SessionManagerTest	RegisterUser_WithUsernameNull_Exception
SessionManagerTest	RegisterUser_WithUsernameOnSession_SameUsername