Architektur eines Simulationsbaukastens basierend auf dem Komponentenmodell JavaBeans

Herbert Praehofer[1], Johannes Sametinger[2], Alois Stritzinger[2]

[1]*Institut für Systemwissenschaften - Systemtheorie und Informationstechnik*
[2]*Institut für Wirtschaftsinformatik - Software Engineering*
Johannes Kepler Universität, A-4040 Linz / Austria

Kurzfassung

Modulare, hierarchische Modellierung und das Komponentenmodell JavaBeans bilden die Grundlage für die Entwicklung eines Kompontenrahmenwerkes zur Realisierung von Simulationsanwendungen. Systemtheoretische Formalismen stellen die formale Basis für die modulare, hierarchische Modellierung und Simulation dar; das Komponentenmodell JavaBeans wird als Implementierungplattform herangezogen. Das Ergebnis ist ein flexibles, mächtiges Baukastensystem, bestehend aus einer Reihe von Schnittstellenspezifikationen und einem Satz flexibler Simulationsbausteine zum interaktiven Aufbau neuer Simulationssysteme.

In der Folge präsentieren wir eine kurze Einführung in systemtheoretischen Modellierungskonzepte und in die Konzepte von JavaBeans. Anschließend stellen wir die Grundkonzepte und die Gesamtarchitektur unseres Rahmenwerks vor. Die Art und Weise, wie Simulationskomponenten über Schnittstellen hierarchisch gekoppelt werden, zeigen wir anhand von Komponenten zur Simulation von Systemen mit Stückgutcharakter. Abschließend fassen wir die gewonnenen Erkenntnisse zusammen und geben einen Ausblick auf weitere Arbeiten.

Keywords: Simulation, Komponenten, JavaBeans, komponentenbasierte Softwareentwicklung

Inhalt

1 Einleitung

Die Entwicklung von Softwaresystemen aus vorgefertigten Komponenten bildet den Grundgedanken der komponentenbasierten Softwareentwicklung [Sam97, Str97]. Komponentenrahmenwerke sind maßgeschneiderte Architekturen für bestimmte Anwendungsgebiete mit Kommunikationsmechanismen zur zweckmäßigen Kopplung der darauf abgestimmten Komponenten [Szy98]. Komponentenrahmenwerke (*component frameworks*) haben Ähnlichkeit mit Anwendungsrahmenwerken (*application frameworks*). Sie definieren allgemeine Strukturen und implementieren gemeinsames Verhalten. Der wichtigste Teil ist aber eine Menge von Schnittstellenbeschreibungen, die von anwendungsspezifischen Komponenten implementiert werden müssen. Dadurch ist es möglich, derartige Komponenten an passenden Stellen in das Rahmenwerk „hineinzustecken“, bzw. Komponenten untereinander zu koppeln.

Der systemtheoretische Formalismus zur Modellierung diskreter, ereignisorientierter, hierarchisch-strukturierter, modularer Systeme DEVS [Zei84, Zei90, Zei99] wird als Modellierungsansatz verwendet. Mittels Erweiterungen läßt sich der Ansatz auch auf kontinuierliche bzw. kombiniert diskret-kontinuierliche Syteme anwenden [Pra91]. Zur Implementierung der Komponenten setzen wir das Komponentenmodell JavaBeans [Sun97] ein. Die wichtigste Zielsetzung besteht in der Entwicklung einer Menge von Komponenten zum Bau von Simulationsanwendungen. Die zu erwartenden Ergebnisse dieses Projektes sind in zweierlei Hinsicht interessant: Für das Software Engineering wird anhand einer praxisnahen Aufgabenstellung die komponentenbasierte Softwareentwicklung im allgemeinen und die JavaBeans-Technologie im besonderen studiert. Für die Simulation wird mittels einer neuen vielversprechenden Software-Technologie untersucht, inwieweit dadurch Simulationsanwendungen flexibler gestaltet werden können. Zu diesem Zweck wurde ein Satz von elementaren Simulationskomponenten sowie dazupassende Visualisierungs-, Animations- und Auswertungsbausteine, die interaktiv mit einem Werkzeug zusammengesetzt werden können, entwickelt. Wir gehen von folgenden Annahmen aus:

1. Die Modelierung und Implementierung von Simulationsanwendungen eignet sich für komponentenbasierte Softwareentwicklung.

2. Die modulare, hierarchische Systemmodellierung fördert die Entwicklung wiederverwendbarer Komponentenarchitekturen.

In Abschnitt 2 wird die Vision einer komponentenbasierten Simulationsmethodik vorgestellt. Formale und technische Grundlagen, d.h. der systemtheoretisch Formalismus DEVS sowie das Komponentenmodell JavaBeans werden in Abschnitt 3 beschrieben. In Abschnitt 4 erläutern wir die Architektur des entwickelten Komponentenrahmenwerkes. Die Kopplung von Komponenten auf Basis dieses Rahmenwerkes beschreiben wir in Abschnitt 5. In Abschnitt 6 folgt beispielhaft die Beschreibung einer auf das Komponentenrahmenwerk basierenden Basisbibliothek zur Stückgutverarbeitung. Eine Zusammenfassung und einen Ausblick geben wir schließlich in Abschnitt 7.

2 Vision einer komponentenbasierten Simulations-entwicklungsmethodik

Für komponentenbasierte Simulation werden Komponentenbibliotheken für unterschiedliche Anwendungsgebiete, auf verschiedenen Abstraktionsniveaus und für verschiedene Benutzergruppen benötigt. Das Hauptziel ist dabei die Wiederverwendung von möglichst vielen vorgerfertigten, flexiblen Komponenten. Simulationssysteme sollen möglichst einfach durch Auswahl von geeigneten Komponenten, durch zweckmäßige Anordnung, durch Einstellung diverser Parameter sowie, falls notwendig, durch Erweiterung und Ergänzung von Komponenten realisiert werden können. Neben den entsprechenden Komponenten bedarf es noch zwei weiterer nicht minder wichtiger "Zutaten": Erstens ist es Aufgabe des Komponentenrahmenwerkes, alle wichtigen zentralen Strukturen zu definieren, das allgemein benötigte Verhalten zu implementieren, sowie die von den Komponenten zu benutzenden Kommunikationsmechanismen festzuschreiben. Zweitens benötigt man für den Zusammenbau der Komponenten ein geeignetes Werkzeug. Da wir auf dem Komponentenmodell JavaBeans aufbauen, können - mit Einschränkungen - kommerziell verfügbare Werkzeuge dazu eingesetzt werden.

Damit das angestrebte Ziel einer möglichst komfortablen Arbeitsweise erreicht werden kann, müssen unterschiedliche Aufgaben von verschiedenen Personen(gruppen) durchgeführt werden:

- *Simulationstechniker*
  Der Simulationstechniker modelliert mit vorgefertigten Komponenten die einzelnen Anlagenteile, stellt diverse Komponentenparameter ein und setzt die einzelnen Komponenten interaktiv zusammen. Nach dem Zusammenbau führt der Simulationstechniker einfache Experimente aus und nimmt Änderungen am Modell oder an der Visualisierung/Auswertung vor. Kurze „turn around“-Zeiten und eine gleichbleibende Benutzeroberfläche sind für eine explorative Arbeitsweise wichtig. Simulationstechniker benötigen nur geringe Programmierkenntnisse, sodaß von ihnen nicht erwartet werden kann, daß sie komplexe Java-Klassen an neue Gegebenheiten anpassen oder neu erstellen. Lediglich die Eingabe von einfachen Scripts und Regeln zur Beschreibung von Verhalten in einem beschränkten, wohldefinierten Kontext, kann und muß vom Simulationstechniker verlangt werden. Der Simulationstechniker profitiert von der Komponententechnologie insofern, als er in der Lage versetzt wird, komfortabel und effizient die an ihn gestellten Aufgaben zu lösen.

- *Simulationsprogrammierer*
  Der Simulationsprogrammierer entwirft und programmiert die für einen bestimmten Anwendungsbereich benötigten Komponenten. Dazu kann er sich elementarer Komponenten aus Komponenten- und/oder Klassenbibliotheken bedienen. In unserem Fall handelt es sich um einen Java-Programmierer, der die Konzepte des Komponentenrahmenwerks verstehen und anwenden muß. Typischerweise muß der Simulationsprogrammierer kein hoch-versierter Software-Ingenieur sein, da viele Entwurfsentscheidungen bereits durch das Rahmenwerk vorweggenommen sind.

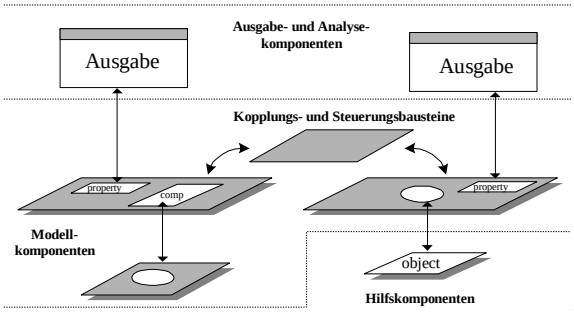
- *Simulationsexperte*
  Der Simulationsexperte realisiert die Basiskomponenten und die notwendige Infrastruktur, d.h. das Komponentenrahmenwerk. Typischerweise werden für jedes neue Anwendungsgebiet neue Schnittstellendefinitionen und neue Basiskomponenten erstellt. Das erfordert ein umfassendes Verständnis der zugrundeliegenden Modellierungs- und Simulationskonzepte sowie der JavaBeans-Technologie. Eine wichtige Herausforderung für den Entwurf dieser Teile liegt in der Forderung nach möglichst hoher Wiederverwendbarkeit bei gleichzeitiger Angemessenheit für einen konkreten Anwendungsfall.

Die Bibliothek der Basiskomponenten setzt sich wie folgt zusammen:

- Basiskomponenten zur Modellierung und Simulation diskreter Ereignissysteme
- Basiskomponenten zur Modellierung und Simulation kontinuierlicher und kombinierter Systeme
- Basiskomponenten zur Visualisierung, Datenausgabe, Animation und für statistische Auswertungen
- Zusätzliche Komponenten für häufig benötigte Dienste
- Schnittstellendefinitionen zur Festlegung der "Rollen", die Komponenten spielen können, sowie deren Schnittstellen

Abbildung 1 zeigt, wie Simulationssysteme aus Komponenten zusammengefügt und konfiguriert werden. Eine Reihe von Aktivitäten sind dazu notwendig:

- *Konfiguration*: Auswahl und Anordnung von zweckmäßigen Modellkomponenten
- *Anpassung*: Einstellung aller erforderlichen Parameter
- *Kopplung*: In-Beziehung-setzen der Komponenten, d.h. Zuordnung von zueinander passenden Eingangs- und Ausgangschnittstellen
- *Anschließen*: Zuordnen von Visualisierungs-, Ausgabe-, Animations- und Auswertungskomponenten



EINBETTEN

Abb. 1: SEQARABISCHKonfiguration eines Simulationssystem aus Komponenten

3 Formale und technische Grundlagen

3.1 Modulare, hierarchische Systemmodellierung

Modulare, hierarchische Systemmodellierung [Zei84, Zei90, Pic92] ist ein Ansatz zur Modellierung von komplexen dynamischen Systemen, in dem modulare Komponenten mit exakt definierten Schnittstellen in hierarchischer Form miteinander gekoppelt werden. Man unterscheidet zwischen atomaren und gekoppelten Modellen. Atomare Modelle werden spezifiziert durch eine Menge von Zuständen und Überführungsfunktionen [Cel91, Zei99]; gekoppelte Modelle werden durch die involvierten Komponenten und ihre Kopplungsstruktur spezifiziert. Die Modularität gestattet den Aufbau von Systemen aus wiederverwendbaren Komponenten, die mittels ihrer exakt definierten Ein- und Ausgangsschnittstellen miteinander kommunizieren können. Modulare Systemmodellierung bedeutet also schnittstellenkonformes Koppeln von Komponenten. Komponentenbasierte Softwareentwicklung zielt in die gleiche Richtung: Softwarekomponenten definieren Eingangsschnittstellen in Form von ausführbaren Methoden und Ausgangsschnittstellen in Form von auslösenden Ereignissen.

Bei der modularen hierarchischen Systemmodellierung werden atomare Simulationsmodelle durch eine Zustandsraumdarstellung beschrieben [Cel91]. Für die verschiedenen Arten von Systemen werden unterschiedliche Formalismen verwendet: Klassische Differentialgleichungssysteme (*Differential Equation Specified Systems, DESS*) zur Beschreibung kontinuierlicher Systeme; der DEVS-Formalismus zur Spezifikation diskreter ereignisbasierter Systeme [Zei90]; und der DEV&DESS-Formalismus zur Beschreibung kombiniert diskret-kontinuierlicher Systeme [Zei99, Pra91].

Die modulare hierarchische Systemmodellierung stellt einen zweckmäßigen Ansatz zur Erhöhung der Wiederverwendbarkeit von Modellen dar. Dabei kommt der Gestaltung der Modellschnittstellen und der darauf aufbauenden Modellkopplung entscheidende Bedeutung zu. Im *Abschnitt 6* beschreiben wir Schnittstellendefinition und Kopplungsstrukturen für Simulationsanwendungen zur Verarbeitung von Stückgut.

3.2 Das Komponentenmodell JavaBeans

JavaBeans ist das Komponentenmodell von Java [Sun97]. Eine JavaBeans-Komponente, genannt ein *Bean*, ist eine wiederverwendbare Softwarekomponente, die interaktiv modifiziert und mit anderen Komponenten kombiniert werden kann. Werkzeuge zur Montage von Beans können von unterschiedlicher Komplexität sein. Mögliche Kandidaten sind etwa Werkzeuge zur Erstellung graphischer Benutzeroberflächen oder komponentenbasierte, visuelle Entwicklungswerkzeuge. Beans können mit beliebigen JavaBeans-basierten Werkzeugen bearbeitet werden. Damit werden die Entwicklung von Komponenten und von Werkzeugen entkoppelt. Beans werden durch Java-Klassen, die einigen wenigen Konventionen genügen müssen, definiert und implementiert. Zum Beispiel müssen Bean-Klassen einen "public null-constructor" besitzen, damit sie in einem beliebigen Kontext instanzierbar sind. Außerdem müssen Klassen von Beans serialisierbar sein, damit diese persistent gespeichert werden können. Die Merkmale (Methoden, Eigenschaften und Ereignisse) von Beans können durch einen Introspektionsmechanismus abgefragt werden. Die für uns in diesem Kontext wichtigsten Konzepte von JavaBeans sind Ereignisse (*events*) und Eigenschaften (*properties*). Diese werden wir im folgenden kurz beschreiben.

Ereignisse

JavaBeans definiert ein Standardmodell zur Kommunikation von Komponenten über einen Ereignismechanismus. Der Ereignismechanismus erlaubt – im Unterschied zum klassischen Methodenaufruf – die weitgehende Entkopplung von Sender- und Empfängerkomponenten. Die Sender definieren eine Ereignisschnittstelle, welche die Anmeldung von Empfängern für ein Ereignis erlaubt. Empfänger definieren Methodenschnittstellen, die bei Auftreten der Ereignisse angesprochen werden.

Eigenschaften

Eigenschaften sind benannte Attribute eines Beans, die das Erscheinungsbild und/oder das Verhalten beeinflussen können. Sie sind Teil des Zustands von Beans. Die Werte von Eigenschaften können interaktiv von Werkzeugen gelesen und gesetzt werden. Entwicklungswerkzeuge zeigen in der Regel eine Eigenschaftsliste von Beans mit den aktuell eingestellten Werten und gestatten das interaktive Verändern mittels spezieller Eigenschaftseditoren.

Gebundene Eigenschaften signalisieren deren Änderungen durch Ereignisse an ihre Umgebung. Jedes interessierte Bean muß sich entsprechend registrieren, um solche Änderungen mitgeteilt zu bekommen.

4 Architektur

Das Komponentenrahmenwerk für Simulationsanwendungen sowie die darauf aufbauenden Basiskomponenten bezeichnen wir als SimBeans, deren grundsätzliche Architektur in Abbildung 2 dargestellt ist. In Abstimmung mit den oben erwähnten Benutzergruppen existieren verschiedene Architekturschichten mit jeweils verschiedenen Subsystemen für verschiedene Aufgabenbereiche. Je höher die Schicht, desto speziellere Aufgaben werden in den jeweiligen Subsystemen realisiert.

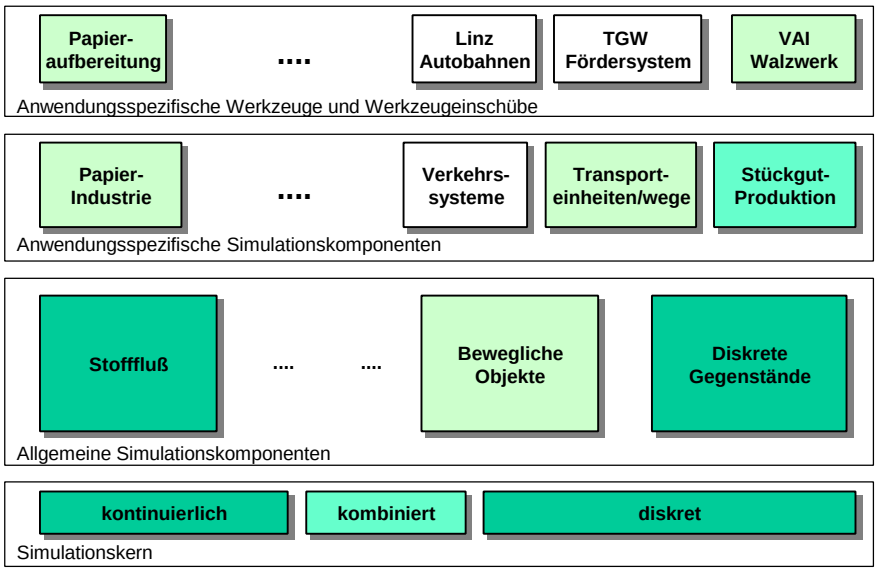


Abb. 2: Schichenarchitektur von SimBeans

Java/JavaBeans-Erweiterungen

Die unterste Schicht von SimBeans bildet die Sprache Java selbst, sowie die zentralen Klassenbibliotheken von Java und JavaBeans. Mit der Java 2 Standard Edition (jdk 1.2) hat der Funktionsumfang der Java-Bibliotheken einen für unsere Zwecke hinreichenden Umfang erreicht, lediglich der Ereignisverteilungsmechanismus wurde von uns um einen *asynchronen Ereignisverteilungsmechanismus* erweitert. Der Standardmechanismus ist für unsere Zwecke ungeeignet, da die JavaBeans-Spezifikation keine Sematik für die Reihenfolge für die Ereignisverteilung definiert. Es wird lediglich eine Standardimplementierung zur synchronen Ereignisverteilung angeboten. Damit werden die Ereignisempfänger in der Reihenfolge ihrer Registrierung unmittelbar bei der Auslösung eines Ereignisses synchron benachrichtigt. Wenn jedoch bereits der erste Ereignisempfänger den Zustand der Ereignisquelle verändert (Rückkopplung), dann kann es vorkommen, daß den später benachrichtigten Empfängern sinnlose oder falsche Ereignisse übermittelt werden. Aus diesem Grund haben wir einen asynchronen Verteilungsmechanismus entwickelt, der aus-

gelöste Ereignisse in einer globale Ereignisliste registriert. Bevor der Simulator das jeweils nächste Zeitereignis behandelt werden alle registrierten Ereignisse asynchron in der Reihenfolge ihrer Registrierung (FIFO) abgearbeitet.

Simulationskern

Der Simulationskern umfaßt die allgemeine Simulationsinfrastruktur sowie die Basisinfrastruktur für diskrete Ereignissimulation, kontinuierliche Simulation und kombinierte Simulation. Die wichtigsten Teile sind: grundlegende Mechanismen zum hierarchischen Aufbau von Simulationsmodellen, elementare Klassen zur Implementierung von Modellvariablen, Mechanismen zur Ereigniseinplanung und Ereignisbehandlung, numerische Integration, Funktionen zur Zufallszahlenerzeugung, sowie elementare Ausgabe- und Visualisierungskomponenten.

Basiskomponenten und -schnittstellen

Die Basiskomponentenschicht enthält einfache Komponenten und Schnittstellendefinitionen, welche die Grundlage für ein bestimmtes Anwendungsgebiet darstellen. So existieren etwa Basiskomponenten und Schnittstellendefinitionen zur Modellierung von diskreten Stückgut-Systemen oder von chemischen Prozessen. Typische Komponenten dieser Schicht sind z.B. für Stückgut-Systeme: Verarbeitungsstationen, Warteschlangen und Warteräume, Transporteinrichtungen, klassische Steueralgorithmen und Kopplungsschemata, Ausgabe- und Auswertungskomponenten. Die Teile dieser Schicht müssen mit besonderer Sorgfalt entworfen werden, da hier die Wiederverwendbarkeit bzw. der Einsatzbereich von Komponenten und Schnittstellen festgelegt wird.

Anwendungsspezifische Komponenten

In dieser Schicht werden die realen Komponenten des Anwendungsbereichs modelliert. Sie legen die Gestaltungsmöglichkeiten der Simulationstechniker fest. Die Implementierung der anwendungsspezifischen Komponenten stützt sich auf die Klassen und Schnittstellen der darunterliegenden Schichten.

Anwendungsspezifische Werkzeuge bzw. Werkzeugeinschübe

Die Spitze der Schichtenarchiktur bilden anwendungsspezifische Entwicklungs- und Simulationswerkzeuge, bzw. Werkzeugteile, welche die Anpassung universeller Werkzeuge auf bestimmte anwendungsspezifische Bedürfnisse gestatten (Werkzeugeinschübe bzw. *plug-ins*). Diese Schicht berücksichtigt, daß Simulationstechniker von unterschiedlichen Anwendungsbereichen verschiedene Arbeitsweisen mit individuellen Darstellungsformen und Benutzungsoberflächen bevorzugen.

Komponentenbasierte Softwaretechnologie soll nicht nur für die Erstellung von konkreten Simulationsanwendungen sondern auch für die Entwicklung von Werkzeugen selbst eingesetzt werden. Dabei geht es allerdings weniger um eine interaktive Konfiguration der Werzeuge, sondern um eine möglichst flexible, modulare, einfach adaptier- und konfigurierbare Werkzeugarchitektur.

5 Kopplung von Simulationskomponenten

Ein zentrales Entwurfsziel war die strikte Trennung von Komponentenschnittstellen und Kopplungsstrukturen. So existieren eine Reihe von generischen Schnittstellendefinitionen für unterschiedliche Simulationsanwendungen. Die Simulationskomponenten implementieren die entsprechenden Schnittstellen. Eigene Kopplungskomponenten dienen zur Kopplung der Basiskomponenten über die definierten Schnittstellen, wobei die realisierten Kopplungsstrukturen mehr oder weniger komplex sein können. Komplexe gekoppelte Modelle können hierarchisch aus einfacheren Komponenten zusammengesetzt werden. Modellstrukturen, die wiederverwendet werden sollen, können mittels sogenannter Stecker (*plugs*) eine Schnittstelle zu ihrer Umgebung festlegen und dann als eigenständige Komponenten abgelegt werden. Diese Komponenten können dann wie vollständig programmierte Komponenten in beliebige andere Modelle eingesetzt werden.

Bei der Modellierung von Stückgutsystemen unterscheiden wir vier grundlegende Modellkonzepte:

- *Ressourcen*
  Ressourcen werden von Gegenständen belegt. Sie können diese bearbeiten, lagern, reihen, etc. Ressourcen können aktiv (z.B. Bearbeitungsstationen) oder passiv (z.B. Lager) sein.
- *Gegenstände*
  Gegenstände (*items*) fließen durch das System und belegen die Ressourcen.

- *Kopplungen*
  Kopplungen stellen Bindeglieder zwischen Ressourcen dar und legen den möglichen Fluß von Gegenständen fest.

- *Steuerungen*
  Steuerungen regeln den Fluß von Gegenständen aufgrund von Vorgabewerten, Regeln sowie Zuständen von Ressourcen und Gegenständen.

Simulationssysteme sind als Menge von gekoppelten Ressourcen, die mit einer Anzahl von Gegenständen belegt werden können und die diese Gegenstände bearbeiten können, aufgebaut. Die Steuerungen bestimmen welche Gegenstände von welchen Ressourcen wann zu welchen gekoppelten Ressourcen fließen. Im allgemeinen trifft diese abstrakte Beschreibung auf alle Modelle von Stückgutverarbeitungsanlagen zu. Konkrete Systeme unterscheiden sich allerdings hinsichtlich der Ausprägungen der Ressourcen, Gegenständen, Kopplungsstrukturen und insbesondere in den Steuerungsmechanismen. Diese können vielfältige Ausprägungen haben: zentrale, komplexe Steuerungen mit hoher Flexibilität (z.B. Robotersysteme) oder dezentrale, einfache Steuerungen (z.B. Fließbandsysteme).

Bei anderen Systemen, insbesondere in der kontinuierlichen Simulation, lassen sich ähnliche Grundkonzepte identifizieren. Zum Beispiel wurde eine ähnliche Architektur für die Simulation von Stoffflüssen bei der Papiererzeugung realisiert [Pra 2000]:

- *Reservoirs*
  Reservoirs werden mit kontinuierlich fließenden Stoffen (Flüssigkeiten, Gase, Ströme, etc.) gefüllt. Sie stellen das Pendant zu Ressourcen in diskreten Systemen dar.

- *Stoffe*
  Stoffe fließen kontinuierlich durch das System und befüllen Reservoirs.

- *Kanäle*
  Kanäle oder Leitungen legen die möglichen Stoffflüsse fest. Sie entsprechen den Kopplungen in diskreten Systemen.

- *Steuerungen und Regler*
  Steuerungen und Regler regulieren den Stoffluß durch die Kanäle. Kontinuierliche Zustandsvariable signalisieren Änderungen von beliebig feiner Granularität und beeinflussen dadurch die Regler.

Im folgenden beschränken wir uns auf die Beschreibung der Basisbibliothek für Stückgutsimulationen, da die dafür gemachten Aussagen grundsätzlich auch für Bibliotheken anderer Simulationssysteme gelten.

6 Basisbibliothek zur Stückgutverarbeitung

6.1 Schnittstellendefinitionen

Entsprechend den vier Grundkonzepten für die Modellierung diskreter Systeme wurden eine Reihe von Schnittstellen und Basiskomponenten definiert:

Item

Die Schnittstellendefinition *Item* bestimmt die Minimalanforderungen an alle Gegenstände. Alle Gegenstände bekommen bei ihrer Erzeugung ein eindeutiges Identifikationsmerkmal zugewiesen und können auf Ereignisse von Ressourcen reagieren.

ItemEvent

Simulationskomponenten signalisieren Ereignisse, die bei Gegenständen auftreten können. Zum Beispiel signalisiert jede Komponente, wann sie einen Gegenstand erhält (*itemReceived*) oder einen Gegenstand abgibt (*itemProvided*). Eine Verarbeitungskomponente signalisiert den Bearbeitungsbeginn eines Gegenstands (*itemStarted*) sowie deren Ende (*itemFinished*). Andere Komponenten können sich bei diesen Ereignissen anmelden und dementsprechend darauf reagieren.

ItemProvider und *ItemReceiver*

Die Schnittstellen *ItemProvider* und *ItemReceiver* bilden die Grundlage für die Realisierung von Ressourcekomponenten und stellen die Basis für die Realisierung der Kopplung von Komponenten dar. *ItemProvider* definiert die Ausgangsschnittstelle für Ressourcen, die Gegenstände abgeben. Die gebundene Eigenschaft *hasItem* signalisiert, daß ein Gegenstand in einer Ressource verfügbar ist. Die Methode *inspectItem()* greift auf den verfügbaren Gegenstand zu, um ihn genauer untersuchen zu können, aber ohne ihn zu entnehmen. Durch die Methode *retrieveItem()* kann ein Gegenstand abgeholt werden. Das Ereignis *itemProvided* signalisiert dies. *ItemReceiver* definiert die Eingangsschnittstelle für Ressourcen, die Gegenstände aufnehmen können und bildet daher das Gegenstück zum *ItemProvider*. Die gebundene Eigenschaft *needsItem* signalisiert, daß ein Gegenstand aufgenommen werden kann. Über die Methode *testItem()* kann geprüft werden, ob eine Komponente einen bestimmten Gegenstand übernehmen möchte. Mit der Methode *receiveItem()* wird der Komponente ein Gegenstand übergeben. Das Ereignis *itemReceived* signalisiert den Empfang eines Gegenstandes.

6.2 Ressourcekomponenten

Auf der Grundlage der *ItemProvider*- und *ItemReceiver*-Schnittstellen wurden einige Basiskomponenten realisiert: *Generator*, *Sink*, *SingleServer*, *MultipleServer*, *Queue*, *Place* und *Delay*. Abbildung 4 zeigt die Vererbungs- bzw. Implementierungsbeziehungen dieser Komponenten.

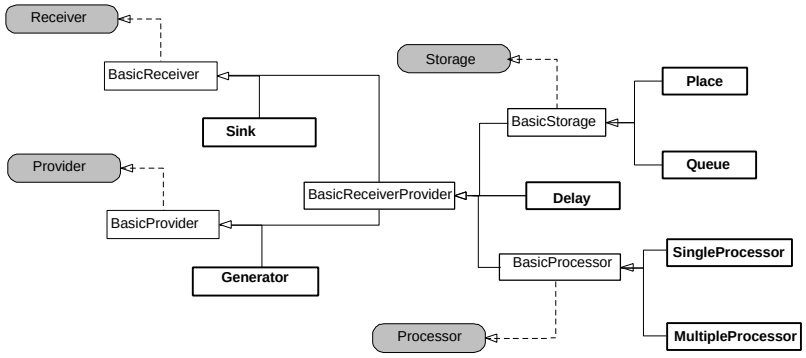


Abb. 4: Klassenhierarchie für atomare Komponenten
(Schnittstellendefinitionen sind grau)

Die Klasse *Generator* implementiert nur die *ItemProvider*-Schnittstelle und stellt nach einer einstellbaren Ankunftszeitverteilung neue Gegenstände zur Verfügung. Der *SingleServer* kann jeweils einen Gegenstand aufnehmen. Nach der Übernahme eines neuen Gegenstandes wird der Server als belegt markiert und die Bearbeitung gestartet. Wenn die Bearbeitungszeit verstrichen ist, wird ein *hasItem*-Ereignis signalisiert und auf die Entnahme gewartet. Der *MultipleServer* kann gleichzeitig eine einstellbare Anzahl (*capacity*) von Gegenständen bearbeiten. Die *Delay*-Komponente kann jederzeit beliebig viele Gegenstände übernehmen und verzögert deren Weitergabe um eine einstellbare Zeitdauer. Die *Place*-Komponente kann genau einen Gegenstand aufnehmen und wartet dann passiv bis dieser wieder entnommen wird. Entsprechend kann eine *Queue* eine einstellbare Anzahl von Gegenständen übernehmen und in FIFO-Reihenfolge wieder abgeben.

6.3 Kopplung und Steuerung

Komponenten werden nicht direkt sondern über explizite Kopplungskomponenten miteinander gekoppelt, die den Fluß der Items durchführen. Die Kopplungskomponenten arbeiten auf Basis der Schnittstellen *ItemProvider* und *ItemReceiver*. Kopplungskomponenten werden als Listener bei den *hasItem*- bzw. *needsItem*-Eigenschaften ihrer Ressourcen angemeldet und können beim Auftreten eines Ereignisses entsprechend ihrer Steuerungslogik und den Umgebungsbedingungen Items weiterleiten. Kopplung und Steuerung können von unterschiedlicher Komplexität sein: Es existieren einfache direkte Verbindungen, die lediglich Items von einem Provider zu einem Receiver weiterleiten, bis zu flexiblen Transportsystemen, die viele Provider und Receiver mit komplexer Steuerungslogik und Transporteinrichtung miteinander verbinden.

Die Komponente *ProviderReceiverConnection* realisiert einen direkten Itemfluß von einem *ItemProvider* zu einem *ItemReceiver*. Wenn eine derartige Verbindung ein *hasItem*- oder *needsItem*-Ereignis empfängt, wird geprüft, ob sich die andere Seite im gegenteiligen Zustand befindet, falls ja, wird das Item übergeben.

Die Komponente *ReceiverDecisionPoint* wird verwendet, um einen einzigen *ItemProvider* mit einer Menge von *ItemReceiver* zu koppeln. Die Auswahl desjenigen Receivers, der das nächste verfügbare Item erhält, wird von einem Steuerbaustein *ReceiverSelection* vorgenommen. Es existieren verschiedene Ausprägungen von *ReceiverSelection*-Komponenten, die unterschiedliche Auswahlstrategien implementieren. (z.B. Zufallsauswahl, prozentuell vorgegebene Zuordnung, auf Basis von Wartezeiten), bzw. sollen vom Anwender individuell definiert werden.

Die Komponente *ProviderDecisionPoint* koppelt in analoger Weise mehrere *ItemProvider* mit einer einzigen *ItemReceiver*-Komponente. Mit den Kopplungskomponenten *ProviderReceiverConnection*, *ProviderDecisionPoint* und *ReceiverDecisionPoint* (Abb. 5) können einfache Fließbandsysteme (Flow Shop-Modelle) realisiert werden.

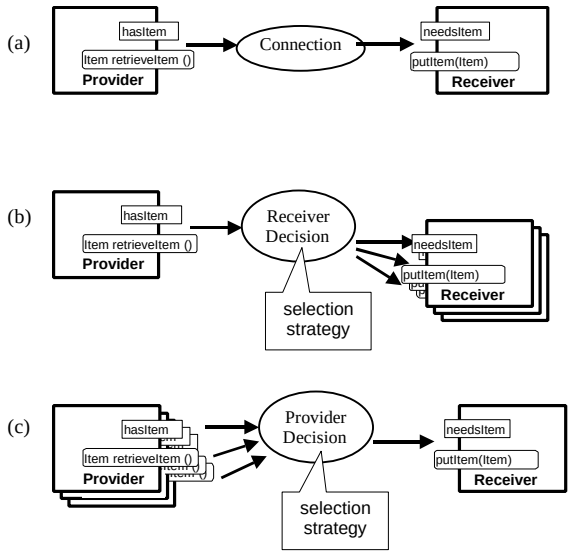


Abb. 5: Die Komponenten *Connection*, *ReceiverDecisionPoint* und *ProviderDecisionPoint*

Bei einer flexiblen Fertigungszelle mit einem Roboter wird ein komplexeres Kopplungs- und Steuerungssystem eingesetzt. Die Roboter-Komponente hat direkten Zugriff auf mehrere Provider und Receiver. Die dazugehörige Steuerung wird von vielen Zuständen im gesamten System beeinflußt. Die Steuerung des Roboters ist aber ähnlich wie die direkte Verbindung bei allen Providern und Receivern als Listener registriert und erhält damit entsprechende Ereignisse. In der Folge werden Transportbefehle für den Roboter erzeugt.

6.4 Visualisierungs-, Animations- und statistische Auswertungskomponenten

Visualisierung, Animation, Ausgabe und Auswertung werden gleichfalls mittels Komponenten realisiert. Es werden keinerlei Ausgaben oder Auswertungen fix mit Modellkomponenten verbunden. Statt dessen existieren in den Modellkomponenten gebundene Eigenschaften und Ereignisse, an denen Ausgabe- und Auswertungskomponenten angemeldet werden können. Ausgabe- und Auswertungskomponenten sind demnach lediglich Beobachter [Gam94] der Zustände der Modellkomponenten.

Zur Veranschaulichung der Flexibilität des komponentenbasierten Ansatzes, zeigen wir die Realisierung einer statistischen Auswertungskomponente. Bei der statistischen Auswertung unterscheidet man zwei grundsätzliche Arten: zeitabhängige Statistiken (Werte pro Zeiteinheit) und beobachtungsabhängige Statistiken (Werte pro Beobachtung). Die zeitabhängige Auswertungskomponente *VariableStatistics* kann an beliebige gebundene numerische Eigenschaften angedockt (attached) werden. Die beobachtete Variable meldet jede Zustandsänderung an die Auswertungskomponente, die die Daten akkumulieren und auswerten kann. Die Berechnung von Zeitauswertungen (z.B. mittlere Aufenthaltsdauer eines Items in einer Warteschlange) wird durch eine *ItemEventStatistics*-Komponente durchgeführt. Diese Komponente erhät von Items *startEvent* und *endEvent*-Ereignisse, mißt die dazwischen verstrichene Zeit und berechnet daraus statistische Maßzahlen.

7 Zusammenfassung und Ausblick

Modellierungs- und Simulationskonzepte wurden in Form einer prototypischen Komponentensammlung realisiert und in der Ausbildung von Informatikern und Wirtschaftsinformatikern eingesetzt. Die Hauptziel war die Evaluierung von Java und JavaBeans als Plattform für komponentenbasierte Simulation [Ber98] sowie die Überprüfung der Eignung des erstellten Baukastens für die Ausbildung in der Simulation [Pra98]. Unsere Erfahrungen mit JavaBeans als Standard-Komponentenmodell für Java sind überwiegend positiv. Lediglich im Bereich der Ereignisverarbeitung mußten wir einen asynchronen Mechanismus einführen.

Vielversprechend war die Verfügbarkeit von JavaBeans-konformen Entwicklungswerkzeugen, sodaß nicht schon zu Beginn eine Modellierungsumgebung entwickelt werden mußte. Unsere ersten Erfahrungen mit dem Einsatz von JavaBeans-konformen Entwicklungswerkzeugen (VisualAge von IBM, VisualCafe von Symantec und JBuilder von Borland) waren teilweise entmutigend. Kaum eines der Werkzeuge unterstützte JavaBeans in vollem Umfang, vielfach wurde Code generiert, der manuell nur schwer ergänzt werden konnte, oder es war nicht möglich, anwendungsspezifische Eigenschaftseditoren zu verwenden. Darüberhinaus verhielten sich die Werkzeuge hinsichtlich Laufzeit und Stabilität auch vielfach unbefriedigend. Erst mit JBuilder 2.0 fanden wir ein Werkzeug, daß unsere Anforderungen und Bedürfnisse einigermaßen erfüllte. Zusammenfassend läßt sich sagen, daß die derzeitigen Entwicklungswerkzeuge das Experimentieren mit unseren Komponenten ermöglichen; für eine professionelle Arbeit mit Komponenten ist die Entwicklung eines eigenen Werkzeuges allerdings unverzichtbar.

Im Moment ist nach der interaktiven Auswahl und Zusammenstellung der Komponenten und deren Voreinstellung manuelle Programmiertätigkeit notwendig, um die Kopplungen zwischen den Modellkomponenten und die Verbindungen zu den Ausgabe- und Auswertungskomponenten zu erstellen. Dies ist für Programmierer einfach zu bewerkstelligen, kann aber Simulationstechnikern in der Praxis nicht zugemutet werden. Zur Zeit entwickeln wir neue, mächtige Komponenten, um unsere Konzepte auch an praxisrelevanten Aufgaben zu erproben. Weiters arbeiten wir an einer komponentenbasierten Entwicklungsumgebung, die mit geringem Aufwand an unterschiedliche Bedürfnisse der verschieden Anwendergruppen angepaßt werden kann. Wichtige Fragestellungen hierbei betreffen Benutzerunterstützung, -führung und -kontrolle (unzulässige Strukturen dürfen nicht zusammengebaut werden können) sowie Unterstützung des Benutzers bei der Definition von dynamischem Verhalten.

8 Literatur

[Bar97] Barros, F.J., Modeling Formalisms for Dynamic Structure Systems, *ACM Transaction on Modeling and Simulation*, Vol 7, No. 4, 1997, pp. 501- 515.

[Ber98] Berg, K., B. Marek, G. Pomberger, J. Sametinger, A. Stritzinger, *Java - Ein Schnappschuß*, Praxis der Wirtschaftsinformatik, Heft 202, Aug. 98, pp. 102-126

[Boo98] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language, Users Guide*, Addison Wesley, 1998.

[Cel91] Cellier, F.E., *Continuous System Modeling*, Springer-Verlag 1991.

[Gam94] Gamma, E., et al, *Desgin Patterns*, Addison Wesley, 1994

[Gos95] Gosling, J. and H. McGilton, *The Java Language Environment – A White Paper*; Sum Microsystems, MV California, 1995.

[Har98] Harel, D. and M. Politi, *Modeling Reactive Systems with Statecharts*, McGrawHill, 1998.

[Hel98] Helmquist et al 98, ModelicaTM – *A Unified Object-Oriented Language for Physical Systems Modeling; Tutorial and Rationale, Version* 1.1, December 15. 1998, URL: http://www.modelica.org.

[Hil99] Hilpold, T., *JavaBeans Property Editors*, Technical report, C. Doppler Laboratory for Software Engineering, Johannes Kepler University, Linz, Austria 1999 (in German).

[Mat96] Mathworks Inc., *SIMULINK User's Guide*, 1996.

[Pay61] Payntner, H. M., *Analysis and Design of Engineering Systems*, MIT Press, Cambridge 1961.

[Pic92] Pichler, F. and H. Schwärtzel (eds.), *CAST Methods in Modeling*, Springer, 1992.

[Pra96] Praehofer, H., An Environment for DEVS-Based Multiformalism Modeling and Simulation in C++. *Proc of Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, San Diego, CA, 1996.

[Pra98] Praehofer, H., A. Stritzinger, and J. Sametinger, Using JavaBeans to teach Simulation and using Simulation to teach JavaBeans, *ESM98, 12th European Simulation Multiconference*, Manchester, UK, June 16-19, 1998.

[Pra93] Praehofer, H., and D. Pree., Visual Modeling of DEVS-based Systems Based on Higraphs. *Winter Simulation Conference*, 1993, S. 595-603.

[Pra97] Praehofer, H., *Systems Theoretic Foundations for Combined Discrete-Continuous System Simulation*, PhD Thesis, Johannes Kepler University, Linz, Austria 1991.

[Pra 00] Praehofer, H., Schoeppl, A., A Continuous and Combined Simulation Platform in Java and its Application in Building Paper Mill Training Simulators, *WebSim 2000 Conference*, SCS, San Diego, CA, January 2000.

[Sam97] Sametinger, J., *Software-Engineering with Reusable Components*, Springer-Verlag, 1997.

[Sch99] Schöppl, A., *Training simulators for papermill operators*, Diploma thesis, Dept of Systems Theory and Information Engineering, Johannes Kepler University, Linz, Austria 1999 (in German).

[Str97] Stritzinger, A., *Komponentenbasierte Softwareentwicklung*, Addison-Wesley, 1997.

[Sun97] Sun Microsystems: *JavaBeans 1.01 API Specification.* 1997. see http://java.sun.com/Beans/spec.html

[Szy98] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Tho94] Thomas, C., Interface-based classification of simulation models, *Winter Simulation Conference* 94, Orlando, FL, 1994.

[Uhr96] Uhrmacher, A.M. and B.P. Zeigler, *Variable Structure Modelling in Object-Oriented Simulation*, *International Journal on General Systems*, Vol. 24(4), 359-375, 1996.

[Zei76] Zeigler, B.P., *Theory of Modeling and Simulation*; Wiley 1976

[Zei84] Zeigler, B.P., *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, 1984.

[Zei90a] Zeigler, B.P., *Object Oriented Simulation with Modular, Hierarchical Models.* Academic Press, 1990.

[Zei90b] Zeigler, B.P. and H. Praehofer, Sytems Theory Challanges in the Simulation of Variable Structure Systems, *EUROCAST* '89; LNCS 410, Springer-Verlag, 1990, pp. 41-50.

[Zei99] Zeigler, B.P., H. Praehofer, and T.G. Kim, *Theory of Modeling and Simulation*, 2nd Edition. Academic Press, 1999 (in preparation).