

Hochschule für Technik und Wirtschaft Dresden



Fakultät Informatik/Mathematik

Diplomarbeit

im Studiengang Informatik

Thema: Konzeption einer grafischen Nutzeroberfläche zur
prozessorientierten Modellierung und Simulation

eingereicht von: Martin Domnick

eingereicht am: 14. November 2011

Betreuer: Prof. Dr.-Ing. Wilfried Nestler
Hochschule für Technik und Wirtschaft Dresden
Fachbereich Informatik / Mathematik
Friedrich-List-Platz 1 D-01069 Dresden

2 Analyse etablierter Simulationspakete für ereignisorientierte Simulationen

Um ein beständiges und nachhaltiges Konzept zu entwickeln, ist eine fundierte Analyse unabdingbar. Hierbei erscheint es zweckmäßig, sich an am Markt etablierten Lösungen für DES zu orientieren. Anhand ausgewählter Basiskonzepte, die der ereignisorientierten Simulation zu eigen sind, sollen einzelne Simulationsprogramme verglichen werden. Hierbei ist es nicht Zielstellung das beste Simulationspaket am Markt zu bestimmen, sondern vielmehr gewonnene Erkenntnisse über sinnvolle Funktionalitäten in die Planung der Eigenentwicklung einfließen zu lassen.

Da es auf dem Markt eine Vielzahl von Softwarelösungen für ereignisorientierte Computersimulationen gibt, musste eine Auswahl getroffen werden. Die Simulationssoftwarepakete Simul8, Simprocess und Micro Saint Sharp wurden für diesen Zweck gewählt. Die in diesem Kapitel verwendeten englischen Terminologien sind jeweils aus den untersuchten Applikationen übernommen wurden.

2.1 Simulationszeit

Bei der DES geht es um die Simulation einer Abfolge von Prozessen über einen endlichen Zeitraum. Eine Simulation am Computer erfolgt niemals in Echtzeit. Die Dauer (Simulationszeit) von Computersimulationen kann sich abhängig von der Problemstellung über Sekunden, Minuten, Stunden, Tage, Wochen oder gar Jahre erstrecken. Eine Flexibilität bei der Planung von Ereignissen ist daher sehr wichtig.

Eine Simulation wird beendet, wenn keine weiteren Ereignisse mehr geplant sind oder die Maximaldauer der zentral festgelegten Simulationsdauer überschritten worden ist. Es besteht darüber hinaus die Möglichkeit, die Simulation explizit zu beenden, wenn gewisse Zustände im System erreicht worden sind. Dies kann z.B. über Programmcode, der zu einem festgelegten Simulationszeitpunkt ausgeführt wird, erreicht werden. Ebenso bieten einige Modellbausteine die Möglichkeit, die Simulation bei Erreichen eines Zustandes vorzeitig zu beenden.

2.1.1 Zeitangaben

Bei der Planung von Ereignissen ist der richtige Umgang mit der Simulationszeit essentiell. Die Genauigkeit von Zeitangaben kann abhängig vom verwendeten Simulationspaket bis auf eine Nanosekunde genau festgelegt werden. Ein Gegensatz stellt die Verwendung von Einheitslosen Datentypen dar, wodurch Modellentwickler die Möglichkeit erhalten, ihre eigene Basiseinheit zu definieren. In diesem Abschnitt soll es darum gehen, die verfolgten Ansätze, der dieser Analyse zugrunde liegenden Simulationspakete, zu bewerten.

2.1.1.1 Simul8

Die Basiseinheit für Simul8 wird zentral definiert. Zur Auswahl stehen Sekunden, Minuten, Stunden und Tage. Alle Zeitangaben in Simul8 beziehen sich immer ausschließlich auf diese Basiseinheit.

Es erfolgt eine direkte Festlegung der Simulationsdauer für eine definierte Zeiteinheit. Feinere Granularität ist durch Angaben von Bruchzahlen möglich.

Abbildung 2.1.: Dialog zur Konfiguration der Simulationszeit in Simul8

Bei Simul8 besteht zusätzlich die Möglichkeit, für die Dauer eines Tages die Anzahl der Stunden festzulegen. Weiterhin lässt sich auch die Anzahl der Wochentage festlegen. Eine fünf Tage Arbeitswoche á acht Stunden kann beispielsweise wie in Abbildung 2.1 dargestellt für eine Simulationsdauer (Result Collocations Period) von 2400 Minuten definiert werden.

2.1.1.2 Simprocess

In Simprocess wird keine Basiseinheit zentral definiert. Dafür kann jede Zeitangabe in der Simulationssoftware Simprocess exakter und ohne Umrechnung in Bruchzahlen angegeben werden.

Die Simulationsdauer wird durch die Bestimmung der Start- und Endzeit der gesamten Simulation durch Angabe eines konkreten Datums und einer konkreten Uhrzeit festgelegt.

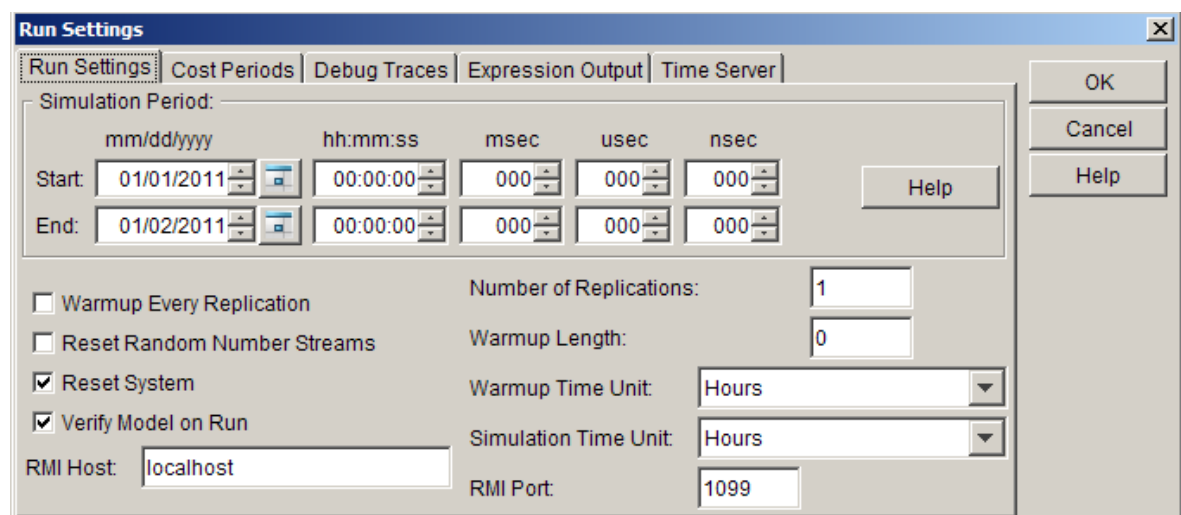


Abbildung 2.2.: Dialog zur Konfiguration der Simulationszeit in Simprocess

2.1.1.3 Micro Saint Sharp

Simulationszeit wird in Micro Saint Sharp lediglich über einen *Double*-Wert repräsentiert.

2.1.1.4 Bewertung

Die verwendeten Ansätze zur Verwaltung der Simulationszeit unterscheiden sich in allen untersuchten Applikationen recht deutlich voneinander. So mag es sinnvoll sein,

sich einmal vor Augen zu führen, welche Kriterien eine Verwaltung der Simulationszeit erfüllen soll. Insbesondere die Softwareanwendungen Simprocess und Simul8 sind mit dem Ziel erschaffen worden, Menschen, die begrenzte Kenntnisse in der Programmierung besitzen, anzusprechen. Somit überrascht es nicht, dass diese Softwaresysteme konkrete Zeitangaben verwenden. Der Vorteil, der sich hieraus ergibt ist eine Art Selbstdokumentation des Simulationsmodells. Weiterhin wird der Modellentwickler dadurch praktisch an die Hand genommen und spart sich verwirrende Umrechnungen in Bruchzahlen. Für die Präsentation einer Simulation ist damit auch weniger Erklärungsbedarf vorhanden. Ein nicht zu verachtender Nachteil, der sich aus dieser Herangehensweise ergibt, ist allerdings die Einschränkung auf spezielle Anwendungsgebiete. Für gewisse Anwendungsfälle kann es sich als nützlich erweisen, Basiseinheiten zu verwenden, die unter oder über dem Bereich liegen als das, was von der Simulationssoftware ermöglicht wird.

Mirco Saint Sharp verwendet einen Einheitslosen Datentyp, womit größtmögliche Flexibilität gewahrt bleibt. Dokumentation darüber, welche Zeiteinheit auf diese Einheitslosen Datentyp abgebildet wird, obliegt aber dem Modellentwickler selbst. Schaut man genauer hin, erkennt man, dass die Software Simul8 eine Mischlösung verwendet, die es Anwendern erlaubt eine Basiseinheit aus vordefinierten Typen zu wählen. Die Auswahlmöglichkeiten sind allerdings auch sehr Eingeschränkt, sodass der zuvor genannte Nachteil nicht aufgehoben wird.

Um Zeiteinheiten abzubilden, die zwischen einem Zeitschritt einer gewählten Basiseinheit liegen, verwenden Simul8 sowie Micro Saint Sharp Gleitkommazahlen. Gleitkommazahlen sind jedoch nur eine angenäherte Darstellung von den aus der Mathematik bekannten *reellen Zahlen*. Ein Vorteil von Gleitkommazahlen ist deren großer Wertebereich. Dieser wird allerdings durch eine ungenaue Abbildung auf *reelle Zahlen* erkauft. Zwei nahe beieinander liegende reelle Zahlen können daher in einem ungünstigen Fall auch auf die selbe Gleitkommazahl abgebildet werden. Auf die Zeitplanung von Ereignissen, die eng beieinander liegen, kann sich das durchaus auswirken. Denn im Zweifelsfall muss die Software entscheiden, welches Ereignis Vorrang hat und in Ermangelung genauer Daten kann dies zu einem Tausch der Reihenfolge führen. Fehler können sich fortsetzen und letztlich das Gesamtergebnis einer Simulation verfälschen. Betrachtet werden muss also die Wahrscheinlichkeit, mit welcher solch ein Problem auftreten kann und welche Auswirkungen dies auf das Ergebnis hat. Nun handelt es sich bei dem hier verfolgten Ansatz der Computersimulation keinesfalls um eine exakte Wissenschaft, als vielmehr um eine Annäherung an die Realität. Somit bleibt es Ermessenssache, inwieweit diese Ungenauigkeiten das Ergebnis einer Computersimulation verfälschen können. Generell ist es aber sehr schwierig diese Art von Programmfehlern zu finden. Daher würde sich die Verwendung eines stabileren Datentyps anbieten. Ein stabilere

Datentyp der einen ebenso großen Wertebereich umfasst, wie beispielsweise der Datentyp *Double*, würde selbstredend mit hohen Performance-Einbußen einhergehen. Da die meisten modernen Prozessoren 64 Bit-Register besitzen, sind für die Verwendung von Datentypen, die größer als 64 Bit sind, mehr Rechenzyklen für sonst gleichwertige Operationen notwendig.

Zusammenfassend soll festgehalten werden, dass die konkrete Wahl eines Ansatzes für die Verwaltung von Simulationszeit vom Anwendungsgebiet abhängig ist. Da die SimNet-Bibliothek bereits den Datentyp *Double* verwendet, ist für die angestrebte Eigenentwicklung bereits eine Vorgabe in dieser Angelegenheit gegeben, von der ohne Änderungen in der SimNet-Bibliothek auch nicht abgewichen werden kann. Das .Net Umfeld bietet als einzige Alternative für Gleitkommazahlen den 128-Bit Datentyp *Decimal*. Dieser besitzt mehr signifikante Stellen als beispielsweise der Datentyp *Double*, dafür ist der verwendete Wertebereich weniger umfangreich.

2.1.2 Aufwärmphase

Ziel einer Aufwärmphase ist es, das System in einen typischen Urzustand zu versetzen. Statistische Daten über den Verlauf der Simulation sind ein wichtiges Hilfsmittel, um die Simulation auszuwerten. Solche Daten werden erst nach einer sogenannten *WarmUp-Phase* erfasst. Eine Aufwärmphase ist nicht für jede Simulation notwendig und wird deshalb von einigen Softwaresystemen optional zur Verfügung gestellt.

2.1.2.1 Simul8

Bei Simul8 geht die Aufwärmphase nicht in die festgelegte Simulationsdauer mit ein, sondern verlängert die tatsächliche Dauer zusätzlich. Alternativ besteht die Möglichkeit, nach dem Warmup die Simulationszeit auf den Initialwert zurückzusetzen.

2.1.2.2 Simprocess

Die Dauer der Aufwärmphase kann für Simprocess direkt festgelegt werden. Es ist zu beachten, dass die Warmup-Phase direkt mit in die Simulationszeit eingeht und somit die Statistiken erst nach *Startzeit + Warmupzeit* für den Zeitraum *Endzeit – Warmupzeit – Startzeit* erfolgen.

2.2 Wahrscheinlichkeitsverteilungen

Die meisten Ereignisse lassen sich nicht oder nur ungenügend mit statischen Zeitpunkten und Häufigkeiten modellieren. Um beispielsweise die Abläufe in einem Callcenter zu simulieren, sind Unregelmäßigkeiten beim Eintreffen der Anrufer und der jeweiligen Gesprächsdauer zu berücksichtigen. Statistische Verteilungsfunktionen sind dabei ein vorzügliches Werkzeug, um solche Problemstellungen anzugehen. Der Tabelle 2.1 kann entnommen werden, welche Verteilungsfunktionen in welchem Softwarepaket zur Verfügung stehen. Bei der Bibliothek SimNet handelt es sich um die zugehörige Bibliothek für die Spracherweiterung der Programmiersprache C#, welche an der HTW-Dresden entwickelt worden ist.

Verteilungsfunktion	Simprocess	Simul8	Micro Saint Sharp	SimNet
Bernoulli	✗	✓	✓	✗
Beta	✓	✓	✓	✗
Binomial	✓	✓	✓	✗
Erlang	✓	✓	✗	✓
Exponential	✓	✓	✓	✓
Extreme Value Type A	✗	✗	✓	✗
Extreme Value Type B	✗	✗	✓	✗
Gamma	✓	✓	✓	✗
Geometric	✓	✓	✓	✗
Hyper Exponential	✓	✗	✗	✗
Inverse Gaussian	✓	✗	✓	✗
Inverted Weibull	✓	✗	✗	✗
Johnson SB	✓	✗	✗	✗
Johnson SU	✓	✗	✗	✗
Log-Laplace	✓	✗	✗	✗
Log-Logistic	✓	✗	✓	✗
Log-Normal	✓	✓	✓	✓
Negative Binomial	✓	✓	✓	✗
Normal	✓	✓	✓	✓
Pareto	✓	✗	✓	✗
Pearson Type V	✓	✓	✓	✗
Pearson Type VI	✓	✓	✓	✗
Poisson	✓	✓	✗	✗

Tabelle 2.1.: Statistische Verteilungsfunktionen

Verteilungsfunktion	Simprocess	Simul8	Micro Saint Sharp	SimNet
Random Walk	✓	✗	✗	✗
Triangular	✓	✓	✓	✓
Uniform	✓	✓	✓	✓
Weibull	✓	✓	✓	✓

Tabelle 2.1.: Statistische Verteilungsfunktionen

Wie zu erwarten war, sind in allen drei Systemen die wichtigsten Verteilungsfunktionen implementiert. Darunter auch einige exotische Varianten, die in der Praxis evtl. nicht so häufig Anwendung finden. Diese Liste zeigt allerdings auch, dass bei der SimNet-Bibliothek in diesem Punkt noch etwas nachgerüstet werden kann.

2.3 Modell

Die umfangreichsten und zugleich bedeutendsten Unterschiede zwischen den einzelnen Simulationswerkzeugen sind beim eigentlichen Simulationsmodell zu erkennen. Um nicht den Rahmen zu sprengen, kann dieser Abschnitt nur als Zusammenfassung der wesentlichsten Merkmale dienen. Bevor auf die Details der Realisierung zwischen den einzelnen Simulationspaketen eingegangen werden kann, ist eine Betrachtung der Basiskonzepte notwendig.

2.3.1 Basiskonzepte

Jedes Computermodell besteht aus einer Anzahl von Kanten und Knoten. Zwischen den Knoten entlang der Kanten bewegen sich Token. Die Simulationsbausteine, die im Modell als Knoten dargestellt werden, werden auch als Aktivitäten bezeichnet und haben einen maßgeblichen Einfluss auf den Verlauf einer Simulation. Die Funktionalitäten, die durch Aktivitäten repräsentiert werden, können sehr vielseitig sein. So können Aktivitäten in das Modell neue Token, d.h. Entitäten, einführen, eingehende Entitäten vernichten oder an einen anderen verbundenen Zielknoten, d.h. an eine Folgeaktivität, weiterleiten. Aktivitäten werden durch eine Fülle von Merkmalen beschrieben, sodass Modellentwickler über die Manipulation der Zustände dieser Merkmale das Verhalten einer Aktivität innerhalb eines Simulationsmodells beeinflussen können. Daher werden Aktivitäten auch nach ihrer Aufgabe voneinander unterschieden. Ein Simulationsmodell besitzt üblicher Weise min. eine Quelle zum Eintritt neuer Entitäten in das Modell

sowie min. eine Senke zum Austritt von Entitäten aus dem Modell. Unüblich ist hingegen die Modellierung von geschlossenen Systemen ohne Quelle und Senke, denn in solch einem System können Entitäten die Simulation weder betreten noch verlassen. Einige Simulationsprogramme, wie beispielsweise Simul8, erlauben allerdings auch die Modellierung von geschlossenen Systemen. In diesem Fall können Entitäten bereits für den Initialzustand der Simulation als Bestandteil in das Modell integriert sein.

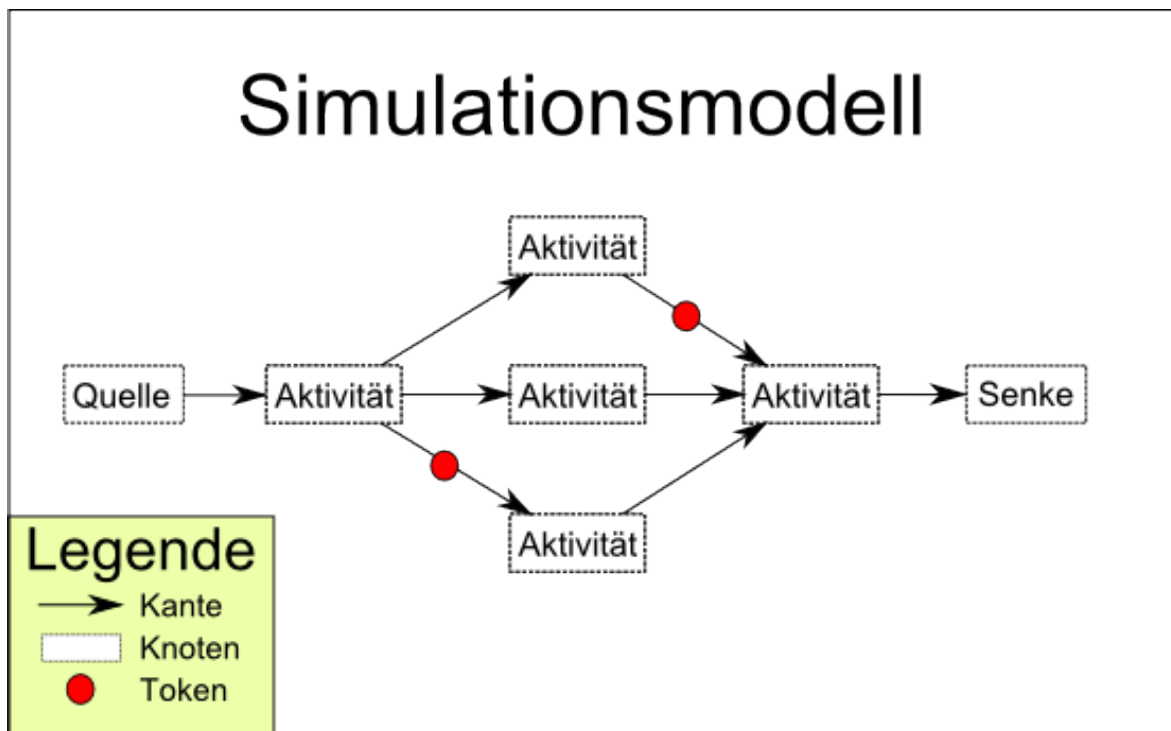


Abbildung 2.3.: Aufbau eines Simulationsmodells

Der größte Unterschied zwischen den untersuchten Simulationspaketen liegt in der Varietät bei der Gestaltung der Aktivitäten. Durch die angestrebte Spezialisierung bzw. Generalisierung bei der Bereitstellung von Modellbausteinen, erklärt sich der unterschiedliche Umfang bezüglich der Menge bereitgestellter Aktivitäten. In der nun folgenden Betrachtung soll daher gezeigt werden, wie sich die unterschiedlichen Ansätze in der Praxis bewähren.

2.3.2 Simul8

Wie so oft besitzt auch die Simulationssoftware Simul8 ihre eigene Terminologie. In den allermeisten Fällen stellen die verwendeten Bezeichnungen nur Synonyme für bereits vorgestellte Begriffe dar. Um Verwirrung zu vermeiden, ist daher zu erwähnen, dass Entitäten in Simul8 als *Work Items* bezeichnet werden.

Da sich die Aktivitäten aus Simul8 recht deutlich voneinander unterscheiden, ist eine Vorbetrachtung gemeinsamer Eigenschaften an dieser Stelle nicht notwendig.

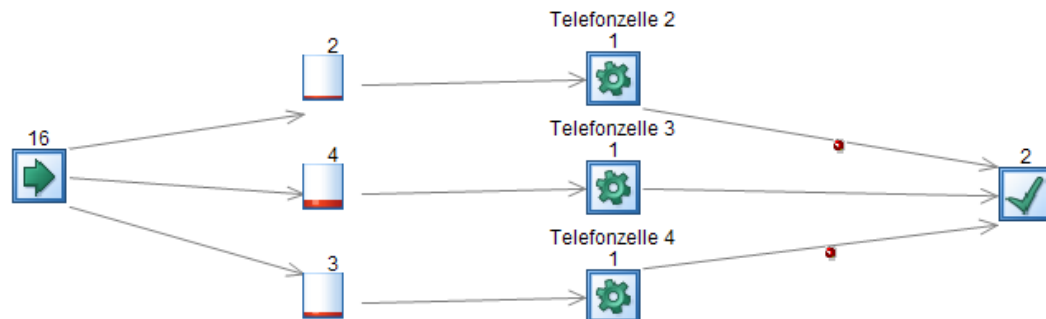


Abbildung 2.4.: Beispiel für ein Modell in Simul8

2.3.2.1 Simulationsbausteine

2.3.2.1.1 Work Entry Point



Der Eintrittspunkt für jedes *Work Item* stellt der Work Entry Point dar. Dieser kann mit den Generatoren aus anderen DES-Systemen verglichen werden. Die wichtigsten Eigenschaften des *Work Entry Points* sind in der Tabelle 2.2 aufgeführt.

Eigenschaft	Beschreibung
Interarrival Time	Die Ankunftszeiten neuer <i>Work Items</i> können über Verteilungsfunktionen beschrieben werden.
Batching	Die Anzahl der <i>Work Items</i> , die zum selben Zeitpunkt eintreffen, kann ebenfalls über eine Verteilungsfunktion beschrieben werden.
Routing	Wenn ein <i>Work Entry Point</i> mit mehreren Folgeaktivitäten verbunden ist, muss definiert werden, zu welcher Aktivität das <i>Working Item</i> weitergeleitet wird
Label Actions	Vor dem Verlassen eines <i>Work Entry Points</i> können Attribute des <i>Work Items</i> manipuliert werden. Dies kann insbesondere für den späteren Programmfluss nützlich sein, da diese Attribute von anderen Aktivitäten ausgewertet werden können.

Tabelle 2.2.: Eigenschaften des Work Entry Points

Eigenschaft	Beschreibung
Finance	Für die Kostenkalkulation bietet sich eine Festsetzung von Fixkosten (<i>Capital Cost</i>) für den <i>Work Entry Point</i> und ein Festsetzung von variablen Kosten für jede im System eintreffende Einheit an.
Carbon	Der Treibhauseffekt wird durch die Festlegung von erzeugten CO2 Emissionen berücksichtigt. Analog zu den Finanzen kann ein fester Wert für den <i>Work Entry Point</i> und ein variabler Wert für jede im System eintreffende Einheit festgelegt werden.
Results	Für diese Aktivität werden unter anderem die folgenden Statistiken gesammelt: <ul style="list-style-type: none"> ➤ Anzahl eingetroffener <i>Work Items</i> ➤ Anzahl verloren gegangener <i>Work Items</i> ➤ Anzahl <i>Work Items</i>, welche den <i>Work Entry Point</i> verlassen haben, ohne dabei verloren gegangen zu sein.

Tabelle 2.2.: Eigenschaften des Work Entry Points

2.3.2.1.2 Storage Bin



Warteschlangen stellen in Simul8 eines der wichtigsten Bestandteile dar. Diese werden am sinnvollsten vor *Work Center* platziert, um *Work Items*, die für die Bearbeitung in einem Work Center vorgesehen sind, aufzufangen und bereitzuhalten. Tabelle 2.3 führt die wichtigsten Eigenschaften einer Warteschlange aus Simul8 auf.

Eigenschaft	Beschreibung
Capacity	Die Kapazität der Warteschlange ist in der Regel unbegrenzt, kann aber auch auf einen Limit gesetzt werden.
Shelf Life	Die maximale Verweilszeit eines <i>Work Items</i> in der Warteschlange kann begrenzt werden.
Finance	Für die Kostenkalkulation bietet sich eine Festsetzung von Fixkosten (<i>Capital Cost</i>) für die <i>Storage Bin</i> und eine Festsetzung von variablen Kosten für jedes in der Warteschlange befindliche <i>Work Item</i> pro Zeiteinheit an.

Tabelle 2.3.: Eigenschaften einer Warteschlange in Simul8

Eigenschaft	Beschreibung
Carbon	Der Treibhauseffekt wird durch die Festlegung der von der Warteschlange erzeugten CO ₂ Emissionen berücksichtigt. Analog zu den Finanzen kann ein fester Wert für die <i>Storage Bin</i> und ein variabler Wert für jedes in der Warteschlange befindliche <i>Work Item</i> je Zeiteinheit festgelegt werden.
Start - Up	Festlegung der Anzahl von <i>Work Items</i> , welche sich bei Beginn der Simulation in der Warteschlange befinden.
First In First Out (FIFO), PRIORITIZE, LIFO	Die Reihenfolge, in welcher <i>Work Items</i> in einer Warteschlange sortiert werden, kann nach folgenden Kriterien erfolgen <ul style="list-style-type: none"> ➤ First In First Out (Standard) ➤ Last In First Out ➤ Prioritize (Sortierung nach Priorität über ein <i>Label</i> (Attribut) des <i>Work Items</i>)
Results	Auflistung statistischer Werte hinsichtlich der Anzahl von <i>Work Items</i> in der Warteschlange über die Dauer der Simulation. (Minimum, Maximum, Durchschnitt, Anzahl von eingetroffenen <i>Work Items</i>) Auflistung statistischer Werte hinsichtlich der Verweildauer von <i>Work Items</i> in der Warteschlange (Minimum, Durchschnitt, Maximum, Standardabweichung)

Tabelle 2.3.: Eigenschaften einer Warteschlange in Simul8

2.3.2.1.3 Work Center



Bei dem *Work Center* handelt es sich um eine weitere entscheidende Kernkomponente eines jeden Simul8-Modells. Die Bezeichnung *Work Center* ist äußerst passend, denn diese Aktivität kann zur Simulation von Arbeit genutzt werden, da Simulationszeit durch das Festhalten von *Work Items* voranschreitet, noch bevor ein *Work Item* die Aktivität wieder verlässt. Die Möglichkeiten für ein *Work Center* auf Simulationen einzuwirken sind daher immens. Aufgrund der Komplexität dieses Modellbausteins eignet sich ein *Work Center* für eine Vielzahl von Anwendungsfällen. Beispielsweise kann ein *Work Center* dazu verwendet werden, den Zusammenbau von Einzelkomponenten bei der Fertigung von Produkten zu simulieren. Damit unterscheidet sich Simul8 im Vergleich zu anderen Simulationsprogrammen insofern, dass für

viele der Anwendungsfälle, die ein *Work Center* abdecken kann, in anderen Softwareprodukten mehrere einzelne Bausteine verwendet werden müssen. Insbesondere ist die optimierte Zusammenarbeit mit Warteschlangen und Ressourcen zu erwähnen. Die wichtigsten Eigenschaften können aus der Tabelle 2.4 entnommen werden.

Eigenschaft	Beschreibung
Distribution	Mittels statistischer Verteilungsfunktionen kann die zu verbrauchende Simulationszeit festgelegt werden, mit welcher Arbeitszeit simuliert werden soll.
Efficiency	Es kann festgelegt werden, mit welcher Auslastung das <i>Work Center</i> arbeitet. Zusätzlich können Reparaturzeiten simuliert werden.
Priority	Eine Priorität kann vergeben werden. Dies ist insbesondere wichtig, wenn mehrere <i>Work Center</i> um die selben Ressourcen konkurrieren.
Label Actions	Ebenso wie beim <i>Work Entry Point</i> können Attribute des <i>Work Items</i> vor dem verlassen des <i>Work Centers</i> verändert werden.
Replicate	Festlegung der Anzahl der <i>Work Items</i> , die im <i>Work Center</i> gleichzeitig bearbeitet werden können.
Routing In	Einige Einstellungsoptionen, die regeln, nach welchen Vorgaben ein <i>Work Center</i> von vorgeschalteten Aktivitäten <i>Work Items</i> annimmt. Viele dieser Einstellungen sind nur in Verbindung mit einer Warteschlange sinnvoll. Eine weitere interessante Möglichkeit besteht darin, ein <i>Work Center</i> durch Arbeit von einem anderen <i>Work Center</i> unterbrechen zu lassen. Dabei wird das aktuelle <i>Work Item</i> in eine mit dem <i>Work Center</i> verbundene Warteschlange geschoben, um später die Arbeit wieder aufnehmen zu können.
Routing Out	Wenn ein <i>Work Center</i> mit mehreren Folgeaktivitäten verbunden ist, muss definiert werden zu welcher Aktivität das <i>Work Item</i> weitergeleitet wird.
Contents	Wenn die Simulation angehalten wird, kann der Inhalt des <i>Work Centers</i> , das heißt die darin befindlichen <i>Work Items</i> sowie deren Attribute, betrachtet werden.

Tabelle 2.4.: Eigenschaften eines *Work Centers* in Simul8

Eigenschaft	Beschreibung
Results	Während der Simulation werden vom <i>Work Center</i> Statistiken gesammelt. U.a. sind eine Reihe von Informationen bezüglich der Anzahl der <i>Work Items</i> im <i>Work Center</i> verfügbar. (Aktuell, Maximum, Minimum, Durchschnitt, Anzahl beendeter Jobs). Viel interessanter sind sicherlich die Werte über die Auslastung des <i>Work Centers</i> . Diese werden in Prozent angegeben und ergeben somit zusammen 100 Prozent. Mit diesen Größen kann man neben der tatsächliche Arbeitszeit erkennen, wie lange und weshalb das <i>Work Center</i> nicht im Betrieb war.
Finance	Für die Kostenkalkulation bietet sich eine Festsetzung von Fixkosten (<i>Capital Cost</i>) für das <i>Work Center</i> und eine Festsetzung von Stückkosten für jedes im <i>Work Center</i> bearbeitete <i>Work Item</i> an. Außerdem können Kosten für jede vergangene Zeiteinheit berücksichtigt werden.

Tabelle 2.4.: Eigenschaften eines *Work Centers* in Simul82.3.2.1.4 *Work Exit Point*

Work Exit Points dienen in einem Simul8-Modell als Senke. Wenn beispielsweise ein *Work Item* einen *Work Exit Point* erreicht hat, so verlässt es über diesen die Simulation. Im Vergleich zu anderen Modellbausteinen in Simul8 ist der *Work Exit Point* weniger komplex. In der Tabelle 2.5 werden die wichtigsten Eigenschaften des *Work Exit Points* aufgezählt

Eigenschaft	Beschreibung
Halt Simulation at Limit	Nach Erreichen einer definierten maximalen Anzahl von <i>Work Items</i> die am <i>Work Exit Point</i> eintreffen, wird die Simulation beendet
Finance	Festsetzung von Fixkosten (<i>Capital Cost</i>) für den <i>Work Exit Point</i> . Weiterhin können Einnahmen in Form eines Stückpreises (<i>Revenue Costs</i>) pro <i>Work Item</i> , welche über diese Senke die Simulation verlassen, definiert werden

Tabelle 2.5.: Eigenschaften eines *Work Exit Points* in Simul8

Eigenschaft	Beschreibung
Carbon	Der Treibhauseffekt wird durch die Festlegung der erzeugten CO2 Emissionen berücksichtigt. Analog zu den Finanzen kann ein Wert für den <i>Work Exit Point</i> festgelegt werden. Zudem ist es möglich, eine Verringerung der CO2 Emissionen für jede die Simulation verlassende Einheit zu definieren.
Results	<ul style="list-style-type: none"> ➤ Statistiken über die Verweildauer der <i>Work Items</i> in der Simulation (Minimum, Durchschnitt, Maximum, Standardabweichung) ➤ Anzahl <i>Work Items</i>, die über diesen <i>Work Exit Point</i> die Simulation verlassen haben

Tabelle 2.5.: Eigenschaften eines *Work Exit Points* in Simul8

2.3.2.1.5 Resource



Ressourcen können in Simul8 als grafischer Baustein in das Modell integriert werden. Aber im Gegensatz zu den bisher vorgestellten Simulationsbausteinen sind Ressourcen keine Aktivitäten und daher nicht über Kanten mit anderen Simulationsbausteinen verbunden. *Work Center* können um Ressourcen konkurrieren. Damit eignen sich Ressourcen vorzüglich, um Abhängigkeiten zu simulieren. Die Tabelle 2.6 enthält eine Auflistung der wichtigsten Eigenschaften von Ressourcen in Simul8.

Eigenschaft	Beschreibung
Travel	Definition von Reisezeiten für Ressourcen.
Availability	Festlegung eines Prozentsatzes für die Verfügbarkeit der Resource.
Shifts	Ein sehr hilfreiches Feature ist die Bestimmung von Arbeitsschichten, womit die Verfügbarkeit von Ressourcen ebenfalls eingeschränkt werden kann.
Finance	Definition von Fixkosten für jede Einheit und variable Kosten je Zeiteinheit.

Tabelle 2.6.: Eigenschaften einer *Ressource* in Simul8



















	Aktivität	Warteschlange	Simulationszeit	Ressourcen	Input-Pads	Output-Pads
	Generate	✗	✓	✗	0	1
	Dispose	✓	✗	✗	1	0
	Delay	✓	✓	✓	1	1
	Assemble	✓	✓	✓	2	2
	Branch	✓	✓	✓	1	1
	Merge	✗	✗	✗	1	1
	Batch	✓	✓	✓	1	1
	Unbatch	✓	✓	✓	1	1
	Split	✓	✓	✓	1	2
	Join	✓	✓	✓	1	2
	Transform	✓	✓	✓	1	1
	Transfer	✗	✗	✗	0/1	0/1
	Gate	✓	✓	✓	2	1
	Assign	✓	✓	✓	1	1
	Synchronize	✓	✓	✓	variabel	variabel
	Get Resource	✓	✗	✓	1	1
	Free Resource	✗	✗	✓	1	1

Tabelle 2.7.: Aktivitäten in Simprocess

2.3.3.1 Simulationsbausteine


2.3.3.1.1 Generate

 Generatoren dienen in Simprocess als Quelle für das Simulationsmodell. Diese erzeugen gemäß eines Plans (engl. *Schedule*) neue Entitäten. Hiervon existieren eine Reihe von unterschiedlichen Arten, wovon die wichtigsten in der Tabelle 2.8 zusammengefasst werden. Eine Besonderheit ist, dass einem Generator mehrere solcher *Schedules* zugewiesen werden können. Womit ein Generator in die Lage versetzt wird, verschiedene Typen von Entitäten zu erzeugen.

Scheduletyp	Beschreibung
Periodic	Der <i>Periodic Schedule</i> ist wohl der am häufigsten verwendete Scheduletyp in Simprocess. Über statistische Verteilungsfunktionen kann über solch einen Plan der Rhythmus zur Erzeugung von Entitäten bestimmt werden. Ebenfalls kann über eine Verteilungsfunktion die Anzahl der Entitäten bestimmt werden, die zum selben Zeitpunkt im Simulationsmodell eintreffen.
Calendar	Der <i>Calendar Schedule</i> kann wie der <i>Periodic Schedule</i> über statistische Verteilungsfunktionen die Anzahl der Entitäten bestimmen, die zum selben Zeitpunkt erzeugt werden sollen. Die Besonderheit liegt aber darin, dass feste Zeitpunkte bestimmt werden können, wann neue Entitäten im Simulationsmodell eintreffen. Beispielsweise kann ein exaktes Datum oder aber eine exakte Definition von monatlich, wöchentlich, täglich oder stündlich verwendet werden.
Weekly	Der <i>Weekly Schedule</i> ist im Prinzip eine Erweiterung des <i>Periodic Schedules</i> . Einzig wird die Möglichkeit ergänzt, Zeiträume festzulegen, innerhalb welcher dieser Schedule aktiv ist. Nützlich ist dies, um beispielsweise Arbeitszeiten zu simulieren.

Tabelle 2.8.: Schedulearten in Simprocess

2.3.3.1.2 Dispose

 Das Gegenstück zu *Generate* ist der *Dispose* Baustein. Über diesen verlassen Entitäten das Simulationsmodell. *Dispose* kann auch dazu verwendet werden, die Simulation vorzeitig abzuberechnen, indem eine Maximalanzahl von Entitäten bestimmt wird. Dabei muss es sich nicht um einen festen Wert handeln, denn auch die Verteilungsfunktionen, die in der Tabelle 2.1 auf Seite 9 in der Spalte Simprocess aufgelistet worden sind, können verwendet werden.

2.3.3.1.3 Delay



Eine der elementarsten Komponente ist sicherlich die *Delay*-Aktivität. Diese wird eingesetzt, um ankommende Entitäten für einen festgelegten Zeitraum zu blockieren.

2.3.3.1.4 Assemble



Assemble kann verwendet werden, um beispielsweise den Zusammenbau von Komponenten zu simulieren. Notwendige Materialien werden in Form von Entitäten gesammelt. Die Gesamtheit der benötigten Materialien kann sich aus unterschiedlichen Entitätstypen zusammensetzen. Dabei lässt sich auch die Anzahl der notwendigen Entitäten über statistische Verteilungsfunktionen definieren. Sobald eine *Assemble*-Aktivität die notwendigen Materialien hat, beginnt diese mit der Arbeit. Zuletzt muss noch der Entitätstyp für die zu erzeugende Entität festgelegt werden. Da es im Gegensatz zu vielen anderen Aktivitäten mehrere Input- und Output-Pads gibt, werden diese in der Tabelle 2.9 aufgeführt.

Pad	Typ	Beschreibung
Trigger (Optional)	Input	Die <i>Assemble</i> -Aktivität kann zusätzlich auf einen Trigger warten, dieser wird ausgelöst, sobald die Aktivität von einer Entität über den Trigger-Pad erreicht worden ist. Die eintreffende Entität wird sofort vernichtet und landet damit nicht in der Warteschlange der <i>Assemble</i> -Aktivität.
Component	Input	Entitäten, die als Material für die Fertigung verwendet werden sollen, erreichen die <i>Assemble</i> -Aktivität über dieses Pad.
Out	Output	Erzeugte Entitäten verlassen die <i>Assemble</i> -Aktivität über dieses Pad.
NoMatch (Optional)	Output	Entitäten, die für den Fertigungsprozess nicht verwendet werden können, können über dieses Pad die <i>Assemble</i> -Aktivität wieder verlassen

Tabelle 2.9.: Schedulearten in Simprocess

2.3.3.1.5 Branch



Die *Branch*-Aktivität kann als Abzweigung eintreffende Entitäten auf verschiedenen Wegen zu verbundenen Aktivitäten weiterleiten. Die Tabelle 2.10 listet die möglichen Arten der Abzweigung auf.

Abzweigungsart	Beschreibung
Probability	Die Wahrscheinlichkeit bestimmt, welcher Weg gewählt wird. Die Wahrscheinlichkeit für alle angeschlossenen Verbindungen muss insgesamt 1.0 betragen.
Attribute	Für die Wahl des Weges können auch Attribute von Entitäten der Aktivität oder dem Modell herangezogen werden. Über Vergleichsoperationen wird entschieden, welchen Weg eine Entität wählt.
Entity Type	Der Typ der Entität kann ebenfalls als ausschlaggebendes Kriterium bei der Wahl des Weges herangezogen werden.
Priority	Bei der Priorität handelt es sich lediglich um eine Komfortfunktion, denn diese ist im Grunde auch nur ein Attribut der Entität.

Tabelle 2.10.: Abzweigungsvarianten für die *Branch*-Aktivität in Simprocess

2.3.3.1.6 Merge



Die *Merge*-Aktivität kann als Gegenstück zur *Branch*-Aktivität betrachtet werden. Alle eintreffenden Entitäten werden auf dem selben Weg weiter versandt. Diese Aktivität ist nicht unbedingt notwendig, da alle InputPads genau nach eben diesem Prinzip arbeiten. Allerdings hilft sie, die Struktur des Simulationsmodells übersichtlicher zu gestalten. Außerdem erhalten alle eintreffenden Entitäten, wie bei jeder anderen Aktivität, einen neuen Zeitstempel ¹. Im Gegensatz zu den meisten anderen Aktivitäten kann die *Merge* Aktivität daher auch keine Simulationszeit verbrauchen oder Ressourcen verwenden.

¹Vgl. [08] S. 141

2.3.3.1.7 *Batch*

Die *Batch*-Aktivität fasst eine zuvor bestimmte Anzahl von Entitäten zu einer neuen Entität zusammen. Je nach Einstellung müssen zusätzliche Kriterien erfüllt sein, bevor mit dem *Batch*-Prozess begonnen wird. Ist die maximale Aufenthaltsdauer für Entitäten in der Warteschlange erreicht sowie das Kriterium für die minimale Anzahl von Entitäten in der Warteschlange erfüllt, wird der *Batch*-Prozess gestartet. Weiterhin können eingehende Entitäten nach der Priorität oder dem Typ der Entität getrennt werden.

2.3.3.1.8 *Unbatch*

Die *Unbatch*-Aktivität kehrt den Prozess der *Batch*-Aktivität um, indem sie Entitäten, die sich in eintreffenden Container-Entitäten befinden, auspackt. Dabei besteht die Option, auch verschachtelte Pakete zu berücksichtigen. Container-Entitäten können je nach Einstellung ebenfalls erhalten bleiben.

2.3.3.1.9 *Split*

Mit der *Split*-Aktivität lassen sich von einer eintreffenden Entität beliebig viele Kopien erstellen. Für das spätere Zusammenführen der Kopien mit dem Original wird ein Familienname vergeben. Eine Beschreibung der beiden Output-Pads kann der Tabelle 2.11 entnommen werden.

Output-Pad	Beschreibung
Original	Über dieses Pad verlassen Original-Entitäten die <i>Split</i> -Aktivität.
Clones	Kopien des Originals verlassen über dieses Pad die <i>Split</i> -Aktivität. Mit diesem Pad können über einen <i>Connector</i> beliebig viele Aktivitäten verbunden werden. Die Anzahl der Verbindungen entspricht somit der Anzahl der Klone.

Tabelle 2.11.: Output-Pads der *Split*-Aktivität

2.3.3.1.10 Join



Die *Join*-Aktivität dient zur Zusammenführung der Originale mit ihren Kopien, wie sie von der *Split*-Aktivität erzeugt worden sind. Dabei befinden sich Kopien und Originale so lange in der Warteschlange der *Join*-Aktivität, bis alle zugehörigen Komponenten eingesammelt worden sind. Die *Join*-Aktivität besitzt zwei Output-Pads, deren Beschreibung der Tabelle 2.12 entnommen werden kann.

Output-Pad	Beschreibung
Join	Über dieses Pad verlassen Original-Entitäten die <i>Join</i> -Aktivität.
NoMatch	Dieses Output-Pad dient zur Aussortierung von Entitäten, die für eine <i>Join</i> -Operation nicht in Frage kommen.

Tabelle 2.12.: Output-Pads der *Join*-Aktivität

2.3.3.1.11 Transform



Die *Transform*-Aktivität wandelt den Typ von eintreffenden Entitäten in einen anderen Typ um. Dabei können globale Attribute sowie der Erstellungszeitpunkt, die Zeitstempel und die Priorität der ursprünglichen Entität übernommen werden.

2.3.3.1.12 Transfer




Die *Transfer*-Aktivität unterscheidet sich von allen bisher vorgestellten Aktivitäten entscheidend. *Transfer*-Aktivitäten stellen eine Möglichkeit dar, auch ohne Verbindungen Entitäten an andere *Transfer*-Aktivitäten weiterzuleiten. Bei *Transfer*-Aktivitäten muss zwischen zwei verschiedenen Zuständen unterschieden werden. In der Tabelle 2.13 werden die Unterschiede erläutert.

Modus	Beschreibung
Send	Befindet sich eine <i>Transfer</i> -Aktivität im <i>Send</i> -Zustand, besitzt diese ein Input-Pad und es werden alle eingehenden Entitäten an die <i>Transfer</i> -Aktivität weitergeleitet, die als Empfänger definiert worden ist.
Receive	Eine <i>Transfer</i> -Aktivität, die sich im <i>Receive</i> -Zustand befindet, kann von beliebig vielen <i>Transfer</i> -Aktivitäten Entitäten empfangen. In der Folge werden alle eingetroffenen Entitäten an die über das Output-Pad angebundene Aktivität weitergeleitet.

Tabelle 2.13.: Zustände der *Transfer*-Aktivität


2.3.3.1.13 Gate

 Die *Gate*-Aktivität sammelt eingehende Entitäten und hält diese solange in ihrer Warteschlange auf, bis entweder die Anzahl der Entitäten in der Warteschlange größer als eine definierte Schranke ist oder eine Entität über das *Trigger*-Pad die *Gate*-Aktivität erreicht hat.

2.3.3.1.14 Assign

 Die *Assign*-Aktivität stellt eine einfache Möglichkeit dar, um Werte von benutzerdefinierten Attributen sowie Prioritäten für Entitäten zu definieren.

2.3.3.1.15 Synchronize

 Die *Synchronize*-Aktivität besitzt eine variable Anzahl von Input- und Output-Pads. Sobald an jedem Eingang der Aktivität jeweils mindestens eine Entität eingetroffen ist, werden an allen korrespondierenden Ausgängen Entitäten wieder freigelassen.

2.3.3.1.16 Get Resource



Häufig ist es eine Notwendigkeit, eine Ressource für eine Abfolge von Aktivitäten zu belegen. Mit der Aktivität *Get Resource* ist dies möglich. Wenn eine Entität bei dieser Aktivität angelangt ist und sofern eine Ressource bereitsteht, darf die Entität die *Get Resource* Aktivität wieder verlassen. In jedem anderen Fall werden eingehende Entitäten in der Warteschlange der Aktivität aufbewahrt.

2.3.3.1.17 Free Resource



Um eine Ressource wieder freizugeben, wird die Aktivität *Free Resource* benötigt. Sobald eine Entität diese Aktivität betritt, werden alle von ihr belegten Ressourcen wieder freigegeben.

2.3.3.2 Bewertung

Die Simulationssoftware Simprocess bietet eine enorme Anzahl unterschiedlicher Simulationsbausteine, die sich jeweils durch ihre Spezialisierung auszeichnen. Ein netter Nebeneffekt ist es, dass komplexe Modelle dadurch wesentlich übersichtlicher erscheinen und somit die Abläufe in einem Simulationsmodell bedeutend schneller nachvollzogen werden können. Es hat sich außerdem herausgestellt, dass die integrierten Warteschlangen ebenfalls dazu beitragen, das Modell aufzuräumen. Zudem ist die Kommunikation der Aktivitäten untereinander damit deutlich vereinheitlicht. Eine weitere interessante Komponente stellen auch die Pads für die Beschreibung der Ein- und Ausgänge der Aktivitäten dar. Denn durch dieses Konzept ist es möglich, mehrere Eingänge mit unterschiedlichen Aufgaben für eine Aktivität zu definieren. Dies ist ein Ansatz, welcher sich auch für die aus dieser Diplomarbeit hervorgehenden Eigenlösung empfiehlt. Anhand der Analyse der von Aktivitäten bereitgestellten Funktionalitäten ist außerdem gut zu erkennen, welchen Platz Simulationsbausteine in der Vererbungshierarchie einnehmen.

2.3.4 Micro Saint Sharp

In Simul8 und Simprocess ist ein rein grafischer Modellaufbau möglich, wenngleich für viele erweiterte Szenarien auch mit zusätzlichem Programmcode Einfluss auf das Geschehen in der Simulation genommen werden kann. Das Simulationssoftwarepaket

Micro Saint Sharp geht dagegen einen etwas anderen Weg. Bei der Entwicklung eines Simulationsmodells in Micro Saint Sharp kann auf Programmierkenntnisse in C# nicht verzichtet werden, da viele Eigenschaften der Simulationsbausteine über Programmcode als Funktion in eigens dafür vorgesehenen Codefenstern beschrieben werden müssen.

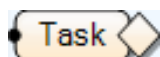
Diese Simulationsbausteine nennt man im Micro Saint Sharp *Tasks*. Das Verhalten einer *Task* ist sehr universell angelegt. Neben der Standard-*Task* gibt es noch eine kleine Anzahl weiterer Tasks, welche sich aber nur in Detail unterscheiden, da diese häufig nur wenige Eigenschaften ergänzen.

Token, die sich durch das Simulationsmodell bewegen, werden wie in Simprocess als Entität (*Entity*) bezeichnet. Diese verfügen über Attribute, auf welche eine Task zugreifen kann. Es lassen sich überdies jederzeit eigene Attribute einer Entität hinzufügen. Ebenfalls können globale Variablen für das gesamte Modell definiert werden.

Für die Wahl der Datentypen steht für Attribute und globale Variablen gleichermaßen nur eine begrenzte Auswahl zur Verfügung. Neben den aus C# bekannten Datentypen Boolean, Integer, FloatingPoint (double), Object, String, Hashtable kann aber zusätzlich auch der Datentyp Entity genutzt werden. Weiterhin können Listen und Arrays verwendet werden, die aber ebenfalls auf die eben benannten Datentypen beschränkt sind. Diese Einschränkungen gelten hingegen nicht für lokale Variablen, die in den in die Oberfläche der Software eingebetteten Codefenstern verwendet werden können.

2.3.4.1 Simulationsbausteine

2.3.4.1.1 Task



Release Condition Bevor eine Entität zur Bearbeitung freigegeben wird, wird zunächst die *Release Condition* ausgewertet. Bei der *Release Condition* handelt es sich um eine Funktion, die einen Wahrheitswert (*Boolean*) zurückgibt.

Effects Wenn eine Entität eine Task erreicht, werden sogenannte Effekte ausgelöst. Diese Effekte können mittels Programmcode beschrieben werden. Effekte können zum Beispiel genutzt werden, um Attribute von Entitäten zu setzen oder globale Variablen zu aktualisieren.

Effekt	Beschreibung
Beginning Effect	Der <i>Beginning Effect</i> wird ausgelöst bevor die Task ausgeführt wird. Erst im Anschluss wird die Ausführungszeit der Task berechnet. ²
End Effect	Der <i>End Effect</i> wird ausgelöst wenn die Task die Ausführung beendet und bevor die Entität die Task verlässt. ³
Launch Effect	Der <i>Launch Effect</i> wird ausgeführt, nachdem die Ausführungszeit berechnet wurde. ⁴

Tabelle 2.14.: Effekte einer *Task* in Micro Saint Sharp

Timing Wahrscheinlichkeitsverteilungen dienen zur Modellierung der Ausführungszeit einer Task. Die Eigenschaften einer solchen Verteilungsfunktion werden jeweils über den Rückgabewert (*Double*) einer C#-Funktion definiert.

Paths Eine Task kann über eine unbegrenzte Zahl von ausgehenden Verbindungen mit Folgetasks verbunden sein. Wenn eine Entität eine Task verlässt, wird diese gemäß einer zuvor definierten Strategie eine oder mehrere dieser Verbindungen wählen. Als Entscheidungsgrundlage dienen hierbei Callback-Funktionen, wovon für jede Verbindung eine existiert. Je nach Wahl der Strategie (*Multiple, Probabilistic, Tactical*) erfüllen diese Callback-Routinen unterschiedliche Aufgaben. Eine Auflistung dieser findet sich in der Tabelle 2.15 wieder.

Decision Type	Beschreibung
Multiple	Alle verbundenen Tasks erhalten eine Kopie der Entität, für die die jeweilige Callback-Routine den Wahrheitswert <i>true</i> liefert. Auch eine Rückverbindung zur selben Task kann gewählt werden. In diesem Fall kann eine Task wie ein Generator verwendet werden, wenngleich es dafür auch einen spezialisierten Task-Typ gibt, der diesen Vorgang vereinfacht.
Probabilistic	Jede Callback-Routine gibt einen Wert zurück, der die Wahrscheinlichkeit angibt, über welchen Weg Entitäten die Task verlassen. Die Summierung aller Werte muss durch eine Zehnerpotenz teilbar sein. .

Tabelle 2.15.: Routing-Strategien in Micro Saint Sharp

²Vgl. [05] S. 95

³Vgl. [05] S. 95

⁴Vgl. [05] S. 96

Decision Type	Beschreibung
Tactical	Es wird die Verbindung gewählt, deren Callback-Routine den höchsten Wert zurück gegeben hat.

Tabelle 2.15.: Routing-Strategien in Micro Saint Sharp

Queues Warteschlangen sind ein integrierter Bestandteil einer Task. Es existieren 4 verschiedene Typen von Warteschlangen

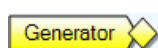
Typ	Beschreibung
None	Auch wenn die Bezeichnung <i>None</i> impliziert, dass damit für die Task keine Warteschlange vorhanden ist, ist dies nicht der Fall. Lediglich die Einflussmöglichkeiten mittels Effekte sind bei dieser Voreinstellung nicht gegeben. Entitäten werden nach dem FIFO-Prinzip in der Warteschlange aufgehalten, bis die Task wieder bereit ist.
FIFO	Entitäten verlassen die Warteschlange in der Reihenfolge, wie sie diese betreten haben.
Last In First Out (LIFO)	Entitäten verlassen die Warteschlange in der umgekehrten Reihenfolge, wie sie diese betreten haben.
Sorted	Entitäten werden innerhalb der Warteschlange gemäß einer über eine Funktion kalkulierten Priorität sortiert.

Tabelle 2.16.: Arten einer Warteschlange in Micro Saint Sharp

Effekt	Beschreibung
Queue Entering Effect	Wird ausgeführt, wenn eine Entität die Warteschlange betritt.
Queue Departing Effect	Wird ausgelöst, wenn eine Entität die Warteschlange verlässt.

Tabelle 2.17.: Effekte einer Task mit integrierter Warteschlange in Micro Saint Sharp

2.3.4.1.2 Generator Task



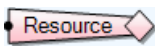
Die Generator Task kann im Simulationsmodell ausschließlich als Quelle für neue Entitäten verwendet werden. Mit Ausnahme der Warteschlange und den Ef-

fekten sowie der *Release Condition*, erbt diese Task alle sonstigen Eigenschaften, die eine normale Task besitzt.

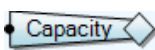
Eigenschaft	Beschreibung
Start Time	Startzeitpunkt des Generators
Repeat Type	<p>Durch die Festlegung dieser Eigenschaft werden Einschränkungen für den Generator verfügbar.</p> <ul style="list-style-type: none"> ➤ <i>RepeatForever</i> Der Generator läuft, bis die Simulation beendet wird. ➤ <i>RepeatToCount</i> Der Generator läuft, bis die definierte Anzahl von Entitäten erzeugt wurde. ➤ <i>RepeatToTime</i> Der Generator läuft, bis zu einem festgelegten Zeitpunkt.
Entity Generation Code	Hierbei handelt es sich um eine Routine, die bei der Generation einer Entität ausgeführt wird. Dies ist hilfreich, um zum Beispiel Attribute einer erzeugten Entität zu manipulieren.

Tabelle 2.18.: Spezielle Eigenschaften eines Generators in Micro Saint Sharp

2.3.4.1.3 Resource Task

 Auch bei der *Resource Task* handelt es sich um eine Modifikation der regulären Task. Globale Variablen vom Typ Integer können in Micro Saint Sharp als Ressourcen verwendet werden. Deren Initialwert gibt dabei die maximale Anzahl an verfügbaren Einheiten an. Jede *Resource Task* benötigt für die Ausführung eine Einheit der zugewiesenen Ressource. Für eine *Resource Task* können allerdings keine weiteren Einschränkungen über eine *Release Condition* definiert werden, wie dies bei der allgemeinen Task der Fall ist. Abgesehen davon gibt es keine Unterschiede zur Standard-Task.

2.3.4.1.4 Capacity Task

 Die *Capacity Task* ist eine Erweiterung der regulären Task. Diese Task kann bis zu einem festgesetzten Limit für mehrere Entitäten gleichzeitig ausgeführt werden.

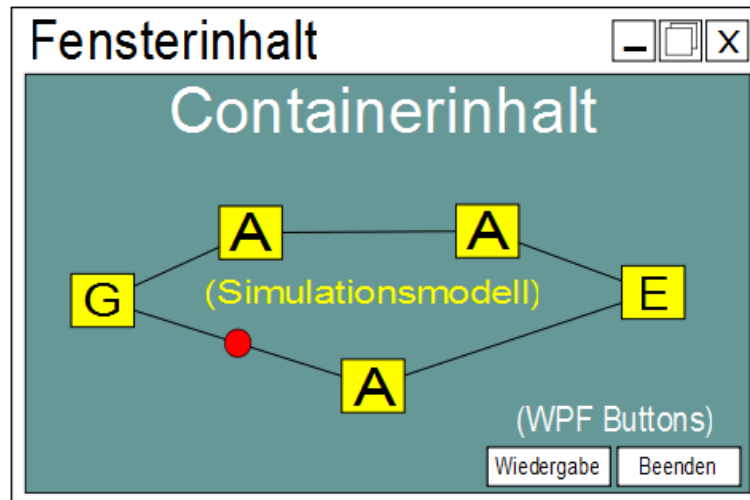


Abbildung 3.1.: Oberflächenprototyp für eine Anwendung, die auf der SimNetUI-Bibliothek basiert.

Als minimale Ausgangslage für die Entwicklung einer eigenen Simulation auf Basis der *SimNetUI* Bibliothek kann folgender Extensible Application Markup Language (XAML)-Quellcode dienen.¹

```

1 <Window x:Class="Example.Test.MainWindow"
2       xmlns="http://schemas.microsoft.com/winfx/2006/\Gls{xaml}/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/\Gls{xaml}"
4       Title="MainWindow" Height="350" Width="687"
5       xmlns:my="\Gls{SimNetUI}"
6       >
7     <my:SimulationContainer HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
8       Name="simulation1">
9       <Button Content="{Binding RelativeSource={RelativeSource self},Path=
10         Command.Text}" Command="MediaCommands.Play" CommandTarget="{Binding
11         ElementName=simulation1}" Canvas.Left="12" Canvas.Top="12" Height="23"
12         HorizontalAlignment="Left" VerticalAlignment="Top" Width="83" />
13       <Button Content="{Binding RelativeSource={RelativeSource self},Path=
14         Command.Text}" Command="MediaCommands.Stop" CommandTarget="{Binding
15         ElementName=simulation1}" Canvas.Left="101" Canvas.Top="12" Height="23"
16         HorizontalAlignment="Left" VerticalAlignment="Top" Width="75" />
17       <TextBlock Canvas.Left="563" Canvas.Top="12" Height="23" Text="
18         Simulationszeit:" Width="90" />
19       <Slider Canvas.Left="182" Canvas.Top="12" Height="23" Width="260" Value="{
20         Binding Path=AnimationClockSpeed, ElementName=simulation1}" TickFrequency="1
21         " Maximum="25" />

```

¹ Für nachfolgende XAML-Quellcode-Auszüge stellt dieses Rahmenprogramm die Grundlage dar. Wobei folgendes zu beachten ist:

- Steuerelemente sind immer in der Hierarchie dem Simulationskontainer als Kind-Elemente untergeordnet
- Zu Demonstrationszwecken werden häufig nur Attribute dargestellt, welche für die Betrachtung von einer konkreten Eigenschaft unmittelbar von Bedeutung sind. Eine ausführbare Anwendung ohne Veränderung des Quellcodes ergibt sich somit nicht.

```
12      <TextBlock Canvas.Left="462" Canvas.Top="12" Height="23" Text="{Binding
      ElementName=simulation1, Path=AnimationClockSpeed}" Width="61" TextAlignment
      ="Center" />
13      <TextBlock Canvas.Left="563" Canvas.Top="31" Height="18" HorizontalAlignment="
      Left" Name="textBlock10" Text="{Binding ElementName=simulation1, Path=
      SimulationTime,StringFormat={}{0:0.0000}}" TextAlignment="Center"
      VerticalAlignment="Top" Width="78" />
14  </my:SimulationContainer>
```

Listing 3.1: Minimales Beispiel für ein Rahmenprogramm ohne Simulationsmodell

3.2 Aktivitäten

Aktivitäten stellen wichtige Kernkomponenten für den Aufbau eines Simulationsmodells dar. Sie besitzen Eigenschaften, deren Zustand sich auf den Verlauf einer Simulation entscheidend auswirken kann. Insofern ist es wichtig, neue Aktivitäten möglichst auf eine Weise zu konstruieren, dass diese den bestehenden Satz an Aktivitäten möglichst gut ergänzen.

Eine wichtige Erkenntnis aus der Analyse ist es, dass es Sinn macht, viele Aktivitäten mit spezialisierten Aufgaben zu implementieren. Dies hat den Vorteil, dass es sowohl für den Betrachter als auch den Entwickler wesentlich einfacher nachzuvollziehen ist, was im Modell passiert, als wenn Aktivitäten zu viele Nebeneffekte haben, wenngleich die Gesamtheit der verfügbaren Aktivitäten immer noch so allgemein sein muss, dass diese sich auf eine Vielzahl von Anwendungsfällen abbilden lassen.

Durch die Betrachtung der Softwarepakete Simul8, Simprocess und Micro Saint Sharp hat es sich allerdings auch gezeigt, dass es sinnvoll ist, gewisse Aufgaben als Funktionalitäten für die Mehrzahl der Aktivitäten direkt zu implementieren, statt zusätzliche Aktivitäten zu schaffen. Dies hat technische wie auch praktische Gründe.

3.2.1 Allgemeine Eigenschaften von Aktivitäten

3.2.1.1 Warteschlangen

Im Hinblick auf die technische Realisierung hat es sich gezeigt, dass es wesentlich einfacher ist, Warteschlangen in ausgewählte Aktivitäten zu integrieren, anstatt einen extra Baustein in Form einer Aktivität bereitzustellen. Dies hat unter anderem auch damit zu tun, dass die Kommunikation der Aktivitäten dadurch untereinander wesentlich vereinfacht wird, insofern da ein allgemein weniger fehleranfälliger Ansatz verfolgt werden

```

1 <my:Generator Canvas.Left="49" Canvas.Top="171" Name="generator1">
2   <my:Generator.Entity>
3     <my:Entity />
4   </my:Generator.Entity>
5   <my:Generator.Schedule>
6     <my:Schedule>
7       <my:UniformDouble Max="5" Min="0" my:Schedule.Duration="Infinity" Seed="1" />
8     </my:Schedule>
9   </my:Generator.Schedule>
10  <my:Out Connector="Out">
11    <my:Target Activity="exit1" Connector="In" ConnectionPoints="C 179.33,199 179
12      .33,104" />
13    <my:Target Activity="exit2" Connector="In" ConnectionPoints="L 216.5,199 L 216
14      .5,247 L" />
15  </my:Out>
16 </my:Generator>
17 <my:Exit Canvas.Left="328" Canvas.Top="76" Name="exit1" />
18 <my:Exit Canvas.Left="328" Canvas.Top="219" Name="exit2" />

```

Listing 3.2: Ausschnitt eines XAML-Quelltextes zur Demonstration des Verbindungscode

In diesem Beispiel werden drei Aktivitäten erzeugt.

- Ein Generator, der für die Erzeugung neuer Entitäten zuständig ist
- Zwei Exit-Aktivitäten, über welche Entitäten das Simulationsmodell verlassen.

Der Generator besitzt eine Liste von ausgehenden Verbindungen. Solch eine Liste ist als *Content*-Element jeder Aktivität zugeordnet. Im Falle der Exit-Aktivität können allerdings keine ausgehenden Verbindungen beschrieben werden, da diese Aktivität kein Ausgangsverbindungsstück besitzt.

Zunächst muss definiert werden, über welches Verbindungsstück eine Aktivität mit einer Folgeaktivität verbunden ist. Wenn eine Aktivität mehrere Ausgangsverbindungsstücke besitzt, könnte der XAML-Code wie folgt aussehen.³

```

1 <my:ActivityTypeName>
2   <!-- Weitere Eigenschaften der Aktivität -->
3   <my:Out Connector="One">
4     <!-- Verbindungscode -->
5   </my:Out>
6   <my:Out Connector="Two">
7     <!-- Verbindungscode -->
8   </my:Out>
9 </my:ActivityTypeName>

```

Listing 3.3: Beispiel für eine Aktivität mit mehreren Ausgängen

³Hinweis: Dies ist nur eine theoretische Betrachtung. Derzeit besitzt keine Aktivität mehr als ein ausgehendes Verbindungsstück.

Wie also aus dem Quelltext 3.2 entnommen werden kann, verlassen Entitäten über einen Ausgang, welcher die Bezeichnung *Out* trägt, den Generator. Über diese Verbindungsstelle ist die *Generator*-Aktivität an die beiden *Exit*-Aktivitäten angeschlossen. Für jedes *Target*-Element müssen der Name der Zielaktivität sowie der Name des Eingangs definiert werden.

Diese Architektur mag zunächst unnötig kompliziert erscheinen, ist aber unvermeidbar, wenn Aktivitäten über mehrere ausgehende Verbindungsstücke verfügen sollen.

Dank der Erweiterung des Visual Studio Designers muss sich der Entwickler über diese Details aber keine Gedanken machen. Denn dieser Code wird automatisch generiert. Für den Fall, dass aber dennoch einmal Änderungen direkt im XAML-Code vorgenommen werden müssen, ist ein tieferes Verständnis des Codes jedoch von Nutzen.

Zuletzt soll noch die Zeichenkette beschrieben werden, welche für die Darstellung der Verbindung entscheidend ist. Das *Target*-Element besitzt eine Eigenschaft *ConnectionPoints*. Für die Interpretation dieser Zeichenkette wird die Methode *Parse* der Klasse *Geometry* verwendet, welche Bestandteil der WPF-API ist. An dieser Stelle sei das Buch *WPF 4 Unleashed* von *Adam Nathan* aus dem Jahre 2010 empfohlen. In diesem Buch ist im Kapitel 15 *2D Graphics* ab Seite 487 eine detaillierte Beschreibung zum Aufbau dieser Zeichenkette beschrieben. Es wurde bereits erwähnt, dass die Anfangspunkte und Endpunkte vorgegeben werden und nicht den Änderungen des Programmierers, der sich der SimNetUI-Bibliothek bedient, unterliegen. Es gibt somit 2 Modifikationen an der Zeichenkette, der sich der Entwickler stets bewusst sein muss. Der folgende Code soll die Veränderungen an der im XAML-Code hinterlegten Zeichenkette demonstrieren.

```
1 return  
2 "M " + PointToString(Start) + " " + ConnectionPoints + " " + PointToString(End);
```

Aus dem Beispiel aus Quelltext 3.2 ergibt sich somit die Zeichenkette $M\ x1,y1\ C\ 179.33,199\ 179.33,104\ x2,y2$, wobei die beiden Koordinatenpaare $x1,y1$ und $x2,y2$ Platzhalter für die errechneten tatsächlichen Werte der Koordinaten der Verbindungsstücke mit deren Bezug innerhalb des Simulationscontainers sind.

3.2.3 Verteilungsfunktionen

Im Kapitel 2.2 wurde unter anderem betrachtet, welche Bedeutung statistischen Verteilungsfunktionen im Rahmen eines Simulationsmodells zukommen. Die zugrunde liegende *SimNet*-Bibliothek implementiert bereits einige dieser Wahrscheinlichkeitsver-

teilungen⁴. Diese Verteilungsfunktionen bilden ebenso für die SimNetUI-Bibliothek die Grundlage für die Berechnung von Simulationszeit. Derzeit verwenden lediglich die Wait-Aktivität⁵ sowie die Generator-Aktivität⁶ diese Möglichkeit.

Alle Wahrscheinlichkeitsverteilungen besitzen einen *Seed*. Setzt man diesen auf einen Wert ungleich 0, werden für jede Simulation immer die gleichen Werte generiert. Ein Wert von 0 hingegen initialisiert den Seed auf einen pseudo- zufälligen Wert.

Weiterhin gibt es 2 Verteilungsfunktionen in der SimNetUI-Bibliothek, die nicht zu den Wahrscheinlichkeitsverteilungen gezählt werden können. Hinzugekommen sind die *Fixed*-Distribution, die es erlaubt einen statischen Wert zu setzen sowie die *NoEvent*-Distribution, die nur für Generatoren innerhalb eines *Schedules* sinnvoll eingesetzt werden kann, da sie es erlaubt, für einen definierten Zeitraum einen Generator zu deaktivieren.

Der nachfolgende Quelltext soll aufzeigen, wie die einzelnen Verteilungsfunktionen in XAML erzeugt werden können. Zu Demonstrationszwecken werden alle Verteilungsfunktion als WPF-Ressourcen definiert. Im Regelfall wird man diese den Attributen der Aktivitäten allerdings direkt zuordnen.

```

1 <my:SimulationContainer>
2   <my:SimulationContainer.Resources>
3     <!-- Beispiele für Wahrscheinlichkeitsverteilungen -->
4     <my:Erlang x:Key="erlang" Alpha="0.0" Beta="1.0" Seed="0.0" />
5     <my:Exponential x:Key="exponential" Alpha="1.0" Seed="0.0" />
6     <my:LogNormal x:Key="logNormal" Alpha="0.0" Beta="1.0" Seed="1.0" />
7     <my:Normal x:Key="normal" Alpha="0" Beta="1" Seed="1" />
8     <my:UniformDouble x:Key="uniformDb1" Min="1.0" Max="2.0" Seed="0.0" />
9     <my:UniformInt x:Key="uniformInt" Min="0.0" Max="2.0" Seed="0" />
10    <my:Triangular x:Key="triang" Min="0.0" Mode="1.0" Max="2.0" Seed="0.0" />
11    <my>Weibull x:Key="weibull" Alpha="0.0" Beta="1.0" Seed="0.0" />
12    <!-- Sonstige Verteilungen -->
13    <my:Fixed x:Key="fixed" Value="5" />
14    <my>NoEvent x:Key="noEvent" />
15  </my:SimulationContainer.Resources>
16 </my:SimulationContainer>

```

Listing 3.4: Verwendung von Verteilungsfunktionen in XAML

Auch wenn diese Verteilungsfunktionen derzeit nur für die Kalkulation der Wartezeiten von Aktivitäten eingesetzt werden, so ist eine erweiterte Anwendung durchaus denkbar. So sollte für nachfolgende Versionen der SimNetUI-Bibliothek über eine Verwendung

⁴Die in der SimNet-Bibliothek realisierten Wahrscheinlichkeitsverteilungen können auch der Tabelle 2.1 aus Kapitel 2.2 entnommen werden

⁵Eine Abhandlung der Wait-Aktivität ist im Kapitel 3.2.5.3 zu finden

⁶Eine umfassende Beschreibung für Generatoren können dem Kapitel 3.2.5.1 entnommen werden

von Mengenverteilungen nachgedacht werden. Beispielsweise wäre es vorstellbar, dass zu einem Zeitpunkt eine zufällige Anzahl von Entitäten gleichzeitig in der Simulation eintrifft. Für diese Modellierungsmöglichkeiten könnten dieselben Klassen eingesetzt werden.

3.2.4 Events

Wie bereits in der Einleitung erwähnt, reicht ein rein grafisches System für viele Anwendungsfälle nicht aus. Der Nutzer eines Simulationswerkzeugs wird oft nicht umhin kommen, selbst zusätzlichen Programmcode zu schreiben. Da die Simulationsbibliothek, die aus dieser Arbeit hervorgeht, bereits aufgrund ihrer technischen Grundlage in ein System eingebettet ist, welches eine Erweiterung durch den Programmierer erlaubt, erscheint es nur sinnvoll, Events bereitzustellen, welche der Programmierer nutzen kann, um zusätzlichen Programmcode auszuführen. Dieses Konzept findet sich auch in den untersuchten Simulations-Applikationen wieder und tritt besonders deutlich in Micro Saint Sharp zum Vorschein, da die dort verwendeten sogenannten Effekte einen wesentlichen Bestandteil der Simulationslogik ausmachen. Durch die Einbindung der Simulationsbibliothek in Visual Studio tritt dieser Aspekt ebenso in den Vordergrund.

Für Ereignisse gibt es drei konkrete Anwendungsfälle

- Der Programmierer möchte Entitäten mit zusätzlichen Informationen ausstatten.
- Der Programmierer möchte Zustände überwachen und auf Veränderungen reagieren, um beispielsweise selbst Ereignisse auszulösen oder Modelleigenschaften dynamisch anzupassen.
- Steuerungslogik lässt sich besonders gut über Ereignisse realisieren, da sie dem Programmierer eine erweiterte Mächtigkeit gibt, die sich über zusätzliche Eigenschaften nur begrenzt ausdrücken ließe.

Die nun folgenden Abschnitte gehen auf die, in die Aktivitäten der Simulationsbibliothek *SimNetUI* integrierten Events näher ein. Es sei noch angemerkt, dass nicht jede Aktivität notwendigerweise jedes dieser Ereignisse besitzen muss.

3.2.4.1 EntityLeft

```
1 void EntityLeft(object sender, EntityLeavingEventArgs e)
```

Listing 3.5: Prototyp des Ereignisses, welches beim Verlassen einer Aktivität durch eine Entität ausgelöst wird.

Noch bevor eine Entität eine Aktivität verlässt, wird dieses Ereignis ausgelöst. Über eine Instanz der Klasse *EntityLeavingEventArgs* kann auf eine Referenz der verlassenden Entität und auf eine Referenz auf die Zielaktivität zugegriffen werden.

Dieses Ereignis eignet sich insbesondere dafür, Eigenschaften von Entitäten zu manipulieren. Zum Beispiel kann die visuelle Darstellung der Entität dynamisch angepasst werden, bevor die Entität die Aktivität verlässt.

Die Aktivität *Exit* stellt insofern eine Ausnahme dar, denn für diese Aktivität existiert dieses Ereignis nicht, da sie keine ausgehenden Verbindungen erlaubt.

3.2.4.2 EntityEntered

```
1 void EntityEntered(object sender, EntityEnteringEventArgs e)
```

Listing 3.6: Prototyp des Ereignisses, welches von einer Aktivität beim Eintreffen einer Entität ausgelöst wird.

Dieses Ereignis wird ausgelöst, nachdem eine Entität eine Aktivität erreicht hat. Über eine Instanz der Klasse *EntityEnteringEventArgs* erhält der Programmierer Zugriff auf eine Referenz der Aktivität, von welcher die eben eingetroffene Entität stammt. Weiterhin kann ebenso wie beim *EntityLeft*-Event auf die Eigenschaften der Entität direkt eingewirkt werden.

Lediglich für die *Generator*-Aktivität wird dieses Ereignis nie ausgelöst, da die *Generator*-Aktivität keine eingehenden Verbindungen besitzt.

3.2.4.3 EntityRouted

```
1 void EntityRouted(object sender, EntityRoutingEventArgs e)
```

Listing 3.7: Prototyp des Ereignisses, welches zur Steuerung der Weiterleitung verwendet wird

Das *EntityRouted* Event erlaubt die Programmierung von Steuerungslogik, um Einfluss auf die Wahl der Folgeaktivität zu nehmen. Dieses Ereignis ist immer dann sinnvoll, wenn an eine Aktivität *mehrere* Folgeaktivitäten angeschlossen sind. Es wird ausgelöst, nachdem die Aktivität bereit ist, die Entität zum Verlassen freizugeben und noch bevor das Ereignis *EntityLeft* ausgelöst worden ist.

Über eine Instanz der Klasse *EntityRoutingEventArgs* erhält der Entwickler Zugriff auf die Entität selbst sowie einer Liste mit möglichen Zielen. Um Einfluss auf das *Routing*

zu nehmen, muss das Property *TargetIndex* den Index für ein Ziel erhalten, welcher die Nummer des Eintrags in der Liste der möglichen Ziele darstellt.

Zur besseren Veranschaulichung soll dies im folgenden Quelltext dargestellt werden. Im konkreten Beispiel sind an die Folgeaktivität mehrere Aktivitäten vom Typ *Wait* angeschlossen. Die Entität wird in der Folge an eine Aktivität weitergeleitet, die entweder noch nicht belegt ist oder die kürzeste Warteschlange besitzt.

```
1 private void generator1_EntityRouted(object sender, EntityRoutingEventArgs e)
2 {
3     // Den kleinsten Wert auswählen.
4     var minInQueue = e.Targets.Min(
5         (target) => target.GetActivity<Wait>().Statistics.InQueue +
6         target.GetActivity<Wait>().Statistics.InWork);
7
8     // Aktivitäten auswählen, die das Kriterium minInQueue erfüllen.
9     var selection = (from target in e.Targets
10         where target.GetActivity<Wait>().Statistics.InQueue +
11             target.GetActivity<Wait>().Statistics.InWork == minInQueue
12         select target).ToArray();
13
14     // Setzen des ZielIndex auf einen zufälligen Wert aus der Vorselektion
15     e.TargetIndex = selection[r.Next(selection.Count())].index;
16 }
```

Listing 3.8: Beispiel für die Verwendung des EntityRouted Events

Insofern dieses Ereignis nicht weiter über Benutzercode beschrieben wird, werden die Entitäten abwechselnd der Reihe nach zu einer der angeschlossenen Aktivitäten weitergeleitet.

Da die Aktivität *Exit* keine ausgehenden Verbindungen erlaubt, existiert für diese das Ereignis *EntityRouted* nicht.

3.2.5 Realisierte Aktivitäten

3.2.5.1 Generator

Generatoren dienen in einem Simulationsmodell als Quelle. Die Aufgabe von Generatoren ist es, neue Entitäten zu erzeugen. Die Eigenschaften eines Generators, wie er für die SimNetUI-Bibliothek realisiert wurde, kann der Tabelle 3.2 entnommen werden.

Entität in einer Simulation zukommt, hängt von deren Aufgabe im Simulationsmodell ab. Denkbare Zuordnungen sind beispielsweise Personen, Atome oder materielle Güter jeder Art. Entitäten werden über ihre Attribute beschrieben. Eine Auswahl von gängigen Attributen, die für alle Arten von Entitäten sinnvoll sind, sind in der SimNetUI-Bibliothek bereits implementiert. In nachfolgender Tabelle sind diese Attribute beschrieben.

Attribut	Beschreibung
Priorität	Über eine Priorität ist es möglich, einzelnen Entitäten mit eine Gewichtung zu bewerten. Unter anderem kann die Priorität Einfluss auf die Sortierung in einer Warteschlange nehmen. ⁷
Typ	Oft ist es dienlich, zwischen verschiedenen Arten von Entitäten innerhalb eines Modells zu unterscheiden. Über das Typ-Attribut wird diese Unterscheidung möglich. Der Typ einer Entität wird als Zeichenkette notiert.
Erscheinungsbild	Die grafische Repräsentation einer Entität kann über das Property <i>Visual Appearance</i> angepasst werden. Dem Entwickler sind an dieser Stelle keine Grenzen auferlegt. Jedes WPF-Steuerelement, das von der Klasse <i>FrameworkElement</i> erbt, kann als visuelle Darstellung einer Entität verwendet werden.
Aktivität betreten	Sobald eine Entität eine Aktivität erreicht, wird das Property <i>ActivityEntered</i> mit einem neuen Zeitstempel versehen. Für Simulationentwickler ist dieses Property allerdings schreibgeschützt.
Aktivität verlassen	Ebenso wie beim Betreten einer Aktivität erhält eine Entität auch beim Verlassen einer Aktivität einen neuen Zeitstempel. Dieser kann über das schreibgeschützte Property <i>ActivityLeft</i> ausgelesen werden.

Tabelle 3.5.: Eigenschaften von Entitäten aus der SimNetUI-Bibliothek

Für einen Generator können Attribute für zu erzeugende Entitäten wie folgt definiert werden.

```

1 <my:Generator>
2   <my:Generator.Entity>
3     <my:Entity Priority="3" Type="Anrufer">
```

⁷Siehe hierzu Kapitel 3.2.1.1

```
4      <!-- Das Erscheinungsbild für Entitäten kann an dieser Stelle direkt als
        untergeordnetes Element definiert werden -->
5      <Image Source="entity.png" />
6  </my:Entity>
7  </my:Generator.Entity>
8 </my:Generator>
```

Listing 3.9: Festlegung von Eigenschaften für Entitäten wie sie von einem Generator erzeugt werden

3.3.1 Entwurfsmuster für eigene Entitätstypen

Nun mag es eher die Ausnahme sein, dass die im letzten Abschnitt vorgestellten Attribute für Entitäten allen Anforderungen komplexer Modelle reichen. Vielmehr ist es sehr wahrscheinlich, dass für konkrete Anwendungsfälle Entitäten mit zusätzlichen Informationen ausgestattet werden müssen.

Hierfür bietet es sich an, von der aus der SimNetUI-Bibliothek bekannten Klasse *Entity* direkt abzuleiten.

```
1 using SimNetUI.View.Entity;
2 namespace Example
3 {
4     public class CustomEntity : Entity
5     {
6         public double value { get; set; }
7     }
8 }
```

Listing 3.10: Eigene Entitätsklasse auf Basis der *Entity*-Klasse in C#

Nun ist es möglich, dieses Attribut auch in XAML einzusetzen. Das Beispiel aus Quelltext 3.9 wird hierzu ein wenig angepasst.

```
1 <my:Generator xmlns:local="clr-namespace:Example">
2   <my:Generator.Entity>
3     <local:CustomEntity value="10" Priority="3" Type="Anrufer">
4       <Image Source="entity.png" />
5     </my:Entity>
6   </my:Generator.Entity>
7 </my:Generator>
```

Listing 3.11: Eigene Entitätsklasse in XAML verwenden

Sofort ersichtlich ist die Verwendung des Kürzels *local* als eigener *Namespace* im XAML-Dokument. Darüber hinaus gibt es nichts weiter zu beachten.

Wer eigene Attribute definiert, wird diese aller Wahrscheinlichkeit nach an gegebener Stelle manipulieren wollen. Hierfür bietet es sich an, die Events, die im Kapitel 3.2.4 beleuchtet worden sind, zu verwenden. Hierzu ist noch eine Umwandlung in den richtigen Datentyp, wie im Quelltext 3.12 gezeigt, notwendig.

```
1 private void wait1_EntityLeft(object sender, SimNetUI.View.Activities.Events.  
   EntityLeavingEventArgs e)  
2 {  
3     var customEntity = e.Entity as CustomEntity;  
4     if (customEntity != null)  
5     {  
6         customEntity.value = customEntity.ActivityLeft - customEntity.ActivityEntered;  
7     }  
8 }
```

Listing 3.12: Verwendung eigener Entitätsklassen in C#

3.4 Ressourcen

Aus den vorangegangenen Abschnitten lässt sich bereits entnehmen, dass Ressourcen beim Modellaufbau eine bedeutende Rolle zukommt. Ressourcen stellen ein vorzügliches Mittel dar, um Abhängigkeiten zwischen verschiedenen Aktivitäten zu simulieren. Weiterhin eignen sich Ressourcen hervorragend, um kritische Abschnitte abzusichern. Sodass beispielsweise gewährleistet werden kann, dass zu jeder Zeit nur eine begrenzte Anzahl von Entitäten sich innerhalb eines solchen kritischen Abschnitts befinden können. Für den Modellaufbau lassen sich somit 2 Gruppen von Ressourcen unterscheiden.

1. Ressourcen, die an Aktivitäten gebunden sind
2. Ressourcen, die an Entitäten gebunden sind

Zum ersten Punkt sei angemerkt, dass Ressourcen sicherlich nicht für jede Aktivität sinnvoll eingesetzt werden können. Eine Voraussetzung dafür, dass Aktivitäten in eine Beziehung mit Ressourcen treten können ist, dass zunächst einmal geregelt sein muss, wie mit Entitäten verfahren wird, falls der Aktivität benötigte Ressourcen nicht zur Verfügung stehen. Dies hat den Hintergrund, dass Aktivitäten wie sie in der SimNetUI-Bibliothek konzipiert sind, ohne Rücksicht auf den Zustand der Folgeaktivität Entitäten versenden. Im Falle der *Wait*-Aktivität, die derzeit die einzige Aktivität darstellt, an welche Ressourcen gebunden werden können, ist dies durch eine Warteschlange gelöst.

Punkt 2 ist durch die Aktivitäten *Assign Resource* und *Release Resource*, wie in den Abschnitten 3.2.5.4 und 3.2.5.5 bereits beschrieben, realisiert.

3.4.1 Definition von Simulationsressourcen in XAML

Damit Ressourcen Entitäten oder Aktivitäten zugewiesen werden können, müssen diese zunächst einmal definiert werden. Daher wird nun ein Blick auf den XAML-Code geworfen, der hierfür notwendig ist.

In der WPF lassen sich Objekte, die häufiger wiederverwendet werden sollen, als Ressourcen mit einem Namen (engl. *key*) versehen.⁸ Hierzu besitzt jedes Steuerelement (engl. *Control*) eine eigene Liste für Ressourcen (engl. *resource collection*). Jedes Element in der Hierarchie unterhalb des sogenannten *logical tree* eines solchen Elementes erhält damit Zugriff auf diese Ressource. Diese Möglichkeit macht sich auch die SimNetUI-Bibliothek zunutze.

```
1 <my:SimulationContainer>
2 <my:SimulationContainer.Resources>
3   <my:Resource x:Key="Arbeiter" Capacity="5" />
4   <my:Resource x:Key="Angestellter" Capacity="10" />
5 </my:SimulationContainer.Resources>
6 </my:SimulationContainer>
```

Listing 3.13: XAML-Quellcode für die Definition von Simulationsressourcen

Eine Empfehlung für die Definition von Simulationsressourcen ist es, diese direkt dem Simulationscontainer unterzuordnen, so wie dies auch aus dem Quelltext 3.13 zu entnehmen ist.

Wiederum wurde Gebrauch von den Erweiterungsmöglichkeiten des Designers der WPF gemacht, sodass diese Funktionalität in Form eines Editors nachgerüstet wurde, womit dem Entwickler lästige Tipparbeit abgenommen und ihm zudem ein einheitliches Entwurfsmuster vorgegeben wird.

Über die XAML-Markuperweiterung *StaticResource* können Aktivitäten auf eine gemeinsame Simulationsressource zugreifen. Beispielhaft dargestellt wird dies im Quelltext 3.14.

⁸Da es hier bezüglich der verwendeten Begriffe zu Verwirrung kommen kann, erfolgt in diesem Abschnitt eine Unterscheidung zwischen Ressourcen, wie sie im Kontext der WPF-Technologie eingesetzt werden und Simulationsressourcen, wie sie für die Modellierung in der SimNetUI-Bibliothek Anwendung finden.

```
1 <my:Wait Name="wait1">
2   <my:Wait.ResourceDependencies>
3     <my:ResourceDependency Count="5" Resource="{StaticResource Angestellter}" />
4   </my:Wait.ResourceDependencies>
5 </my:Wait>
```

Listing 3.14: XAML-Quellcode für den Zugriff auf Simulationsressourcen

Die Klasse *ResourceDependency* dient als Container, um Ressourcen Aktivitäten oder Entitäten zuzuweisen. Mittels des Attributs *Count* kann in XAML die Anzahl der benötigten Elemente festgelegt werden. Über die Eigenschaft *Resource* sollte eine Referenz auf eine Simulationsressource gesetzt werden.

3.5 Begleitobjekte

Neben den bisher betrachteten Komponenten, die für den Aufbau eines Simulationsmodells unentbehrlich sind, können Begleitobjekte dazu beitragen, die Präsentation einer Simulation erheblich aufzuwerten. Begleitobjekte sind Objekte, die auf das eigentliche Simulationsmodell nicht einwirken, aber in der Lage sind, Zustände für solch ein Modell innerhalb einer Simulation grafisch zu virtualisieren. Hierzu werden Begleitobjekte mit Aktivitäten durch Datenbindung in eine Beziehung gesetzt.

3.5.1 QueueCompanion

Das Begleitobjekt für Warteschlangen stellt derzeit das einzige implementierte Begleitobjekt für die SimNetUI-Bibliothek dar. Wie bereits im Kapitel 3.2.1.1 aufgezeigt, besitzen ausgewählte Aktivitäten eine eingebaute Warteschlange. Über eine Datenbindung zu genau solchen Aktivitäten wird dem Begleitobjekt ermöglicht, die Warteschlange dieser Aktivitäten darzustellen. Konkret bedeutet dies, dass die grafische Repräsentation aller Entitäten, die sich in der Warteschlange einer an das Begleitobjekt gebundene Aktivität befinden, in der Reihenfolge ihrer Sortierung innerhalb der Warteschlange von rechts nach links angeordnet, dargestellt werden.

Wiederum soll demonstriert werden, wie diese Datenbindung im XAML-Code zu bewerkstelligen ist.

```
1 <my:Wait Name="wait1" />
2 <my:QueueCompanion Name="queueCompanion1" ActivityQueue="{Binding ElementName=wait1}"
  />
```

Listing 3.15: XAML-Quellcode zur Demonstration der Datenbindung eines Begleitsobjektes für Warteschlangen

Aus Quelltext 3.15 wird ersichtlich, dass das *ActivityQueue* Property des *QueueCompanion* Objektes eine Referenz auf eine Aktivität mit eingebauter Warteschlange benötigt, welche es in diesem Beispiel über eine Datenbindung erhält.

3.6 Statistiken

Für die Auswertung von Simulationen sind Statistiken über den Verlauf einer Simulation unentbehrlich. Die Konzeption für die SimNetUI-Bibliothek sieht vor, dass Statistiken für Aktivitäten und Ressourcen jederzeit abrufbar sind. Hierzu ist es vorgesehen, dass WPF-Entwickler Statistiken an beliebiger Stelle mittels Datenbindung sich anzeigen lassen können. Es ist angedacht, dieses Konzept für eine spätere Version der SimNetUI-Bibliothek auch auf Entitäten auszuweiten.

3.6.1 Aktivitäten

Jede Aktivität besitzt ein eigenes Property *Statistics*. Da sich Statistiken von Aktivität zu Aktivität unterscheiden können, existiert für jede Aktivität jeweils eine eigene Statistik-Klasse. Statistiken sind schreibgeschützt, da diese für Simulationsentwickler nicht zum Bearbeiten vorgesehen sind. Im folgenden Beispiel wird demonstriert, wie über Datenbindung auf statistische Informationen zugegriffen werden kann.

```
1 <my:Generator Name="generator1" />
2 <TextBlock Text="{Binding ElementName=generator1, Path=Statistics.DepartedEntities,
  StringFormat='Generated Entities: {0}'}" />
```

Listing 3.16: XAML-Quellcode zur Demonstration der Datenbindung an Statistiken von Aktivitäten

3.6.2 Ressourcen

Ebenso wie Aktivitäten besitzen Ressourcen ein Property *Statistics*. Im Quelltext 3.17 wird demonstriert wie auf Statistiken von Ressourcen mittels Datenbindung zugegriffen

werden kann.

```
1 <my:SimulationContainer Name="simulation1">
2   <my:SimulationContainer.Resources>
3     <my:Resource x:Key="shopping cart" Capacity="10" />
4   </my:SimulationContainer.Resources>
5   <TextBlock Text="{Binding Source={StaticResource shopping cart}, Path=
      Statistics.AvailableResources}" />
6 </my:SimulationContainer>
```

Listing 3.17: XAML-Quellcode zur Demonstration der Datenbindung an Statistiken von Ressourcen

3.7 Simulationssteuerung

Notwendigerweise müssen Applikationen, die auf Basis der SimNetUI-Bibliothek entwickelt werden, mit Bedienelementen zur Steuerung der Simulation ausgestattet werden. In der gegenwärtigen Version sind die Bedienmöglichkeiten allerdings noch stark eingeschränkt. Klar ist, dass die Steuerung einer Simulation nur über einen zugehörigen Simulationscontainer funktionieren kann. Zum einen kann eine Datenbindung zu Eigenschaften des Simulationscontainers aufgebaut werden, zum anderen besteht aber auch die Möglichkeit, eine Bindung an WPF-Kommandos aufzubauen.

3.7.1 Kommandos

In der WPF stellen Kommandos eine bequeme Möglichkeit dar, um Aktionen an Bedienelemente zu binden. Einige wesentliche Vorteile, die dazu geführt haben, auch in der SimNetUI-Bibliothek von dieser Technik Gebrauch zu machen, sollen zunächst erörtert werden.

- Steuerelemente können direkt im XAML-Dokument an Kommandos gebunden werden. Entwickler, die sich nur der Kommandos bedienen, müssen keinen zusätzlichen Programmcode schreiben. Darüber hinaus kann eine beliebige Anzahl von Bedienelementen an dasselbe Kommando gebunden werden.
- Kommandos besitzen eine erweiterte Funktionalität, die es erlaubt, auf Zustände zu reagieren. Wenn es Bedingungen gibt, die die Ausführung von Aktionen verhindern, können zugehörige Kommandos deaktiviert werden. WPF-Steuerelemente sind entsprechend in der Lage, zu reagieren und diese veränderten Zustände dem Nutzer optisch anzuzeigen.
- Durch die Verwendung von Standardkommandos stehen darüber hinaus lokalisierte Bezeichnungen für Bedienelemente zur Verfügung.

3.7.1.1 Simulation starten

Das von der WPF vordefinierte Kommando *MediaCommands.Play* wird für den Start einer Simulation verwendet. Sollte die Simulation laufen, ist dieses Kommando deaktiviert, was dazu führt, dass eine an das Kommando gebundene Schaltfläche nicht mehr betätigt werden kann. Wie im Quelltext 3.18 demonstriert, ist die Schaltfläche an den lokalisierten Text des Kommandos gebunden, sodass je nach Einstellung im Betriebssystem die Sprache des Nutzers als Bezeichnung für das Bedienelement verwendet wird.

```
1 <Button Content="{Binding RelativeSource={RelativeSource self},Path=Command.Text}"
  Command="MediaCommands.Play" CommandTarget="{Binding ElementName=simulation1}" Name=
  "btnStart" />
```

Listing 3.18: Kommandobindung für ein Bedienelement, um eine Simulation zu starten

3.7.1.2 Simulation beenden

Über das Kommando *MediaCommands.Stop* kann es einem Bedienelement ermöglicht werden, eine Simulation vorzeitig zu beenden. Dieses Kommando ist nur aktiv, wenn die Simulation läuft. Ähnlich wie das Kommando *MediaCommands.Play* kann das Kommando *MediaCommands.Stop* an ein Bedienelement gebunden werden und ebenso wird der Darstellungstext der Schaltfläche in die Sprache des Nutzers übersetzt.

```
1 <Button Content="{Binding RelativeSource={RelativeSource self},Path=Command.Text}"
  Command="MediaCommands.Stop" CommandTarget="{Binding ElementName=simulation1}" Name=
  "btnEnd" />
```

Listing 3.19: Kommandobindung für ein Bedienelement, um eine Simulation zu beenden

3.7.2 Simulationseigenschaften

3.7.2.1 Animationsgeschwindigkeit

Auch auf die Animationsgeschwindigkeit kann Einfluss genommen werden. Der Simulationscontainer stellt hierzu ein eigenes Property zur Verfügung mit dem es möglich ist, einen Beschleunigungsfaktor zu bestimmen. Im Normalfall dauert jede Bewegung einer Entität von einer Aktivität zur nächsten eine Sekunde. Mittels XAML-Code können Steuerelemente an dieses Property gebunden werden.

```
1 <Slider Name="slider1" Width="260" Value="{Binding Path=AnimationClockSpeed,
    ElementName=simulation1}" TickFrequency="1" Maximum="25" />
```

Listing 3.20: Datenbindung für ein Bedienelement, um die Animationsgeschwindigkeit zur Laufzeit zu manipulieren

3.8 Designer Erweiterungen

In diesem Abschnitt soll die Benutzung der Erweiterungen, die für den Visual Studio WPF-Designer entwickelt worden sind, thematisiert werden. Von Anbeginn war es eine wichtige Zielstellung dieser Arbeit, eine möglichst einfache Entwicklung von Simulationsmodellen zu ermöglichen. Um so erfreulicher war es festzustellen, dass für den WPF-Designer ein Framework zur Verfügung gestellt wird, mit dessen Hilfe eine Erweiterung des WPF-Designers problemlos möglich ist. Damit lassen sich viele Programmierarbeiten, welche andernfalls nur manuell durch schreiben von Programmcode zu bewerkstelligen gewesen wären, durch grafische Bedienelemente erledigen. Hierdurch wird die SimNetUI-Bibliothek für Programmierer wesentlich zugänglicher, da sich die Verwendung der SimNetUI-Bibliothek auch ohne eingehende Betrachtung der Dokumentation schneller erschließen lässt.

Zusätzlichen Funktionalitäten werden über eine Bedienleiste, die durch auswählen des Simulationscontainers eingeblendet werden, zur Verfügung gestellt.

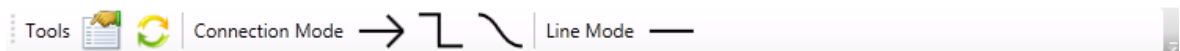


Abbildung 3.2.: Bedienleiste zum Zugriff auf Funktionalitäten des Designers für die Entwicklung von Simulationsmodellen mit der SimNetUI-Bibliothek

3.8.1 Aufbau der Modelltopologie

3.8.1.1 Verbindungsmodus

Um eine Verbindung zwischen zwei Aktivitäten aufzubauen, muss zunächst eine Aktivierung des Verbindungsmodus erfolgen. Dies geschieht durch Anwählen einer der 3 Schaltflächen, die sich hinter dem Bezeichnungsfeld *Connection Mode* in der Bedienleiste befinden. Im Verbindungsmodus werden die Verbindungsstellen der Aktivitäten

farblich hervorgehoben. Dabei stehen rote Bereiche für Verbindungsstellen für ausgehende Verbindungen und blaue Bereiche für Verbindungsstellen für eingehende Verbindungen. Neue Verbindungen lassen sich ausschließlich zwischen Verbindungsstellen unterschiedlichen Typs erstellen.

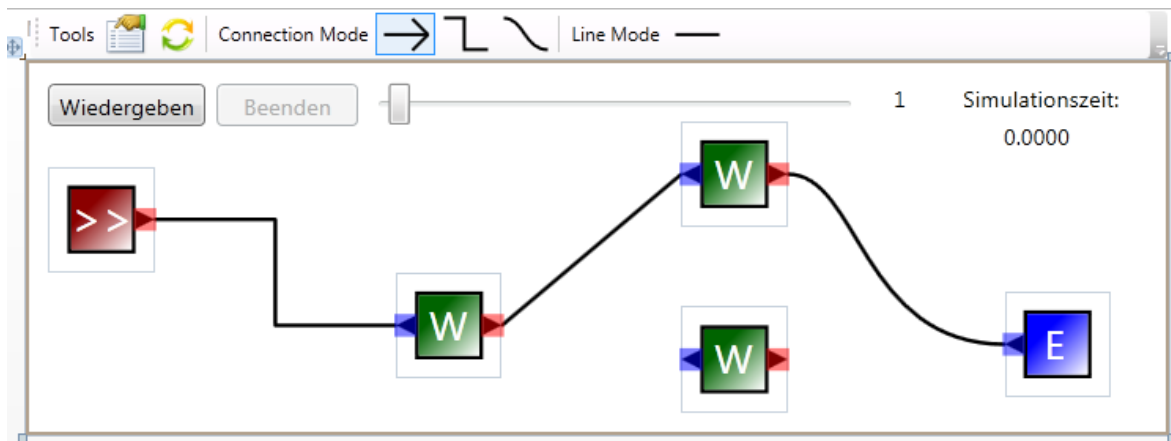


Abbildung 3.3.: Aktivierter Verbindungsmodus im Visual Studio WPF-Designer

Klickt man nun auf eine Verbindungsstelle, so wird eine Linie ausgehend vom Mittelpunkt des Verbindungsstück gezeichnet. Durch loslassen der Maustaste über eine weitere Verbindungsstelle wird eine Verbindung zwischen zwei Aktivitäten aufgebaut.

Drei unterschiedliche Darstellungsarten von Verbindungslinien stehen zur Wahl.




Variante	Beschreibung
	Die einfachste Variante zeichnet lediglich eine einfache Verbindungslinie zwischen 2 Verbindungsstellen.
	Die 2. Variante verwendet 2 Stützstellen, wodurch diagonale Linien vermieden werden.
	Die 3. Option verwendet Splines, um eine Verbindung zwischen zwei Aktivitäten darzustellen.

Tabelle 3.6.: Sortierungsarten von Warteschlangen

3.8.1.2 Linienmodus

Eigenschaften bestehender Verbindungen können im Linienmodus bearbeitet werden. Der Linienmodus wird durch anwählen der Schaltfläche hinter dem Bezeichnungsfeld *Line mode* aktiviert. Befindet sich der Mauszeiger über einer Verbindung, kann durch einen Rechtsklick auf diese Verbindung ein Kontextmenü geöffnet werden. Neben der

Möglichkeit die Darstellungsform der Linie zu verändern, besteht außerdem die Möglichkeit eine Linie zur Laufzeit der Anwendung ausblenden zu lassen. Durch Auswahl dieser Option wird die Linie zur Entwicklungszeit im Visual Studio WPF-Designer gestrichelt dargestellt. Weiterhin lässt sich über dieses Kontextmenü eine bestehende Verbindung zwischen zwei Aktivitäten löschen.

3.8.2 Werkzeuge

3.8.2.1 Ressourcen Editor

Für die Erstellung von Simulationsressourcen wird ebenso ein Editor zur Verfügung gestellt. Im Abschnitt *Tools* in der Bedienleiste befindet sich hierzu eine Schaltfläche, die die Bezeichnung *Open Resource Editor* trägt. Die Bedienung dieses Editors ist relativ selbst erklärend. Jede Ressource ist durch eine Zeile in einer Liste dargestellt. Durch einen Doppelklick auf einen Zelleintrag lässt sich eine Eigenschaft einer Ressource bearbeiten. Neue Ressourcen können durch Betätigen der Schaltfläche *Add Ressource* erzeugt werden, sofern im Textfeld links neben dieser Schaltfläche eine gültige Bezeichnung für die neue Ressource gewählt worden ist.

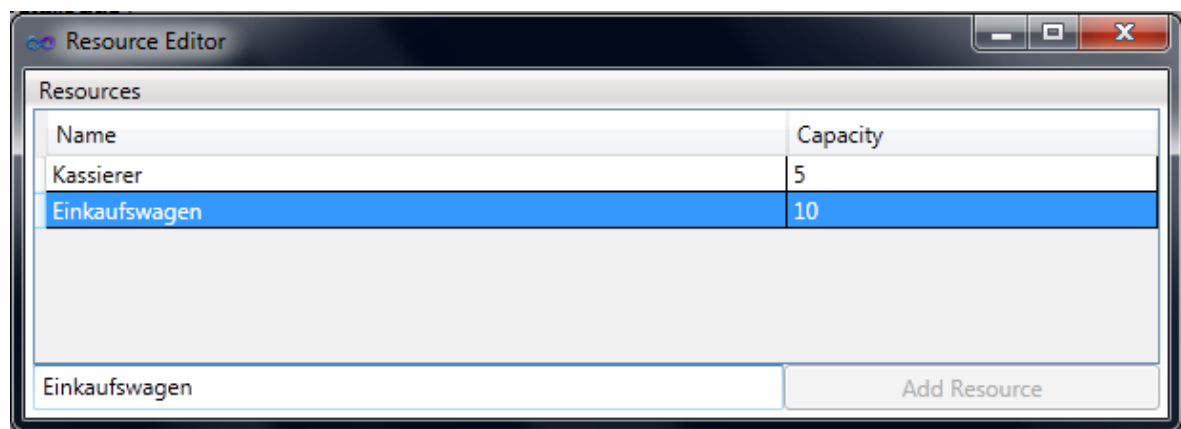


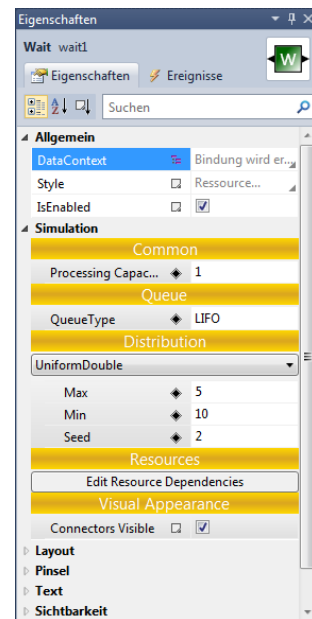
Abbildung 3.4.: Ressourcen Editor der SimNetUI-Bibliothek für den WPF-Designer

3.8.2.2 Aktualisierung der Darstellung

Sehr selten kann es zu Darstellungsfehlern kommen. In diesen Fällen ist es sehr nützlich, die Darstellung des Simulationscontainers aktualisieren zu können. Im Abschnitt *Tools* der Bedienleiste befindet sich daher eine Schaltfläche, die die Bezeichnung *Reload this control* trägt, wodurch bei einem Klick ein komplettes Neuzeichnen des Simulationscontainers erzwungen wird.

3.8.3 Property-Grid

Nicht unerwähnt sollen die zahlreichen Anpassungen des Eigenschaftsfensters bleiben. Da Objekte der WPF von Haus aus sehr viele Eigenschaften besitzen, die durch Vererbung auch auf Aktivitäten der SimNetUI-Bibliothek übertragen werden, hat es sich angeboten all die Properties, die auf das Simulationsmodell einwirken, in eine eigene Kategorie *Simulation* zu packen. Weiterhin ist es durch das WPF-Extensibility-Framework auch möglich, Einfluss auf die Art und Weise der Modifizierungen von Eigenschaften über das Property-Grid zu nehmen. Solche Änderungen sind an vielen Stellen notwendig gewesen, da die Standardwerkzeuge im Property-Grid von Visual Studio nicht für alle Konstellationen gleich gut geeignet sind.



4 Technische Realisierung der SimNetUI-Bibliothek

Die Anforderungen, die an eine moderne Software gestellt werden, können sehr vielfältig formuliert werden. Zum einen gibt es eine Reihe von Qualitätsmerkmalen, die gute Software kennzeichnen.

- Zuverlässigkeit
- Sicherheit
- Fehlerfreiheit
- Performance
- Stabilität
- Korrektheit
- Nutzerfreundlichkeit
- Erweiterbarkeit

Diese Liste stellt nur eine Auswahl von Qualitätsmerkmalen dar und kann zweifelsohne endlos erweitert werden. Nicht jede Software wird instande sein, jeden dieser Punkte ausreichend abzudecken. Häufig geht es vielmehr darum, den bestmöglichen Kompromiss zu finden, der für die Aufgabe, die die Software zu erfüllen hat, am sinnvollsten erscheint.

Softwareentwicklung stellt somit eine sehr anspruchsvolle Aufgabe dar, denn es gilt vieles miteinander abzuwägen. Um so wichtiger ist daher ein überlegter Softwareentwurf, denn ohne solides Fundament lässt es sich sehr schwer darauf aufbauen. Da im Laufe der Zeit verschiedene Programmierer an einem Projekt arbeiten können, ist zudem eine Dokumentation über die Prinzipien der Softwarearchitektur unerlässlich. Und genau diesem Punkt soll sich dieses Kapitel widmen.

Der sinnvolle Einsatz eines bewährten Entwurfsmusters erleichtert die Anwendungsentwicklung erheblich. Zum einen führt eine robuste Grundlage zu weniger Programmfehlern und stellt damit auch einen erheblicher Zeitgewinn ein, denn erfahrungsgemäß ist die Phase des so genannten *Debuggings* die zeitaufwendigste. Zum anderen erleichtern gute Entwurfsmuster die Weiterentwicklung einer Software durch unabhängige Programmierer, was nicht zuletzt auch mit einer vorgegeben sinnvollen Organisation des Quellcodes zu tun hat.

Entwurfsmuster lassen sich allerdings nicht in jedem Fall beliebig anwenden, denn auch der technische Kontext einer Anwendung muss mit in Betracht gezogen werden. Als

Beispiel für ein Entwurfsmuster, welches sich für die Entwicklung von Anwendungen mit der WPF eignet, kann das MVVM-Pattern genannt werden. Es hat sich allerdings gezeigt, dass die Gegebenheiten, wie sie für die SimNetUI-Bibliothek vorliegen, es nicht ohne weiteres ermöglichen, auf Basis dieses Entwurfsmuster zu arbeiten. Denn die SimNetUI-Bibliothek stellt keine klassische Endnutzer-Anwendung dar, da sie als Entwicklungswerkzeug für die Entwicklung von Simulationsapplikationen konzipiert worden ist und in diesem Sinne auch lediglich als Sammlung von Benutzersteuerelementen für die WPF betrachtet werden kann. Wenngleich diese Bezeichnung der Komplexität der Funktionalitäten, die durch die Bibliothek bereitgestellt werden, nicht gerecht wird. Auch wenn es aus diesem Grunde als unmöglich erscheint, ein Entwurfsmuster vollständig zu übernehmen, so können ausgewählte Konzepte, insbesondere aus dem MVVM Entwurfsmuster, auf die besondere Ausgangslage angepasst werden.

4.1 Komponentenmodell der SimNetUI-Bibliothek

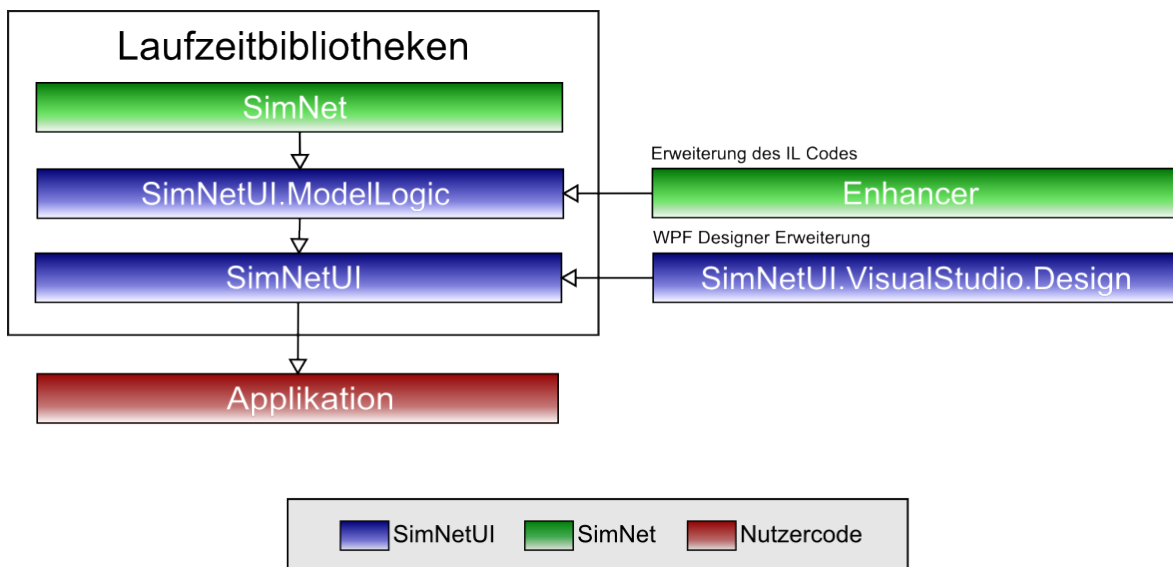


Abbildung 4.1.: Komponentenarchitektur der SimNetUI-Bibliothek

Als Basis für die Betrachtung der einzelnen Komponenten der Softwarearchitektur einer Simulationsanwendung soll die Abbildung 4.2 dienen. Jede einzelne dieser Komponenten ist als eigenes Visual Studio Projekt mit der Programmiersprache C# realisiert worden. Die Pfeile in der Grafik stellen die Beziehung zwischen den Komponenten dar.

Die SimNet-Bibliothek, welche aus einer Diplomarbeit aus dem Jahre 2005 mit dem Titel *Untersuchung zur Einbettung von Sprachelementen der prozessorientierten Simulation in C# unter .Net* an der HTW-Dresden hervorgegangen ist, stellt als unterste

Programmcode im UI-Thread nur noch ausgeführt, wenn Änderungen auftreten, die auf die Darstellung einwirken, wie beispielsweise die Initialisierung einer Animation oder wenn Code von Anwendungsentwicklern auf das Geschehen der Simulation einwirken soll. Der Thread der Modellschicht wartet nun, bis er vom UI-Thread signalisiert bekommt, dass er weiter arbeiten darf. Dieser Sachverhalt ist etwas komplexer und soll daher in Form einer Grafik veranschaulicht werden.

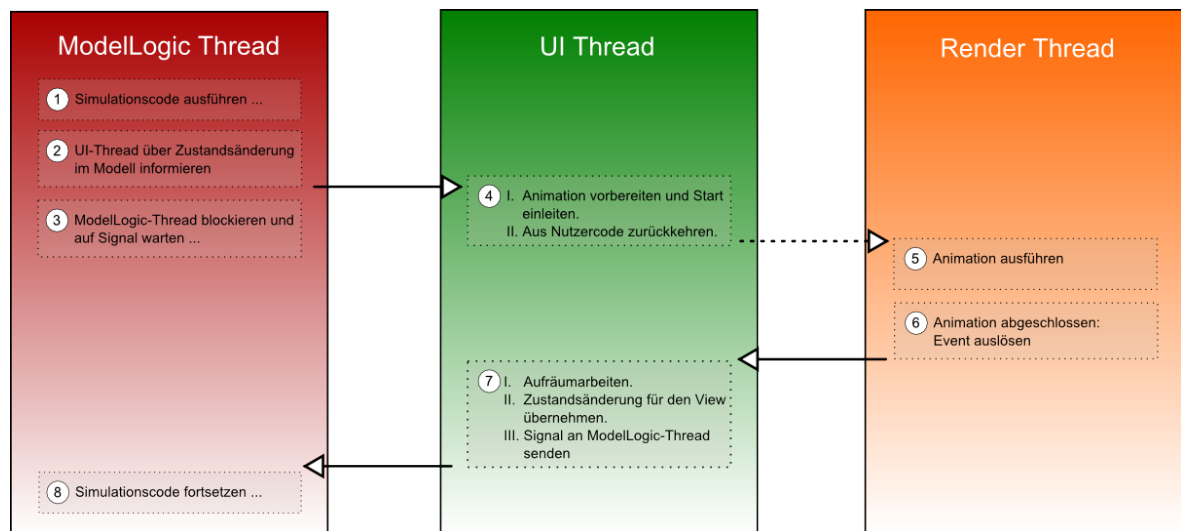


Abbildung 4.2.: Kommunikation zwischen Threads

Die meisten Klassen der Ansichtsschichtkomponente erben von der WPF-Klasse *DependencyObject*. Eine Eigenschaft von instanziierten *DependencyObjects* ist es, dass diese an den Thread gebunden sind, in welchem sie erstellt worden sind. Somit können Objekte aus der Ansichtsschicht nicht in einem separaten Thread verwendet werden. Daher ist es ungeheuer praktisch, dass die Modellschicht ihre eigene von der grafischen Darstellung losgelöste Repräsentation des Simulationsmodells besitzt.

4.2 Realisierung der Ansichts- und Modellebene

Nachdem betrachtet worden ist, weshalb eine Aufteilung des Programmcodes auf mehrere Komponenten notwendig ist, werden nun Implementationsdetails der Ansichts- und Modellschicht beleuchtet. Aufgrund des engen Zusammenhangs beider Ebenen erscheint eine gleichzeitige Betrachtung sinnvoll.

4.2.1 Beziehung zwischen Ansichts- und Modellschicht

Ein Grundprinzip, das bei der Ausarbeitung der Konzepte stets im Vordergrund stand, ist die möglichst lose Kopplung der Modellschicht an die Ansichtsschicht. Eine Auswirkung dieser Entscheidung ist darin zu erkennen, dass die Modellebene keine Verweise auf die Ansichtsebene besitzt. Somit besteht nur eine Abhängigkeit von der Ansichtsschicht zur Modellschicht, keinesfalls aber in umgekehrter Richtung. In der Softwareentwicklung spricht man deshalb auch von einem *Schichtenmodell*. Diese klare Trennung erleichtert es, die Abläufe innerhalb der SimNetUI-Bibliothek zu verstehen. Auf diese Weise ist es zumindest theoretisch möglich, einen Komponententest (engl. *unit test*) speziell für die Modellschicht zu implementieren.

Da Ansichts- und Modellschicht jeweils ihre eigene Repräsentation des Simulationsmodells besitzen, existieren für die meisten Klassen aus der Ansichtsebene in der Modellebene entsprechende korrespondierende Klassen. Ausnahmen bilden Elemente, die nur für die Darstellung wichtig sind, wie dies bei den Begleitobjekten der Fall ist. Eine typische Implementation einer Klasse aus der Ansichtsschicht sieht somit eine Referenz auf eine Klasse der Modellschicht vor. Modellschichtklassen sind immer am Suffix *ML* klar erkennbar. Bei der Referenz auf eine Modellschichtklasse handelt es sich selbstverständlich um ein Implementationsdetail, welches unter keinen Umständen durch Code von Applikationsebene erreichbar sein darf.

```
1 internal static readonly DependencyProperty ModelLogicProperty =
2     DependencyProperty.Register("ModelLogic",
3         typeof (ActivityBaseML), typeof (ActivityBase),
4         new FrameworkPropertyMetadata(OnModelLogicPropertyChanged));
5
6 [Browsable(false)]
7 internal ActivityBaseML ModelLogic
8 {
9     get { return (ActivityBaseML) GetValue(ModelLogicProperty); }
10    set { SetValue(ModelLogicProperty, value); }
11 }
```

Listing 4.1: Implementation des ModelLogic-Properties für Aktivitäten (Quellcodeauszug aus der Klasse *SimNetUI.Activities.Base.ActivityBase*)

Eine Instanz einer zugehörigen Modellklasse wird idealerweise im Konstruktor der Ansichtsklasse erstellt.

4.2.1.1 Datenbindung zwischen Modell- und Ansichtsschicht

Klassen in der Modellschicht besitzen alle eine gemeinsame Basisklasse², die das `INotifyPropertyChanged` Interface implementiert. Dieses Interface ist eine Voraussetzung, um Datenbindung für Klassen zu ermöglichen, die nicht von `DependencyObject` erben und somit auch keine `DependencyProperties` implementieren können. Damit ist eine wichtige Grundlage geschaffen, um einen reibungslosen Datenaustausch zwischen der Modell- und der Ansichtsschicht zu ermöglichen.

Wie aus dem Quelltext 4.1 ersichtlich ist, besitzt das `ModelLogic` Property immer eine Callback-Funktion, die aufgerufen wird, sobald sich der Wert des Properties verändert hat. Diese Callback-Methode wird unter anderem dafür verwendet, die Datenbindung zu initialisieren, mit welcher Eigenschaften zwischen der Modellschicht und der Ansichtsschicht miteinander abgeglichen werden. Der folgende Auszug aus der `SimNetUI`-Bibliothek soll zeigen, wie dies funktionieren kann.

```
1 private static void OnModelLogicPropertyChanged(DependencyObject obj,
2     DependencyPropertyChangedEventArgs e)
3 {
4     var activityBase = obj as ActivityBase;
5     var activityBaseML = e.NewValue as ActivityBaseML;
6     if (activityBaseML != null)
7     {
8         // set up binding
9         activityBase.SetUpBinding(NameProperty);
10    }
11 }
```

Listing 4.2: Datenbindung von Properties aus der Ansichtsschicht zu Properties aus der Modellschicht (Quellcodeauszug aus der Klasse `SimNetUI.Activities.Base.ActivityBase`)

Es ist Konvention, dass der Name eines Properties in der Modellschicht dem Namen des Properties der Ansichtsschicht entspricht. Auf diese Weise können die wichtigen Informationen, die für die Datenbindung Voraussetzung sind, vollständig aus den Metadaten des *Dependency-Properties* gewonnen werden.

Eine klare Festlegung darüber, welcher Code auf Eigenschaften einwirken darf, hilft dabei, die Richtung der Datenbindung der Properties zu bestimmen. So gilt allgemein die Regel, dass Properties, die der Anwendungsentwickler durch Verwendung der Benutzersteuerelemente aus der Ansichtsschicht setzen kann, eine einseitige Bindung von

²Die gemeinsame Basisklasse `ModelLogicBase` befindet sich im Namespace `SimNetUI.ModelLogic.Base`

der *Ansichts-* zur *Modellebene* besitzen. Sind Properties für Anwendungsentwickler nur lesbar, besteht hingegen eine einseitige Datenbindung von der Modellschicht zur Ansichtsschicht.

4.2.1.2 Ereignisorientierte Kommunikation

Für die Kommunikation zwischen der Modell- und der Ansichtsebene reicht eine Datenbindung zwischen Properties alleine nicht aus. Es mag erforderlich sein, dass die Modellschicht die Ansichtsschicht über Zustandsänderungen im Modell informiert oder Informationen aus der Ansichtsschicht anfordert. Ein Problem hierbei ist, dass die Modellschicht bekanntlich keine Verweise auf Objekte aus der Ansichtsschicht besitzt. Für diese Problematik empfiehlt sich eine Ereignisorientierte Kommunikation. Die Modellschicht besitzt Events, die bei Bedarf ausgelöst werden, um mit der Ansichtsschicht zu kommunizieren. Diese Ereignisse werden von der Ansichtskomponente abonniert.

Eine Sache, die hierbei berücksichtigt werden muss, ist dass eine Kommunikation über Threadbarrieren hinweg erfolgen muss. Wie bereits betrachtet, gilt für die meisten Objekte aus der Ansichtsschicht eine sogenannte Threadaffinität. Diese Gebundenheit von Objekten an den UI-Thread verhindert es, dass andere Threads Zugriff auf Eigenschaften dieser Objekte erhalten. Eine Methode aus der Ansichtsschicht, die ein Ereignis abonniert hat, welches sich in der Modellschicht befindet, wird daher zunächst im Thread der Modellschicht ausgeführt.

Eine Kommunikation zwischen verschiedenen Threads wird in der WPF über einen Dispatcher ³ ermöglicht. Über diesen Dispatcher können Nachrichten in Form eines Delagates an den UI-Thread versandt werden. Hierzu verwaltet der Dispatcher eine nach Priorität sortierte Liste von Nachrichten. Die meisten Klassen aus der WPF erben von der Klasse *DispatcherObject* und erhalten damit Zugriff auf die Funktionalitäten, die durch den Dispatcher über eine Referenz auf ein Dispatcherobjekt bereitgestellt werden. Auf diese Weise erben auch alle relevanten Klassen aus der Ansichtsschicht der SimNetUI-Bibliothek ohne Ausnahme von der *DispatcherObject* Klasse. Die Methoden *Invoke* und *BeginInvoke* stellen eine Möglichkeit dar, Nachrichten an den UI-Thread zu versenden. Während die Methode *Invoke* bei Aufruf den aufrufenden Thread blockiert, bis der Code, der mit der übermittelten Nachricht assoziiert wird, ausgeführt worden ist, erlaubt die Methode *BeginInvoke* eine parallele Ausführung von Programmcode in beiden Threads.

Da es sich hierbei um einen Vorgang handelt, der häufiger Anwendung findet, existiert für die Ansichtsebene eine Klasse *RegisterEvent*. Diese stellt Methoden zur Verfügung,

³[Wil10]

die eine Abonnieerung von Ereignissen vereinfachen. Über die überladene Methode *register* wird ein Wrapper für eine Methode erstellt, die ein Ereignis abonnieren soll. Dieser Wrapper kümmert sich darum, dass der Code im richtigen Thread ausgeführt wird.

```
1 public static Func<T1, R> register<T1, R>(Func<T1, R> ev, Dispatcher dispatcher)
2 {
3     var wrapper = (Func<T1, R>) delegate(T1 t1) { return (R) dispatcher.Invoke(ev, t1);
4         };
5     return wrapper;
6 }
```

Listing 4.3: Beispiel für die überladene Methode *register*. Quellcodeauszug aus der Klasse *SimNetUI.Util.RegisterEvent*.

Die Abonnieerung von Ereignissen kann erst erfolgen, nachdem eine Klasse aus der Ansichtsschicht eine entsprechende Modellklasse besitzt. Daher bietet sich hierfür erneut die Verwendung der Callback-Methode *OnModelLogicPropertyChanged* an. Illustriert wird dies im Quelltext 4.4.

```
1 private static void OnModelLogicPropertyChanged(DependencyObject obj,
2     DependencyPropertyChangedEventArgs e)
3 {
4     var activity = obj as ActivityRouteBase;
5     var activityML = e.NewValue as ActivityRouteBaseML;
6     if (activityML != null)
7     {
8
9         // register events
10        activityML.ProvideEntity +=
11            RegisterEvent.register<OutConnectorML, EntityML>(activity.
12                InteractionML_ProvideEntityML, activity.Dispatcher);
13    }
```

Listing 4.4: Beispiel für die Abonnieerung von Ereignissen aus der Modellschicht. Quellcodeauszug aus der Klasse *ActivityRouteBase*.

4.2.1.3 Threadsicherheit

Bei den bisher betrachteten Kommunikationsmechanismen stellt sich die Frage, ob eine threadsichere Kommunikation stattfindet. Threadsicherheit bedeutet, dass keine *Race-Condition* oder kein *Deadlock* zwischen den beiden Threads auftreten dürfen.

Datenbindung, wie sie in der SimNetUI-Bibliothek verwendet wird, ist im allgemeinen Threadsicher. Eine Ausnahme bilden Collections, die auch in der aktuellen .Net Version 4.0 nicht Threadsicher sind.⁴.

Für die SimNetUI-Bibliothek ist zu beachten, dass für Elemente einer Collection aus der Ansichtsschicht korrespondierende Elemente innerhalb einer Collection aus der Modellschicht existieren. Derzeit wird dieser Abgleich in der Ansichtsschicht im UI-Thread vollzogen. Die wichtigsten Collections in der SimNetUI-Bibliothek stellen zum einen die Liste der Aktivitäten dar, zum anderen deren Verbindungen zu anderen Aktivitäten. Weiterhin besitzen Generatoren einen sogenannten *Schedule*, der ebenfalls als Collection realisiert worden ist. Da aber die interne Modellrepräsentation für die Ansichtsschicht sowie die Modellschicht noch vor dem Start einer Simulation initialisiert wird, können keinerlei Probleme bezüglich der Threadsicherheit auftreten. Applikationsentwickler können während eines Simulationslaufs auf Zustände des Simulationsmodells einwirken, sobald ein Ereignis einer Aktivität ausgelöst worden ist. Wenn Nutzercode im UI-Thread ausgeführt wird, wird der Thread auf welchem die Modellschicht arbeitet blockiert. Ein dynamischer Umbau des Modells während eines Simulationslaufes wird allerdings nicht explizit unterstützt und kann Probleme verursachen. Normale Properties von Aktivitäten und Entitäten können aber jederzeit auch problemlos an dieser Stelle manipuliert werden.

⁴ Bea Stollitz, die für Microsoft als Entwicklerin für das WPF- und Silverlight-Framework tätig war, erläutert diesen Sachverhalt in einem Artikel auf ihrem Blog bezogen auf die 1. Version der WPF [Sto06].

4.2.2 Aktivitäten

4.2.2.1 Vererbungshierarchie

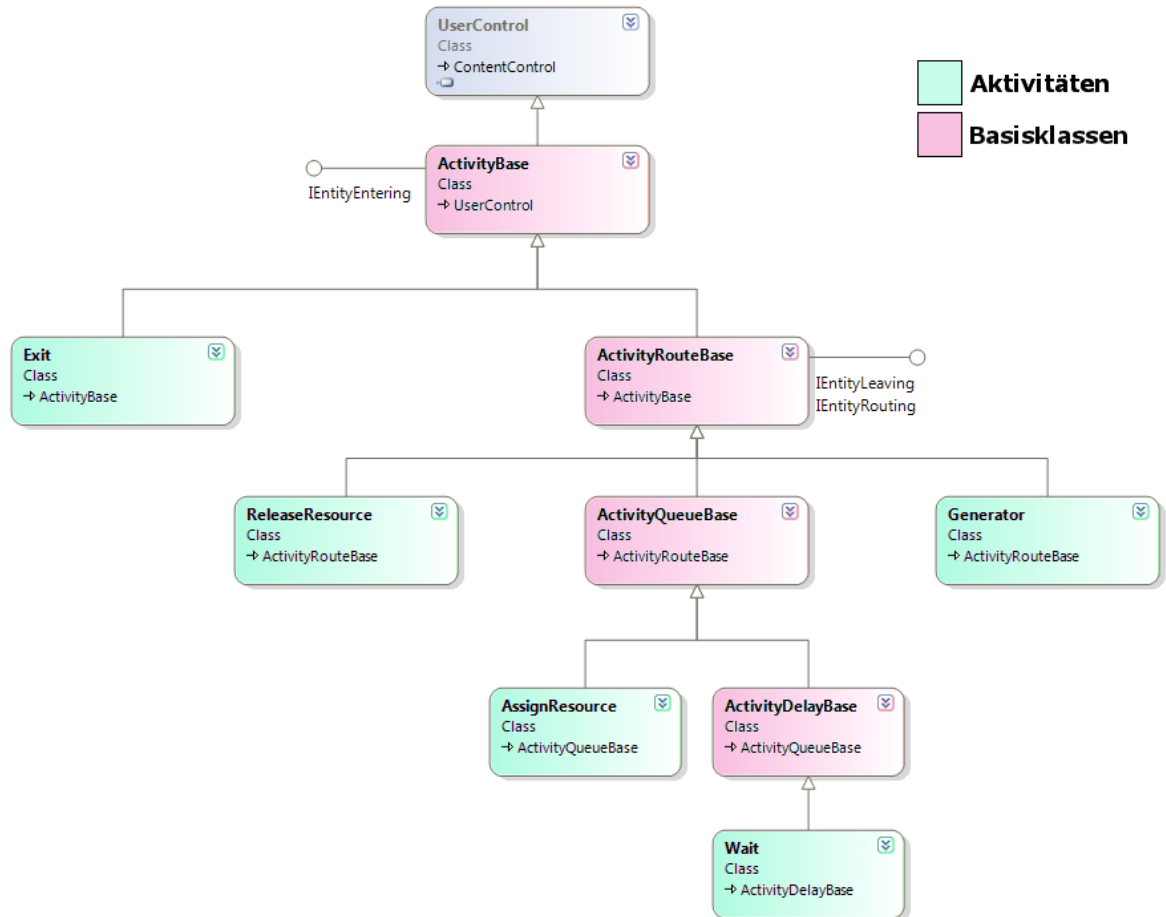


Abbildung 4.3.: Vererbungshierarchie für Aktivitäten aus der Ansichtsebene der SimNetUI-Bibliothek

Da Aktivitäten eine Reihe von gemeinsamen Eigenschaften besitzen, lässt sich das Paradigma der Vererbung aus der objektorientierten Programmierung sinnvoll anwenden. Aus Abbildung 4.4 ist eine Aufteilung von Basisklassen und Aktivitäten zu erkennen. Basisklassen stellen ein internes Implementationsdetail dar und sind nicht für die Verwendung im XAML-Code von Simulationsanwendungen vorgesehen. Für die Modellschicht existiert ein äquivalente Vererbungshierarchie.

4.2.2.1.1 *ActivityBase*

Die Klasse *ActivityBase* stellt die Grundlage für alle Aktivitäten aus der SimNetUI-Bibliothek dar. Diese erbt von der WPF-Framework-Klasse *UserControl*, wodurch eine

Verwendung von Aktivitäten als eigenes Steuerelement in der WPF ermöglicht wird. Die Klasse *ActivityBase* legt das Fundament für die Verwaltung von Verbindungen zwischen einzelnen Aktivitäten. Es werden die Datenstrukturen aufbereitet, die für das Zeichnen der Verbindungen zwischen den Aktivitäten intern verwendet werden. Außerdem wird ableitenden Klassen bereits ein Entwurfsmuster vorgegeben, indem eine Reihe von virtuellen Methoden bereitgestellt werden, die von ableitenden Klassen überschrieben werden können. Im Regelfall werden Aktivitäten sich nicht direkt von dieser Klasse ableiten. Die Aktivität *Exit* stellt hierbei eine Ausnahme dar, da nur eine eingeschränkte Kommunikation mit anderen Aktivitäten stattfindet. Denn die Aktivität *Exit* besitzt keine ausgehenden Verbindungen.

4.2.2.1.2 *ActivityRoute*

Die *ActivityRoute*-Klasse erweitert die Konzepte der *ActivityBase*-Klasse. Aktivitäten, die sich von dieser Basisklasse herleiten, besitzen die Möglichkeit, Entitäten an andere Aktivitäten weiterzuleiten. Durch die Implementation des Interfaces *IEntityRouting* können zudem Nutzer der SimNetUI-Bibliothek auf das Routing von Aktivitäten, die von der Klasse *ActivityRoute* erben, über das Event *EntityRouted* Einfluss nehmen. Weiterhin wird das Interface *IEntityLeaving* implementiert, welches das Event *EntityLeft* bereitstellt.

4.2.2.1.3 *ActivityQueueBase*

Aktivitäten die eine Warteschlange besitzen, erben von der Basisklasse *ActivityQueueBase*. Diese Klasse besitzt eine Liste aller Entitäten, die sich in der Warteschlange befinden. Über das Datenfeld *QueueType* kann Einfluss auf die Sortierung innerhalb dieser Liste genommen werden. Ableitende Klassen müssen an geeigneter Stelle die Methode *SortQueue()* aufrufen, um eine Neusortierung der Liste nach den gewählten Kriterien zu erlauben.

4.2.2.1.4 *ActivityDelayBase*

Der Aufgabenbereich von Aktivitäten, die von der Basisklasse *ActivityDelayBase* erben, ist in der Form definiert, dass ein Verbrauch von Simulationszeit ermöglicht wird. Die Aktivität *Wait* stellt derzeit die einzige Aktivität dieser Kategorie dar. Für eine Erweiterung der SimNetUI-Bibliothek bietet es sich an, auf dieser Basis weitere spezialisierte Aktivitäten zu implementieren.

4.2.2.2 Statistiken

Mit Ausnahme der Basisklassen besitzen alle Aktivitätsklassen ein Property *Statistics*. Da für jede Aktivität andere Statistiken kalkuliert werden, existiert für jede Aktivität in der Ansichts- sowie der Modellschicht eine eigene Statistikklasse. Die Vererbungshierarchie der Statistikklassen ist an die Vererbungshierarchie der Aktivitäten angelehnt.

Statistiken werden ausschließlich in der Modellschicht berechnet. Mittels *Data-Binding* werden die Werte in der Ansichtsschicht aktualisiert.

Die abstrakte Basisklasse *StatisticInfoBaseML* aus der Modellschicht stellt zwei virtuelle Methoden zur Verfügung, die von ableitenden Klassen überschrieben werden sollten.

```
1 abstract public class StatisticInfoBaseML : ModelLogicBase
2 {
3     internal virtual void Reset() {}
4     internal virtual void UpdateTimeBasedStatistics() {}
5 }
```

Listing 4.5: Implementation der *StatisticInfoBaseML*-Klasse

Die Methode *Reset* ist für das Rücksetzen aller Werte einer Statistikklasse verantwortlich. Die Methode *UpdateTimeBasedStatistics* dient zur Aktualisierung von Statistiken, deren Werte bei Fortschreiten der Simulationszeit angepasst werden müssen.

4.2.2.3 Kommunikation

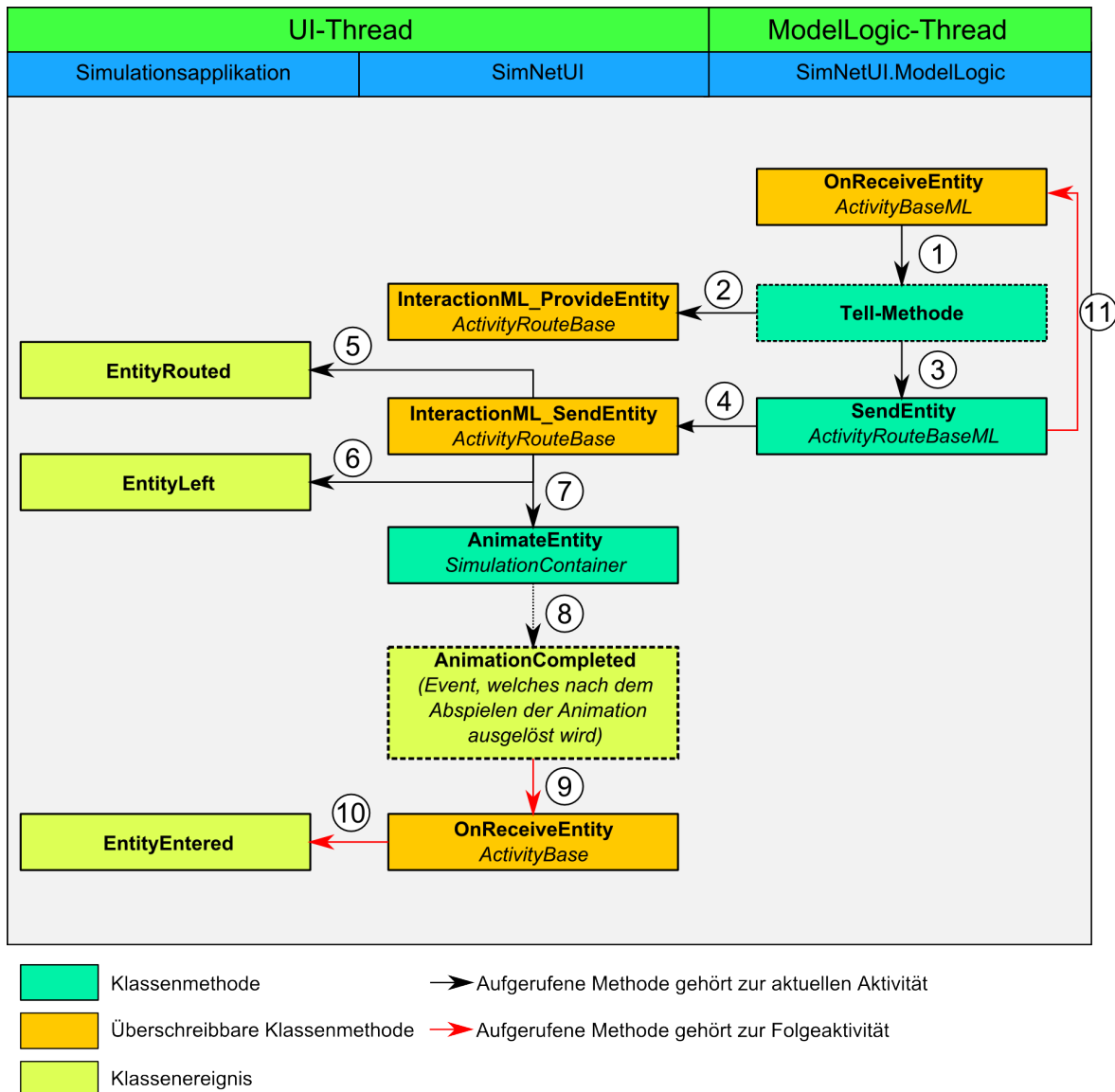


Abbildung 4.4.: Kommunikationszyklus von Aktivitäten

Der komplexe Ablauf der Kommunikation von Aktivitäten ist in der Abbildung 4.4 dargestellt. Der hier vorgestellte Ablauf kann, je nachdem wo sich eine Aktivität in der Vererbungshierarchie befindet, variieren. Bei der *Exit*-Aktivität, die von der Klasse *ActivityBase* erbt, existiert in der Ansichts- wie auch Modellschicht jeweils nur die Methode *OnReceiveEntity*. Auch wenn die *Generator*-Aktivität sich von der Klasse *ActivityBase* herleitet, besitzen diese Methoden (*OnReceiveEntity*) aus der Ansichts- und Modellschicht für diese Aktivität keine Relevanz. Da Generatoren keine eingehenden Verbindungen besitzen, wird die *OnReceiveEntity* Methode niemals aufgerufen. Aufgrund einer nicht vorhandenen Unterstützung einer Art von Mehrfachvererbung in C#

ist es sehr schwierig, eine konsistente Klassenhierarchie zu entwerfen, die auf solche Kompromisse nicht angewiesen ist.

4.2.2.3.1 *OnReceiveEntity*

Durch die Methode *OnReceiveEntity* werden Aktivitäten über die Ankunft neuer Entitäten informiert.

In der Ansichtsschicht ist es Aufgabe dieser Methode, über eintreffende Entitäten Buch zu führen. Aktivitäten, die eine Warteschlange besitzen, werden beispielsweise eintreffende Entitäten in der dafür vorgesehenen Datenstruktur⁵ aufbewahren, um später durch die Methode *InteractionML_ProvideEntity* die zugehörige Modellklasse einer Entität für die Modellschicht bereitzustellen. Außerdem wird das Event *EntityEntered* ausgelöst, wodurch Programmcode von Applikationsentwicklern, die sich der SimNetUI-Bibliothek bedienen, ausgeführt wird.

Die *OnReceiveEntity*-Methode aus der Modellschicht leitet die Ausführung des Simulationscodes einer Aktivität ein. Außerdem können Statistiken einer Aktivität an dieser Stelle aktualisiert werden.

4.2.2.3.2 *Tell-Methoden*

Es existiert kein konkretes Entwurfsmuster, welches vorschreibt wo und mit welcher Bezeichnung eine Tell-Methode einer Aktivität implementiert sein muss. Sofern eine Aktivität aber direkt auf das Simulationsgeschehen einwirken soll, ist eine Implementation einer Tell-Methode unausweichlich. Solch eine Tell-Methode muss für den Scheduler der SimNet-Bibliothek innerhalb der Methode *OnReceiveEntity* aus der Modellschicht bekannt gemacht werden.

```
1 internal override void OnReceiveEntity(InConnectorML targetConnectorML, OutConnectorML
   startConnectorML)
2 {
3     var logic = new NestedClassSimulationLogic(this);
4     Simulation.Tell(logic.wait, 0, 0, null);
5 }
```

Listing 4.6: Beispiel für die Registrierung einer Tell-Methode in der Modellschicht

⁵Damit ist das Dependency-Property *Queue* der Klasse *ActivityQueueBase* aus der Ansichtsschicht gemeint

Die Klasse *ActivityRouteBase* stellt ableitenden Klassen die Methode *GetProvideEntity* zur Verfügung, durch deren Aufruf ein Ereignis ausgelöst wird, welches von der *InteractionML_ProvideEntity*-Methode aus der Ansichtsschicht abonniert worden ist. Für die Implementation einer Tell-Methode ist es somit vorgesehen, auf diesem Wege an eine Referenz einer Entität zu gelangen. Hierdurch wird es der Tell-Methode ermöglicht, auf Attribute (Properties) von Entitäten einzuwirken. Im Anschluss wird die Methode *Send Entity* ausgeführt.

4.2.2.3.3 *SendEntity*

Die Methode *SendEntity* aus der Modellschicht hat die Aufgabe, die Nutzeroberfläche über Veränderungen im Modell zu informieren. Zunächst wird die Nutzeroberfläche über das Voranschreiten der Simulationszeit informiert. Daraufhin wird ein Ereignis ausgelöst, welches von der *InteractionML_SendEntity*-Methode aus der Ansichtsschicht abonniert worden ist. Im nächsten Schritt wird der Folgeaktivität durch Aufruf der Methode *OnReceiveEntity* das Eintreffen einer Entität signalisiert.

4.2.2.3.4 *InteractionML_SendEntity*

Die Methode *InteractionML_SendEntity* löst die Ereignisse *EntityRouted* und *EntityLeft* aus, wodurch Applikationsentwickler die Möglichkeit erhalten, auf das Simulationsgeschehen direkt einzuwirken. Im Anschluss wird die Methode *AnimateEntity* einer Instanz der Klasse *SimulationContainer* ausgelöst.

4.2.2.3.5 *AnimateEntity*

Diese Methode initiiert eine Animation, die die Bewegung einer Entität von einer Aktivität zu einer Zielaktivität visualisiert. In der WPF werden Animationen über *Storyboards* beschrieben. Eine Besonderheit ist es, dass Animationen in der WPF über einen speziellen Render-Thread gesteuert werden, welcher sich allerdings der Kontrolle von Programmierern, die sich der WPF bedienen völlig entzieht. Ein jedes *Storyboard* besitzt ein Ereignis *completed*, welches nach dem eine Animation abgespielt worden ist, ausgelöst wird.

4.2.2.3.6 *AnimationCompleted*

Das Ereignis *AnimationCompleted* wird genau an zwei Stellen ausgelöst. Einmal nachdem das *completed*-Event eines Storyboards, welches die Bewegung einer Entität animiert hat, ausgelöst worden ist. Da im Falle eines vorzeitigen Abbruchs eines Simulationslaufs das Event *completed* nicht mehr ausgelöst wird, wird das *AnimationCompleted*-Event ebenso nach initiieren des Abbruchs im UI-Thread ausgelöst.

4.2.2.4 Konstruktion mit XAML

Aktivitäten, wie sie für die SimNetUI-Bibliothek konzipiert worden sind, sind grafische Komponenten, die sich auf einem Formular, den Simulationscontainer, anordnen lassen. Die Darstellung dieser Aktivitäten wird, wie in der WPF üblich, mittels XAML beschrieben. Aktivitäten werden, wie im Kapitel 4.2.2.1 bereits betrachtet, als sogenannte User-Controls implementiert. Eine Aktivität erbt stets von einer der aus Abbildung 4.4 gekennzeichneten Basisklassen. Der Quellcode einer Aktivität besteht aus einer XAML-Datei für die Darstellung und einer C# Datei für den Programmcode der Aktivität.

In diesem Abschnitt soll anhand der Aktivität *Wait* beispielhaft dargestellt werden, wie Aktivitäten in der SimNetUI-Bibliothek mittels XAML implementiert werden.

4.2.2.4.1 *Namensräume*

Namensräume sind ein Konzept des XML-Standards. XML-Dokumenten sind häufig Namensräume zugeordnet, die meist auf ein entsprechendes XML-Schema verweisen, wodurch klar geregelt ist, welche Elemente und Attribute in einem XML-Dokument zulässig sind. XAML erweitert dieses Konzept insofern, dass XAML Namensräume sich beispielsweise auch auf .Net Namensräume beziehen können, wodurch der komplette Satz von öffentlichen Klassen, die einem .Net Namespace zugeordnet sind, für die Verwendung im XAML-Code verfügbar gemacht wird.

Beim Anlegen einer XAML-Datei in Visual Studio, werden einige elementare Namensräume bereits dem Dokument hinzugefügt. Dabei handelt es sich unter anderem um XAML-Namensräume, die sich auf Namensräume der WPF-Klassenbibliothek beziehen, bzw. um XAML-Namensräume, die für den Designer von Visual Studio dienlich sind.

Somit muss für Klassen, die in der SimNetUI-Bibliothek implementiert werden und im XAML-Code einer Aktivität verwendet werden sollen, ein entsprechender Namensraum

im XAML-Dokument hinzugefügt werden.

```
1 <base:ActivityDelayBase
2   x:Class="SimNetUI.Activities.Controls.Wait"
3   xmlns="http://schemas.microsoft.com/winfx/2006/XAML/presentation"
4   xmlns:x="http://schemas.microsoft.com/winfx/2006/XAML"
5   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7   xmlns:base="clr-namespace:SimNetUI.Activities.Base"
8   xmlns:con="clr-namespace:SimNetUI.Activities.ControlParts.Connection"
9   mc:Ignorable="d"
10  DataContext="{Binding RelativeSource={RelativeSource Self}}"
11  TooltipService.ShowDuration="120000" />
```

Listing 4.7: Namensräume der Aktivität *Wait*. Auszug aus der Datei *Wait.xaml*

Mittels eines Verweises über ein Attribut, dem das Präfix *xmlns* vorangestellt ist, wird die Verwendung von Elementen eines Namensraums innerhalb eines XAML-Dokuments zulässig. Durch die Einbindung des Namensraums *base* kann die Basisklasse *ActivityDelayBase*, von der sich die Aktivität *Wait* herleitet, verwendet werden. Der Namensraum *con* ist ebenfalls von zentraler Bedeutung, da hierdurch die Verwendung von Verbindungsstellen, über welche Aktivitäten miteinander verbunden sind, erst ermöglicht wird.

4.2.2.4.2 Darstellung

Um die Präsentation von Simulationsmodellen aufzuwerten, besitzen Aktivitäten das Property *VisualAppearanceTemplate*, über das Nutzer der SimNetUI-Bibliothek das Aussehen einer Aktivität verändern können. Die Standarddarstellung der *Wait*-Aktivität kann dem Quelltext 4.8 entnommen werden.

```
1 <base:ActivityDelayBase.VisualAppearanceTemplate>
2   <Border Name="border" Height="36" Width="36" BorderThickness="2">
3     <Border.Style>
4       <Style TargetType="{x:Type Border}">
5         <Setter Property="BorderBrush" Value="Black" />
6         <Style.Triggers>
7           <DataTrigger Binding="{Binding Path=
8             Statistics.IsProcessing,UpdateSourceTrigger=PropertyChanged,Mode=OneWay}"
9             Value="true">
10            <Setter Property="BorderBrush" Value="Red" />
11          </DataTrigger>
12        </Style.Triggers>
13      </Style>
14    </Border.Style>
15    <Border.Background>
16      <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
```

```

15         <GradientStop Offset="0" Color="DarkGreen" />
16         <GradientStop Offset="0.3" Color="DarkGreen" />
17         <GradientStop Offset="1" Color="White" />
18     </LinearGradientBrush>
19 </Border.Background>
20 <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Text="W"
    Foreground="White" FontSize="22" FontStretch="ExtraExpanded" Width="21" Height="
    32" />
21 </Border>
22 </base:ActivityDelayBase.VisualAppearanceTemplate>

```

Listing 4.8: Standarddarstellung der *Wait*-Aktivität, ohne Verbindungsstücke. Auszug aus der Datei *Wait.xaml*

Der Teil der Darstellung, der von Nutzern der SimNetUI-Bibliothek nicht verändert werden darf, gehört selbstverständlich nicht zu diesem Template. Dies gilt insbesondere für die Verbindungsstellen, die als Start- und Endpunkte von Verbindungen zwischen Aktivitäten dienen. Diese Verbindungsstellen müssen ebenfalls im XAML-Code einer Aktivität hinterlegt sein. Im Quelltext 4.9 ist die Struktur der *Wait*-Aktivität dargestellt. Das Darstellungstemplate aus Quelltext 4.8 wird in diesem Fall innerhalb einer 3x3 Gitterstruktur in der mittleren Zelle positioniert. Rechts befindet sich die Verbindungsstelle für ausgehende Verbindungen, wohingegen links sich die Verbindungsstelle für eingehende Verbindungen befindet.

```

1  <base:ActivityDelayBase.Content>
2  <Grid>
3      <Grid.RowDefinitions>
4          <RowDefinition Height="10" />
5          <RowDefinition Height="*/>
6          <RowDefinition Height="10" />
7      </Grid.RowDefinitions>
8      <Grid.ColumnDefinitions>
9          <ColumnDefinition Width="10" />
10         <ColumnDefinition Width="*/>
11         <ColumnDefinition Width="10" />
12     </Grid.ColumnDefinitions>
13     <ContentPresenter Grid.Row="1" Grid.Column="1" Content="{Binding
        VisualAppearanceTemplate,UpdateSourceTrigger=PropertyChanged}" />
14     <con:InConnector x:Name="In" Grid.Row="1" Grid.Column="0" HorizontalAlignment="
        Right" VerticalAlignment="Center" />
15     <con:OutConnector x:Name="Out" Grid.Row="1" Grid.Column="2" HorizontalAlignment="
        Left" VerticalAlignment="Center" />
16 </Grid>
17 </base:ActivityDelayBase.Content>

```

Listing 4.9: Aufbau der *Wait*-Aktivität. Auszug aus der Datei *Wait.XAML*

4.2.2.4.3 Tooltip

Damit Tooltips für Aktivitäten einem einheitlichen *Look and Feel* unterliegen, ist ein WPF-Style erstellt worden, der zu diesem Zweck von allen Aktivitäten verwendet wird. Um diesen WPF-Style verwenden zu können, muss das passende *Resource-Dictionary*, in das XAML-Dokument der Wait-Aktivität eingebunden werden.

```
1 <base:ActivityDelayBase.Resources>
2   <ResourceDictionary Source="Themes\Generic\Gls{xaml}" />
3 </base:ActivityDelayBase.Resources>
```

Listing 4.10: Einbindung eines Resource-Dictionaries für die *Wait*-Aktivität. Auszug aus der Datei *Wait.XAML*

Statistiken einer Aktivität werden durch Data-Binding über einen Tooltip verfügbar gemacht. Der Data-Context bezieht sich, wie im Quelltext 4.7 gezeigt, auf die Aktivität selbst. Somit muss für eine Datenbindung an einen statistischen Wert, jeweils nur noch der Pfad zum passenden Property angegeben werden.

```
1 <base:ActivityDelayBase.ToolTip>
2   <ToolTip Style="{StaticResource ToolTipStyle}">
3     <Grid>
4
5       <Grid.ColumnDefinitions>
6         <ColumnDefinition Width="120" />
7         <ColumnDefinition Width="60" />
8       </Grid.ColumnDefinitions>
9
10      <Grid.RowDefinitions>
11        <RowDefinition />
12        <RowDefinition />
13      </Grid.RowDefinitions>
14
15      <TextBlock Grid.Column="0" Grid.Row="0" Text="Currently processing:" />
16      <TextBlock Grid.Column="1" Grid.Row="0" Text="{Binding Mode=OneWay,Path=
17        Statistics.InWork}" />
18
19      <TextBlock Grid.Column="0" Grid.Row="1" Text="Currently in queue:" />
20      <TextBlock Grid.Column="1" Grid.Row="1" Text="{Binding Mode=OneWay,Path=
21        Statistics.InQueue}" />
22    </Grid>
23  </ToolTip>
24</base:ActivityDelayBase.ToolTip>
```

Listing 4.11: XAML-Quellcode für den Tooltip der Wait-Aktivität. Verkürzter Auszug aus der Datei *Wait.xaml*

4.2.3 Assembly Manifest

Programme und Bibliotheken, die für die .Net Plattform entwickelt werden, besitzen immer ein eigenes Manifest, welches mit Metadaten zur Beschreibung der Assembly ausgestattet werden kann⁶ Jede der Komponenten aus Abbildung 4.2 besitzt daher ihr eigenes Manifest. In diesem Zusammenhang ist die Verwendung spezieller Attribute, die als Metadaten im Manifest der Ansichtsschicht verwendet werden, zu erwähnen⁷.

4.2.3.1 XmlnsDefinition

Das Attribut *XmlnsDefinition* wird verwendet, um die Menge der Namensräume aus dem Programmcode der Ansichtsschicht, die die Klassen beinhalten, die für die Verwendung innerhalb des XAML-Quellcodes von Simulationsapplikationen vorgesehen sind, auf einen eigenen XAML-Namensraum abzubilden. Durch diese Maßnahme kann in der SimNetUI-Bibliothek der Programmcode sinnvoller organisiert werden, was zu einer besseren Abgrenzung von Klassen bezüglich deren Aufgaben und Funktionalitäten führt, ohne den XAML-Code von Simulationsapplikationen durch die Einbindung unzähliger komplizierter Namensräume zu überfrachten.

```
1 [assembly: XmlnsDefinition("SimNetUI", "SimNetUI.Controls")]
2 [assembly: XmlnsDefinition("SimNetUI", "SimNetUI.Activities.Controls")]
3 [assembly: XmlnsDefinition("SimNetUI",
4                             "SimNetUI.Activities.PropertyObjects.Connections")]
5 [assembly: XmlnsDefinition("SimNetUI",
6                             "SimNetUI.Activities.PropertyObjects.Distributions")]
7 [assembly: XmlnsDefinition("SimNetUI",
8                             "SimNetUI.Activities.PropertyObjects.Schedule")]
9 [assembly: XmlnsDefinition("SimNetUI",
10                            "SimNetUI.Activities.PropertyObjects.Resources")]
11 [assembly: XmlnsDefinition("SimNetUI", "SimNetUI.Companions.Controls")]
12 [assembly: XmlnsDefinition("SimNetUI", "SimNetUI.Entity")]
13 [assembly: XmlnsDefinition("SimNetUI", "SimNetUI.Resources")]
```

Listing 4.12: Abbildung der Namensräume aus der SimNetUI Bibliothek auf einen einheitlichen Namensraum zur Verwendung in Xaml

⁶Vgl. [Küh10] Kapitel 1.3

⁷Diese Attribute, die zur Beschreibung des Manifests verwendet werden, befinden sich in der Datei *AssemblyInfo.cs* im Verzeichnis *Properties*

4.2.3.2 InternalsVisibleTo

Möchte man einem Assembly Zugriff auf die als intern (*internal*) gekennzeichneten Member einer anderen Assembly gewähren, so kann man durch Verwendung des Attributs *InternalsVisibleTo* dies bewerkstelligen.

Die Assembly, die für die Bereitstellung der Erweiterungen für den WPF-Designer zuständig ist, erhält durch die Verwendung dieses Attributs Zugriff auf interne Member der Ansichtsschicht.

```
1 [assembly: InternalsVisibleTo("SimNetUI.VisualStudio.Design")]
```

Listing 4.13: Interne Member einer anderen Assembly verfügbar machen. Quellcodeauszug aus der Datei *AssemblyInfo.cs* aus der Ansichtsschicht.

4.3 Realisierung der Erweiterungen des WPF-Designers

Für die Erweiterung des WPF-Designers existiert ein Framework, welches mit einer eigenen API daher kommt. Mittels dieser API ist es möglich eigene Werkzeuge, Fenster, Adorner und eigene Eigenschaften-Editoren zu erstellen⁸. Die größte Schwierigkeit, die sich bei der Programmierung eigener Erweiterungen einstellt, ist die nur unzureichende Dokumentation dieser Technologie, da diese für den Großteil der WPF-Entwicklergemeinschaft eher ein Nischendasein darstellt. Literatur, die sich mit der Thematik eingehend befasst, ist daher Mangelware. Die umfangreichste Quelle für die Entwicklung von Erweiterungen stellt das Internet dar. Zu erwähnen ist hierbei insbesondere die Dokumentation aus der MSDN-Bibliothek, welche online abrufbar ist. Dennoch ist auch die MSDN-Bibliothek keine all umfassende Informationsquelle, die alles für die Entwicklung von Erweiterungen für den WPF-Designer notwendige Wissen in einer übersichtlichen Form darstellt. Eine gute Einarbeitung in diese Thematik ist durch das Studium von Beispielprojekten möglich, welche oft auch in der MSDN-Bibliothek referenziert werden⁹.

⁸Vgl. [MSD11b]

⁹Einige Beispielprojekte, die die Verwendung des WPF-Extensibility-Frameworks demonstrieren, sind zusätzlich auf der CD, welche dieser Diplomarbeit beiliegt, enthalten

4.3.1 Projekt für WPF-Designer Erweiterungen

Um eine bestehende WPF-Benutzersteuerelement-Bibliothek mit Erweiterungen für den Visual Studio WPF-Designer anzureichern, ist ein eigenes Projekt für die Programmierung dieser Erweiterungen zu erstellen.

Damit dies funktionieren kann, ist eine Konvention bezüglich der Benennung dieses Projektes einzuhalten. Gemäß dieser Richtlinie ist der Projektname *SimNetUI. VisualStudio.Design* gewählt worden. Die Ergänzung *VisualStudio.Design* zum Projektnamen zeigt an, dass die Erweiterungen sich nur auf den WPF-Designer von Visual Studio beziehen. Damit ist auch die Verwendung dieser Erweiterungen für den XAML-Designer des Softwareproduktes *Microsoft Expression Studio* ausgeschlossen.

Als Ausgabepfad für die kompilierte Assembly ist das selbe Verzeichnis anzugeben, in dem auch die DLL für das Projekt *SimNetUI* abgelegt wird. Die Assembly *SimNetUI.VisualStudio.Design* wird nur zur Entwicklungszeit von Simulationsapplikationen benötigt, die die *SimNetUI* Bibliothek referenzieren, und muss daher für Endanwender dieser Simulationsapplikationen nicht mit ausgeliefert werden.

4.3.2 Unterscheidung zwischen Entwicklungszeit und Laufzeit

Auch im Programmcode der Ansichtsschicht kann auf das Entwicklungserlebnis zur Entwicklungszeit Einfluss genommen werden. Daher ist eine Begriffsabgrenzung bezüglich der Entwurfszeit (*engl. design time*) und der Laufzeit (*engl. run time*) zwingend notwendig. Dabei gilt es sich zu vergegenwärtigen, dass der WPF-Designer zur Entwurfszeit auch Programmcode aus der *SimNetUI*-Bibliothek ausführen muss, um eine grafische Ansicht und Entwicklung von XAML-Code überhaupt erlauben zu können. Folglich wird allerdings auch Code ausgeführt, der für die korrekte Darstellung im Designer nicht notwendig ist.

Um ein Beispiel zu nennen, sei die Datenbindung von Properties aus der Ansichtsschicht zur Modellschicht erwähnt. Initialisiert man diese Datenbindung auch zur Entwurfszeit, wird in der 2010er Version des WPF-Designers *Cider* an einigen Stellen der Programmcode für die Datenbindung zur Modellschicht mit in den XAML-Code übernommen!

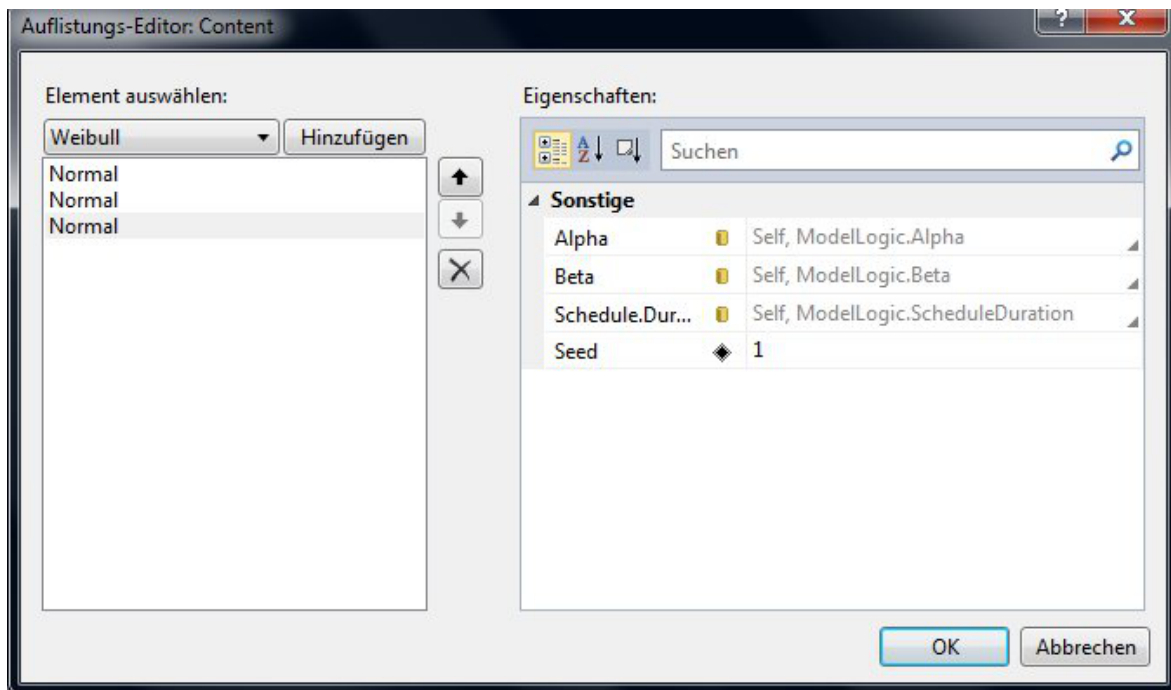


Abbildung 4.5.: Übernahme des Codes zur Datenbindung nach XAML

Eine einfache Maßnahme für die Lösung dieses Problems ist das Hinzufügen einer Abfrage, ob sich ein Dependency-Objekt im Entwurfsmodus befindet.

```
1 public Normal() : base()  
2 {  
3     if (!DesignerProperties.GetIsInDesignMode(this))  
4         this.ModelLogic = new NormalML();  
5 }
```

Listing 4.14: Abfrage bezüglich des Modus für ein Dependency-Objekt. Quellcodeauszug aus der Datei *Normal.cs* aus der Ansichtsschicht

4.3.3 Attribute für die Bereitstellung von Metadaten zur Entwurfszeit

Viele Informationen, die zur Entwurfszeit mit Dependency-Objects verbunden sind, können über Attribute beschrieben werden. Einige Attribute, die in diesem Zusammenhang im Projekt *SimNetUI* verwendet werden, werden in den folgenden Abschnitten aufgeführt.

4.3.3.1 DisplayName

Durch die Verwendung des Attributs *DisplayName* kann ein .Net Property mit einer eigenen Bezeichnung für den WPF-Designer versehen werden. Wird dieses Attribut nicht verwendet, wird stattdessen der ursprüngliche Name des Properties im Property-Grid angezeigt.

4.3.3.2 CategoryAttribute

Das *CategoryAttribute* ist ein wichtiges Attribut, dessen Aufgabe es ist, Properties einer bestimmten Kategorie zuzuweisen. Hierdurch werden in der SimNetUI-Bibliothek Properties von *Dependency-Objects* der Kategorie *Simulation* hinzugefügt.

```
1 [CategoryAttribute("Simulation")]
2 [DisplayName("Processing Capacity")]
3 public uint Capacity
4 {
5     get { return (uint) GetValue(CapacityProperty); }
6     set { SetValue(CapacityProperty, value); }
7 }
```

Listing 4.15: Beispiel für die Verwendung der Attribute *DisplayName* und *CategoryAttribute*. Quellcodeauszug aus der Datei *Wait.XAML.cs* aus der Ansichtsschicht

4.3.3.3 DesignTimeVisible

Um ein Ausblenden von Komponenten im Designer zur Entwurfszeit zu erzwingen, bietet sich die Verwendung des Attributs *DesignTimeVisible* an. Ebenso kann auf diesem Wege explizit angegeben werden, wenn eine Komponente, d.h. eine Klasse oder ein Property, zur Entwurfszeit eingeblendet werden soll.

```
1 [DesignTimeVisible(false)]
2 public class ActivityDelayBase : ActivityQueueBase { }
```

Listing 4.16: Beispiel für die Verwendung der Attribute *DesignTimeVisible*

4.3.4 Attributtabelle zur Bereitstellung von Metadaten für die Entwurfszeit

Für Werkzeuge, die als Erweiterung in den WPF-Designer *Cider* integriert werden sollen, müssen zunächst Metadaten zur Beschreibung dieser Erweiterungen in einer Attributtabelle abgelegt werden. Im Projekt *SimNetUI.VisualStudio.Design* existiert hierfür eine Klasse *RegisterMetadata*. Diese Klasse implementiert das *IProvideAttributeTable* Interface und stellt über das Property *AttributeTable* eine solche Attributtabelle zur Verfügung. Somit eignet sich die Datei *RegisterMeta.cs* als Einstiegspunkt für das Studium der WPF-Designer-Erweiterungen der SimNetUI-Bibliothek.

Visual Studio instanziiert die Objekte, die für den WPF-Designer benötigt werden selbstständig. Daher wird über die Attributtabelle lediglich der Typ der entsprechenden Klassen bereitgestellt. Häufig wird hierfür das Attribut *FeatureAttribut* verwendet.

```
1 internal class RegisterMetadata : IProvideAttributeTable
2 {
3     public AttributeTable AttributeTable
4     {
5         get
6         {
7             builder = new AttributeTableBuilder();
8             builder.AddCustomAttributes(typeof(Generator), new FeatureAttribute(typeof(
9                 GeneratorInitializer)));
10            return builder.CreateTable();
11        }
12    }
```

Listing 4.17: Minimales Beispiel für die Bereitstellung von Metadaten in einer Attributtabelle unter Verwendung des Attributs *FeatureAttribut*

4.3.5 Bearbeitungsmodell für den Zugriff auf Benutzeroberflächenobjekte

Vielerorts im Programmcode des *SimNetUI.VisualStudio.Design*-Projektes erfolgt der Zugriff auf Objekte aus der Ansichtsschicht über eine Abstraktion in Form eines Bearbeitungsmodells. Folglich werden Eigenschaften von Objekten im Designer selten über Referenzen auf instanziierte Objekte aus der Ansichtsschicht direkt verändert. Hier hilft es sich in Erinnerung zu rufen, dass der WPF-Designer selbstständig Instanzen der Benutzersteuerelemente, die über die Ansichtsschicht bereitgestellt werden, instanziiert. Somit muss zwischen den vom WPF-Designer instanziierten Objekten und dem XAML-Quellcode, der hierfür ausgewertet wird, unterschieden werden. In der Regel ist es gewollt, dass sowohl die Eigenschaften der Instanzen im WPF-Designer aktualisiert

werden, als auch der zugehörige XAML-Quellcode. Durch ein abstraktes Bearbeitungsmodell wird all dies ermöglicht.

In diesem Zusammenhang ist eine Betrachtung des *Microsoft.Windows.Design.Model*-Namespace von Nutzen¹⁰. Jede Instanz der *ModelItem*-Klasse stellt ein Element im Bearbeitungsmodus dar. Hierbei kann es sich um jede Art von .Net Objekten handeln. Am häufigsten wird über die *ModelItem*-Klasse auf Klassen aus der SimNetUI-Bibliothek zugegriffen, die von der Klasse *DependencyObject* erben. Neue Objekte können über eine sogenannte *ModelFactory* erstellt werden. Die Verwendung dieses Mechanismus soll an einem Beispiel demonstriert werden.

```
1 ModelItem distribution = ModelFactory.CreateItem(item.Context, typeof(UniformDouble));
2 distribution.Properties[PropertyNames.UniformDouble.SeedProperty].SetValue(1);
3 distribution.Properties[PropertyNames.UniformDouble.MinProperty].SetValue(0.0);
4 distribution.Properties[PropertyNames.UniformDouble.MaxProperty].SetValue(5.0);
```

Listing 4.18: Beispiel zur Demonstration der Verwendung des Bearbeitungsmodells zum Zugriff auf Objekte, die für den Designer instanziiert werden.

Der Zugriff auf Properties eines Objektes über ein *ModelItem* erfolgt, wie im Quellcode 4.18 ersichtlich, über einen Bezeichner. Bei diesem Bezeichner kann es sich um eine Zeichenkette handeln oder aber eine Instanz der Klasse *PropertyIdentifier*. Dieses Zugriffsmodell kann allerdings auch eine häufige Fehlerquelle sein. Denn Tippfehler werden erst zur Laufzeit entdeckt, da an dieser Stelle keine Syntaxfehler vorliegen und somit keine Chance besteht, dass der Compiler diese Fehler selbstständig findet. Eine Möglichkeit diese Fehlerquelle wenigstens zu minimieren, ist die Bereitstellung von vordefinierten Bezeichnern, sodass für jede Eigenschaft eines Objektes nur eine Instanz eines Bezeichners an zentraler Stelle im Code des Erweiterungsprojektes instanziiert werden muss. Dies hat im speziellen auch den Vorteil, dass bei Namensänderungen in der Ansichtsschicht der Aufwand geringer ist, diese Umstellungen auch im Quellcode des Erweiterungsprojektes durchzuführen. Im Gültigkeitsbereich der Klasse *PropertyNames* werden daher Instanzen auf Basis der Klasse *PropertyIdentifier* zur Verfügung gestellt.

```
1 internal class PropertyNames {
2     public class UniformDouble : ProbabilityDistributionBase {
3         new public static readonly TypeIdentifier TypeId =
4             new TypeIdentifier(
5                 "SimNetUI.Activities.PropertyObjects.Distributions.UniformDouble");
6         public static readonly PropertyIdentifier MinProperty =
7             new PropertyIdentifier(TypeId, "Min");
8         public static readonly PropertyIdentifier MaxProperty =
```

¹⁰ Vgl. [MSD11a]

```
9      new PropertyIdentifier(TypeId, "Max");
10  }
11  public class ProbabilityDistributionBase : DistributionBase {
12      new public static readonly TypeIdentifier TypeId =
13          new TypeIdentifier(
14              "SimNetUI.Activities.PropertyObjects.Distributions.ProbabilityDistributionBase
15              ");
16      public static readonly PropertyIdentifier SeedProperty =
17          new PropertyIdentifier(TypeId, "Seed");
18  }
19  public class DistributionBase {
20      public static readonly TypeIdentifier TypeId =
21          new TypeIdentifier(
22              "SimNetUI.Activities.PropertyObjects.Distributions.DistributionBase");
23  }
```

Listing 4.19: Bereitstellung

von Bezeichnern für den Zugriff auf Objekte über das Bearbeitungsmodell des WPF-Designer-Extensibility-Frameworks. Quellcodeauszug aus der Datei *PropertyNames.cs*

4.3.6 Erweiterungskomponenten

4.3.6.1 Adorner

Zur Entwurfszeit können Benutzersteuerelemente im Visual Studio WPF-Designer zusätzliche Bedienelemente erhalten. Diese werden in Form einer Überlagerung im Designer dargestellt. Die Bedienleiste, die im Kapitel 3.8 vorgestellt worden ist, ist ein Beispiel für deren Verwendung. In der SimNetUI-Bibliothek sind des Weiteren Adorner auch als grafische Hilfsmittel für die Erstellung von Verbindungen zwischen Aktivitäten implementiert worden. So existiert beispielsweise für jede Verbindungstelle einer Aktivität eine farbig markierte Fläche in Form eines eigenen Adorners. Weiterhin werden ebenso vom Entwickler ausgewählte Linien im Linienmodus mittels Adorner farblich hervorgehoben.

Eine zentrale Bedeutung für die Bereitstellung von Adornern kommt der Klasse *SimulationContainerAdornerProvider* zu. Diese Klasse erweitert die Klasse *AdornerProvider* und überschreibt die beiden wichtigen Methoden *Activate* sowie *Deactivate*, wobei eine von diesen vom WPF-Designer immer dann aufgerufen wird, wenn ein Benutzersteuerelement vom Typ *SimulationContainer* aus der SimNetUI-Bibliothek selektiert (*Activate*) oder die Auswahl aufgehoben (*Deactivate*) worden ist.

1 `[UsesItemPolicy(typeof(SelectionPolicy))]`

```
2 internal class SimulationContainerAdornerProvider : AdornerProvider {
3     protected override void Activate(ModelItem item) {
4         // ...
5         var ToolbarPanel = new AdornerPanel();
6         ToolbarPanel.Children.Add(optionsAdorner);
7         // Die Toolbar der Liste der Adorner für das Benutzersteuerelement
8         // SimulationContainer hinzufügen.
9         Adorners.Add(ToolbarPanel);
10        // ...
11        base.Activate(item);
12    }
13    protected override void Deactivate() {
14        // ...
15        base.Deactivate();
16    }
17 }
```

Listing 4.20: Prototypen der Methoden *Activate* und *Deactivate*. Quellcodeauszug aus der Datei *SimulationContainerAdornerProvider.cs*

Wie im vorangestellten Quelltext 4.20 zu erkennen ist, wird der Methode *Activate* noch zusätzlich eine Instanz vom Typ *ModelItem* übergeben, womit in diesem Fall eine Abstraktion für den Zugriff auf den Simulationscontainer bereitgestellt wird. Dies erlaubt unter anderem auch eine komplette Analyse des Modells, wodurch die Grundlage für weitere Adorner geschaffen ist, um Verbindungsstellen sowie Verbindungen durch farbige Markierungen hervorzuheben.

Die Oberfläche eines Adorners kann sich aus beliebigen WPF-Controls zusammensetzen, welche aber innerhalb eines *AdornerPanels* angeordnet sein müssen. Die Oberfläche der Bedieneiste ist beispielsweise vollständig in einer separaten XAML-Datei als *UserControl* implementiert, welches innerhalb eines *AdornerPanels* platziert worden ist. Die Positionierung eines *AdornerPanels* wird am Benutzersteuerelement, welches es dekoriert, ausgerichtet. Dies kann eine beliebige Position innerhalb des Bereiches welcher vom jeweiligen Benutzersteuerelement aufgespannt wird oder außerhalb entlang den Rändern des UserControls sein.

4.3.6.2 Editoren

Die allermeisten Eigenschaften der Benutzersteuerelemente werden im Property-Grid des Visual Studio WPF-Designers bearbeitet. Dieses Eigenschaftsfenster kann durch eigene Bedienelemente angereichert werden. Für einzelne Properties von Benutzersteuerelementen kann dies mittels der Klasse *PropertyValueEditor* bewerkstelligt werden. Da sich aber alle Properties, die für die Benutzersteuerelemente der SimNetUI-Bibliothek angepasst werden sollen, in eine eigene Kategorie zusammenfassen lassen, ist es aber

sinnvoller, für jedes Control einen eigenen *CategoryEditor* zu entwickeln. Die Klasse *CategoryEditor* besitzt zwei Properties. Durch das Property *TargetCategory* wird die Kategorie festgelegt, welche mit dem *CategoryEditor* assoziiert werden soll. Über das Property *EditorTemplate* wird dem WPF-Designer *Cider* ein *DataTemplate* zur Verfügung gestellt. Innerhalb eines solchen *DataTemplates* werden Steuerelemente angeordnet, welche durch Data-Binding an Properties der Benutzersteuerelemente aus der SimNetUI-Bibliothek gebunden sind. Alle *DataTemplates* sind im Erweiterungsprojekt zur SimNetUI-Bibliothek in einem *ResourceDictionary* mit XAML implementiert worden¹¹. Die Klasse *GeneralCategoryEditor* im Erweiterungsprojekt erbt von der Framework-Klasse *CategoryEditor* und wird verwendet, um ableitenden Klassen eine Bereitstellung eigener Templates auf einfache Art und Weise für die Kategorie *Simulation* zu erlauben.

```
1 internal class GeneratorCategoryEditor : GeneralCategoryEditor {
2     public GeneratorCategoryEditor() : base("GeneratorCategorieEditorTemplate") { }
3 }
4
5 internal class ExitBaseCategoryEditor : GeneralCategoryEditor {
6     public ExitBaseCategoryEditor() : base("ExitCategoryEditorTemplate") { }
7 }
8
9 // ... weitere Proxyklassen
```

Listing 4.21: Bereitstellung eigener Templates über Proxyklassen. Quellcodeauszug aus der Datei *CategoryEditors.cs*

Für Kategorieeditoren muss innerhalb der Attributtabelle dem jeweiligen Benutzersteuerelement aus der SimNetUI-Bibliothek ein Attribut hinzugefügt werden, um auf die Klasse zu verweisen, welche das passenden *DataTemplate* bereitstellt.

```
1 AddCustomActivityBaseAttributes(typeof(Generator),
2     new FeatureAttribute(typeof(GeneratorInitializer)),
3     new EditorAttribute(typeof(GeneratorCategoryEditor),
4         typeof(GeneratorCategoryEditor))
5 );
```

Listing 4.22: Hinzufügen eines *EditorAttributes* zu einem Benutzersteuerelement aus der SimNetUI-Bibliothek. Quellcodeauszug aus der Datei *RegisterMetadata.cs*

Im Gegensatz zu anderen Erweiterungskomponenten, die über das abstraktes Bearbeitungsmodell, welches im Kapitel 4.3.5 vorgestellt worden ist, Zugriff auf Properties von Benutzersteuerelementen aus der SimNetUI-Bibliothek erhalten, verwenden Kategorieeditoren ein eigenes Zugriffsmodell. Zum besseren Verständnis bietet es sich daher

¹¹Der XAML-Code für alle Kategorieeditoren ist in der Datei *EditorResources.XAML* zu finden.

an, einen Blick in die Dokumentation aus der MSDN-Bibliothek zu werfen und den Framework-Namespace *Microsoft.Windows.Design.PropertyEditing* eingehend zu studieren.

4.3.6.3 Initializer

Eigenschaften von *DependencyObjects* können zur Entwurfszeit mit Standardwerten initialisiert werden. Bei *DependencyProperties* werden in der Regel die Standardwerte verwendet, die bei der Registrierung als Metadaten mit angegeben worden sind. Wenn aber eine flexiblere Lösung benötigt wird, bietet sich die Verwendung von Initializern an. Ein Initializer ist als eine eigene Klasse zu implementieren, die von der Framework-Klasse *DefaultInitializer* erbt. Die Methode *InitializeDefaults* muss überschrieben werden. Wiederum wird das aus Kapitel 4.3.5 bekannte Bearbeitungsmodell verwendet.

```
1 public override void InitializeDefaults(ModelItem item)
2 {
3     if (item != null)
4     {
5         base.InitializeDefaults(item);
6         ModelItem distribution = ModelFactory.CreateItem(item.Context, typeof(
7             UniformDouble));
8         distribution.Properties[PropertyNames.UniformDouble.MaxProperty].SetValue(5.0);
9         item.Properties[PropertyNames.ActivityDelayBase.DistributionProperty].SetValue(
10             distribution);
11     }
12 }
```

Listing 4.23: Beispiel für die Verwendung eines Initializers. Quellcodeauszug aus der Datei *ActivityDelayBaseInitializer.cs*

Über einen Eintrag in der Attributtabelle, wie aus dem Quelltext 4.22 entnommen werden kann, wird die Verwendung des korrekten Initializers bestimmt.

4.3.6.4 Tasks

Wenn im Simulationsmodell eine Aktivität gelöscht wird, so müssen alle eingehenden Verbindungen der zu löschenden Aktivität ebenso entfernt werden. Die Schwierigkeit, die sich hierbei ergibt, ist, dass nur ausgehende Verbindungen der Hierarchie einer Aktivität untergeordnet sind. Das bedeutet, dass der XAML-Quellcode für eingehende Verbindungen nicht gelöscht wird, da dieser anderen Aktivitäten zugeordnet ist.

Tasks sind eine Möglichkeit, um auf Benutzerinteraktion im WPF-Editor einzuwirken. So lassen sich unter anderem WPF-Standardkommandos über Kommandobindungen an selbst geschriebenen Programmcode koppeln. Im beschriebenen Problemfall ist dies für das Löschkommando von Benutzersteuerelementen eine gute Lösung. Die Klasse *ActivityBaseTaskProvider* nimmt sich im Erweiterungsprojekt der SimNetUI-Bibliothek dem hier beschriebenen Problem an, wodurch beim löschen einer Aktivität auch ausgehende Verbindungen anderer Aktivitäten gelöscht werden, sofern diese mit der zu löschenden Aktivität verbunden sind.

4.3.6.5 DesignModeValueProviders

Es mag Eigenschaften von Benutzersteuerelementen geben, die zur Entwurfszeit anders interpretiert werden sollen als zur Laufzeit. Beispielsweise ist es sinnvoll, Aktivitäten die zur Laufzeit ausgeblendet werden sollen, zur Entwurfszeit im Editor dennoch anzuzeigen. Auf diese Weise wird eine übersichtliche Bearbeitung des Simulationsmodells zur Entwurfszeit gewährleistet. Für diesen Anwendungsfall eignen sich *DesignModeValueProvider*. Die Klasse *ActivityDesignModeValueProvider.cs* im Designer-Erweiterungsprojekt der SimNetUI-Bibliothek sorgt dafür, dass Aktivitäten zur Entwurfszeit eingeblendet werden, auch wenn deren Property *Visibility* den Wert *Hidden* besitzt.

neue Anrufer ein. Betritt ein neuer Anrufer die Simulation, wählt dieser eine leere Telefonzelle aus oder stellt sich bei der Telefonzelle mit der kürzesten Warteschlange an. Gibt es mehr als eine Telefonzelle mit einer kürzesten Warteschlange, so stellt sich der Anrufer zufällig bei einer dieser Telefonzellen an. Ein Anrufer telefoniert gemäß einer Gleichverteilung(*UniformDouble*) zwischen 5,5 bis 30 Minuten. Die Simulation ist beendet, wenn kein Anrufer mehr telefoniert und neue Anrufer nicht mehr eintreffen.

Dieses Projekt eignet sich hervorragend, um sich mit der SimNetUI-Bibliothek vertraut zu machen. Eintreffende Anrufer betreten die Simulation über eine Generator-Aktivität. Telefonzellen werden durch Wait-Aktivitäten repräsentiert. Über eine Exit-Aktivität verlassen Anrufer die Simulation. Die Generator- und die Exit-Aktivität sowie Verbindungslinien zwischen den Aktivitäten werden zur Laufzeit der Simulationsapplikation ausgeblendet. Zu jeder Wait-Aktivität existiert ein Begleitobjekt vom Typ *QueueCompanion* welches mit der zugehörigen Warteschlange über eine Datenbindung verknüpft ist.



Abbildung A.1.: Screenshot der Beispielanwendung Example.Phonecell während eines Simulationslaufes

A.9 Example.MarketPlace

Das Projekt *Example.MarketPlace* ist ein weiteres Beispiel, welches die Verwendung der SimNetUI-Bibliothek demonstriert. Simuliert wird ein Geschäft welches von 8 Uhr Früh bis 20 Uhr Abends geöffnet hat. Während dieser Zeit treffen neue Kunden gemäß einer Exponentialverteilung mit $\text{Alpha} = 2 \text{ Minuten}$ ein. Für Geschäftskunden stehen insgesamt 15 Einkaufswagen zur Verfügung. Jeder Kunde der das Geschäft betritt, nimmt einen solchen Einkaufswagen mit. Sollten nicht genug Einkaufswagen zur Verfügung stehen, müssen eintreffende Kunden warten, bis andere Kunden ihren Einkaufswagen wieder abgegeben haben. Die Einkaufsdauer von Kunden wird über eine Dreiecksverteilung mit einem Minimalwert von 2 Minuten, einem Mittelwert von 12 Minuten und einen Maximalwert von 25 Minuten bestimmt. Es stehen 3 Kassen zur Verfügung. Wenn alle 3 Kassen belegt sind, wählt sich der Kunde die Kasse mit der kürzesten Warteschlange aus. Das Kassieren wird durch eine Gleichverteilung von 1 bis 3 Minuten beschrieben. Kunden die abkassiert worden sind, geben ihren Einkaufswagen ab und verlassen das Geschäft.

Dieses Beispiel ist besonders Interessant, da es die Verwendung von Ressourcen demonstriert. In dieser Simulation stellen Entitäten die Geschäftskunden dar. Neue Entitäten betreten die Simulation über eine *Generator*-Aktivität. Eine Ressource, die die Bezeichnung *shopping cart* trägt, wird an Entitäten gebunden, wenn diese nach Eintritt in das Geschäft eine Aktivität vom Typ *Assign Ressource* durchlaufen. Um das Einkaufen von Kunden zu simulieren, wird eine Aktivität vom Typ *Wait* verwendet. Damit mehrere Kunden gleichzeitig einkaufen können, wird der Wert *Processing Capacity* bei dieser Aktivität auf einen nicht erreichbaren Wert erhöht. Die drei Kassen werden durch 3 *Wait*-Aktivitäten repräsentiert. Nach dem Kassieren durchlaufen alle Entitäten eine Aktivität vom Typ *Release Ressource*, um die Ressource *shopping cart* wieder freizugeben. Letztendlich verlassen Kunden das Geschäft über eine *Exit*-Aktivität. Die Verbindungslinien zwischen den Aktivitäten werden zur Laufzeit ausgeblendet.

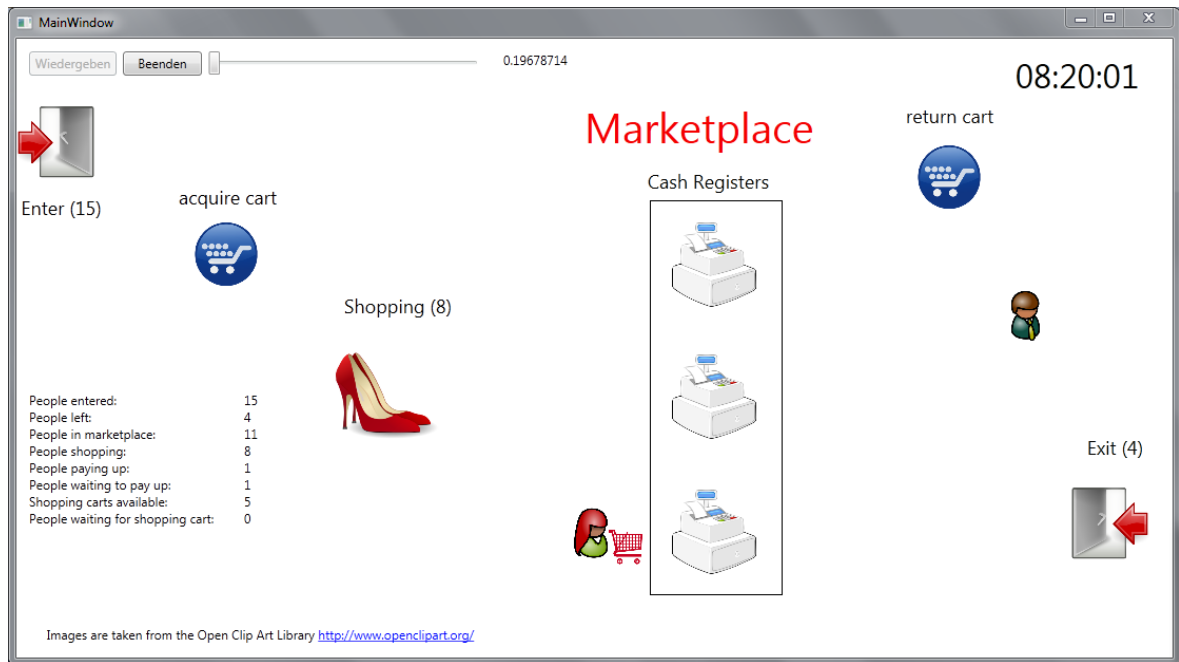


Abbildung A.2.: Screenshot der Beispielanwendung *Example.MarketPlace* während eines Simulationslaufes

A.10 *Example.Text*

Das *Example.Text* Projekt ist ein Spielwiese, um die Funktionsweise der SimNetUI-Bibliothek zu testen.

D Installationsleitung für die SimNetUI-Bibliothek

Applikationsentwickler, die auf Grundlage der SimNetUI-Bibliothek Computersimulationen entwickeln, soll ein Einstieg in die Materie nach aller Möglichkeit erleichtert werden. Eine Nahtlose Integration in die Visual Studio 2010 IDE ist dabei Voraussetzung. Auf der dieser Diplomarbeit beiliegenden CD befinden sich im Ordner *Software\Anwendung* die für die Installation notwendigen Dateien.

1. Die DLLs, die sich im Ordner *Software\Anwendung\Bibliotheken* auf der beiliegenden CD befinden, sind in das Verzeichnis *C:\Program Files\SimNetUI* zu kopieren. Wenn das Verzeichnis noch nicht existiert, ist dieses zu erstellen.
2. Eine Projektvorlage wird über die Datei *Software\Anwendung\SimNetUI.ProjectTemplate.Setup.vsix* als Erweiterung für Visual Studio installiert. Eine Deinstallation ist über den *Erweiterungs-Manager* in Visual Studio jederzeit möglich.
3. Bei der ersten Verwendung der Projektvorlage sind die Benutzersteuerelemente aus der SimNetUI-Bibliothek der Toolbox in Visual Studio noch hinzuzufügen. Die folgende Anleitung erklärt, wie dies in der deutschen Version von Visual Studio 2010 zu bewerkstelligen ist.
 - a) Mauszeiger über die Toolbox bewegen und mit rechter Maustaste das Kontextmenü öffnen. Jetzt den Eintrag *Registerkarte hinzufügen* auswählen. Die neue Registerkarte sollte idealerweise mit der Bezeichnung *SimNetUI* versehen werden.
 - b) Die Registerkarte *SimNetUI* aufklappen. Den Mauszeiger über den leeren Bereich innerhalb der SimNetUI-Gruppe bewegen und mit rechter Maustaste das Kontextmenü öffnen. Im Kontextmenü auf den Eintrag *Elemente auswählen ...* klicken. Im Dialogfenster *Toolboxelemente auswählen* die Registerkarte *WPF-Komponenten* auswählen. Durch betätigen der Schaltfläche *Durchsuchen* wird ein Dialog zur Auswahl von Dateien geöffnet. In diesem Dialog die Datei *C:\Program Files\SimNetUI\SimNetUI.DLL* auswählen. Abschließend sollte noch kontrolliert werden, ob alle Steuerelemente der SimNetUI-Bibliothek tatsächlich ausgewählt worden sind. Das Dialogfenster *Toolboxelemente auswählen* kann nun durch betätigen der Schaltfläche OK geschlossen werden.

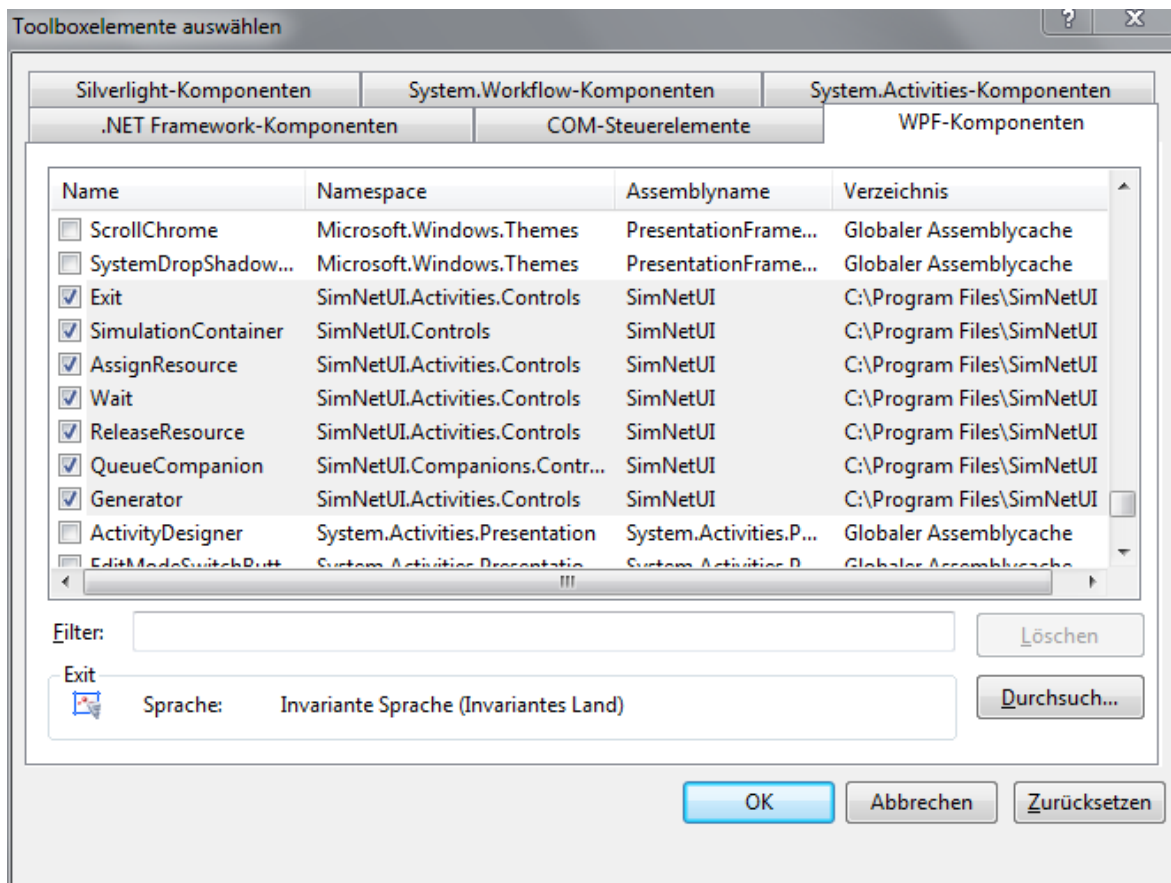


Abbildung D.1.: Screenshot des *Toolboxelemente auswählen* Dialogs.