UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

CoMICS II
Ein event-orientiertes Mikrosimulationssystem

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von
Peter Hassenpflug und Carsten Busch

Erstgutachter:	Prof. Dr. Klaus G. Troitzsch Institut für Wirtschafts- und Verwaltungsinformatik, Fachbereich Informatik
Zweitgutachter:	Dr. Michael Möhring Institut für Wirtschafts- und Verwaltungsinformatik, Fachbereich Informatik

Koblenz, im September 2008

Erklärung

Wir versichern, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek sind wir einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimmen wir zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

.....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Mit dem Projektpraktikum „MicSim“ wurde 2004 der Grundstein für ein auf Java basierendes Mikrosimulationssystem gelegt. Für das gleichnamige Produkt diente UMDBS von Prof. Dr. Thomas Sauerbier, Fachhochschule Giessen-Friedberg als Vorlage, welche eine Modellimplementierung in MISTRAL vorsieht. Da MISTRAL eine veraltete prozedurale Programmiersprache ist und somit nicht mehr dem objektorientierten Zeitgeist entspricht, entschloss sich die Projektgruppe eine gänzlich neue objektorientierte Umgebung zu erstellen. Diese sollte sowohl den Funktionsumfang von UMDBS beinhalten, als auch eine bequeme Möglichkeit bieten, objektorientierte Modelle einer Mikrosimulation in Java zu erstellen. Das Projektpraktikum endete 2005 mit einem lauffähigen objektorientierten Nachfolger von UMDBS. Da jedoch MicSim noch einige Schwachstellen aufwies, wurde dieses System in der Studienarbeit „MicSim - Agentenbasierte Mikrosimulation“ von Pascal Berger, Dennis Fuchs, Peter Hassenpflug und Christian Klein 2006 weiterentwickelt und zu dem heutigen „CoMICS“ umbenannt.[22, S.5] Jedoch konnten nicht alle Probleme der Software beseitigt werden. So blieb das schlecht skalierende Ablaufverhalten einer Simulation und die unzureichende Unterstützung des Benutzers bei syntaktischen Programmierfehlern weiterhin bestehen. Dies führte zu dem Entschluss, ausgehend von CoMICS, die Idee eines java-basierten Mikrosimulationssystems weiter auszubauen. Im Zuge einer Analyse dieses Systems kristallisierten sich sehr schnell einige Punkte heraus, die uns verbesserungswürdig erschienen. Die gravierendste Veränderung stellte hierbei die Umstellung auf eine event-orientierte Mikrosimulation dar. Hierzu musste die komplette Modellstruktur, Ablauflogik und sogar jegliche Modellierungswerkzeuge neu entwickelt werden. All dies führte dazu, dass CoMICS

II von Grund auf neu entwickeln werden musste.

Diese Arbeite dokumentiert diese Entwicklung, hierzu wird im 1. Kapitel einige Grundlagen der Mikrosimulation näher ausgeführt, sowie CoMICS I kurz vorgestellt. Das 2. Kapitel beschäftigt sich mit technischen Details und Gegebenheiten, die in CoMICS II Anwendung finden. Die anfänglichen Überlegungen bis hin zu Architektur werden anschießend im Abschnitt 3 ausgeführt und leitet das Kapitel 4 ein, welche einen Einblick in die eigentliche Implementation gewährt. Abschließend wird der Stand der Entwicklung, sowie mögliche Ansatzpunkte für zukünftige Weiterentwicklungen beleuchtet.

Kapitel 2

Grundlagen

2.1 Modellierung und Simulation

2.1.1 Modell und Modellierung

Abbildung 2.1: Modellierung einer Person

Ein Modell ist eine abstrakte, immaterielle Repräsentation von natürlichen oder künstlichen Originalen, die selbst auch wieder Modelle sein können. Die Originale werden oft als Real-Objekte oder Realität, das abstrakte Abbild als Modell bezeichnet. Es ist besonders wichtig, nicht alle oder möglichst viele, sondern nur die wirklich erforderlichen Attribute der Realität abzubilden. Sehr häufig werden Modelle im Informatikbereich eingesetzt. Hier ist es ganz besonders wichtig, nur die notwendigen Attribute zu betrachten, da bei einer zu detaillierter Betrachtung unnötig viele Ressourcen auf den Computersystemen ausgenutzt werden würden. In der Regel kann man sich diese Modelle sehr gut vorstellen, da sie auf die wesentlichen Eigenschaften reduziert sind. Modellierung bezeichnet den Prozess der Abstraktion, wobei die besondere Schwierigkeit darin besteht zu entscheiden, welche

Attribute im geforderten Kontext weggelassen und welche verwendet werden sollten. Ergebnis der Modellierung ist immer ein Modell. Für unterschiedliche Anwendungen können vom gleichen realen Objekt ganz unterschiedliche Modelle erzeugt werden. [24] Überall in der Wissenschaft werden Modelle angewendet um komplexe Sachverhalte besser verstehen zu können. In der Quantenphysik z.B. wird das von Niels Bohr 1913 entwickelte, bohrsche Atommodell verwendet, um Effekte auf Atomebene anschaulich darstellen bzw. erklären zu können. Im Bereich der sozioökonomischen Wissenschaften werden Modelle von kompletten Gesellschaften oder Personengruppen gebildet. In unserem konkreten Anwendungsfall der Mikrosimulation werden, wie in Abbildung 2.1 dargestellt, zunächst Modelle der einzelnen Personen gebildet. Das Modell der Gesellschaft oder Personengruppe setzt sich dann aus einer repräsentativen Anzahl an Personenmodellen zusammen.

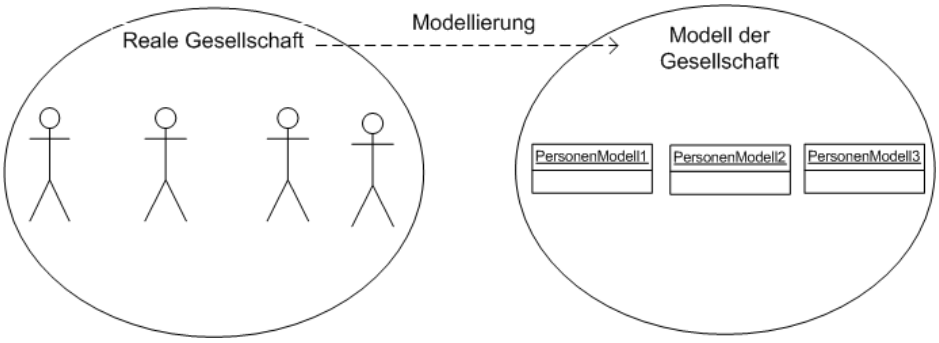


Abbildung 2.2: Modellierung der Gesellschaft

2.1.2 Simulation

Dieser Abschnitt orientiert sich an den Ausführungen aus [21, S.6-10].

2.1.2.1 Historischer Überblick

In den Sozialwissenschaften haben Computersimulationen bereits seit ca. 20 Jahren ihre Anwendung gefunden. Neben einigen Ausnahmen etablierten sich Simulationsmodelle in der Sozialforschung erst mit Einzug von Computersystemen in den Universitätsinstituten. Sie basierten zunächst auf diskreter Eventsimulation oder System Dynamics. Auf Basis von stochastischen Prozessen bildete sich um 1955 das Gebiet der Mikrosimulation heraus, welches in den letzten Jahren Software-

tools wie UMDBS von Prof. Dr. Thomas Sauerbier hervorgebracht hat. Genauere Ausführungen zum Thema Mikrosimulation werden im Abschnitt 2.2 behandelt.

Eine ganz andere Richtung nahm die Entwicklung von Simulationssystemen in den 90er Jahren. Aus dem Bereich der Künstlichen Intelligenz wurden Multi-Agenten-Systeme für die Sozialsimulation entdeckt. Diese basieren auf autonomen Agenten und ihrer Interaktion untereinander. Da dieses Simulationsmodell sehr intuitiv an der Realität angelehnt ist, hat es in den letzten Jahren im Bereich der Sozialsimulationen zunehmend Anwendung gefunden. Das Thema Multi-Agenten-Simulation wird im späteren Kapitel 2.3 noch genauer behandelt. Im Rahmen dieser Diplomarbeit soll das Simulationstool CoMICS auf eine neue Version ausgebaut werden. CoMICS arbeitet in der ersten Version intern als Mikrosimulation und bietet dem Simulationsanwender eine agentenbasierte, intuitive Sicht auf die Gesellschaft. Beide Konzepte wurde vereint um die Performance der Mikrosimulation und die Intuitive Bedienung der Multi-Agenten-Systeme zu erhalten. CoMICS bildet intern die Agenten auf Datenbankzeilen ab. Somit lassen sich große Gesellschaftsszenarien simulieren, die den Rahmen jedes MAS-Systems überschreiten würden. Eine genauere Beschreibung zur Funktionsweise der ersten CoMICS Version wird im Abschnitt 2.4 behandelt. In der zweiten Version von CoMICS wird das Konzept der diskreten Eventsimulation zur Steigerung des Laufzeitverhaltens mit ins Programm integriert. Mit der Vereinigung dieser verschiedenen Konzepte ist CoMICS II das zur Zeit einzige Mikrosimulationswerkzeug seiner Art.

2.1.2.2 Simulation sozialer Systeme

Wie im historischen Überblick bereits beschrieben, verwenden die Sozialwissenschaften bei der Simulationsentwicklung bereits fertige Konzepte, welche sich bereits in anderen Wissenschaften - wie z.B. Physik, Künstliche Intelligent oder Mathematik - durchgesetzt haben. Ohne Simulation lassen sich komplexere Gesellschaftsentwicklungen nur sehr schwer berechnen. Selbst wenn die einzelnen Individuen nur nach wenigen einfachen Regeln agieren, kann sich für die komplette Gesellschaft eine extrem komplexe nichtlineare Entwicklung ergeben. Konventionelle mathematische Verfahren basieren meist auf Annahmen von linearen Abhängigkeiten zwischen Variablen. Einen wissenschaftlichen Ansatz, nichtlineare Problemstellungen zu lösen, bietet die Komplexitätstheorie (Waldrop 1992, Kauffman 1995, Sole and Goodwin 2002). Eine wichtige Idee aus der Komplexitätstheorie ist die Formale Beschreibung der Emergenz, welche davon ausgeht, dass Verhaltensweisen

von Objekten auf einer Ebene Auswirkungen auf Objekte anderer Ebenen haben können. Mit diesem Ansatz lassen sich bereits viele soziale Systeme abbilden. Systeme, bestehend aus Menschen unterscheiden sich von anderen durch ihre Fähigkeit Wissen über andere Ebenen, das komplette System oder die Auswirkunen ihrer eigenen Entscheidungen zu haben. (Drogoul and Ferber 1994) In unserem Anwendungsfall der Mikrosimulation, MAS und event-orientierter Simulation werden die soziotechnischen Systeme modelliert und anschließend simuliert, die Ansätze zur direkten analytischen Berechnung werden im Rahmen dieser Diplomarbeit nicht weiter ausgeführt.

2.1.2.3 Zusammenfassung

Mit der Hilfe von Simulatonsmodellen ist es möglich, dynamische Sachverhalte innerhalb soziotechnischer Systeme modellieren zu können. Insbesondere geht es hierbei darum, wie sich einfache Verhaltensweisen einzelner Individuen, auf komplette soziotechnische Systeme auswirken können. Jedem Individuum werden überschaubare Entscheidungsmöglichkeiten zur Verfügung gestellt, zwischen denen es die Auswahl hat. Im Ergebnis kann aus den Einzelentscheidungen ein sehr komplexer Effekt auf das Gesamtsystem resultieren. Sozialwissenschaftler verwenden Simulationen um ihre Annahmen zu überprüfen, da in der Regel eine Beweisführung, wie in der klassischen Mathematik oder Physik, nicht möglich sind. Zunächst wird ein soziotechnisches Phänomen gesucht, welches genauer untersucht werden soll. Durch Abstraktion wird ein zu dem Phänomen analoges Modell erstellt. Auch hier sollten wiederum nur die Informationen verwendet werden, welche auch wirklich für die Untersuchung notwendig sind. Auf das resultierende Modell wird nun die eigentliche Simulation angewendet. Innerhalb der Simulation muss der Forscher seine eigenen Annahmen und ggf. vorliegende statistische Werte formulieren. Dies geschieht in der Regel über spezielle Simulationsprogramme oder direkt über Programmiersprachen auf der untersten Ebene. Nach dem Simulationslauf kann der Forscher die simulierten Daten auswerten und mit den Daten, die in der Realität tatsächlich eintreten, vergleichen.

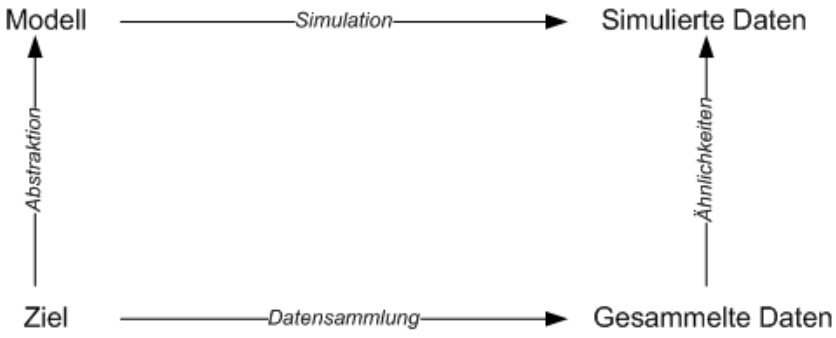
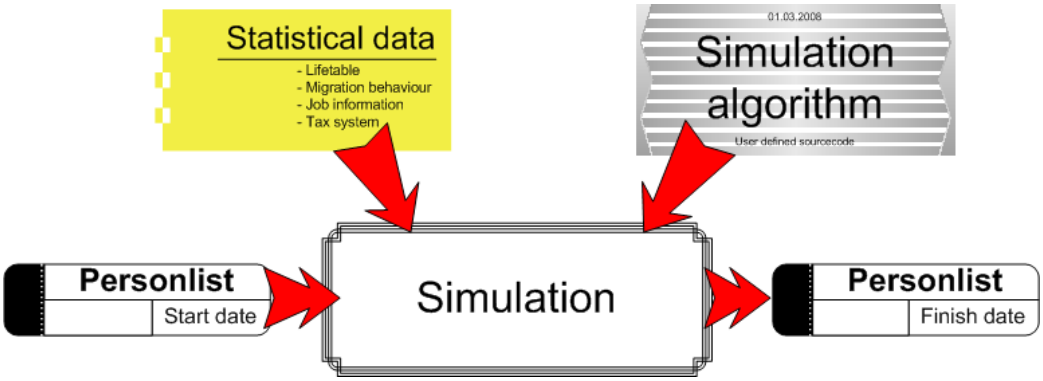


Abbildung 2.3: Logik einer Simulation als Methode

2.2 Mikrosimulation

In Simulationsansätzen wie System Dynamics werden stets Modelle gebildet, welche nicht weiter unterteilbar sind. Mit diesem Ansatz ist es sehr schwierig Phänomene zu untersuchen, welche auf Aktionen basieren, die auf einer niedrigeren Ebene stattfinden. Mikrosimulation ist der erste Ansatz um dieses Problem zu lösen. [21, S.57-65] Die Simulation selbst läuft üblicherweise auf den unteren Ebenen der Individuen und Gruppen ab. Somit lassen sich Individual- und Gruppeneffekte sowie Auswirkungen auf das Makrosystem sehr anschaulich simulieren. Ausgangspunkt für eine solche Simulation ist stets eine Datei mit Daten über die zu simulierenden Mikroobjekte, wie der Mikrozensus, das SOEP oder die EVS. Zwischen der konkreten Datenerhebung und der Veröffentlichung der Datei liegen in der Regel mehrere Jahre. Für aktuelle Simulationen ist es somit erforderlich die Datensätze ebenfalls entsprechend altern zu lassen. [14, S.5-9] Auf eine genaue Beschreibung der hierfür verwendeten Verfahren wird im Rahmen dieser Diplomarbeit verzichtet, da in CoMICS bereits aktuelle Basisdaten vorausgesetzt werden. Möglicherweise würde es in einer späteren Ausbaustufe des Programms Sinn machen, verschiedene Algorithmen und Assistenten bereitzustellen, welche die Datenanpassung erleichtern.
Im Zuge der eigentlichen Simulation wird versucht das Verhalten bzw. die Entwicklung großer Personengesellschaften vorherzubestimmen oder zu erklären. Ausgangspunkt einer solchen Simulation ist stets eine repräsentative Auswahl von Personen zu einem gewissen Startzeitpunkt. Zu jeder Person wird lediglich eine für den jeweiligen Simulationszweck geeignete Auswahl an Eigenschaften betrachtet. Die Daten werden üblicherweise in einer Tabelle modelliert. Jede Person oder jedes Objekt der Simulation repräsentiert eine Zeile in dieser Tabelle. In den einzelnen

Spalten werden die jeweils betrachteten Attribute und Eigenschaften der Personen aufgeführt. Die zugrunde liegenden Daten müssen durch Umfragen erstellt, oder bei staatlichen Stellen angefordert werden. Je nach Erstellungsdatum der Basisdaten müssen ggf. in der Realität bekannte Entwicklungen nach gepflegt werden.
Kernelement der Mikrosimulation ist der Simulationsalgorithmus. Er enthält alle Annahmen und Vorgaben die für die Entwicklung der Gesellschaft von Bedeutung sind. Innerhalb des Algorithmuses können Personendaten ausgelesen und auch geändert werden. Zusätzlich kann auf statistische Daten der Gesellschaft zugegriffen werden. Diese erhält man von Regierungsbehörden, wissenschaftlichen Einrichtungen oder auch durch eigene Annahmen.



Als Ergebnis der Mikrosimulation wird mit dem Simulationsalgorithmus die Basisgesellschaft in die Zielgesellschaft transformiert. Generell wird bei Mikrosimulationen zwischen zwei Simulationszielen unterschieden. Zum einen wird bei prädiktiven Simulationen eine gegenwärtige in eine zukünftige Gesellschaft umgeformt. Ziel hierbei sind Kurz- und Langzeitvorhersagen, die sich auf die Entwicklung der betrachteten Personengruppe beziehen. Zum anderen werden bei Deskriptiven Simulationen Basis- und Zielgesellschaft komplett in der Vergangenheit positioniert. Man versucht einen Algorithmus zu entwerfen, der die reale Entwicklung möglichst genau abbildet. Anhand des Algorithmuses kann dann versucht werden Erklärungen für reale Entwicklungsphänomene zu finden.

2.2.1 Software

Mikrosimulationen werden in der Regel mittels Programmiersprachen implementiert und auf Computern zur Ausführung gebracht. Einige Beispiele für Mikrosi-

mulationstools sind in [21, S.65] aufgeführt:

- MICSIM 3.x, for Windows 3.x/NT/95, using Visual C++ and ORACLE (Merz 1996);
- UMDBS, for Windows NT/2000/XP, with its own programming language (MISTRAL) and query language 2002 which was developed from the older Darmstadt Micro Macro Simulator (Heike et al. 1996);
- STINMOD (Lambert et al. 1994) and DYNAMOD (Antcliff 1993), for Windows (National Center for Social and Economic Modeling, see http://www.natsem.canberra.edu.au/index.html and Brown and Harding 2002);
- CORSIM (Caldwell 1993); see also http://www.strategicforecasting.com/corsim/.

CoMICS fügt dieser Softwarelandschaft ein modernes Plattformunabhängiges Werkzeug hinzu, welches durch die Kombination verschiedener Konzepte intuitiv und sehr Leistungsfähig arbeitet.

2.3 Agentenbasierte Simulationen

In diesem Kapitel werden die speziellen Techniken und Eigenschaften der Multi-Agenten-Simulation (MAS) beschrieben. Mit dem Simulationskonzept vom MAS lassen sich komplexe Verhaltensweisen von Individuen und ihre Interaktion sowohl miteinander als auch mit ihrer Umwelt abbilden. Dieses Kapitel orientiert sich an den Erläuterungen von K.G. Troitzsch und N. Gilbert [21, S.172-198]

2.3.1 Agent

In der soziotechnischen Simulation werden Agenten verwendet um die einzelnen Personen eines Gesellschaftssystems zu modellieren. Bei der Erstellung von Software-Agenten wird stets drauf geachtet, dass sie sich 'intelligent' in ihrer Zielumgebung zurechtfinden. Als Vorbild für einen Agenten dient stets die Natur des Menschen selbst. So sollen sie Merkmale wie eigene Entscheidungen, Willen und Emotionen besitzen. Das Agentendesign wurde maßgeblich durch die Forschung im Bereich Künstliche Intelligent - artificial intelligence AI - vorangetrieben. Das Teilgebiet

distributed artificial intelligence (DAI [6], [8]) beschäftigt sich ganz besonders mit den Eigenschaften und dem Design von kompletten Netzwerken bestehend aus interagierenden Agenten. Die verwendeten Agenten besitzen reichhaltige Methoden um ihr Wissen untereinander auszutauschen und zu verarbeiten. Solche Netze können dazu verwendet werden sehr komplexe Problemstellungen in einer verteilten Art und Weise zu lösen.

2.3.1.1 Eigenschaften

Software-Agenten besitzen je nach Anwendungsfall in unterschiedlichen Ausprägungsformen die folgende Eigenschaften nach Woolbridge und Jennings 1995, in Analogie zu [21]:

- **autonomy** - agents operate without others having direct control of their actions and internal state;
- **social ability** - agents interact with other agents througt some kind of 'language' (a computer language, rather than natural language);
- **reactivity** - agets are able to perceive their environmant (which may be the physical world, a virtual world of electronic networks, or a simulated world including other agents) and respond to it;
- **proactivity** - as well al reacting to theyr environment, agents are also able to take the initiative, engaging in goal-directed behaviour.

2.3.1.2 Attribute

Um ein möglichst menschliches Verhalten zu simulieren, können die Agenten mit einem gewissen Grad an Intention ausgestattet werden. Mit diesen Attributen können menschliche Bedürfnisse oder Prinzipien wie Willen, Glauben oder Emotionen softwaretechnisch modelliert werden.

- **Wissen und Glauben**
  Um Entscheidungen treffen zu können, werden Agenten einer Wissensbasis ausgestattet, in der Informationen über ihre Umwelt und über andere Agenten abgelegt werden können. Diese Informationen werden dem Agenten zum Teil initial fest mitgegeben und zum anderen - weit ausgeprägteren - Teil

zur Laufzeit gelernt. Die Informationen in der Wissensbasis müssen nicht zwangsläufig der Wahrheit entsprechen. Dies kann verwendet werden um eine verzerrte Wahrnehmung der einzelnen Agenten zu modellieren. Im Ergebnis führt dies zu fehlerhaften Folgerungen und Entscheidungen der einzelnen Agenten. Intelligente Agenten führen zwei Datenbanken. In der einen werden bestätigte Informationen und in er anderen unbestätigte Informationen abgelegt. Solche Agenten verwenden intelligente Methoden um Informationen zwischen den beiden Datenbanken zu übertragen.

- **Inferenz**
  Um Entscheidungen möglichst intelligent treffen zu können soll der Agent in der Lage sein aus einer gegebenen Menge an Annahmen weitere Informationen zu schlussfolgern. Dies kann auch dazu verwendet werden andere Informationen und Annahmen auf Korrektheit zu prüfen.

- **Soziale Modelle**
  In einigen Simulationen kann es erforderlich sein eine soziale Beziehung zwischen den einzelnen Agenten zu implementieren. Die Agenten werden mit Methoden und Eigenschaften ausgestattet um selbst solche Beziehungen auszubilden und Beziehungen anderer Agenten in ihrer Welt erkennen zu können.

- **Wissensrepräsentation**
  Um ihr Wissen über ihre Umgebung und andere Agenten abzulegen können verschiedene Ansätze, die ursprünglich in der AI Forschung entwickelt worden sind, verwendet werden.

  - die Speicherung in prädikatenlogischen deklarativen Anweisungen;
  - das Halten in semantischen baumartigen Strukturen;

- **Ziele**
  Um Agenten möglichst eigenständig und zweckbestimmt zu machen, werden sie mit eigenen Zielen und Prinzipien ausgestattet. Entscheidungen der einzelnen Agenten orientieren sich mehr oder weniger stark an den vorgegebenen Maximen. Damit in Konflikt stehende - oder gar sich ausschließende - Vorgaben beurteilen werden können, werden die einzelnen Angaben priorisiert abgelegt.

- **Planung**
  Um seine Entscheidungen so zu treffen, dass der Agent möglichst schnell seine internen Ziele erreicht, kann ihm ein Planungsalgorithmus mitgeliefert werden. Diese Art der Planung setzt beim Agenten ein Verständnis für die Existenz verschiedener Wege zur Erreichung seiner Ziele voraus. Einzelne Entscheidungsstufen werden innerhalb des Algorithmuses als Zustände modelliert. In diesen Knoten versucht der Agent seine Zielknoten möglichst schnell zu erreichen.

- **Sprache**

> Sprache ist eine ausschließlich dem Menschen eigene, nicht im Instinkt wurzelnde Methode zur Übermittlung von Gedanken, Gefühlen und Wünschen mittels eines Systems von frei geschaffenen Symbolen.
> Edward Sapir: zitiert nach John Lyons [19, S.13]

  Zentraler Aspekt der Multi-Agenten-Modelle ist die Interaktion zwischen den einzelnen Agenten untereinander. Diese kann direkt von Agent zu Agent, oder indirekt über eine veränderte Umwelt erfolgen. Das Spektrum der Kommunikationsmöglichkeiten reicht von einem einfachen Informationsaustausch bis hin zu komplexen Verhandlungskompetenzen der einzelnen Agenten. Die Kommunikation kann über festgelegte formale Sprachen oder über den direkten Austausch der Informationen erfolgen. Bei der direkten Übertragung in modernen Programmiersprachen werden Informationen direkt als Objekte übertragen, hierbei entfällt das aufwändige Parsen der Informationen.

- **Emotionen**
  Reale Personen haben Emotionen, wie Traurigkeit, Glücklichkeit, Ärger oder Liebe. Bei den bisherigen Forschungen im Bereich der Künstlichen Intelligenz werden diese Eigenschaften bis heute nicht ausgeprägt behandelt. Es bleibt bis heute ungeklärt in welcher Art und Weise emotionale Zustände oder Merkmale aus anderen bewussten oder unbewussten Zuständen hervorgehen. Der Zusammenhang in wie weit der emotionale Zustand Auswirkungen auf getroffene Entscheidungen nimmt, ist ebenfalls noch nicht wissenschaft-

lich genau geklärt. Je nach Modell lassen sich natürlich emotionale Zustände definieren, die Auswirkungen auf die vom Agenten getroffenen Entscheidungen nehmen können. In welcher Art und Weise dies genau geschieht muss der Simulationsentwickler selbst festlegen.

2.3.2 Architektur

siehe [21, S.178-180]

- **Produktionssystem**
  In Multi-Agenten-Simulationen verwenden Agenten eine Art Regelsystem, welches in der einfachsten Form durch ein Produktionssystem repräsentiert wird. Ein solches Produktionssystem setzt sich aus drei Komponenten zusammen.
  - Regelmenge
  - Arbeitsspeicher
  - Regelinterpreter

  Die Regelmenge besteht aus jeweils zwei Teilen. Im ersten Teil werden Bedingungen spezifiziert, die angeben wann genau eine Regel angewendet werden soll. Im zweiten Teil wird beschrieben, welche Aktionen durch die Regeln ausgeführt werden sollen. Die Aktionen zu der Regel werden ausgeführt, wenn die Regelbedingung erfüllt ist. Der Agent wendet die Regeln auf seine Wissensdatenbank an, die in seinem internen Arbeitsspeicher abgelegt wird. Werden Bedingungen erfüllt, führt er die in der Regel hinterlegten Aktionen aus. Dieses Verhalten wird beim Agenten durch einen Regelinterpreter erreicht, welcher in der Regel als Softwaremodul eingebunden wird. Die Reihenfolge, in der die Regeln ausgewertet werden, kann vom Agenten selbst aufgrund seiner Erfahrungen bestimmt werden. Ob zu einem speziellen Zeitpunkt alle Regeln, oder nur der erste Treffer ausgeführt werden hängt von der vom Modellierer festgelegten Verarbeitunsstrategie ab. Nach der Ausführung einer Aktion, welche immer eine Auswirkung auf die interne Wissensdatenbank des Agenten oder auf seine Umwelt hat, sollten alle Regeln erneut geprüft werden. Intelligente Agenten können konkurrierende Regeln erkennen, miteinander Vergleichen und die für das angestrebte Ziel günstigste Regel auswählen.

- **Objektorientierung**
  Programmierung von Multi-Agenten-Systemen wurde durch die Entwicklung von modernen objektorientierten Programmiersprachen deutlich vereinfacht. Jeder Agent lässt sich als Klasse abbilden, von der später beliebig viele Instanzen erzeugt werden können. Für heterogenen MAS ist es sehr einfach realisierbar für die einzelnen Agententypen verschiedene Klassen vorzusehen. Über Mechanismen der Vererbung und Interfaces können die Agenten im späteren System miteinander kommunizieren. Im späteren Abschnitt 3.1.2 wird das Thema Objektorientierung noch genauer beschrieben.

- **Simulationsumgebung**
  Die Simulationsumgebung bezeichnet den Raum, in dem sich die Agenten befinden. Dieser kann in verschiedene Bereiche aufgeteilt sein und Agenten können die Fähigkeit haben sich in dieser Umgebung frei zu bewegen. Bei solchen Agenten spricht man von mobilen Agenten. In unserem konkreten Anwendungsfall der Gesellschaftssimulation wird die räumliche Komponente allenfalls über Attribute wie Heimatadresse oder Haushaltszugehörigkeit realisiert. Natürlich ist es den Personen möglich sich durch beispielsweise einen Wechsel der Heimatadresse in gewisser Weise räumlich zu bewegen, jedoch ist dieser Aspekt in den meisten Anwendungsfällen eher zu vernachlässigen. In weiteren Ausbaustufen der Anwendung könnte es hier jedoch Sinn machen den Agenten hier mehr Mobilität zu verleihen.

2.3.3 Zusammenfassung

Bei Multiagentensystemen (MAS) handelt es sich um Systeme, die sich aus mehreren autonomen Einheiten zusammensetzen, um kollektiv eine Problemstellung zu lösen. Diese eigenständigen Einheiten werden als Agenten bezeichnet. In homogenen Systemen sind alle Einheiten gleich. Bei heterogenen Systemen hingegen, können Agenten unterschiedliche Spezialisierungen haben und sehr unterschiedliche Verhaltensweisen ausprägen. Um die Agenten lernfähig zu machen sind sie meist mit einem eigenen Speicher und einem gewissen Grundniveau an Eigenintelligenz ausgestattet. Diese Eigenintelligenz kann, je nach Anwendungsfall, auch durch eine sehr leistungsstarke Künstliche Intelligenz (AI) implementiert werden. Innerhalb des Systems können die Agenten miteinander kommunizieren, kooperieren oder auch konkurrieren. Sie sind mit Sensoren ausgestattet um ihre Umwelt wahrneh-

men zu können. Innerhalb des Systems können sich die Agenten frei bewegen und mit Systemelementen, oder anderen Agenten agieren oder reagieren. Generell müssen nicht alle aufgezählten Eigenschaften vorhanden oder speziell ausgeprägt sein. Beispiele für MAS sind Robocup oder Webcrawler.

Im Bereich der soziotechnischen, Multi-Agenten-Simulation werden in der Regel Personen als Agenten modelliert, die dann im System aufeinander wirken können. Hierbei arbeiten die Agenten nicht zusammen um ein spezielles Problem zu lösen, sondern versuchen Entwicklungen in der Gesellschaft nachzubilden. Die Anwendungsgebiete sind weit gefächert.

Evakuierungssimulationen können Schwachstellen im Evakuierungsplan von öffentlichen Gebäuden, Schiffen oder sogar ganzen Gebieten aufzeigen. Agenten modellieren in diesen System die Personen, die alle das Ziel verfolgen, den Gefahrenbereich schnell und sicher zu verlassen. Bei erweiterten Systemen werden auch Polizei, Feuerwehr und sonstige Rettungskräfte modelliert, die jeweils andere Zielsetzungen verfolgen.

Straßenverkehrssimulationen können Verkehrsverhalten in Vorfeld simulieren, um frühzeitig gezielte Stauvermeidungsstrategien entwickeln zu können. Die Agenten repräsentieren im System die einzelnen Fahrzeuge, die alle nach bestimmten Strategien versuchen ihr Fahrziel zu erreichen. Dies bietet sich ganz besonders vor großen Urlaubsphasen, aber auch im täglichen Pendelverkehr an.

2.4 CoMICS I

2.4.1 Überblick

Die Diplomarbeit basiert auf dem Simulationssystem CoMICS I. Dieses Programm ist mit dem Ziel entworfen worden das mittlerweile veraltete *UMDBS* von Prof. Dr. Thomas Sauerbier abzulösen. Hauptziel war es eine plattformunabhängige, objektorientierte und standardisierte Version zu schaffen, welche den von *UMDBS* gewohnten Funktionsumfang möglichst gerecht wird. Zunächst wurde das Programm im Rahmen eines Projektpraktikums von Prof. Troitzsch an der Universität in Koblenz konzipiert und teilweise entwickelt. Es basiert auf einer Kombination aus Mikrosimulation und Multi-Agenten-Simulation. Die Personendaten werden in einer Tabelle gehalten, wie es bereits im Abschnitt Mikrosimulation beschrieben

worden ist. Intern läuft eine Mikrosimulation durch. Um für den Simulationsentwickler etwas Komfort beim Zugriff auf die Personen bereitzustellen, werden aus den Datenzeilen der Mikrosimulationsmodells zur Simulationszeit Personenobjekte generiert, die viele Eigenschaften eines Agenten erfüllen. So können die einzelnen Personen mit andern Personen oder Elementen des Systems wechselwirken. Sie können sich innerhalb ihrer eigenen Attribute Informationen merken und über den Simulationsalgorithmus eigene Entscheidungen treffen. Da es sich in konkreten Anwendungsfällen immer um Simulationen mit sehr großen Individuenzahlen handelt, ist es nicht möglich alle Personen zeitgleich zu instantiieren. Da die Simulation nicht in Echtzeit sondern in Periodenschritten abläuft, werden immer gerade nur so viele Personenobjekte erzeugt wie benötigt werden. Dies geschieht alles verborgen im Hintergrund, der Simulationsentwickler kann einfach und bequem auf den Objekten arbeiten ohne sich um Pagingvorgänge[1] kümmern zu müssen.
Das Programm wurde in Java 1.4 implementiert und bietet eine auf SWT basierte GUI an, um bequem Simulationen erstellen, starten und später auszuwerten zu können. In einer daran anknüpfenden Studienarbeit wurde das Programm fertig gestellt und zum wissenschaftlichen Gebrauch freigegeben. Im Verlaufe dieser Diplomarbeit soll CoMICS I nun weiterentwickelt und in der Performance stark beschleunigt werden. In diesem Abschnitt soll zunächst das grundlegende Konzept von CoMICS I veranschaulicht werden.

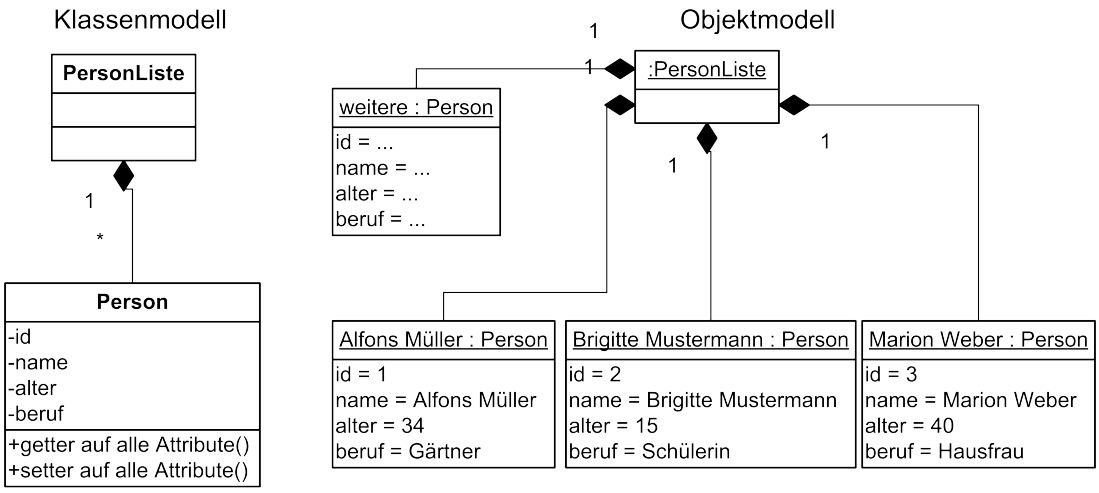
2.4.2 Die Individuen

Eine der Hauptanforderung an CoMICS I war es, die Simulation in einer objektorientierten Art und Weise zu gewährleisten. Hierzu werden die einzelnen Datensätze einer Mikrosimulationstabelle als Individuen interpretiert. Dies ist im Programm als Objektmodell umgesetzt. Über diese Objekte können die einzelnen Individuen und ihre Eigenschaften angesprochen werden. Der Zugriff für den späteren Anwender erfolgt ähnlich wie bei Agentenbasierten Simulationsmodellen.

2.4.3 Das Schichtensystem

In CoMICS I ist es möglich die einzelnen Individuen über zuvor selbst definierte Schichten zu gruppieren. Die Individuen selbst bilden die Basisschicht auf der un-

[1] Paging wird in diesem Zusammenhang für ein stückweise Lieferung von Daten verwendet, um den Hauptspeicher zu entlasten.



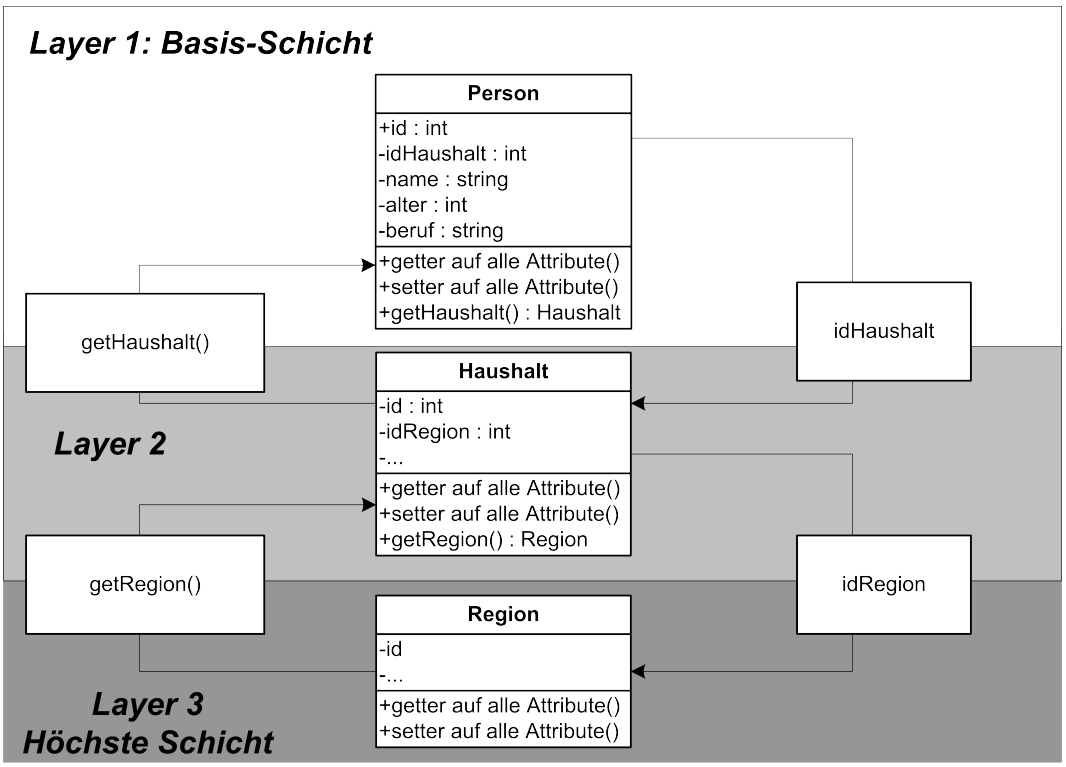
tersten Ebene. Oberhalb können beliebig viele weitere Schichten angelegt werden. Die einzelnen Objekte werden diskunkt nach oben hin gruppiert. Zwischen den einzelnen Schichten besteht eine beidseitige Verzeigerung. Jede Schicht kann also zum einen auf die nächst höhere sowie auf die nächst tiefere Schicht zugreifen. In vielen Anwendungsfällen der Sozialsimulation ergibt sich eine Baumstruktur mit Personen auf der untersten Ebene.

2.4.4 Die *Simulationsperioden*

Perioden sind die zunächst kleinsten, unteilbaren Zeiteinheiten, mit der CoMICS I arbeitet. In jeder einzelnen Periode wird auf jedes Individuum der betrachteten Gesellschaft genau einmal der Simulationsalgorithmus angewendet. Die Simulation beginnt zunächst an einer zuvor fest definierten Startperiode. Im Anschluss wird die gewünschte Anzahl an Perioden durch simuliert. Sehr oft werden in soziotechnischen Simulationen Periodenlängen von einem Jahr gewählt. Je nach Ziel der Simulation kann es Sinn machen sehr lange (Jahrzehnte, Jahrhunderte) oder sehr kurze (Monate, Tage, Sekunden) Simulationsperioden zu wählen.

2.4.5 Der *Zufallsgenerator*

Mikrosimulationen basieren auf unterster Ebene auf den Einzelpersonen und ihren individuell getroffenen Entscheidungen und Verhaltensweisen. Den Personen ein reales Entscheidungsspektrum zur Verfügung zu stellen, würde jede Simulation


Layer 1: Basis-Schicht
Person
+id : int
-idHaushalt : int
-name : string
-alter : int
-beruf : string
+getter auf alle Attribute()
+setter auf alle Attribute()
+getHaushalt() : Haushalt
getHaushalt()
idHaushalt
Haushalt
-id : int
-idRegion : int
-...
+getter auf alle Attribute()
+setter auf alle Attribute()
+getRegion() : Region
Layer 2
getRegion()
idRegion
Region
-id
-...
+getter auf alle Attribute()
+setter auf alle Attribute()
Layer 3
Höchste Schicht

stark überfordern. In CoMICS wird den Personen immer wieder eine endliche Menge an Entscheidungsmöglichkeiten zur Auswahl gestellt. Diese werden entsprechend ihrer Eintrittswahrscheinlichkeit gewichtet. Im Anschluss wird ein Zufallsgenerator verwendet, um die Entscheidung für die jeweilige Person zu treffen.

2.4.6 Die *Simulation*

Der eigentliche Simulationsalgorithmus läuft später zur Simulationszeit in einem eigenständigen Thread[2] ab. Er ist als doppelt geschachtelte Schleife implementiert, wie es in Mikrosimulationen üblich ist. Die äußere Schleife läuft durch die einzelnen Perioden, die Innere durch die Personenliste. Die Simulation beginnt bei der Startperiode und terminiert nach Ablauf der Endperiode. Natürlich steht es dem Benutzer frei, die Simulation auch schon früher abzubrechen, die Ergebnisdaten sind dann aber nur teilweise verwertbar. Innerhalb der einzelnen Perioden wird jede Person der Personenliste genau einmal durchlaufen. Bei jedem Durchlauf wird der Simulationsalgorithmus auf die gerade aktuelle Person angewandt. Der Simulationsalgorithmus basiert in der Regel auf statistischen Wahrscheinlichkeiten, den Eigenschaften der aktuellen Person und einem internen Zufallsgenerator. Somit kann für jede Person in jeder Periode entschieden werden, was genau mit ihr geschehen soll. Der strukturelle grobe Ablauf der Simulation ist in Abbildung 2.4 noch einmal im Java-Syntax abgebildet:

2.4.7 Nachteile

2.4.7.1 Performance

Generell sind Mikrosimulationen darauf ausgelegt große Datenmengen über möglichst lange Zeiträume simulieren zu können. Durch die doppelt geschachtelte Schleife ist der konzeptionelle Rechenaufwand von CoMICS I sehr hoch und der Simulationsalgorithmus läuft bei großen Datenmengen oder sehr vielen Periodenschritten verhältnismäßig lang. Hierzu ein kleines Rechenbeispiel:
Wir gehen zunächst einmal von einer Grundgesellschaft von 100 000 Personen aus. Es soll die Entwicklung dieser Gesellschaft über einen Zeitraum von 50 Jahren be-

[2]Ein Thread ist ein sequenzieller Abarbeitungslauf eines Prozesses. Ein Prozess kann aus mehreren Threads bestehen

```
public void run (){
     //Aktuelles Jahr auf Startdatum setzen
     int year = startYear;

     //Personenindex auf erste Person setzen
     int personIndex = 0;

     //Anzahl der Personen bestimmen
     int nrOfPersons = personList.size();

     //Äußere Schleife läuft durch die Perioden
     while (year < endYear){

        //Innere Schleife läuft durch die Personenliste
        while (personIndex < nrOfPersons){

            //Aktuelle Person aus der Liste holen
            currentPerson = personList.get(personIndex);

            //Aktuelle Person durchsimulieren
            doSimulation ();

            //Zeiger auf nächste Person setzen
            personIndex ++;
         }
        //Zeiger auf nächste Periode setzen
         year++;
}
```

Abbildung 2.4: Das Konzept von CoMICS I

trachtet werden. Im Bereich der Mikrosimulation sind durchaus auch weit größere Individuenzahlen sowie deutlich längere Laufzeiten denkbar. Der Erzeuger der Simulation formuliert seine Annahmen in einem Simulationsalgorithmus der 10 000 Zeilen Quellcode enthält, von dem durchschnittlich 5000 Zeilen pro Person durchlaufen werden. Zusammenfassend ergibt sich folgende Berechnung der Anzahl an Quellcodezeilen, die während der Simulation durchlaufen werden müssen.

- G = 100 000 Individuen
- T = 50 Perioden
- C = 5000 Zeilen Simulationsquellcode

$$G \times T \times C = 100\,000 \times 50 \times 5000 = 25\,000\,000\,000$$

Wir sind bei der Rechnung davon ausgegangen, dass die Anzahl der Individuen der Grundgesellschaft über die Perioden hinweg im Mittel gleich verbleibt. Wachsende Personenzahlen verlangsamen, schrumpfende beschleunigen die Simulation zusätzlich.
Je nach Performance des Rechners, auf dem die Simulation ausgeführt wird, läuft die Simulation Stunden oder sogar Tage. Diese Laufzeit schränkt natürlich die Anwendungsgebiete von CoMICS I etwas ein. Um einen performanten Ablauf der Simulation zu gewährleisten sollten nicht zu viele Personen über einen nicht zu langen Zeitraum mit einem möglichst kurzen Simulationsalgorithmus auf einem möglichst schnellen Rechner ausgeführt werden.
Hauptproblem des beschriebenen Konzeptes ist, dass in jedem Jahr für jede Person und für jeweils jedes einzelne Simulationselement entschieden werden muss, ob es auf die aktuelle Person angewendet werden soll oder nicht. Es wird beispielsweise in jedem Jahr für jede Person berechnet, ob eben diese Person stirbt oder nicht. Diese ggf. auch sehr aufwändige Berechnung führt fast immer zu dem Ergebnis, dass die aktuelle Person weiterleben darf. Trotzdem muss die Berechnung nach diesem Konzept immer und immer wieder durchgeführt werden.

2.4.7.2 Debug Funktionen

Die Hauptaufgabe des Schöpfers einer Simulation besteht darin, den Simulationsalgorithmus zu entwerfen. Hierzu stellt CoMICS I einen relativ simplen Texteditor

zur Verfügung, der in seiner Funktionalität bei weitem nicht an gewohnte Entwicklungsumgebungen heran reicht. Somit ist es für den Ersteller zum Teil sehr schwierig und aufwendig, Fehler in der Simulation zu entdecken und zu beseitigen.

2.4.7.3 Datenbank

Daten werden in CoMICS I nur intern in einer HSQL Datenbank gehalten. Export und Import sind ausschließlich über CSV-Dateien möglich. Da HSQL ab einer bestimmten Anzahl an Datenbankeinträgen deutlich langsamer wird, wurde entschieden die Personendaten direkt in den Tabellen abzuändern und eine Historisierung ausschließlich über ein alljährliches herausschreiben als CSV-Datei bereitzustellen. Diese Variante ist zwar Funktional korrekt, schlägt sich allerdings in einer deutlich längeren Laufzeit nieder. Nachträglich auf einer anderen Datenbank als HSQL zu arbeiten oder datenbankinterne Constraints zu verwenden war leider nicht Vorgesehen.

2.4.7.4 Betriebssystem

Das Programm benötigt zwingend ein Windows basiertes Betriebssystem mit einer installierten Version der Eclipse IDE. Um es auf Linux, MAC oder Solaris eingeschränkt lauffähig zu bekommen bedarf es einem enormem Aufwand. Zudem fehlt ein Installationsprogramm, welches CoMICS I sauber auf einem Rechner installiert bzw deinstalliert. Es muss von Hand eine ganz spezielle Ordnerstruktur erstellt werden, damit das Programm überhaupt lauffähig ist.

2.5 Diskrete Event-Simulation

In der diskreten Event Simulation werden alle Operationen in dem betrachteten System als chronologische Sequenz von Events angesehen. Jedes Event findet an einem festgelegten diskreten Zeitpunkt statt und ändert meist den Status des Systems. Alle diskreten Event Systeme beinhalten nach George S. Fishman [13] folgende sieben Konzepte.

- Work
- Resources
- Routing

- Buffers
- Scheduling
- Sequencing
- Performance

> *Work* denotes the items, jobs, customers, etc. that enter the System seeking service. ... *Resources* include equipment, conveyances, and manpower that can provide the services. ... Associated with each unit or batch of work is a *route* delineating the collection of required services, the resources to provide them, and the order in which the services are to be performed. ... *Buffers* are waiting rooms that hold work awaiting service. They may have infinite capacity ... or may have finite capacity ... When buffers have finite capacity, explicit rules must be adapted that account for what happens to arriving work that finds a buffer full. *Scheduling* denotes the pattern of availability of resources. ... *Sequencing* denotes the order in which resources provide services to their waiting work. It may be in first-come-first-served order; it may be dictated by the amount of work awaiting the resource or other resources from which it receives or transmits work. Sometimes the rule for sequencing is called the queueing discipline.

[13, S.7-8]

In CoMICS soll die eventorientierte Simulation dazu verwendet werden das Laufzeitverhalten des Mikrosimulationsmoduls deutlich zu verbessern. Während der Zugriff auf die einzelnen Personen wie in Multi-Agenten-Systemen erfolgt, sollen sämtliche Aktionen, die eine Statusänderung am System auslösen können, als diskrete Events modelliert werden. Der Status der Simulation zum Zeitpunkt Φ wird durch die Datenbasis der Mikrosimulation bestimmt. In ihr enthalten sind alle Personen, Haushalte, Regionen ... usw. mit allen Attributsbelegungen zum Zeitpunkt Φ. Die zentralen Elemente im konkreten Anwendungsfall zur Optimierung von Mikrosimulationen sind:

- Einheitliches Zeitsystem
- Event Liste

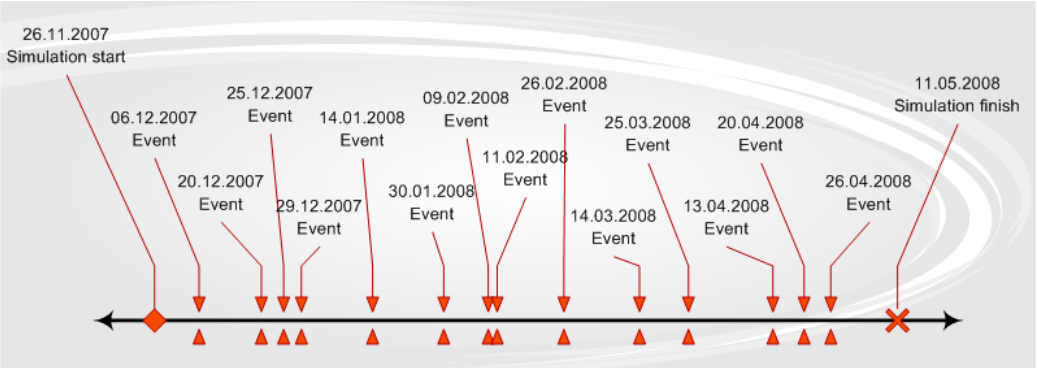
- Personen (Agenten)
- Zufallsgenerator
- Abbruchbedingung

Alle Elemente der Eventsimulation müssen auf der gleichen synchronen Zeitbasis arbeiten, da sonst die Ausführungsreihenfolge der einzelnen Aktionen nicht vorhersehbar sein könnte. Das Zeitsystem ist Basis für die oben aufgeführten Konzepte Scheduling, Sequencing und Routing. Um aktive Zustandsänderungen am System herbeizuführen werden Events verwendet. In diesen Events müssen alle dynamischen Aktionen und Operationen bereitgestellt werden, die im Simulationsmodell verwendet werden sollen. Die Events bilden die Konzepte Work, Routing, Resources und Performance ab. Sie besitzen einen diskreten Ausführungszeitpunkt und werden chronologisch in einer EventListe angeordnet. Diese Liste spiegelt das Buffer Konzept wieder und wird in der neuen Version von CoMICS durch den Eventkalender repräsentiert, welcher im nächsten Kapitel noch ausführlicher behandelt wird. Den einzelnen Events werden Personen zugeordnet, über die eine Änderung an der Datenbasis und somit auch eine Zustandsänderung im System möglich wird. Simulationen benötigen immer ein gewisses Maß an Zufälligkeit, welches natürlich immer vom jeweiligen Simulationszweck abhängt. Im konkreten Anwendungsfall der Sozialsimulation ist es sehr oft notwendig Entscheidungen für die Personen innerhalb des Systems zu treffen. Hierfür ist eine Kombination aus Zufallsgenerator und statistischen Wahrscheinlichkeiten unabdingbar. Theoretisch könnten Simulationen ewig laufen, was jedoch in den meisten Anwendungsfällen wenig Sinn machen würde. Deshalb ist es für jede EventSimulation entscheidend, bei welcher exakten Bedingung die Simulation terminiert. Besonders Wichtig hierbei ist die Berücksichtigung von möglichen Fehlerfällen und der saubere Abbruch der Simulation.

2.5.1 Der Kalender

Zentrales Element bei diesem Konzept ist ein Kalender der die einzelnen Events aufnimmt und später dann in chronologischer Reihenfolge abarbeitet. Zu Beginn sind im Kalender der Start- und Endzeitpunkt der Simulation fest eingetragen. In der Regel werden diese auch während der Simulation nicht dynamisch geändert sondern können als konstant angenommen werden. Die Simulation beginnt beim

Start- und läuft einmal bis zum Endzeitpunkt durch. Geordnet nach ihrem Ausführungszeitpunkt werden die einzelnen Events dann abgearbeitet. Bei Erreichen des Endzeitpunktes terminiert die Simulation.



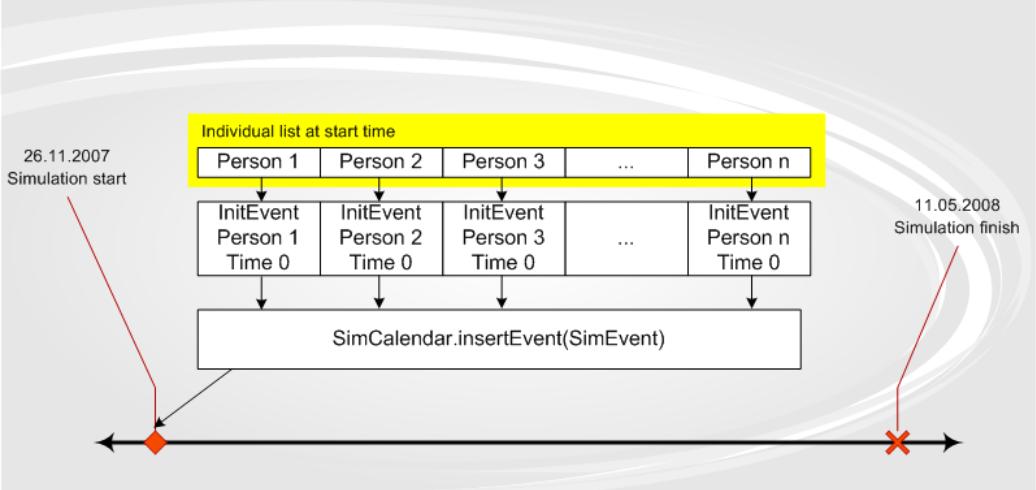
2.5.2 Die Events

Die Events sind die eigentlichen Träger des Simulationsalgorithmuses. Jedes Event besitzt einen speziellen Ausführungszeitpunkt und ggf. noch eine oder mehrere Personen, die mit diesem konkreten Event verknüpft sind. Innerhalb des Events werden die verknüpften Personen bestimmten Rollen zugeordnet. Bei einem Hochzeitsevent sind z.B. die Rollen Braut und Bräutigam mit konkreten Personen aus der Basisgesellschaft verbunden. Jedes Event ist von einem Eventtyp abgeleitet. Diese müssen vor dem eigentlichen Simulationslauf bereits komplett vorliegen und können in der Simulation als Vorlage für beliebig viele Events dienen. Häufig auftretende Eventtypen sind z.B. Geburt, Hochzeit und Tod.
Jedes Event verwendet eine Execute-Funktion um Änderungen an der Datenbasis vorzunehmen. Innerhalb dieser Funktion kann auf die verknüpften Personen über den Rollennamen zugegriffen werden. Die Personendaten können gelesen und geschrieben werden. Möglichkeiten zum Erstellen und Löschen von Personen ist ebenfalls gegeben. Hiermit lassen sich beispielsweise Geburt, Tod und Migrationsverhalten modellieren. Um innerhalb der Events Entscheidungen zu treffen steht ein Zufallsgenerator und diverse statistische Objekte zur Verfügung, die zu Beginn der Simulation eingelesen werden können. Des weiteren kann ein Event beliebig viele neue Events erzeugen und im Kalender ablegen. Beispielsweise könnte ein

Geburt-Event sofort das Einschulungs-Event berechnen, erstellen und in den Kalender eintragen.

2.5.3 Der Initialisierungsprozess

Wird die Simulation gestartet befinden sich im Kalender zunächst lediglich der Start- und Endzeitpunkt der Simulation. Zu jeder Person der Basisgesellschaft wird ein *InitEvent* erzeugt, welches in den Kalender mit Ausführungszeitpunkt 0 eingetragen wird. Diese *InitEvents* werden vor dem eigentlichen Simulationslauf ausgeführt und legen alle weiteren Events an, die für die Simulation erforderlich sind. Die *InitEvents* sind Ausgangspunkt der Simulation und sollten mit sehr viel Sorgfalt entwickelt werden. Sind alle *InitEvents* abgearbeitet läuft die Simulation ganz normal durch. Die folgenden Events, die auch wieder weitere Events anlegen können, werden abgearbeitet bis der Endzeitpunkt erreicht ist.



2.6 Zusammenfassung

Dieses Kapitel führt in die Thematik der soziotechnischen Simulation ein. Die Grundbegriffe Modellierung und Simulation werden hier eingeführt und voneinander abgegrenzt. Die Simulationsverfahren Mikrosimulation und Agentenorientierte Simulation wurden ausführlich beschrieben, da CoMICS auf diesen beiden Ansätzen zur Simulation aufbaut. Die erste Version von CoMICS zeichnet sich durch

die objektorientierte Individuendarstellung, das Schichtensystem und eine im Hintergrund ablaufende Mikrosimulation aus. Der Entwickler arbeitet ausschließlich mit den Personenobjekten direkt, was dem Agentenorientierten Konzept sehr nahe kommt. Erst durch die Umsetzung in eine Mikrosimulation ist es überhaupt möglich große Datenmengen über längere Zeiträume hinweg simulieren zu können. Hauptnachteil dieses Konzeptes ist der Simulationsalgorithmus mit seinen doppelt verschachtelten Schleifen. Der Rechenaufwand für eine solche Simulation ist bereits von Grund auf sehr Aufwendig.

Kapitel 3

Technische Umsetzung

3.1 Java Technik

Auf der Suche nach einer Neuorientierung, begann SUN 1991 ein geheimes Projekt. Das später als *Green-Project* bekanntes Vorhaben beschäftigte sich mit einem Prototyp zur Steuerung und Integration von Geräten. Im Herbst 1992 wurde der Prototyp firmenintern unter dem Namen **7 (Star Seven)* präsentiert und löste soviel Begeisterung im Management von SUN aus, dass die Firma *First Person, Inc.*, mit dem Ziel den Prototyp zur Serienreife zubringen und dieses Gerät zu vermarkten, gegründet wurde. Jedoch scheiterte die Vermarktung, sodass im April 1994 *First Person, Inc.* aufgelöst wurde.[18, S.33-34] In diesem Moment wäre die Geschichte von Java schon fast beendet gewesen, wenn nicht besonders Bill Joy [1] in der Plattformunabhängigkeit, der für **7* entwickelten Programmiersprache großes Potential für das *World Wide Web* sah. So erschien im Herbst 1994 der *WebRunner*, ein Web-Browser, der neben HTML-Seiten auch sogenannte *Applets* aus dem Internet laden und ausführen konnte. Diese *Applets* wurden in *Oak* geschrieben, einer objektorientierten Programmiersprache, die später in Java umbenannt und erstmalig am 23. März 1995 unter diesem Namen der öffentlichkeit vorgestellt wurde.[18, S.35]

Bereits vor der offiziellen Erscheinung des *JDK*[2] Version 1.0 Anfang 1996, wurden schon hunderter frei verfügbare *Applets* angeboten. Ein weiterer Meilenstein wurde das *JDK* 1.2, das im Frühjahr 1998 erschien und sehr bald in *Java 2 Platt-*

[1] Bill Joy ist Mitbegründer von SUN
[2] Java Development Kit

form umbenannt wurde. Jedoch war diese Version noch recht langsam und insbesondere das *Swing-Toolkit* trug zu dem Ruf bei, Java sei zu langsam für "echte" Anwendungen.[18, S.37] Diesen Umstand beseitigten die nachfolgenden Versionen z.B. durch die Einführung eines *Just-In-Time-Compiler*.

Mittlerweile hat Java viele Bereiche unseres Lebens erreicht. Sei es die Spiele auf einem Mobiltelefon, die *Applets* auf vielen Internetseiten oder gar Applikationsserver für die Integration von unterschiedlichen Programmen. Wegen dieser Vielgestaltigkeit wird das SDK, neben den verschiedenen Plattformen, in drei unterschiedlichen Auslieferungen unterteilt:

- J2SE (*Java 2 Standard Edition*), für IBM kompatible Computer und für diese Arbeit ausschlaggebend.
- J2ME (*Java 2 Micro Edition*), für eingeschränkte Geräte wie Mobiltelefone, PDAs oder andere eingebettete Systeme.
- J2EE (*Java 2 Enterprise Edition*), für die Entwicklung von verteilten Business-Applikationen.

Zudem verfügt Java über einige Eigenschaften, die für diese Arbeit besonders interessant sind, die im Nachfolgenden näher begutachtet werden.

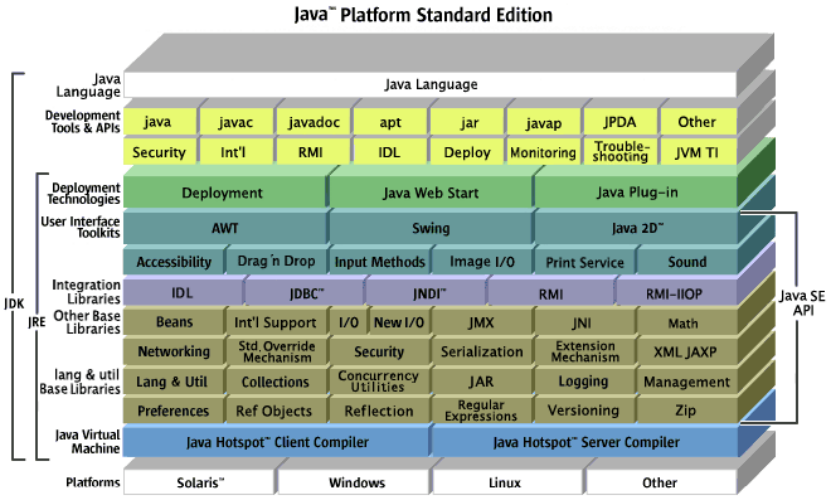


Abbildung 3.1: Modulare Architektur von Java[26, Chapter 1]

3.1.1 Plattformunabhängigkeit

Ein Erfolgsfaktor von Java ist die Plattformunabhängigkeit. Diese Eigenschaft macht es möglich, ein und das selbe Programm auf unterschiedlichsten Maschinen auszuführen, ohne den Quelltext[3] neu zu kompilieren. Gewährleistet wird dies indem der Quelltext nicht in Maschinencode[4] sondern in einem, von der Zielplattform unabhängigen, Zwischencode übersetzt wird. Dieser so genannten *Bytecode* kann über eine Laufzeitumgebung (RE) [5] ausgeführt werden. Die Java Laufzeitumgebung (JRE) besteht aus der Virtuellen Maschine (VM) und aus einer Java API. Beide Bestandteile sind auf die jeweilige Plattform angepasst. Die Virtuelle Maschine führt ein Java-Programm, unter Verwendung der API aus, indem der *Bytecode* interpretiert wird. Hierbei übernimmt die VM die Speicherverwaltung, u.a. mit der Hilfe der Garbage Collector (GC), der für die Speicherbereinigung sorgt.

Um eine plattformübergreifende Portabilität gewährleisten zu können, gibt es für die meisten Betriebssysteme die passenden JRE's. Ein weiter Möglichkeit ist die Verwendung s.g. Java-Prozessoren, die den *Bytecode* direkt ausführen können. Java bietet zwei plattformunabhängige Bibliotheken an, die eine grafische Interaktion mit dem Benutzer ermöglichen. Das *Abstract Widget Toolkit (AWT)* setzt auf das jeweiligen Betriebssystems auf und bildet den gemeinsamen Nenner der verschiedenen Systeme. Daher stehen mit dem *AWT* nur elementare Grafik- und Fensterfunktionen zur Verfügung. Weiter gehende Funktionalität bietet *Swing*, mit dessen Hilfe sehr komplexe grafische Oberflächen erstellt werden können. Des weiteren kann durch das *Pluggable Look-and-Feel* das Erscheinungsbild der Anwendung während der Laufzeit verändert werden.[18, S.43] Leider trug gerade *Swing* zu dem Ruf bei, dass Java sehr inperformant sei.[18, S.37] Daher wurde bei der Entwicklung des JDK 1.3 besonderes Augenmerk auf die Performance gelegt. War das JDK 1.2 im Vergleich zu nativen Anwendungen[6] noch langsam und träge hat sich dieser Zustand u.a. durch die Einführung des *HotSpot*-Kompilers mit dem JDK 1.3 deutlich verbessert. Dieser Kompiler analysiert zu Laufzeit den *Bytecode* und übersetzt häufig durchlaufende Stellen in Machinencode. Hierdurch konnte eine weitere Annäherung an die Performance einer C oder C++ Applikation geschaffen werden.[18, S.37,S.48-49]

[3] Vom Menschen lesbare Ansammlung von Instruktionen an eine Maschine
[4] Ansammlung von Instruktionen die von einer Maschine ausgeführt werden können.
[5] engl. runtime environment
[6] In Maschinecode vorliegende Applikation

3.1.2 Objektorientierung

In den 90er Jahren befand sich die Softwaretechnik in einer Krise. Immer leistungsfähigere Computer ermöglichten größere und komplexere Anwendungen, die auch zunehmend Fehleranfälliger wurden. Dies führte zu einem Paradigmenwechsel, von dem Denken in Vorgängen und Funktionen hin zu dem Modellieren von Dingen bzw. Objekten und deren Beziehungen. Ein Objekt kapselt Daten und ist in der Objektorientierung das zentrale Bauteil. Ian Sommerville definiert ein Objekt wie folgt:

> Ein Objekt ist eine Entität. Es besitzt einen Zustand sowie eine definierte Menge von Handlungsmöglichkeiten (Operationen), die auf diesem Zustand aufsetzen. Der Zustand wird als Menge von Objektattributen dargestellt. Die Operationen dieses Objekts bieten anderen Objekten ihre Dienste an, um z.B. Berechnungen auszuführen.
>
> Objekte werden gemäß der Definition einer Objektklasse erzeugt. Eine solche Klassendefinition dient als Vorlage für die Erstellung von Objekten und beinhaltet Deklarationen aller Attribute und Operationen, die einem Objekt dieser Klasse zugeordnet werden sollten. [25, S.271]

In Java sind Methoden eines Objekts die in der Definition genannten Operationen und Attribute werden durch so genannte Instanzvariablen abgebildet. Eine Objektklasse werden wir im folgenden Java-typisch Klasse nennen.
Ein wichtiges Element der objektorientierten Programmierung ist die Abstraktion. Sie manifestiert sich durch die Unterscheidung von Bauplan und Bauteil. Eine Klasse gibt an wie ein Objekt erzeugt wird, wie es zu verwenden ist und wie es sich verhalten soll. Das Objekt selbst erfüllt dann die jeweils gestellten Aufgaben.[18, S.151]

Weitere Grundkonzepte der Objektorientierung sind:

- Datenkapselung
- Vererbungsmechanismen
- Polymorphie

[4, S.12]
Wie schon erwähnt, ist ein zentrales Konzept der objektorientierten Programmierung die Datenkapselung. Wie aus der Definition 3.1.2 bekannt, wird eine Klasse

durch die Zusammenfassung einer Menge von Objektattributen und Operationen definiert. Eine Menge von Objektattributen wird in Java durch einen Satz von Instanzvariablen repräsentiert und für jedes instanziierte Objekt der Klasse neu belegt. Im Gegensatz dazu sind die Operationen bzw. Methoden einer Klasse im Programmcode nur einmal vorhanden und operieren auf den jeweiligen Daten des Objektes. Diese Datenkapselung ist in Abbildung 3.2 illustriert, die zwei Instanzen[7] einer Klasse zeigt.

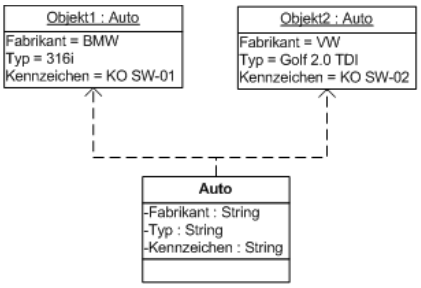


Abbildung 3.2: UML: Zwei Instanzen einer Klasse Auto

Die Abstraktion und die Datenkapselung erhöhen die Wiederverwendbarkeit, indem Eigenschaften, die oft verwendet werden, in Klassen und somit Objekten zur Verfügung gestellt werden. Da es jedoch unterschiedliche Abstufungen zwischen Objekten gibt, zum Beispiel ist ein Fahrrad und ein Motorrad beides ein Zweirad, wird die Wiederverwendung auch durch die Vererbung gefördert. Hierzu werden alle öffentlichen Methoden und Instanzvariablen einer Klasse, den Klassen die davon abgeleitet werden zur Verfügung gestellt, sozusagen vererbt. Zum Beispiel gibt es eine Klasse Person, die zwei Attribute Name und Vorname besitzt. Die Klasse Student erbt von Person und fügt die Variable Martikelnummer bei, gezeigt in Abbildung 3.3.

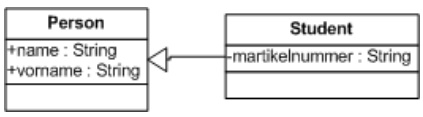


Abbildung 3.3: UML: Stundent erbt von Person

Die Vererbung ermöglicht ein weiteres wichtiges Grundkonzept von Java, den Polymorphismus. Polymorphismus ist in etwa äquivalent mit dem deutschen Begriff

[7]Instanz und Objekt werden oft als Synonym verwendet, wir schließen uns diesem Wortgebrauch an. [18, S.152]

'Vielgestaltigkeit'. Dahinter verbirgt sich, dass jede konkrete Klasse mit der Klasse identifizierbar werden kann, von der sie spezialisiert wurde. In dem Beispiel 3.3 können alle Klassen durch Person identifiziert werden, die von dieser abgeleitet sind. Hier ist Student abgeleitet von Person, jedoch wären auch Hausfrau oder Bauarbeiter mögliche Unterklassen.[18, S.151-157]

3.1.3 Java 5.0

Um den Entwicklungsstand von JRE 1.5.0 besser hervorzuheben und den Unterschied zu der Vorgängerversion klar herauszustellen, wurde eine neue externe Versionsnummer eingeführt. So heißt das JRE 1.5.0 nun JRE 5.0.

Neben der Versionsänderung wurde eine Vielzahl von neuen Konzepten eingeführt, die den Umgang mit Java 5.0 stark erleichtert.

3.1.3.1 Verwendung von *Generics*

Mit der Einführung von *Generics* orientierte sich Java an die von C++ bekannten Templates. Ab dem JDK 5.0 ist es möglich, in der Klassendefinition so genannte Wildcards zu integrieren, die bei der Instanziierung eines entsprechenden Objekts mit Typ-Informationen angereichert werden. Eine Motivation für die Einführung dieses Mechanismus, war für die Java-Entwickler die fehlende Typsicherheit der *Collections*[8]. Am deutlichsten werden die Folgen der Einführung von Generics an einem kleinen Quellcode-Beispiel ersichtlich. Eine Quellcode geschrieben in Java 1.4 könnte wie folgt ausgesehen haben:
Hierbei besteht das Problem in Zeile 3. Da eine Liste für Java nur eine sequenzielle

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.get(0);
```

Abbildung 3.4: Liste ohne die Verwendung von Generics

Anordnung von Elemente vom Typ *Object* ist, muss der Rückgabewert der Methode *get()* durch einen *Cast* als *Integer* identifiziert werden. Da die Liste jedoch alle Objekte aufnehmen kann, könnte es an dieser Stelle zu einem Fehler kommen. Dieser würde sich durch eine *ClassCastException* bemerkbar machen.

[8]Das Interface *java.util.Collection* wird von Datenstrukturen wie zum Beispiel Queues, Listen oder Sets implementiert

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.get(0);
```

Abbildung 3.5: Liste unter Verwendung von Generics

Der äquivalente Quellcode in Java 5.0 könnte so aussehen:
Die in der 1. Zeile instanziierte Liste ist nun, im Gegensatz zu dem vorherigen Beispiel, eine sequenzielle Anordnung von Elementen des Typs *Integer*. Sie beinhaltet ausschließlich Objekte dieses Typs, daher ist ein 'unsicherer *Cast*' an dieser Stelle nicht mehr nötig.
Dieses Konzept ist jedoch nicht nur auf *Collection*s beschränkt, sondern kann auch für eigene Klassen eingesetzt werden. Hierzu müssen Wildcards in der Methoden- bzw. Klassen-Spezifikation mit einbezogen werden.[7, S.2-3]

3.1.3.2 Einführung der *For-Each* Schleife

Die Java-**Generics** ermöglichen den Einsatz einer 'Für Alle'-Schleife. Diese Schleife iteriert über jedes Element in einer *Collection*. So könnte die Ausgabe aller Elemente einer Liste in die Konsole in Java 1.4 wie folgt aussehen:
Dieser Art der Implementation ist umständlich und kommt jedoch im Zuge einer

```
List<Object> myList = new LinkedList<Object>();
for (int i = 0, n = myList.size(); i<n; i++)
    System.out.println(myList.get(i));
```

Abbildung 3.6: Iteration mit Hilfe einer For-Schleife

Entwicklung sehr oft vor. Durch die Verwendung einer *For-Each*-Schleife in Java 5.0 kann der Quellcode deutlich vereinfacht werden:[27]

```
List<Object> myList = new LinkedList<Object>();
for (Object o: myList)
    System.out.println(o);
```

Abbildung 3.7: Iteration mit einer For-Each-Schleife

3.1.3.3 Automatisches Umwandeln von primitiven Datentypen

Ein weiterer lästiger Punkt in einer Java-Entwicklung, ist das Umwandeln von primitiven Datentypen in ein Objekt der dazugehörigen Wrapperklasse[9]. Um diesen Umstand zu verbessern wurde das 'Autoboxing' eingeführt. Hierdurch werden primitive Datentypen automatisch und komplett transparent für den Entwickler in ein Objekt umgewandelt, falls dieses im entsprechenden Kontext verlangt wird. Zum Beispiel:

In Abbildung 3.8 entfällt das 'new Integer(0)', um den Zahlenwert '0' zur Liste

```
List<Integer> myList = new LinkedList<Integer>();
myList.add(0);
int datatype = myList.get(0);
Integer wrapper = myList.get(0);
```

Abbildung 3.8: Demonstration von Autoboxing

hinzuzufügen. Dementsprechend ist der Umgang mit primitiven Datentypen und deren Wrapper-Objekten sehr erleichtert wurden. Jedoch sollte dieser Mechanismus nicht all zu bedenkenlos eingesetzt werden, da das Erzeugen eines Objektes bzw. das Umwandeln in einen Datentyp zu Einbußen der Performance führt.[27]

3.1.3.4 Bereitstellung von *Enums*

In der Praxis hat man es sehr oft mit Werten zutun, die einem gewissen Vorrat an vordefinierten Werten entnommen werden. Ein Beispiel wären die vier Jahreszeiten, die als Konstanten definiert so aussehen könnten: Dies Art der Erstellung von

```
// int Enum-Pattern - verurscht mehrere Probleme
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL = 3;
```

Abbildung 3.9: Konstantenbildung nach dem int-enum-Pattern

Aufzählungen sorgt jedoch für einige Probleme:

- **Fehlende Typsicherheit** - Es kann vom Kompiler nicht kontrolliert werden, dass ausschließlich die Werte aus dem Wertevorrat verwendet werden.

[9] Eine Wrapperklasse ist eine Klasse, die es ermöglicht, einen primitiven Datentyp durch ein Objekt abzubilden.

- Es gibt **keinen Namensraum** - Um einzelne Fälle unterscheiden zu können, muss der Bezeichner der Konstanten mit einen Präfix versehen werden, wie z.B. 'SEASON_'

- Die **Konstanten sind 'brüchig'** - Werden die Werte geändert oder zwischen drin Konstanten hinzu geführt, muss auch jede Programmstelle, die auf diesen Werten arbeitet angepasst bzw. kontrolliert werden.

- **Ausgabetext ist nicht informativ**, da nur der Wert hinter der Konstanten ausgegeben wird.

In Programmiersprachen wie z.B. C++, gibt es für solche Fälle *enum*s. *Enum*s sind primitive Datentypen, die einen von evtl. mehreren vordefinierten Werten annehmen können. Dieses Konzept wurde in Java 5.0 übernommen und erweitert. Das Beispiel der 4 Jahreszeiten könnte mit *enum*s so aussehen:
Hierbei wirkt ein *enum* wie eine Vereinbarung über mögliche Daten und sorgt

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

Abbildung 3.10: Erzeugung eines *enum*

somit für die erwünschte Typsicherheit. Eine weiter Möglichkeit in Java 5.0 ist es, eigene Aufzählungstypen zu definierten. Diese *Enum*s werden hierbei ähnlich definiert wie Klassen und besitzen alle Eigenschaften einer solchen. So beinhaltet ein so erzeugter *Enum* alle Methoden der Klasse *Object*, zuzüglich eigen definierte Methoden und Variablen.[27]

3.1.3.5 Variable Parameterlisten

Methoden haben in Java 1.4 immer eine definierte Menge an Parametern. Um nun eine undefinierte Menge von Werten einer Methode übergeben zu können, muss entweder ein Array des entsprechenden Typs oder eine *Collection* übergeben werden. In Java 5.0 wurden daher die variablen Parameterlisten eingeführt. Hierbei kann am Ende einer Parameterliste ein variabler Teil definiert werden, der innerhalb der Methode als entsprechendes Array verwendet werden kann.[27] Die Methode aus Abbildung 3.11 kann nun mit einer beliebigen Anzahl von Parametern vom Typ *String* aufgerufen werden. (*demonstration(„param1“,“param2“,....“paramN“);*)

```
public void demonstration(String...parameterList){
    for (String parameter: parameterList)
        System.out.println(parameter);
}
```

Abbildung 3.11: Variable Parameter-Liste

3.1.3.6 Statische Imports

Statische Imports ermöglichen einen Unqualifizierten Zugriff auf öffentliche Klassenvariablen und -methoden. Ein Zugriff auf die Klasse *java.util.Math* sieht ohne statischen Import wie folgt aus:

```
double result = Math.cos(Math.PI, theta);
```

Mit einem solchen Import kann die Methode *cos()*, sowie die Konstante *PI* wie ein klassen-eigenes Element verwendet werden. Das obere Beispiel sieht mit statischem Import wir folgt aus:

```
import static java.lang.Math.*;
double result = cos(PI, theta);
```

[27]

3.1.3.7 Annotationen

In viele Szenarien ist es nötig, Meta-Informationen über Elemente eines Systems zu erhalten. Ein Beispiel hierfür sind *JavaBean*s, die eine *BeanInfo*-Datei bzw. bei *Enterprise JavaBean*s (EJB) ein *deployment description* verwenden, um eine Interaktion mit anderen *JavaBean*s zu ermöglichen.
Ein weiteres Beispiel von Meta-Informationen in Java ist das Javadoc *@deprecated*, womit eine Klasse bzw. Methode als nicht mehr zu benutzen gekennzeichnet wird. Java 5.0 stellt nun einen generellen Mechanismus zur Angabe von Meta-Daten bereit. Hierzu können eigene *Annotationen* erzeugt werden. Dies erfolgt ähnlich wie die Erstellung eines Interface, jedoch mit dem Schüsselwort *'@interface'*. Diese *Annotation* kann anschließend vor einem Java-Element (z.B. einer Klasse, Methode oder Variablen) eingefügt und mit Daten gefüllt werden.
Diese zusätzlichen Informationen greifen jedoch nicht aktiv in die Semantik eines Programms ein, sondern stellen ausschließlich Informationen bereit, die über die Quell-Datei, über die Klassen-Datei oder über Reflektion bezogen werden können.

```
public @interface Copyright {
    String value();
}
@Copyright("2008 Universitaet Koblenz-Landau")
public class Comics {

}
```

Abbildung 3.12: Erstellung und Verwendung einer Annotation

Ein Beispiel für die ausgiebige Verwendung von Annotationen bietet Hibernate, das wir im folgenden Kapitel ausgiebig behandeln werden.[27]

3.2 Grafische Darstellung

Eine Grundidee von Java war es, eine einfache Möglichkeit zu bieten, grafische Oberflächen zu entwickeln. Daher stellt Java seit der ersten Version das *Abstract Widget Toolkit* (*AWT*) zur Verfügung. Diese Grafikbibliothek beinhaltet Primitivoperationen zum Zeichnen von Linien und Texten, einen Event-Mechanismus[10] zur Steuerung des Programmablaufs, Dialogelemente für die Interaktion mit dem Benutzer und Fortgeschrittene Grafikfunktionen zur Darstellung und Manipulation von Bitmaps, sowie der Ausgabe von Sound.[18, S.531-533] Dabei setzt *AWT* auf die jeweiligen Funktionen des Betriebssystems auf und ist daher in seinem Umfang beschränkt, was die Erstellung von komplexer Oberflächen erschwert.

Eine weiter Möglichkeit der Entwicklung von grafischen Oberflächen bietet *Swing*, das seit dem JDK 1.2 fester Bestandteil von Java ist. Diese Bibliothek erweitert das *AWT* und verwendet nur noch eingeschränkt plattformspezifische GUI-Ressourcen. Abgesehen von Hauptfenster, Dialogen und grafischen Primitivoperationen erzeugt *Swing* alle GUI-Elemente selbst. So wird z.B. ein *Swing*-Button nicht mehr vom Windows-UI-Manager dargestellt, sondern eigens von *Swing* gezeichnet. Dies sorgt für ein höheres Maß an Plattformunabhängigkeit und Portabilität im Vergleich zu *AWT*, der Quellcode vereinfacht sich und es stehen viel mehr und komplexere grafische Elemente zur Verfügung. Dies wird erreicht, da *Swing* nicht mehr den gemeinsamen Nenner aller Plattformen berücksichtigen muss. Ein weiteres Feature ist ein einheitliches Look-and-Feel der Anwendung auf allen Plattformen, das sogar während der Laufzeit geändert werden kann.

[10] Basiert auf das Observer-Design-Pattern

Die eigene Berechnung und Darstellung von GUI-Elementen ist jedoch sehr Rechenintensiv und benötigt viel Arbeitsspeicher, wodurch *Swing*-Anwendungen oft langsam und träge erscheinen. [18, S-779-782].

3.2.1 Standard Widget Toolkit (SWT)

Eine weitere Möglichkeit effiziente grafische Oberflächen in Java zu entwickeln ist das *Standard Widget Toolkit* (*SWT*). Diese Grafikbibliothek wurden von IBM entwickelt und setzt auf Ressourcen des jeweiligen Betriebssystems auf, ähnlich wie das AWT. Hierbei werden mehr als nur Primitivoperationen des ausführenden Betriebssystems verwendet und über JNI[11] integriert. In diesem Kontext teilt sich die Implementation in eine java-basierende und eine native[12] Bibliothek auf. Der Java-Teil wird als JAR[13] ausgeliefert und beinhaltet alle Klassen und Schnittstellen für die Verwendung in Java und sorgt für die Anbindung der nativen Bibliothek. Der Native-Teil wird z.B. in Windows von einer DLL-Datei repräsentiert und übernimmt die eigentliche Ansteuerung der entsprechenden Resourcen des Operationssystems. Dieser Teil muss immer spezifisch für die jeweilige Plattform vorliegen und ist für die verbreitetsten Betriebssysteme verfügbar. Darunter Microsoft Windows, Apple MacOS, Linux, Solaris, u.a.. Daher kann man auch *SWT* als nahezu Plattformunabhängig ansehen. Durch die Verwendung von 'schwergewichtigen' grafischen Komponenten des jeweiligen Betriebssystems, sind *SWT*-Anwendungen sehr performant, arbeiten Speicher-Effizient und haben ein, an die jeweilige Plattform angepasstes Look-and-Feel.[20, S.2-19]

Einen zentralen Bestandteil, sowie die Schnittstelle für die Kommunikation mit dem Betriebssystem stellt die Klasse *OS* dar. Diese Klasse steuert die jeweiligen Funktionen des Betriebssystems mit der Hilfe von *JNI* an. Eine weitere zentrale Klasse von *SWT* ist *org.eclipse.swt.widgets.Display*. Ein Objekt dieser Klasse vermittelt zwischen der *OS*-Klasse und den einzelnen *Widget*s. Hierbei nimmt es eine vermittelnde Rolle für die Verwaltung und der Übersetzung der System-Events ein. Zudem verwaltet es Fenster und Dialoge, Farben, Schriften, und neben läufige

[11] Java Native Interface, ist eine Schnittstelle zur Anbindung von plattformspezifischen Funktionen.

[12] Native (lat. für 'heimisch'), wird in der Softwaretechnik für Programme verwendet, die für eine bestimmte Plattform übersetzt wurden.

[13] Java Archiv (JAR) ist eine Datei, die Java-Klassen und zusätzliche Metadaten beinhaltet

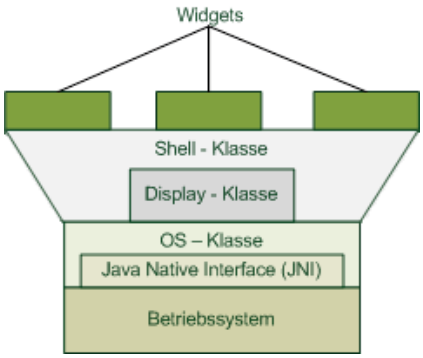


Abbildung 3.13: Die Klassen-Kommunikations-Struktur von SWT

Prozesse (*Threads*). Ein Fenster wird in *SWT* von **.Shell* repräsentiert und kann mit anderen *Widget*s angereichert werden. Das *org.eclipse.swt.widgets.Widget* ist die abstrakte Oberklasse aller grafischen Elemente, welche der Abbildung 3.14 zu entnehmen sind.[20, S.19-22]

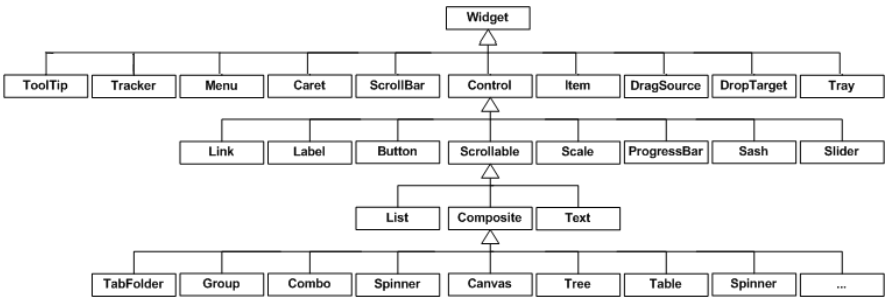


Abbildung 3.14: Die Klasse *Widget* und deren Unterklassen

3.2.2 JFace

Die von *SWT* zur Verfügung gestellten elementaren grafischen Objekte werden von *JFace* verwendet, um höherwertige GUI-Komponenten zu realisieren. An dieser Stelle abstrahiert *JFace* von den technischen Details des *SWT* und realisiert das MVC-Architekturmuster[14]. Hierbei werden Tabellen, Bäumen oder Listen durch die Oberklasse *Viewer* bzw. deren Unterklasse *ContentViewer* repräsentiert. Um das MVC-Prinzip sicherstellen zu können, beinhaltet ein *ContentViewer* einen *Con-*

[14]Das Model-View-Controller-Architekturmuster beschreibt eine Trennung des Datenmodells, der Programmsteuerung und der Darstellung.

tentProvider, der das Datenmodell vorbereitet, und einen *LabelProvider*, der für die Präsentation der Daten verwendet wird. Die Programmsteuerung übernimmt hierbei der entsprechende *Viewer*, z.B. die Klasse *TableViewer*. [10, S.153]Eine ähnliche Abstraktionsebene wird für *Menu*s, *ToolBar* und *CoolBar* mit Hilfe von *IContributionManager* realisiert. Dieses Interface beinhaltet *IContributionItem*s bzw. *IAction*s. Hierbei repräsentiert der *IContributionItem* den darstellenden und eine *IAction* den ausführenden Teil. Hierzu beinhaltet die *IAction* u.a. eine *run()*-Methode, die bei Interaktion des Benutzers ausgeführt wird.[20, S.63-65]

Des Weiteren bietet *JFace* komplexe Fensterkonstrukte, die über definierte Schnittstellen für eigene Anwendungen genutzt werden können. Die Oberklasse dieser Fenster wird von *Window* repräsentiert, welches die Grundlage für das *ApplicationWindow*, den *Dialog* und den *PopupDialog* bildet. Ein berühmter Einsatzplatz des *ApplicationWindow* bietet die Eclipse, deren Hauptfenster durch das *WorkbenchWindow*, einer Unterklasse von *ApplicationWindow*, realisiert wurde.[10, S.150-151]
Ein weiteres Konzept ist der *Wizard*. Ein *Wizard* besteht aus einer Folge von Dialogen, die den Anwender durch verschiedene Arbeitsschritte leitet, z.B. ein Installations-Wizard. In *JFace* wird ein *Wizard* durch das Interface *IWizard* repräsentiert, der aus einer Ansammlung von *IWizardPage*s besteht. Das Interface

Abbildung 3.15: Klassenstruktur des JFace-Wizards

IWizard und *IWizardPage* wird von den abstrakten Klassen *Wizard* und *WizardPage* implementiert und stehen für eigene Unterklassen zur Verfügung. [20, S.239-244]

3.3 Eclipse-Rich-Client Plattform (RCP)

Unter Rich-Client wird im Rahmen einer Server-Client-Architektur, ein Client bezeichnet, der zusätzlich zu der Verarbeitung der Daten vor Ort auch eine reichhaltige Problemlösung anbietet. Meist ist diese Software durch Module oder Plugins erweiterbar. Eine solche Lösung bietet die Eclipse-Rich-Client Plattform. Die Eclipse wurde von OTI, einer IBM Kanada Tochter, als Nachfolger von *IBM Visual Age*

for Java enwickelt und vorerst als reine IDE konzipiert. Bis einschließlich der Version 2.1 wurde sie als Entwicklungsumgebung ausgeliefert, die modular erweiterbar war. Gleichzeitig wurde deutlich, dass diese Plugin-Struktur auch für andere Anwendungen eine geeignete Basis darstellen könnte. Als Folge dieser Erkenntnis, wurde die IDE-Funktionalität in einzelne Plugins ausgelagert, sodass die Eclipse IDE, ab der Verion 3.0, nun mehr eine erweiterte RCP-Anwendung darstellt. [10, S.9-10]

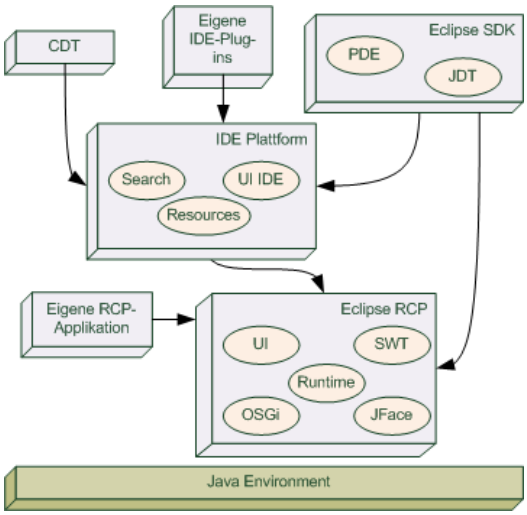


Abbildung 3.16: Grobstruktur von RCP - Eclipse

3.3.1 Der Plugin Mechanismus

Eine weitere Änderung in der Eclipse Version 3.0 war die Umstellung auf das OSGI[15]-Framework[16]. Diese Plattform bietet eine standardisierte Ablaufumgebung für Java-Module, die so genannten *Bundles*. Konzipiert wurde dieser Standard von einem Zusammenschluss von führenden amerikanischen Unternehmen wie SUN Microsystems, IBM, Oracle und Echelon und sollte eine neue Generation von Internet Services im Heim, Auto, kleinen Büros und Fabriken ermöglichen. Ein OSGI-konformer Dienst kann über das Netzwerk in Form eines Bundles verteilt werden und auf OSGI-konformen Servern gestartet werden. Der in der Abbildung 3.17 er-

[15]Open Service Gateway Initiative

[16]Ein Framework ist ein softwaretechnisches Rahmenwerk, das über Schnittstellen in eigene Anwendungen integriert werden kann.

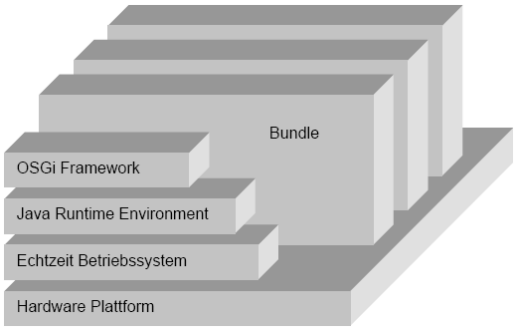


Abbildung 3.17: Schichtenmodell einer OSGi - Implementierung

sichtliche modulare Aufbau und die Möglichkeit des Einbinden bzw. Ausschließen von Bundles während des Betriebes zeichnen diese Plattform besonders aus. Gedacht war es für eingebettete (engl. embedded) Systeme, deren Leistungsfähigkeit stark eingeschränkt ist und so ausschließlich die Dienste beziehen und ausführen sollten, die sie für ihren Aufgabenbereich benötigten. [23, S.1-2]

Ein RCP-Plugin ist äquivalent zu einem OSGI-Bundle, obwohl sie durch unterschiedliche Klassen repräsentiert werden. Diese Plugins werden über die Eclipse Runtime ausgeführt. Hierbei bekommt jedes Plugin einen eigene Instanz von der Klasse *ClassLoader* zugewiesen und somit einen eigenen Namensraum. Dies garantiert, dass die unterschiedlichsten Plugins, ohne Konflikte im Namensraum, nebeneinander ausgeführt werden können. Interaktionen zwischen einzelnen Plugins werden ausschließlich über den Eclipse-Kernel vermittelt.[10, S.27-28]
Eine Plugin ist eine Ansammlung von Dateien und eines Manifests. Die Dateien können Programmcode beinhalten, aber auch Bilder, Dokumentationen und ähnliches. Das Manifest beschreibt das Plugin, sowie seinen Beziehungen zu anderen Modulen. In Abbildung 3.18 ist der Aufbau eines typischen Manifests dargestellt. [17, S.15-20] Um Erweiterungen für ein Plugin zu ermöglichen, müssen so genannte *ExtensionPoint*s definiert werden. Die Definition wird in der Datei *plugin.xml* durchgeführt. Auf der anderen Seite werden diese *ExtensionPoint*s durch *Extensions* erweitert. Ein Beispiel hierzu sind *ActionSets*. Die Aktionen werden von dem Modul *org.eclipse.ui* verwaltet. Für diesen Zweck besitzt dieses Plugin den *ExtensionPoint* „actionSets“, wie in Abbildung 3.19 ersichtlich. Dieses System macht es für andere Plugins möglich, ihre eigenen *Action*s anzubinden. Hierfür muss zum einen eine *Action*-Klasse, die das Interface *IActionDelegate* implementiert erstellt werden und zum anderen die Erweiterung in der „plugin.xml“ registriert werden.

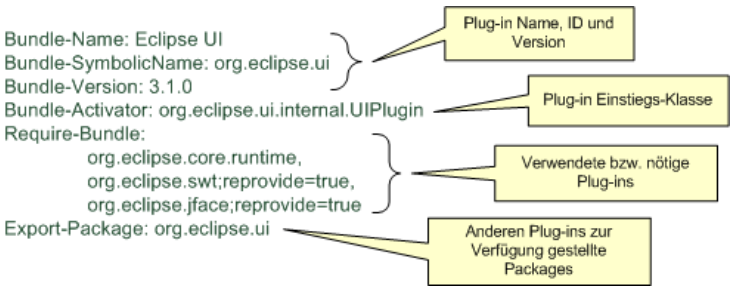


Abbildung 3.18: Plugin Manifest

```
<extension-point id="actionSets" name="Action Sets"/>
```

Abbildung 3.19: RCP-Extension-Point in „**org.eclipse.ui/plugin.xml**“

Ein Beispiel hierfür zeigt die Abbildung 3.20. In der XML-Datei aus Abbildung

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet id="beispiel.demoActionSet">
       <action
           id="beispiel.demo"
           class="beispiel.DemoAction"
           icon="icon/demo.gif"
           label="Demonstration"/>
   </actionSet>
</extension>
```

Abbildung 3.20: RCP-Extension in „**beispiel/plugin.xml**“

3.20 wird die Identifikation, die Klasse, ein Bild, sowie der Namen der Aktion gesetzt.[17, S.17-22]

In RCP gibt es zwei unterschiedliche Varianten von Produkten. Das reine Plugin-Projekt, das sich in eine schon bestehende Applikation einbinden lässt, oder als eine RCP-Applikation, die komplett eigenständig abläuft. Beide Arten benötigen Eintrittsklassen. Hierzu dient das Interface *BundleActivator*, das von der abstrakten Klasse *Plugin* implementiert wird. Für grafische Plugins, die in RCP Manier mit UI bezeichnet werden, steht die abstrakte Klasse *AbstractUIPlugin* zur Verfügung. Da eine Applikation in erster Linie auch ein Plugin darstellt, unterscheidet sich die Implementierung an dieser Stelle nicht. Für eine Applikation muss jedoch noch eine Klasse vom Interface *IApplication* erzeugt werden. Ob *BundleActivator* oder *IApplication*, beide besitzen eine *start()*- und *stop()*-Methode. Der einzige Unterschied ist der Kontext der Ausführung. So wird dem *BundleActiovator* ein *BundleContext* und der *IApplication* ein *IApplicationCon-*

text als Parameter übergeben. Die eigene Implementation von *IApplication* muss in der Datei „plugin.xml“ registriert werden. Hierzu existiert der Extension-Point *org.eclipse.core.runtime.applications*. [17, S.460][10, S.26-29]

```
<extension
      id="de.unikoblenz.beispiel"
      point="org.eclipse.core.runtime.applications">
   <application>
      <run
            class="de.unikoblenz.beispiel.Application">
      </run>
   </application>
</extension>
```

Abbildung 3.21: Registration der *IApplication*-Klasse in der Datei *plugin.xml*

Ein Fenster benötigt einen *WorkbenchWindowAdvisor*, der die Wiedergabe des Fensters kontrolliert. Hierzu beinhaltet er z.B. die Methode *preWindowOpen()*, die vor dem Öffnen des Fensters ausgeführt wird. Ein weitere Methode ist *createActionBarAdvisor*, die zur Erzeugung des *ActionBarAdvisor* verwendet wird. Die Instanz vom *ActionBarAdvisor* erzeugt *IActions*, die für das Fenster benötigt wird und positioniert sie entsprechend.[17, S.50-52].

3.3.2 Der Eclipse-Workspace

Um auf das native Dateisystem zuzugreifen bietet Java die Pakete *java.io* und *java.nio* an. Eine Klasse dieser Pakete ist *File*, die eine Datei bzw. Ordner des Dateisystem darstellt. Jedoch erschweren die plattformspezifischen Pfadangaben die plattformunabhängige Implementierung einer Anwendung. So verwenden Unix-Systeme oder deren Derivate (Linux, Mac OS X) den Schrägstrich als Trennzeichen zwischen den Pfadsegmenten, wobei Windows den umgekehrten Schrägstrich verwendet. Dies kann man noch durch die Verwendung der Klassenvariablen *File.separator* kompensieren, jedoch die unterschiedliche Abbildung der Wurzelverzeichnisse ist so einfach nicht auszugleichen. Windows besitzt für jede Partition ein eigenes Wurzelverzeichnis (c:, d:, ...), wobei die Unix-Welt nur ein mit ’/’ gekennzeichnetes Wurzelverzeichnis kennt. [10, S.225]

Daher abstrahiert die RCP komplett von dem darunter liegenden nativen Dateisystem und bietet somit ein plattformübergreifendes Konzept. Dies wird mit dem *Workspace* erreicht, der ein in sich abgeschlossenes Dateisystem für eine RCP-Anwendung darstellt. Wie von Unix-Systemen bekannt, wird der Schrägstrich als einziges Wurzelverzeichnis behandelt. Zudem bietet es spezielle Funktionalitäten

wie eine lokale Historie, Markierungen und ähnliches.
Der *Workspace* wird von dem Plugin *org.eclipse.core.resources* bereitgestellt. Die entsprechende Instanz bekommt man von der Klasse *ResourcesPlugin* durch den Aufruf der Methode *getWorkspace()*. Diese Methode liefert eine Objekt des Interface *IWorkspace*. Um auf die einzelnen Ressourcen zugreifen zu können, benötigt man das entsprechende Wurzelverzeichnis des Workspace. Dies wird repräsentiert durch ein Objekt vom Interface *IWorkspaceRoot* und kann vom *IWorkspace* durch *getRoot()* bezogen werden. Eine Instanz von *IWorkspaceRoot* beinhaltet ausschließlich *IProject*s. Ein jedes Workspace-Objekt implementiert *IResource*. Eine Ressource gliedert sich in die Interfaces *IContainer* und *IFile*. Ein *IContainer* kann ein Verzeichnis (*IFolder*), ein Projekt (*IProject*) oder das Wurzelverzeichnis (*IWorkspaceRoot*) sein. Die konkreten Abhängigkeiten können der Abbildung 3.22 entnommen werden. [10, S.225-229]

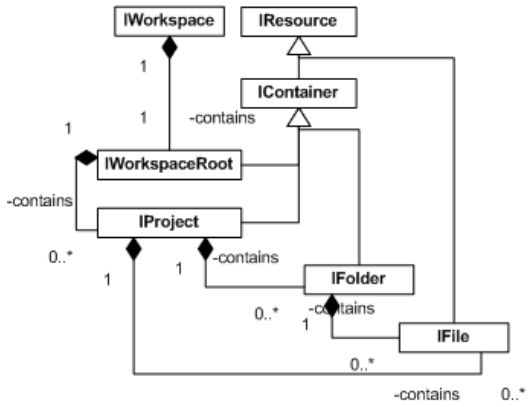


Abbildung 3.22: Der Eclipse-Workspace

Das Projekt wird von dem Plugin *org.eclipse.core.resources* und ist daher sehr abstrakt definiert. Um zusätzlichen Plugins die Einführung spezieller Eigenschaften bzw. Verhaltensmuster eines Projektes zu ermöglichen, besitzt ein jedes Objekt von *IProject* eine bestimmte Art in Form einer Klasse vom Interface *IProjectNature*. So besitzt ein Java-Projekt die Art *JavaProject*, die für java-spezifische Funktionalität und Einstellungen bereitsteht. Zum Beispiel die JDK-Einstellungen, den Classpath, usw.. [11, Project natures]

3.3.3 Die generische Benutzeroberfläche

Das Basiselement der RCP-Benutzeroberfläche ist die Workbench, die durch eine Instanz des Interface *IWorkbench* repräsentiert wird. Dieses Objekt wird durch einen Aufruf der statischen Methode *getWorkbench()*, der Klasse *PlatformUI* geliefert. Eine Workbench besteht aus einem oder mehreren Instanzen des Interface *IWorkbenchWindow*, welches jeweils ein Fenster darstellt. Um auf diese Fenster zuzugreifen, verfügt die Workbench über die Methoden *getWorkbenchWindows()* und *getActiveWorkbenchWindow()*. Die erste Methode liefert ein Array von *IWorkbenchWindow* und die zweite Methode das gerade aktive Fenster. Ein *IWorkbenchWindow* besteht seinerseits aus verschiedenen *IWorkbenchPage*s, die in der Anwendung als differente Perspektiven erscheinen. Ein Zugriff auf die Workbech-Pages kann über die Methode *getPages()* oder der Methode *getActivePage()* erfolgen. Analog zu dem Bezug von dem *WorkbenchWindow* liefert der Aufruf der entsprechenden Methode ein Array von *IWorkbenchPage*s bzw. die gerade aktive Seite. Jeder dieser Perspektiven bestehen aus Komponenten, die durch das Interface *IWorkbenchPart* abgebildet werden. Eine solches Bestandteil ist entweder eine *View* oder ein *Editor*.

Jede Workbench-Komponente implementiert das *IAdaptable*-Interface. Dieses stellt die Methode *getAdapter()* zur Verfügung, mit der eine Komponente nach einem konkreten Dienst gefragt werden kann. Falls die Komponente den Dienst unterstützt, wird eine Instanz der entsprechenden Klasse zurück gegeben, andernfalls liefert die Methode *null*.
Um weiter gehende Informationen zu der Ablaufumgebung bzw. um Zugriff auf die Manifest-Deklarationen zu erhalten, gibt es die Interface *IWorkbenchSite*, *IWorkbenchPartSite* mit den Erweiterungen *IEditorSite* und *IViewSite*. Die Anordnung der *View* und der *Editor* wird in einer Unterklasse von *IPerspectiveFactory* angegeben. Hierzu wir die Methode *createInitialLayout()* verwendet, die als Paramter ein *IPageLayout* erwartet. Dieses Layout besteht initial ausschließlich aus einem Editor-Bereich. Relativ zu diesem Bereich können *View*s mit der Methode *addView()* bzw. View-Bereiche mit Hilfe der Methode *createFolder()* angeordnet werden. Ein *Folder* wird über eine Unterklasse von *IFolderLayout* repräsentiert, welches wiederum in Bereiche unterteilt werden kann. Mit dem Aufruf *setEditorAreaVisible(false)* kann der Editor-Bereich ausgeblendet werden. Die *PerspectiveFactory* muss in gewohnter Weise in der „plugin.xml“ registriert werden. Hierzu stellt das

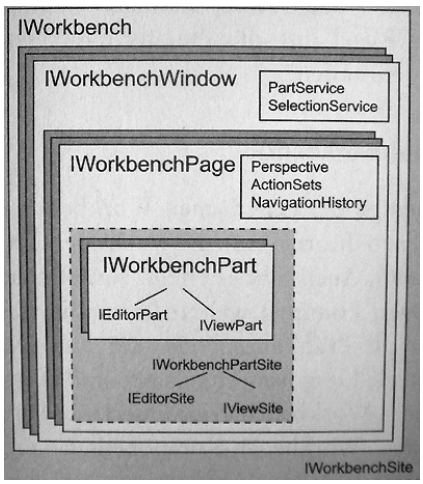


Abbildung 3.23: Strukturierung des RCP-Workbench

```
public class PerspectiveFactory implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
        IFolderLayout ersterBereich = layout.createFolder("ErsterBereich",
                IPageLayout.LEFT, 0.5f, layout.getEditorArea());
        ersterBereich.addView("beispiel.view");
    }

}
```

Abbildung 3.24: PerspectiveFactory im Einsatz

Plugin *org.eclipse.ui* den *ExtensionPoint perspectives* zur Verfügung. Die *Views* und *Editors* werden, ebenfalls unter diesem Plugin, mit Hilfe der ExtensionPoints *views* und *editors* eingebunden. [10, S.42-46]

```
<extension
      point="org.eclipse.ui.perspectives">
   <perspective
         name="beispiel.perspektive"
         class="de.unikoblenz.beispiel.PerspectiveFactory"
         id="de.unikoblenz.beispiel.PerspectiveFactory">
   </perspective>
</extension>
<extension
      point="org.eclipse.ui.views">
   <view
         name="beispiel.view"
         icon="icons/beispiel.gif"
         class="de.unikoblenz.beispiel.View"
         id="de.unikoblenz.beispiel.View">
   </view>
</extension>
```

Abbildung 3.25: Einbinden der *PerspectiveFactory* und der *View*

3.3.4 Java Development Tooling (JDT)

Die Eclipse beinhaltet eine vollständige Java Entwicklungsumgebung (IDE). Die komplette java-spezifische Funktionalität wird hierbei von dem *Java Development Tooling* (JDT) zur Verfügung gestellt. Diese Sammlung von Plugins ermöglicht es Benutzern Java Programme zu schreiben, zu editieren, zu kompilieren und zu testen. Hierbei unterteilt sich das JDT hauptsächlich in drei Komponenten.

- Das **JDT Core** beinhaltet eine komplette Infrastruktur zum kompilieren und manipulieren von Java-Code.
- Die **JDT UI** stellt die grafische Oberfläche der Java IDE.
- Das Modul **JDT Debug** ermöglicht das Ausführen, sowie das Testen von Java-Programmen.

3.3.4.1 Das Java-Modell

Das *JavaModel* ist ein essentieller Teil des *JDT*. Es wird verwendet um Elemente der Sprache Java innerhalb der IDE zu repräsentieren, die Beziehungen abzubilden und in einer Baumstruktur anzuordnen. Der Zugriff auf Java-Elemente wie

Klassen, Methoden und Variablen erfolgt ausschließlich über das Java-Modell. Das Wurzelelement vom Interface *IJavaModel* kann mit der Hilfe der Klasse *JavaCore* bezogen werden. Zu diesem Zweck beinhaltet sie die statische Methode *create()*. Jedes Element eines Java-Modell erweitert bzw. implementiert das Interface *IJavaElement*. Dieses Inteface sieht die Methode *getParent()* vor, die zu jedem Element den jeweilige Vorgänger ermitteln kann. Hierdurch wird eine Baumstruktur errichtet, die von den Blättern zu der Wurzel durchlaufen werden kann (Bottom-Up). Um auch die inverse Richtung zu ermöglichen, gib es das Interface *IParent*, mit der Methode *getChildren()* (Top-down). In Abbildung 3.26 ist die Vererbungshierachie ersichtlich. Die einzelnen Spezialisierungen von *IParent* und *IJavaElement* bilden

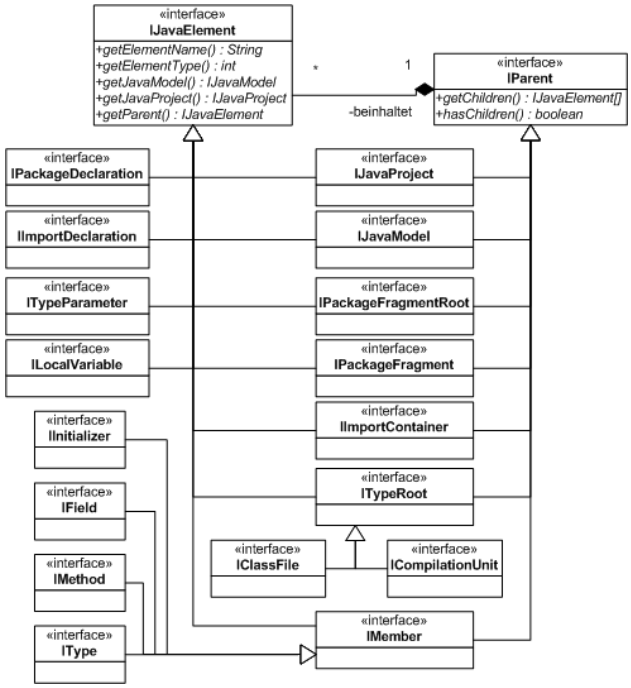


Abbildung 3.26: Die Vererbungshierachie des JDT-Java-Modells

die speziellen Java-Elemente ab. Besonders zu nennen sind hier:

- **IPackageFragmentRoot**, welches eine Liste von Elementen beinhaltet. Es kann sich hierbei um ein Verzeichnis oder einer Archiv-Datei handeln.
- **IPackageFragment**, ist das mögliches Unterobjekt von *IPackageFragmentRoot* und repräsentiert ein Java-Paket.
- **ICompilationUnit**, stellt eine Java-Quelldatei dar.

- **IPackageDeclaration**, bildet die Paket-Deklaration in einer *ICompilationUnit* ab.
- **IImportContainer**, beinhaltet alle *Import*-Anweisungen einer Java-Datei.
- **IImportDeclaration**, stellt eine *Import*-Anweisung dar und ist das Unterobjekt von *IImportContainer*.
- **IType**, repräsentiert eine Klasse, die durch eine *ICompilationUnit* erzeugt wird.
- **IField**, entspricht eine Variable innerhalb einer Klasse.
- **IMethod**, stellt eine Methode einer *IType* dar.
- **IInitializer**, bildet den Konstruktor einer Klasse ab.
- **IClassFile**, repräsentiert die kompilierte 'class'-Datei.
- **ILocalVariable**, realisiert die lokalen Variablen einer Methode oder eines Konstruktors.

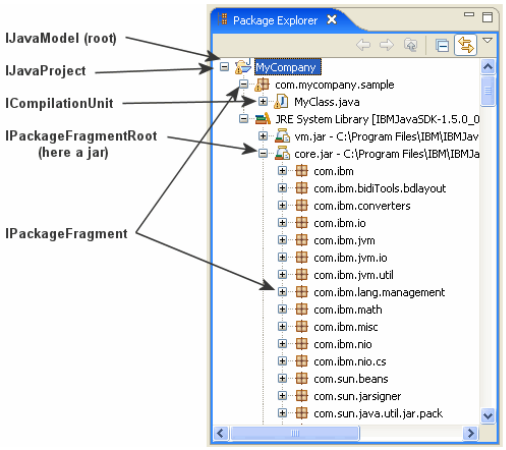


Abbildung 3.27: Eclipse IDE - Package-Explorer und das Java-Modell

Einige dieser Elemente sind in den Abbildungen 3.27 und 3.28 zu sehen. Diese Abbildungen zeigt zum einen den Package-Explorer und zum anderen den Java-Editor der Eclipse und deren Beziehungen zum Java-Modell. [11, JDT Plugin Developer Guide]

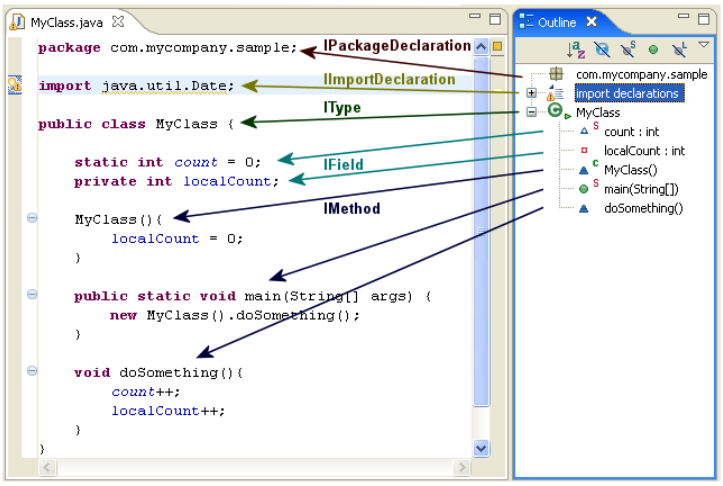


Abbildung 3.28: Eclipse IDE - Java-Editor und das Java-Modell

3.3.4.2 Der *Abstract Syntax Tree* (AST)

Das Java-Modell abstrahiert von dem darunter liegenden Quellcode und ermöglicht spezielle Operationen auf Java-Elemente (wie Umbenennen, Hinzufügen von Elementen usw.), sowie die Präsentation und das Traversieren von einem Java-Modell. Um jedoch ein solches Modell zu erhalten, müssen vorher Techniken des Compilerbaus auf den Quelltext angewendet werden. Hierzu durchläuft der Quellcode die Analyse-Phasen eines Compilers, um den Text in einen abstrakten syntaktischen Baum (engl. *abstract syntax tree*) zu transformieren. Die erste Phase ist die lexikalische Analyse, die mit Hilfe eines so genannten Scanners einzelne Zeichen des Quelltexts einliest und daraus Symbole[17] erzeugt, die von dem Parser syntaktisch analysiert werden können. Am Ende der syntaktischen Analyse steht der AST, worauf wiederum das Java-Modell aufsetzt. [2, S.101-104]

Der zentrale bestandteil dabei ist *ASTNode*. Diese abstrakte Klasse besitzt zahlreiche Unterklassen, die jeweils für ein syntaktisches Element steht. In der Abbildung 3.30 sind einige dieser Elemente ersichtlich.

Die Java-Elemente können nach den syntaktischen Regeln zu einem Baum aufgespannt werden. Mit Hilfe dieser Baumdarstellung können syntaktische Fehler erkannt werden und somit auch in der GUI dargestellt werden. Hierzu beinhaltet die Klasse *CompilationUnit* keine, eine oder mehrere Objekte vom Type *IProblem*. Ein solches Objekt beschreibt eine Warnungen oder ein Fehler durch den syntak-

[17] Menge von Zeichenfolgen, die eine grammatikalische Bedeutung besitzen.

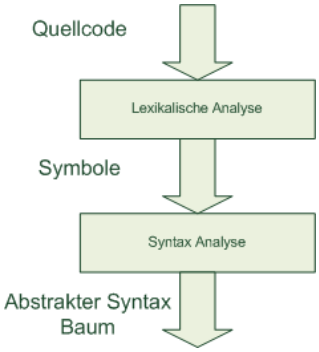


Abbildung 3.29: Vereinfachte Phasen-Darstellung hin zum AST

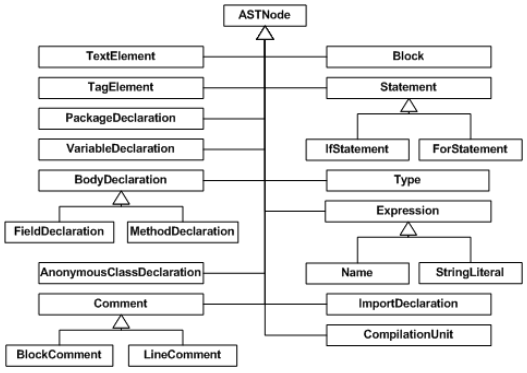


Abbildung 3.30: Auszug der Unterklassen vom *ASTNode*

tischen Grund und die entsprechende Position im Quelltext.

Der AST wird durch eine Instanz der Klasse *ASTParser* erzeugt. Hierzu kann

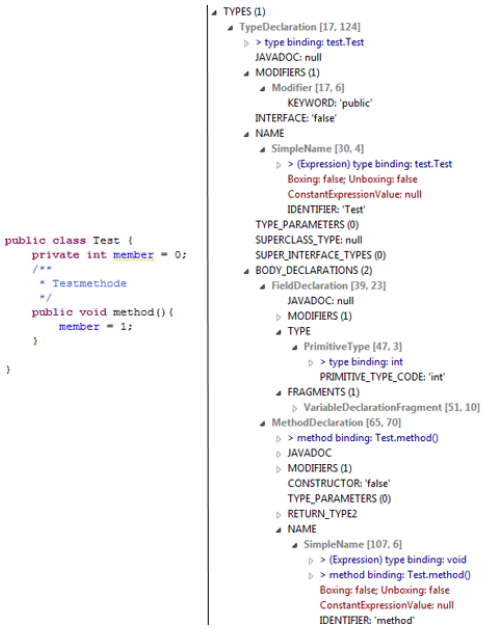


Abbildung 3.31: Quelltext und die Darstellung im AST

diesem Objekt der eigentliche Quellcode oder ein Objekt des Interfaces *IClassFile*, *ITypeRoot* sowie *ICompilationUnit* übergeben werden. Durch einen Aufruf der Methode *createAST()* wird der AST erzeugt und das oberste Element in Form eines *ASTNode*-Objektes zurückgegeben. Mit Hilfe des *ASTRewriter* können Änderungen des AST in Quelltext geschrieben werden, unter Berücksichtigung von Formatierungsrichtlinien und der minimal nötigen Veränderungen am Quellcode. Mit Hilfe der statischen Methode *ASTRewrite.create()* kann eine Instanz eines *ASTRewriter* bezogen werden. Als Parameter wird eine Instanz von *AST* erwartet. Die Klasse *AST* ist eine *ASTNode*-Factory und erzeugt jedes *ASTNode*-Objekt. Um eine Instanz des *AST* zu erhalten, muss die statische Methode *AST.newAST()* mit Angabe der Java Language Specification (JLS) aufgerufen werden. Für die Angabe der JLS werden zwei Konstanten in der Klasse *AST* vorgehalten. Hierbei steht die Konstante *JLS2* für Java bis einschließlich der Version 1.4 und die Konstante *JLS3* für die Versionen ab dem J2SE 5.[11, Manipulating Java code]

3.3.4.3 Compilieren von Java-Code

Das RCP beinhaltet einen inkrementeller Projekt-Erbauer (*project builder*), der die einzelnen Ressourcen des Projektes in einer bestimmten Weise transformiert um ein Produkt bzw. irgendein Artefakt zu erhalten. Zu diesem Zweck stellt die Plattform zwei unterschiedliche Arten des *project builder*s zur Verfügung.

- Der **full build** transformiert immer alle Ressourcen und ignoriert den Bezug auf vergangene Durchläufe.
- Der **incremental build** verwendet den Status des letzten Durchlaufs und transformiert lediglich die Ressourcen, die sich seit der letzten Transformation geändert haben.

Zudem gibt es noch die Möglichkeit, alle Transformationen zu löschen (*clean*). In diesem Fall würde der *incremental build* neu aufsetzen und im ersten Durchlauf alle Ressourcen transformieren.
Der Transformationsvorgang kann für ein spezifisches Projekt oder für den gesamten Workspace erfolgen. So besitzen die Interfaces *IProject* und *IWorkspace* die Methode *build()*, der die Art der Transformation übergeben werden muss. Hierfür wurden in der Klasse *IncrementalProjectBuilder* eine Reihe von Konstanten definiert. Neben den zwei oben genannten Arten gibt es noch zwei Arten, die jedoch keinen direkten Einfluss auf die Transformation haben. Der *clean build* löscht alle vorherigen Transformationen und baut das Projekt neu. Der *auto build* führt die Transformation automatisch durch, falls sich eine Ressource geändert hat.[11, Incremental project builders]

Das JDT setzt auf diesen *Build*-Mechanismus auf und erweitert diese durch die konkrete Umsetzung einer Transformation in Form eines Übersetzungsvorgangs in Java. Hierzu verwendet das JDT keinen eigenen Java-Compiler sondern setzt auf ein System, bzw. ein für das entsprechende Projekt angegebenes JDK auf. [11, Compiling Java code]

3.3.4.4 Ausführen von Java-Code

Sobald Java-Code compiliert wurden ist, bietet das *JDT Debug* zwei Möglichkeiten an, um ein Java-Programm auszuführen.
Die erste Variante funktioniert mit Hilfe des *IVMRunner*. Eine Instanz dieses *IVMRunner* kann über ein Objekt vom Typ *IVMInstall* bezogen werden, wobei ein

IVMInstall für die System-VM oder die speziell für ein Projekt konfigurierte VM steht. Das somit erhaltene Objekt kann, wie in der Abbildung 3.32 beschrieben, zur Ausführung von Java-Code genutzt werden.

```
// Ermitteln der Installierten VM
IVMInstall vmInstall = JavaRuntime.getVMInstall(myJavaProject);
// Falls es keine spezielle Projekt-VM gibt, wird die System-VM verwendet
if (vmInstall == null)
    vmInstall = JavaRuntime.getDefaultVMInstall();
// Ermitteln der Starterklasse, die die entsprechend VM ansprechen kann.
IVMRunner vmRunner = vmInstall.getVMRunner(ILaunchManager.RUN_MODE);
if (vmRunner != null) {
    // Ermitteln des ensprechenden Classpath des Projekts
    String[] classPath = JavaRuntime
            .computeDefaultRuntimeClassPath(myJavaProject);
    // Eine Konfiguration aus auszuführender Klasse und Classpath erzeugen.
    VMRunnerConfiguration vmConfig = new VMRunnerConfiguration(
            "MyClass", classPath);
    // Erzeugen eines Launch-Instanz, der später Informationen über das
    // laufende Programm liefern kann.
    ILaunch launch = new Launch(null, ILaunchManager.RUN_MODE,
            null);
    // Starten der Anwendung durch Angabe der Konfiguration und einer
    // Launch-Instanz
    vmRunner.run(vmConfig, launch, null);
}
```

Abbildung 3.32: Ausführen eines Java-Programms mit Hilfe des *IVMRunner*

Die zweite Möglichkeit ist das Ausführen einer Anwendung über eine Start-Konfiguration. Eine solche Konfiguration wird durch eine Instanz des Interfaces *ILaunchConfiguration* repräsentiert. Sie bietet eine Abstraktion von der Ausführungsumgebung und von der Lokalisierung der zu verwendeten VM, indem die Auswahl bzw. Ansteuerung komplett der Start-Konfiguration überlassen wird. Diese Art der Ausführung ist in der Abbildung 3.33 beschrieben.[11, Running Java code]

```
// Ermitteln eines ILaunchManager, der die Launch-Konfigurationen verwaltet.
ILaunchManager manager = DebugPlugin.getDefault().getLaunchManager();
// Ermitteln eines Konfigurationstyps
ILaunchConfigurationType type = manager
        .getLaunchConfigurationType(IJavaLaunchConfigurationConstants.ID_JAVA_APPLICATION);
// Setzten der Parameter, die für die Ausführung benötigt werden
ILaunchConfigurationWorkingCopy wc = type.newInstance(null,
        "SampleConfig");
wc.setAttribute(IJavaLaunchConfigurationConstants.ATTR_PROJECT_NAME,
        "myJavaProject");
wc.setAttribute(IJavaLaunchConfigurationConstants.ATTR_MAIN_TYPE_NAME,
        "myClass");
ILaunchConfiguration config = wc.doSave();
// Starten der Anwendung
config.launch(ILaunchManager.RUN_MODE, null);
```

Abbildung 3.33: Ausführen von Applikationen durch die *ILaunchConfiguration*

3.4 Qualitätssicherung und Qualitätskontrolle

Die Qualitätssicherung bzw. Qualitätskontrolle beschäftigt sich mit der Sicherstellung eines festgelegten Verhalten eines Produkts und stellt dafür eine Vielzahl von Maßnahmen zur Verfügung. Hierbei steht im Vordergrund, einen gegebenes Niveau zu garantieren, nicht den Ausbau des Produktes. Die Maßnahmen der Qualitätssicherung sind im Rahmen der Normenreihe *EN ISO 9000* zum Qualitätsmanagement dokumentiert.[25, S.547]
Des weiteren ist in der DIN-ISO-Norm 9126 der Begriff Software-Qualität wie folgt definiert:

> Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen. [16, S.6]

Diese Definition betont, dass sich Software-Qualität nicht an dem einem Kriterium beschreiben lässt, sondern ein Zusammenspiel aus mehreren Merkmalen ist.

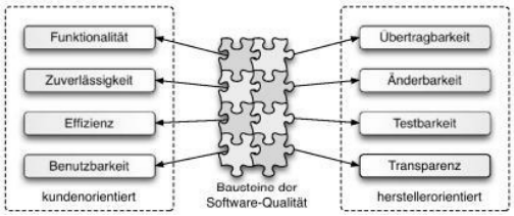


Abbildung 3.34: Qualitätsmerkmale einer Software [16, S.7]

Um Software-Qualität sicherstellen zu können, bedarf es eines Qualitätsmanagement, das sich laut Ian Sommervill u.a. in Qualitätssicherung und Qualitätskontrolle unterscheidet.

> *Qualitätssicherung:* Die Einrichtung eines Systems von organisationsspezifischen Verfahren und Standards, die zur Entwicklung qualitativ hochwertiger Software führen.
>
> *Qualitätskontrolle:* Die Definition und Umsetzung von Vorgehensweisen, die sicherstellen, dass die für die Produktqualität wichtigen Verfahren und Standards vom Softwareentwicklungsteam eingehalten werden. [25, S.546]

Im Rahmen der Qualitätssicherung und Qualitätskontrolle kann man durch Tests die Zuverlässigkeit sicherstellen. Gerade wenn eine Software geändert wird, betrifft eine Änderung oft Bereiche der Anwendung, die auf den ersten Blick nicht ersichtlich sind. Hierdurch werden neue Probleme geschaffen, die evtl. durch Tests der bestehenden Funktionalität verhindert werden können. Man unterscheidet Tests wie folgt:

- *Komponententest*, hierbei wird die angebotene Funktionalität einer atomaren Programmeinheit getestet. In objektorientierten Sprachen entspricht eine solche Einheit meistens einer Klasse, jedoch unterstützt diese vage Formulierung auch die Bildung komplexere Einheiten.
- *Integrationstest* beziehen sich auf das Zusammenspiel der Programmeinheiten untereinander. Hierzu wird die Funktionalität getestet, die eine Beziehung zu anderen Einheiten darstellt.
- *Systemtest* ist die Überprüfung des Verhaltens der gesamten Software gemäß den zuvor gesetzten Anforderungen.
- *Abnahmetest* wird vom Kunden bzw. Anwender durchgeführt, um sicherzustellen, das die gegebenen Anforderungen durch die Software abgedeckt wird.

[16, S.159-160]

3.4.1 Automatisierte Tests mit JUnit

Eine Möglichkeit Komponententests (Unit-Tests) in Java zu realisieren, ist die Verwendung des frei verfügbaren Frameworks *JUnit*. Dieses Framework ist standardmäßig in der Eclipse integriert und bietet einige Vorteile:

- Automatisiertes Ausführen von Testfällen
- Ausführen von mehreren Tests gleichzeitig
- Isolierte Ausführung einzelner Testfälle
- Einfaches Erlernen und Realisieren von Tests
- ...

[S.4,9][3]

Den Rahmen eines *JUnit*-Test bildet die Implementation der abstrakten Klasse *TestCase*. Die Namensgebung sollte sich nach dem Namen der zu testenden Klasse richten und mit 'Test' enden (z.B. *ConnectionTest* für den Test der Klasse *Connection*). Einzelne Testfälle werden durch parameterlose Methoden ohne Rückgabewert repräsentiert. Hierbei muss der Name einer solchen Methode immer mit 'test' beginnen (z.B. *testIsAvailable()*). Dieses Prefix wird von *JUnit* genutzt, um per Reflektion alle Testfälle eines Tests ermitteln zu können. Vor bzw. nach jedem Aufruf eines Testfalls werden die Methoden *setUp()* bzw. *tearDown()* ausgeführt. Dabei kann die Methode *setUp()* genutzt werden, um das Umfeld für den Testfall vorzubereiten. Der Aufruf von *tearDown()* kann zum Rücksetzten der Umgebung genutzt werden. Zum Beispiel könnte in *setUp()* eine Datei angelegt werden, die von einem Testfall genutzt wird. Die Methode *tearDown()* würde in einem solchen Fall diese Datei wieder löschen.
Innerhalb dieser Testmethoden kann durch Zusicherungen (engl. asserts) Sachverhalte geprüft werden. Zum Beispiel ob die Annahmen '*zahl > 10*' oder '*name != null*' wahr sind. Hierzu stellt die Klasse *Assert* einige Methoden zur Verfügung, wie zum Beispiel *assertTrue()* oder *assertNotNull()*. Falls die Zusicherung fehlschlägt, wird ein *AssertionFailedError* ausgelöst und es wird mit dem nächsten Testfall fortgefahren. Erfolgreich ausgeführte und fehlgeschlagene Testfälle werden in einem Objekt von Typ *TestResult* zusammengefasst und zum Beispiel von der Eclipse grafisch ausgegeben. Die Abbildung 3.35 gibt einen Einblick in die wichtigsten Klassen und Methoden von *JUnit*.

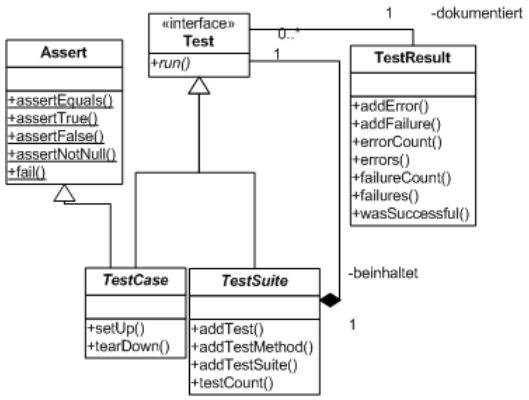


Abbildung 3.35: Klassenstruktur von *JUnit*

Einzelnen Test können zu einer *TestSuite* zusammengefasst werden. Eine *TestSuite* kann u.a. durch die implementation der statischen Methode *suite()* mit dem Rückgabewert *Test*, in der entsprechenden Unterklasse von *TestCase* erzeugt werden. Wird nun diese Testklasse ausgeführt, wird als Grundlage für die Ausführung einzelner Testfälle die *TestSuite* verwendet.[3, S.10-21]

3.5 Externe Datenbank

Verwendete CoMICS I noch eine integrierte Datenbank, so setzt CoMICS II nun auf eine externe relationale Datenbank auf. Der Einsatz einer integrierten Datenbasis führt zwangsläufig zu folgenden Problemen:

- **Beschränkte Zugriffsmöglichkeiten** - Es ist nicht ohne erhöhten Aufwand möglich, Daten in einer anderen Anwendung zu verwendet.
- **Probleme des Mehrbenutzerbetriebs** - Ein gleichzeitiger Zugriff auf die Datenbasis von unterschiedlichen Rechnern oder Programmen ist nicht möglich.
- **Verlust der Daten** - Die Gefahr von Datenverlust ist bei einer integrierten Datenhaltung viel höher.
- **Integritätsverletzung** - Die Integrität der Daten wird nicht, oder nur unzureichend gewährleistet.
- **Sicherheritsprobleme** - Nicht jeder Benutzer darf Zugriff auf alle Daten haben.
- **Hohe Entwicklungskosten** - Bei Neuentwicklungen muss auch oft die Datenbasis neu modelliert und entwickelt werden.

Nicht alle diese Punkte treffen auch auf eine Anwendung wie CoMICS zu, jedoch gibt es auch einige Parallelen. So ist kaum ein paralleler Zugriff von einem externen Auswertungsprogramm auf die Daten möglich. Zudem gab es nur sehr eingeschränkte Integrität-Überprüfungen. Diese Missstände können durch den Einsatz einer eigenständigen Datenbank beseitigt werden.[1, S.17-18]

3.5.1 Relationale Datenbanken

Um einen gewissen Grad der Datenunabhängigkeit zu gewährleisten zu können, beinhaltet ein DBS (Datenbanksystem) 3 Ebenen, die eine Abstraktion von den Daten und deren Persistenz ermöglichen. Durch wohldefinierte Schnittstellen zwi-

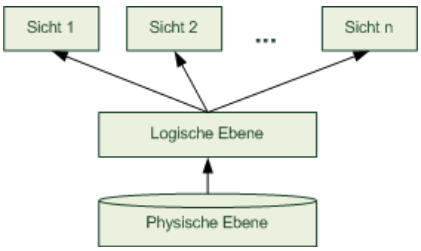


Abbildung 3.36: Abstraktionsebenen eines Datenbanksystems

schen den Ebene werden die darunter liegenden Schichten verdeckt, wodurch sich die Implementation der einzelnen Schichten verändern kann, Voraussetzung die Schnittstellen werden beibehalten. Hierdurch ergeben sich zwei Stufen der Datenunabhängigkeit:

- **Physische Datenunabhängigkeit:** Modifikationen an der physischen Speicherstruktur hat keinen Einfluss auf die logische Ebene.
- **Logische Datenunabhängigkeit:** Änderungen an der logischen Ebene ziehen keine Anpassungen an die physischen Ebene und im Idealfall auch keine Anpassungen in der Applikation nach sich.

Da jedoch Anwendungen meistens direkt auf der Datenstruktur der logischen Ebene arbeiten, ist eine logische Datenunabhängigkeit in heutigen Systemen meist nicht gegeben.[1, S.18-21]

Die logische Datenbankebene beinhaltet eine Implementation eines Datenmodells. Das Modell beschreibt die Datenobjekte, sowie die Festlegung der anwendbaren Operationen und deren Wirkung. Ein Modell wird in einer Sprache beschrieben, die sich dem entsprechend aus zwei Teilsprachen zusammensetzt:

- Die **Datendefinitionssprache** (engl. Data Definition Language, DDL) und
- die **Datenmanipulationssprache** (engl. Data Manipulation Language, DML).

Die Datenbankobjekte werden mit Hilfe der DDL erzeugt (wie z.B. einer Tabelle), wobei die Strukturbeschreibung aller Datenbankobjekte eines Anwendungsbereich

Datenbankschema genannt wird. Des weiteren kann man die DML in die Abfragesprache (engl. Query Language) und die Sprache zum Ändern, Einfügen und Löschen von Daten unterteilen.[1, S.20-22]

Ein Datenmodell beschreibt einen kleinen Ausschnitt der realen Welt und kann mit Hilfe unterschiedlicher Methoden konzeptuell entworfen werden:

- **Entity-Relationship-Modell (ERM)**
- **Semantisches Datenmodell**
- **Funktionales Datenmodell**
- **Objektorientiertes Entwurfsmodell**

Das objektorientierte Entwurfsmodell orientiert sich hauptsächlich an dem, im Abschnitt 3.1.2 genannte Paradigma. Neben dem objektorientierten Ansatz ist das ERM für diese Arbeit interessant. Dieses Modell besteht grundsätzlich aus Gegenstände (Entities) und deren Beziehungen (Relations). Entities sind unterscheidbare physisch oder gedanklich existierende Konzepte des zu modellierenden Bereichs. Gegenstände werden zu Gegenstandstypen (Entitytypen) gruppiert, die ERM grafisch als Rechtecke darstellt. Beziehungen werden analog zu den Gegenständen in Beziehungstypen eingeordnet, die eine Verbindung zwischen den Entitytypen herstellt. Beziehungstypen werden als Raute dargestellt. Entitytypen und Relationstypen können durch Attribute beschrieben werden. Diese werden durch Kreise bzw. Ovale grafisch dargestellt. Die Attribute, die eine Entität bzw. eine Relation eindeutig repräsentieren, werden unterstrichen und Schlüsselattribut (bzw. Primärschlüssel) genannt.
Der konzeptuelle Entwurf wird nach der Modellierung in eine Datenbank mit Hilfe der DDL übertragen. Eine Datenbanksprache die sich für relationalen Datenbanken durchgesetzt hat ist SQL (Structured Query Language). In einer relationalen Datenbank werden Gegenstandstypen und manche Beziehungstypen durch Tabellen dargestellt. Eine Spalte einer Tabelle ist hierbei ein Attribute und eine Zeile eine Entity. Beziehungen zwischen Entities werden durch Fremdschlüssel hergestellt, d.h. es wird in der entsprechenden Tabelle das Schlüsselattribut der zu verknüpfenden Tabelle angegeben. [1, S.29-36] Die Abbildung 3.37 zeigt eine Tabelle Person, die über einen Fremdschlüssel (foreign key) die Tabelle Haushalt über dessen Primärschlüssel (primary key) anbindet.

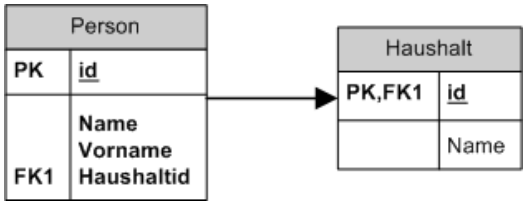


Abbildung 3.37: Tabelle Person verweist auf Haushalt

Eine DBS unterstützt jedoch nicht nur die Speicherung und Verarbeitung großer Datenmengen, sondern auch die Gewährleistung der Konsistenz der Daten. Hierzu dienen die *semantische Integritätsbedingungen*, die aus den Eigenschaften des Datenmodell bzw. des darin abzubildenden Bereichs der Realität extrahiert werden können. Zum Beispiel wäre eine Integritätsbedingung, das ein Primärschlüssel einer Person eindeutig sein muss, oder das der Haushalt, auf den sich eine Person bezieht auch existiert.[1, S.151-153]

3.5.2 PostgreSQL

Der Ursprung von PostgreSQL geht bis ins Jahr 1977 zurück. Zu diesem Zeitpunkt begann die Entwicklungsarbeit der relationalen Datenbank Ingres an der Universität Berkeley. Die Firma Relational Technologies / Ingres Corporation übernahm 1985 dieses Projekt und entwickelten es im kommerziellen Umfeld weiter.
Im Jahr 1986 begann ein neues Forschungsprojekt zur Entwicklung eines objektrelationalen Datenbanksystems auf der Basis von Ingres. Das war die Geburtsstunde von Postgres. Die erste Version von Postgres kam 1989 heraus und hatte nur einen kleinen Kreis von Anwendern. Die Universität Berkeley schloss die Weiterentwicklung 1993 mit der Version 4.2 ab. Zwei Studenten der Universität übernahmen 1994 Postgres und implementierten serverseitig ein SQL-Modul. Diese Modul sorgte für den aktuellen Namen PostgreSQL, der 1996 mit der Version 6.0 eingeführt wurde. PostgreSQL wurde seit dem kontinuierlich weiterentwickelt und liegt 2008 in der Version 8.2 vor.

Hinter PostgreSQL steht keine Firma, die vom Umsatz der Vermarktung des Produkts abhängt ist, sondern das Projekt unterliegt dem Open-Source-Lizenzmodell. Eine Gemeinschaft von freien Entwicklern, die ihre Freizeit PostgreSQL verschrieben haben, entwickeln das Produkt weiter. Das Produkt, der Quellcode, sowie eine Dokumentation ist frei erhältlich und kann ohne Einschränkungen genutzt werden,

die einzige Bedingung ist die Weitergabe des Copyright-Hinweises des Urhebers.[15, S.11-13]

PostgreSQL ist über die Internetseite: http://www.postgresql.org/ kostenlos erhältlich. Es stehen Installationspakete für FreeBSD, Linux, Mac OS X, Solaris und Windows zur Verfügung.

3.6 Objekt-relationales Mapping (ORM)

Das Objekt-relationale Mapping beschreibt eine Technik, mit der eine, in einer objektorientierten Programmiersprache geschriebene Anwendung, ihre Objekte in eine relationale Datenbank persistieren kann. Hierzu werden die Daten von einer Repräsentation in eine andere transformiert. Um diese Transformation durchführen zu könne, müssen die Abbildungsinformationen in Form von Metadaten zur Verfügung stehen. Diese Informationen können u.a. die Schemainformationen, die Zuordnung von Tabelle zu Klasse oder die Beziehungen zwischen den Tabellen beinhalten. Ein solches Mapping impliziert eine gewissen Einbuße der Performance, was jedoch durch verschiedene Optimierungsmaßnahmen gemindert werden kann. Eine ORM-System muss folgende Funktionalitäten anbieten:

- Eine API zur Durchführung von CRUD-Operationen.
- Eine Sprache bzw. API zur Abfrage von Objekten.
- Eine Möglichkeit zur Spezifizierung der Metadaten für das Mapping.
- Eine ORM-Implementations-Technik zur Verwaltung von Objekten bzw. zur Bereitstellung von Optimierungsmaßnahmen.

[9, S.24-27]

3.6.1 Hibernate

Hibernate ist ein 'Werkzeug', das für ein automatisches und transparentes objekt-relationales Mapping sorgt. Hierzu bietet der Kern von Hibernate eine API und die Möglichkeit, Mapping-Meta-Informationen in XML-Dateien zu speichern. Aus diesen Mapping-Informationen können Java-Klassen (auch Entity-Klassen genannt) generiert werden, die für die Entitytypen der Datenbank stehen. Die jeweilige Instanz der Klasse repräsentiert eine konkretes Entity. Mit HQL (syntaktisch ähnlich

wie SQL) bietet Hibernate eine Abfragesprache. Zudem existieren programmatische Abfrage-Interfaces für Kriterien- und Beispiel-Abfragen. Hibernate Core ist unabhängig von irgendeinem Framework oder einer bestimmten Java Laufzeitumgebung.

Neben dem Hibernate Core gibt es zahlreiche Erweiterungen. Eine davon ist Hibernate Annotation, das mit Einführung der Java Annotationen im JDK 5.0 ermöglicht wurde. Java Annotationen wurden im Abschnitt 3.1.3.7 erläutert. Hierdurch können die Mapping-Informationen statt in einer XML-Datei direkt in dem Quellcode der Entity-Klassen angegeben werden. Zu diesem Zweck definiert Hibernate spezielle Annotationen, die einem Java-Elementen eine konkrete Funktion bzw. eine konkrete Speicherposition im relationalen Datenbankschema zuordnet.[9, S.31-32]

Eine Möglichkeit automatisch an die Entity-Klassen zu gelangen, ist das *Reverse Engineering*. Damit wird versucht, möglichst viele Informationen über die Datenbank von JDBC[18] zu beziehen. Da jedoch JDBC nicht alle nötigen Metadaten liefert, wird dieser Prozess durch eine spezielle Reverse-Engineering-XML-Datei unterstützt.[9, S.80,81]

Sind die Entity-Klassen und alle Konfigurationen vorhanden, können die jeweiligen Objekte über eine Instanz der Klasse *Session* bezogen werden. Eine solches Objekt wird von der *SessionFactory* geliefert, die wiederum von der Klasse *Configuration* (bzw. der Unterklasse *AnnotationConfiguration*, falls Hibernate Annotation verwendet wird) geliefert wird. Die *Session* übernimmt die Verwaltung der Datenbank-Verbindung, der Transaktionen, sowie jeder CRUD-Operation. [9, S.86-91]

3.6.1.1 Hibernate Abfragen mit HQL

Die Hibernate Query Language (HQL) entspricht zum größten Teil SQL, jedoch werden die Klassen- statt den Tabellennamen und die Attribute- statt den Spaltennamen angegeben. Mit HQL ist es auch möglich, Objekte direkt in der Datenbank zu aktualisieren oder zu löschen, ohne das Objekt in den Speicher einzulesen. Einfache Anweisungen sind:

- Die **Select**-Anweisung, mit der Daten aus der Datenbank abgefragt werden können. z.B

[18] Java Database Connectivity, ist eine Datenbankschnittstelle von Java

from Person p where p.age > 21 and p.age <31

- Die **Update**-Anweisung, mit der Daten direkt in der Datenbank geändert werden können. z.B

  update Person p set p.isDeath =: true

- Die **Delete**-Anweisung, die Daten direkt aus der Datenbank löscht. z.B.

  delete Person p where p.id = 12

- Die **Insert**-Anweisung, die Daten direkt in die Datenbank eintragen kann. z.B.

  insert into Person (id, name, age) values (13, 'Herbert', 29)

Eine weitere Funktionalität von HQL ist die Unterstützung von Prepared Statements, d.h. es können Platzhalter in der Anfrage gesetzt werden, die später durch konkrete Werte ersetzt werden. Diese Technik führt dazu, dass eine Anfrage nur einmal gesendet werden muss und anschließend für unterschiedliche Werte mehrfach ausgeführt werden kann. Es gibt zwei Möglichkeiten der Verwendung von Prepared Statements mit HQL. Entweder es werden Bezeichner verwendet, die einen vorgestellten Doppelpunkt haben (z.B. '...p.age = :age'), oder es werden Fragezeichen in die Abfrage eingeführt, die nach ihrer Position (gezählt von Links nach Rechts) gefüllt werden (z.B. '...p.age = ?'). [9, S.473-476]

3.7 Der *Mersenne Twister*

Generell spielen in Simulationen die verwendeten Zufallszahlengeneratoren eine zentrale Rolle. Alle dynamischen Elemente der Simulation sind in irgendeiner Art und Weise auf unterster Ebene auf Zufallszahlen zurückzuführen. Aus diesem Grund wurde in der aktuellen Version von CoMICS der von Makoto Matsumoto und Takuji Nishimura entwickelte *Mersenne Twister* als Zufallsalgorithmus verwendet. Er ist einer der modernsten Algorithmen, die zur Zeit im Umlauf sind. Er zeichnet sich durch die folgenden Eigenschaften aus:

- **Periodenlänge** von $2^{19937} - 1 \approx 4,3 \cdot 10^{6001}$

- Der Algorithmus ist sehr **performant**. Jeder andere - ähnlich gute Algorithmus - ist deutlich langsamer
- Ergebnissequenzen sind hochgradig **gleich verteilt**. Die Korrelation zwischen aufeinanderfolgenden Wertefolgen ist sehr gering.
- Die **Bits** auf der untersten Ebene sind ebenfalls in sich gleich verteilt

In CoMICS II wird eine Java-Version, des original in C implementierten Algorithmuses verwendet. Der Originalcode in C, sowie die genaue Funktionsweise des Algorithmuses wird auf der offiziellen *Mersenne Twister* Webpräsenz unter http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html beschrieben.

3.8 *OpenCSV*

In CoMICS werden Eingabe und Ausgabedaten sehr oft als CSV-Files bereitgestellt. Diese Files genießen in der Simulationsentwicklung eine sehr hohe Akzeptanz, da sie zu fast allen Softwareprodukten kompatibel sind. Um diese Files schnell und einfach verarbeiten zu können, wurde in CoMICS II die freie Java-Bibliothek *OpenCSV* von Glen Smith eingebunden. Es ist ein relativ einfach gehaltenes Package, welches die im Folgenden aufgeführten Funktionen unterstützt.

- Beliebig viele Werte pro Zeile.
- Ignorieren von Kommas und Zeilenwechsel innerhalb zusammenhängender Elemente. Hierfür werden die Elemente üblicherweise in Anführungszeichen gesetzt.
- Die Zeichen nach denen separiert und gequoted wird, sind frei konfigurierbar.
- Alle Daten auf einmal oder zeilenweises einlesen wird unterstützt.
- Generierung von CSV-Dateien aus String-Arrays.

Die genauere Java API, der Quellcode sowie weitere Informationen zum Programmpacket können auf der Projekthomepage unter http://opencsv.sourceforge.net eingesehen werden.

3.9 *XStream*

Eine weitere Möglichkeit Daten in CoMICS zu laden, bzw. aus CoMICS heraus zu exportieren wird über XML bereitgestellt. Neben CSV findet der XML Standard immer Anhänger und immer mehr Programme unterstützen XML im-/exports. In der aktuellen Version von CoMICS verwenden wir die freie Bibliothek *XStream*, welche es erlaubt, XML Files einzulesen bzw Java-Objekte in XML-Dateien abzulegen. Hier die Wichtigsten Eigenschaften von *XStream* im Überblick:

- Sehr einfache Handhabung.
- Das Objektmapping wird von XStream selbst übernommen.
- Sehr performantes Laufzeitverhalten.
- Erzeugung von sauberem XML, welcher automatisch durch Einrückung strukturiert ausgegeben wird.
- Mit einer integrierten Fehlerbenachrichtigung lassen sich Inkonsistenten in den Eingabedateien sehr schnell beheben.

Die Bibliothen, Tutorials, Programm API sowie weitere Informationen zum Programm können auf der Entwicklerseite unter `http://xstream.codehaus.org` heruntergeladen werden.

3.10 Zusammenfassung

Die in diesem Kapitel behandelten technischen Grundlagen wurden zur Umsetzung von CoMICS II ausgiebig genutzt. Besonders hervorzuheben ist *Java*, die sowohl als Modellierungssprache als auch als Entwicklungssprache Verwendung fand. Das Erscheinungsbild der Modellierungsumgebung wurde in erster Linie durch *SWT* und *JFace* geprägt, wobei die Statusanzeige der eigentlichen Simulation durch *SWING* realisiert wurde. Ein weiteres sehr wichtiges Framework ist *RCP*, welches die Umsetzung der Modellierungumgebung ermöglichte. Die eigentliche Simulation hält ihre Daten in einer relationalen Datenbank, dessen Integration ein *objekt-relationales Mapping* übernimmt. Dieses Mapping wird von *Hibernate* geleistet, das die Datenbank *PostgreSQL* anbindet, wobei auch andere relationale Datenbanken unterstützt werden können. Weiter wurde der *Mersenne Twister* als Zufallsgenerator für die Simulation verwendet. Dieser Generator kann leicht durch einen anderen ersetzt werden. Zum Einlesen von CSV-Dateien dient uns *OpenCSV*. Metainformationen zu einem Simulationsprojekt werden mit Hilfe von *XStream* in eine XML-Datei gespeichert bzw. aus einer solchen Datei gelesen.

Kapitel 4

Architektur- und Designentscheidungen

4.1 Anforderungen an die Software

4.1.1 Ziele und Vorgaben

Die Nachteile von CoMICS I, die im Abschnitt 2.4.7 ausführlich erläutert wurden, erschweren die Erstellung und die Simulation eines Modells. Diese Arbeit soll Lösung erarbeiten, die zur Verbesserung von CoMICS I beitragen. Nach der Analyse kristallisierten sich folgende Anforderungen als Sinnvoll heraus:

- Verbesserung der Skalierbarkeit
- Beschleunigung der Ausführung
- Bessere Unterstützung für die Modellierung einer Simulation in Java
- Wahrung der Konsistenz der Daten
- Flexibilisierung des Zugriffs auf statistische Daten
- Auswertbarkeit mit externen Werkzeugen

4.2 Überlegungen und Irrwege

An die Anforderungen dieser Arbeit knüpften einige Ideen und Überlegungen an, die jedoch nur Teilweise realisiert wurden.

4.2.1 Erweitern von CoMICS I

Die ersten Überlegungen drehten sich um die Erweiterung der Vorgängerversion. Jedoch konnten einige Probleme von CoMICS I ohne grundlegende Änderungen nicht behoben werden. Das größte Erschwernis war die Art und Weise, wie eine Simulation kompiliert und ausgeführt wird. Die vom Benutzer erstellten Klassen werden zur Laufzeit kompiliert und künstlich dem System-*ClassLoader* des Programms, womit die Klassen erstellt werden, hinzugefügt. Dieser Mechanismus ist sehr komplex und daher auch sehr Fehleranfällig, daher erschien uns eine Entkopplung von Modellierung-Umgebung und Simulation sinnvoll. Neben dieser nötigen Änderung, führte die Paradigma-Änderung, zu einer event-orientierten Mikrosimulation, zu einer kompletten Neuentwicklung.

4.2.2 Event orientiertes Vorgehen

Die Performance-Nachteile von CoMICS I sind hauptsächlich in der Mikrosimulation begründet. Hierbei wird, mit Hilfe einer doppelt verschachtelten Schleife, für jede Periode, jede Zeile des Mikrosensus simuliert, obwohl evtl. gar nicht alle Individuen für die Periode relevant waren. Zudem fallen alle Maßnahmen einer Periode auf einen Zeitpunkt, zum Beispiel heiraten und sterben aller Personen am Jahresende. Die Alternative dieses iterativen Modells ist ein Modell, das in Ereignissen (Events) aufgebaut ist. Hierbei werden alle Events zu jedem Individuum (z.B. Person) erfasst, die anschließend iterativ abgearbeitet werden. Der Vorteil dieses Vorgehens ist, dass nur die Entities einbezogen werden, für die eine Maßnahme durchgeführt wird. Zudem Fallen so nicht alle Ereignisse auf einen Zeitpunkt, sondern sind in der Periode verteilt. Dies sorgt für ein höheres Maß an Flexibilität und eine Annäherung an die Realität. Somit entschlossen wir uns, eine event-orientierte Mikrosimulation zu realisieren.

4.2.3 Die Simulation als Javaprojekt im RCP

Aufgrund der gewünschten Entkopplung der Modellierungsumgebung und der eigentlichen Simulation drängte sich die Überlegung auf, ein Simulationsprojekt als erweitertes Javaprojekt zu behandeln. Da diese Arbeit auf RCP Aufsetzt und dessen *Workspace* in *IProject* unterteilt wird, entschlossen wir diesen Mechanismus auch für diese Arbeit zu verwenden.

4.2.4 Externe Datenbank

Im Gegensatz zu CoMICS I baut diese Arbeit auf eine externe relationale Datenbank auf. Hierdurch wird der Aufwand für den Bezug und das Persistieren der Daten auf die Datenbank verschoben. Dies sorgt zum einen für einen Performance-Vorteil und zum anderen zu einer erweiterten Auswertbarkeit, da nun externe Programme zur Analyse der Daten verwendet werden können.

4.2.5 Eigenes Objekt-Relationales Mapping

Um Daten von einer relationalen Datenbank zu beziehen und anschließend objektorientiert zu verarbeiten wird ein Objekt-Relationales Mapping (ORM) benötigt. Zu Beginn dieser Arbeit wurde ein eigenes ORM spezifiziert, das jedoch keine verschachtelten Objekte unterstützte. Dieses Funktionalität sollte jedoch dem Benutzer zur Verfügung gestellt werden, sodass ein ORM-Framework namens Hibernate eingeführt wurde und machte die eigene Entwicklung überflüssig.

4.3 Verwendete Konzepte und Frameworks

In den letzten Jahren hat der Anteil an frei zugänglicher Software start zugenommen. Sei es bedingt durch die gesteigerte Unterstützung von Hardware-Herstellern für alternative Betriebssyteme wie Linux oder der Erfolg von Mozilla Firefox. Besonders für Java gilt, dass man das Rad nicht immer neu erfinden muss. Hierzu kann der Java-Entwickler auf eine Vielzahl von Open-Source[1]-Projekten zurückgreifen. (z.B. über das Portal `http://sourceforge.net`)

[1] Quelloffene Software, die unter einer OSI(Open Source Initiative) anerkannten Lizenz verfügbar ist.

4.3.1 Java Runtime Environment 6.0

Die aktuelle Java Version bietet viele Vorteile im Gegensatz zu der noch in CoMICS I verwendete Version 1.4.2. Die größten Veränderungen wurden sicherlich mit dem JRE 5.0 realisiert, die schon im Absatz 3.1.3 umfassend beschrieben wurden ist. In dieser Arbeit werden die neuen Möglichkeiten der Sprache eingesetzt, sodass keine Kompatibilität zu den älteren JREs mehr gegeben ist.

Ausgiebig verwendete Konzepte des JRE 6.0 sind die Generics, die For-Each-Schleife, die Annotationen, sowie die variablen Parameterlisten. Die Java Annotationen werden wegen dem Einsatz von Hibernate verwendet und dort auch intensiv verwendet. Die anderen aufgeführten Konzepten dienen:

- der Verbesserung der Typsicherheit in der Anwendung,
- der übersichtlichen Gestaltung des Quellcodes mit daraus folgender Erhöhung der Wartbarkeit / Erweiterbarkeit und
- der Optimierung des Komforts für den Nutzer.

All diese Punkte erhöhen die Qualität des Produkts.

4.3.2 Eclipse Rich-Client-Platform

Die Eclipse wurde schon für die Entwicklung von CoMICS I ausgiebig verwendet. Zum einen als Entwicklungsumgebung, zum anderen die Eclipse Bestandteile *SWT* und *JFace* zur Interaktion mit dem Benutzer. Mit CoMICS II wurde die Ausnutzung von Eclipse Funktionalität weiter ausgebaut. So basiert das neue Mikrosimulationssystem auf die Rich-Client Plattform der Eclipse. Wie in der Passage 3.3 ersichtlich, bietet diese Plattform einige Funktionalität, die für eine eigene und von Grund auf neue Entwicklung von großen Nutzen ist. So können eigene Module durch ein gut Beschriebene Plugin-Technik eingebunden werden und andere Plugins erweitern. Hierdurch kann eigene Plugin auf ein gewaltiges Arsenal von Funktionalitäten zurückgreifen. Ein Beispiel dafür ist das JDT, eine komplett integrierte Java-Entwicklungsumgebung. Diese Plugin bietet viele Möglichkeiten, die bei der Entwicklung eines Java-Mikrosimulationsmodell sehr behilflich sein können (z.B. Codevervollständigung). Zudem liefert das RCP eine Laufzeitumgebung, die zusammen mit den, für das eigene Produkt benötigt Plugins, ein komplette Anwendung darstellt. Eine solche Anwendung kann mit Hilfe der Eclipse zusammengestellt werden und ist schließlich auf der Zielplattform umgehend ausführbar.

4.3.3 Relational Datenbank und Hibernate

CoMICS II setzt auf eine relationales Datenbanksystem auf. Durch diese Umstellung konnte die Performance, sowie die Bewahrung der Konsistenz der Daten optimiert werden. Da das DBS nicht mehr Teil von CoMICS ist, können unterschiedliche DBMS (Database Management System) eingesetzt werden. Als Standard wird das objektrelationale DBMS PostgreSQL angebunden. Einzige Bedingung an die Datenbank ist die Einhaltung des Sprachstandards SQL-92, wodurch auch der Einsatz anderer DBMS ermöglicht wird.

Um die Verbindung zur Datenbank möglichst dynamisch zu halten, jedoch dem Nutzer auch ein hohes Maß an Komfort und Performance bieten zu können, setzt CoMICS II auf Hibernate (Hibernate wurde in Abschnitt 3.6.1 vorgestellt) auf. Dieses Framework ermöglicht ein komplett objektorientierten Umgang mit persistenten Daten. Die Abbildung der Daten in das relationale DBS geschieht hierbei automatisch und völlig transparent.

4.4 Zusammenfassung

Nach unsere Machbarkeitsstudie der einzelnen Technologien im Bezug auf die Anforderungen an diese Arbeit, stellten sich heraus, dass wir CoMICS I nicht erweitern, sondern eine gänzlich neue Anwendung erstellen mussten. Diese Überlegung führte uns zu RCP, dessen Vorteile und Funktionsumfang sehr gut zu den Anforderungen von CoMICS II passten. Somit wurde das neue event-orientierte Mikrosimulationssystem als RCP-Plugin konzipiert und implementiert. Zudem setzt das System auf eine relationale Datenbank auf, die über Hibernate, ein Framework für das objekt-relationales Mapping, angebunden wird.

Kapitel 5

Umsetzung und Implementation

5.1 Die Projekteinteilung

Um einen hohen Grad an Wartbarkeit, sowie die Vermeidung von dupliziertem Quellcode zu erreichen wird CoMICS II in unterschiedliche Projekte eingeteilt. Generell gibt es zwei Ebenen dieser Arbeit. Zum einen die Modellierungsumgebung, die es ermöglicht eine Simulation zu erstellen und zu bearbeiten. Zum anderen die eigentliche Simulation, die in einer eigenen VM abläuft, und somit komplett unabhängig von der eigentlichen Modellierungs-Anwendung ausgeführt werden kann.

5.1.1 Das Projekt *de.unikoblenz.comicsii.simbase*

Die zwei Ablauf-Szenarien verwenden teilweise die selben Komponenten, die im Projekt *Simbase* zusammengefasst werden. Hierzu zählen u.a. die beschreibenden Klassen im Package *com.pecasim.model*, die Basisklassen der Simulation und die Datenbank-Anbindung. Da dieses Projekt sowohl als Bibliothek für die Simulationsprojekte, sowie als Teil der Modellierungumgebung dient, wurde es als RCP-Plugin spezifiziert.

5.1.2 Das Projekt *de.unikoblenz.comicsii.ui*

Die Modellierungumgebung erweitert die RCP-Plattform zuzüglich dem JDT. Alle grafischen Elemente, sowie die Erzeugung und Manipulation von Javaelemente wurde in dem Projekt *de.unikoblenz.comicsii.ui* zusammengefasst. Abhängig ist dieses

RCP-Plugin von den grafischen Plugins *org.eclipse.jdt.ui* und *org.eclipse.debug.ui*. Da die Modellierungsumgebung nicht ausschließlich Funktionalität der beiden Module erweitert, sondern JDT-Funktionalität unterdrücken muss, wurden beide Eclipse-Projekte an die Anforderung dieser Arbeit angepasst..

5.1.3 Das Simulationsprojekt

Das eigentliche Simulationsprojekt wird von der Modellierungsoberfläche als *Java-Projekt* erstellt und bearbeitet.

5.2 Das Modell der Simulation

Da CoMICS ein Tool zur Erstellung von Simulationen ist, stellt sich die Modellierung recht komplex dar. Das Modell der Simulation muss in der Lage sein, alle möglichen Varianten von Simulationen abzubilden. Ein statisches Modell reicht für diese Belange leider nicht aus. Das im Folgenden beschriebene Modell betrachtet die Simulation von einer Meta-Ebene aus. Die für die Modellierung angefertigten Klassen sind im Basisprojekt unter dem Package *com.pecasim.model* zu finden. Die Namen aller Modellierungsklassen enden mit dem Postfix **Descriptor**. Somit ist bereits am Namen erkennbar, dass es sich in der Klasse um Metainformationen und nicht um die Simulationsklassen direkt handelt. Der Zusammenhang der einzelnen beschreibenden Klassen wird in der Abbildung 5.1 deutlich.

5.2.1 Der *ProjectDescriptor*

Auf der obersten Ebene beinhaltet der *ProjectDescriptor* alle Daten und Fakten, die ein konkretes Simulationsprojekt ausmachen. Er verwaltet den Simulationsnamen, einen *DatabaseDescriptor* und einen *SimulationsDescriptor*. Die Einstellungen des kompletten Projektes können über die Methode *saveToXML* in eine XML Datei exportiert werden. Beim Simulationsstart werden die Daten aus der XML Datei, die in **PROJECT_DESCR_FILENAME** angegeben ist, eingelesen.

5.2.2 Der *DatabaseDescriptor*

Im *DatabaseDescriptor* werden alle datenbankspezifischen Informationen abgelegt. Im Attribut **dbDriver** ist der aktuell verwendete Java Datenbanktreiber hinterlegt.

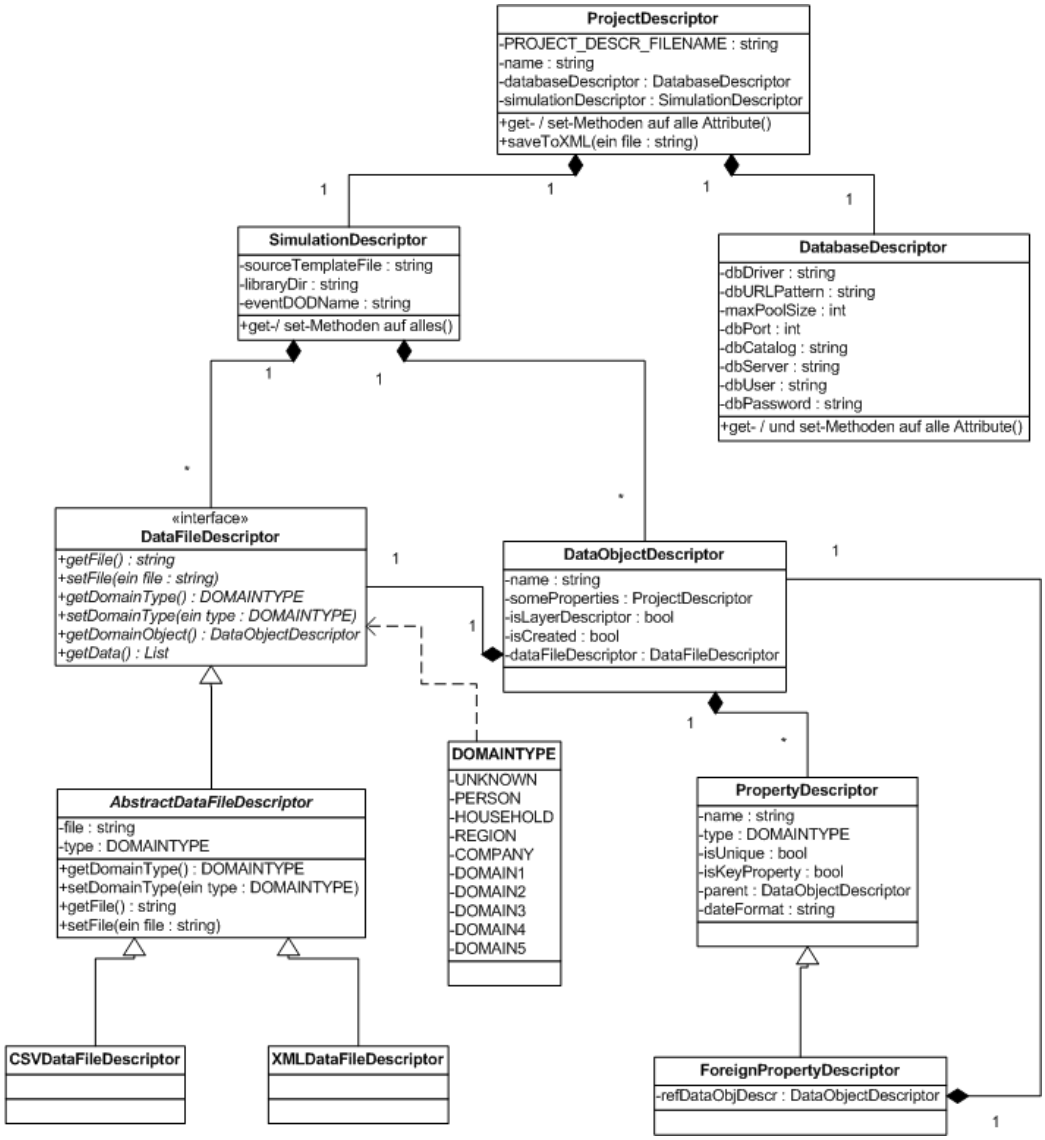


Abbildung 5.1: Die Modellklassen im Überblick

Standardmäßig wird hier der Treiber für die *PostgreSQL*-Datenbank verwendet. Sollte ein anderer verwendet werden, muss der Wert entsprechend geändert und die zugehörigen Bibliotheken mit eingebunden werden. Die Anzahl der gleichzeitig möglichen Datenbanksessions wird in **maxPoolSize** angegeben. Der ConnectionString zur Datenbank wird in der Variablen **dbURLPattern** gespeichert. Er setzt sich aus folgenden Attributen zusammen:

- **dbServer**
  gibt den Netzwerknamen oder die IP des Rechners an, auf dem die *PostgreSQL* Datenbank installiert ist. Im Normalfall ist die Datenbank auf dem gleichen System installiert und es wird per Loopback-Adresse *127.0.0.1* oder *localhost* darauf zugegriffen.

- **dbPort**
  gibt den entsprechenden Port auf dem Datenbankserver an. Voreingestellt ist hier stets der *PostgreSQL*-Standardport.

- **dbCatalog**
  bezeichnet den Namen der Datenbankinstanz auf dem Datenbankserver. Hier ist *comics* voreingestellt.

- **dbUser**
  gibt den autorisierten Datenbankbenutzer an. Er benötigt zwingend Lese- und Schreibrechte auf der verwendeten Datenbankinstanz.

- **dbPassword**
  ist das dem dbUser zugeordnete Datenbankpasswort. Hinweis: In der aktuellen Version wird das Passwort innerhalb von CoMICS nicht explizit verschlüsselt.

5.2.3 Der *SimulationDescriptor*

Der *SimulationDescriptor* verwaltet alle Daten die im logischen Zusammenhang mit der Simulation direkt stehen. Er enthält jeweils eine Liste zur Ablage von *DataObjectDescriptor* und eine für *DataFileDescriptor*-Instanzen. Jeder *DataObjectDescriptor* modelliert genau eine Schicht der realen Gesellschaft. In den *DataFileDescriptor*-Instanzen werden die zu den jeweiligen Schichten zugehörigen Datenquellen abgelegt. Zur Konfiguration wird noch eine Templatedatei in der Variablen

sourceTemplateFile angegeben, welche als Vorlage für die spätere Simulation dienen soll.

5.2.4 Der *DataObjectDescriptor*

Der *DataObjectDescriptor* repräsentiert, wie in den Vorigen Abschnitten bereits erwähnt, jeweils eine Schicht der betrachteten Gesellschaft. Beispielsweise könnten Personen, Haushalte oder Regionen als Gesellschaftsschichten angesehen werden. Diese Real-Objekte werden in CoMICS als *DataObjectDescriptoren* modelliert werden. Das Modell besitzt im Wesentlichen die im folgenden aufgeführten Attribute.

- **name**
  Der eindeutige Name der Schicht.
- **someProperties**
  Eine Liste aus Attributen, die für diese Schicht relevant sind.
- **isLayerDescriptor**
  Gibt an, ob es sich bei dem *DataObjectDescriptor* um eine Schichtenrepräsentation handelt. Der *DataObjectDescriptor* wird in Spezialfällen auch schon mal für andere Objekte eingesetzt. Ein Beispiel für einen solchen Spezialfall wäre die Erzeugung der Event-Tabelle.
- **isCreated**
  In dem Attribut *isCreated* merkt sich das System, ob die Schicht bereits komplett - also auch in der Datenbank - angelegt worden ist.
- **dataFileDescriptor**
  Über den *DataFileDescriptor* bekommt die Schicht die Datenquelle zugeordnet.

5.2.5 Der *PropertyDescriptor*

Der PropertyDescriptor beschreibt ein einzelnes Attribut innerhalb einer Schicht. Das könnte in der Personenschicht ein Attribut wie *name*, *alter* oder *job* sein. Die folgenden Informationen werden in der Klasse abgelegt.

- **name**
  Der eindeutige Name des Attributes.

- **DOMAINTYPE**
  Hier wird angegeben, um welche Schichtebene es sich genau handelt.

- **isUnique**
  Gibt an, ob die Werte des Attributes mehrfach vorkommen dürfen. *IsUnique* wird verwendet um initial Datenbank constraints festlegen zu können.

- **isKeyProperty**
  Hier wird die PrimaryKey-Eigenschaft der Schicht festgelegt. Üblicherweise wird sie auf ID-Felder angewendet. Sie wird immer in Kombination mit *isUnique* gesetzt.

- **parent**
  Jeder PropertyDescriptor verwendet einen *DataObjectDescriptor* um sich seine Elternschicht zu merken. Jedes Attribut kann über diese Verknüpfung auf alle anderen Attribute der Schicht zugreifen.

- **dateFormat**
  Sollte es sich bei dem Attribut um einen Datumswert handeln, kann hier das genaue Format des Datumsstrings beim Einlesen abgelegt werden. Ein Datumsstring könnte als Stringrepräsentation beispielsweise folgendermaßen aufgebaut sein: *yyyy-MM-dd HH:mm:ss*

5.2.6 Der *ForeignPropertyDescriptor*

Der *ForeignPropertyDescriptor* beschreibt ein ForeignKeyAttribut, welches ein Zeiger auf eine andere Schicht repräsentiert. Beispielsweise wäre das Attribut *haushaltID* der Personschicht mit der Haushaltschicht verknüpft. Die Klasse ist von *PropertyDescriptor* abgeleitet und stellt somit alle Eigenschaften und Methoden eines *PropertyDescriptors* bereit. Zusätzlich besitzt er einen *DataObjectDescriptor*, über den er sich die Zielschicht merken kann. Diese Klasse ist ebenfalls sehr wichtig für die automatische Generierung von ForeignKey-Constraints in der Datenbank.

5.2.7 Der *DataFileDescriptor*

Das Interface *DataFileDescriptor* modelliert auf oberster Ebene eine Datei mit Quelldaten. Im eigentlichen Sinne werden hiermit die Daten der einzelnen Schichten eingelesen. Sie sind der ETL-Baustein (Extract Transform Load) von CoMICS. Sie bieten die Möglichkeit Daten unterschiedlichster Art zu vereinen und in der CoMICS-Datenbank abzulegen. Fehlerhafte Datensätze werden nicht ins System übertragen, sondern in ein Fehlerlog geschrieben. Die im folgenden aufgeführten Methoden werden bislang vom System unterstützt. Mit *get-* und *setFile* kann die zugrundeliegende Basisdatei abgefragt, bzw. neu gesetzt werden. Mit *get-* und *set-DomainType*, kann die verwendete Schichtenebene abgefragt, bzw. geändert werden. Die Methode *getDomainObject* liest die Metadaten aus der Basisdatei aus und liefert direkt den *DataObjectDescriptor* der Schicht zurück. Die eigentlichen Daten der Schicht können mit der Methode *getData* abgefragt werden. Die Daten werden als Liste von String-Arrays übergeben.

5.2.7.1 *AbstractDataFileDescriptor*

Die Klasse *AbstractDataFileDescriptor* bietet eine Vorlage für alle späteren *DataFileDescriptor*-Objekte, in der alle abstrakten Attribute bereits angelegt werden. Hierzu gehört die Basisdatei und der DomainType. Die Basisdatei wird im Attribut *file* und der DomainType in *type* abgelegt. Die entsprechenden Zugriffsmethoden, die im Interface verlangt werden, sind ebenfalls auf dieser Ebene schon implementiert.

5.2.7.2 Die *DataFileDescriptor*-Klassen

Es werden in der aktuellen Version von CoMICS nur zwei unterschiedliche *DataFileDescriptoren* verwendet. Die einfachste Variante ist der *CSVDataFileDescriptor*. Er verwendet die Java-Bibliothek **OpenCSV** um die Eingabe-CSV-Dateien einzulesen. In der ersten CoMICS Version war Ein- und Ausgabe ausschließlich über CSV-Dateien möglich. In der aktuellen Version gibt es noch den *XMLDataFileDescriptor*. Er unterstützt das Einlesen von Daten, die als XML-Datei abgelegt sind. Die XML-Funktionalität innerhalb der Klasse wird durch die JavaBibliothek **XStream** bereitgestellt. Alle noch fehlenden Methoden des *DataFileDescriptor*-Interfaces werden hier in diesen beiden Klassen implementiert.
Sollen weitere Formate, wie z.B. Excel-Tabellen vom System unterstützt werden,

muss einfach die entsprechende *DataFileDesctiptor* Datei implementiert und eingebunden werden.

5.3 Die Anbindung der Datenbank

Die Datenbank wird mit Hilfe von Hibernate von der restlichen Anwendung gekapselt, um objektorientiert und performant mit den persistenten Daten arbeiten zu können. Hierzu dient die Klasse *HibernateDatabaseManager* im Projekt *Simbase*. Dieser Manager implementiert das Interface *DatabaseManager*, das ausschließlich für den system-internen Gebrauch gedacht ist. Dieses Interface definiert Methoden zur Verwaltung der Datenbank-Verbindung, Bereitstellung der Events, sowie der Transaktionsverwaltung. Zu den Methoden zu Verwaltung der Verbindung zählt:

- *initialise()*, etabliert die Verbindung zur Datenbank
- *getConnection()*, liefert die aktuelle Datenbank-Verbindung
- *getConfigFile()*, liefert den Dateinamen der Konfigurationsdatei
- *close()*, beendet die Verbindung zur Datenbank

Für den Datenzugriff speziell für Event-Entities werden drei Methoden zur Verfügung gestellt. Die Methoden *getEventClass()* und *popNextEvent()* liefern jeweils ein Event bzw. genau das nächste Event aus der Datenbank. Die Methode *loadCluster()* ermittelt die nächste Menge von Elementen, wobei die Größe der Menge einstellbar ist. Die Transaktionsverwaltung wird über die zwei Methoden *commit()*, bestätigt eine Transaktion, und *rollback()*, verwirft einen Vorgang, realisiert.

Um den Nutzer nur auf die Optionen zu beschränken, die für sein Simulationsmodell relevant sind und somit Fehlbedienung durch den Aufruf falscher Methoden zu vermindern, wurde das Interface *Database* definiert. Diese Schnittstelle beinhaltet die wichtigsten Zugriffsmethoden, wie *get()* um ein Objekt bzw. eine Objekt-Liste aus der Datenbank zu beziehen und *store()* um eine Instanz zu persistieren. Um Daten mit *store* zu verändern, müssen die entsprechenden Daten aus der Datenbank geladen und ein Objekt daraus erzeugt werden. Danach kann diese Instanz geändert und zurückgeschrieben werden. Dieser Aufwand kann durch *executeUpdate()* erheblich reduziert werden, da mit Hilfer dieser Methode die Daten

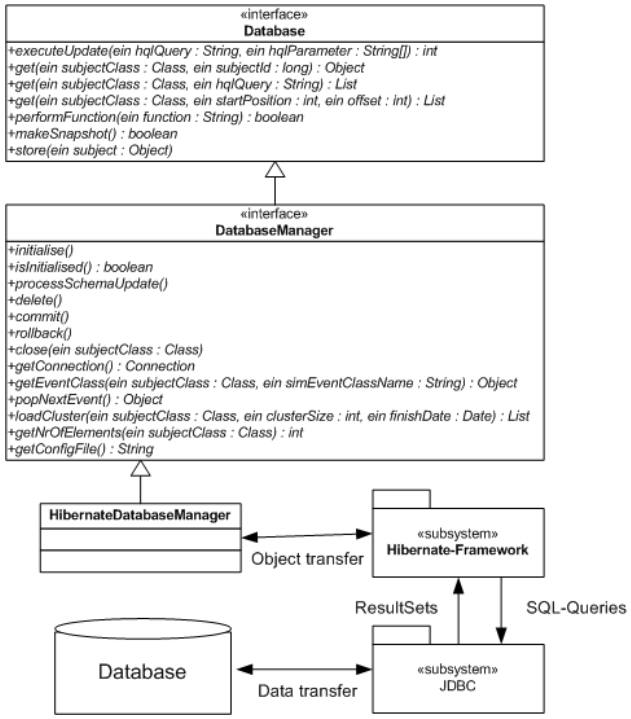


Abbildung 5.2: Die Datenbankanbindung mit Hibernate

direkt in der Datenbank manipuliert werden können, ohne ein Java-Objekt erzeugen zu müssen. Zusätzlich kann mit *performFunction()* eine beliebige Datenbank-Funktion aufgerufen werden. Standardmäßig werden drei Funktionen in der Datenbank bereitgestellt. Mit *clear* werden die Ergebnisse des letzten Simulationsdurchlaufs gelöscht, *makesnapshot* erstellt Momentaufnahmen der Simulations-Subjekte und *dropsnapshots* löscht alle Momentaufnahmen. Da das Erfassen von verschiedenen Zwischenständen essentiell ist, wird dem mit der Methode *makeSnapshot()* im Interface *Database* Rechnung getragen.

Relevant für eine Datenbank-Verbindung mit Hibernate ist die Klasse *Configuration*. In dieser Arbeit verwenden wir jedoch Hibernate Annotations, sodass wir die Unterklasse *AnnotationConfiguration* verwenden müssen. Durch den Aufruf der Methode *configure()* und der Übergabe des Konfigurationsdatei als Parameter, wird eine initialisierte Instanz der Konfiguration geliefert. Mit Hilfe diesem Objekt gelangen wir an die *SessionFactory* und damit auch an eine konkrete Instanz von *Session*. Diese *Session* stellt die Schnittstelle zum objekt-relationalen Mapping und so zur Datenbank dar. Diese Prozedur wird in der Methode *initialise()* der

Klasse *HibernateDatabaseManager* ausgeführt, die in Abbildung 5.3 dargestellt ist. Zusätzlich wird das Datenbankschema geprüft und wenn nötig auf den aktuellen

```
public void initialise() throws DatabaseException {
    if (!isInitialised) {
        try {
            File file = new File(configFile);
            config = (AnnotationConfiguration) new AnnotationConfiguration()
                    .configure(file);
            session = config.buildSessionFactory().openSession();
            connection = session.connection();
            processSchemaUpdate();
            isInitialised = true;
        } catch (Exception e) {
            throw new DatabaseException(this.getClass(), "initialise",
                    MSGCODE.E201, e);
        }
    }
}
```

Abbildung 5.3: Initialisierung des *HibernateDatabaseManager*s

Stand gebracht. Hierfür stellt Hibernate die Klasse *SchemaUpdate* zur Verfügung, die in der Methode *processSchemaUpdate()* genutzt wird. Im Fall eines Fehlers wird eine *DatabaseException* ausgelöst.

Das Speichern oder Löschen von Objekten geschieht im Rahmen einer Transaktion, d.h. die getätigte Aktion wird erst endgültig in der Datenbank persistiert, wenn ein Commit ausgelöst wurde. Daher wird mit *session.getTransaction()* zuerst die aktuelle Transaktion ermittelt und falls keine Aktive vorhanden ist, eine neue Transaktion begonnen. Danach kann die eigentliche Operation ausgeführt werden. Dieser Prozess ist exemplarisch in der Abbildung 5.4 ersichtlich und ist auf *delete()* übertragbar. Die gerade aktive Transaktion kann anschließend mit *tran-*

```
public void store(Object aObject) {
    Transaction tx = session.getTransaction();
    if (!tx.isActive()) {
        tx = session.beginTransaction();
    }
    session.saveOrUpdate(aObject);
}
```

Abbildung 5.4: Das Persistieren eines Hibernate-Objekts

saction.commit() oder *transaction.rollback()* bestätigt bzw. verworfen werden.

Die Abfrage von Daten ist ein weiterer wichtiger Punkt des *HibernateDatabaseManager*. Hierzu implementiert diese Klasse, die von den Interfaces definierten Methoden. Wir verwenden zwei unterschiedliche Varianten Objekte von der *Session* abzufragen. Die Methode *load()* bezieht die Objekte über einen eindeutigen Schlüssel und liefert genau ein Objekt zurück. Die zweite Variante funktioniert über

die Angabe eines *Criteria*, das die gewünschten Eigenschaften der Daten beschreibt und über die Methode *session.createCriteria()* bezogen werden kann. Die zweite Variante wird am Beispiel einer *get*-Methode in Abbildung 5.5 vorgestellt.

```
public <T> List<T> get(Class<? extends T> aClass, int aBegin, int aLength) {
    Criteria c = session.createCriteria(aClass);
    c.setFirstResult(aBegin);
    c.setMaxResults(aLength);
    return c.list();
}
```

Abbildung 5.5: Beziehen von Hibernate-Objekten per *Criteria*

Die einzige Funktionalität, die außerhalb von Hibernate genutzt wird, ist die Ausführung von Datenbank-Funktionen. Die Methode *performFunction()* nutzt hierzu die aktuelle Instanz von *Connection*. Mit der Methode *prepareCall()* wird ein *Statement* erzeugt, das mit *execute()* auf der Datenbank angewendet werden kann. Hierbei ist zu beachten, dass eine Funktion lt. JDBC-Spezifikation mit „call„ beginnen und mit „“ enden muss. Da die Ausführung einer Datenbank-Funktion transaktional abläuft, muss vor dem Start der Funktion die aktuelle Transaktion bestätigt werden. Weitere Einzelheiten können der Abbildung 5.6 entnommen werden.

```
public boolean performFunction(String aFunction) {
    Connection connection = getConnection();
    boolean result = false;
    CallableStatement statement = null;
    try {
        commit();
        statement = connection.prepareCall("{call " + aFunction + "}");
        result = statement.execute();
        if (result) {
            connection.commit();
        }
    } catch (Exception e) {
        try {
            connection.rollback();
        } catch (SQLException e1) {
        }
        e.printStackTrace();
        result = false;
    } finally {
        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {
            }
        }
    }
    return result;
}
```

Abbildung 5.6: Ausführung einer Datenbank-Funktion

Der *DatabaseManager* wird zu Beginn des Programms von dem *DatabaseManagerFactory* bezogen und der konkreten Simulation übergeben. Jedoch liefert die

Simulation ausschließlich die *DatabaseManager*-Instanz als *Database*, die jedoch durch einen Cast entsprechend überführt werden kann. (z.B. *DatabaseManager database = (DatabaseManager) simulation.getDatabase()*)

5.4 Das Datenbankschema

Das relationale Datenbankschema gliedert sich in zwei Teile. Zum Einen die Systemtabellen, die von jedem Simulationsmodell verwendet wird und zum Anderen die benutzerdefinierten Tabellen, die für jedes Simulationsszenario bzw. für jede Datenbasis generiert werden muss. Die Namen der Benutzer-Tabellen weisen immer das Präfix 'lyr_' auf. In Abbildung 5.7 sind die Tabellen und ihre Beziehungen untereinander ersichtlich. Die Tabelle *Calendar* beinhaltet Zeitpunkte, den aktuel-

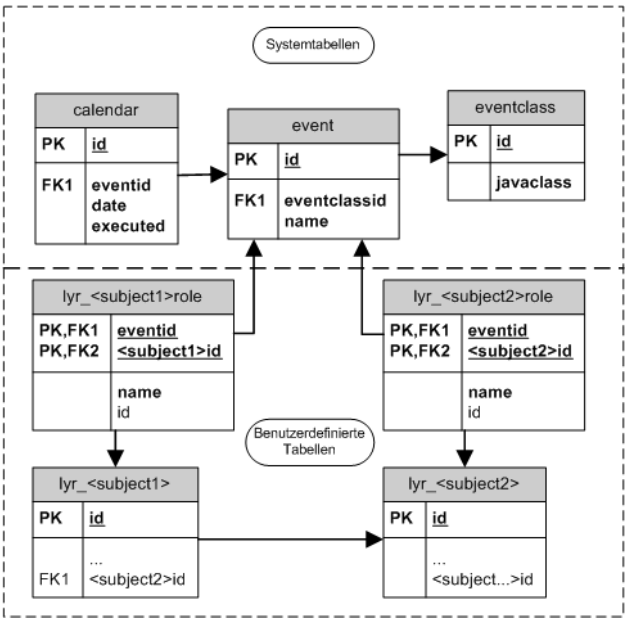


Abbildung 5.7: Das relationale Datenbankschema

len Status (ausgeführt/nicht ausgeführt) und einen Verweis auf die Tabelle *Event*. Die Eindeutigkeit der *Calendar*-Einträge wird über die Spalte *id* sichergestellt. Das *Event* verweist auf die Tabelle *Eventclass*, die den Klassen-Namen des konkreten *SimEvent*s beinhalten. Die Verbindung von *Event*s und den jeweiligen Subjekt wird über eine Rollen-Tabelle hergestellt. Ein Eintrag dieser Tabelle verknüpft ein *Event* mit den konkreten Subjekten, und ordnet ihnen eine Rolle zu. Zum Beispiel neh-

men wir ein *Event* 'Hochzeit', die zwei Personen mit einander verbindet. Die eine Person hat die Rolle Bräutigam, die andere Person die Rolle Braut. Hier hießen die Subjekt-Tabellen also *lyr_personrole* und *lyr_person*. Falls es mehrere Subjekt-Tabellen gibt, müssen diese miteinander in Beziehung stehen. In unserem Beispiel würden mehrere Personen in einem Haushalt zusammengefasst. Dem entsprechend würde die Tabelle *lyr_person* einen Fremdschlüssel auf die Tabelle *lyr_haushalt* beinhalten. Nach dem gleichen Konzept könnten Haushalte zu Regionen und Regionen zu Ländern usw. gruppiert werden.

5.5 Aufbau einer Simulation

In den einleitenden Kapiteln wurde bereits das schlechte Laufzeitverhalten der ersten CoMICS Version erläutert. Um diesen Punkt zu verbessern sollte die alte Konzeption auf eventorientierte Mikrosimulation umgestellt werden. Wichtigste Elemente dieses Konzeptes sind zunächst die Gesellschaft aus **Individuen**, frei definierbare **Events**, ein **Eventkalender** zur Verwaltung und natürlich ein einheitliches **Zeitmanagement**.

5.5.1 Die Individuen

Wie bereits im Kapitel 2.4 beschrieben, modellieren Mikrosimulationen stets eine repräsentative Liste an Individuen, welche sich dann durch die eigentlichen Simulationsläufe weiterentwickeln. In CoMICS I wurden die einzelnen Personen in einer simplen Personenliste gehalten. Da diese nicht immer komplett in den Arbeitsspeicher aktueller Standardrechner passte, wurde sie um ein aufwendiges Pagingsystem erweitert. Zudem wurden disjunkte Gruppierungen über ein Schichtensystem bereitgestellt, welches bereits im einleitenden Kapitel 2.4.3 vorgestellt worden ist.
Die Organisation der Individuen wurde in der neuen Version komplett in die Datenbank verlagert. Die Umsetzung von Objekten in relationale Tabellenzeilen wird komplett von Hibernate bereitgestellt. Hibernate verwendet dazu Domainklassen, um zwischen Tabellenzeilen der relationalen Datenbank und Java-Objekten zu Mappen. Die genaue Umsetzungskonfiguration wird innerhalb der Domainklassen mittels Annotationen an die Attribute der Klassen vermerkt. Die Objekte dieser Klassen sind dann die Simulaitons-Individuen bzw -Schichten. Da die Individuen weitgehend in jeder Simulation unterschiedlich sind, werden die entsprechenden

Klassen in das konkrete Simulationsprojekt hineingeneriert. Sie werden im Package *com.pecasim.custom.domain* abgelegt. In Abbildung 5.8 ist beispielsweise eine Domain-Klasse dargestellt die ein Personen-Objekt modelliert. Die Personen sind in diesem konkreten Anwendungsfall auf die Attribute **Name**, **Haushalt**, **Alter** und **Geschlecht** beschränkt.

```
package com.pecasim.custom.domain;
import javax.persistence.*;

@Entity
@org.hibernate.annotations.GenericGenerator(name = "SEQ_PERSON", strategy = "increment")
@Table(name = "LYR_PERSON")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator="SEQ_PERSON")
    @Column(name = "id")
    public long id;

    @Column(name = "name")
    public String name;

    @ManyToOne(cascade = CascadeType.ALL, optional = false) @JoinColumn(name = "haushaltid")
    public Haushalt haushalt;

    @Column(name = "age")
    public long age;

    @Column(name = "sex")
    public long sex;

    Person(){}
    public Person(String aName, Haushalt aHaushalt){
        name = aName;
        haushalt = aHaushalt;
    }
//  ...
//  Get- und Setmethoden zum sicheren Zugriff auf alle Member der Klasse
//  ...
}
```

Abbildung 5.8: Personen-Domain-Klasse

5.5.2 Die Klasse *SimDate*

In der eventorientierten Mikrosimulation sind Zeitpunkte ein sehr wichtiger Aspekt. Sie beginnen an einem Zeitpunkt, enden an einem Zeitpunkt und verwalten im Kalender unzählige Zeitpunkte, die es abzuarbeiten gilt. Wir haben diese Zeitpunkte als separate Klasse namens *SimDate* modelliert, um einen für die Simulation komfortablen Zugriff zu gewährleisten. Die Klasse ist für jede spätere Simulation gleich befindet sich deshalb im Package *com.pecasim.base.simulation* des Basisprojektes. Um Zeitpunkte vergleichbar und somit auch sortierbar zu machen, implementiert *SimDate* das **Comparable** Interface. Über Enumerations kann auf bequem auf

Wochentage und Monate zugegriffen werden. Zudem stellt die Klasse diverse Methoden zur Zeitspannenberechnung bereit, welche innerhalb der Simulation üblicherweise sehr häufig in Anspruch genommen werden.

5.5.3 Die *SimEvent*s

Die einzelnen Events werden im Basisprojekt *de.unikoblenz.comicsii.simbase* durch das Interface *SimEvent* und die abstrakte Klasse *AbstractSimEvent* strukturell vorgegeben. Sie befinden sich im Package *com.pecasim.base.events*. Diese Elemente sind sehr allgemein gehalten und somit Vorlage für alle *SimEvents*, die in späteren Simulationen von Ihnen abgeleitet werden. Im eigentlichen Simulations-Projekt befinden sich dann die einzelnen Events, die genau zu dieser konkreten Simulation gehören.

5.5.3.1 Eventklassen

Im Erstellungsprozess der Simulation werden vom Simulationsentwickler ausschließlich Eventklassen definiert. Diese sind üblicherweise Javaklassen, welche von *AbstractSimEvent* abgeleitet sind. Sie werden bei der Entwicklung oft als Events bezeichnet, deshalb hier die explizite Unterscheidung. Die Eventklassen sind nicht mehr allgemein gültig, sondern sehr speziell auf die konkrete Simulation ausgelegt. Innerhalb einer Simulation können beliebig viele Eventklassen angelegt werden. Sie werden vom Programm gesammelt im Javapackage *com.pecasim.custom.events* abgelegt. Als besondere Eventklasse ist die wichtige InitEvent-Klasse zu nennen. Sie ist dafür verantwortlich die einzelnen Personen vor dem eigentlichen Beginn der Simulation zu initialisieren. Beispiele für weitere Eventklassen sind Hochzeit, Geburt oder Tod.

5.5.3.2 Events

Bei Ablauf der Simulation werden aus den zuvor definierten Eventklassen konkrete Instanzen gebildet, die innerhalb der Simulation verarbeitet werden können. Diese Instanzen sind die eigentlichen Events der Simulation.

5.5.3.3 Rollen

Zu den Events können beliebige Individuen logisch verknüpft werden. Diese Zuordnung wird über Rollen realisiert. Ein Heirats-Event könnte beispielsweise die Rollen Braut und Bräutigam zugeordnet bekommen. Die Rollen müssen mit Individuen aus der Basisgesellschaft besetzt werden. Um einen komfortablen Zugriff auf die Rollen des Events zu erhalten werden sie in der späteren Eventklasse als Attribute geführt. Rollen werden den Events nicht zwingend vorgegeben, sind sie jedoch deklariert, müssen sie auch besetzt werden. In CoMICS integriert sind auch Analyse- bzw. Auswertungs-Events für die üblicherweise keine Rollen vorgesehen sind.

5.5.3.4 Aufbau

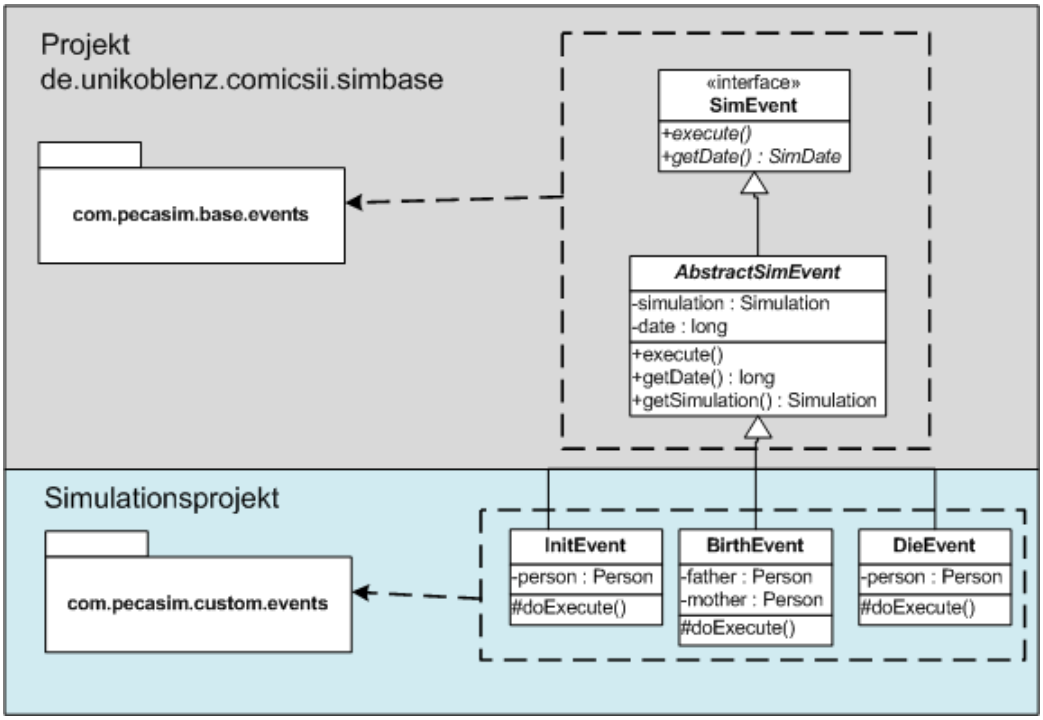


Abbildung 5.9: Die SimEvents

Die Klasse *AbstractSimEvent* implementiert das Interface *SimEvent* und stellt den später abgeleiteten Events den Ausführungszeitpunkt und die *execute*-Methode zur Verfügung. Jedes Event benötigt beides, um sich später in die Simulation einfügen zu können. Der Ausführungszeitpunkt wird als Klassenvariable namens **date**

vom Typ *SimDate* geführt. Die *execute*-Methode wird nach unten hin in *doExecute* weitergeleitet, um ggf. auf allen Ebenen noch benutzerdefinierten Code ausführen zu können. Die konkreten Events des Simulationsprojektes müssen mit einem *SimDate* als Ausführungszeitpunkt initialisiert werden und die entsprechende *doExecute*-Methode implementieren. Hierfür ist der spätere Simulationsentwickler selbst verantwortlich.

5.5.3.5 Die Domainklassen

Genau wie die Individuen kommen die Events auch in sehr großer Anzahl in der Simulation vor. Deshalb werden hier ebenfalls die Daten in die Datenbank ausgelagert und dynamisch nachgeladen. Es werden Hibernate Domainklassen verwendet um Event-Objekte in Tabellenzeilen der Datenbank zu mappen. Da sich die Klassen direkt auf die Datenbank beziehen, wurde aus Normalisierungsgründen die folgende Struktur gewählt.

- **DBCalendar**
  Auf oberster Ebene ist ein DomainEvent-Objekt immer von der Klasse *DBCalendar* abgeleitet. Es enthält im Wesentlichen den Ausführungszeitpunkt, einen Zeiger auf ein *DBEvent*-Objekt und ein Flag namens *executed*, welches angibt ob es schon ausgeführt worden ist oder nicht.

- **DBEvent**
  Die *DBEvents* halten alle weiteren Informationen, die für das Event entscheidend sind. Hier bekommen die Events ihren Namen und ihre Eventklasse zugeordnet. Des Weiteren werden in Listen die für das Event definierten Rollen geführt.

- **EventClass**
  Über die Objekte der Klasse *EventClass* bekommen die Events ihren vollständigen Java-Klassennamen zugeordnet. Beim Mappen der Domain-Objekte in *SimEvents* ist diese Information zwingend erforderlich.

- **Rollenzuordnung** Über die *RollenKlassen* bekommen die Events die mit ihnen verknüpften Personen zugeteilt. Für jede Schicht, die im System angelegt worden ist, wird eine DomainKlasse erzeugt. Im Beispiel sind Die Klassen *PersonRole* und *HaushaltsRole* vorhanden, welche auch um beliebig viele andere erweitert werden könnten.

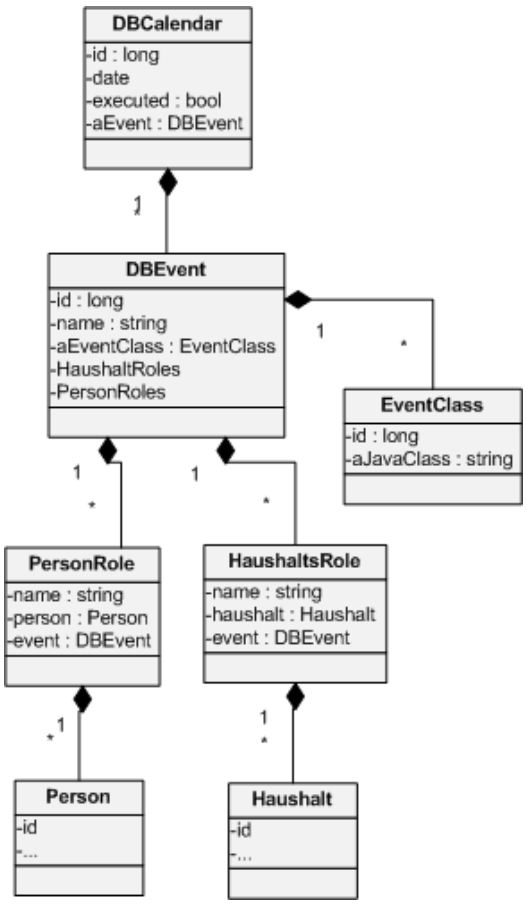


Abbildung 5.10: Die Domainklassen für Events

Die für die Events relevanten Domainklassen sind im Simulationprojekt unter dem Package *com.pecasim.domain* zu abgelegt. Sie werden beim Datenimport automatisch mit angelegt.

5.5.3.6 Die Datenbanktabellen

Hibernate nimmt die Struktur der Domainklassen und baut daraus ein relationales Datenbankschema auf. Die Gestaltung der Domainklassen wurde bereits derart gewählt, dass sich in der Datenbank die Tabellenstruktur aus Abbildung **??** ergibt.

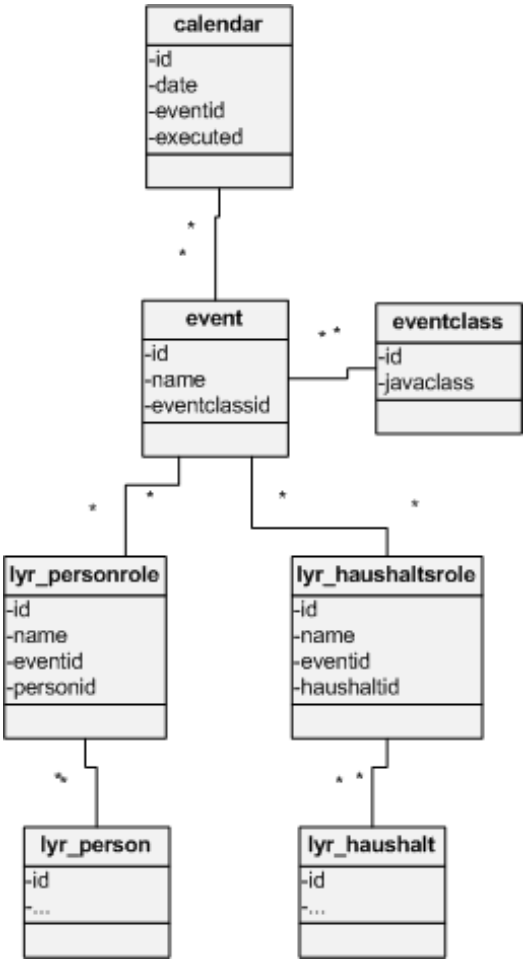


Abbildung 5.11: Datenbankschema der Events

Alle Elemente in der Datenbank sind analog zu den Domainklassen zu sehen und werden hier deshalb nicht noch einmal erläutert.

5.5.3.7 EventMapping

Events können in CoMICS drei Formen annehmen. Sie können Objekte der Klasse SimEvent, Domainobjekte der Klasse DBCalendar oder persistente Datenbankeinträge sein. Für den Simulationsentwickler sind ausschließlich Instanzen der Klasse SimEvent sichtbar, mit denen er seine Simulation komplett gestalten kann. Um die Events in die Datenbank auszulagern und somit den wertvollen Arbeitsspeicher des Simulationsrechners zu schonen, müssen diese SimEvent-Instanzen jedoch zunächst in die Domain-Objekte DBCalendar umgemappt werden. Diese Funktionalität wird im System vom Interface EventMapper zur Verfügung gestellt. Der *EventMapper*

```
package com.pecasim.base.events;

 * MicroSimulation
 * Interface EventMapper liefert die generellen Schnittstellen zum umwandeln von

public interface EventMapper {
    public Object mapSimEvent2DBCalendar(SimEvent se);

    public SimEvent mapDBCalendar2SimEvent(Object dbe);

}
```

Abbildung 5.12: Interface EventMapper

stellt für je eine Richtung eine Map-Methode zum Umwandeln der Objekte zur Verfügung. Er befindet sich im Basisprojekt im Package *com.pecasim.base.events*. Er wird im Simulationsprojekt von der Klasse *EventMapperImpl* aus dem Package *com.pecasim.events* implementiert.
Die komplette Datenbankverwaltung übernimmt das Interface *DatabaseManager*, welches im Package *com.pecasim.base.database* abgelegt ist. Der DatabaseManager stellt alle benötigten Methoden bereit um die Domain-Objekte in die Datenbank einzutragen, aus der Datenbank zu holen oder zu löschen. Er synchronisiert auch das Datenbankschema auf die jeweils vorkonfigurierten Domainklassen. Zudem arbeitet er komplett Transaktionsorientiert und muss alle Anfragen mit **commit** bestätigen oder mit **rollback** abbrechen.

Das komplette Zusammenspiel der betroffenen Klassen ist in Abbildung 5.13 zu sehen. Ausgehend von Event-Objekten der Simulation werden mit dem Event-Manager analoge Domainobjekte erzeugt, die dann persistent in der Datenbank abgelegt werden können. Werden die Objekte in der Simulation wieder benötigt,

werden sie auf die gleiche Art und Weise wieder aus der Datenbank beschafft.

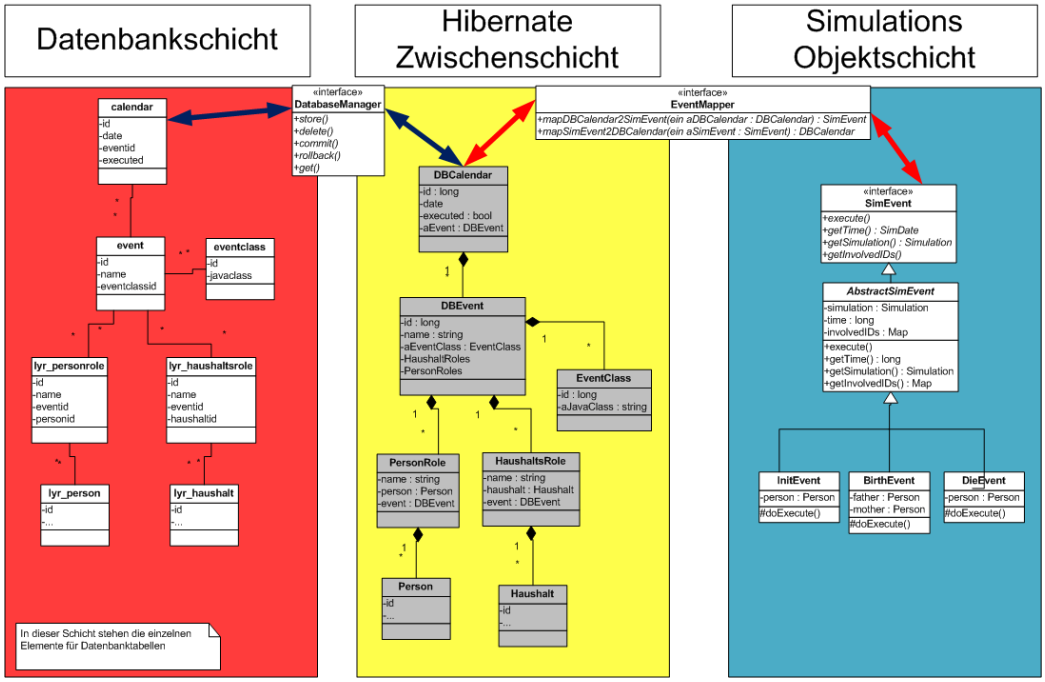


Abbildung 5.13: Umwandlung der Events

5.5.4 Der *Eventkalender*

Ein weiteres zentrales Element der Simulation ist der Eventkalender. Er wird durch die Klasse SimulationCalendarImpl bereitgestellt. Er verwaltet intern einen *Event-Manager* in den er SimEvents ablegen und wieder nachladen kann. Alle Events der Simulation werden hier zentral verwaltet, der Zugriff ist ausschließlich über den Eventkalender möglich.

5.5.4.1 Sichten auf den Kalender

Es existieren zwei Sichten auf den Eventkalender, die als Interfaces implementiert sind.

- Das Interface **SimCalendar** implementiert nur eine Methode, mit der *SimEvent*-Objekte in den Kalender eingetragen werden können. Das neue Event wird entsprechend seiner Ausführungszeit im Kalender positioniert.

```
public interface SimCalendar
{
    public void insertEvent(SimEvent aSimEvent);
}
```

Abbildung 5.14: Das Interface SimCalendar

- Das Interface **ExtendedSimCalendar** aus Abbildung 5.15 ist ein Systemmodul, welches eine erweiterte Sicht auf den Kalender bereitstellt. Neben Methoden zum Eintragen von *SimEvents* werden hier noch Methoden zum Beschaffen und Abarbeiten des jeweils nächsten Events bereitgestellt. Diese erweiterte Sicht auf den Kalender wird innerhalb des Simulationsalgorithmus dazu verwendet, das jeweils chronologisch nächste Element zu erhalten. Für den Benutzer der Simulation reicht die normale Sicht in der Regel völlig aus.

```
public interface ExtendedSimCalendar extends SimCalendar
{
    public SimEvent getNextEvent();

    public EventManager getEventManager();
}
```

Abbildung 5.15: Das Interface ExtendedSimCalendar

Die Kalenderklassen sind allgemein genug um für alle späteren Simulationen verwendet werden zu können. Sie befinden sich im Package *com.pecasim.base.simulation* des Basisprojektes.

5.5.5 Der *EventManager*

In der Simulation läuft die komplette Eventverwaltung über den Eventkalender. Intern verwendet dieser den *EventManager*, der die genauen Ablage- und Cachingmechanismen organisiert. Auf der obersten Ebene im Basisprojekt wird der *EventManager* bereits als Interface zur Verfügung gestellt. Das Interface ist in Abbildung 5.16 dargestellt und definiert die grundlegenden Methoden, die vom Eventkalender verwendet werden können. Die Eventverwaltungsmethoden werden vom Eventkalender einfach in den EventManager umgeleitet. Genau wie beim Eventkalender auch, liefert die **getNextEvent** Methode das *SimEvent* mit dem frühsten Zeitpunkt, welches noch nicht abgearbeitet worden ist. Analog dazu können mit der

```
package com.pecasim.base.events;

import com.pecasim.base.database.DatabaseManager;

 * MicroSimulation
/**
 * Interface Eventmanager bietet eine Vorlage zum Zugriff auf die Eventverwaltung
 *
 * @author peha
 */
public interface EventManager {

    public SimEvent getNextEvent();

    public void insertEvent(SimEvent se);

    public void update(SimEvent simEvent);

    public void setDatabaseManager(DatabaseManager aDatabaseManager);

    public void setFinishDate(SimDate finishDate);

    public void flushCache();
}
```

Abbildung 5.16: Interface EventManager

InsertEvent Methode neue Events ins System eingefügt werden. Um abgearbeitete Events ins System einzutragen wird die **update** Methode verwendet. Mit dem Methodenaufruf werden ebenfalls alle zu dem übergebenen Event verknüpften Individuen aktualisiert. Der EventManager lagert die Events weitgehend in die Datenbank aus. Um Zugriff auf die Datenbank zu bekommen verwendet er einen *DatabaseManager*, der ihm initial über die Methode **setDatabaseManager** mitgegeben werden muss. Der EventManager verwendet den Zeitpunkt *FinishDate* um die Simulation zum entsprechenden Zeitpunkt abzubrechen. Dieser Abbruchzeitpunkt muss ebenfalls vor Beginn des Simulationslaufs über die Methode **setFinishDate** mitgeteilt werden. Um die Simulation erheblich zu beschleunigen werden Events innerhalb des *EventManager* in Datenspeichern zwischengespeichert. Mit der Methode **flushCache** werden diese Daten sofort in der Datenbank aktualisiert. In Abbildung 5.17 sind die beteiligten Klassen als UML-Klassendiagramm dargestellt. Die blau eingefärbten Klassen sind abstrakt genug um im Basisprojekt abgelegt zu werden. Die rot markierten werden von CoMICS ins Simulationsprojekt generiert. Als Vorlage für *EventManager*-Objekte wurde die Klasse *AbstractEventManager* ins Basisprojekt aufgenommen. Sie organisiert im Wesentlichen die Initialisierung des *DatabaseManager*-Objektes. Die eigentliche Implementierung wird in der Klasse *EventManagerImpl* im Package *com.pecasim.events* vorgenommen. Die Events werden hier in der Datenbank abgelegt bzw. nachgeladen. Zur Um-

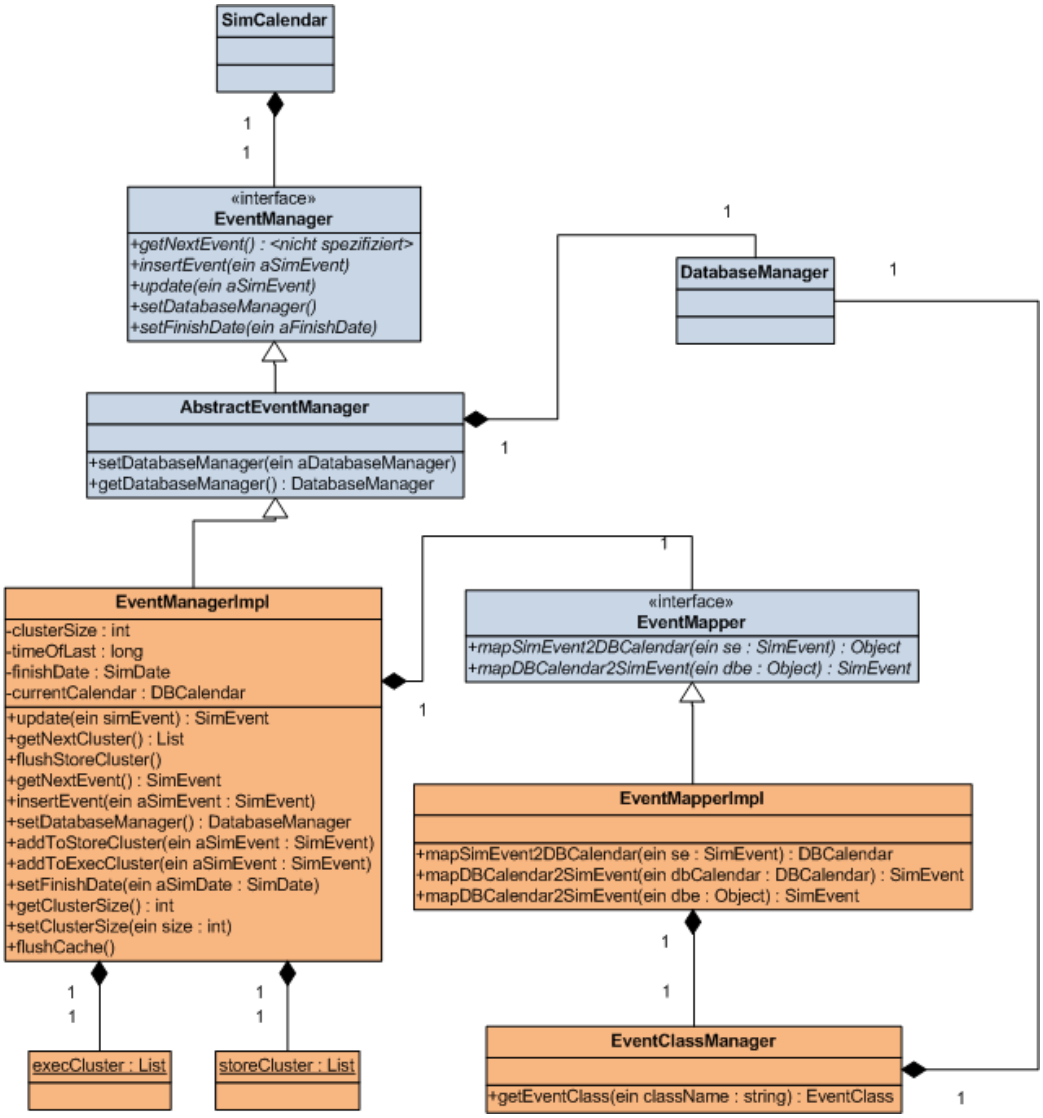


Abbildung 5.17: Aufbau des EventManager

wandlung zwischen den *SimEvents* und Einträgen in der relationalen Datenbank wird das bereits in Abschnitt **??** beschriebene Konzept angewendet. Der *EventMapper* stellt Methoden für die beidseitige Umwandlung von *SimEvents* und den Hibernate Domain-Objekten vom Typ *DBCalendar* bereit. Diese können dann sehr bequem über den *DatabaseManager* in der Datenbank abgelegt bzw. aktualisiert werden. Der *EventMapper* selbst ist ein Interface aus dem Basisprojekt und wird im Simulationsprojekt von der Klasse *EventMapperImpl* implementiert. Er besitzt einen *EventClassManager*, der die Eventklassen in der Datenbank verwaltet. Damit Eventklassen-Einträge nicht mehrfach angelegt werden, schaut der *EventClassManager* zunächst in der Datenbank nach und gibt die gefundene Eventklasse zurück. Sollte die Eventklasse noch nicht in der Datenbank vorhanden sein, wird eine Neue erstellt und zurückgegeben. Um Zugriff auf die Datenbank zu erhalten muss dem *EventClassManager* initial ein *DatabaseManager*-Objekt zur Verfügung gestellt werden. Dieses Verhalten wird komplett in der Methode **getEventClass** implementiert.

5.5.5.1 Zwischenpufferung von Events

Zum beschleunigen des Simulationslaufes verwendet der *EventManager* zwei Puffer, in denen Events abgelegt werden können. Die Größe der Puffer wird in der Variablen **clusterSize** angegeben, welche vom Benutzer mit der Methode *getClusterCacheSize* der Klasse *MySimulation* eingestellt werden kann. Der *execCluster* puffert die Events vor Abgabe an die Simulation, der *storeCluster* vor Ablage in der Datenbank. Mit den Methoden *addToExecCluster* und *addToStoreCluster* können Events sehr einfach in die jeweiligen Puffer abgelegt werden. In Abbildung 5.18 ist der interne Ablauf der *insertEvent*-Methode dargestellt. Der Aufruf dieser Methode geschieht über den *SimCalendar*, der dem Simulationsentwickler direkt zur Verfügung gestellt wird. Im ersten Schritt wird der Ausführungszeitpunkt des als Parameter übergebenen Events analysiert. In der Membervariablen **timeOfLast** der *EventManager*-Klasse wird der Zeitpunkt des spätesten Events im *execCluster* abgelegt. Liegt der Ausführungszeitpunkt des neuen Events vor *timeOfLast*, wird das neue Event zur Ausführung direkt im *execCluster* abgelegt. In diesem Sonderfall muss nach Ablage des Events der *execCluster* neu sortiert werden, damit die chronologische Reihenfolge gewahrt bleibt. Liegt der Ausführungszeitpunkt des neuen Events nach **timeOfLast**, wird es im *storeCluster*, zur Ablage in der

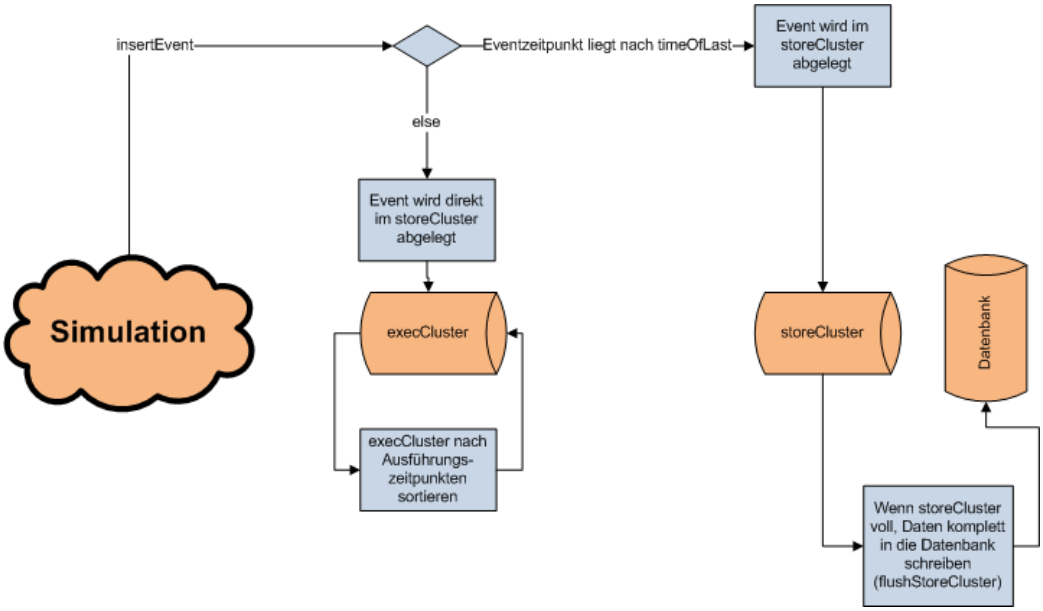


Abbildung 5.18: Ablauf der insertEvent Methode

Datenbank abgelegt. Überschreitet die Anzahl der Elemente im *storeCluster* den Schwellwert **clusterSize**, wird über die Methode *flushStoreCluster* der komplette Pufferinhalt in einem Transaktionsschritt in die Datenbank übertragen.
In Abbildung 5.19 ist dargestellt, wie der Methodenaufruf *getNextEvent*, der üblicherweise über den *SimCalendar* erfolgt, intern abgearbeitet wird. Der Eventkalender arbeitet alle eingetragenen Events in chronologischer Reihenfolge ab. Nach Ausführung eines Events besorgt er sich über den *EventManager* mit der Methode *getNextEvent* das jeweils nächste Event. Der *EventManager* prüft zunächst ob der execCluster leer ist. Sollte dies der Fall sein, verwendet er die *flushStoreCluster* Methode um alle im *storeCluster* gepufferten Änderungen bzw. Einträge in die Datenbank zu übertragen. Jetzt kann er von der aktualisierten Datenbank den nächsten Cluster in den *execCluster* übernehmen. Sollten in der Datenbank keine Events mehr vorhanden sein, wird ein leere Cluster zurückgegeben. Dieser wird als Austrittsbedingung für die Simulationsschleife verwendet. Der Simulationslauf wird erfolgreich beendet. Sind im *execCluster* Events abgelegt, wird das jeweils nächste Event für die Simulation ausgewählt. Sollte der Ausführungszeitpunkt des Events nach den *EventManager*-Attribut *finishDate* liegen, ist der Simulationszeitraum komplett abgearbeitet und der Simulationslauf wird erfolgreich beendet. Liegt der Ausführungszeitpunkt vor *finishDate*, wird das Event ganz normal an die

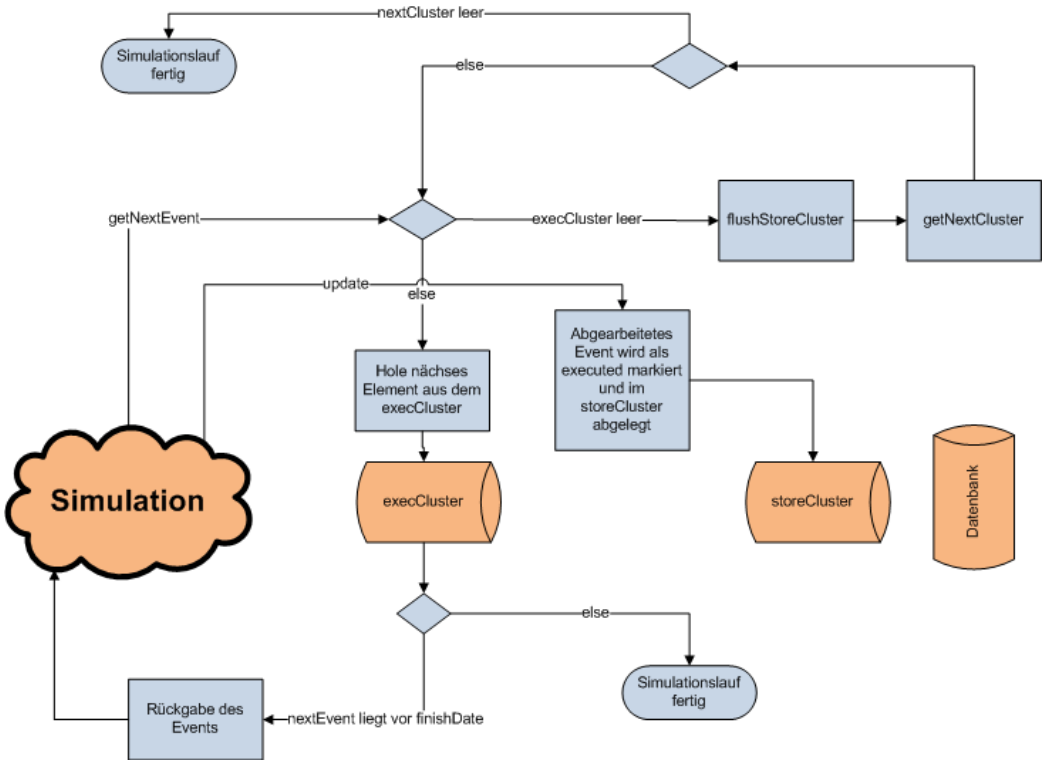


Abbildung 5.19: Ablauf der getNextEvent Methode

Simulation zurückgegeben und ausgeführt. Nach der Ausführung wird es mir der *update*-Methode wieder in den *EventManager* gegeben. Die *update*-Methode setzt das **executed**-Flag auf **true** und legt das abgearbeitete Event in den *storeCluster*. Hat der *storeCluster* die Größe *clusterSize* überschritten, wird er automatisch in einem Schritt in die Datenbank übertragen. Durch dieses Konzept wird verhindert, dass in jedem kleinen Simulationsschritt mehrere langsame Datenbankzugriffe benötigt werden.

5.5.6 Die Klasse *SimRandom*

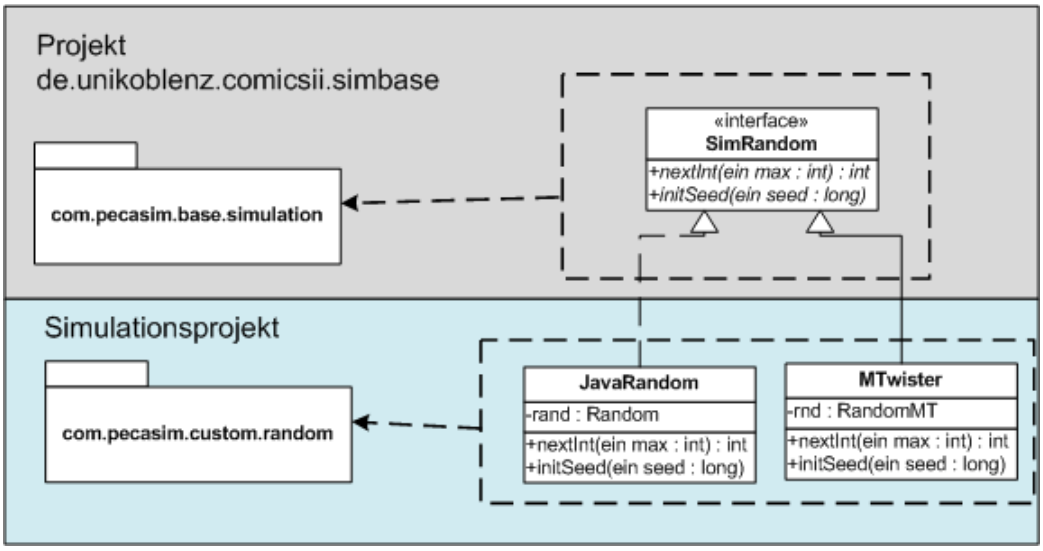


Abbildung 5.20: UML Diagramm des Zufallsgenerators

Als Erweiterung zur ersten Version von CoMICS wurde als Zufallsalgorithmus nicht mehr die Java Random Klasse, sondern der fortschrittliche Mersenne Twister verwendet. Es handelt sich hierbei um einen Zufallszahlengenerator der aktuell neusten Generation. CoMICS arbeitet intern mit dem Interface *SimRandom* des Basisprojektes um Zufallszahlen jeder Art zu erzeugen. Das Interface kommt bislang mit zwei Methoden aus.

- Mit der Methode **initSeed** kann dem *SimRandom*-Objekt eine Basis gegeben werden, auf Grund der er die folgenden Zufallszahlen erzeugt. In diesem Falle werden Generatoren mit dem gleichen Seed, exakt die gleichen Zufallszahlen-

```
public interface SimRandom
{
    public void initSeed(long seed);

    public int nextInt(int max);
}
```

Abbildung 5.21: Das Interface SimRandom

reihen erzeugen. Simulationen die ausschließlich diesen Generator verwenden werden somit rekonstruierbar.

- Die Methode **nextInt** generiert die eigentlichen Zufallswerte. Mit dem Integer Parameter **max** kann der Zahlenbereich nach oben beschränkt werden. Der Zufallsgenerator liefert immer eine ganze Zufallszahl aus dem geschlossenen Intervall *[0, ... , max]*. In CoMICS werden bislang ausschließlich ganzzahlige Zufallswerte verwendet. Wahrheitswerte werden durch den Aufruf *nextInt(1)* simuliert, wobei eine 0 als **false** und eine 1 als **true** interpretiert werden kann. Analog dazu kann eine Zufallsprozentwert durch einen Aufruf *nextInt(100)* dargestellt werden.

Im Simulationsprojekt wird dieses Interface im Package *com.pecasim.custom.random* durch die Klasse *MTwister* implementiert. Es befindet sich in diesem Package auch noch die ältere Klasse *JavaRandom*, sie findet aber in der aktuellen Version von CoMICS keine Anwendung. Sollten weitere Zufallsmethoden benötigt werden, kann das Interface SimRandom natürlich jederzeit erweitert werden. Diese gewünschten Funktionen müssten natürlich ebenfalls in der Klasse MTwister implementiert werden. Durch diese modulare Konzeption kann der Zufallsalgorithmus sehr einfach durch neuere Vesionen ersetzt werden. Hierzu muss erst einmal eine neue Klasse im Package *com.pecasim.custom.random* erstellt werden, die ebenfalls SimRandom mit dem entsprechend neuen Algorithmus implementiert. Der Zufallsgenerator wird in der Klasse MySimulation im Package *com.pecasim.custom.sim* ausgewählt und instanziert. Damit die neue Randomklasse in CoMICS verwendet wird, muss in der Klasse MySimulation die Methode getSimRandomClass nicht mehr die MTwister-, sondern die neue Randomklasse zurückgeben.

```
@Override
protected Class<? extends SimRandom> getSimRandomClass() {
    return MTwister.class;
}
```

Abbildung 5.22: Auswahl des Zufallsgenerators in der Klasse MySimulation

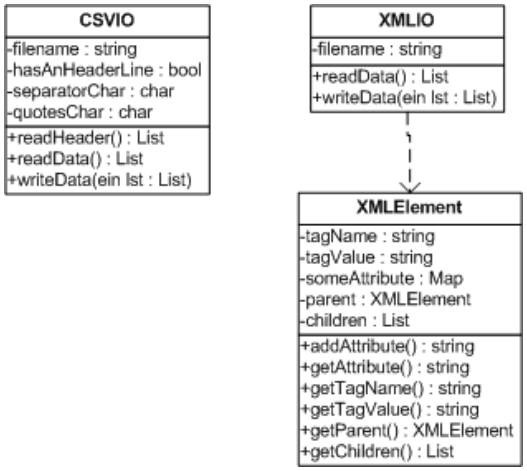


Abbildung 5.23: IO Klassen

5.5.7 Daten einlesen

Mit dem Package *com.pecasim.base.io* des Basisprojektes werden dem User Klassen zur Verfügung gestellt um Daten aus Dateien einzulesen, bzw. in Dateien hinein zuschreiben. Bislang unterstützt CoMICS die Verwaltung von CSV- und XML-Dateien. Weitere IO-Klassen, wie z.B. ein XLSIO für Excel-Dateien, können nach dem gleichen Aufbau nachgepflegt werden. In Abbildung 5.23 sind die Klassen *CSVIO* und *XMLIO* dargestellt. Beide Klassen halten die entsprechende Zieldatei als String in der Variablen **filename**.

5.5.7.1 CSVIO

In der CSV Variante müssen noch genauere Informationen angegeben werden. Die Variable **hasAnHeaderLine** gibt an ob in der CSV-Datei eine Headerzeile verwendet wird, oder ob die Daten direkt in der obersten Zeile anfangen. Das **separatorChar** gibt an, welches Zeichen in der CSV-Datei verwendet wird um die einzelnen Werte zu trennen. Mittels **quotesChar** ist es möglich das **separatorChar** innerhalb einer Wertfolge zu verwenden. Mit der Methode **readHeader** wird

der Header der CSV-Datei als String-Array ausgelesen. Mit **readData** werden alle Werte der Datei als Liste von String-Arrays geliefert. Eine vorgegebene Liste von String-Arrays kann über die Methode **writeData** in die Datei geschrieben werden.

5.5.7.2 XMLIO

Die Klasse XMLIO verwendet die Unterklasse *XMLElement*, um die XML-Daten zu verwalten. Mit den *XMLElement*-Objekten lässt sich jede XML-Datei als Baumförmige Struktur objektorientiert abbilden. Jedes Element stellt zunächst einen XML-Tag dar. Über die Attribute **parent** und **children** sind die Elemente im Baum angeordnet. In den Attributen **tagName** und **tagValue** werden die Werte aus der XML-Datei angegeben. Wie in XML auch, können weitere Attribute an die Tags angehängt werden. Diese werden in der Map **someAttributes** abgelegt. Die Methoden **getParent** und **getChildren** werden zur Navigation im XML-Baum benötigt. Die Tag-Werte können mit **getTagName** und **getTagValue** ausgelesen werden. Die zugehörigen XML-Attribute können mit **addAttribute** hinzugefügt bzw. mit **getAttribute** ausgelesen werden.

5.5.8 Die statistischen Werte

Mikrosimulationen basieren sehr stark auf statistischen Daten. Diese sind Grundlage für alle Entscheidungen, die in der Simulation getroffen werden. Sie können als so genannte *SimTables* in das Programm eingebunden werden. In der späteren Simulation kann aus den Events heraus auf diese Tabellen zugegriffen werden. Da die Daten nicht standardisiert, sondern in sehr unterschiedlicher Art und Weise vorliegen können, muss der Simulationsentwickler selbst für das korrekte Einlesen und den späteren Zugriff sorgen. Er ist für die Implementierung der Tabellenklassen selbst verantwortlich. In der Regel überlegt er sich eine geeignete Datenstruktur und stellt Methoden zum bequemen Zugriff auf die Daten bereit. Die statistischen Werte können in normalen Arrays, Collections oder auch Baumstrukturen abgelegt werden. Sie werden als Membervariablen in der neu implementierten Tabellen-Klasse geführt. Um Entscheidungen direkt treffen zu können, bekommen die Tabellen-Klassen noch Zugriff auf den internen Zufallsgenerator *SimRandom*. Im ersten Schritt können die verschiedenen Handlungsoptionen durch statistische Chancenverteilung anhand der hinterlegten Werte gewichtet werden. Im zweiten Schritt könnte mit dem Zufallsgenerator direkt entschieden werden, welche Hand-

lungsoption im konkreten Fall Anwendung findet.

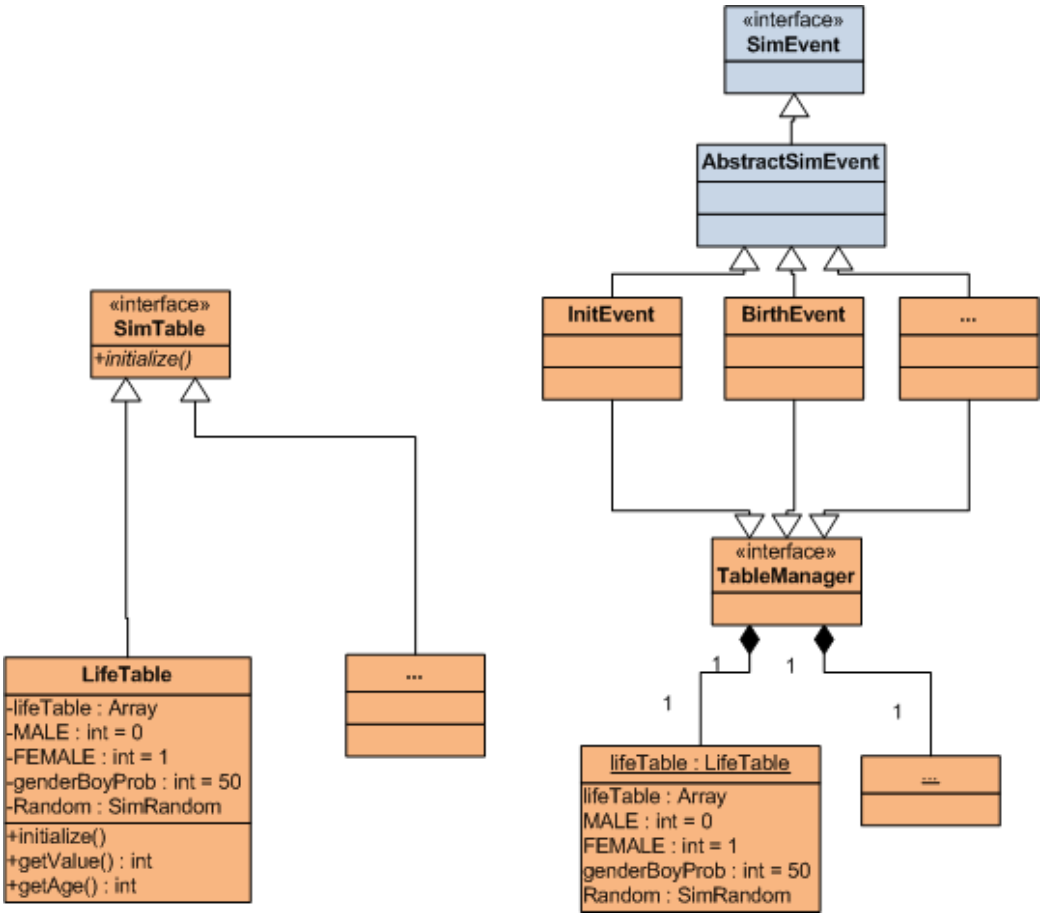


Abbildung 5.24: Die Klassen zur Ablage der statistischen Daten

5.5.8.1 Das Interface *SimTable*

Die Tabellen-Klassen implementieren immer das Interface *SimTable* und stellen damit die **initialize**-Methode zur Verfügung. Diese wird vor dem eigentlichen Simulationslauf aufgerufen und füllt die Datenstrukturen mit den entsprechenden Werten. Dem Benutzer stehen dazu die IO-Klassen aus Abschnitt 5.5.7 zur Verfügung. Es können natürlich auch andere Bibliotheken importiert werden, die beispielsweise Daten aus einer externen Datenbank einlesen können.

5.5.8.2 Das Interface *TableManager*

Der *TableManager* wird vor Simulationsstart ins Package *com.pecasim.custom.tables* des Simulationsprojektes hinein generiert. Er stellt ein ganz besonderes Interface dar, da er keine Methoden, sondern die Tabellenobjekte direkt enthält. Zu jeder Tabellenklasse wird jeweils ein Objekt im Interface als Membervariable abgelegt. Initialisiert wird der *TableManager* und alle in ihm enthaltenen Tabellenobjekte mit der Methode *initializeTables* der Klasse *AbstractSimulation*. Es wird zu jedem Tabellenobjekt die *initialize* Methode aufgerufen. Wie in Abbildung 5.24 dargestellt, implementieren alle *SimEvent*-Klassen dieses Interface und erhalten somit den direkten Zugriff auf die Tabellen-Objekte.

5.5.9 Einbeziehung von eigenen Klassen

Ein weiteres Feature, welches in der zweiten Version von CoMICS hinzugefügt wurde, ist die Möglichkeit benutzerdefinierte Klassen anzulegen. Der Simulationsentwickler kann für den Eigengebrauch frei beliebige Java-Klassen implementieren. Die Klassen werden bereits zur Entwicklungszeit in das Projekt eingebunden. Damit wird erreicht, dass der Eclipse-Editor bereits beim Erstellungsprozess nützliche Funktionen, wie Syntaxvervollständigung oder die automatische Fehleranzeige bieten kann. Zur Simulationszeit können diese Klassen dann instantiiert und verwendet werden.

5.5.10 Die *Simulation*

Der Rumpf der eigentlichen Simulation wird wieder im Basisprojekt bereitgestellt. Das Grundgerüst für die Simulation bildet das Interface *Simulation* und die abstrakte Klasse *AbstractSimulation*. Die Klassen sind im Basisprojekt im Package *com.pecasim.base.simulation* abgelegt. *AbstractSimulation* implementiert die Schnittstelle *Simulation* und stellt zudem alle allgemein gültigen Simulationsattribute zur Verfügung. Im Simulationsprojekt wird schließlich die Klasse *MySimulaiton* dazu verwendet werden, das eigentliche Simulationsobjekt zu instantiieren. *MySimulation* ist von *AbstractSimulation* abgeleitet und kann somit auf alle vordefinierten Attribute und Methoden der Oberklasse zugreifen. Da *MySimulation* zur Laufzeit von CoMICS generiert wird, wurde versucht die Klasse sehr schlank zu halten. Funktionalitäten und Attribute wurden soweit möglich in die Oberklasse verschoben. Die

Klasse *MySimulation* erweitert lediglich die simulationsspezifischen Bestandteile, die nicht vorherzusehen sind.

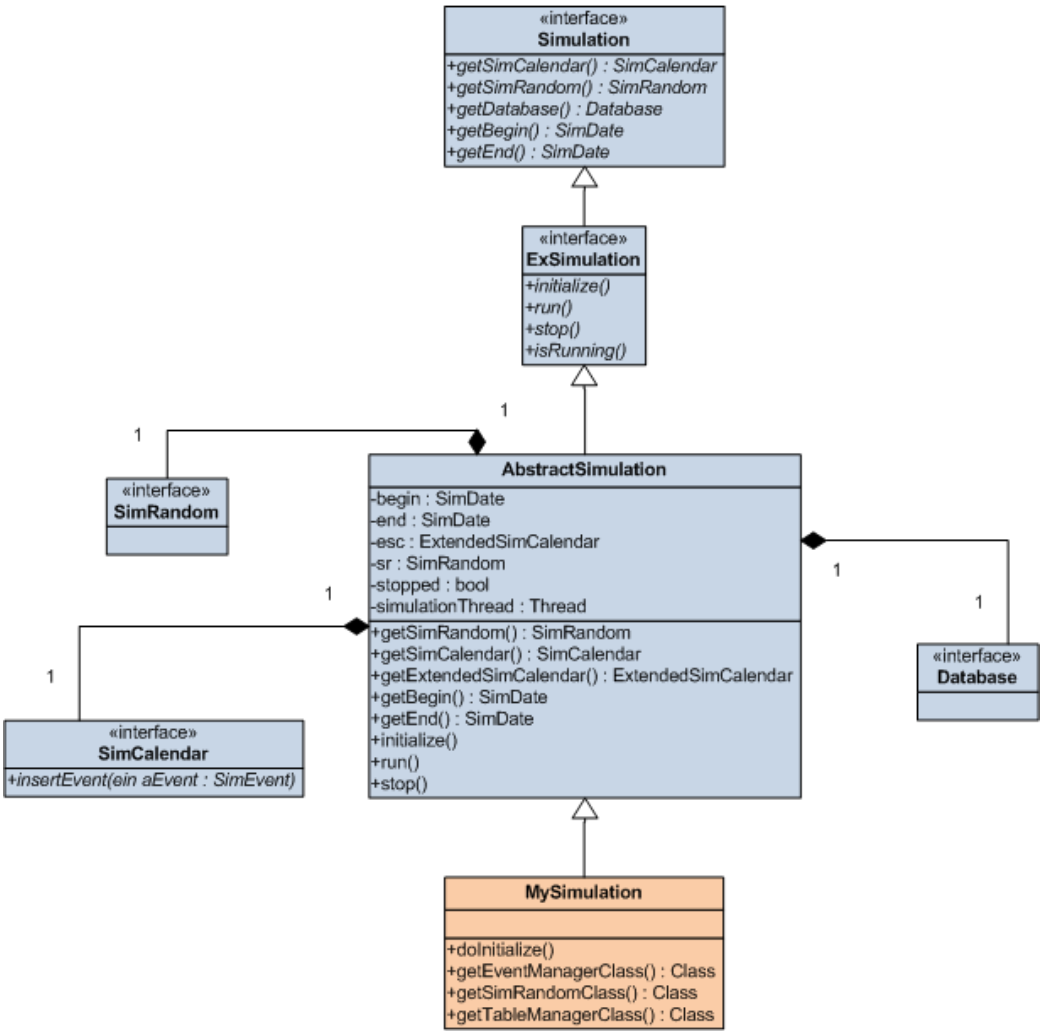


Abbildung 5.25: Zusammenhang der Simulationsklassen

5.5.10.1 Sichten auf die Simulation

In CoMICS II werden zwei Sichten auf Simulationsobjekte bereitgestellt. Die Schnittstelle **Simulation** aus Abbildung 5.26 stellt dem Simulationsentwickler diverse globale Methoden, zum Zugriff auf Basiselemente der Simulation, zur Verfügung.

- Die Methoden **getBegin** und **getEnd** liefern jeweils den Start- und Endzeitpunkt der Simulation. Rückgabewert ist stets ein *SimDate*. Die beiden

```
package com.pecasim.base.simulation;

import com.pecasim.base.database.Database;

 * MicroSimulation
 * Stellt die Standardsicht auf die Simulation bereit
public interface Simulation {

    public SimCalendar getSimCalendar();

    public SimRandom getSimRandom();

    public Database getDatabase();

    public SimDate getBegin();

    public SimDate getEnd();

}
```

Abbildung 5.26: Interface Simulation

Werte werden bei der konkreten Erzeugung des *MySimulation*-Objektes gesetzt. Bislang ist der Zugriff auf die beiden Werte zur Simulationslaufzeit nur lesend möglich. Möchte man innerhalb der Simulation den Start- bzw. Endzeitpunkt verschieben, müssten im Simulationsinterface noch die Methoden **setBegin** und **setEnd** vorgesehen werden. Sollten diese beiden Funktionen zur Verfügung gestellt werden, ist der spätere Simulationsentwickler selbst verantwortlich, keine Endlos-Simulationen zu erstellen.

- Die Methode **getSimCalendar** wird verwendet, um den Eventkalender, der zu dieser Simulation gehört, geliefert zu bekommen. Dieser Aufruf erfolgt im Programm an den unterschiedlichsten Stellen, da es sich beim *SimCalendar*-Objekt um eines der zentralen Elemente der Simulation handelt.

- Analog kann über die Methode **getSimRandom** der intern verwendete Zufallgenerator verwendet werden. Auch hier existiert innerhalb der Simulation nur ein einziges Objekt vom Typ SimRandom.

- Mit der Methode **getDatabase** kann sich der Benutzer ein Interface Database zur aktuellen Datenbank besorgen. Mit ihm können in der Simulation

Datenbankoperationen direkt ausgeführt werden.

Zur Ablaufsteuerung der Simulationen wird das in Abbildung 5.27 dargestellte Interface *ExSimulation* verwendet. Es erweitert das Simulationsinterface erweitert um Verwaltungsmethoden, die dem Simulationsentwickler verborgen bleiben sollen.

```
package com.pecasim.base.simulation;

import com.pecasim.base.ProgressMonitor;

 * MicroSimulation
 * Erweitert die Sicht auf ein Simulationsobjekt um ablaufsteuernde Mehtoden,
public interface ExSimulation extends Simulation {

    public void stop();

    public boolean isRunning();

    public void run(ProgressMonitor aMonitor) throws SimBaseException;

    public void initialise(ProgressMonitor aMonitor) throws SimBaseException;

}
```

Abbildung 5.27: Interface ExSimulation

- Mit der **initialize** Methode werden vorbereitende Maßnahmen zum folgenden Simulationslauf getroffen. Hier werden insbesondere *SimCalendar* und *SimRandom* konfiguriert und erstellt. Als Eingabeparameter bekommt die Methode einen ProgressMonitor, mit dessen Hilfe er dem Benutzer Ausgaben oder Fortschrittsanzeigen zur Verfügung stellen kann.

- Die beiden Methoden **run**, **isRunning** und **stop** werden verwendet, um den Simulationslauf zu starten bzw. eine laufende Simulation abzubrechen.

5.5.10.2 Die Klasse *AbstractSimulation*

In der abstrakten Klasse AbstractSimulation wird die Simulationsschnittstelle bereits teilweise implementiert. Soweit möglich wurden hier bereits allgemeingültige Methoden formuliert.

- Die beiden *SimDates* **begin** und **end**, sowie der Zufallsgenerator *SimRandom*, die Datenbankverwaltung *DatabaseManager* und der Eventkalender *ExtendedSimCalendar* werden bereits in dieser Klasse als Attribute gehalten. Zugriff auf die Werte erfolgt über die Simulationsmethoden **getBegin**, **getEnd**, **getSimRandom** und **getSimCalendar**. Der *DatabaseManager* ist als **protected** angelegt und kann in den Unterklassen direkt verwendet werden.

- Die Klasse implementiert die Methode **initialize**, welche die verwendeten Simulationselemente konfiguriert und anlegt. Nach Ausführung der Methode erfolgt ein Aufruf einer abstrakten *doInitialize*-Methode. Jede Klasse, die von *AbstractSimulation* abgeleitet ist, muss nun irgendwann die Methode *doInitialize* implementieren. Somit wird der Initialisierungsprozess ebenfalls auf die unterste Ebene weitergereicht und es können konkrete Vorbereitungen für die Unterklasse getroffen werden.

- Zur Ablaufsteuerung wurde die Klasse mit dem Thread **simulationThread** und einem Flag **stopped** ausgestattet. Der spätere Simulationslauf wird komplett in simulationThread ausgeführt. In regelmäßigen Abständen wird geprüft, ob der Benutzer über die *stop*-Methode das *stopped*-Flag gesetzt hat. Sobald *stopped* als **true** ausgewertet wird, bricht der Simulationslauf ab.

5.5.10.3 Die Klasse *MySimulation*

Die Klasse *MySimulaiton* ist von *AbstractSimulaiton* abgeleitet und erweitert diese Klasse um spezialisierte Elemente, die für das Basisprojekt zu konkret sind. Ein Beispiel für solche Elemente sind die einzelnen Individuen mit ihren Bezeichnungen und Attributen. Sie sind natürlich in fast jeder Simulation unterschiedlich und müssen somit in der Klasse *MySimulation* initialisiert werden. Im Basisprojekt wären zu den Individuen weder Namen noch Attribute verfügbar.

- Die **doInitialize** Methode wird vom Initialisierungsprozess aufgerufen und bietet der Klasse die Möglichkeit, Vorbereitungen zum folgenden Simulationslauf zu treffen.

- Um dem System die konkreten Klassen von **SimRandom**, dem **EventManager** oder dem **TableManager** bekannt zu machen, wurden an dieser

Stelle die Funktionen *getEventManagerClass*, *getSimRandomClass* und *getTableManagerClass* eingeführt. Sie liefern jeweils die einzelnen Java-Klassen der zugehörigen Elemente. Mit diesem modularen Aufbau ist es sehr einfach möglich komplette Sysemklassen durch andere Versionen zu ersetzen. Vorgabe ist natürlich stets, dass die entsprechenden Systeminterfaces korrekt und vollständig zu implementieren sind.

5.5.11 Der *SimManager*

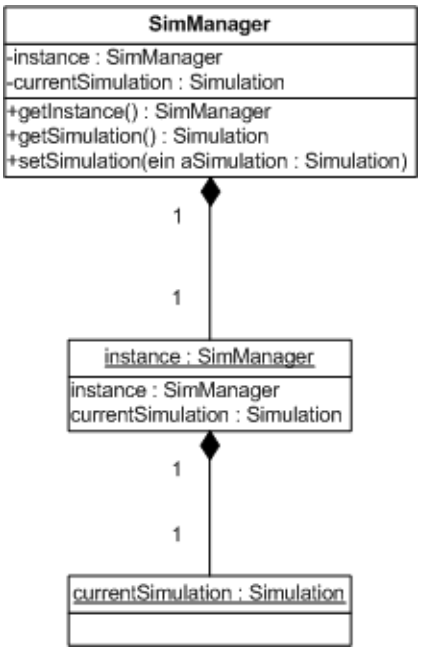


Abbildung 5.28: Aufbau des SimManager

Der *SimManager* ist nach dem Singleton Design Pattern konstruiert. Es existiert also im System nur ein einziges *SimManager*-Objekt, welches aus der kompletten Simulation zugreifbar ist. Dieses Objekt hält er selbst als statische Instanzvariable die mit *instance* bezeichnet ist. Aufgabe des *SimManger* ist es, die einzelnen Simulationen im System zu verwalten. Mit der Methode *getSimulaiton* stellt der SimManager einen systemweiten Zugriff auf gerade aktuelle Simulation zur Verfügung.

Intern hält er die gerade aktuelle Simulation in der Variablen *currentSimulation*. Mit Hilfe von *setSimulation* kann die gerade aktuelle Simulation gewechselt

```
Simulation aktuelleSimulation = SimManager.getInstance().getSimulation();
```

oder zurückgesetzt werden. In einer weiteren Ausbaustufe des Programms könnte der *SimManager* derart erweitert werden, dass er mehrere Simulationen parallel verwalten kann.

5.6 Der Simulationslauf

In der ersten Version von CoMICS wurde die Simulation in einem eigenen Thread ausgeführt. Aus dem Programm heraus konnte man den Thread zur Laufzeit verwalten. Durch diese Verschachtelung konnte die Simulation allerdings ausschließlich auf dem Entwicklungsrechner ausgeführt werden. In der zweiten Version von CoMICS werden beim Ausführungskonzept neue Wege beschritten werden. Um die eigentliche Simulation auch auf schnellere Rechner übertragen zu können, wird die Simulation als eigenständiges Projekt erstellt, welches als normale Java-Applikation ausgeführt werden kann. Auf dem Zielrechner muss nun nicht mehr die Entwicklungsumgebung CoMICS installiert sein. Die Simulation selbst läuft auf jedem Rechner, der das entsprechende Java-Runtime-Environment installiert hat.

5.6.1 Der *SimExecuter*

Die Klasse *SimExecuter* ist der Einsprungspunkt der Simulations-Applikation. Er wird entweder über den *Run*-Button der Entwicklungsumgebung direkt, oder über das exportierte Javaprojekt aufgerufen. Er implementiert ausschließlich eine *main*-Methode, die den Simulationslauf anstößt. Bislang werden die Kommandozeilenparameter, die generell an Java-Main-Methoden weitergereicht werden können, nicht weiter ausgewertet. In weiteren Ausbaustufen des Programms, könnten hierüber noch genaue Startoptionen gegeben werden, beispielsweise für die Steuerung eines Mehrclustersystems. Wie in Abbildung 5.29 zu sehen, legt der *SimExecuter* zu Beginn einen *DatabaseManager* an, der in der Simulation den Zugriff auf die Datenbank zur Verfügung stellt. Sofort nach der Instantiierung des *DatabaseManagers*, wird über die Methode *processSchemaUpdate* ein Schema-update ausgeführt um die fehlenden oder inkorrekten Datenbanktabellen anzulegen bzw. zu korrigieren. Im Anschluss wird aus der Datenbank die genaue Anzahl der Individuen ermittelt

```
package com.pecasim.simulation;
import com.pecasim.base.database.DatabaseManager;

 * MicroSimulation
 * Der SimExecuter ist Einsprungspunkt für den Simulationslauf
public class SimExecuter
{
    public static void main(String[] args)
    {
            try {
                //DatabaseManager initialisieren
                DatabaseManager dbManager = DatabaseManagerFactory
                        .getFactory()
                        .getDatabaseManager(
                                DatabaseManagerFactory.DEFAULT_DATABASE_CONFIG_FILE);

                //Datenbankschema aktualisieren
                dbManager.processSchemaUpdate();

                //Anzahl der Personenobjekte ermitteln
                int nrOfElements = dbManager.getNrOfElements(Person.class);

                //Simulation instantiieren
                MySimulation mySimulation = new MySimulation(dbManager);

                //SimulationControl anlegen und starten
                SimulationControl control = new SimulationControl(mySimulation, nrOfElements);
                control.start();

            } catch (Exception e) {
                e.printStackTrace();
            }
    }
}
```

Abbildung 5.29: SimExecuter als Einsprungspunkt in die Simulation

und in der Variablen **nrOfElements** abgelegt. Diese wird später als Grundlage zur Berechnung des Initialisierungsfortschrittes verwendet. Es wird eine neue Instanz der Klasse *MySimulation* erzeugt, welche den *DatabaseManager* als Parameter übergeben bekommt. Ein Objekt der Klasse *SimulationControl* wird mit diesem *MySimulation*-Objekt und der *nrOfElements* instantiiert und gestartet. Er organisiert den weiteren Ablauf der Simulation.

5.6.2 Der *ProgessMonitor*

Der *ProgressMonitor* stellt eine Oberfläche zur Verfügung, die zur Laufzeit der Simulation angezeigt wird. Er ist bereits zur Entwicklungszeit als Objekt in den *SimEvents* vorhanden und kann direkt angesprochen werden. Über ihn kann der Simulationsentwickler unter Anderem beliebige Zeichenfolgen an ein in der Oberfläche integriertes Konsolenfenster ausgeben. Der *ProgressMonitor* selbst ist im Package *com.pecasim.base* des Basisprojektes als Interface hinterlegt. Er stellt Methoden bereit, die Fortschrittsbalken oder Konsolenfenster verwalten.

5.6.3 Der *SimulationMonitor*

Die Klasse SimulationMonitor implementiert das Interface ProgressMonitor und stellt ein konkretes Simulationsfenster zur Verfügung. In ihm enthalten sind Fortschrittsbalken und ein Konsolenfenster, in welchem die Simulationsausgaben eingetragen werden können. Er verwendet Standard-Java Klassen um die Dialoge aufzubauen. Die genaue Anordnung und Konfiguration der Elemente wird in dieser Klasse vorgenommen.

5.6.4 Die *SimulationControl*

Die Klasse *SimulationControl* steuert den Ablauf der kompletten Simulation. Er verwaltet die Simulation selbst in einem Attribut und lässt sie in einem eigenen Simulations-Thread ablaufen. Er stellt die Methoden *start*, *suspendSimulation* und *stopSimulation* bereit um den Ablauf der Simulation zu steuern. Er verwendet einen *SimulationMonitor* um die Ablaufsteuerung über die Oberfläche ansprechen zu können. Mit der Methode *shutdown* kann das Programm zur Laufzeit abgebrochen werden. Soll die Simulation ohne Oberfläche im Hintergrund ablaufen, kann dazu die Methode *hideOrShowMonitor* verwendet werden. Der Methode *create-*

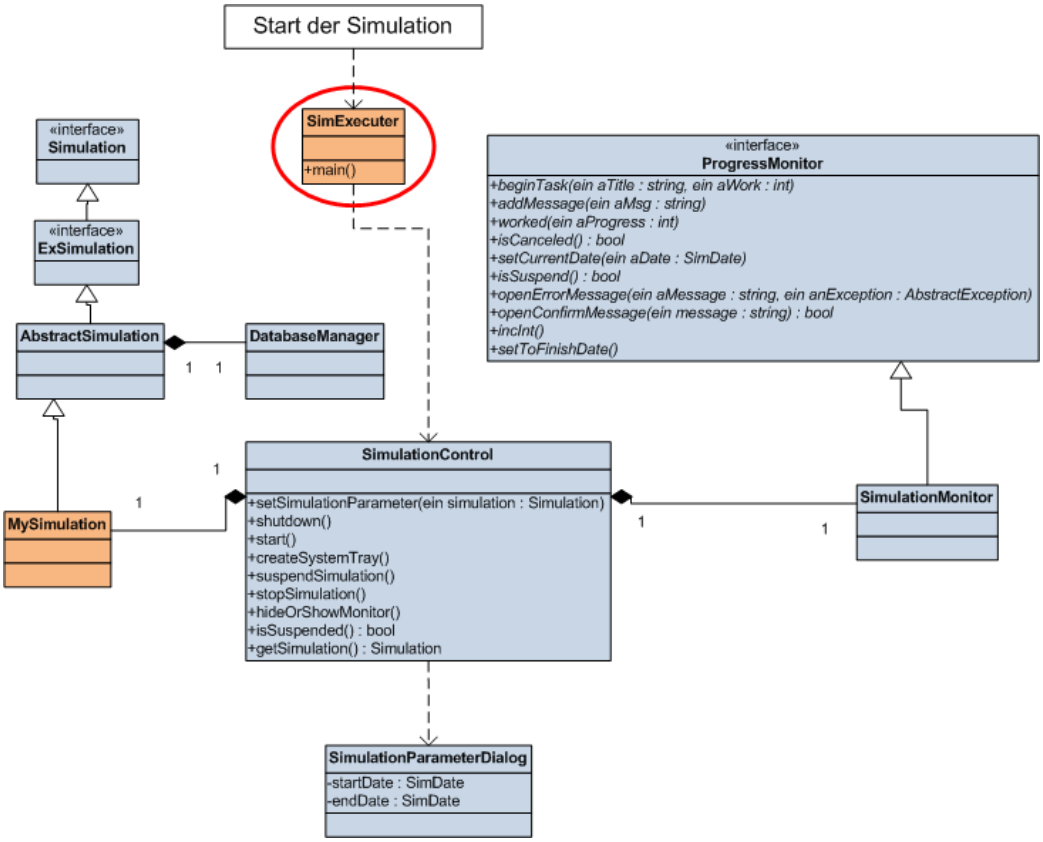


Abbildung 5.30: Zusammenhang der Klassen eines Simulationslaufs

SystemTray wird ein Icon in der Windows-Taskbar erzeugt. Über dieses kann das Fenster des *SimulationMonitors* wieder reaktiviert werden. Zu Simulationsstart verwendet er die Klasse *SimulationParameterDialog* um alle von der Simulation benötigten Parameter einzulesen. Bislang werden nur Start- und Endzeitpunkt der Simulation abgefragt.

5.7 Erzeugung von Simulationselementen

Das eigentliche Simutationsprojekt beinhaltet eine Verzeichnisstruktur, Java-Klassen und Konfigurationsdateien. Viele dieser Dateien sind für den jeweiligen Einsatzzwecks anzupassen. Daher können sich einzelne Projekte deutlich unterscheiden. Ein weiter Faktor ist die Weiterentwicklung von *de.unikobelnz.comicsii.simbase*, die direkten Einfluss auf die Projektstruktur haben kann. Liegt nun die Logik zum Generieren eines Projektes in *de.unikoblenz.comics.ui* und eine Strukturänderung wird durchgeführt, hätte dies Konsequenzen für jedes Modul der Anwendung. Daher muss der Generierungsprozess möglichst unabhängig gestaltet werden. Um dies Unabhängigkeit zu realisieren wurde ein Vorlagen-Mechanismus eingeführt, dessen Klassen in dem Paket *de.unikoblenz.generate* liegen.

5.7.1 Das Simulationsvorlage

Ein Simulationsprojekt kann man in zwei unterschiedliche Mengen unterteilen. Zum einen den unveränderlichen, benutzer-unspezifischen und zum anderen in den benutzer-spezifischen Teil. Der benutzer-unspezifische Teil eines Projektes kann in einer Simulationsvorlage direkt zusammengefasst werden. Den benutzer-spezifischen Teil muss entweder komplett neu generiert, oder durch Platzhalter (engl. Wildcard) anpassbar gemacht werden. Der komplett neu zu generierende Teil hingegen wird für die Vorlage ignoriert. Dies betrifft Hauptsächlich die Domain-Klassen (z.B. Person, Individuum, Haushalt, Region). Da jedoch aus den Modell auf diese Klassen zugegriffen wird, müssen all diese Stellen durch Platzhalter ersetzt werden.

Die Simulationsvorlage (engl. simulation template) setzt sich somit aus den benutzer-unspezifischen und den durch Platzhalter erweiterten Teilen zusammen. Um die möglichen Platzhalter zu definierten wurde das Enum *DOMAINTYPE* erzeugt, das folgende Wildcards beinhaltet: UNKNOWN, PERSON, HOUSEHOLD, REGION, COMPANY, DOMAIN1, ..., DOMAIN5. In der Simulationsvorlage müs-

sen nun alle Stellen, die einen benutzer-spezifischen Domain-Klasse beinhalten könnten, durch diese Platzhalter ersetzt werden. Hierbei besitzt ein solcher Platzhalter im Quelltext immer das Prefix '${' und das Postfix '}'. Die Abbildung 5.31 zeigt Auszüge aus dem *InitEvnet*, das durch Wildcards von der Domain-Klasse losgelöst wurde. Zusätzlich beinhaltet das Template die Konfigurationsdateien '.class-

```
...
import com.pecasim.custom.domain.${PERSON};
import com.pecasim.custom.tables.TableManager;

public class InitEvent extends AbstractSimEvent implements TableManager {
    private ${PERSON} rolePerson;
...
    public InitEvent(SimDate aDate, ${PERSON}... aRoles) {
        super(aDate);
        rolePerson = aRoles[0];
    }
...
}
```

Abbildung 5.31: Das *InitEvent* in der Simulationsvorlage

path' und '.project', die das Simulationsprojekt in seiner Struktur beschreiben. Die Dateien 'dbconfig.xml' und 'systemdod.xml' müssen hingegen immer neu generiert werden.

Alle Verzeichnisse und Dateien, sowie alle dazugehörigen Bibliotheken (z.B. 'de.unikoblenz.comicsii.simbase.jar' und 'comics-libs.zip') werden zu einem ZIP-Archiv zusammengefasst und mit der Endung '.template' abgespeichert und bildet somit die Simulationsvorlage.

Die eigentlich Generierung des Simulationsprojekts wird von einer Instanz der Klasse *ProjectGenerator* durchgeführt. Für die Generierung muss ein neues, leeres Simulationsprojekt im *Workspace* und die Beschreibung der Domain-Klassen vorliegen. In das Verzeichnis des neuen Simulationsprojekts werden alle Verzeichnisse und Dateien der Vorlage kopiert, nachdem sämtliche Wildcards gemäß der Beschreibung ersetzt wurden. Zum Schluss werden die Domain-Klassen generiert und die Konfigurationsdateien erstellt.

5.7.2 Die *SimEvent*-Vorlage

Ähnlich wie die Simulationsvorlage werden *SimEvent*s auch von einer Vorlage abgeleitet. Hierzu stehen wiederum Platzhalter zur Verfügung. Die Generierung wird von der Klasse *EventClassGenerator* geleistet, die auch mögliche Wildcards in dem

```
package com.pecasim.custom.events;

import com.pecasim.base.ProgressMonitor;
import com.pecasim.base.events.AbstractSimEvent;
import com.pecasim.base.simulation.SimDate;
import static com.pecasim.custom.tables.TableManager.*;
import com.pecasim.custom.domain.${RoleClass};

public class ${EventClassName} extends AbstractSimEvent {
${RoleFields}
    protected void doExecute(ProgressMonitor aMonitor) {
    }
    public ${EventClassName}(SimDate aDate, ${RoleClass}...aRoleArray) {
        super(aDate);
${RoleFieldsFiller}
    }
    public ${EventClassName}() {}
}
```

Abbildung 5.32: Inhalt der Datei 'eventclass.template'

Enum *PLACEHOLDER* bereitstellt. Definierte Wildcards sind: *EventClassName*, *RoleFields*, *RoleClass*, *RoleFieldsFiller*. Da das entsprechende Template von dem Simulationsprojekt abhängt, liegt die Datei 'eventclass.template' im Verzeichnis 'templates' des jeweiligen Projekts. Die Abbildung 5.32 zeigt den Inhalt dieser Datei.

5.7.3 Die Manipulation von Quellcode

Eine weitere Möglichkeit Quellcode zu bearbeiten, ist die Verwendung vom AST. Diese Technik wurde in Abschnitt 3.3.4.2 ausführlich beschrieben. Wir verwenden diese Art der Quellcode Generierung und Manipulation für die statistischen Objekte, hauptsächlich um eine Registrierung im Interface *TableManager* zu realisieren, sowie bei der Erzeugung von einfachen Klassen. Um einfache Javaklassen zu erzeugen ist eine Instanz der Klasse *SimpleClassGenerator* zuständig und für die statistischen Elemente die Klasse *TableClassGenerator*.

5.8 Die Modellierungsumgebung

Die Modellierungsumgebung einer Simulation beinhaltet zwei Perspektiven. Das Startcenter gibt einen Überblick über alle erzeugten oder importierten Simulationsprojekte und die Modellierungsperspektive beinhaltet alle Elemente eines Simulationsmodells.

5.8.1 Das Startcenter

Das Startcenter erlaubt es dem Nutzer, die vorhandenen Simulationsprojekte zu verwalten. Mit Hilfe dieser Perspektive kann ein neues Projekt erzeugt oder importiert werden. Wie in Abbildung 5.33 ersichtlich, besteht das Startcenter aus

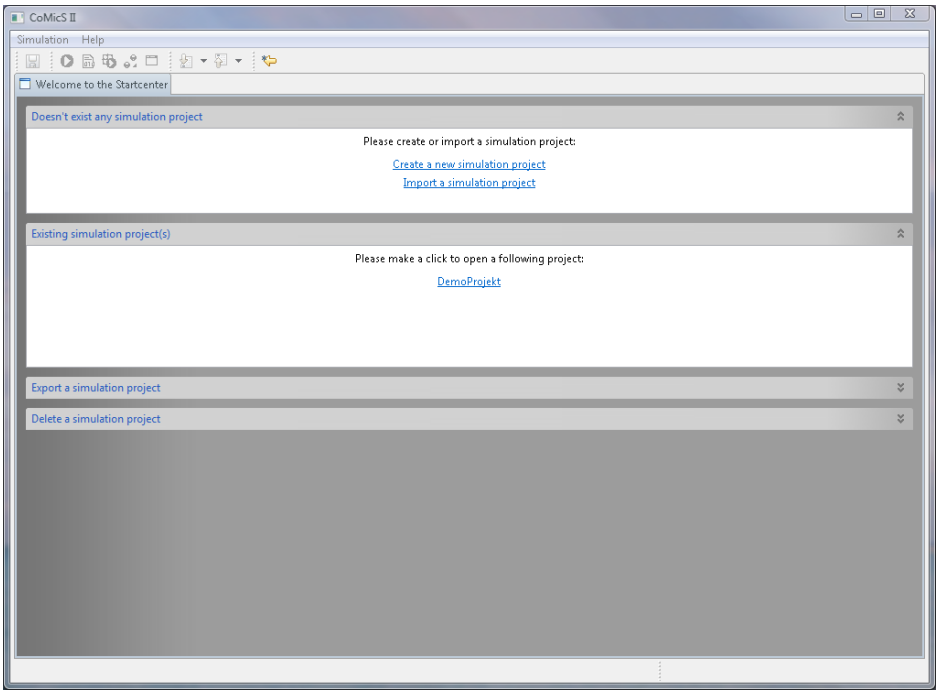


Abbildung 5.33: Das Startcenter von CoMICS II

vier Funktionsbereichen, die ein- bzw. ausklappbar sind. Über den ersten Bereich ist die Erstellung bzw. das Importieren eines Simulationsprojekts möglich. In den jeweiligen anderen Bereichen werden alle Projekte des *Workspace* angezeigt. Durch anklicken des jeweiligen Projekts in einem Funktionsbereich, kann es geöffnet, gelöscht oder exportiert werden.

Die Implementation des Startcenter besteht aus den Klassen im Paket *de.unikoblenz.comicsii.ui.startcenter*. Die Klasse *StartCenterPerspectiveFactory* übernimmt hierbei die Initialisierung der Perspektive, die ausschließlich aus der *StartCenterView* besteht. Eine Instanz von *StartCenterView* beinhaltet mehrere Instanzen einer Unterklasse von *AbstractStartCenterPart* bzw. dem Interface *StartCenterPart* und stellt einen Funktionsbereich dar.

In der Datei 'plugin.xml' muss die Klasse *StartCenterPerspectiveFactory* unter

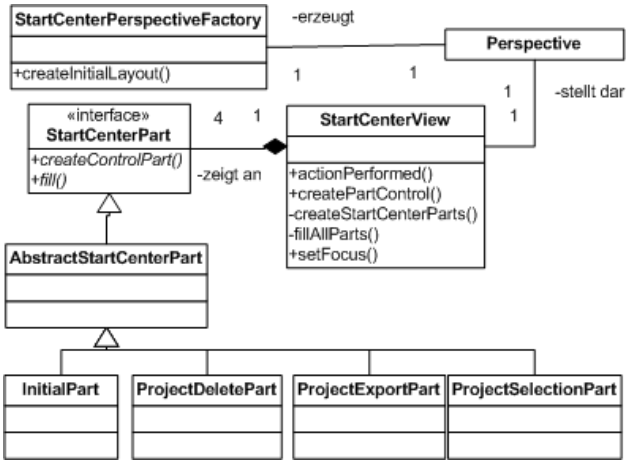


Abbildung 5.34: Implementation des Startcenters

dem Extension-Point *org.eclipse.ui.perspectives*, sowie die Klasse *StartCenterView* unter dem Extension-Point *org.eclipse.ui.views* registriert werden.

5.8.2 Die Modellierungsumgebung

Die Modellierungsumgebung beinhaltet alle zur Modellierung einer Simulation relevante Javaklassen. Diese können auf eigene Bedürfnisse angepasst werden. Hierbei unterstützt der Editor die Implementierung von Javaklassen, indem Warnungen und syntaktische Fehler direkt angezeigt werden. Des weiteren können zur Verfügung stehende Variablen und Methoden als Auswahl angezeigt werden. Dies erfolgt nach der Eingabe eines Punkts hinter der Referenz eines Objektes automatisch, kann aber auch durch gleichzeitiges Drücken von STRG und Leertaste manuell erfolgen.

Die Perspektive gliedert sich generell in den Navigationsbereich auf der rechten Seite des Fensters, dem Editorbereich im oberen linken Sektor und dem Statusbereich rechts unten. Der Navigationsbereich unterteilt sich weiter in die Übersicht der Domain-Klassen (z.B. Personen, Haushalte), der *SimEvent*-Klassen, der statistischen Klassen (z.B. Lebenstafel) und dem Simulationsbereich mit der Klasse *MySimulation* und frei erzeugbaren Elementen. Des weiteren ermöglicht die Navigation das Erzeugen bzw. Entfernen von *SimEvent*s, *SimTable*s und freien Javaklassen.

Mit Hilfe der Werkzeugleiste im oberen Sektor des Fensters wird weitere Funk-

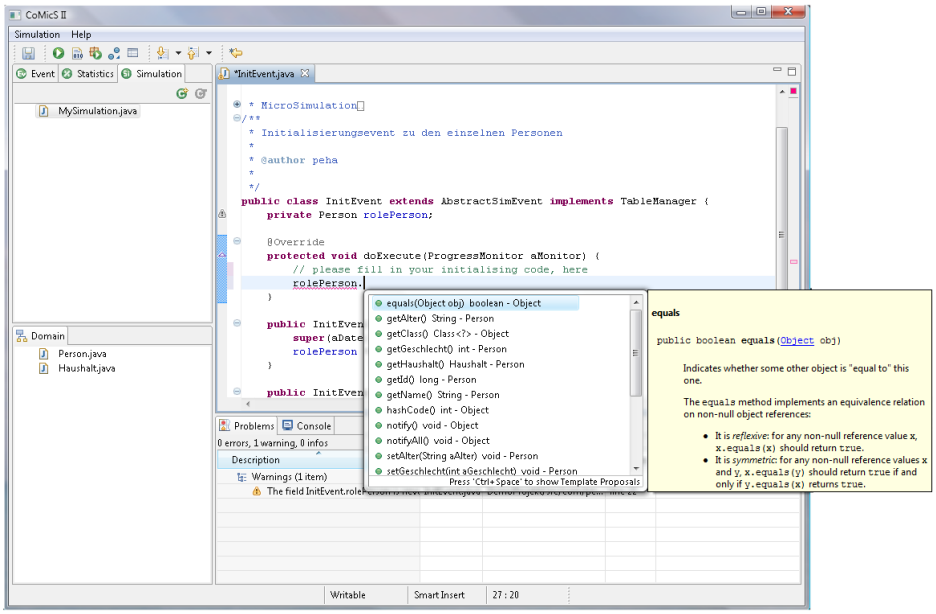


Abbildung 5.35: Die Simulations-Entwicklungsumgebung

tionalität dem Benutzer zur Auswahl gestellt. Die Werkzeugleiste beinhaltet die in Abbildung 5.36 dargestellten Symbole. Diese Elemente ermöglichen von Links

Abbildung 5.36: Die Werkzeugleiste der Modellierungsperspektive.

nach Rechts das abspeichern einer Klasse, das Rekompilieren des Projekts, das Exportieren des Projekts zu einer Jar-Datei, das Anzeigen des Verbindungs- und Import-Dialogs und das Starten des Einstellungsdialogs.

Der Statusbereich beinhaltet die Konsole und die Problem-Ansicht. Die Konsole zeigt alle Programmausgaben an, die z.B. mit *System.out.println(„“)* erzeugt wurden. Zudem kann eine laufende Simulation oder ein aktiver Importvorgang abgebrochen werden. Hierzu beinhaltet die Konsole das rote Stopp-Symbole im oberen rechten Rand.

Die Problem-Ansicht listet alle Warnungen und Fehler im Quellcode in tabellarischer Form auf. Durch einen Doppelklick auf ein Problem, wird die entsprechende Klasse geöffnet und der Cursor selektiert die problematische Stelle im Quellcode. Am rechten Rand des Editors wird zusätzlich ein kleines Symbol auf der Höhe des Problems eingeblendet, das weitere Informationen als Tooltip bereithält.

5.8.3 Realisierung als RCP Plugin

Um eine RCP-Applikation realisieren zu können, werden mehrere Klassen benötigt, die in ihrer Funktion schon näher im Abschnitt 3.3 erläutert wurden. Das Basis-Paket *de.unikoblenz.comicsii.ui* beinhaltet die grundlegenden Klassen, die das Starten einer eigenständigen Applikation mit dem CoMICS II Plugin ermöglichen. Die Klasse *Application* implementiert das Interface *IApplication* und sorgt für die Lauffähigkeit als eigenständige Applikation. Die Klasse *Activator* aktiviert das eigentliche Simulations-Plugin. Alle relevanten Einstellungen für das Hauptfenster werden von der Klasse *ApplicationWorkbenchWindowAdvisor* vorgenommen. Darunter die Initiale Größe, der Fenstertitel oder ob z.B. die Statusleiste angezeigt wird oder nicht. Die Initialisierung der Werkzeugleiste wird von *ApplicationActionBarAdvisor* übernommen. Die Klasse *PECASIM* beinhaltet alle Schlüssel der RCP-Komponenten als Konstante, das können Aktionen, Perspektiven, Ansichten oder Einstellungen sein.

Aufbauend auf dem Basis-Paket beinhalten die untergeordneten Pakete verschiedene Funktionsbereiche. Das Paket *de.unikoblenz.comicsii.ui.actions* stellt alle Aktionen zur Verfügung. Hierzu muss eine Aktion das Interface *IAction* implementieren. Für CoMICS sind zwei unterschiedliche Aktionen vorgesehen. Zum einen die *AbstractAction* und dessen Spezialisierung *AbstractSimulationAction*. Die spezielle Simulationsaktion besitzt eine enge Verknüpfung zu dem aktuell geöffneten Projekt und bezieht sich auf dessen Kontext (z.B. ob das Projekt gerade geschlossen wird). Einen solchen Kontext besitzt die 'normale' Aktion nicht. Erzeugt werden die Aktionen durch die *SimActionFactory*.

Das Paket *de.unikoblenz.comicsii.ui.startcenter* beinhaltet die *PerspectiveFactory*, sowie alle Klassen für das StartCenter. Analog dazu beinhaltet das Paket **.simulation* die Elemente der Modellierungsoberfläche.

Durch die XML-Datei 'plugin.xml' werden die CoMICS-Komponenten an die RCP gebunden. Hierzu zählen die Klasse *Application*, die den Extension-Point *org.eclipse.core.runtime.applications* erweitert und die Erstellung eines eigenständigen Programms ermöglicht. Der Extension-Point *org.eclipse.ui.perspectives* wird durch die *StartCenterPerspectiveFactory*, sowie von *SimulationPerspectiveFactory* erweitert. Zudem registriert diese XML-Datei sämtliche *Views*.

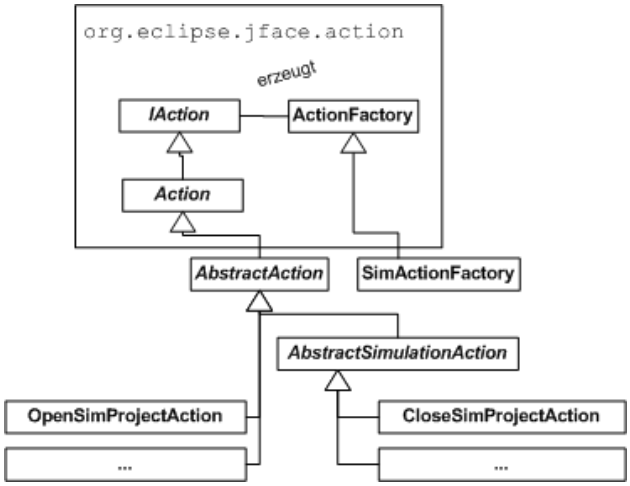


Abbildung 5.37: Die CoMICS II - Aktionen

5.8.4 Die Klasse *SimulationSession*

Die Klasse *SimulationSession* ist ein zentrale Bauteil der CoMICS II Oberfläche. Er verwaltet den aktuellen Modellierungsvorgang und hält Methoden bereit, die u.a. den Datenimport oder die Simulation starten können. Zudem liefert er die Projektbeschreibung, in Form eines Objekts der Klasse *ProjectDescriptor* und eine Instanz von *DatabaseManager*. Hierbei erweitert die *SimulationSession* die zwei Interfaces *SimulationManipulator* und *SimulationProjectStructure*. *SimulationManipulator* spezifiziert die Methoden, die das Erzeugen und Löschen von *SimEvent*s, *SimTables* und einfachen Java Klassen ermöglicht. Die Schnittstelle *SimulationProjectStructure* beinhaltet dagegen Methoden, die den Zugriff auf strukturelle Elemente erlauben. Hierunter fällt die Instanz des *IJavaProject* und dessen Verzeichnis- bzw. Paket-Struktur.

Zu einem Zeitpunkt kann ausschließlich eine Instanz der *SimulationSession* aktiv sein. Um dies zu gewährleisten werden die *SimulationSession*s von der einzigen Instanz der Klasse *SimulationSessionFactory* verwaltet, die gemäß des Singleton Pattern spezifiziert wurde. Die *SimulationSessionFactory* bietet somit Methoden zum Erstellen, Öffnen, Schliessen und Löschen eines Simulationsprojekts. Zudem besitzt die *Factory* einen Event-Mechanismus, der registrierte *SimulationSessionListener* über eine Schliessen- oder Öffnen-Aktion benachrichtigt. Diese Funktionalität wird z.B. von der *AbstractSimulationAction* ausgiebig genutzt, um eine solche Aktion nur bei geöffnetem Simulationsprojekt nutzbar zumachen. Sobald eine *Si-*

mulationSession geöffnet wird, werden alle *AbstractSimulationAction* benachrichtet und aktivieren sich. Analog dazu verhält sich der Mechanismus beim Schliessen einer Session.

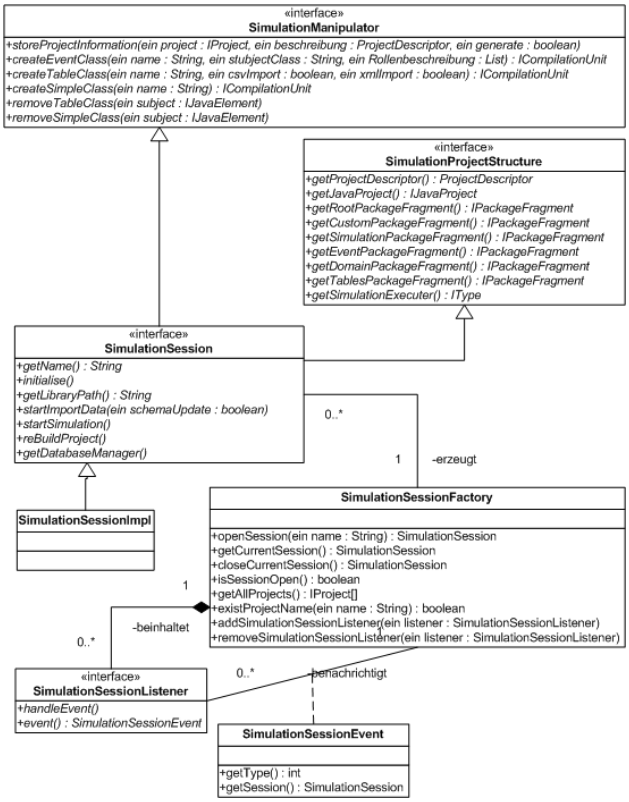


Abbildung 5.38: Klassenstruktur der Verwaltung eines Modellierungsvorgangs

5.8.5 Geführte Benutzer-Interaktion

In vielen Szenarien ist es nötig, eine Menge von Informationen vom Benutzer abzufragen. Diese Informationen können aufeinander aufbauen oder einfach sehr zahlreich vorkommen. Um diese Daten zu erfassen bietet *JFace*, wie im Abschnitt 3.2.2 vorgestellt, den *Wizard*. Ein *Wizard* ermöglicht eine sequenzielle Abarbeitung mehrerer Seiten, die unterschiedliche Inhalte darstellen können.

5.8.5.1 Erzeugung eines neuen Simulationsprojekts

Der *Wizard* zum Erzeugen eines neuen Simulationsprojekts kann über das *Start-Center* gestartet werden. Die erste *WizardPage* wird durch die Klasse *SimProject-ParamsPage* repräsentiert und beinhaltet grundlegende Angaben, wie den Name des Projekts und die zu verwendete Projektvorlage. Hierbei ist zu beachten, dass der Name des neuen Projekts noch nicht im Workspace für ein anderes Projekt vorhanden sein darf. Erst wenn der Name und die Projektvorlage gewählt wurden, kann zu der zweiten Seite gewechselt werden. Die nächste Seite wird von der Klasse *DatabaseConnectionDataPage* visualisiert, die zur Änderung der Verbindungsdaten dient. Die darauf folgende Seite ermöglicht die Angabe der zu importierenden Dateien. Da momentan ausschließlich CSV-Dateien unterstützt werden, ist die einzige Implementation hierzu die Klasse *CsvFileSelectPage*. Neben der Angabe des Separater-Zeichens und anderen datei-spezifischen Angaben, muss der entsprechende Platzhalter selektiert werden. Jede Zeile einer Datei entspricht einem Datenobjekt, dessen Typ durch den *DataObjectDescriptor*(DOD) beschrieben wird. Nachdem alle Dateien hinzugefügt wurden und somit alle DODs fest stehen, muss eine Relation zwischen den Objekten hergestellt werden. Dies vollbringt die letzte Seite, die durch die Klasse *NewSimObjectEditorPage* realisiert wurde. Hier werden alle *DataObjectDescriptors* ähnlich eines Klassendiagramms angezeigt. Jedes DOD

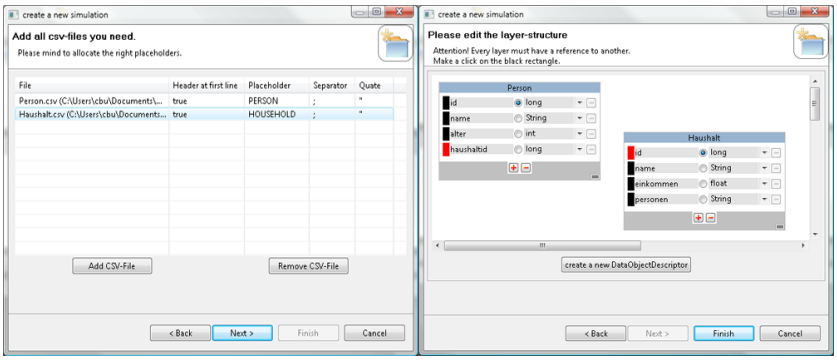


Abbildung 5.39: Die Zuordnung von *DataObjectDescriptor*s

wird mit einer Instanz von *DODEditor* bearbeitet, die wiederum in dem *DODEditorContainer* zusammengefasst werden. Relationen zwischen zwei *DODEditor*s werden durch ein Objekt der Klasse *DODConnector* realisiert und vor dem entsprechenden Attribut farbig dargestellt. Zudem besteht ein solches Attribut aus einem Namen, der Angabe ob es sich um ein Schlüsselattribut handelt und einem

Datentyp. Die Abbildung 5.39 zeigt auf der linken Seite die Selektion der CSV-Dateien und auf der rechten Seite die Bearbeitung der entsprechenden DODs.

Um den Wizard beenden zu können, müssen alle DOD mindestens einem anderen DOD zugeordnet werden. Beim Beenden des *Wizard*s wird eine Instanz vom *ProjectDescriptor* erzeugt und alle Informationen die über den *Wizard* gesammelt wurden sind diesem Objekt übergeben. Anschließend wird mit die Methode *createNewProject()* von der *SimulationSessionFactory* aufgerufen. Diese Methode verlangt als Parameter den vorher erzeugten *ProjectDescriptor* und erzeugt daraus ein neues Simulationsprojekt.

5.8.5.2 Hinzufügen neuer *SimEvent*s

Die Werkzeugleiste über dem Event-Navigator ermöglicht es dem Benutzer neue Events anzulegen bzw. bestehende zu löschen. Für die Erstellung eines neuen Events dient ein *Wizard*, der den Namen, die Rolle- bzw. Subjekt-Klasse und alle Rollennamen erfasst. Diese Informationen werden der Methode *createEventClass()* von der *SimulationSession* übergeben, die die entsprechende Klasse erstellt. Die

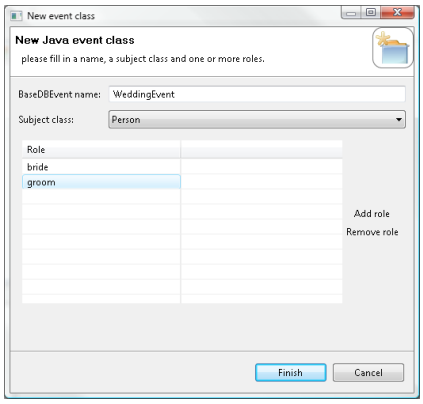


Abbildung 5.40: Der SimEvent-*Wizard*

erzeugte Klasse besteht aus einem leeren Konstruktor, der ausschließlich für den internen Gebrauch gedacht ist und nicht zur Instanziierung eines Objekts verwendet werden darf und einem Konstruktor mit den Paramtern *SimDate* und allen Rollennamen. Zudem werden alle Rollen als Instanzvariablen angelegt, wobei sich der Variablennamen aus dem Präfix 'role' und dem Rollennamen zusammensetzt (z.B. roleBride). Schließlich wird die Methode *doExecute()* erzeugt, die von dem Benutzer entsprechend angepasst werden muss.

5.8.5.3 Erstellung neuer statistischer Klassen

Die statistischen Objekte, die in der früheren Version von CoMICS in Form von Arrays bereitgestellt wurden, werden in CoMICS II durch eigenständige Klassen repräsentiert. Für die Erzeugung einer solchen Klasse steht wiederum ein *Wizard* zur Verfügung, der den Namen des Objektes und die Form in der die statistischen Daten vorliegen erfasst. Eine wichtige Methode hier ist *initialise()*, mit der die Daten eingelesen bzw. erstellt werden. Um diese Daten Ziel gerecht abfragen zu können, kann der Benutzer eigene Methoden erstellen, die den Zugriff auf die gewünschten Daten erlauben. Zudem wird eine solche Klasse im Interface *TableManager* als Konstante angelegt und kann somit aus jedem *SimEvent* direkt angesprochen werden.

5.9 Verwaltung von Einstellungen

Die RCP beinhaltet einen generellen Mechanismus um Einstellungen zu verwalten. Hierzu kann ein Plugin den Extension-Point *org.eclipse.ui.preferencePages* mit einer Implementation von dem Interface *IWorkbenchePreferencePage* erweitern. [12, S451-454]

Da die Verwaltung von Einstellungen mit Hilfe der RCP-Preference Pages von vielen RCP-Plugins ausgiebig genutzt wird, beinhaltet der Preference-Dialog sehr viele Optionen, die die Übersichtlichkeit start einschränken. Da CoMICS II jedoch nur wenig Einstellungen erfordert, wäre der generelle RCP-Mechanismus überdimensioniert. Daher wurde ein eigener Extension-Point für die CoMICS II Einstellungen geschaffen.

5.9.1 Der *SettingDialog*

Der *SettingDialog* stellt eine Menge von *ISettingPage*s dar, die mit Hilfe des Extension-Point Mechanismus registriert werden können. Hierbei wird eine *SettingPage* als eigener Karteireiter im Dialog angezeigt. Eine abstrakte Implementation ist *SettingPage*, dessen Methode **doCreateContent()** für eine eigene Seite überschrieben werden muss. Hier werden die grafischen Elemente angeordnet, die in der Methode **fill()** gefüllt werden können. Registriert wird eine solche Seite in der 'plugin.xml' unter dem Extension-Point *de.unikoblenz.comicsii.ui.simulationSettingPage*. Hierzu muss das Tag **settingPage** mit der Angabe der Attribute **title**, **icon** und **class**

hinzugefügt werden. Wobei der **title** den eigentlichen Anzeigetext der Einstellung (Karteireiterbeschriftung), das **icon** die URL zu einem Bild und die **class** die konkrete Implementation von *ISettingPage* darstellt.

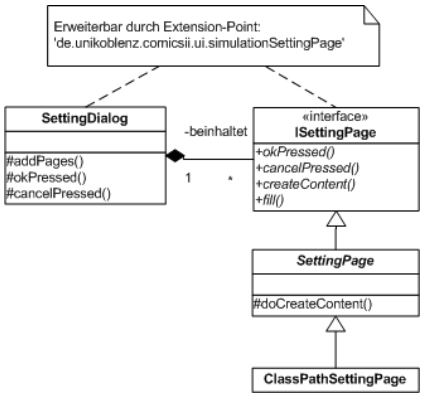


Abbildung 5.41: Der *SettingDialog* und die *ISettingPage*

Die momentan einzige Verwendung dieses Mechanismus ist in Form der Klasse *ClassPathSettingPage*, die die Möglichkeit eröffnet, den Classpath eines Javaprojektes zu verändern.

Kapitel 6

Ausblick und Fazit

6.1 Stand der Entwicklung

Die Umsetzung eines Werkzeugs zur Erstellung von event-orientierten Simulationsmodellen wurde mit dieser Arbeit abgeschlossen. Sowohl die Modellierungsumgebung, als auch das Simulations-Framework ermöglichen eine bequeme Entwicklung eines Simulationsmodells und dessen Ausführung. Die schon im Abschnitt 4.1 herausgestellten Anforderungen wurden wie folgt realisiert:

- **Verbesserung der Skalierbarkeit und Beschleunigung der Ausführung**
  Wurde maßgeblich durch den Paradigmenwechsel zu einer event-orientierten Mikrosimulation erreicht. Zudem wurde eine externe Datenbank angebunden.

- **Bessere Unterstützung für die Modellierung einer Simulation in Java**
  Konnte durch den erweiterten Einsatz von RCP-Bestandteilen weiter ausgebaut werden.

- **Wahrung der Konsistenz der Daten**
  Konsequentere Fehlerbehandlung und den Einsatz von Integritätsbedingungen der Datenbank erhöhten die Konsistenz der Daten.

- **Flexibilisierung des Zugriffs auf statistische Daten**
  Wohldefinierte Schnittstellen ermöglichen dem Benutzer einen programmati-

schen Zugang zu CSV-Daten. Somit werden alle Möglichkeiten der Programmiersprache Java für diesen Zweck nutzbar.

- **Auswertbarkeit mit externen Werkzeugen**
  Die Verwendung des SQL-Standards zur Verwaltung von Daten ermöglicht den Einsatz sehr mächtiger externer Reporting-Werkzeuge.

Die Verwendung des neuen Werkzeugs wird im folgenden an einem elementaren Beispielmodell illustriert.

Um ein neues Modell zu erstellen bietet CoMICS II einen eigenen Wizard an. Neben dem Namen des Simulationsprojekts müssen die entsprechenden CSV-Dateien angegeben werden, welche die Basisdaten beinhalten. Hierzu werden in diesem Beispiel zwei Dateien verwendet. Zum einen die Datei 'person.csv', die acht Personen mit den Attributen 'id', 'name', 'age', 'sex', 'householdid', 'death' und 'partnerid' beinhaltet. Zum anderen die Datei 'household.csv', die über drei Haushalte mit den Attributen 'id' und 'address' verfügt. Nach der Selektion der Dateien und der Wahl der richtigen Platzhalter (person.csv = PERSON und housold.csv = HOUSHOLD), müssen die Verknüpfungen zwischen den Dateien bzw. Datenobjekten hergestellt werden. Eine Verknüpfung wird durch gleiche Farben vor den entsprechenden Attributen angezeigt. Zudem müssen die passenden Datentypen ausgewählt werden. Diese Einstellungen können anhand der Abbildung 6.1 nachvollzogen werden.

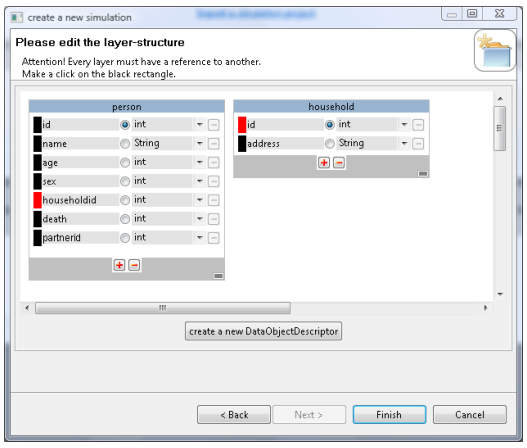


Abbildung 6.1: Einstellung der Datenobjekte

Nachdem der Wizard beendet wurde, erzeugt die Applikation das Grundgerüst des Modells, öffnet die Modellierungsperspektive und beginnt die Daten in

die Datenbank zu importieren. Während des Importvorgangs kann das Modell bereits editiert werden. Unser Beispiel soll, neben dem *InitEvent*, aus drei weiteren *SimEvents* bestehen. Um diese Events zu erzeugen wird wiederum ein Wizard angeboten, der den Eventname, die Subjektklasse (hier Person oder Household) und die verschiedenen Rollen erfasst.
Die drei somit erzeugten *SimEvent*s sind:

- **BirthdayEvent**: Inkrementiert das 'age'-Attribut einer Person und erzeugt ein neues *BrithdayEvent* für das kommende Jahr. Dieses Event ist für größere Modelle so nicht empfehlenswert, da für jedes Individuum in jedem Durchlauf dieses Event ausgeführt werden muss.

```
private Person rolePerson;

/**
 * Implementation of this SimEvent.
 */
protected void doExecute(ProgressMonitor aMonitor) {
    if (rolePerson.getDeath() == 0) {
        rolePerson.setAge(rolePerson.getAge() + 1);
        getSimCalendar().insertEvent(
                new BirthdayEvent(getDate().addYear(1), rolePerson));
    }
}
```

- **DeathEvent**: Setzt den Wert '1' in das 'death'-Attribut einer Person und kennzeichnet sie damit als gestorben. Diese Markierung muss in anderen Events berücksichtigt werden, da z.B. eine Hochzeit nach dem Tod eines Individuums nicht mehr möglich ist.

```
private Person rolePerson;


/**
 *  Implementation of this SimEvent.
 **/
protected void doExecute(ProgressMonitor aMonitor)
{
    rolePerson.setDeath(1);
    aMonitor.addMessage(rolePerson.getName() + " ist verstorben!");
}
```

- **WeddingEvent**: Lässt zwei Personen heiraten, indem das 'partnerid'-Attribut mit der 'id' der anderen Person belegt wird. Zudem wird ein neuer Haushalt erzeugt und dem Ehepaar zugeordnet.

```
private Person rolebride;
private Person rolebridegroom;

/**
 * Implementation of this SimEvent.
 */
protected void doExecute(ProgressMonitor aMonitor) {
    if (rolebride.getDeath() == 0 && rolebridegroom.getDeath() == 0
            && rolebride.getPartnerid() == 0
            && rolebridegroom.getPartnerid() == 0) {
        rolebride.setPartnerid(rolebridegroom.getId());
        rolebridegroom.setPartnerid(rolebride.getId());
        Household household = new Household("Weddingstreet");
        rolebride.setHouseholdid(household);
        rolebridegroom.setHouseholdid(household);
        aMonitor.addMessage(rolebride.getName() + " hat "
                + rolebridegroom.getName() + " geheiratet!");
    }
}
```

Die Initialisierung dieser drei *SimEvent*s wird beim Simulationsstart für jedes Individuum von dem *InitEvent* geleistet. Dort wird in diesem Beispiel für jede Person ein *BirthdayEvent* erzeugt, wobei der Monat der Ausführung zufällig ermittelt wird. Ebenfalls wird für jedes Individuum ein *DeathEvent* angelegt, wobei der Ausführungszeitpunkt mit Hilfe der Lebenstafel ermittelt wird. Die Lebenstafel beinhaltet zu jedem Alter und Geschlecht die Anzahl an Personen, die relativ zu 100000 Menschen noch am Leben sind. Im ersten Schritt wird dieser Bestand für die entsprechende Person ermittelt. Anschließend wird auf Basis dieser Zahl eine kleinere Zufallszahl erzeugt und das erste Alter ermittelt, dessen Bestand kleiner als diese Zufallszahl ist. Dieses Alter wird als Sterbealter der Person angenommen.
Im Gegensatz dazu wird das *WeddingEvent* nur für Personen erzeugt, welche noch keinen Partner haben. Dies wird durch den Wert '0' in dem Attribut 'partnerid' angezeigt. Der potentielle Partner wird mit Hilfe einer Datenbankabfrage ermittelt, welche alle Menschen mit dem passenden Geschlecht und die über keinen Partner verfügen selektiert. Anschließend wird das erste Individuum aus der Ergebnismenge als Partner ausgewählt. Die Hochzeit dieser zwei Personen wird für fünf Jahre nach der Initialisierung angenommen. Ein Quellcode-Ausschnitt des *InitEvent* ist in der Abbildung 6.2 ersichtlich.

Das erstellte Beispielmodell wurde nach Fertigstellung von 2008 bis 2050 simuliert. Als Ergebnis sind 5 Personen gestorben, zwei Individuums haben geheiratet und ein Haushalt wurde erstellt. Die Veränderungen sind in Abbildung 6.3 ersichtlich. Die Auswirkungen des *BirthdayEvent* wurden blau, des *DeathEvent* wurde grau und des *WeddingEvent* wurde rot markiert dargestellt. Die Abbildung 6.4 zeigt exemplarische einige Einträge der Tabellen 'Calendar' und 'Event'. Das At-

```
private Person rolePerson;

@Override
protected void doExecute(ProgressMonitor aMonitor) {
    // BirthdayEvent
    int monthToBirthday = getSimRandom().nextInt(12)+1;
    getSimCalendar().insertEvent(
            new BirthdayEvent(getDate().addMonth(monthToBirthday),
                    rolePerson));

    // DeathEvent
    int exist = lifetable.getValue(rolePerson.getAge(), rolePerson.getSex());
    int random = getSimRandom().nextInt(exist);
    int ageToDie = 100;
    for (int age = rolePerson.getAge(); age < 100; age++){
        exist = lifetable.getValue(age, rolePerson.getSex());
        if (exist < random){
            ageToDie = age;
            break;
        }
    }
    getSimCalendar().insertEvent(
            new DeathEvent(getDate().addYear(ageToDie-rolePerson.getAge()), rolePerson));

    // WeddingEvent
    if (rolePerson.getPartnerid() == 0) {
        List<Person> lstPerson = getSimulation().getDatabase().get(
                Person.class,
                "from Person where partnerid=0 and sex="
                        + (rolePerson.getSex() == 0 ? 1 : 0));
        if (!lstPerson.isEmpty()) {
            getSimCalendar().insertEvent(
                    new WeddingEvent(getDate().addYear(5), rolePerson,
                            lstPerson.get(0)));
        }
    }
}
```

Abbildung 6.2: *InitEvent* des Beispielmodells

tribut 'eventid' verknüpft ein Calendereintrag mit einem Event. Dies wurde in der Abbildung farblich gekennzeichnet. Dieses kleine Beispiel gibt einen Überblick über

Datenbasis vor Simulation

id [PK]	age integer	death integer	name charact	partnerid integer	sex inte	householdid integer
1	55	0	herbert	2	0	1
2	54	0	anna	1	1	1
3	44	0	heidi	4	1	2
4	42	0	markus	3	0	2
5	80	0	josef	6	0	3
6	79	0	mariane	5	1	3
7	18	0	peter	0	0	1
8	15	0	susi	0	1	2

Datenbasis nach Simulation

id [PK]	age integer	death integer	name charact	partnerid integer	sex inte	householdid integer
1	70	1	herbert	2	0	1
2	55	1	anna	1	1	1
3	76	1	heidi	4	1	2
4	84	0	markus	3	0	2
5	86	1	josef	6	0	3
6	88	1	mariane	5	1	3
7	60	0	peter	8	0	4
8	57	0	susi	7	1	4

Abbildung 6.3: Basisdaten vor und nach der Simulation

Die Tabelle 'Calendar'				Die Tabelle 'Event'		
id [PK] bigint	date timestam	executed boolean	eventid bigint	id [PK] big	name character var	eventclassid bigint
15105	2008-02-01	TRUE	15105	15100	InitEvent	1600
15106	2008-02-01	TRUE	15106	15101	InitEvent	1600
15107	2008-02-01	TRUE	15107	15102	InitEvent	1600
15100	2008-02-01	TRUE	15100	15103	InitEvent	1600
15101	2008-02-01	TRUE	15101	15104	InitEvent	1600
15102	2008-02-01	TRUE	15102	15105	InitEvent	1600
15103	2008-02-01	TRUE	15103	15106	InitEvent	1600
15104	2008-02-01	TRUE	15104	15107	InitEvent	1600
15116	2008-03-01	TRUE	15116	15108	BirthdayEvent	1601
15112	2008-04-01	TRUE	15112	15109	DeathEvent	1602
15120	2008-11-01	TRUE	15120	15110	BirthdayEvent	1601
15108	2008-11-01	TRUE	15108	15111	DeathEvent	1602

Abbildung 6.4: Die Calendar- und Event-Tabelle des Beispielmodells

die Leistungsfähigkeit und die Flexibilität von CoMICS II. Die Ausführungszeit dieses Modells betrug nur wenige Sekunden. Dies lag, neben der sehr beschränkten Datenbasis, an dem sehr effizienten und gut skalierbaren Datenmanagement. Die Auswertung einer solchen Simulation kann durch externe Reporting-Werkzeuge schon während des Simulationsdurchlaufs erfolgen.

6.2 Potentiale und Weiterentwicklung

Obwohl die Entwicklung von CoMICS II hiermit abgeschlossen ist und alle vorgenommenen Punkte realisiert wurden, gibt es noch einige Punkte die zu einer Verbesserung des Produkts beitragen könnten.

- **Verteilte Simulation**
  Hierzu müssen die Daten bzw. Ereignisse partitioniert werden, um auf unterschiedlichen Maschinen abgearbeitet zu werden. Dies erfordert eine Synchronisierung der einzelnen Prozessen.

- **Teils grafische Modellierung**
  Zu prüfen wäre die Möglichkeit einer grafischen Unterstützung während der Modellierung. Schön wäre hier eine schematische und grafische Anordnung der Ereignisse im Kalender. Hierdurch könnte die Übersichtlichkeit der Simulationsstruktur erheblich verbessert werden.

- **Erweiterung der Nutzung von Hibernate**
  Optimierungspotential besitzt hier auch die verwendeten Domainklassen, die von Hibernate genutzt werden (z.B. Person, Haushalt). Eine Relation auf die nächst höhere Ebene ist bereits realisiert. Leider wurde wegen der Gefahr auf Zyklen auf eine Relation der oberen Schicht auf die Unteren verzichtet. So muss, um an alle Personen eines Haushalt zu erhalten eine entsprechende Anfrage an die Datenbank gesendet werden.

- **Nutzung von ETL**
  Für den Import der Daten in die Datenbank könnten verschiedene ETL (Extract Transform Load) Bausteine ausgebaut werden, welche das Einlesen der verschiedensten Daten stark vereinfachen würden. Alle Daten werden in CoMICS zur Zeit unverschlüsselt abgelegt und verwaltet. Hier sollten ggf. Datenbank Zugangsdaten oder andere sicherheitsrelevante Daten mit entsprechenden Verfahren abgesichert werden. Nach Ablauf der Simulation könnte CoMICS auch weitere Analysefunktionen bereitstellen, die sonst erst nach einem Datenimport in einem speziellen Analyseprogramm - wie z.B. Microsoft Excel - zur Verfügung stehen.

Da CoMICS II sehr modular aufgebaut wurde, ist die Möglichkeit von Erweiterungen bzw. Ergänzungen durch andere Projektgruppen gegeben. Als Voraussetzung

für die Arbeit an diesem Projekt sind gute Kenntnisse in JAVA, Erfahrung im Umgang mit relationalen Datenbanken und die Einarbeitung in RCP zu nennen. Falls diese Gegebenheiten vorliegen, steht der Erweiterung und der ausgiebigen Nutzung dieses Produktes nichts mehr im Wege.

Literaturverzeichnis

[1] Alfons Kemper, André Eickler: *Datenbanksysteme, eine Einführung*. Oldenbourg, 5. Auflage, 2004.

[2] Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullmann: *Compilerbau Teil 1*. Oldenbourg, 2. Auflage, 1999.

[3] Beck, Kent: *JUnit, Pocket Guide*. O'Reilly, 2004.

[4] Bernhard Lahres, Gregor Rayman: *Praxisbuch Objektorientierung, Von den Grundlagen zur Umsetzung*. Galileo Computing, 2006.

[5] Boenigk, Cornelia: *PostgreSQL, Grundlagen, Praxis, Anwendungsentwicklung mit PHP*. dpunkt.verlag, 1. Auflage, 2003.

[6] Bond und Gasser: *Readings in Distributed Artificial Intelligence*, 1988.

[7] Bracha, Gilad: *Generics in the Java Programming Language*. SUN, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.

[8] Chaib-draa: *Trends in Distributed Artificial Intelligence*, 1992.

[9] Christian Bauer, Gavin King: *Java Persistence mit Hibernate*. HANSER, 2007.

[10] Daum, Berthold: *Rich-Client-Entwicklung mit Eclipse 3.1*. dpunkt.verlag, 2005.

[11] eclipse, http://help.eclipse.org/help32/index.jsp: *Eclipse documentation - Archived Release*.

[12] Eric Clayberg, Dan Rubel: *Eclipse, Building Commercial-Quality Plug-ins*. Addison-Wesley, 2. Auflage, 2006.

[13] Fishman, George S: *Discrete-Event Simulation.* Springer.

[14] Harding, A.: *Microsimulation and Public Policy.* North Holland.

[15] Hartwig, Jens: *PostgreSQL, Professionell und praxisnah.* Addison-Wesley, 2001.

[16] Hoffmann, Dirk W.: *Software-Qualität.* Springer, 2008.

[17] Jeff McAffer, Jean-Michel Lemieux: *Eclipse Rich Client Platform, Designing, Coding, and Packaging Java Applications.* Addison-Wesley, 2005.

[18] Krüger, Guido: *Handbuch der Java-Programmierung.* Addison-Wesley, 4. Auflage, 2006.

[19] Lyons, John: *Die Sprache.* 4. Auflage, 1992.

[20] Matthew Scarpino, Stephen Holder, Stanford Ng Laurent Mihalkovic: *SWT / JFace in Action.* Manning, 2005.

[21] Nigel Gilbert, Klaus G. Troitzsch: *Simulation for the Social Scientist.* Open University Press, 2. Auflage, 2005.

[22] Pascal Berger, Dennis Fuchs, Peter Hassenpflug Christian Klein: *MicSim, Agentenbasierte Mikrosimulation.*

[23] Pavel Haiduk, Peter Heusinger, Michael Wagner: *Einsatz von OSGi auf Hardwarelimitierten Modulen.* Fraunhofer Institut Integrierte Schaltungen.

[24] Robinson, Stewart: *The Practice of Model Development and Use.* John Wiley and Sons, 2003.

[25] Sommerville, Ian: *Software Engineering.* PEARSON Studium, 6. Auflage, 2006.

[26] SUN Microsystems, http://java.sun.com/products/hotspot/whitepaper.html,: *The Java HotSpot Performance Engine Architecture.*

[27] SUN Microsystems, http://java.sun.com/j2se/1.5.0/docs/index.html: *JDK 5.0 Documentation.*