Aloys-Fischer-Schule Deggendorf Staatliche Fachoberschule und Berufsoberschule Sozialwesen - Technik - Wirtschaft

20. November 2013

Jahnstraße 5

94469 Deggendorf

Seminararbeit

Schuljahr 2010/ 2011

Thema:

Simulation dynamischer Systeme implementiert mit der Programmiersprache Java

vorgelegt von

Markus Richter, FB13W

Abgabetermin: 17.01.2011

betreuende Lehrkraft: Rudolf Muhr, StR

Inhaltsverzeichnis

1 Einleitung

1.1 Thema und Motivation

> Though this be madness, yet there is method in't.
>
> William Shakespeare, Hamlet

Wissenschaftler, Ingenieure und Ausführende veschiedener Berufe stützen sich seit langem auf die Erstellung von Modellen um bestimmte Phänomene zu studieren. Traditionelle mathematische Methoden wie Differenzialgleichungen werden dabei seit Jahrzehnten als Hauptwerkzeug zur Analyse und zum Entwurf angewandt, aber auch um Vorhersagen zu Systemen verschiedener Bereiche treffen zu können. Die zunehmende Komplexität jedoch, vor allem die der von Menschen geschaffenen Systeme ab dem 20ten Jahrhundert stellten dieses Verfahren vor unüberwindbare Hürden. Beispiele hierfür finden sich überall um uns herum: Computer- und Kommunikationsnetze, Luftverkehrkontrollsysteme, automatisierte Fertigungssysteme usw.[1]

Das Aufkommen digitaler Computer ermöglichte die Durchführung komplizierter Berechnungen und so wundert es nicht, dass die Simulation zu einer der frühesten Anwendungen in der Computertechnologie gehört. Die Computersimulation bietet alternative Methoden zur Analyse sowohl natürlicher als auch künstlicher Systeme und erlaubt das Lösen von Problemen mit einem bisher nicht gekannten Grad an Komplexität. Computer-simulierte Modelle bieten darüber hinaus noch weitere, entscheidende Vorteile: Sie können sicher und einfach in einer kosten-effizienten, risikofreien Umgebung ausgeführt werden und das beliebig oft.[2]

Ziel dieser Arbeit ist die Nahebringung von Simulationen dynamischer Systeme und deren Implementierung mit der Programmiersprache Java. Im Rahmen dieser Seminararbeit beschränke ich mich auf die diskrete Ereignissimulation die in einem breiten Bereich Anwendung findet, vor allem aber in der Logistik, der Produktion und bei großen Aufkommen von Personen oder Gütern (Autobahn-Mautstellen, öffentliche Verkehrssysteme, Bahnhöfe etc.).[3]

1.2 Überblick

Zur Klärung der Grundlagen werden im nächsten Abschnitt wichtige Begriffe der Modellierung und Simulation erläutert. Darauf aufbauend wird im dritten Abschnitt die diskrete Ereignissimulation behandelt mit Fokus auf deren Funktionsweise, Bestandteile und die zwei wichtigsten Modellierungsstile.

Den Hauptteil der Arbeit nimmt der vierte Abschnitt ein, in dem ein Einblick in das Java-Framework DESMO-J gegeben wird, welches zur Implementierung diskreter Ereignissimulationen entwickelt wurde. Die wichtigsten Aspekte wie dessen Aufbau und Funktionsweise werden vorgestellt und erläutert.

Abschließend werden die Ergebnisse dieser Arbeit zusammengefasst und ein Fazit gezogen. Im Anhang finden sich, zur praktischen Demonstration der im Abschnitt 4 gewonnen Erkenntnisse, jeweils ein Fallbeispiel für die zwei Modellierungsstile, sowie der komplette Quellcode dazu.

[1] vgl. [Wai09] und [CL08]
[2] vgl. [PK05]
[3] vgl. [Wai09] und [CL08]

2 Begriffe und Grundlagen

Die folgende Einführung bereitet auf den nächsten Abschnitt Diskrete Ereignissimulation vor indem es die nötigen Begriffe und Grundlagen bzgl. dem Thema Simulation näherbringt.

2.1 System

> „A combination of components that act together to perform a function not possible with any of the individual parts." [Rad97]
>
> „According to systems theory, a system is a natural or artificial entity, real or abstract, that is a part of a given reality constrained by an environment. It can be seen as an ordered set of related objects that evolve through different activities, interacting to achieve a goal." [Wai09, S. 24]

Ein System ist eine abstrahierte, also auf das Wesentliche beschränkte Teilmenge der Realität, deren Untersuchung Fragen beantworten soll. Es kann materielle (z. B. Objekte aus der realen Welt) als auch immaterielle Aspekte (z. B. Ideen) umfassen. Wie der Abbildung 1 zu entnehmen ist hat ein System eine *Systemstruktur*, d. h. es besteht aus einer Zahl von bestimmten und klar identifizierbaren Komponenten, die wiederum Systeme - auf einer niederen Ebene - sein können. Die Kombination der Komponenten ergibt eine *Systemidentität* und das Entfernen von Komponenten führt zur Nichterfüllung des ursprünglichen Zwecks und damit zur Änderung der Identität. Die Wirkungsbeziehungen (Kopplungen) unter den Komponenten bestimmen die Funktionalität und damit den *Systemzweck*. Ein System ist also eine Verknüpfung von Komponenten, die zusammen einem Zweck gerecht werden, dem die individuellen Komponenten allein nicht gerecht werden können.[4]

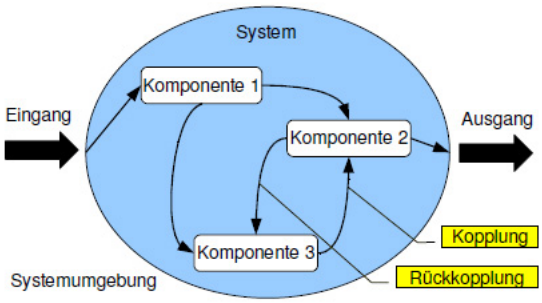

Abbildung 1: Aufbau eines Systems
(entnommen aus [KFA11, S. 3])

[4] vgl. [PK05, S. 4] und [KFA11, S. 3]

In diesem Kontext werden folgende Begriffe verwendet:

Begriff	**Beschreibung**
Elemente/Objekte/ Komponenten	Bestandteile des Systems, die nicht weiter zerlegbar sind bzw. nicht weiter zerlegt werden soll (können evtl. selbst Systeme sein).
Eigenschaften/Attribute	Variablen, die den Zustand des Systems speichern.
Systemzustand	Wird durch die Werte einer Menge von Variablen zu einem bestimmten Zeitpunkt bestimmt.
Systemverhalten	Ist der Vektor der Systemzustände über einen Zeitraum hinweg.
Systemkomplexität	Hängt von der Anzahl der Zustandsvariablen und der Dichte deren Verbindungen ab.
Systemgrenzen	Grenzen, die das System von der Außenwelt abschotten.
Offene Systeme	Haben mindestens eine Interaktion mit der Außenwelt, z. B. eine Fabrik mit Material, Aufträgen und Energie als Input und Produkten und Abfall als Output.
Geschlossene Systeme	Haben keine Interaktionen mit der Außenwelt, z. B. ein Aquarium.
Statische Systeme	Enthalten keinen Bezug zur Zeit.
Dynamische Systeme	Zeigen zeitabhängiges Verhalten.

Tabelle 1: Begriffe im Kontext von Systemen
(auf Basis von [PK05, S. 5])

2.2 Modell

> „Modelle sind materielle oder immaterielle Systeme die andere Systeme in solcher Weise repräsentieren als dass experimentelle Manipulation der modellierten Strukturen und Zustände möglich werden"[Nie77, S. 57]
>
> „Models can help us improve understanding of system behaviour and the effects of interactions among components. We use abstraction and idealization to map real world systems to models. A model is therefore always a simplification of its original."[PK05, S. 6]

Ein Modell ist die Repräsentation eines Systems innerhalb eines bestimmten Experimentrahmens, z. B. muss ein Modell einen Zweck oder eine Menge von Fragen haben, die es beantworten kann. Durch die *Modellbildung* wird das zu untersuchende System so weit vereinfacht bis ein Modell entsteht, welches eine vereinfachte Handhabung ermöglicht, jedoch dem ursprünglichem System in irgendeiner Weise ähnelt. Die Vereinfachung muss ständig im Einklang mit der Zielsetzung des Modells vollzogen werden.

Modelle können nach Art ihrer *Zustandsübergänge* oder nach ihrer *Zielsetzung* unterteilt werden. Im ersteren Fall kann zwischen *statischen* und *dynamischen* Modellen unterschieden werden. Das Hauptaugenmerk dieser Arbeit liegt auf den dynamischen Modellen, die sich weiter in *diskrete* und *kontinuierliche* unterteilen. Beide Arten können weiter in jeweils *deterministische* bzw. *stochastische* Modelle unterteilt werden. In der Tabelle 2 werden diese Begriffe erklärt.

Art	Beschreibung
statisch	Keine Zustandsänderungen/Kein Zeitverlauf.
dynamisch	Zustandsänderungen über einen Zeitverlauf.
diskret	Zustandsänderungen ereignen sich an diskret verteilten Zeitpunkten auf der Zeitachse (z. B. Wartezimmer: Patient betritt/verlässt Raum). Man spricht von *Ereignissen* (*events*), die jeweils zu einem *Ereigniszeitpunkt* (*event time*) eintreten.
kontinuierlich	Zustandsänderungen erfolgen andauernd statt an diskret verteilten Zeitpunkten (z. B. Bewegung der Planeten im Sonnensystem).
deterministisch	Alle Zustandsänderungen sind eindeutig festgelegt. Z. B. geht man für die Simulation in der Infektionsepidemiologie davon aus, dass Ereignisse (Infektionen) einer Influenzapandemie bei großer Population so häufig vorkommen, dass ein Zufall vernachlässigt werden kann (vgl. [Eic11]).
stochastisch	Zustandsänderungen ereignen sich durch zufällige Ereignisse, die durch *Wahrscheinlichkeitsverteilungen* beschrieben werden.

Tabelle 2: Modellarten
(auf Basis von [PK05, S. 6,7] und [Mül11, S. 3])

In Hinsicht auf die Zielsetzung kann ein Modell auf der anderen Seite zum Zweck der *Prognose*, der *Erklärung*, des *Entwurfs* bzw. der *Planung* oder der *Optimierung* erstellt werden.

Ein Modell besteht aus zwei verschiedenen Arten von Komponenten, welche in Tabelle 3 beschrieben werden.

Art	Beschreibung
statisch	Komponenten, die keine Änderungen am Systemzustand vornehmen können. Beispielsweise *Queues* (Warteschlangen), *Wahrscheinlichkeitsverteilungen* oder *Datenkollektoren*.
dynamisch	Komponenten, die Änderungen am Systemzustand vornehmen können. Z. B. Ereignisse eines Modells (z. B. Beginn der Verarbeitung eines Materials).

Tabelle 3: Komponentenarten eines Modells
(auf Basis von [PK05, S. 109])

2.3 Simulation

> „Simulation is the process of describing a real system and using this model for experimentation, with the goal of understanding the system's behaviour or to explore alternative strategies for its operation." [Sha75]
>
> „Simulation ist das Nachbilden eines dynamischen Prozesses in einem System mit Hilfe eines experimentierfähigen Modells, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind." [Ver93]

Bei einer Simulation wird versucht das Verhalten eines Systems in einem Modell zu imitieren um es zu analysieren. Aus den gewonnen Erkenntnissen werden Rückschlüsse auf das Originalsystem gezogen, ohne jedoch dieses zu beeinflussen.[5]

Bei der *Computersimulation* handelt es sich um die Art der Simulation die mit Hilfe digitaler Computer ausgeführt wird. Das System wird durch formale Methoden beschrieben was zu einem mathematischen Modell führt. Die Implementierung dieses Modells am Computer ist der *Simulator* bzw. das *Rechenmodell*, welches anschließend experimentell untersucht wird.[6]

Für gewöhnlich werden bei Computersimulationen dynamische Modelle betrachtet, was also den Zeitaspekt mit einbezieht (vgl. Tabelle 2). An dieser Stelle muss zwischen drei Zeitbegriffen unterschieden werden, wie in Tabelle 4 dargestellt.

Zeitbegriff	**Beschreibung**
Realzeit	Die Zeit im realen System mit den Zeitpunkten an denen sich im realen System Zustandsänderungen ereignen.
Modellzeit	Nachbildung der Realzeit anhand einer Variable.
Simulationszeit	Rechendauer eines Computers für die Berechnung eines Simulationsschrittes oder der gesamten Simulation eines Modells. Ist die Simulationszeit kleiner oder gleich der Realzeit so spricht man von *Echtzeitsimulation.*

Tabelle 4: Die drei Zeitbegriffe
(auf Basis von [KFA11, S. 4])

Simulationen können im Gegensatz zu analytischen Modellen keine Lösung garantieren, da letztere eine Reihe von Gleichungen aufstellen für die eine Lösung erzielt werden kann [7]. Simulationen können dagegen nur schrittweise Wertveränderungen von Zustandsvariablen verfolgen, während der Modellierer entscheiden muss welche Zustandsvariablen betrachtet werden sollen und wie detailliert dies geschehen soll. Daher sollte die Simulation nur dann zur Lösungsfindung angewandt werden, wenn analytische Modelle darin scheitern relevante Bereiche eines komplexen Systems abzudecken, z. B. wegen ungeeigneter Vereinfachung komplexer Verbindungen oder komplexen Verhaltens.[8]

[5] vgl. [KFA11, S. 3]
[6] vgl. [KFA11, S. 4]
[7] vgl. [Sto11]
[8] vgl. [PK05, S. 9-11] und [Sto11]

3 Diskrete Ereignissimulation

3.1 Einführung

Bei der diskreten Ereignissimulation treten Zustandsänderungen zu diskreten Zeitpunkten ein. Allgemein werden diese Zustandsänderungen *Ereignisse* genannt. Die durch ein Ereignis resultierende Zustandsänderung wird in einer *Ereignisroutine* angegeben, die bei der Ereignisrealisierung ausgeführt wird. Zu unterscheiden ist an dieser Stelle zwischen *Zeitereignis* und *Zustandsereignis*. Ersteres hat einen bekannten Ereigniszeitpunkt der gleichzeitig der Auslöser ist. Zweiteres hat kein bekannten Ereigniszeitpunkt und wird durch Erfüllung bestimmter, vom Systemzustand abhängiger, Bedingungen aktiviert.[9] Darüber hinaus wird zwischen *externen* und *internen* Ereignissen unterschieden. Externe Ereignisse werden von außen angestoßen, z. B. durch Stromausfälle, während interne aus den Zustandsänderungen interner Modellentitäten hervorgerufen werden. [10]

Die Grundlage jeder diskreten Ereignissimulation ist die *Ereignisliste* oder auch *Ereigniskalender* genannt, welcher geordnet nach Zeitpunkt ihres Auftretens alle Ereignisse enthält. Deckt sich die aktuelle Modellzeit mit dem Zeitpunkt eines Ereignisses, wird dessen Ereignisroutine ausgeführt, was zu Zustandsänderungen des Systems oder zur Erstellung weiterer Ereignisse und deren Eintragung in die Ereignisliste führen kann. Abbildung 2 zeigt die Ausführung des Ereignisses E_0, worauf ein neues Ereignis E_3 zum Zeitpunkt t_3 eingeplant wird. Während der Abarbeitung eines Ereignisses vergeht keine Modellzeit, stattdessen schreitet die Modellzeit erst dann voran, wenn das Ereignis abgearbeitet worden ist. Die Modellzeit geht dann zum Zeitpunkt des nächstgelegenen Ereignisses in der Ereignisliste über, was in Abbildung 2 durch den Zeitsprung von t_0 zu t_1 dargestellt wird. Ereignisse zur gleichen Zeit werden *Parallelereignisse* genannt. Zur korrekten Ausführung dieser werden entweder für die Ereignisse definierte *Prioritäten* herangezogen oder vordefinierte Strategien angewandt, z. B. FCFS ("First come, first served").[11]

Typische Anwendungsbereiche der diskreten Ereignissimulation sind Systeme mit Warteschlangen, wo *Clients* auf Dienstleistungen von *Servern* warten müssen, da diese in ihrer Kapazität beschränkt sind. Ereignisse wie Ankunft eines neuen Clients oder Fertigstellung einer Dienstleistung werden als Ereignisse zu diskreten Zeitpunkten betrachtet.[12]

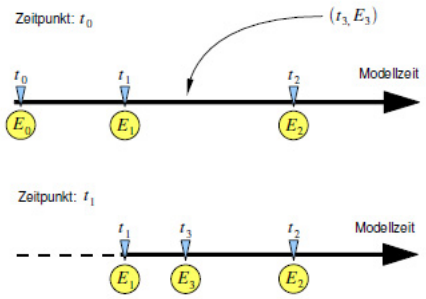

Abbildung 2: Zeitfortschritt bei Ereignisrealisierung von E_0
(entnommen aus [KFA11, S. 5])

[9] vgl. [KFA11, S. 5]
[10] vgl. [PK05, S. 27]
[11] vgl. [PK05, S. 26] und [KFA11, S. 6]
[12] vgl. [PK05, S. 24]

3.2 Elementare Bestandteile

Zur Implementierung eines Modells der diskreten Ereignissimulation werden einige Komponenten benötigt, die sich dem *Simulator* oder der Simulation selbst zuordnen lassen.

3.2.1 Simulator

Wichtige Bestandteile eines Simulators sind:

Bestandteil	**Beschreibung**
Ereignisliste/Ereigniskalender	Ereignislisten werden im Abschnitt vorher (Abschnitt 3.1) behandelt. Hinzuzufügen ist jedoch, dass der Simulator auch mehrere Listen verwalten kann.
Simulationsuhr	Zeigt die aktuelle Modellzeit, welche nicht kontinuierlich fortschreitet sondern sprunghaft von einem Zeitpunkt zum nächsten fortschreitet, da die Ereignisse selbst keine Modellzeit verbrauchen. Die Zeiteinheit kann dem zu untersuchenden System entsprechend gewählt werden.
Simulationssteuerung	Wird häufig in Form eines *Schedulers* realisiert, der u. a. die Verwaltung der Simulationsuhr und der Ereignisliste übernimmt. Die Simulationssteuerung bestimmt wiederholt das nächste Ereignis und setzt dementsprechend die aktuelle Modellzeit auf den Ausführungszeitpunkt des Ereignisses, worauf dessen Ereignisroutine aktiviert wird. Auch das Prüfen der *Abbruchbedingung* und das Starten der *Auswertungsmechanismen* gehört zu den Aufgaben der Simulationssteuerung.
Zufallszahlengeneratoren	Diese werden benötigt um variable Zeitintervalle und Häufigkeiten zu modellieren. Hierfür werden *Pseudozufallszahlen* verwendet, die zwar vielfältige Verteilungsarten ermöglichen, aber dennoch deterministisch reproduzierbar sind, um Simulationen exakt wiederholen zu können.

Tabelle 5: Bestandteile eines Simulators
(auf Basis von [KFA11, S. 6-7] und [PK05, S. 26, 28-29 und 31])

3.2.2 Simulation

Wichtige Bestandteile einer Simulation sind:

Bestandteil	Beschreibung
Statistik	Hierzu gehören alle statistischen Daten, die während der Simulation gesammelt und von dieser verwaltet werden, z. B. Daten zur Auslastung von Systemkomponenten wie z. B. der Server oder der Client-Queues und anderen Ressourcen. Auch Daten bzgl. der Nutzung von Zufallszahlgeneratoren und die Bestimmung von Kennwerten wie Mittelwert und Standardabweichung gehören dazu.
Abbruchbedingungen	Theoretisch kann eine diskrete Ereignissimulation ewig laufen, da Ereignisse neue Ereignisse produzieren können. Der Modellierer muss daher anhand einer Abbruchbedingung festlegen wann die Simulation beendet werden soll. Die Abbruchbedingung kann ein Zeitpunkt, ein statischer Wert oder das Erreichen eines festgelegten Systemzustands sein.
Auswertungsmechanismen	Die durch die Simulation enstandenen Daten werden durch Auswertungsmechanismen wie z.B. der *Ergebnis-* und *Ereignisroutinen* gesammelt und ausgewertet. Das Ergebnis sind Protokolle oder Berichte mit aufbereiteten statistischen Daten und Informationen. Sie ermöglichen eine Überprüfung hinsichtlich des Untersuchungsziels und helfen Fragen zu beantworten um so Rückschlüsse auf das Systemverhalten ziehen zu können.

Tabelle 6: Bestandteile einer Simulation
(auf Basis von [KFA11, S. 7-8] und [PK05, S. 26, 28-29 und 31])

3.3 Modellierungsstile

In der diskreten Ereignissimulation dominieren zwei *Modellierungsstile* bzw. *Modellierungsansätze*: der *ereignisorientierte* (*event oriented*) und der *prozessorientierte* (*process oriented*). Daneben gibt es noch weitere wie z. B. den transaktions-basierenden (GPSS) oder den aktivitätsorientierten Modellierungsstil (ECSL). Diese werden jedoch aufgrund ihrer relativ geringen Bedeutung und Verbreitung außer Acht gelassen.[13]

3.3.1 Ereignisorientiert

Ereignisorientierte Frameworks sind älter als prozessorientierte und konzentrieren sich auf sämtliche Änderungen aller betroffenen Entitäten zu einem bestimmten Zeitpunkt. Die Ereignisse werden nach dem Zeitpunkt ihres Auftretens gebündelt, können ansonsten jedoch unabhängig und ohne Bezug zueinander sein. Ein Bündel bildet somit das Verhalten eines Systems zu einem

[13] vgl. [PK05, S. 97]

bestimmten Zeitpunkt, indem jedes Ereignis des Bündels gleichzeitig, im Sinne von zur gleichen Modellzeit, ausgeführt wird. Die Ereigniszeitpunkte werden mit Hilfe von Zufallszahlengeneratoren modelliert. [14]

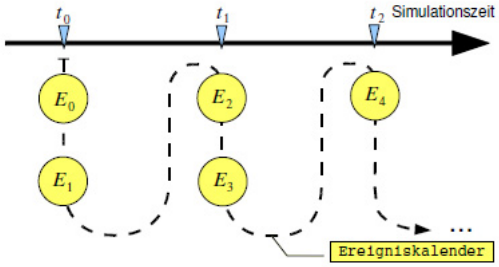

Abbildung 3: Ablauf eines ereignisorientierten Modells
(entnommen aus [KFA11, S. 9])

Um ein ereignisorientiertes Modell zu entwerfen müssen zunächst sämtliche notwendigen Objekte und deren Eigenschaften erfasst werden. Dazu bedarf es einer „Vogelperspektive" aus der alle Zustände, Ereignisse und Änderungen von Entitäten und deren Interaktionen untereinander erkennbar sind, was jedoch schnell zum Verlust der Übersicht bei komplexen Systemen führen kann[15]. Alle Zustandsänderungen, die hervorgerufen wurden durch Ereignisse, welche dieselben Entitäten zum selben Zeitpunkt betreffen, können zu Ereignis-Klassen eines bestimmten Typs zusammengefasst werden.[16]

Im Gegensatz zum prozessorientierten Modellierungsstil, wo die Beschreibung der Systemstruktur und des -verhaltens verflochten sind ist bei dem ereignisorientierten Ansatz eine klare Trennung zwischen der Spezifikation der Systemstruktur und des Systemverhaltens möglich. [17]

3.3.2 Prozessorientiert

Der prozessorientierte Ansatz hat sich über die letzten 20 Jahre zum meistangewandten Modellierungsstil entwickelt. Das Hauptmerkmal dieser Modellierungsart sind hauptsächlich die *Prozesse*, die sämtliche Aktivitäten und Attribute einer Entität vereinen und damit den *Lebenszyklus* der Entität darstellen. Ein Prozess kann sich abwechselnd entweder in der *aktiven* oder in der *passiven* Phase befinden. Ist ein Prozess aktiv so führt er modellierte Aktionen der entsprechenden Entität aus, die zwar Modellzeit verbrauchen können, deren hervorgerufene Zustandsänderungen am System jedoch unverzögert umgesetzt werden. Nach jeder Verschiebung der Modellzeit gibt der Prozess die Kontrolle an den Scheduler und erhält sie anschließend wieder. Somit ist sichergestellt, dass Prozesse, deren aktive Phase im Rahmen einer aktiven Phase anderer Prozesse beginnen, ebenfalls ausgeführt werden. Dies führt zu quasi-simultanen Prozessabläufen, die bei einkernigen Prozessoren natürlich nicht parallel ausgeführt sondern auf deren sequentiellen Ausführungsthread abgebildet werden. Die Reihenfolge dieser Sequenzabfolge kann durch Maßnahmen wie Prioritätsvergaben kontrolliert werden.[18]

[14] vgl. [PK05, S. 108] und [KFA11, S. 10]
[15] vgl. [KFA11, S. 10]
[16] vgl. [PK05, S. 108] und [DES11b]
[17] vgl. [PK05, S. 108]
[18] vgl. [PK05, S. 98] und [Mül11, S. 41]

Abbildung 4 zeigt den Ablauf eines prozessorientierten Modells, in dem der Ereigniskalender vom Startzeitpunkt t_0 über die Zeitpunkte t_1, t_2 usw. verläuft und an den Zeitpunkten t_0 und t_2 quasi-simultan Prozesse anstößt.

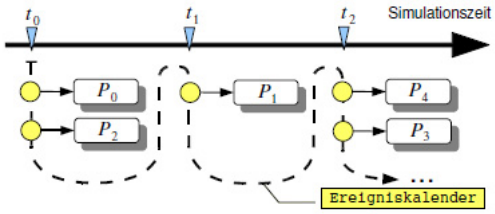

Abbildung 4: Ablauf eines prozessorientierten Modells
(entnommen aus [KFA11, S. 8])

Ein Prozess kann in die passive Phase gelangen wenn er absichtlich in den Wartezustand übergeht, was bedeutet, dass er selbst seine eigene Aktivierung in den Ereigniskalender für den Zeitpunkt einträgt an dem der Wartezustand vorbei sein wird. Andererseits kann er auch bewusst in die passive Phase übergehen ohne nächsten festgelegten Aktivierungszeitpunkt. Im Gegensatz zum ereignisorientiertem Modellierungstil wird ein Prozess jedoch in der Regel nie terminiert, sondern kann später durch externe Ereignisse wie Eintritt eines bestimmten Zeitpunktes nach einem Wartezustand (Zeitereignis), oder durch andere Prozesse (Zusandsereignis) reaktiviert werden. Der Prozess macht dann an der Stelle weiter, an der er vorher in die passive Phase gewechselt ist.[19]

Der prozessorientierte Modellierungsansatz erlaubt es dem Modellierer mit Hilfe der Prozesse zu jeder Zeit einen Einblick in Teile des Systemzustandes zu bekommen. Der Modellierer braucht keinen Gesamtüberblick des Systems, sondern muss nur um die Beziehungen zwischen allen Entitäten und deren Verhaltensbeschreibungen bzw. aktuelle Zustände wissen.[20]

3.3.3 Vergleich und Beurteilung

Beide Modellierungsstile haben viele Gemeinsamkeiten, beispielsweise nutzen beide die Ereignisliste und verarbeiten Ereignisse auf ähnliche Weise. Trotzdem unterscheiden sich beide Ansätze stark in der Modellierung und Implementierung eines Modells. Während ereignisorientierte Modelle nur einzelne Ereignisse kennen, die zu einem bestimmten Zeitpunkt aus der Ereignisliste heraus ausgeführt werden und dabei eine Referenz auf die betreffende Entität besitzen, fassen prozessorientierte Modelle alle Ereignisse einer Entität zu einem Prozess zusammen.[21]

Der Modellierer eines ereignisorientierten Modells muss also eine „Vogelperspektive" einnehmen, aus der alle Ereignistypen und Zustandsänderungen ersichtlich werden, denn die Ereignisse und damit die Zustandsänderungen sind „zerstreut" zu erfassen. Auf der anderen Seite muss der Modellierer eines prozessorientierten Modells eine „Froschperspektive" einnehmen aus der stets nur ein Ausschnitt des Systems ersichtlich wird. Diese begrenzte Sichtweise macht jedoch die Modellierung komplexer Systeme übersichtlicher und einfacher erweiterbar.[22]

[19] vgl. [PK05, S. 99-100], [KFA11, S. 8-9] und [Mül11, S. 41]
[20] vgl. [KFA11, S. 9]
[21] vgl. [PK05, S. 117] und [KFA11, S. 10]
[22] vgl. [KFA11, S. 10]

Obwohl sich beide Konzepte sehr unterscheiden ist doch eine Überführung möglich, z. B. ließen sich einzelne Aktionen eines Prozesses der prozessorientierten Modellierung als einzelne Ereignisse in das ereignisorientierte Modell übertragen. Aus Gründen der Übersicht ist aber oft genau das Gegenteil sinnvoll, nämlich die Zusammenfassung einzelner Ereignisse zu Prozessen. In einigen Fällen ist jedoch die Verwendung von Ereignissen anstatt Prozessen sinnvoller, z. B. wenn ein Prozess nur eine Aktion umfassen würde. Eine Kombination aus beiden Ansätzen kann aber auch in Betracht gezogen werden. Ein Beispiel hierfür ist eine Maschine, bei der ein Defekt eintritt. Die Entität Maschine selbst hat einen Prozess, der einen Defekt nicht zwangsläufig einschließt. Die Erfassung des Defekt-Ereignisses wird an dieser Stelle aus prozessorientierter Sicht problematisch, denn die Erstellung einer eigenen Entität mit zwecks hierfür definiertem Prozess ist umständlich und erzeugt aufgrund des Prozessobjekts Overhead. Ein einzelnes Ereignis würde das Problem lösen. Es ist also stets eine Frage der Problemstellung und der geschickten Modellierung welche Möglichkeit zu wählen ist.[23]

4 DESMO-J

4.1 Einführung

DESMO-J[24] ("Discrete-Event Simulation Modelling in Java") ist ein kostenloses, unter der Apache-Lizenz[25] stehendes, objektorientiertes Framework für die Programmiersprache Java zur Implementierung diskreter Ereignissimulationen. Das Framework wurde von 1999 an ursprünglich zu Lehrzwecken an der Universität Hamburg unter der Leitung von Professor Bernd Page entwickelt. Es ist jedoch in der Funktionalität stetig gewachsen, sodass der Leistungsumfang inzwischen über die Grundanforderungen an ein Framework für diskrete Ereignissimulation hinausgeht (vgl. Abschnitt 3.2 auf Seite 9). Neben der Simulation selbst komplexer Modelle erlaubt es auch die Visualisierung durch GUI-Elemente und Grafiken. Das Framework liegt aktuell in der Version 2.0.0 vor (Stand 16.01.2011).

4.2 Struktur und Funktionalität

Das DESMO-J-Framework ist eine Mischung aus *Black-Box-* und *White-Box-Komponenten*. Erstere sind solche, deren Funktionalität klar definiert und eingegrenzt ist und die nur wenig Modifikationsaufwand und Programmierkenntnisse erfordern. Die Funktionalität solcher Komponenten sind oder sollen dem Anwender verborgen bleiben. Letztere dagegen sind abstrakter und erfordern gute Kenntnisse in der jeweiligen Programmiersprache, da ihre Funktionalität vom Anwender implementiert oder erweitert werden muss.[26]

Abbildung 5 zeigt vereinfacht die Klassenhierarchie des DESMO-J-Frameworks mit Hervorhebung der Black-Box- und White-Box-Komponenten, wobei letztere als Hot-Spot-Komponenten aufgeführt werden.

Die zwei großen Oberklassen sind *Schedulable* und *Reportable*. Aus der Sicht des Modellierers sind sämtliche Unterklassen von *Schedulable* dynamisch, da sie den Modellzustand ändern können, z. B. durch Ereignisse. Unterklassen von *Reportable* sind dagegen statisch, denn sie können

[23] vgl. [PK05, S. 134] und [KFA11, S. 10]
[24] vgl. [DES11c]
[25] vgl. [The11]
[26] vgl. [PK05, S. 264]

keine Änderungen bewirken. Andererseits können sie Informationen über sich veröffentlichen, die gesammelt und später in Report-Dateien angezeigt werden.[27]

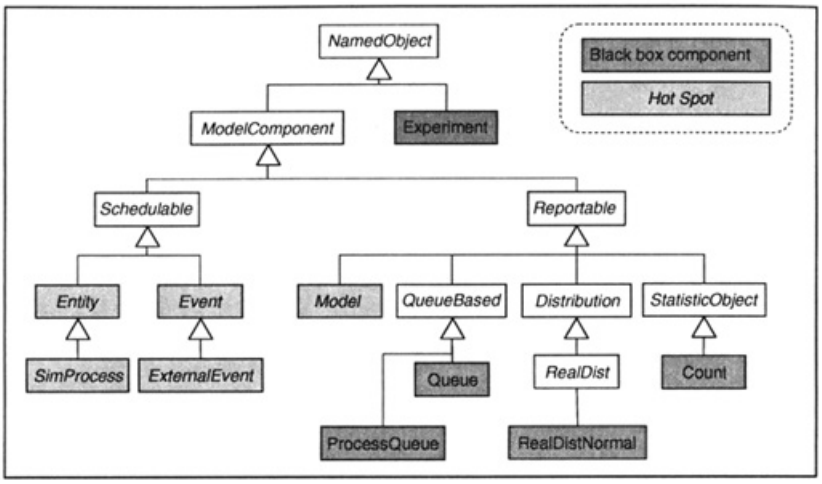

Abbildung 5: Die vereinfachte Klassenhierarchie von DESMO-J mit Darstellung der Black-Box- und White-Box-Komponenten (Hot Spots)
(entnommen aus [PK05, S. 266])

Sämtliche elementaren Bestandteile einer ereignisorientierten Simulation (vgl. Abschnitt 3.2 auf Seite 9) gehören zu den Black-Box-Komponenten, da sie nur instanziiert und parametriert werden müssen, während ihre Funktionalität festgelegt ist. Elemente des Simulators werden im DESMO-J-Framework als Teilfunktionalität in der Klasse `Model` untergebracht während die Implementierung selbst gekapselt in der Klasse `Experiment` erfolgt. Elemente der Simulation befinden sich ebenfalls in der Klasse `Experiment`. Darüber hinaus bietet das DESMO-J-Framework als Black-Box-Komponenten Konstrukte zur Modellierung von Queues und Zufallsprozessen - letztere basieren auf unterschiedlichen Wahrscheinlichkeitsverteilungen (distributions) - sowie Konstrukte zur Sammlung von statistischen Daten durch Datenkollektoren (data collectors).[28]

Über White-Box-Komponenten wird die Kernfunktionalität der Black-Box-Komponenten um modellspezifische Bestandteile wie z. B. Entitäten oder Ereignisse erweitert. Zu diesem Zweck bietet das Framework die abstrakten, generischen Klassen *Entity*, *Event*, *SimProcess* und *Model*, die in Tabelle 7 auf der nächsten Seite beschrieben werden.[29] Die Klasse *Model* nimmt eine besondere Stellung ein, da sie den Rahmen des Modells vorgibt, in dem sich alles abspielt. Die Grundstruktur einer Unterklasse von *Model* wird in Listing 1 auf der nächsten Seite abgebildet.

[27] vgl. [PK05, S.265]
[28] vgl. [PK05, S. 265] und [DES11a, Klassen Model und Experiment]
[29] vgl. [PK05, S. 265]

Klasse	Beschreibung
Entity	Repräsentiert eine Entität, welche immer passiv ist. Wird bei ereignisorientierter Modellierung verwendet.
Event	Repräsentiert ein Ereignis, welches eine Entität betrifft. Wird bei ereignisorientierter Modellierung verwendet.
SimProcess	Repräsentiert eine Entität, die aktiv sein kann. Wird bei prozessorientierter Modellierung verwendet. *SimProcess* enthält im Gegensatz zu *Entity* die Methode *lifeCycle()*, welche das Verhalten der Entität definiert ohne auf Ereignisse (*Events*) angewiesen zu sein.
Model	Die Klasse *Model* hat eine besondere Rolle, denn sie ist der Rahmen des Modells, in dem sich alles ereignet. Auf Grund dessen besteht sie sowohl aus Black-Box- als auch aus White-Box-Komponenten.

Tabelle 7: White-Box-Klassen
(auf Basis von [DES11a])

```
1  public class ModelExample extends Model
2  {
3      //Wahrscheinlichkeitsverteilungen
4      private RealDistExponential dist1;
5      private RealDistUniform dist2;
6
7      //Queues
8      protected Queue<ExampleEntity> entitiyQueue;
9      protected ProcessQueue<ExampleProcess> processQueue;
10
11     public ModelExample(Model owner, String modelName, boolean showInReport
           , boolean showInTrace)
12     {
13         super(owner, modelName, showInReport, showInTrace);
14     }
15
16     public String description()
17     {
18         return "beschreibung";
19     }
20
21     public void doInitialSchedules()
22     {
23         //Ereignisorientiert:
24         InitialEreignis ereignis = new InitialEreignis(this, "
               InitialEreignis", true);
25         ereignis.schedule(new TimeSpan(0));
26
27         //Prozessorientiert:
28         new InitialProzess(this).activate(new SimTime(this.dist2.sample()))
               ;
29
30     }
31
32     public void init()
```

```
33      {
34          dist2= new RealDistUniform(this, "ServiceTimeStream", 3.0, 7.0,
                true, false);
35          dist1= new RealDistExponential(this, "TruckArrivalTimeStream", 3.0,
                 true, false);
36          dist1.setNonNegative(true);
37          entitiyQueue = new Queue<ExampleEntity>(this, "EntityQueue", true,
                true);
38          processQueue = new ProcessQueue<ExampleProcess>(this, "ProcessQueue
                ", true, true);
39      }
40
41      public static void main(java.lang.String[] args)
42      {
43          ModelExample model = new ModelExample(null, "ModelExample", true,
                true);
44
45          Experiment exp = new Experiment("ExampleExperiment");
46          model.connectToExperiment(exp);
47
48          exp.setShowProgressBar(true);
49          exp.stop(new TimeInstant(1500, TimeUnit.MINUTES));
50          exp.tracePeriod(new TimeInstant(0), new TimeInstant(100, TimeUnit.
                MINUTES));
51          exp.debugPeriod(new TimeInstant(0), new TimeInstant(50, TimeUnit.
                MINUTES));
52          exp.start();
53          exp.report();
54          exp.finish();
55      }
56  }
```

Listing 1: Grundstruktur einer *Model*-Unterklasse

Im Folgenden wird genauer auf die Grundstruktur einer *Model*-Unterklasse eingegangen. Bei der Implementierung müssen im Konstruktor (Zeile 13) wichtige Parameter an die Oberklasse weitergegeben werden, deren Beschreibung in Tabelle 8 erfolgt. In Tabelle 9 auf der nächsten Seite werden die restlichen Bestandteile der Unterklasse beschrieben.

Parameter	**Beschreibung**
Model owner	Eine Referenz auf das Modell, zu dem das aktuelle Modell gehört. Ermöglicht Verschachtelung von Modellen.
String modelName	Der Name des Modells.
showInReport	Legt fest, ob eine Report-Datei zu diesem Modell generiert werden soll.
showInTrace	Legt fest, ob Trace-Meldungen bzgl. dieses Modells erfolgen sollen, die später in der Trace-Datei aufgeführt werden.

Tabelle 8: Konstruktor-Parameter einer *Model*-Unterklasse
(auf Basis von [DES11a, Klasse Model])

Zeile	Beschreibung
4-5	Definition benötigter Variablen. In diesem Fall werden zwei verschiedene Wahrscheinlichkeitsverteilungen und zwei verschiedene Queues definiert. Für Entities und für SimProcesses gibt es eigene Queue-Typen.
11-14	Der Konstruktor. Siehe Tabelle 8.
16-19	Die Methode *description()* liefert die Beschreibung des Modells. Die Methode ist wichtig für den Report-Generator, da die Beschreibung später in der Report-Datei aufgeführt wird.
21-30	Die Methode *doInitialSchedules()* sorgt dafür, dass die Simulation ins Rollen kommt. Es bedarf dazu eines Initialereignisses oder eines Initialprozesses.
32-39	Die Methode *init()* ist zuständig für die zu treffenden Vorkehrungen, d. h. hier werden die Variablen gesetzt, die Queues initialisiert etc.
41-55	In der Methode *main()* werden das Modell und das Experiment erstellt und anschließend verbunden. Es wird eine Endbedingung festgelegt (*exp.stop(...)*), in diesem Fall wird das Experiment nach 1.500 Minuten unterbrochen. Es sind aber auch Bedingungen möglich. Daraufhin wird das Experiment gestartet (*exp.start()*) und nachdem das Experiment die Endbedingung erfüllt hat kommt die Applikation wieder zurück und führt *exp.report()* aus, also den Report-Generator, der die Report-Dateien erstellt. Abschließend erfolgen Aufräumarbeiten (*exp.finish()*). Die *main()*-Methode muss zwar nicht in dieser Klasse implementiert werden, es empfiehlt sich jedoch, da die Model-Klasse der Kern eines Experiments ist.

Tabelle 9: Bestandteile einer Modell-Unterklasse
(auf Basis von [DES11a, Klasse Model])

DESMO-J trennt strikt zwischen Modell und Simulation, wie Abbildung 6 auf der nächsten Seite zeigt. Zur Modellierung gehören die erwähnten White-Box-Komponenten, die das Modell definieren, aber auch die Black-Box-Konstrukte wie Queues oder Zufallsprozesse sind für die Modellierung entscheidend und werden daher ebenfalls dem Modell zugeordnet. Typische Black-Box-Elemente werden dagegen isoliert und der Simulation zugerechnet und bleiben bis auf die Implementierung in Form von Instanzierung und Parametrierung - meist in der Klasse *Model* - unangetastet. Beide Bereiche müssen separat implementiert und anschließend in der `main()`-Methode durch die Methode `model.connectToExperiment(Experiment exp)` verbunden werden.[30]

[30] vgl. [PK05, S. 265] und [SIM11]

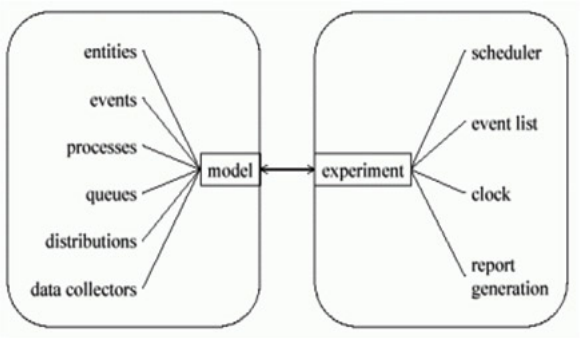

Abbildung 6: Trennung zwischen Modell und Experiment im DESMO-J-Framework (entnommen aus [SIM11])

Die Modellierung unter DESMO-J kann in folgende Schritte unterteilt werden[31]:

- Auswahl benötigter Black-Box-Komponenten wie Datenkollektoren, Verteilungsarten oder Queues.
- Implementierung dynamischer White-Box-Komponenten durch Ableitung und Anpassung geeigneter Klassen wie *SimProcess*, *Entity* oder *Event*.
- Implementierung der Funktionalität auf höchster Ebene in der *Model*-Unterklasse, z. B. Instanzierung und Parametrierung von Modellkomponenten oder Einplanung aktiver Prozesse im Scheduler.

4.2.1 Ereignisorientiert

Um ein ereignisorientiertes Modell mit DESMO-J zu implementieren müssen die White-Box-Klassen *Entity* und *Event* abgeleitet werden. Die Grundstruktur einer Unterklasse von Entity wird in Listing 2 auf der nächsten Seite und die einer Unterklasse von *Event* in Listing 3 auf der nächsten Seite dargestellt.[32]

Unterklassen von *Entity* repräsentieren Entitäten mitsamt der benötigten Attribute und Unterklassen von *Event* definieren das Verhalten von Ereignissen mittels der zu definierenden Methode *eventRoutine(Entity who)*. Ein Ereignis bezieht sich immer auf eine Entität, daher muss eine Verbindung zwischen beiden hergestellt werden, was über den Übergabeparameter `Entity who` geschieht. Unterklassen beider Arten haben Zugriff auf die Ereignisliste obwohl diese an den Scheduler und damit an die Klasse *Experiment* gebunden ist. Auf diese Weise sind Objekte der beiden Unterklassen in der Lage sich selbst in die Ereignisliste einzuplanen. Scheduler-Methoden für diesen Zweck der Klasse *Event* werden in Tabelle 10 auf der nächsten Seite beschrieben. Die Scheduler-Methoden der Klasse *Entity* sind analog dazu aufgebaut mit dem Unterschied, dass statt dem Übergabeparameter *Entity who* ein *Event what* verlangt wird, um die Verbindung zwischen Ereignis und betreffender Entität herzustellen. Objekte der Klasse *SimTime* repräsentieren Zeitpunkte einer Simulation basierend auf einem Wert vom Typ *double*.[33]

[31] vgl. [PK05, S. 266]
[32] vgl. [PK05, S. 268]
[33] vgl. [PK05, S. 268] und [DES11a, Klassen Event und Entity]

Methode	Beschreibung
schedule(Entity who, SimTime when)	Plant ein Ereignis für die betreffende Entität *who* zu einer bestimmten Modellzeit *when* ein.
scheduleAfter(Schedulable after, Entity who)	Plant ein Ereignis für die betreffende Entität *who* nach dem Ereignis, welches die Entität *after* betrifft, ein.
scheduleBefore(Schedulable before, Entity who)	Plant ein Ereignis für die betreffende Entität *who* vor dem Ereignis, welches die Entität *before* betrifft, ein.
cancel()	Entfernt das betreffende Ereignis aus der Ereignisliste.

Tabelle 10: Scheduler-Methoden der Klasse *Event*
(auf Basis von [PK05, S. 268-272] und [DES11a, Klassen Entity und Event])

```
1  public class ExampleEntity extends Entity
2  {
3      protected Eigenschaft eigenschaft1;
4
5      public ExampleEntity(Model owner, String name, boolean showInTrace)
6      {
7          super(owner, name, showInTrace);
8      }
9
10     public Eigenschaft getEigeschaft1()
11     {
12         return this.eigenschaft1;
13     }
14
15     public void setEigenschaft1(Eigenschaft wert)
16     {
17         this.eigenschaft1 = wert;
18     }
19 }
```

Listing 2: Grundstruktur einer *Entity*-Unterklasse

```
1  public class ExampleEvent extends Event<ExampleEntity>
2  {
3      public ExampleEvent(Model owner, String name, boolean showInTrace)
4      {
5      super(owner, name, showInTrace);
6      }
7
8      public void eventRoutine(ExampleEntity entity)
9      {
10         entity.setEigenschaft1(new Eigenschaft("wert"));
11         //...
12     }
13 }
```

Listing 3: Grundstruktur einer *Event*-Unterklasse

4.2.2 Prozessorientiert

Anders als bei der Implementierung ereignisorientierter Modelle wird bei der Implementierung prozessorientierter Modelle für die Abbildung der Entität und der zugehörigen Ereignisse nur eine Klasse benötigt, die Klasse *SimProcess.* Auch sie dient dabei als abstrakte Oberklasse die zur Verwendung erst durch eine Unterklasse implementiert werden muss. Ihre Unterklassen können sowohl die für die Entität relevanten Attribute als auch die Zustandsänderungen, die durch Ereignisse mit Bezug zu dieser Entität entstehen, enthalten. Die Entität und deren Ereignisse werden also zu einem Prozess zusammengefasst. Sämtliche Ereignisse werden in der Methode *lifeCycle()* gebündelt, was aber nicht bedeutet, dass die Methode vom Anfang bis zum Ende durchlaufen muss. Wie bereits im Abschnitt 3.3.2 auf Seite 11 beschrieben wird durchläuft ein Prozess Phasen in denen er aktiv oder passiv ist. Hierfür bietet *SimProcess* jeder Unterklasse entsprechende Scheduler-Methoden, die in Tabelle 11 erklärt werden. Wurden alle Anweisungen der *lifeCycle()*-Methode ausgeführt wird der Prozess dauerhaft passiv. In Listing 4 wird die Grundstruktur einer *SimProcess*-Unterklasse dargestellt.[34]

Methode	**Beschreibung**
passivate()	Unterbricht den Prozess für eine unbestimmte Zeit.
hold(SimTime dt)	Unterbricht und verzögert einen Prozess um *dt* Modellzeiteinheiten. In *dt* Modellzeiteinheiten wird der Prozess also wieder aktiv.
activate(SimTime dt)	Aktiviert einen Prozess nach *dt* Modellzeiteinheiten.
activateAfter(Schedulable after)	Aktiviert einen Prozess unmittelbar nach dem Objekt *after,* einem Objekt einer Unterklasse von *Schedulable*, d. h. auch Ereignisse (Events) sind möglich.
activateBefore(Schedulable before)	Aktiviert einen Prozess vor dem Objekt *before*, einem Objekt einer Unterklasse von *Schedulable.*
reActivate(TimeSpan dt)	Reaktiviert einen Prozess innerhalb der Zeitspanne *dt* ausgehend von der aktuellen Modellzeit

Tabelle 11: Scheduler-Methoden der Klasse *SimProcess*

```
1  public class ExampleProcess extends SimProcess
2  {
3      protected Eigenschaft eigenschaft1;
4
5      public ExampleProcess(Model owner, String name, boolean showInTrace)
6      {
7          super(owner, name, showInTrace);
8      }
9
10     public void lifeCycle()
11     {
12         eigenschaft1 = new Eigenschaft("wert");
13         hold(new TimeSpan(10, TimeUnit.MINUTES));
14         eigenschaft1 = new Eigenschaft("wert2");
15         passivate();
```

[34] vgl. [PK05, S. 273] und [DES11a, Klassen SimProcess]

```
16          eigenschaft1 = new Eigenschaft("endWert");
17          //passiv für immer...
18      }
19  }
```

Listing 4: Grundstruktur einer *SimProcess*-Unterklasse

4.2.3 Kombination aus Prozessen und Ereignissen in einem Modell

DESMO-J erlaubt eine Kombination aus ereignisorientierter und prozessorientierter Modellierung um die besten Modellierungsansätze bei verschiedenen Aspekten zu ermöglichen. Die Grundlage hierfür ist die *Schedulable*-Klasse, von der sowohl *Event* als auch *Entity* und damit auch dessen Unterklasse *SimProcess,* erben. Der Scheduler kann mit *Schedulable*-Objekten umgehen, was an vielen Scheduler-Methoden ersichtlich wird, die eben ein solches Objekt als Parameter erwarten. Ein Prozess kann also z. B. statt einen anderen Prozess ein Ereignis, auf das er eine Referenz hat, angeben wie in Listing 5 dargestellt. Umgekehrt funktioniert es analog.[35]

```
1   public class ExampleProcess extends SimProcess
2   {
3       //...
4       public void lifeCycle()
5       {
6           //...
7           activateAfter(event1)
8           //...
9       }
10  }
```

Listing 5: Kombination von Prozessen und Ereignissen

4.2.4 Queues

Warteschlangen sind für die Modellierung von Systemen mit begrenzten Ressourcen von elementarer Bedeutung. DESMO-J bietet hierfür die Klassen *Queue* für normale Entitäten und *ProcessQueue* für Prozesse. Beide Queues erben von *QueueBased*, die wiederum von *Reportable* erbt. Queues sammelt automatisch Daten, die durch Nutzung der Einrichtungen für die Berichterstattung (reporting) der Oberklasse *Reportable*, dazu verwendet werden können statistische Messwerte darzustellen, z. B. in den Report-Dateien oder in Form von Grafiken (z. B. Histogramme).

Queues haben eine Reihe von wichtigen Methoden, die in Tabelle 12 auf der nächsten Seite aufgeführt und beschrieben werden. Sowohl die Klasse *Queue* als auch *ProcessQueue* sind generische Klassen mit den Typparametern E bei der Klasse *Queue* für Entities und P bei der Klasse *ProcessQueue* für SimProcesses. Es wird also bereits bei der Instanziierung der Klassen ein bestimmter Untertyp von Entity oder SimProcess festgelegt mit dem gearbeitet wird. Die Methoden in der Tabelle 12 auf der nächsten Seite gelten also analog für die Klasse *ProcessQueue*, nur mit dem Typparameter T statt E. Bei der Instanziierung der Queues muss auch eine Sortierug festgelegt werden, z. B. FIFO (First In First Out). FIFO ist als Standardwert definiert.[36]

[35] vgl. [PK05, S. 274-275]
[36] vgl. [PK05, S. 276-277] und [DES11a, Klassen Queue, ProcessQueue und Reportable]

Methode	Beschreibung
insert(E e)	Fügt das Ereignis *e* in die Liste ein.
remove(E e)	Entfernt das Ereignis *e* aus der Liste.
insertAfter(E e, E after)	Fügt das Ereignis *e* nach Ereignis *after* ein.
insertBefore(E e, E before)	Fügt das Ereignis *e* vor Ereignis *before* ein.
isEmpty()	Prüft, ob die Queue leer ist, falls ja wird *true* zurück gegeben.
first()	Liefert das Ereignis an erster Stelle der Queue zurück.
last()	Liefert das Ereignis an letzter Stelle der Queue zurück.
succ(E e)	Liefert das Nachfolgeereignis von Ereignis *e* zurück.
pred(E e)	Liefert das Vorgängerereignis von Ereignis *e* zurück.

Tabelle 12: Wichtige Methoden der Klasse *Queue*
(auf Basis von [PK05, S. 276] und [DES11a, Klasse Queue])

4.2.5 Wahrscheinlichkeitsverteilungen

Wahrscheinlichkeitsverteilungen werden benötigt um Zufallszahlen bzw. -werte zu generieren um damit zufälliges Verhalten zu simulieren. Sämtliche Wahrscheinlichkeitsverteilungen von DESMO-J basieren auf *java.util.Random* der Standard-Java-Bibliothek. Die Oberklasse aller Wahrscheinlichkeitsverteilungen ist die abstrakte Klasse *Distribution* die von *Reportable* erbt, was, wie bei den Queues auch, zur Darstellung statistischer Messwerte genutzt werden kann. Je nach Verteilungsart müssen bei der Instanziierung Werte wie Start- und Endwert bzw. Wertebereiche (*seed*s) definiert werden aus denen die Zahlen gezogen werden sollen. Ausgabewerte können ganze Zahlen, rationale Zahlen oder boolesche Werte sein.[37]

Wichtige Methoden der Klasse *Distribution*, und damit deren Unterklassen auch, werden in Tabelle 13 beschrieben.

Methode	Beschreibung
sample()	Liefert einen Zufallswert zurück.
setNonNegative(boolean newValue)	Legt fest ob die Zufallswerte negativ sein dürfen oder nicht.

Tabelle 13: Wichtige Methoden der Klasse *Distribution*
(auf Basis von [PK05, S. 277,278] und [DES11a, Klasse Distribution])

DESMO-J unterstützt folgende Wahrscheinlichkeitsverteilungen[38]:

- Gleichverteilung (z. B. Klasse *RealDistUniform*)
- Normalverteilung (Klasse *RealDistNormal*)
- Exponentialverteilung (Klasse *RealDistExponential*)
- Bernoulli-Verteilung (Klasse *BoolDistBernoulli*).

[37] vgl. [PK05, S. 277-278] und [DES11a, Klasse Distribution und Unterklassen]
[38] vgl. [SIM11] und [DES11a]

4.2.6 Experiment

Die Implementierung des Experiments befindet sich in der Black-Box-Klasse *Experiment*. Wie in Abschnitt 4.2 auf Seite 13 in Listing 1 zu sehen ist erfolgt die Instanziierung des Experiments in der *main()*-Methode. Ausführlicher zu beschreiben ist an dieser Stelle der Report-Generator, da er die Report-Dateien erzeugt, welche für die Auswertung eines Experiments entscheidend sind. Die Daten hierfür werden von den statischen Black-Box-Komponenten, die von *Reportable* erben, automatisch erzeugt und gesammelt. Voraussetzung dafür ist, dass diese Klassen bei der Instanziierung dafür parametriert werden. Listing 6 zeigt den Konstruktor einer Queue. Wichtig ist hierbei der Parameter *boolean showInReport*, der festlegt, ob Daten über diese Queue in die Report-Datei kommen oder nicht. Analog dazu verhält es sich für alle anderen statischen Modellkomponenten.[39]

```
1 public Queue(Model owner, String name, int sortOrder, int qCapacity,
    boolean showInReport, boolean showInTrace)
```

Listing 6: Konstruktor der Klasse *Queue*

Darüber hinaus werden auch die Dateien für Trace-(Parameter *boolean showInTrace*), Debug- und Error-Meldungen generiert. Nachdem in der *main()*-Methode *exp.report()* aufgerufen wurde werden die Dateien im Hauptordner erzeugt. Die Namen werden dabei nach folgendem Muster gebildet[40]:

- <Experimentname>_report.html
- <Experimentname>_trace.html
- <Experimentname>_debug.html
- <Experimentname>_error.html

5 Fazit

Im Rahmen dieser Arbeit wurde zu Beginn die Motivation für Simulationen - speziell Computersimulationen - genannt. Die hohe Komplexität heutiger von Menschen geschaffener Systeme geht weit über die Möglichkeiten der analytischen Modelle hinaus und so bietet die dynamische Simulation einen vergleichsweise komfortablen Weg um solche Systeme mittels Modellen zu analysieren und Rückschlüsse auf das Originalsystem zu ziehen.

Diese Arbeit hat einen möglichen Weg genannt um dynamische Systeme zu simulieren. In ihrer Ausführung ist diese Seminararbeit keinesfalls vollständig, allerdings sollte das vermittelte Wissen ausreichen um Fuß in der Materie fassen zu können. Zum weiterführenden Studium dieses Fachs ist das Literaturverzeichnis zu empfehlen, speziell die Quellen [PK05], [Wai09] und [CL08]. Um tieferen Einblick in das DESMO-J-Framework zu bekommen sei die Quelle [PK05] und die sehr gut dokumentierte API [DES11a] nahegelegt.

Mit einem Blick in die Zukunft lässt sich erkennen, dass Computersimulationen immer wichtiger werden, da verschiedene Systeme um uns herum komplizierter und kostspieliger werden. Unzählige Einzelheiten müssen beachtet werden damit solche Systeme funktionieren und deren Erprobung in der Realität wäre in vielen Fällen viel zu teuer. Als Lösung wird daher verstärkt die Computersimulation angewandt und damit ist sie - zumindest meiner Erkenntnis nach - ein äußerst interessantes und stark wachsendes Feld der Computertechnologie.

[39] vgl. [PK05, S. 280-282] und [DES11a, Klassen Model, Queue und Reportable]
[40] vgl. [PK05, S. 283]

Abbildungsverzeichnis

Literatur

[CL08] Cassandras, Christos G. ; Lafortune, Stéphane: *Introduction to discrete event systems*. Springer, 2008

[DES11a] DESMO-J Entwickler-Team: *DESMO-J (2.2.0) API*. `http://desmoj.sourceforge.net/doc/index.html`. Version: 16. Januar 2011

[DES11b] DESMO-J Entwickler-Team: *Identification of Relevant Entities*. `http://desmoj.sourceforge.net/tutorial/events/design0.html`. Version: 16. Januar 2011

[DES11c] DESMO-J Entwickler-Team: *Offizielle DESMO-J-Homepage*. `http://www.desmo-j.de`. Version: 16. Januar 2011

[Eic11] Eichner, Prof. Dr. Martin: *Deterministische und stochastische Simulationsmodelle in der Infektionsepidemiologie*. `http://www.fernuni-hagen.de/epigrid/pdf/abstract\_eichner.pdf`. Version: 16. Januar 2011

[KFA11] Kluth, Ronald ; Fischer, Prof. Dr. Joachim ; Ahrens, Dr. Klaus: *Ereignisorientierte Computersimulation mit ODEMx*. `http://edoc.hu-berlin.de/series/informatik-berichte/218/PDF/218.pdf`. Version: 16. Januar 2011

[Mül11] Müller, Norbert T.: *Einführung in die ereignisgesteuerte Simulation Vorlesung im SS 2004*. `http://www.informatik.uni-trier.de/~mueller/Lehre/2004-simulation/simu04.pdf`. Version: 16. Januar 2011

[Nie77] Niemeyer, Gerhard: *Kybernetische System- und Modelltheorie: system dynamics*. Vahlen, 1977

[PK05] Page, Bernd ; Kreutzer, Wolfgang: *Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005

[Rad97] Radatz, Jane: *The IEEE Standard Dictionary of Electrical and Electronics Terms*. Institute of Electrical and Electronics Engineers, 1997

[Sha75] Shannon, Robert E.: *Systems simulation: the art and science*. Prentice-Hall, 1975

[SIM11] SIM- Leibniz Universität Hannover: *Einführung in das Modellieren mit DesmoJ*. `http://www.sim.uni-hannover.de/~svs/wise0809/pds/unterlagen/DesmoJ\_Vorstellung.odp`. Version: 16. Januar 2011

[Sto11] Stockinger, Thomas: *6.1 Analytische Modelle*. `http://www.dorn.org/uni/sls/kap06/f01.htm`. Version: 16. Januar 2011

[The11] `http://www.apache.org/licenses/LICENSE-2.0.html`

[Ver93] Verein Deutscher Ingenieure: *Simulation von Logistik-, Materialfluss- und Produktionssystemen.* VDI, 1993

[Wai09] Wainer, Gabriel A.: *Discrete-event modeling and simulation: a practitioner's approach.* CRC Press, 2009