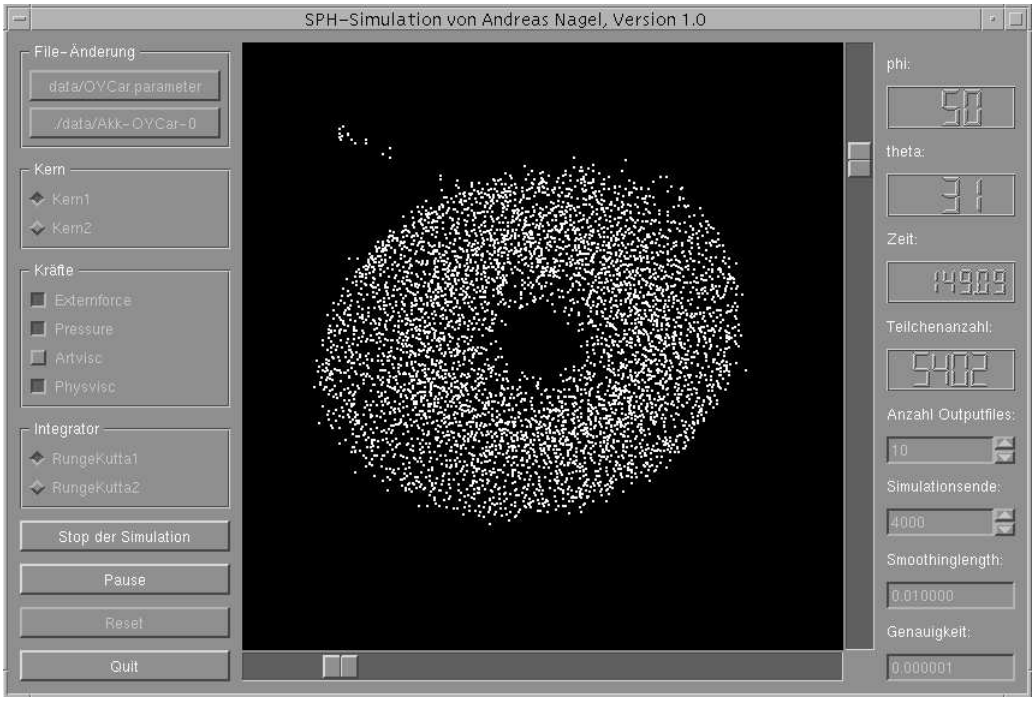
Implementierung von SPH-Methoden in eine objektorientierte Klassenbibliothek

Diplomarbeit
vorgelegt von

Andreas Nagel



November 2000

Implementierung von SPH-Methoden in eine objektorientierte Klassenbibliothek

Diplomarbeit
vorgelegt von

Andreas Nagel

Angefertigt am

Institut für Astronomie und Astrophysik
der Eberhard-Karls-Universität Tübingen

und

Wilhelm-Schickard-Institut für Informatik
der Eberhard-Karls-Universität Tübingen

November 2000

Einleitung

Begriffe wie C++, Java, Objektorientierung, UML[1], Design-Pattern[2], Unified Process[3], u.s.w. sind heutzutage ***die Schlagworte*** der Softwareindustrie. Dabei geht es um die effiziente Herstellung und Wartung von Softwareprodukten, wobei man hier unter Effizienz den Bau von gut zu erweiternder und wiederverwendbarer Software versteht.

Die Aufgabe dieser Diplomarbeit ist es nun, ein in C geschriebenes SPH[4]-Simulationsprogramm in UML und mit Hilfe des CASE[5]-Tool ***Together***[6] objektorientiert zu strukturieren und in C++ zu implementieren. Ein besonderes Augenmerk soll dabei auf Design-Pattern, d.h. auf die leichte Erweiterung und Wartung des Programms, gelegt werden. Die dadurch entstandene SPH-Klassenbibliothek (im weiteren ***SPH++*** genannt) wird dadurch die oben beschriebene effizientere Struktur gegenüber dem in C geschriebenen Programm besitzen.

Darüberhinaus soll die Bediehnung von SPH++ mit Hilfe einer graphischen Oberfläche erfolgen. Das Titelbild zeigt die mit Hilfe der Qt[7]-Bibliothek entstandene GUI[8] von SPH++.

Die Arbeit gliedert sich in sechs Kapitel. Das erste Kapitel gibt eine kurze Einführung in die Kataklysmischen Variablen, um den physikalischen Hintergrund dieser Diplomarbeit besser zu verstehen. Im zweiten Kapitel werden die physikalischen Grundlagen der Hydrodynamik und die SPH-Näherung eingeführt. Das dritte Kapitel beschreibt die Anforderungen, die SPH++ erfüllen soll. Anhand dieser Anforderungen wird in Kapitel 4 die Architektur bzw. das Design entwickelt und implementiert. Im fünften Kapitel wird diese Implementierung getestet und mit dem ursprünglichen C-Programm verglichen. Kapitel 6 enthält eine Zusammenfassung dieser Diplomarbeit und gibt einen möglichen Ausblick in die weitere Zukunft von SPH++.

Im Anhang wird auf drei elementare Bereiche der Software-Entwicklung näher eingegangen. Der erste Teil dieses Anhangs ist eine kurze Einführung in UML, der Beschreibungssprache objektorientierter Programme. Der zweite Teil beeinhaltet die in dieser Diplomarbeit verwendeten Design-Pattern und der letzte Teil gibt einen kurzer Überblick über den Unified Process als Beispiel eines Software-Entwicklungsprozesses.

Auf der dieser Diplomarbeit beiliegenden CD befindet sich all die Dinge, die während der Diplomarbeit entwickelt oder verwendet wurden. Die HTML-Seite „/cdrom/index.html“ gibt dabei einen Überblick über den Inhalt der CD, wobei „/cdrom“ für das Verzeichnis der gemounteten CD steht.

[1] UML steht für „Unified Modelling Language“. Siehe Anhang A.
[2] Siehe Anhang B.
[3] Siehe Anhang C.
[4] SPH steht für „Smoothed Particle Hydrodynamics“ und ist ein spezielles Verfahren zur Lösung von partiellen Differentialgleichungen. Siehe auch Kapitel 2.
[5] CASE steht für „Computer-Aided Software/System Engineering“.
[6] Siehe www.togethersoft.com.
[7] Qt ist eine GUI-Bibliothek der Firma Troll Tech. Siehe http://www.troll.no.
[8] Abkürzung für Graphical User Interface (graphische Benutzeroberfläche).

Noch eine abschließende Bemerkung zu dem gewählten Namen der SPH-Bibliothek „SPH++“. Dieser ergab sich einerseits in Anlehnung an die Programmiersprache C++ und andererseits aus der in Zukunft verwendeten parallelen Umgebung TPO++[9].

[9] TP0++ steht für Tübinger parallele Objekte.

Inhaltsverzeichnis

Kapitel 1

Kataklysmische Variable

Der physikalische Hintergrund dieser Diplomarbeit sind die sog. Kataklysmischen Variablen. Sie werden im nächsten Abschnitt näher spezifiziert und anhand der beobachteten Lichtkurven in Klassen eingeteilt. Anschließend wird anhand des Roche-Modells die Funktionsweise der kataklysmischen Variablen beschrieben. Für tiefergehende Betrachtungen wird auf [Accretion Power] und [Hack & la Dous] verwiesen.

1.1 Klassifizierung der Kataklysmischen Variablen

Unter dem Begriff „Kataklysmische Variable“(engl. cataclysmic variable, CV) werden veränderliche[1] Objekte zusammengefaßt, die folgende Eigenschaft besitzen:

> ***CVs sind Doppelsternsysteme, die durch das Roche-Modell (Abschnitt 1.2) beschrieben werden und deren Primärstern ein Weißer Zwerg ist. Der Sekundärstern ist dabei ein Hauptreihenstern, der sein Roche-Volumen vollständig ausfüllt und daher Masse an den Primärstern verliert (Abschnitt 1.3).***

Das Besondere an CVs sind die großen und schnell verlaufenden Helligkeitsänderungen, die sog. „Ausbrüche“. Nach deren Häufigkeit und Stärke kann man die CVs in folgende 4 Klassen einteilen:

- ***Novae***: Als Novae bezeichnet man Sterne, für die innerhalb historischer Zeit nur ein Ausbruch beobachtet wurde. Der Helligkeitsanstieg dieses Ausbruchs liegt zwischen 10 und 20 mag[2] und klingt innerhalb von Monaten bis Jahren wieder ab. Als Ursache dieses Ausbruchs wird eine thermonukleare Explosion auf der Oberfläche des weißen Zwerges angenommen.

- ***Rekurrierende Novae***: Bei einer Rekurrierenden Novae wurden mehrere Ausbrüche in Abständen von Monaten bis Jahrzehnten beobachtet. Die Helligkeitsänderung beträgt 8 bis 15 mag und klingt nach einigen Monaten wieder ab. Der Unterschied zur Novae liegt wahrscheinlich nur in deren sehr langer Ruhephase, die nach einem weiteren beobachteten Ausbruch daher zu einer rekurrierenden Novae wird.

- ***Zwergnovae***: Als Zwergnovae werden Objekte mit quasiperiodische Ausbrüchen von 3 bis 5 mag bezeichnet. Die Dauer der Ausbrüche beträgt einige Tage und deren Zeitabstände 10 bis 100 Tagen. Als Ursache dieser Ausbrüche werden thermische Instabilitäten der Akkretionsscheibe angenommen. Aufgrund unterschiedlicher Langzeitlichtkurven wird diese Klasse der Zwergnovae noch in weitere Unterklassen aufgeteilt, die nach dem jeweils ersten beobachteten Vertreter benannt sind:

[1] d.h. Sterne, deren Helligkeit nicht konstant ist.

[2] Die Abkürzung „mag“ stammt von dem lateinischen Wort magnitudo (Größe) und ist die Maßeinheit, mit der die Helligkeit von Sternen, Planeten und anderen Himmelskörpern angegeben wird. Eine Änderung um 1 mag entspricht einer 2,5-fachen Helligkeitsänderung.

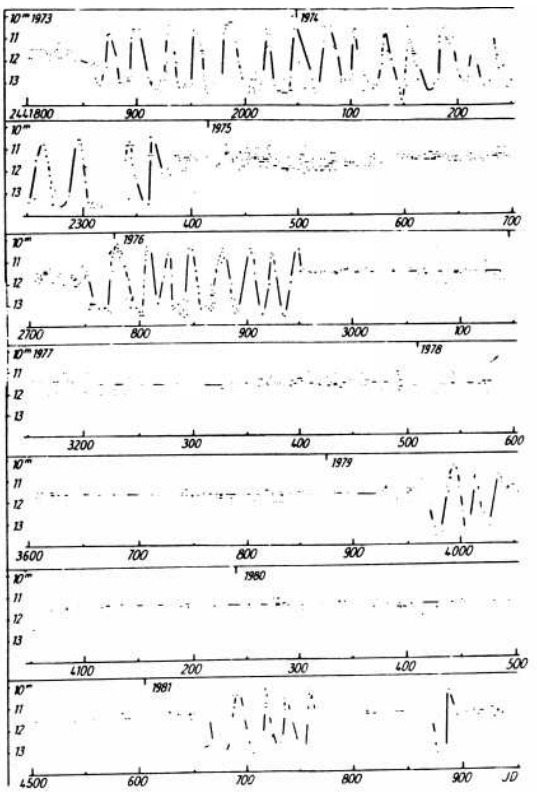


Abbildung 1.1: Ausbruchslichtkurve der Zwergnova Z Cam (siehe [Hack & la Dous] S.18). Ab und zu zeigen sich Stillstände mit einer Dauer von einigen Monaten bis hin zu mehreren Jahren.

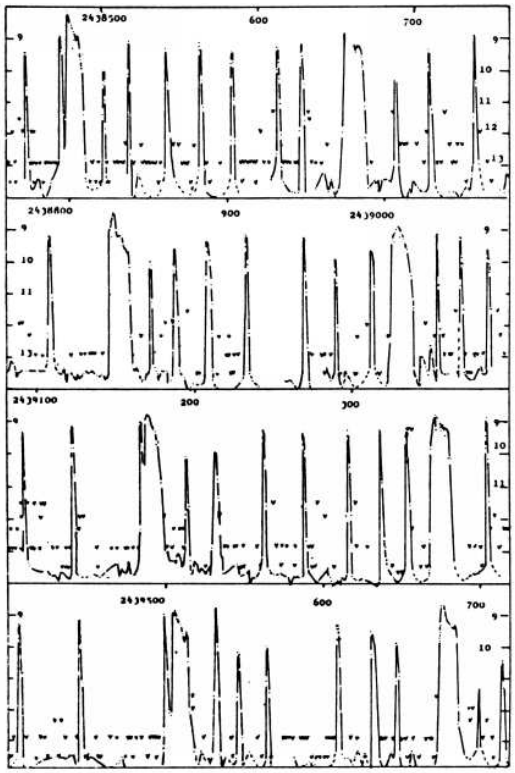


Abbildung 1.2: Ausbruchslichtkurve der Zwergnova VW Hyi (siehe [Hack & la Dous] S.18). Neben den normalen Ausbrüchen gibt es noch sog. Superausbrüche, die heller und länger sind.

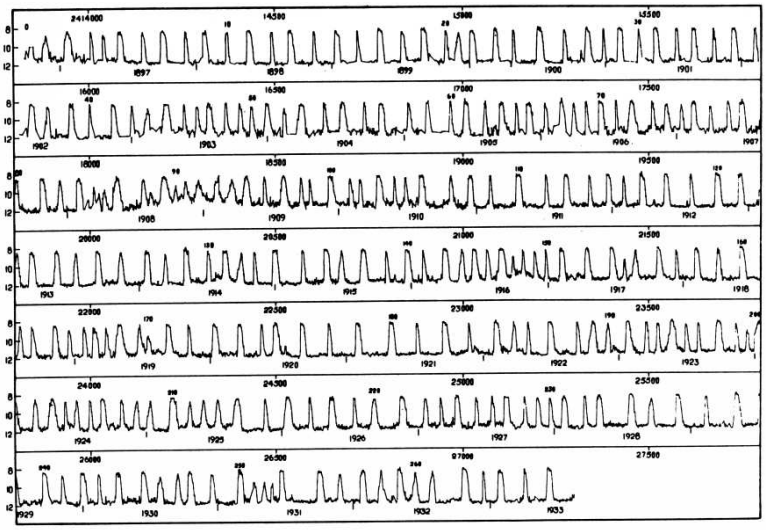


Abbildung 1.3: Ausbruchlichtkurve der Zwergnova SS Cygni von 1896 - 1933 (siehe [Hack & la Dous] S.18). Kürzere und längere Ausbrüche treten in unregelmäßiger zeitlicher Folge auf.

  - ***Z-Camelopardalis-Sterne***: Diese Sterne zeigen zu manchen Zeiten sog. Stillstände: Dabei bleibt die Helligkeit nach dem Ende eines Ausbruchs für unbestimmte Zeit, die zwischen einigen Tagen und mehreren Jahren betragen kann, konstant. Die Helligkeit dieser Stillstände ist etwa 0.7 mag unter der Maximalhelligkeit des Ausbruchs (siehe Abb. 1.1).
  - ***SU-Ursae Majoris-Sterne***: Bei diesen Sternen gibt es außer den normalen Ausbrüchen noch sog. „Superausbrüche“, die ca. 1 mag heller sind und 5 bis 10 mal länger dauern als die normalen Ausbrüche (siehe Abb. 1.2).
  - ***SS-Cygni- oder U-Geminorum-Sterne***: Diese Sterne zeigen nur einfache Ausbrüche in quasiperiodischen Abständen (siehe Abb. 1.3).

- ***Novaähnliche Sterne***: Das Ausbruchverhalten dieser Klasse ist nicht regelmäßig, aber sie ist photometrisch und spektroskopisch den Zwergnovae sehr ähnlich.

1.2 Das Roche-Modell

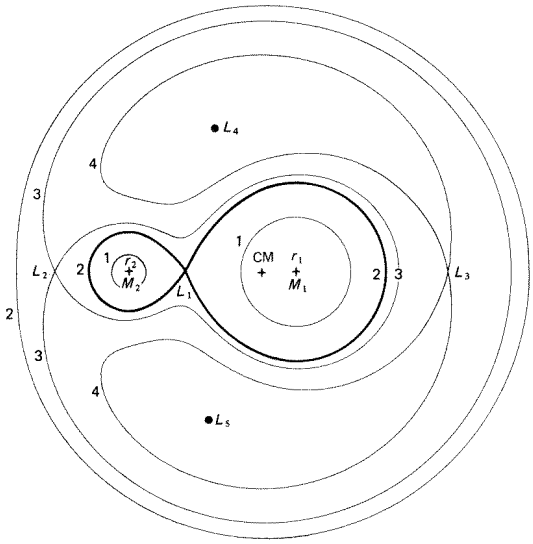


Abbildung 1.4: Roche-Potential: Äquipotentiallinien, Schnitt durch die Bahnebene für das Massenverhältnis q=M_1/M_2=5. Die dicke Linie stellt die Roche-Lobes dar. Aus [Hack & la Dous].

Das Roche-Modell beschreibt die Eigenschaften eines Doppelsternsystems durch das sog. Roche-Potential Φ_R. Die zwei Sterne bewegen sich dabei kreisförmig um den gemeinsamen Schwerpunkt. Im mitrotierenden Koordinatensystem ergibt sich das Potential zu:

$$\Phi_R(\vec{r}) = -\frac{GM_1}{|\vec{r}-\vec{r}_1|} - \frac{GM_2}{|\vec{r}-\vec{r}_2|} - \frac{(\vec{\omega}\times\vec{r})^2}{2}. \tag{1.1}$$

Hierbei ist G die Gravitationskonstante, $\vec{r}_1$, $\vec{r}_2$, M_1 und M_2 die Orte bzw. die Massen der Sterne und $\vec{\omega}$ die Winkelgeschwindigkeit des Systems. Ihr Betrag ist

$$\omega = \sqrt{\frac{G(M_1+M_2)}{|\vec{r}_1-\vec{r}_2|^3}}. \tag{1.2}$$

Die rechte Seite der Gl. (1.1) besteht aus den beiden Gravitationstermen und dem Zentrifugalterm. Die Corioliskräfte, die auf ein bewegtes Teilchen im System wirken, sind hier jedoch nicht

enthalten. In Abb. 1.4 sind einige Äquipotentiallinien des Roche-Potentials in der Bahnebene der zwei Sterne zu sehen. Die Extremalpunkte des Potenzials wurden mit L_1 bis L_5 gekennzeichnet, wobei die dicke Äquipotentiallinie, die durch den Lagrangepunkt L_1 geht, den sog. „Roche-Lobe“ umschließt. Der Punkt L_1 ist dabei ein Sattelpunkt, der die Potentialmulden der beiden Sterne voneinander trennt.

1.3 Anwendung des Roche-Modells auf Kataklysmische Variable

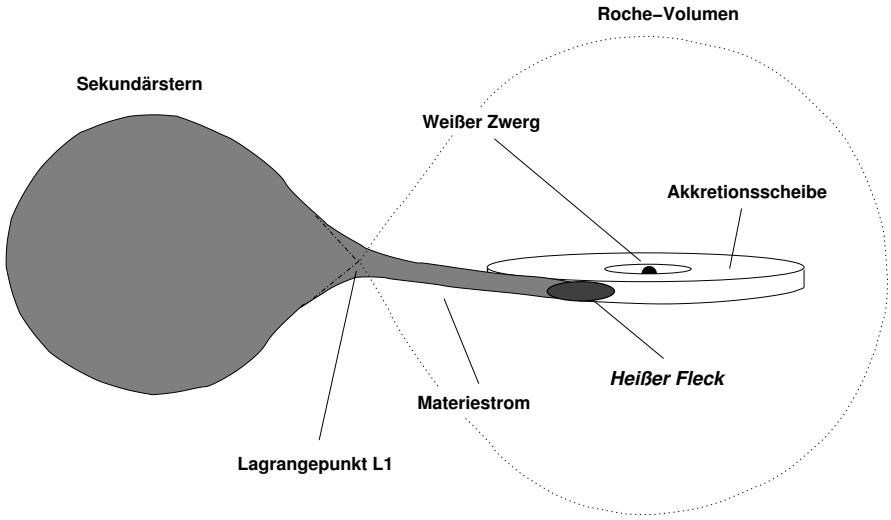


Abbildung 1.5: Skizze einer kataklysmischen Variablen. Da der Sekundärstern seinen Roch-Lobe vollständig ausfüllt, gibt es einen Materiestrom, der zur Ausbildung einer Akkretionsscheibe führt.

Bei den CVs füllt der Hauptreihenstern seinen Roche-Lobe vollständig aus und ist daher nicht rund, sondern birnenförmig. Dabei verliert er Masse über den Lagrangepunkt $L1$ an den weißen Zwerg, der viel kleiner als sein Roche-Volumen ist (Roche-lobe-overflow). Diese Masse fliegt nun aufgrund der Corioliskraft aber nicht direkt auf den Primärstern zu, sondern bildet eine sog. Akkretionsscheibe um den weißen Zwerg (siehe Abb. 1.5). Der Auftreffpunkt der überfließenden Masse mit der Akkretionsscheibe wird als „Hot Spot“ bezeichnet, der, wie der Name schon sagt, eine höhere Temperatur als die Scheibe besitzt und deshalb mehr Strahlung als diese emittiert.

Die Viskosität in der Scheibe führt schließlich dazu, daß die Materie in der Scheibe sich nach innen bewegt und von dem weißen Zwerg „akkretiert“, d.h. aufgenommen, wird. Dabei wird ein Teil der potentiellen Energie als Strahlung abgegeben und stellt ein nicht zu vernachlässigten Anteil an der Gesamthelligkeit des Systems dar. Bei den Ausbrüchen von Zwergnovae-Systemen ist dieser Anteil der sogar weitaus größte.

Die beiliegende CD enthält ein MPEG-Video[3], das die Entstehung der Lichtkurve der doppeltbedeckten Zwergnovae OY Carinae veranschaulicht.

Das in dieser Diplomarbeit entwickelte Programm simuliert die zeitliche Entwicklung solch einer Akkretionsscheibe mit Hilfe der SPH-Methode, die im nächsten Kapitel eingeführt wird.

[3] siehe „/cdrom/data/films/kurve.mpg“. Dieses Video wurde der Seite http://www.tat.physik.uni-tuebingen.de/Forschung/Astrophysik/CVS/mpg/kurve.html entnommen.

Kapitel 2

Smoothed Particle Hydrodynamics

Akkretionsscheiben, die ein Teil der im vorigen Kapitel eingeführten Kataklysmischen Variablen sind, bewegen sich nach den Gesetzen der im folgenden Abschnitt behandelten ***Hydrodynamik***. Um die Bewegung einer Akkretionsscheibe auf dem Rechner zu simulieren, muß man also die Grundgleichungen der Hydrodynamik lösen. Eines von mehreren Lösungsverfahren heißt ***Smoothed Particle Hydrodynamics (SPH)***(Abschnitt 2.2), das ein System partieller Differentialgleichungen in ein System gewöhnlicher gekoppelter Differentialgleichungen überführt, die dann mit numerischen Standardmethoden gelöst werden können. Diese Überführung der hydrodynamischen Grundgleichungen in ihre SPH-Form wird im Abschnitt 2.3 durchgeführt. Der letzte Abschnitt dieses Kapitels zeigt, wie man eine künstliche Viskosität in diese Gleichungen einbaut.

2.1 Hydrodynamische Grundgleichungen

Die Hydrodynamik ist die Theorie der Bewegung von Flüssigkeiten und Gasen (zusammengefaßt unter dem Oberbegriff ***Fluide***) und ist aus der klassischen Physik abgeleitet. Ihre Gleichungen besagen unter anderem, daß die Bewegung eines Fluids durch dessen Geschwindigkeitsfeld $\vec{v}(\vec{r},t)$ und zwei beliebigen thermodynamischen Größen vollständig festgelegt ist. Üblicherweise nimmt man für diese Größen die Dichte $\rho(\vec{r},t)$ und den Druck $p(\vec{r},t)$. Im folgenden werden die für diese Diplomarbeit wichtigen Gleichungen kurz zusammengefaßt. (Für eine ausführliche Herleitung siehe [Landau]).

2.1.1 Die Kontinuitätsgleichung

Aus der ***Massenerhaltung*** des Fluids folgt für ein beliebiges Volumen V, daß die Änderung der Dichte des Fluids in V gleich dem Massenstrom durch dessen Oberfläche sein muß. Diese Massenbilanz lautet daher folgendermaßen:

$$\frac{\partial}{\partial t}\iiint_V \rho\, dV = -\oiint_{\partial V} \rho\,\vec{v}\cdot d\vec{f}$$

Mit Hilfe des Gaußschen Satzes und da V beliebig ist, folgt daraus die ***Kontinuitätsgleichung***:

$$\frac{\partial \rho}{\partial t} + \nabla(\rho\,\vec{v}) = 0 \iff \frac{d\rho}{dt} + \rho\,\nabla\vec{v} = 0. \tag{2.1}$$

2.1.2 Die Navier-Stokes-Gleichung

Die gleichen Überlegungen kann man für die ***Impulserhaltung*** des Fluids anstellen. Für ein beliebiges Volumen V ist die Änderung des Impulses in V gleich dem Impulsstrom durch dessen

Oberfläche plus den auf V wirkenden Kräften nach dem Newtonschen Gesetz. Diese Kräfte kann man weiter in ***Oberflächenkräfte*** $\vec{k}$ (z.B. Druck, Reibung) und in externe ***Volumenkräfte*** $\vec{f}$ (z.B. Gravitationskraft) aufteilen. Die Impulsbilanz lautet also folgendermaßen:

$$\frac{\partial}{\partial t}\iiint_V \rho\vec{v}\,dV = -\oiint_{\partial V}(\rho\,\vec{v})\,\vec{v}\cdot\vec{df} + \oiint_{\partial V}\vec{k}df + \iiint_V \rho\vec{f}dV$$

Werden die Oberflächenkräfte $\vec{k}$ noch in einer senkrechten Anteil, dem sog. ***Druck*** $-p\hat{\vec{n}}$ ($\hat{\vec{n}}$ ist der Normalenvektor) und in einen tangentialen Anteil ($\neq 0$, falls Reibung auftritt) aufgeteilt, so ergibt sich mit Hilfe des Gaußschen Satzes und da V beliebig ist die ***Navier-Stokes-Gleichung*** (über doppelt vorkommende Indizes wird summiert) :

$$\rho\frac{dv_\alpha}{dt} = -\frac{\partial p}{\partial x_\alpha} + \frac{\partial T_{\alpha\beta}}{\partial x_\beta} + \rho f_\alpha. \tag{2.2}$$

Der Tensor T wird als ***zäher bzw. Reibungs-Spannungtensor*** bezeichnet und hat im Falle der ***Newtonsche Fluide*** (Beispiele sind Wasser und Luft), für die T nur von den ***ersten*** räumlichen Ableitungen der Geschwindigkeit ***linear*** abhängt, folgende Gestalt:

$$T_{\alpha\beta} = \eta\underbrace{\left(\frac{\partial v_\alpha}{\partial x_\beta} + \frac{\partial v_\beta}{\partial x_\alpha} - \frac{2}{3}\delta_{\alpha\beta}\frac{\partial v_\gamma}{\partial x_\gamma}\right)}_{=:\sigma_{\alpha\beta}} + \zeta\delta_{\alpha\beta}\frac{\partial v_\gamma}{\partial x_\gamma}. \tag{2.3}$$

Dabei ist der erste Teil der rechten Seite $\eta\sigma_{\alpha\beta}$ so gewählt worden, daß seine Spur verschwindet. η ist der Koeffizient der ***Scherviskosität*** ($\nu := \frac{\eta}{\rho}$ ist die kinematische Zähigkeit) und ζ der Koeffizient der ***Volumenviskosität***. Beide Koeffizienten sind von der Geschwindigkeit unabhängig und positiv. $\sigma_{\alpha\beta}$ wird als ***Schertensor*** bezeichnet. Da der Anteil der Volumenviskosität meist gering ist, wird ζ üblicherweise gleich null gesetzt. Man spricht dann von ***Maxwellschen Fluiden***.

Bei idealen Fluiden (ohne Reibung und Wärmeleitung) verschwindet der Spannungstensor vollständig und man erhält die einfachere ***Eulergleichung***:

$$\rho\frac{d\vec{v}}{dt} = -\nabla p + \rho\vec{f}. \tag{2.4}$$

2.1.3 Die polytrope Zustandsgleichung

Da es zu den drei Unbekannten: ρ, $\vec{v}$ und p bis jetzt nur zwei Gleichungen gibt, wird im allgemeinen die ***Energiegleichung*** zu dem Gleichungssystem hinzugenommen. Die Energiegleichung folgt dabei aus dem ersten Hauptsatz der Thermodynamik, d.h. aus der ***Energieerhaltung*** des Fluids. Bei der Herleitung der Energiegleichung wird wie schon vorher zuerst die Bilanzgleichung aufgestellt und diese danach in eine integrale Form umgewandelt. Nun hat man zwar eine Gleichung hinzugewonnen, jedoch gibt es auch eine neue Unbekannte ϵ, die spezifische innere Energie. Zum Abschluß des Gleichungssystem wird daher noch nach einer (materialabhängigen) Zustandsgleichung gesucht, die eine Verbindung zwischen den vier Unbekannten herstellen soll.

Im Falle dieser Diplomarbeit hat diese Zustandsgleichung die Form:

$$p = k\rho^\gamma, \tag{2.5}$$

die man als ***Polytrope Zustandsgleichung*** bezeichnet und die für ideale Gase gültig ist. Dabei ist $\gamma = \frac{2}{f} + 1$ der ***Polytropenindex***, wobei f die Zahl der Freiheitsgrade angibt. Für ein einatomiges Gas ist γ gleich $\frac{5}{3}$. Die Konstante k ist vom Material abhängig. Da die Gleichung (2.5) die Unbekannte ϵ nicht enthält, kann man aus dem Gleichungssystem die Energiegleichung wieder entfernen, wie es in dieser Diplomarbeit auch geschah.

2.2 Das SPH-Verfahren

Smoothed Particle Hydrodynamics (SPH) ist ein Näherungsverfahren, um die Bewegung eines Fluids zu berechnen. Dazu werden sogenannte SPH-Teilchen eingeführt, die im Gegensatz zu festen Gittermethoden sich mit dem Fluid mitbewegen. Auch werden dichtere Gebiete durch mehr Teilchen angenähert, was den Vorteil hat, daß die Auswertung leerer Gebiete keine Rechenzeit benötigt. Der Nachteil liegt jedoch in der nicht überall gleichen Ortsauflösung. Die Größe der Wechselwirkung mit den Nachbarteilchen bestimmt der sog. ***Kernel***. Für eine ausführliche Darstellung der SPH-Methode, insbesondere im Zusammenhang mit astrophysikalischen Anwendungen, wird auf [Speith] verwiesen.

2.2.1 Das Smoothing

Definition (Smoothing). Als Smoothing einer Funktion $f : \mathbb{R}^3 \to \mathbb{R}$ bezeichnet man ihr Faltungsintegral

$$< f(\vec{r}) >:= \iiint_{\mathbb{R}^3} f(\vec{r}\,')W(\vec{r},\vec{r}\,',h)\,dx'^3 \tag{2.6}$$

wobei die Kernelfunktion (engl. ***kernel***) $W : \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}$ folgende Eigenschaften hat:

1. Konvergenz: $lim_{h\to 0} W(\vec{r},\vec{r}\,',h) = \delta(\vec{r}-\vec{r}\,')$
2. Normierung: $\iiint_{\mathbb{R}^3} W(\vec{r},\vec{r}\,',h)\,dx'^3 = 1$
3. Translationsinvariant und Kugelsymmetrie: $W(\vec{r},\vec{r}\,',h) = W(|\vec{r}-\vec{r}\,'|,h)$
4. Stetige Differenzierbarkeit, d.h. $W \in C^1_{\mathbb{R}^3}(\mathbb{R})$

Wegen Punkt 3 wird ab jetzt der Definitionsbereich von W auf $\mathbb{R}\times\mathbb{R}$ reduziert mit dem neuen Parameter $r := |\vec{r}-\vec{r}\,'|$. Der Parameter h wird ***Smoothing Length*** genannt und ist frei wählbar. Er ist ein Maß für die Breite des Kernels. Als Kernel wird meist ein kubischer Spline mit kompaktem Träger genommen, d.h. $W(r,h) = 0$ für $|r| > h$. Durch Taylorentwicklung von (2.6) kann gezeigt werden, daß der Fehler beim Smoothen von zweiter Ordnung in h ist, d.h. $f(\vec{r}) =< f(\vec{r}) > +O(h^2)$.

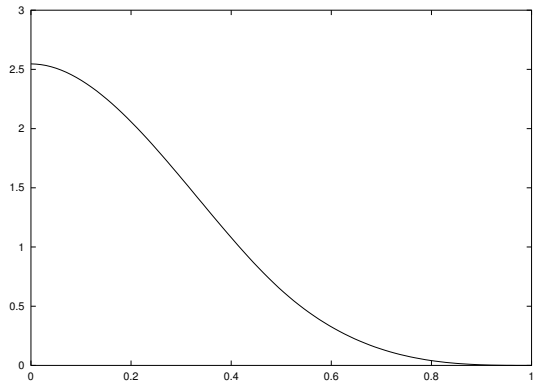


Abbildung 2.1: W(r,h) mit h=1.

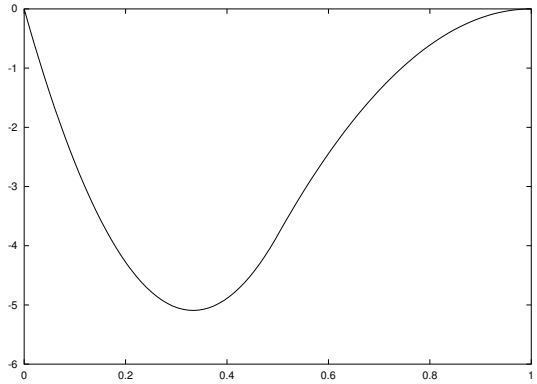


Abbildung 2.2: $\frac{\partial}{\partial x}W(r,h)$ mit h=1.

Der in dieser Diplomarbeit verwendete Kernel (wobei leicht weitere Kernel alternativ eingebunden werden können) ist aus Polynomen dritten Grades zusammengesetzt und hat folgende Form (siehe Abb. 2.1):

$$W(r,h) = \begin{cases} \frac{8}{\pi h^3}\left(6\frac{r^3}{h^3} - 6\frac{r^2}{h^2} + 1\right) & : \quad 0 \le r \le \frac{h}{2} \\ \frac{16}{\pi h^3}\left(1-\frac{r}{h}\right)^3 & : \quad \frac{h}{2} < r \le h \\ 0 & : \quad r > h \end{cases} \tag{2.7}$$

Die Ableitung von W(r,h) hat folgende Form (siehe Abb. 2.2):

$$\frac{\partial}{\partial x_\alpha} W(r,h) = \begin{cases} \frac{48}{\pi h^5}\left(3\frac{r}{h} - 2\right)x_\alpha & : \quad 0 \le r \le \frac{h}{2} \\ -\frac{48}{\pi h^4 r}\left(1 - \frac{r}{h}\right)^2 x_\alpha & : \quad \frac{h}{2} < r \le h \\ 0 & : \quad r > h \end{cases} \tag{2.8}$$

2.2.2 Die gesmoothten Ortsableitungen

Wenn man die erste Ableitung von f in (2.6) einsetzt, so ergibt sich :

$$\begin{aligned} < \frac{\partial f(\vec{r})}{\partial x_\alpha} > \quad &= \quad \iiint_{\mathbb{R}^3} \frac{\partial f(\vec{r}\,')}{\partial x'_\alpha} W(|\vec{r} - \vec{r}\,'|, h)\, dx'^3 \\ &\overset{part.Int.}{=} \quad - \iiint_{\mathbb{R}^3} \big(f(\vec{r}\,') + g(\vec{r})\big) \frac{\partial W(|\vec{r} - \vec{r}\,'|, h)}{\partial x'_\alpha}\, dx'^3 \\ &= \quad \iiint_{\mathbb{R}^3} \big(f(\vec{r}\,') + g(\vec{r})\big) \frac{\partial W(|\vec{r} - \vec{r}\,'|, h)}{\partial x_\alpha}\, dx'^3 \end{aligned} \tag{2.9}$$

Bei der partiellen Integration wurde der Oberflächenterm wegen des kompakten Trägers gleich null. Die freie Integrationskonstante $g(\vec{r})$ wird durch Erhaltungsgrößen festgelegt. Bemerkenswert an der Gleichung ist noch, daß die Ortsableitung von der Funktion auf den Kernel übergeht, d.h. setzt man diese Gleichung in eine partielle DG ein, so wird eine gewöhnliche DG daraus. Weiter kann man zeigen, daß der Fehler beim „Smoothen“ von 2-ter Ordnung ist, d.h. $< \frac{\partial f(\vec{r})}{\partial x_\alpha} >= \frac{\partial f(\vec{r})}{\partial x_\alpha} + O(h^2)$ $(*)$.

Einsetzen der zweiten Ableitung von f in (2.6) ergibt:

$$\begin{aligned} < \frac{\partial^2 f(\vec{r})}{\partial x_\alpha \partial x_\beta} > \quad &= \quad \iiint_{\mathbb{R}^3} \frac{\partial^2 f(\vec{r}\,')}{\partial x'_\alpha \partial x'_\beta} W(|\vec{r} - \vec{r}\,'|, h)\, dx'^3 \\ &\overset{part.Int.}{=} \quad \iiint_{\mathbb{R}^3} \frac{\partial f(\vec{r}\,')}{\partial x'_\alpha} \frac{\partial W(|\vec{r} - \vec{r}\,'|, h)}{\partial x_\beta}\, dx'^3 \\ &\overset{(*)}{\approx} \quad \iiint_{\mathbb{R}^3} < \frac{\partial f(\vec{r}\,')}{\partial x'_\alpha} > \frac{\partial W(|\vec{r} - \vec{r}\,'|, h)}{\partial x_\beta}\, dx'^3 \\ &\overset{part.Int.}{=} \quad \iiint_{\mathbb{R}^3} \iiint_{\mathbb{R}^3} f(\vec{r}\,'') \frac{\partial W(|\vec{r}\,' - \vec{r}\,''|, h)}{\partial x'_\alpha} \frac{\partial W(|\vec{r} - \vec{r}\,'|, h)}{\partial x_\beta}\, dx''^3\, dx'^3 \end{aligned} \tag{2.10}$$

Da nur erste Ableitungen des Kerns auftreten, genügt die Forderung $W \in C^1_{\mathbb{R}^3}(\mathbb{R})$. Der Fehler ist wiederum von $O(h^2)$.

2.2.3 Die Diskretisierung

In einer weiteren Näherung wird das Integral von (2.6) durch eine Summe über die Teilchenorte $\vec{r}_i$ ersetzt:

$$f(\vec{r}_i) \approx \sum_j \frac{f(\vec{r}_j)}{n_j} W(|\vec{r}_i - \vec{r}_j|, h)$$

oder mit den Abkürzungen $n_j := n(\vec{r}_j)$ als Teilchendichte, $f_i := f(\vec{r}_i)$ und $W_{ij} := W(|\vec{r}_i - \vec{r}_j|, h)$:

$$f_i \approx \sum_j \frac{f_j}{n_j} W_{ij} \tag{2.11}$$

Der Diskretisierungsfehler den man hier macht ist umso kleiner, je mehr Teilchen sich in der Smoothing-Umgebung befinden.

Die gesmoothte Ortsableitung (2.9) sieht diskretisiert so aus:

$$\begin{aligned} \frac{\partial f_i}{\partial x_\alpha} &\approx \iiint_{\mathbb{R}^3} \big(f(\vec{r}\,') + g_i\big) \frac{\partial W(|\vec{r} - \vec{r}\,'|, h)}{\partial x_\alpha}\, dx'^3 \\ &\approx \sum_j \frac{f_j + g_i}{n_j} \frac{\partial W_{ij}}{\partial x_\alpha} \end{aligned} \tag{2.12}$$

2.3 Die SPH-Form der hydrodynamischen Gleichungen

2.3.1 Die SPH-Kontinuitätsgleichung

Die SPH-Form der Massendichte ρ_i ergibt sich nach (2.11) :

$$\rho_i = m_i n_i \overset{(2.11)}{=} \sum_j m_j W_{ij} \tag{2.13}$$

Die Lösung der Kontinuitätsgleichung ist nicht erforderlich um ρ_i zu erhalten, da W_{ij} ja schon bekannt ist.

2.3.2 Die SPH-Navier-Stokes-Gleichung

Zuerst werden die rechten Terme der ***Navier-Stokes-Gleichung*** (2.2) in die SPH-Form gebracht:

- Für den ***Druckterm*** ergibt sich

$$\frac{\partial p_i}{\partial x_\alpha} \overset{(2.12)(2.13)}{\approx} \sum_j m_j \frac{p_j + g_i}{\rho_j} \frac{\partial W_{ij}}{\partial x_\alpha} \tag{2.14}$$

  Mit der Forderung nach Impulserhaltung ergibt sich für die Integrationskonstante $g_i = p_i$. Dann ist auch der Drehimpuls erhalten, solange der viskose Term nicht hinzukommt. Die Implementierung des Druckterms in SPH++ wird auf Seite 35 beschrieben.

- Für den ***viskosen Term*** ergibt sich für $\zeta = 0$:

$$\frac{\partial T_{i_{\alpha\beta}}}{\partial x_\beta} \overset{(2.12)}{\approx} \sum_j m_j \frac{\eta_j \sigma_{j_{\alpha\beta}} + \eta_i \sigma_{i_{\alpha\beta}}}{\rho_j} \frac{\partial W_{ij}}{\partial x_\beta} \tag{2.15}$$

  mit folgender Abkürzung:

$$\begin{aligned} \sigma_{i_{\alpha\beta}} &:= V_{i_{\alpha\beta}} + V_{i_{\beta\alpha}} - \frac{2}{3}\delta_{\alpha\beta} V_{i_{\gamma\gamma}} \\ V_{i_{\alpha\beta}} &:= m_i \sum_j \frac{(\vec{v}_i - \vec{v}_j)_\alpha}{\rho_j} \frac{\partial W_{ij}}{\partial x_\beta} \end{aligned} \tag{2.16}$$

  Die Implementierung der physikalischen Viskosität in SPH++ wird auf Seite 37 beschrieben.

Aus (2.14) und (2.15) ergibt sich die ***SPH-Form der Navier-Stokes-Gleichung***:

$$\frac{dv_{i_\alpha}}{dt} = \sum_j m_j \frac{(-p_i \delta_{\alpha\beta} + \eta_i \sigma_{i_{\alpha\beta}}) + (-p_j \delta_{\alpha\beta} + \eta_j \sigma_{j_{\alpha\beta}})}{\rho_i \rho_j} \frac{\partial W_{ij}}{\partial x_\beta} + f_{i_\alpha} \tag{2.17}$$

2.3.3 Die SPH-Zustandsgleichung

Da in der polytropen Zustandsgleichung des idealen Gases (2.5) keine Ableitungen vorkommen, ändert sich die Gleichung beim Smoothen nicht.

2.4 Die künstliche Viskosität

Die künstliche Viskosität wurde eingeführt, um die numerische Auflösung von Schocks zu verbessern (siehe [Monaghan]). Sie kann als zusätzlicher Druckterm p_v in den hydrodynamischen Gleichungen aufgefaßt werden, d.h. p geht über in $p + p_v$, wobei zwei Formen für p_v üblich sind:

- Die ***künstliche Volumenviskosität***

$$p_v = \begin{cases} -\alpha \rho c_s l \nabla \vec{v} & : \quad \nabla \vec{v} < 0 \\ 0 & : \quad \nabla \vec{v} \geq 0 \end{cases} \tag{2.18}$$

- Die ***von Neumann-Richtmyer-Viskosität***

$$p_v = \begin{cases} \beta \rho l^2 (\nabla \vec{v})^2 & : \quad \nabla \vec{v} < 0 \\ 0 & : \quad \nabla \vec{v} \geq 0 \end{cases} \tag{2.19}$$

Dabei sind α und β frei wählbare Parameter, c_s ist die Schallgeschwindigkeit und l ist eine typische Länge des Systems (in SPH wird meist $l = h$ mit der Smoothing Length h gewählt). Die Bedingung $p_v = 0$ für $\nabla \vec{v} \geq 0$ bedeutet, daß die künstliche Viskosität nur bei Schocks wirksam sein soll.

Nach [Monaghan] ist es ungünstig die SPH-Form der Divergenz $\nabla \vec{v}$ direkt zu verwenden. Besser ist die Näherung:

$$\mu_{ij} := \frac{h(\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j)}{(\vec{r}_i - \vec{r}_j)^2 + \epsilon h^2} \tag{2.20}$$

Der Term ϵh^2 wurde eingefügt, um das Divergieren des Ausdrucks bei kleinen Teilchenabständen $|\vec{r}_i - \vec{r}_j|$ zu vermeiden.

Die SPH-Form der künstlichen Viskosität lautet dann

$$\Pi_{ij} := \begin{cases} \frac{-\alpha \bar{c}_{s_{ij}} \mu_{ij} + \beta \mu_{ij}^2}{\bar{\rho}_{ij}} & : \quad (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) < 0 \\ 0 & : \quad (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) \geq 0 \end{cases} \tag{2.21}$$

mit den Mittelwerten $\bar{c}_{s_{ij}} := \frac{c_{s_i} + c_{s_j}}{2}$ und $\bar{\rho_{ij}} := \frac{\rho_i + \rho_j}{2}$. Der Beitrag zur Navier-Stokes-Gleichung (2.17) hat also folgende Form:

$$\left. \frac{dv_{i_\alpha}}{dt} \right|_{art.visc.} = -\sum_j m_j \Pi_{ij} \delta_{\alpha\beta} \frac{\partial W_{ij}}{\partial x_\beta} \tag{2.22}$$

Die Implementierung der künstlichen Viskosität in SPH++ wird auf Seite 36 beschrieben. Zu bemerken ist noch, daß für den Limes $h \to 0$ die künstliche Viskosität im Gegensatz zur physikalischen Viskosität (2.15) verschwindet, d.h. sie beschreibt kein wirklich viskoses Fluid.

Kapitel 3

Anforderungen an SPH++

Am Anfang jeder Programm-Entwicklung steht die Frage, was das Programm einmal können soll. Diese Frage entspricht dem Arbeitsfluß „Bestimmungen der Anforderungen“ beim Unified Process (siehe Abb. C.5), der in der Etablierungsphase (siehe Abschnitt C.3.2) die wichtigste Rolle spielt. Im Falle von SPH++ sind die Anforderungen, wie in der Einleitung schon bemerkt, durch die Aufgabenstellung dieser Diplomarbeit schon festgelegt worden und sollen noch einmal hervorgehoben werden:

> ***SPH++ soll dieselbe Funktionalität wie das zugrundeliegende C-Programm besitzen und durch das objektorientierte Design besser zu erweitern und zu warten sein.***

Da das C-Programm also die Ausgangsbasis von SPH++ darstellt und auch die Funktionalität von SPH++ bestimmt, wird in Abschnitt 3.2 näher auf das C-Programm eingegangen. Zuvor wird jedoch in Abschnitt 3.1 die Hard- und Software-Umgebung bestimmt, auf der SPH++ entwickelt und ausgeführt werden soll. Im Unified Process entspricht dies der „Entwicklung des Domain- bzw. Business-Modells“. Der letzte Abschnitt 3.3 beschreibt die verschiedenen Programmier-Paradigmen, wobei besonders auf die Programmiersprachen C und C++ eingegangen wird.

3.1 Die Hard- und Softwareumgebung

Die Hardware, auf der SPH++ zum größten Teil entstand, war der private Rechner[1], da dieser um einiges schneller ist als die damaligen zur Verfügung stehenden Sun-Workstation[2] für Diplomanten.

Auf der Softwareseite wurde als Betriebssystem Solaris 7 bzw. Solaris 8[3] ausgewählt, da DTS (wird vom C-Programm benötigt, siehe Abschnitt 3.2) auf Solaris entwickelt wurde und deshalb auch dort am Besten angepaßt ist. Als Designtool wurde Together[4] eingesetzt, das laut einer Werbeanzeige ein „sourcecodebasierendes Werkzeug für objektorientierte Analyse, Design, Implementierung, Dokumentation und Test“ ist. Ein großer Vorteil gegenüber anderen Werkzeugen ist das sog. ***simultaneous round-trip engineering***, d.h. das Design-Modell und die Implementierung bleiben immer 100% synchron. Compiliert wurde mit dem GNU-Compiler gcc-2.95.2[5] mit der zusätzlichen GUI-Bibliothek qt-2.1.1[5].

[1] Intel Celeron(128k) mit 375Mhz bzw. 416Mhz.
[2] Sparc 10-Rechner.
[3] Ist (fast) kostenlos unter www.sun.com zu erwerben.
[4] In der Version 2.2 bis 4.1; siehe http://www.togethersoft.com.
[5] Auf der beiliegenden CD befinden sich der Source-Code im Verzeichnis /cdrom/data/src.

3.2 Das C-Programm

In diesem Abschnitt wird das C-Programm von Stefan Hüttemann und Stefan Kunze in der Version 1.1 vom 7.10.98 beschrieben, das sich auf der beiliegenden CD als orginale[6] und als fehlerbereinigte[7] Version befindet. Die Bugs des orginalen C-Programms wurden erst in der Testphase von SPH++ in Kapitel 5 gefunden, als die Ergebnisse beider Programme miteinander verglichen wurden.

3.2.1 Konfigurationsmöglichkeiten des C-Programms

Die Konfiguration des C-Programmes besteht aus zwei Teilen. Der erste Teil besteht aus dem Ein- bzw. Auskommentieren von Makros und anschließender Neukompilierung und der zweite Teil besteht aus dem Editieren des Konfigurationsfiles bzw. durch Parameterübergabe beim Start des Programmes. Diese zwei Teile sollen nun näher betrachtet werden.

3.2.1.1 Statische Konfigurierung durch Setzen von Makros

Die Konfiguration durch das Ein- bzw. Auskommentieren von Makros kann in den folgenden zwei Files durchgeführt werden:

- ***Makefile***: Im Makefile kann man die verschiedenen Terme der Navier-Stokes-Gleichung (siehe Kapitel 2) auswählen und entscheiden ob diese parallel oder nicht parallel ausgeführt werden sollen. Weiter kann die Genauigkeit auf „float“ oder auf “double“[8] gesetzt werden. Der für die Konfiguration wesentliche Teil des Makefiles hat folgende Gestalt:

```
## use float (1) or double (2) precision
SPH += -DSPH_PREC=1

##
## Die folgenden Flags erlauben es die verschiedenen Terme der DGL wegzulassen
##
## Verwenden von einfachem Runge-Kutta Integrator
#SPH += -DNO_ODEINT
## das Keplerpotential
#SPH += -DNO_KEPLERPOT
## Berechnung der SPH Massendichte (notwendig f"ur ViskTensor !)
#SPH += -DNO_MASSENDICHTE
### Viskositaetsterm berechnen
#SPH += -DNO_VISKTENSOR
## Berechnung der Wechselwirkungspaarliste
#SPH += -DNO_WWPAARLISTE
##
## switch parallel execution of the following functions: (default is parallel)
##
## MaxError()
#SPH += -DNOT_DIST_MAXERROR
## KeplerVel()
#SPH += -DNOT_DIST_KEPLERVEL
## KeplerPot()
#SPH += -DNOT_DIST_KEPLERPOT
## SteppingUp()
#SPH += -DNOT_DIST_STEPPINGUP
## Massendichte()
#SPH += -DNOT_DIST_MASSENDICHTE
## ViskTensor()
#SPH += -DNOT_DIST_VISKTENSOR
## CreateList(), WWPaarListe()
#SPH += -DNOT_DIST_CREATELIST
##
## to trace a particle (not implemented)
##
#SPH += -DKEPLERTRACE
##
## DEBUG FLAGS
##
## switch multi-threaded execution (default is multi-threaded)
```

[6] Siehe /cdrom/data/sph_src/sph.tar.gz.

[7] Siehe /cdrom/data/sph_src/sph_2000.tar.gz.

[8] Daß das orginale C-Programm einen Bug enthält, ergibt sich aus der Tatsache, daß das C-Programm mit der Einstellung „double“ auf Intel (nicht auf Sparc !) mit einer Fehlermeldung abbricht.

```
#SPH += -DNOT_DISTRIBUTED
## eneable debug messages
SPH += -DSPH_DEBUG
## enable timing of execution time of several functions
#SPH += -DSPH_TIMEING
```

- ***header.h***: Hier kann unter Anderem ausgewählt werden, mit welcher Dimension gerechnet wird und ob Teilchen eingefüttern werden sollen oder nicht. Der für die Konfiguration wichtige Auschnitt aus „header.h“ hat folgende Gestalt:

```
#define WITH_SECONDARY         /* true fuer binaersystem */

#define PRESSURE               /* with pressure forces */

#define ARTVISK                /* use artificial viscosity */

#define INSERT                 /* Teilchen koennen eingeuttert werden  */

#define DIMENSION  3      /* die Dimension des Ortsraums */
```

Nachdem die Makros ein- bzw. auskommentiert wurden, muß das C-Programm neu kompiliert werden. Jedoch ist das C-Programm noch nicht soweit ausgereift, daß es mit jeder angebotenen Konfigurationsmöglichkeit auch kompiliert werden kann. Allgemein benötigt man für die Kompilierung des C-Programms den Compiler „dtscc“, d.h. die Programme dts[9] und pvm[10] müssen installiert sein. Auf der beiliegenden CD sind die Sourcen dieser beiden Programme[11] enthalten. Wenn jedoch so konfiguriert wurde, daß keine Parallelisierung mehr vorliegt, kann als Compiler auch der „gcc“ genommen werden, d.h. die zwei Programme sind nicht mehr nötig.

3.2.1.2 Dynamische Konfigurierung durch Ändern des Konfigurationsfiles

Ein Teil der Konfiguration geschieht durch Editieren des Parameterfiles (siehe Abb. 5.2), das dem C-Programm beim Start mit übergeben wird. Einige Parameter können auch beim Start direkt angegeben werden, wie man im folgenden sieht:

```
> sph_main

3d Shared Memory Version 1.1 der SPH Ring Simulation
Neu: Mit Sekundaerstern, Berechnung erfolgt im mitrotierenden K.S.

usage: sph_main <Parameter-Datei> [Optionen]

  Die Optionen ueberschreiben die <Parameter-Datei>:

       [-N Teilchenzahl] : Anzahl der Teilchen
       [-G Grainsize]    : 'Koernigkeit der Parallelitaet'
       [-O Ausgabedatei]
       [-I Eingabedatei]
       [-h Smoothinglength]
       [-T DTS Thread Type]
       [-D Debugcategory.Debuglevel]

       *** Program Status:

       *** Using single precision (float)
       *** SPH_DEBUG   is set
       *** Debug Category set to 1000 level 10
       *** SPH_TIMEING is NOT set



> sph_main parameter.binary
```

3.2.2 Der Struktur des C-Programms

Wie in Abschnitt 3.3 noch dargelegt werden wird, sind bei einer prozeduralen Programmiersprache wie C die Datenstrukturen und die Algorithmen voneinander getrennt. Diese Trennung bestimmt auch die Struktur des C-Programms, die im folgenden beschrieben wird.

[9] dts steht für distributed thread system. Siehe http://www-ti.informatik.uni-tuebingen.de/dts.
[10] pvm steht für Parallele Virtuelle Maschine. Siehe http://www.epm.ornl.gov/pvm/pvm_home.html.
[11] Siehe /cdrom/data/src.

3.2.2.1 Die Datenstrukturen des C-Programms

In der Datei „header.h“ werden die im C-Programm benutzten Strukturen folgendermaßen definiert:

- ***„Parameter“***: Diese Struktur beschreibt das Physikalische System der Simulation. Zum Beispiel ist in dieser Struktur die Masse und der Ort der beteiligten Sterne und die Winkelgeschwindigkeit des Systems enthalten:

```
typedef struct _Parameter
{
#if defined(WITH_SECONDARY)
  SPH_REAL M1;                  /* Masse des Primaersterns */
  SPH_REAL M2;                  /* Masse des Sekundaersterns */
  SPH_REAL Porb;                /* Bahnperoide des Systems */
  SPH_REAL Omega;               /* Winkelgeschw. des Systems */
  Vector R1;                    /* Ort des Primaersterns */
  Vector R2;                    /* Ort des Primaersterns */
  SPH_REAL L1;                  /* X-Koordinate des inneren Lagrangepunktes */
  SPH_REAL Stardist;            /* Abstand der Sterne Voneinander */
#if defined(INSERT)
  signed int rate;            /* Teilchen/periode */
  /*  signed int atonce;         wie viele Teilchen auf einmal einfuettern*/
  SPH_REAL dt_insert;          /* Zeitintervall zwischen Einfuetterungen */
  SPH_REAL m; /* Masse der eingefuetterten Teilchen */
#endif
#else /* Single star */
  SPH_REAL M; /* Sonnenmasse */
  Vector R; /* Ort des Zentralgestirns */
#endif
  SPH_REAL nue; /* konstante Viskositaet */
#if defined(ARTVISK)
  SPH_REAL alpha;               /* Parameter der kuenstl. Viskositaet */
  SPH_REAL beta;                /* Parameter der kuenstl. Viskositaet */
#endif
  SPH_REAL h; /* smoothing length */
  Vector mingitter;             /* Untere Gittergrenzen */
  Vector maxgitter;             /* Obere Gittergrenzen */
  SPH_REAL min; /* kleinster Abst. vom Zentralgestirn */
  SPH_REAL max; /* groesster Abst. vom Zentralgestirn */
  signed int tanz; /* die aktuelle Teilchenanzahl */
  signed int maxtanz;         /* die maximale Teilchenanzahl */
} Parameter ;
```

- ***„WWPartner“***: Diese Struktur beschreibt einen Wechselwirkungspartner eines Teilchens. Wie man in der später beschriebenen Struktur „RK5Particle“ sehen kann, wird jedem RK5Particle eine Liste dieser WWPartner zugeordnet:

```
#define WWPARTNER 300 /* geschaetzte Zahl der WW Partner */
typedef struct _WWPartner
{
  signed int i; /* index des Partners in Teilchenliste */
  SPH_REAL q; /* Abstand der Zwei Teilchen */
  Vector dW;         /* Kernableitungen */
} WWPartner;
```

- ***„Particle“***: Diese Struktur enthält die Eigenschaften eines physikalischen Teilchens, wie z.B. dessen Ort und Geschwindigkeit. Für die Integration mit dem Runge-Kutta Integrator benötigt ein RK5Particle acht dieser Teilchen:

```
typedef struct _Particle
{
  Vector r; /* Ortskoordinaten */
  Vector v; /* Geschwindigkeitskoordinaten (dr/dt) */
  Vector b; /* Beschleunigungen (dv/dt) */
  SPH_REAL rho; /* die Dichte am Ort r  */
} Particle ;
```

- ***„RK5Particle“***: Diese Struktur ist das zentrale Teilchen für den Runge-Kutta Integrator. Es enthält z.B. für jeden Runge-Kutta Schritt ein Objekt des weiter oben eingeführten „Particle“ und weitere Variablen, in der Zwischenergebnisse und die berechneten Kräfte gespeichert sind:

```
#define RKMAXSTEPS 8 /* die Anzahl der RK Schritte + Hilfsvars. */
#define RKSTART 0 /* die Ausgangskoordinaten */
#define RKFIRST 1 /* der 1.Schritt */
#define RKSECND 2 /* der 2.Schritt */
#define RKTHIRD 3 /* der 3.Schritt */
#define RKFORTH 4 /* der 4.Schritt */
#define RKFIFTH 5 /* der 5.Schritt */
#define RKRESUL 6 /* das (vorlaeufige) Ergebniss */
#define RKERROR 7 /* zur Fehlerabschaetzung */

typedef struct _RK5Particle
{
#if defined(KEPLERTRACE)
  signed int trace;
#endif
  Particle rk[RKMAXSTEPS]; /* fuer jeden RK Schritt ein Teilchen ! */
  SPH_REAL m;                   /* Teilchenmasse */
  SPH_REAL T;
  SPH_REAL p;
  SPH_REAL e;
  SPH_REAL Tdsdt;
  /* aux. Variables */
  SPH_REAL sigmaxx;             /* Elemente des viskosen Spannungstensors */
  SPH_REAL sigmaxy;
  SPH_REAL sigmayy;
#if DIMENSION == 3
  SPH_REAL sigmaxz;
  SPH_REAL sigmayz;
  SPH_REAL sigmazz;
#endif
#if !defined(NO_KEPLERPOT)
  Vector dvdt_grav;             /* gravitationskraefte */
#endif
#if defined(PRESSURE)
  Vector dvdt_press;            /* die Druckkraefte   */
#endif
#if !defined(NO_VISKTENSOR)
  Vector dvdt_visc;             /* die viskose Kraft/Masse */
#endif
#if defined(ARTVISK)
  SPH_REAL cs;                  /* Schallgeschwindigkeit */
  Vector dvdt_artvisc;          /* Kraefte wg. kuenstl. Viskositat */
#endif
  signed int num_wwpart; /* Anzahl der WW Partner des Teilchens */

  WWPartner wwpl[WWPARTNER]; /* Liste der WW Partner */
} RK5Particle ;
```

- ***„RungeKutta“***: Diese Struktur enthält die Informationen für den Runge-Kutta Integrator, wie z.B. die gewünschte Genauigkeit und die Dauer der Integration.

```
typedef struct _RungeKutta
{
  SPH_REAL dt; /* Integrationschrittweite */
  SPH_REAL dtmin; /* untere Grenze fuer Schrittweite */
  SPH_REAL t; /* Integrationsvariable */
  SPH_REAL Ta; /* Integrationsanfang */
  SPH_REAL Te; /* Integrationsende */
#if defined(WITH_SECONDARY)
#if defined(INSERT)
  SPH_REAL t_next_insert;       /* Zeitpunkt des naechsten Einfuetterns */
#endif
#endif
  signed int infile_number;
  signed int num_dumps; /* number of dumps in integration loop */
  SPH_REAL
  dump_time[RK_MAX_DUMP_TIMES]; /* list of dump times */
  signed int TSteps; /* (max.) Integrationsschritte */
  SPH_REAL eps; /* Integrationsgenauigkeit */

} RungeKutta ;
```

3.2.2.2 Die Algorithmen des C-Programms

Die Algorithmen bzw. Funktionen des C-Programms sind in Abbildung 3.1 als Rechtecke abgebildet. Die Pfeile, die von einem Rechteck ausgehen, zeigen die Funktionsaufrufe an, wobei die Numerierung der Pfeile deren Reihenfolge kennzeichnen. Zum Beispiel ruft die Funtion „main“

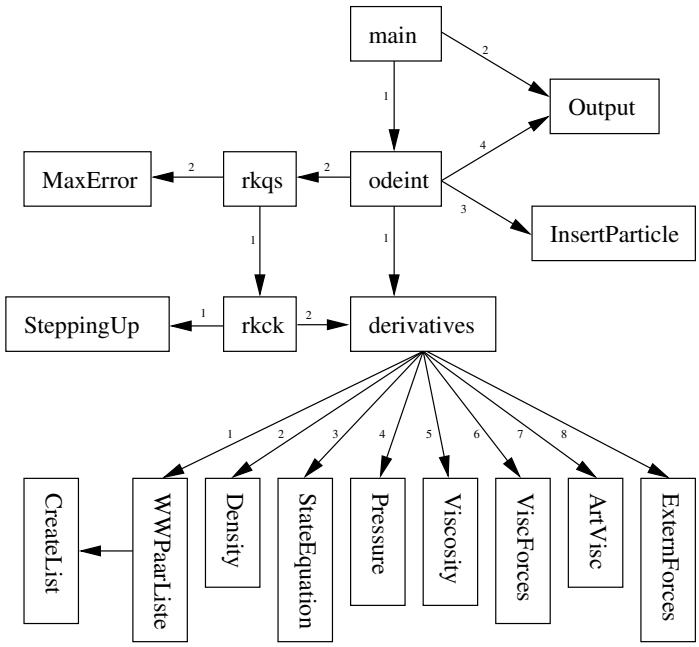


Abbildung 3.1: Struktur des C-Programms: Pfeile sind Funktionsaufrufe, Numerierungen kennzeichnen die Reihenfolge der Funktionsaufrufe.

zuerst den Integrator „odeint“ auf und erst danach die Funktion „Output“, die das Ergebnis in ein File schreibt. Die einzelnen Funktionen werden im folgenden kurz beschrieben:

- ***„main“***: Die „main“-Funktion startet die Simulation. Dabei wird zuerst die Parameterdatei ausgelesen bzw. die Übergabeparameter aus der Kommandozeile übernommen. Mit Hilfe dieser Daten werden die Variablen der Datenstrukturen deklariert und initialisiert. Falls in der Konfiguration des C-Programmes die Parallelisierung eingeschaltet wurde, wird dananch das DTS initialisiert. Anschließend wird der Integrator „odeint“ aufgerufen und das Resultat in der Funktion „Output“ ausgegeben.

- ***„odeint“***: „odeint“ ist eine Driver-Funktion für ein Runge-Kutta-Verfahren 5-ter Ordnung mit adaptiver Schrittweitensteuerung. Aus den Anfangswerten (Ort, Geschwindigkeit) werden die Orte und Geschwindigkeiten zu einem späteren Zeitpunkt errechnet. Siehe auch [Numerical Recipes] auf Seite 721.

- ***„rkqs“***: „rkqs“ ist die Abkürzung von „Runge-Kutta quality stepper“ und ist der Stepper des Runge-Kutta-Verfahrens. Wie der Name schon andeutet, überwacht der Stepper den Integrationsfehler bei einem Integrationsschritt. Weicht dieser Wert von einem vorgegebenem Referenzwert ab, so wird der Integrationsschritt mit einer korrigierten Schrittweite wiederholt. Siehe auch [Numerical Recipes] auf Seite 719.

- ***„rkck“***: „rkck“ ist die Abkürzung von „Runge-Kutta Cash-Karp“ und wird als Runge-Kutta-Algorithmus bezeichnet. Er berechnet aus den Anfangswerten die Werte nach einem Integrationsschritt und gibt zusätzlich den dabei gemachten Fehler zurück. Siehe auch [Numerical Recipes] auf Seite 719.

- ***„MaxError“***: Aus dem zurückgegeben Fehler des Runga-Kutta Algorithmus „rkck“ wird der Fehler des Integrationsschittes berechnet. Wie diese Berechnung aussieht, hängt von dem zu integrierenden Problem ab. Dieser Fehler wird im Stepper dann für die Qualität des Integrationsschrittes benutzt.

- ***„SteppingUp“***: Dieser Algorithmus ist eine Hilfsfunktion für den Runga-Kutta Algorithmus „rkck“. Hier werden die eigentlichen Runge-Kutta Schritte durchgeführt.
- ***„Output“***: Diese Funktion schreibt die momentanen Werte wie Ort, Geschwindigkeit, Dichte, u.s.w. der Teilchen in ein File.
- ***„InsertParticle“***: Diese Funktion füttert ein Teilchen am Ort L1 ein. Der Ort und die Geschwindigkeit des eingefütterten Teilchens werden dabei mit Hilfe eines Zufallsgenerators berechnet.
- ***„derivatives“***: Hier wird die DGL 1.Ordnung in den Orts- und Impulskoordinaten berechnet. In diesem speziellen Fall wird angenommen, daß auf der rechten Seite der DGL nur eine Funktion der abhängigen Variablen steht. Eventuell auftretende partielle Gleichungen werden mit Hilfe der SPH-Methode berechnet. Die Numerierung der Funktionen in Abb. 3.1 beschreibt die Reihenfolge, in der die Funktionen aufgerufen werden. Diese Reihenfolge darf im allgemeinen Fall nicht verändert werden, da die Funktionen voneinander abhängig sind:
  - ***„WWPaarliste“***: Hier werden alle Wechselwirkungspaare nach dem Verfahren von O.Flebbe (siehe [Flebbe]) bestimmt.
  - ***„CreateList“***: Dies ist eine Hilfsfunktion der „WWPaarliste“.
  - ***„Density“***: Die Massendichte (siehe Gl. 2.13) jedes Teilchens wird berechnet
  - ***„StateEquation“***:Hier wird die polytrope Zustandsgleichung (siehe Gl. 2.5) und die Temperatur aus der idealen Gasgleichung berechnet.
  - ***„Pressure“***: Die Druckkräfte (siehe Gl. 2.14), die auf jedes Teilchen wirken, werden berechnet.
  - ***„Viscosity“***: Der viskose Spannungstensor (siehe Gl. 2.16), der für die Berechnung der physikalische Viskosität benötigt wird, wird berechnet.
  - ***„ViscForces“***: Die auf jedes Teilchen wirkende physikalische Viskositätskraft (siehe Gl. 2.15) wird berechnet.
  - ***„ArtVisc“***: Die auf jedes Teilchen wirkende künstliche Viskosität (siehe Gl. 2.22) wird berechnet.
  - ***„ExternForces“***: Die Gravitationskraft, die auf jedes Teilchen wirkt, wird berechnet.

3.3 Programmier-Paradigmen

Dieser Abschnitt beschreibt die verschiedenen Programmier-Paradigmen, die sich im Laufe der Zeit herausgebildet haben. Besonders die Beziehung dieser Paradigmen zu dem im vorigen Abschnitt behandelten C-Programm und zu SPH++ wird behandelt.

Allgemein kann man ein Programm in zwei Hauptbestandteile zerlegen:

- Einer Menge von ***Algorithmen*** (deren Aufgabe es ist ein bestimmtes Problem zu lösen).
- Einer Menge von ***Daten***, auf die die Algorithmen zugreifen um eine eindeutige Lösung des Problems zu erhalten.

Die Wechselwirkung zwischen diesen zwei Hauptbestandteilen eines Programms bezeichnet man als ***Programmier-Paradigma*** (engl. ***programming paradigm***). Folgende drei Paradigmen entwickelten sich im Laufe der Zeit:

- ***Prozedurales Programmier-Paradigma*** (engl. ***procedural programming paradigm***): Hier wird ein Problem direkt in eine Folge von Algorithmen zerlegt. Die Daten werden separat gespeichert und entweder global oder als Argumente den Prozeduren übergeben. Drei bekannte Prozedurale Sprachen sind FORTRAN, C und PASCAL. Auch C++ unterstützt dieses Paradigma, da C++ eine Obermenge von C ist.

- ***Objekt-basiertes Programmieren*** (engl. ***object based programming***): Der zentrale Begriff bei diesem Paradigma ist das des ***abstrakten Datentyps***, der in C++ der ***Klasse*** entspricht. Ein Problem wird hier in eine Menge von Klassen zerlegt. Eine Klasse enthält eine Menge von Algorithmen, die als (public) ***Interface*** bezeichnet wird und Daten, die versteckt vom übrigen Programm sind (private). Das dynamische Verhalten eines Programms besteht nun in der Wechselwirkung der Objekte dieser Klassen untereinander. Programmiersprachen, die dieses Paradigma unterstützen sind CLU, Ada, Modula2 und C++.
- ***Objekt-orientiertes Programmieren*** (engl. ***object-oriented programming, OOP***): OOP erweitert die abstrakten Datentypen durch den ***Vererbungs***mechanismus (engl. ***inheritence***) und durch ***Polymorphie***. Durch Vererbung kann das Gemeinsame einer Menge von Klassen in eine abstrakte Oberklasse verschoben werden, wodurch die abgeleiteten Klassen nur noch Ihre speziellen Eigenschaften enthalten. Um polymorphes Verhalten in C++ zu erzielen, müssen die aufgerufenen Funktionen virtuell sein und die Objekte müssen über Zeiger oder Referenzen manipuliert werden. Bekannte objektorientierte Programmiersprachen sind Simula, Smalltalk, Java und C++.

C++ ist also eine Multiparadigmen Sprache. Obwohl man bei C++ zuerst an eine OOP-Sprache denkt, unterstützt es auch die zwei anderen Paradigmen. Der Vorteil ist, daß man das geeignetste Paradigma für sein Problem auswählen kann, wodurch C++ aber auch größer und komplizierter als z.B. Java ist.

Das im vorigen Abschnitt beschriebene C-Programm ist nach dem Prozeduralen Programmier-Paradigma angeordnet. Das hat zur Folge, daß alles mit allem irgendwie zusammenhängt und es deshalb für das Verständnis des Programms notwendig ist, es Zeile für Zeile durchzuarbeiten. Das dies sehr zeitaufwendig ist, kann man aus dem vorigen Abschnitt erahnen. Auch folgt aus der prozeduralen Programmierung eine schlechte Erweiterbarkeit und Wartbarkeit des C-Programms, da die Übersichtlichkeit bei größer werdendem C-Programm immer mehr abnimmt.

Ein weiterer Nachteil des C-Programmes ist die umständliche Konfiguration der Simulation. Zuerst muß man die gewünschten Makros in den diversen Files (hauptsächlich im Makefile) ein- bzw. auskommentieren und anschließend das komplette Programm neu kompilieren. Dies kostet nicht nur Zeit, sondern der Benutzer muß sich auch zuerst in die Konfigurationsfiles einarbeiten und benötigt zusätzlich für das Kompilieren eine entsprechende Entwicklerumgebung.

Diese Nachteile versucht man mit der Objekt-orientierten Programmierung, wo man definierte Schnittstellen hat, und sog. Design-Pattern zu umgehen. Man kann Design-Pattern natürlich auch in C einsetzen, aber dies erfordert einen viel größeren Aufwand, da C im Gegensatz zu einer objektorientierten Sprache dafür keine direkte Unterstützung bietet. Durch Design-Pattern wird eine zusätzliche Abstraktionsebene hinzugefügt, die den Vorteil hat, daß man die Struktur eines Programmes besser überblicken kann, ohne die darunterliegende Implementierung zu kennen. Darüberhinaus gibt es keine Trennung der Algorithmen mehr von den dazugehörigen Daten, das zur Folge hat, daß die einzelnen Teile des Programms unabhängiger voneinander sind. Diese Vorgehensweise verbessert also die Erweiterbarkeit und Wartbarkeit von Programmen. Auch die Konfiguration in einem Objekt-orientierten Programm ist einfacher als im C-Programm, da sie in der Regel zur Laufzeit durchgeführt werden kann.

Im folgenden Kapitel wird nun die Entwicklung von SPH++ beschrieben, das die oben beschriebenen Vorteile eines Objekt-orientierten Programms besitzen soll. Diese Vorteile kann man auch als weitere Anforderungen an SPH++ ansehen.

Kapitel 4

Entwurf und Implementierung von SPH++

Nachdem im vorigen Kapitel die Anforderungen an SPH++ beschrieben wurden, wird in diesem Kapitel gezeigt, wie darauf aufbauend die Architektur von SPH++ entwickelt und implementiert wurde. Dieser Teil entspricht der „Entwurfs- und Konstruktionsphase“ des Unified-Process (siehe Abb. C.5). Entwickelt wurde dabei „iterativ und inkrementell“ (siehe Abschnitt C.2.3), das einer der Hauptpfeiler des Unified Process ist. Diese vielen Iterationen wurden in drei Phasen zusammengefaßt. In der ersten Phase wird versucht, die gewünschten Anforderungen an das Programm in Klassendiagramme umzusetzen. Die zweite Phase beinhaltet die erste Implementierung mit nur einer Kraft und in der dritten Phase wird das Programm um SPH-Kräfte erweitert.

Bei der Entwicklung der Architektur wurde auf zwei Punkte besonderen Wert gelegt:

- Die Struktur des Simulationsmodells soll so genau wie möglich auf das Programm übertragen werden, was ja in einer objektorientierten Sprache sehr gut möglich ist.
- Das Programm soll gut erweiterbar und gut zu warten sein, d.h. es sollen Design-Pattern eingesetzt werden.

4.1 Phase 1: Erstes objektorientiertes Design mit Design-Pattern

Zuerst wird das Programm in einzelne Aufgabengebiete zerlegt (siehe Abb. 4.1).

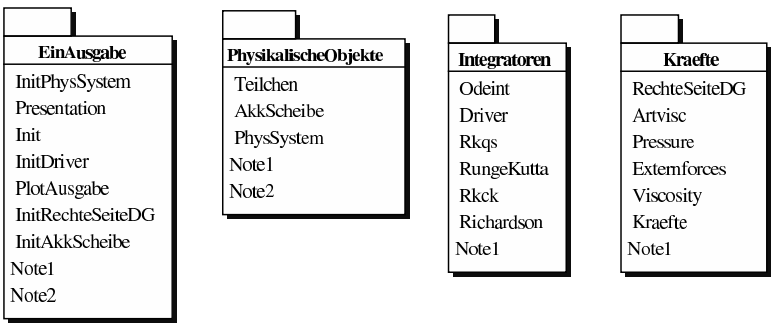


Abbildung 4.1: Die Zerlegung des Programms in einzelne Pakete in der Phase 1.

Jedes Aufgabengebiet wird durch ein Paket (siehe Abb. A.7) dargestellt, das einem Verzeichnis unter Unix entspricht. In den folgenden Abschnitten werden die einzelnen Pakete näher erläutert:

4.1.1 Paket: „PhysikalischeObjekte“

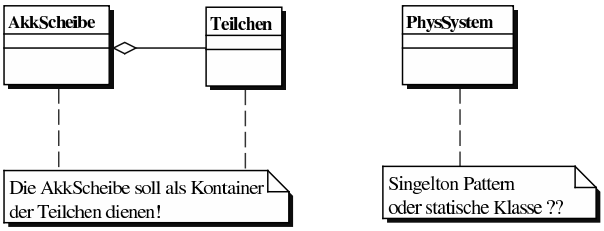


Abbildung 4.2: Das Paket der physikalischen Objekte in der Phase 1.

Das physikalische System im Paket „PhysikalischeObjekte“ (siehe Abb. 4.2) wird nach ihrer Zeit- bzw. nicht Zeitabhängigkeit aufgeteilt:

- Die von der ***Zeit unabhängigen*** Parameter werden in der Klasse „PhysSystem“ zusammengefaßt. Da es nur ***ein*** instanziiertes Objekt dieser Klasse geben kann, soll diese Klasse als Singelton Pattern (siehe Abschnitt B.2.4) realisiert werden.

- Das wichtigste ***zeitabhängige*** Objekt ist die Akkretionsscheibe. Sie ist zu Beginn der Simulation gegeben und wird durch den Integrator zu späteren Zeitpunkten neu berechnet. Die Akkretionsscheibe wird durch zwei Klassen realisiert:

  – Die Klasse „AkkScheibe“ ist die Containerklasse der Teilchen und enthält z.B. die Eigenschaft der Erzeugung und Vernichtung von Teilchen und die Zugriffsmöglichkeit auf beliebige einzelne Teilchen.

  – Die Klasse „Teilchen“ enthält die Eigenschaften eines Simulationsteilchens wie z.B. Ort, Geschwindigkeit, u.s.w. .

4.1.2 Paket: „Kraefte“

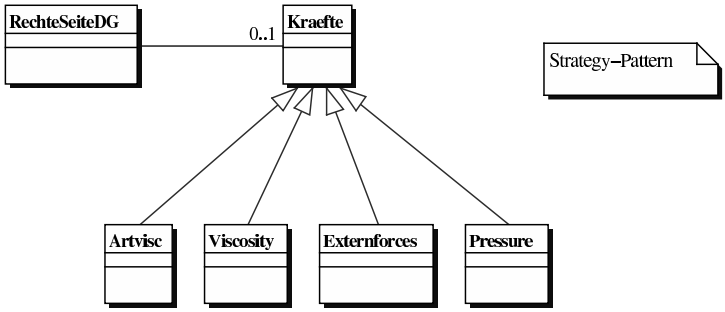


Abbildung 4.3: Das Paket der Kräfte in der Phase 1.

Die Kräfte in diesem Paket (siehe Abb. 4.3) werden nach dem Strategy-Pattern (Abschnitt B.2.2) angeordnet.

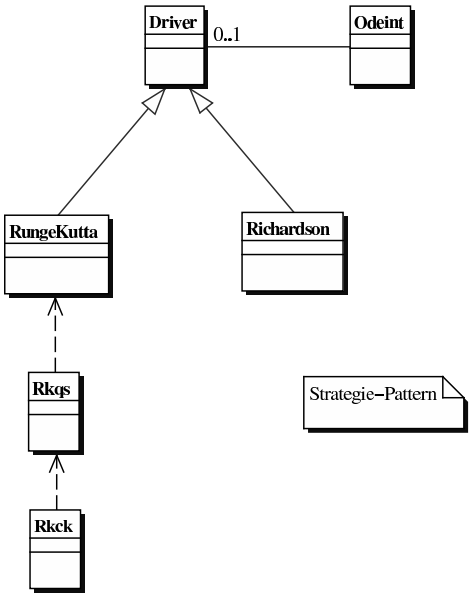


Abbildung 4.4: Das Paket der Integratoren in der Phase 1.

4.1.3 Paket: „Integratoren“

Wenn mehrere alternative Algorithmen zur Auswahl stehen, werden diese am besten durch das Strategy-Pattern (siehe Abschnitt B.2.2) angeordnet. Dieses Pattern wurde deshalb auch im Paket „Integratoren“ (siehe Abb. 4.4) gewählt. Die alternativen Algorithmen sind in diesem Fall der Runge-Kutta bzw. der Richardson Integrator. Der Vorteil dieses Pattern ist wiederum die leichte Erweiterbarkeit mit zusätzlichen Integratoren und die Wahl des gewünschten Integrators zur Laufzeit.

4.1.4 Paket: „EinAusgabe“

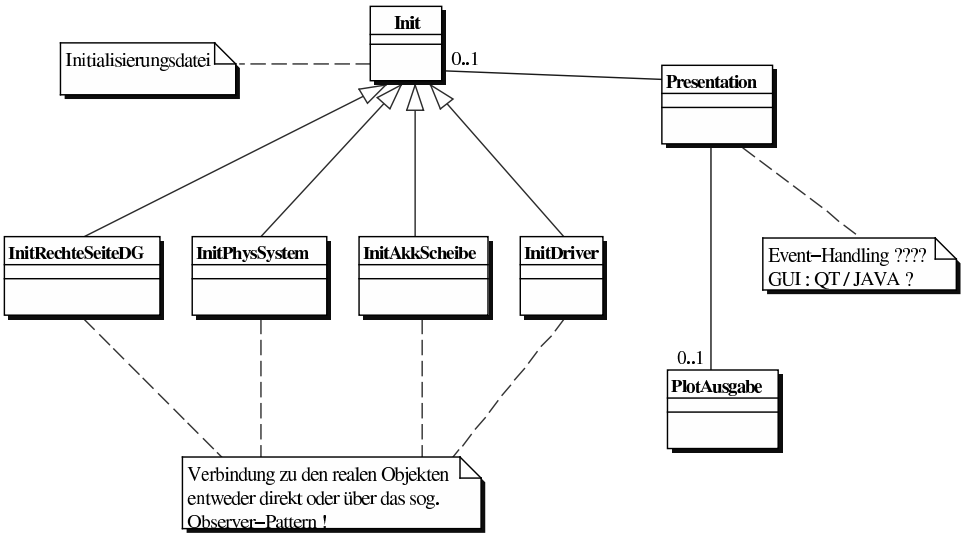


Abbildung 4.5: Das Paket der Ein-/Ausgabe in der Phase 1.

Das „EinAusgabe“-Paket (siehe Abb. 4.5) besteht aus zwei Teilen:

- Die Konfigurationsdaten sollen durch die Objekte der von der Init-Klasse abgeleiteten Klassen verwaltet werden. Diese Objekte werden durch eine „Initialisierungsdatei“ initialisiert und können danach interaktiv durch die „Presentation“-Klasse verändert werden. Diese Initialisierungsdatei kann entweder direkt vor der Laufzeit des Programms oder über ein Objekt der „Presentation“-Klasse während der Laufzeit verändert werden. Die Kommunikation der Initialisierungsobjekte mit der eigentlichen Simulation geschieht entweder interaktiv durch das Observer-Pattern (siehe Abschnitt B.2.5) oder beim Start der Simulation durch direkte Übergabe der Initialisierungsobjekte. Ein Vorteil des Observer-Patterns ist die lose Kopplung zwischen den Initialisierungsklassen auf der einen Seite und den Simulationsklassen auf der anderen Seite.

- Die grafische Benutzer-Schnittstelle (GUI) wird durch die „Presentation“-Klasse repräsentiert. Diese kann beispielsweise in Java oder mit Hilfe der Qt[1]-Bibliothek realisiert werden. Die PlotAusgabe-Klasse ist ein Beispiel der zusätzlichen Möglichkeiten der GUI. In der „Presentation“-Klasse sollen alle Parameter der „Initialisierungsdatei“ graphisch ausgewählt werden, z.B. die Auswahl der gewünschten Kräfte bzw. des Kernels oder die Genauigkeit und Dauer der Integration. Auch soll der Start von dort aus erfolgen und die Statusmeldungen und die Zwischenergebnisse während der Simulation von dort graphisch kontrolliert werden!

4.2 Phase 2: Implementierung ohne SPH-Kräfte

Das Ziel dieser Phase ist die Implementierung des in Phase 1 erstellten Designs. Die Vorgehensweise der Implementierung ist dabei evolutionär, d.h. zuerst wird nur das Grundgerüst implementiert und ausgetestet, bevor weitere Funktionalität eingebaut wird. In dieser Phase ändert sich gegenüber der ersten Phase folgendes:

- Es gibt ein neues Paket „Simulationen“, das im Abschnitt 4.2.5 beschrieben wird.

- Die Verzeichnisstruktur des Programms ist in Abbildung 4.6 abgebildet. Im Verzeichnis „lib“ ist die SPH-Library „libsph.a“ nach dem Kompilieren enthalten. Zusammen mit den im Verzeichnis „include“ enhaltenen Include-Files können externe Programme auf die SPH-Library zugreifen. Beispiele für externe Programme sind im Verzeichnis „examples“ enthalten und im Verzeichnis „src“ sind die Programmpakete enthalten.

- Die Verwaltung des Kompilierens und die Erstellung der Library übernehmen Makefiles. Um das Programm zu kompilieren muß daher nur „make“ im obersten Verzeichnis eingegeben werden bzw. „make clean“ um die ganze Sache rückgängig zu machen.

Nun werden die einzelnen Pakete dieser Phase beschrieben:

4.2.1 Paket: „PhysikalischeObjekte“

Die Aufteilung des Pakets „PhysikalischeObjekte“ (siehe Abb. 4.7) in zwei Teile hat sich gegenüber der Abbildung 4.2 der ersten Phase nicht geändert:

- In der Klasse „PhysSystem“ sind die ***zeitunabhängigen*** Parameter wie z.B. _M1, _M2 und _L1 (Masse der zwei Sterne und der erste Lagrangepunkt) enthalten, die als „private“ Membervariable implementiert sind. Auf jeden Parameter kann man durch die entsprechenden Zugriffsfunktionen wie z.B. M1(), M2() und L1() zugreifen, die als „public“ Memberfunktionen implementiert sind. Die Initialisierung eines Objekts der Klasse erfolgt durch ein Objekt der Klasse „InitPhysSystem“ aus dem „EinAusgabe“-Paket (siehe Abbildung 4.10), deren Objekt dem Konstruktor bei der Initialisierung übergeben wird.

[1] Siehe http://www.troll.no.

Abbildung 4.6: Die Verzeichnisstruktur des Programms in der Phase 2.

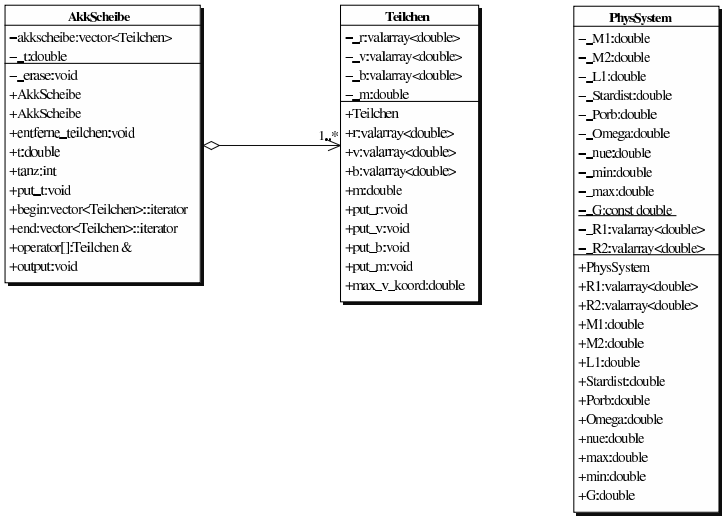


Abbildung 4.7: Das Paket der Physikalischen Objekte in der Phase 2.

- Der ***zeitabhängige*** Teil ist wie in Phase 1 die Akkretionsscheibe, die sich aus den Klassen „AkkScheibe“ und „Teilchen“ zusammensetzt:
  - Die Kontainerklasse „AkkScheibe“ enthält Objekte der Klasse „Teilchen“. Die Implementierung dieser Kontainerklasse geschieht mittels der STL[2]-Klasse „***vector***“. Da die „AkkScheibe“ zeitabhängig ist, enthält sie noch die „private“ Membervariable _t, die der Zeit entspricht. Der Zugriff auf Teilchen der „AkkScheibe“ erfolgt wie bei einem Array über die Funktion „operator[ ]“. In diese Klasse gehören auch solche Funktionen, die für die Verwaltung der Teilchen zuständig sind, z.B. die Erzeugung und Vernichtung von Teilchen. Die Initialisierung eines Objektes der Klasse erfolgt durch ein Objekt der Klasse „InitAkkScheibe“ aus dem „EinAusgabe“-Paket (siehe Abbildung 4.10), das dem Konstruktor bei der Initialisierung übergeben wird.
  - Die Klasse „Teilchen“ enhält die „privaten“ Membervariablen _r, _v, _b, _m (Ort, Geschwindigkeit, Beschleunigung, Masse), die den Zustand eines Teilchens vollständig beschreiben.

4.2.2 Paket: „Kraefte“

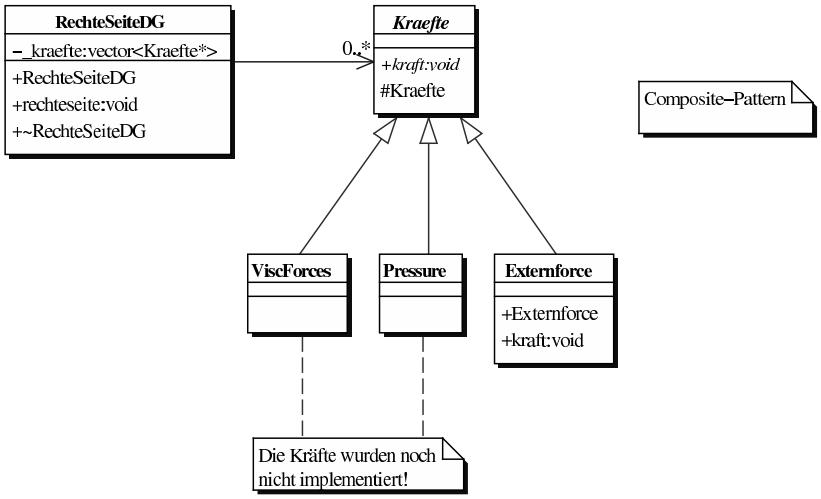


Abbildung 4.8: Das Paket der Kräfte in der Phase 2.

Als einzige Kraft wird in dieser Phase die Gravitationskraft als Klasse „Externforce“ im Paket „Kraefte“ (siehe Abb. 4.8) implementiert. Die Initialisierung eines Objekts der Klasse „Externforce“ erfolgt durch ein Objekt der Klasse „InitRechteSeiteDG“ aus dem „EinAusgabe“-Paket (siehe Abb. 4.10).

4.2.3 Paket: „Integratoren“

Die Struktur des Pakets „Integratoren“ (siehe Abb. 4.9) ist wie in der ersten Phase das Strategy-Pattern. Hier ist jedoch nur ein Integrator implementiert worden und zwar ein Runge-Kutta-Integrator 5-ter Ordnung mit adaptiver Schrittweitenkontrolle (siehe [Numerical Recipes] S.714). Die Initialisierung eines Objektes der Klasse „Odeint“ erfolgt durch ein Objekt der Klasse „InitOdeint“ aus dem „EinAusgabe“-Paket (siehe Abbildung 4.10), das dem Konstruktor bei der Initialisierung übergeben wird.

[2] Abkürzung für Standard Template Library.

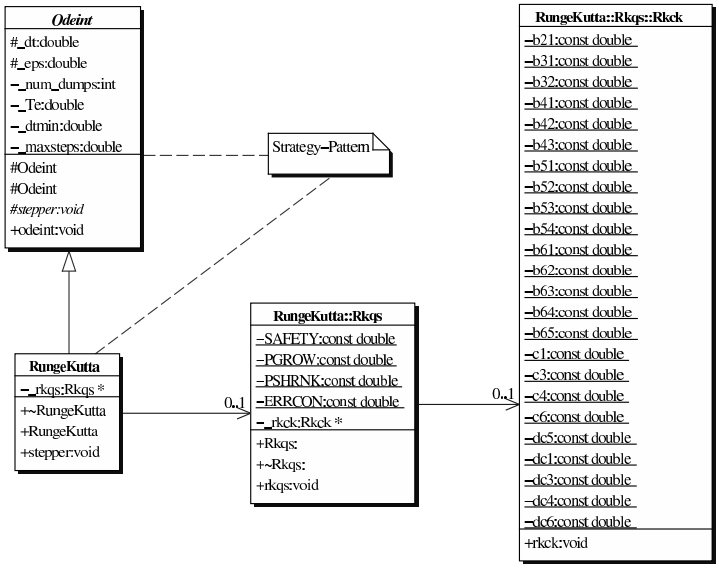


Abbildung 4.9: Das Paket der Integratoren in der Phase 2.

4.2.4 Paket: „EinAusgabe“

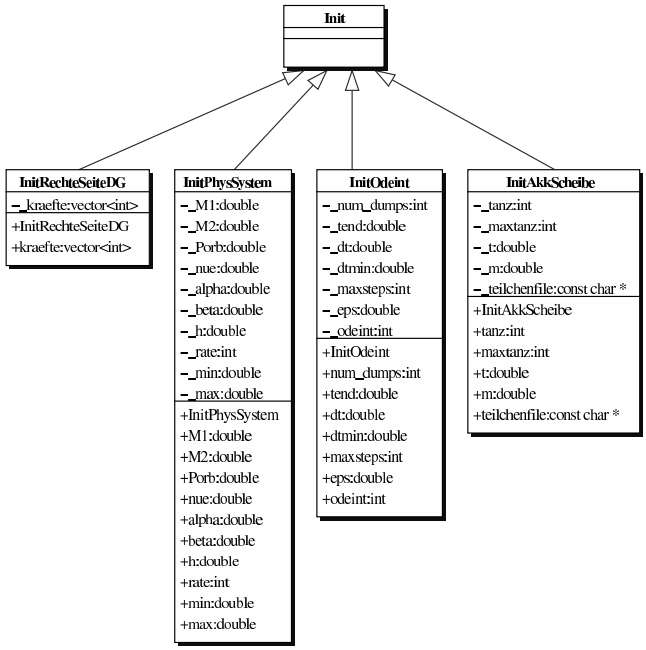


Abbildung 4.10: Das Paket der Ein-/Ausgabe in der Phase 2.

In dem Paket „EinAusgabe“ (siehe Abb. 4.10) wird nur ein Teil der ersten Phase implementiert. Das bedeutet, daß die Initialisierung der Objekte (und dadurch auch des ganzen Programms) vorläufig nur aus einem Initialisierungsfile erfolgt und (noch) nicht mit Hilfe einer GUI. Dieses Initialisierungsfile (oder genauer dessen Filenamen) wird als Parameter den Konstruktoren der einzelnen Klassen übergeben.

4.2.5 Paket: „Simulationen“

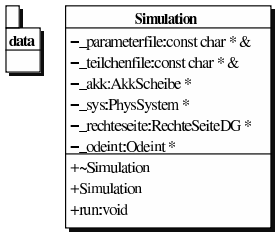


Abbildung 4.11: Das Paket der Simulationen in der Phase 2.

Das Paket „Simulationen“ (siehe Abb. 4.11) enthält nur eine Klasse. Diese hat die Aufgabe einer „main“-Funktion, d.h. sie steuert den Ablauf des Programms. Dazu wird in einem Objekt dieser Klasse zuerst Objekte der Klassen „AkkScheibe“, „PhysSystem“, „Odeint“ und der zugehörigen Initialisierungsklassen instanziiert. Mit diesen Objekten wird dann die Simulation gestartet.

4.3 Phase 3: Re-Design, Implementierung mit SPH-Kräften

Das Ziel dieser Phase ist die noch fehlende Implementierung der SPH-Kräfte. Dies erfordert jedoch eine Neustrukturierung einiger Teile des Programms, die im folgenden beschrieben wird. Zusätzlich wird in dieser Phase im Paket „Simulationen“ eine GUI mit Hilfe der Qt-Bibliothek entwickelt. Ein Beispielprogramm mit dieser GUI findet man im neuen Verzeichnis „examples/testqt“.

Für die Berechnung der SPH-Kräfte benötigt man für jedes Teilchen seine Wechselwirkungspartner. Weiterhin müssen die mit diesen Wechselwirkungsteilchen berechneten SPH-Kräfte bzw. deren Zwischenergebnisse irgendwo gespeichert werden. Im folgenden Abschnitt sollen diese grundlegenden Dinge näher beleuchtet werden. Die daran folgenden Abschnitte beschreiben dann die einzelnen Pakete.

4.3.1 Wesentliche Designänderungen

Die Kräfte die auf ein Teilchen wirken, müssen bei jedem Integrationsschritt in Abhängigkeit vom verwendeten Integrator mehrmals berechnet werden (Beim Runge-Kutta-Verfahren 5-ter Ordnung mit adaptiver Schrittweitensteuerung insgesamt sechs mal!). Das bedeutet natürlich auch, daß die Wechselwirkungsteilchen bei jedem Integrationsschritt ***mehrmals*** neu berechnet werden müssen. Da diese Berechnung also häufig auftritt, muß versucht werden, diese Wechselwirkungssuche möglichst effektiv zu implementieren, was in den nächsten Unterabschnitten beschrieben wird. Danach folgt eine grundsätzliche Diskussion über das Designproblem der SPH-Kräfte.

4.3.1.1 Wechselwirkungsobjekte

Bei dem C-Programm wird jedem „RK5Particle“ eine Wechselwirkungs-Partner-Liste „WWPartner wwpl“ zugeordnet (Abschnitt 3.2.2.1). Dadurch werden Wechselwirkungsterme, die nur vom Teilchenabstand $|i-j|$ abhängen, doppelt berechnet, da diese Terme bei der Berechnung der Wechselwirkung für das Teilchen i ***und*** für das Teilchen j benötigt werden.

Eine alternative Methode die SPH-Kräfte zu berechnen, ohne der doppelten Berechnung von symmetrischen Termen, ist die Berechnung mit Hilfe von Wechselwirkungsobjekten. Diese wird im folgenden näher erläutert:

Die Wechselwirkungskraft zwischen zwei Teilchen ist ein Vektor. Graphisch wird das in Abbildung 4.12 dargestellt. Wie in der Einleitung zu diesem Kapitel schon angemerkt wurde, versucht

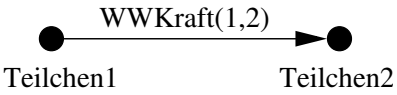


Abbildung 4.12: Die Wechselwirkungskraft zwischen zwei Teilchen.

man so gut wie möglich, die Struktur des Simulationsmodells auf das Programm zu übertragen. In diesem Fall versucht man also die Wechselwirkungskraft, die ein Vektor ist, als eigenständiges Objekt zu implementieren (siehe Abb. 4.13).

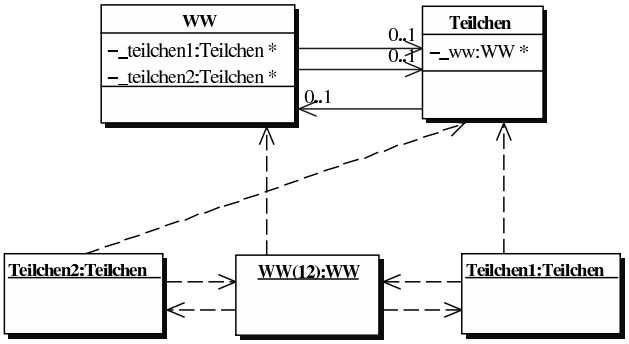


Abbildung 4.13: Ein Wechselwirkungsobjekt in UML ausgedrückt.

4.3.1.2 Vor- und Nachteile von Wechselwirkungsobjekte gegenüber der Verwendung von Teilchenlisten

Mit Hilfe von Abb. 4.14 soll anhand von fünf Teilchen die zwei Verfahren zur Berechnung der Wechselwirkungskräfte gegenübergestellt werden:

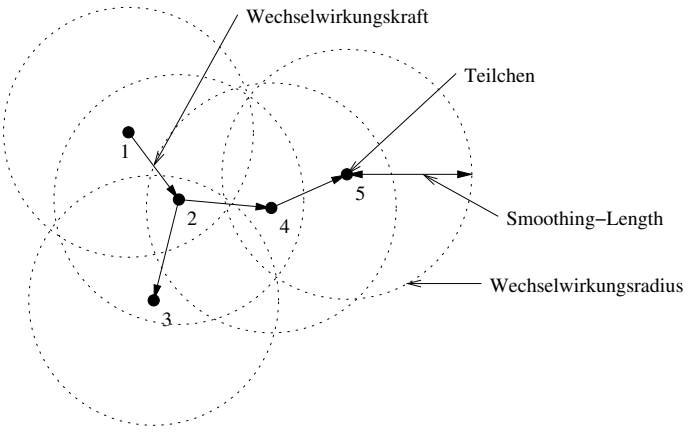


Abbildung 4.14: Die Wechselwirkungsreichweiten von 5 Teilchen.

- ***Die Berechnung der SPH-Kräfte mit Hilfe von Teilchenlisten*** (siehe Abb. 4.15).
  Jedes Teilchen bekommt eine Liste von Wechselwirkungsteilchen zugeordnet. Diese wird am

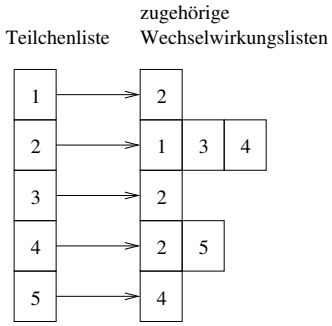


Abbildung 4.15: Berechnung der SPH-Kräfte mit Hilfe von Teilchenliste.

Besten durch einen „STL-Vektor“ von Pointern, die auf die Wechselwirkungsteilchen zeigen, implementiert. Eine andere Möglichkeit ist die Verwendung von verketteten Listen (siehe C-Programm), die aber nicht besonders übersichtlich und wahrscheinlich auch nicht schneller ist.

Die ***Vorteile*** dieses Verfahrens gegenüber der Wechselwirkungsteilchen jedoch ist:

- Es müssen keine Wechselwirkungsobjekte erzeugt und vernichtet werden, d.h. es wird Rechenzeit gespart.
- Die Struktur ist einfacher als mit Wechselwirkungsobjekten.

- ***Die Berechnung der SPH-Kräfte mit Hilfe von Wechselwirkungsobjekten*** (siehe Abb. 4.16).
  Zuerst wird eine Liste von Wechselwirkungsobjekten erzeugt. Jedes Wechselwirkungsobjekt

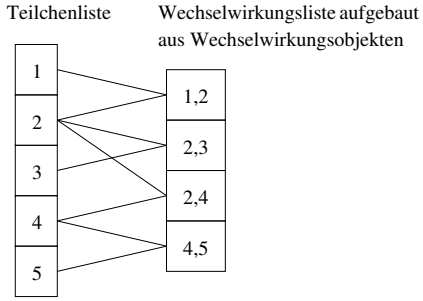


Abbildung 4.16: Berechnung der SPH-Kräfte mit Hilfe von Wechselwirkungsobjekten.

enthält den Anfangs- und Endpunkt des Vektors, d.h. die zwei wechselwirkenden Teilchen (bzw. deren Pointer), den Betrag des Vektors, der kleiner als h sein muß und andere Dinge (z.B. verschiedene Wechselwirkungsterme).
Jedes Teilchen enthält auch eine Liste von Wechselwirkungsobjekten (genauer: eine „STL-Liste“ von Pointer auf Wechselwirkungsobjekte). Wird nun ein Wechselwirkungsobjekt erzeugt, so wird der Pointer auf dieses Objekt den Listen der zwei beteiligten Teilchen angehängt.

Achtung: Der Wachstumsprozeß eines STL-Vektors ist dynamisch. Das bedeutet, daß wenn der Vektor um ein Element zunimmt und dieses Element nicht mehr in den reservierten Speicher paßt, so wird ein größerer Speicherbereich (meist doppelt so groß) allokiert, in den dann der alte Vektor plus das neue Element kopiert wird. Wenn natürlich noch Pointer auf

den alten Speicherplatz existieren, so zeigen diese nach diesem Wachstumsprozeß ins Nichts und verursachen dadurch einen Speicherfehler bzw. erfüllen nicht das gewünschte Verhalten. Deshalb dürfen die Listen von Wechselwirkungsobjekten bzw. die Teilchenlisten ***nicht*** als STL-Vektoren implementiert werden, sondern am Besten mit einer „STL-Liste“!

Die ***Vorteile*** bei der Verwendung von Wechselwirkungsobjekten gegenüber der Benutzung von Teilchenlisten sind:

- Wie oben schon angedeutet, stimmt dieses Verfahren besser mit der Struktur des Simulationsmodells überein.
- Es können Zwischenergebnisse in den Wechselwirkungskräften ablegt werden. Damit kann man die Formeln fast genau in den Code mit übernehmen (siehe Abschnitt 4.3.2.1).
- Die meisten Wechselwirkungskräfte $WW_{i,j}$ sind symmetrisch bzw. antisymmetrisch bezüglich i und j, d.h. $WW_{i,j} = W_{j,i}$ oder $WW_{i,j} = -W_{j,i}$. Daraus folgt, daß die Kräfte nur halb so oft berechnet werden müssen wie in dem anderen Verfahren.

4.3.1.3 Berechnung der Wechselwirkungsobjekte mit Hilfe eines Gitters

Bei der Berechnung der Wechselwirkungsobjekte sucht man die Teilchenpaare, deren Abstand kleiner als die Smoothing-Length sind. Würde man den Abstand von jedem Teilchen berechnen, so steigt der Aufwand mit der Ordnung O(n^2). Wie in dem C-Programm ist es daher effektiver die Teilchen zuerst in ein Gitter zu sortieren. Im Gegensatz zu dem C-Programm, in dem die Gitterbreite frei gewählen werden kann, wird jedoch hier der Einfachheit halber die Gitterbreite gleich der Smooting-Length h gesetzt. Dadurch muß man „nur“ die einem Teilchen direkt angrenzenden Quadrate (insgesamt 26) auswerten. Die Wahrscheinlichkeit, daß ein Teilchen aus den 27 Quadraten ein Wechselwirkungsteilchen ist, liegt bei ca. 15.5%[3].

Da Wechselwirkungsobjekte erzeugt werden und keine Teilchenlisten, muß nur die Hälfte der angrenzenden Quadrate durchsucht werden. Dies soll mit Hilfe der Abbildung 4.17 genauer beschrieben werden: Als erstes wird ein 3-dim. Gitter N^3 mit der Kantenlänge h so angelegt, daß es

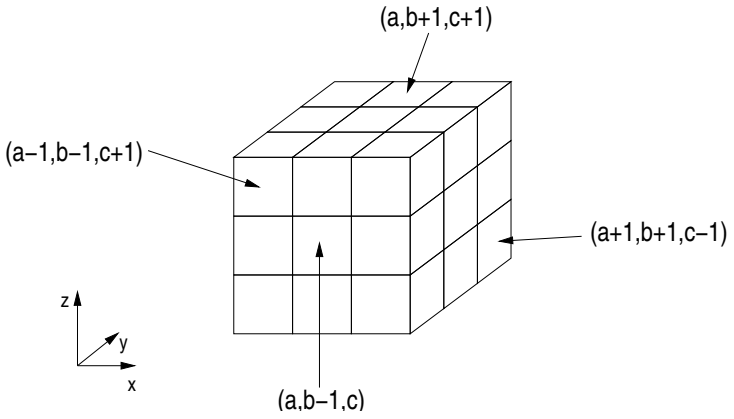


Abbildung 4.17: Veranschaulicht die Berechnung der Wechselwirkungsobjekte.

alle Teilchen überdeckt. Für N^3 gilt: $N^3 := \{(a,b,c)|a,b,c \in [0,N] \subset \mathbb{N} \text{ und } N \in \mathbb{N}\}$. Nun wird mit einer 3-fachen Schleife (über die x,y,z-Achse) jedes Kästchen des Gitters angewählt und mit folgender Prozedur die gesuchten Wechselwirkungsobjekte gefunden:

[3]Herleitung: $\frac{\text{Volumen der Einheitskugel}}{27 \cdot \text{Volumen des Einheitswürfels}} = \frac{\frac{4}{3}\pi}{27} \simeq 0.155$.

Es werden die Wechselwirkungsobjekte aller Teilchen des Kästchens (a,b,c) gesucht, wobei $a, b, c \in [0, N] \subset \mathbb{N}$ ist. Wie auf der Abbildung 4.17 dargestellt, besitzt dieses Kästchen 26 Nachbarn, in der sich die potentiellen Wechselwirkungsteilchen befinden. 13 dieser 26 Kästchen dürfen aber nicht mehr nach Wechselwirkungsteilchen durchsucht werden, da dies durch die 3-fach Schleife schon früher geschah. Die zu durchsuchenden Kästchen sind also:

- $\{(a-1, d, e) | d, e \in \{a-1, a, a+1\}\}$: 9 Kästchen.
- $\{(a, b-1, e) | e \in \{a-1, a, a+1\}\}$: 3 Kästchen.
- $(a, b, c-1)$: 1 Kästchen.

4.3.1.4 Designproblem der SPH-Kräfte

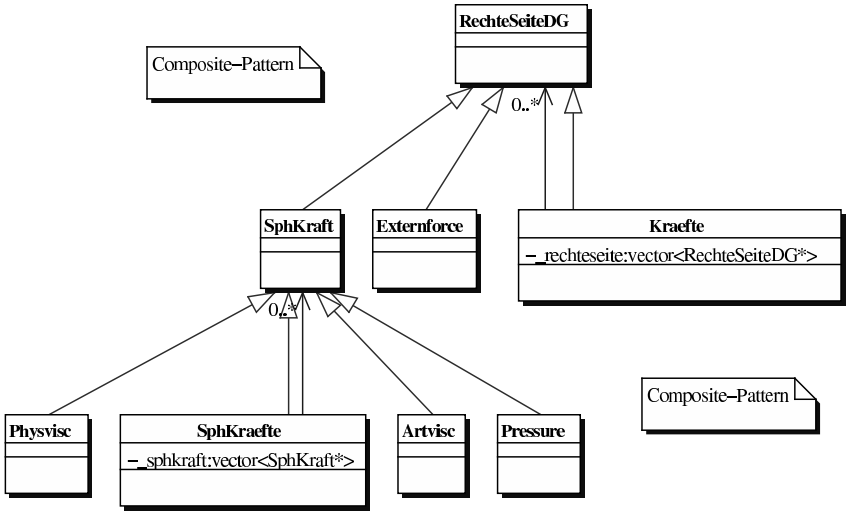


Abbildung 4.18: Die Struktur der SPH-Kräfte besteht aus zwei Composite-Pattern.

Ein grundsätzliches Problem liegt in der Frage an welchem Ort die allen SPH-Kräften ***gemeinsamen*** Daten berechnet bzw. gespeichert werden sollen. Dazu gehören z.B. die Wechselwirkungspartner oder die Dichte jedes Teilchens. Im folgenden werden zwei Vorschläge diskutiert:

- ***Vorschlag 1***: Bei diesem Vorschlag, wie er in Abbildung 4.18 gezeigt wird, gibt es zwei Composite-Pattern. In der Klasse „SPH-Kraefte“ werden die gemeinsamen Daten gespeichert. Dieser Vorschlag hat jedoch folgende Probleme:
  - Bei der Erweiterung einer SPH-Kraft werden Daten von anderen SPH-Kräften gebraucht, d.h. die SPH-Kräfte sind im allgemeinen Fall voneinander abhängig. Dafür ist das Composite-Pattern aber nicht geeignet.
  - Jedem Teilchen und jedem Wechselwirkungsteilchen muß bei jeder einzelne SPH-Kraft ein „Daten-Teilchen“ und ein „Daten-Wechselwirkungsteilchen“ zugeordnet werden. Solch eine Trennung stimmt aber mit der Physik überhaupt nicht überein und kostet Rechenzeit, da diese Datenteilchen mit den ursprünglichen Teilchen synchronisiert werden müssen.

  Ein Vorteil jedoch ist es, daß die benötigten Daten an dem Ort gespeichert werden, an dem sich auch die Algorithmen befinden.

- ***Vorschlag 2***: Bei diesem Vorschlag, der dann auch realisiert wurde, werden die Daten der SPH-Kräfte direkt in den Teilchen bzw. Wechselwirkungsteilchen gespeichert. Dies hat den

Vorteil, daß jede SPH-Kraft auf die Daten jeder anderen SPH-Kraft zugreifen kann. Auch bleibt die Struktur des „Kraefte-Pakets“ mit nur einem Composite-Pattern recht einfach. Dafür handelt man sich aber andere Probleme ein. Zum einen muß bei der Erweiterung einer SPH-Kraft auch die Teilchen bzw. die Wechselwirkungsteilchen erweitert werden und zum anderen werden mit diesem Schritt die Algorithmen von ihren Daten getrennt, was ja nicht besonders schön in einer objektorientierten Sprache ist. Das letztere Problem löste sich während der Implementierung, indem der größte Teil der SPH-Algorithmen auch in die Teilchen bzw. Wechselwirkungsteilchen verschoben wurde.

Im folgenden wird die Implementierung des zweiten Vorschlags, aufgeteilt in Paketen, beschrieben.

4.3.2 Paket: „PhysikalischeObjekte“

Wie im vorigen Abschnitt schon angedeutet, hat sich in diesem Paket am meisten geändert (siehe Abb. 4.19):

- ***Kernel***: Die Form der Reichweite von SPH-Kräften wird durch den Kernel bestimmt. Damit man leicht zusätzliche Kernel hinzufügen und zur Laufzeit auswählen kann, werden die Kernel als Strategy-Pattern (siehe Abschnitt B.2.2) implementiert. Da die Kernel zeitunabhängig sind, kann man über die Klasse „PhysSystem“, die ja die zeitunabhängigen physikalischen Parameter enthält, auf sie zugreifen.

- ***Akkretionsscheiben***: Wie in Abschnitt 4.3.1.4 erläutert, werden die Daten, die bei der Berechnung der SPH-Kräfte gebraucht werden in diesem Paket zur Verfügung gestellt. Im Laufe der Implementierung stellte es sich jedoch heraus, daß wenn man die Algorithmen zur Berechnung der SPH-Kräfte in dem Kräfte-Paket (wo sie ja eigentlich auch hingehören) beläßt, die Implementierung recht kompliziert wird. Diese Kompliziertheit kommt daher, daß sich die Daten entweder in den Teilchen oder in den Wechselwirkungsteilchen befinden. Wenn man nun von außen darauf zugreifen möchte, dann ist erstens eine geeignete Schnittstelle für die Daten notwendig und zweitens geht der Zugriffspfad auf diese Schnittstelle über mehrere Objekte. Um dies zu vermeiden, wurden die Algorithmen einfach auch in dieses Paket verschoben.

  Im folgenden wird der Ableitungsbaum der Akkretionsscheiben beschrieben:

  - Die oberste Ebene besteht aus den Klassen „AkkScheibe“ und „Teilchen“ und wurde schon im Abschnitt 4.2.1 erläutert.
  - Die Ebene darunter besteht aus den Klassen „SphAkkScheibe“, „SphTeilchen“ und „SphWW“. Diese Klassen enhalten die gemeinsamen Daten aller SPH-Kräfte. Die Klassen werden nun einzeln besprochen:
    * Die Klasse ***„SphAkkScheibe“*** ist eine Ableitung der Klasse „AkkScheibe“. Statt des Vektors aus Teilchen enthält sie einen Vektor aus „SphTeilchen“. Zusätzlich generiert sie eine Liste von Objekten der Klasse „SphWW“ (siehe Abschnitt 4.3.1.3).
    * Die Klasse ***„SphTeilchen“*** ist abgeleitet von der Klasse „Teilchen“ und enthält zusätztlich einen Vektor aus „SphWW“ und der Dichte ρ.
    * Die Klasse ***„SphWW“*** beschreibt die Wechselwirkungsteilchen.
  - Die dritte Ebene beschreibt die Akkretionsscheibe, die für die Berechnung des Druckes und der künstlichen Viskosität benötigt wird. Diese Ebene besteht aus den Klassen „PressArtviscAkkScheibe“, „PressArtviscTeilchen“ und „PressArtviscWW“. Diese Klassen sind von den Klassen der zweiten Ebene abgeleitet, und werden folgend einzeln beschrieben:
    * Die Klasse ***„PressArtviscAkkScheibe“*** ist von der Klasse „SphAkkScheibe“ abgeleitet. Statt eines Vektors aus „SphTeilchen“ enthält sie einen Vektor aus „PressArtviscTeilchen“ und statt des Vektors aus „SphWW“ einen Vektor aus „PressArtviscWW“.

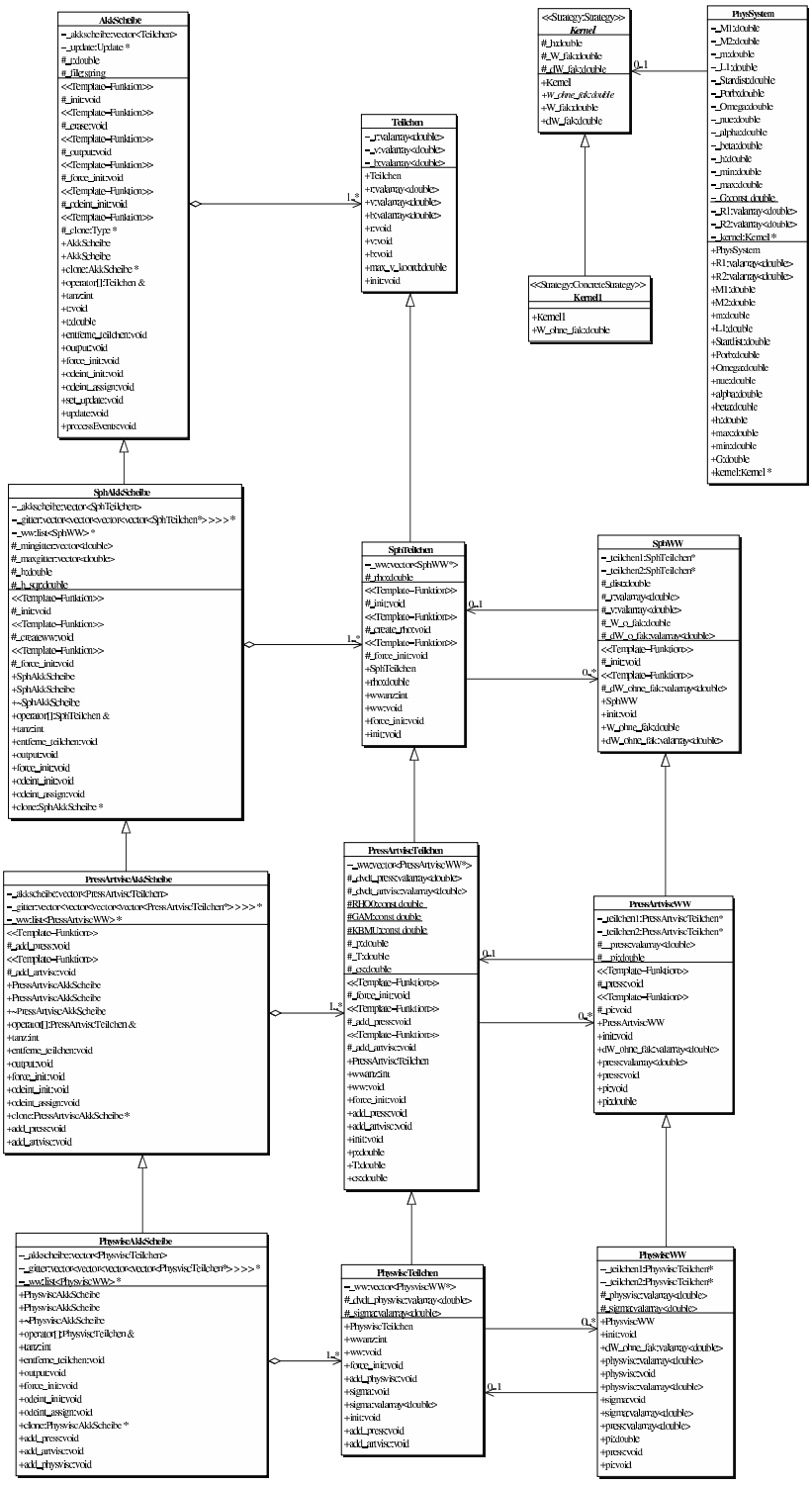


Abbildung 4.19: Das Paket der Physikalischen Objekte in der Phase 3.

    - Die Klasse ***„PressArtviscTeilchen“*** ist abgeleitet von der Klasse „SphTeilchen“.
    - Die Klasse ***„PressArtviscWW“*** ist abgeleitet von der Klasse „SphWW“.
  - Die letzte Ebene enthält die Akkretionsscheibe für die zusätztliche Berechnung der physikalischen Viskosität. Sie besteht aus den Klassen „PhysviscAkkScheibe“, „PhysviscTeilchen“ und „PhysviscWW“:
    - Die Klasse ***„PhysviscAkkScheibe“*** ist von der Klasse „PressArtviscAkkScheibe“ abgeleitet. Statt des Vektors aus „PressArtviscTeilchen“ enthält sie einen Vektor aus „PhysviscTeilchen“ und statt des Vektors aus „PressArtviscWW“ einen Vektor aus „PhysviscWW“.
    - Die Klasse ***„PhysviscTeilchen“*** ist von der Klasse „PressArtviscTeilchen“ abgeleitet.
    - Die Klasse ***„PhysviscWW“*** ist abgeleitet von der Klasse „PressArtviscWW“.

- ***Templatefunktionen***: Es ist noch anzumerken, daß viele Funktionen Templatefunktionen sind. Beispielsweise unterscheiden sich diese Funktionen in der „SphAkkScheibe“ von denen in der „AkkScheibe“ nur durch „vector< $SphTeilchen$ >“ statt „vector< $Teilchen$ >“.

4.3.2.1 Berechnung der SPH-Kräfte

Die Berechnung der SPH-Kräfte wird in diesem Abschnitt anhand eines „Demo“-Source Codes erläutert. Der Unterschied der „Demo“-Klassen, deren Namen mit dem Wort „Demo“ beginnen, zu den tatsächlich implementierten Klassen ist das Weglassen aller derjenigen Teile, die für das Verständnis der SPH-Kräfte-Berechnung keine Rolle spielen. Weggelassen wurde z.B. die Ableitungsstruktur, der etwas verwirrende Funktion-Template-Mechanismus und die Konstruktoren.

Nachfolgend stehen die Definitionen der „Demo“-Klassen ,die für die Berechnung der drei SPH-Kräfte benötigt werden. Durch Aufrufen der Memberfunktionen „add_press(...)“, „add_artvisc(...)“ und „add_physvisc(...)“ werden die drei SPH-Kräfte für jedes Teilchen berechnet und der Gesamtkraft hinzuaddiert.

```
#include <vector>
#include <list>
#include <valarray>

class DemoTeilchen;
class DemoWW;

class DemoKernel
{
private:
  double _W_fak, _dW_fak;                                  // Faktor des Kernels bzw. dessen Ableitung.
public:
  double W_fak() {return _W_fak;}
  double dW_fak() {return _dW_fak;}
  double W_ohne_fak(double dist, valarray<double>& dW); // gibt "W_ohne_fak" und "dW_ohne_fak" zurück.
};

class DemoPhysSystem
{
private:
  double _m;            // Masse eines Teilchens
  DemoKernel* _kernel;  // benutzter Kernel.
  double _h;            // SmoothingLenght.
  double _alpha, _beta; // Parameter der künstlichen Viskosität.
  double _nue;          // konstante Viskosität.
public:
  double m() {return _m;}
  DemoKernel* kernel() {return _kernel;}
  double h() {return _h;}
  double alpha() {return _alpha;}
  double beta() {return _beta;}
  double nue() {return _nue;}
};

class DemoAkkScheibe
{
```

```
private:
  vector<DemoTeilchen> _akkscheibe;
  list<DemoWW>* _ww;
public:
  void add_press(DemoPhysSystem& sys);    // Fügt zur Gesamtkraft die Druckkraft hinzu.
  void add_artvisc(DemoPhysSystem& sys);  // Fügt zur Gesamtkraft die künstliche Viskositätskraft hinzu.
  void add_physvisc(DemoPhysSystem& sys); // Fügt zur Gesamtkraft die phyikalische Viskositätskraft hinzu.
};

class DemoTeilchen
{
private:
  vector<DemoWW*> _ww;
  valarray<double> _dvdt_press;    // Ist die innere Druckkraft des Teilchens.
  valarray<double> _dvdt_artvisc;  // Ist die künstliche Viskositätskraft des Teilchens.
  valarray<double> _dvdt_physvisc; // Ist die physikalische Viskositätskraft des Teilchens.
  valarray<double> _sigma;         // Reihenfolge der Parameter des sigma-Tensors: xx, xy, xz, yy, yz, zz
  valarray<double> _b;             // Beschleunigung bzw. die Gesamtkraft, die auf das Teilchen wirkt.
  double _rho;                     // die Dichte des Teilchens.
  double _p;                       // der Druck aus der Polytropen Zustandsgleichung.
  double _cs;                      // die Schallgeschwindigkeit.
public:
  void add_press(DemoPhysSystem& sys);   // berechnet die innere Druckkraft und addiert sie zur Gesamtkraft.
  void add_artvisc(DemoPhysSystem& sys); // berechnet die künst. Visk.kraft und addiert sie zur Gesamtkraft.
  void sigma(DemoPhysSystem& sys);       // Diese Funktion berechnet den sigma-Tensor.
  valarray<double> sigma() {return _sigma;}
  void add_physvisc(DemoPhysSystem& sys);// berechnet die phys. Visk.kraft und addiert sie zur Gesamtkraft.
  valarray<double> b() {return _b;}
  void b(valarray<double> b) { _b=b; }
  double rho() {return _rho;}
  double p() {return _p;}
  double cs() {return _cs;}
};

class DemoWW
{
private:
  DemoTeilchen *_teilchen1, *_teilchen2; // die Wechselwirkung bezieht sich auf 2 Teilchen.
  double _dist;                          // Abstand zwischen den Teilchen.
  valarray<double> _r;                   // Orts-Differenzvektor zwischen den Teilchen.
  valarray<double> _v;                   // Geschwindigkeits-Differenzvektor zwischen den Teilchen
  valarray<double> _press;               // Für die Druckberechnung benötigter Term.
  double _pi;                            // Für die künstliche Viskositätsberechnung benötigter Term.
  valarray<double> _physvisc;            // Für die physikalische Viskositätsberechnung benötigter Term.
  valarray<double> _sigma;               // Für die Berechnung des sigma-Tensors benötigter Term.
  double _W_o_fak;                       // Kernel-Wert W ohne den Faktor (siehe Kernel-Klasse).
  valarray<double> _dW_o_fak;            // Kernel-Ableitung dW ohne Faktor ( siehe Kernel-Klasse )
public:
  valarray<double> press() {return _press;}
  void press(DemoPhysSystem& sys);               // Berechnet die Variable "__press".
  double pi() {return _pi;}
  void pi(DemoPhysSystem& sys);                  // Berechnet die Variable "__pi".
  void physvisc(DemoPhysSystem& sys);            // Berechnet die Variable "_physvisc".
  valarray<double> physvisc(DemoTeilchen* t);    // gibt den pos. bzw. neg. Wert von "_physvisc" zurück.
  void sigma(DemoPhysSystem& sys);               // Die Funktion berechnet die Variable "_sigma".
  valarray<double> sigma(DemoTeilchen* t);       // gibt den pos. bzw. neg. Wert von "_sigma" zurück.
  double W_ohne_fak() {return _W_o_fak;}         // Gibt den Kernel ohne den Faktor zurück.
  valarray<double> dW_ohne_fak(DemoTeilchen* t); // Gibt die Kernel-Abl. dW in Abh. von t zurück.
};

valarray<double> DemoWW::dW_ohne_fak(DemoTeilchen* t)
{
  if (t == _teilchen1)
    return _dW_o_fak;
  else if (t == _teilchen2)
    return  (- _dW_o_fak);
  else {
    cout << "Angegebenes DemoTeilchen gehört nicht zu dieser DemoWW !" << endl;
  }
}
```

Die Implementierung der einzelnen SPH-Kräfte wird im Folgenden beschrieben. Davor soll jedoch kurz auf die einzelnen Schritte eingegangen werden, die notwendig sind, um eine allgemeine SPH-Kraft $\left.\frac{dv_{i_\alpha}}{dt}\right|_{allg.} = \sum_j a(i,j) ... \sum_k b(i,k) \sum_l c(i,l)$ in die „DemoTeilchen“- und „DemoWW“-Klasse zu implementieren:

- Der Summand der innersten Summe (hier: $c(i,l)$) wird in einen konstanten Teil c_1 und in

den Rest $c_2(i,l)$ aufgeteilt: $c(i,l) = c_1 c_2(i,l)$. Der Term $c_2(i,l)$ wird nun in der „DemoWW"-Klasse berechnet.

- Die Summe $\sum_l c(i,l) = c_1 \sum_l c_2(i,l)$ wird in der „DemoTeilchen"-Klasse berechnet.
- Die oberen zwei Schritte werden für die nächst äußeren Summen solange wiederholt, bis die ganze Gleichung implementiert ist.

Der Vorteil dieser Methode liegt in der besseren Effizienz. Diese folgt daher, daß die meisten Summanden symmetrisch bzw. antisymmetrisch sind (z.B. $c(i,l) = c(l,i)$) und daher nur halb so oft berechnet werden müssen, als wenn sie in der „DemoTeilchen"-Klasse implementiert wären.

Berechnung der inneren Druckkräfte: Durch Aufrufen der Memberfunktion „add_press" aus der Klasse „DemoAkkScheibe" werden die inneren Druckkräfte jedes Teilchens berechnet und der Gesamtkraft hinzugefügt.

Aus (2.14) gilt für die innere Druckkraft, die auf ein Teilchen wirkt:

$$\left.\frac{dv_{i_\alpha}}{dt}\right|_{pressure} = -\frac{1}{\rho_i}\frac{\partial p_i}{\partial x_\alpha} \overset{(2.14)}{=} -\frac{1}{\rho_i}\sum_j m_j \frac{p_j + p_i}{\rho_j}\frac{\partial W_{ij}}{\partial x_\alpha}$$

wobei für die Simulation die Masse $m_j = m \ \forall j$ gilt. Dadurch sieht die zu implementierende Gleichung folgendermaßen aus:

$$\left.\frac{dv_{i_\alpha}}{dt}\right|_{pressure} \overset{m_j=m}{=} \underbrace{-m\sum_j \overbrace{\frac{p_j + p_i}{\rho_i \rho_j}}^{=:\_press}\frac{\partial W_{ij}}{\partial x_\alpha}}_{=:\_dvdt\_press} \tag{4.1}$$

Die Implementierung dieser Gleichung teilt sich auf in die Memberfunktion „add_press" aus der Klasse „DemoAkkScheibe", in die Memberfunktion „add_press" aus der Klasse „DemoTeilchen" und in die Memberfunktion „press" aus der Klasse „DemoWW" und ist ein „zweistufiger" Prozeß:

1. Für jedes WW-Objekt wird die Membervariable $\_press := \_press(i,j) := \frac{p_i + p_j}{\rho_i \rho_j}$ berechnet.
2. Dann wird die Membervariable $\_dvdt\_press := \_dvdt\_press(i) := -\sum_j (i,j) \_press(i,j) \frac{\partial W_{ij}}{\partial x_\alpha}$ für jedes Teilchen berechnet.

Der Source-Code dazu sieht so aus:

```
void DemoAkkScheibe::add_press(DemoPhysSystem& sys)
{
  for ( list<DemoWW>::iterator iter = _ww->begin(); iter != _ww->end(); ++iter)
    iter->press(sys);

  for (int i=0; i<_akkscheibe.size(); i++)
    _akkscheibe[i].add_press(sys);
}

void DemoWW::press(DemoPhysSystem& sys)
{
  _press = ( _teilchen1->p() + _teilchen2->p()) / ( _teilchen1->rho() * _teilchen2->rho() );
}

void DemoTeilchen::add_press(DemoPhysSystem& sys)
{
  for (int i=0; i<3; i++) _dvdt_press[i] = 0.0;
  for (int i=0; i<_ww.size(); i++)
    _dvdt_press -= _ww[i]->press() * _ww[i]->dW_ohne_fak(this);
  _dvdt_press *= sys.m() * sys.kernel()->dW_fak();

  b( b() + _dvdt_press );
}
```

Berechnung der künstlichen Viskosität: Durch Aufrufen der Memberfunktion „add_artvisc“ aus der Klasse „DemoAkkScheibe“ wird die künstliche Viskositätkraft jedes Teilchens berechnet und der Gesamtkraft hinzugefügt.

Aus (2.20), (2.21) und (2.22) gilt für die künstliche Viskositätskraft, die auf ein Teilchen wirkt:

$$\left.\frac{dv_{i_\alpha}}{dt}\right|_{art.visc.} \overset{(2.22)}{=} -\sum_j m_j \Pi_{ij} \delta_{\alpha\beta} \frac{\partial W_{ij}}{\partial x_\beta}$$

mit der künstlichen Viskosität:

$$\Pi_{ij} \overset{(2.21)}{:=} \begin{cases} \frac{-\alpha \bar{c}_{s_{ij}} \mu_{ij} + \beta \mu_{ij}^2}{\bar{\rho}_{ij}} & : \quad (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) < 0 \\ 0 & : \quad \underbrace{(\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j)}_{=:vr} \geq 0 \end{cases}$$

Dabei sind $\bar{c}_{s_{ij}} := \frac{c_{s_i} + c_{s_j}}{2}$ und $\bar{\rho_{ij}} := \frac{\rho_i + \rho_j}{2}$ Mittelwerte und

$$\mu_{ij} \overset{(2.20)}{:=} \frac{h(\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j)}{(\vec{r}_i - \vec{r}_j)^2 + \epsilon h^2}.$$

Wie oben ist $m_j = m$ und daher bekommt die zu implementierende Gleichung folgende Gestalt:

$$\left.\frac{dv_{i_\alpha}}{dt}\right|_{art.visc.} \overset{m_j=m}{=} \underbrace{-m \sum_j \overbrace{\Pi_{ij}}^{=:\_pi} \frac{\partial W_{ij}}{\partial x_\alpha}}_{=:\_dvdt\_artvisc} \tag{4.2}$$

Die Implementierung dieser Gleichung teilt sich auf in die Memberfunktion „add_artvisc“ aus der Klasse „DemoAkkScheibe“, in die Memberfunktion „add_artvisc“ aus der Klasse „DemoTeilchen“ und in die Memberfunktion „pi“ aus der Klasse „DemoWW“ und ist ein „zweistufiger“ Prozeß:

1. Für jedes WW-Objekt wird die Membervariable $\_pi := \_pi(i,j) := \Pi_{ij}$ berechnet.
2. Dann wird die Membervariable $\_dvdt\_artvisc := \_dvdt\_artvisc(i) := -m \sum_j \Pi_{ij} \frac{\partial W_{ij}}{\partial x_\alpha}$ für jedes Teilchen berechnet.

Der Source-Code dazu sieht folgendermaßen aus:

```
void DemoAkkScheibe::add_artvisc(DemoPhysSystem& sys)
{
  for ( list<DemoWW>::iterator iter = _ww->begin(); iter != _ww->end(); ++iter)
    iter->pi(sys);

  for (int i=0; i<_akkscheibe.size(); i++)
    _akkscheibe[i].add_artvisc(sys);
}

void DemoWW::pi(DemoPhysSystem& sys)
{
  static const double epsilon = 0.01; // vermeidet Singularitäten !
  double vr = inner_product(&_v[0], &_v[3], &_r[0], 0.0);

  if ( vr < 0.0 ) {
    double mu = sys.h() * vr / ( _dist * _dist + epsilon * sys.h() * sys.h() );
    _pi = ( - sys.alpha() * ( _teilchen1->cs() + _teilchen2->cs() ) + 2 * sys.beta() * mu ) * mu
          / ( _teilchen1->rho() + _teilchen2->rho() );
  }
  else _pi = 0.0;
}

void DemoTeilchen::add_artvisc(DemoPhysSystem& sys)
{
  for (int i=0; i<3; i++) _dvdt_artvisc[i] = 0.0;
```

```
  for (int i=0; i<_ww.size(); i++)
    _dvdt_artvisc -= _ww[i]->pi() * _ww[i]->dW_ohne_fak(this);
  _dvdt_artvisc *= sys.m() * sys.kernel()->dW_fak();

  b( b() + _dvdt_artvisc );
}
```

Berechnung der physikalischen Viskosität: Durch Aufrufen der Funktion „add_physvisc“ aus der Klasse „DemoAkkScheibe“ wird die physikalische Viskositätskraft jedes Teilchens berechnet und der Gesamtkraft hinzugefügt.

Aus (2.15) und (2.17) gilt für die physikalische Viskositätskraft, die auf ein Teilchen wirkt:

$$\left.\frac{dv_{i_\alpha}}{dt}\right|_{phys.visc.} \overset{(2.15 2.17)}{=} -\sum_j m_j \frac{\eta_i \sigma_{i_{\alpha\beta}} + \eta_j \sigma_{j_{\alpha\beta}}}{\rho_i \rho_j} \frac{\partial W_{ij}}{\partial x_\beta} \tag{4.3}$$

mit folgender Abkürzung:

$$\begin{aligned} \sigma_{i_{\alpha\beta}} &:= V_{i_{\alpha\beta}} + V_{i_{\beta\alpha}} - \frac{2}{3}\delta_{\alpha\beta} V_{i_{\gamma\gamma}} \\ V_{i_{\alpha\beta}} &:= m_i \sum_j \frac{(\vec{v}_i - \vec{v}_j)_\alpha}{\rho_j} \frac{\partial W_{ij}}{\partial x_\beta} \end{aligned}$$

Für die Simulation gilt wieder wie oben $m_j = m$ und zusätzlich soll η nur von der Dichte abhängen: $\eta_i = \eta\rho_i$. Eingesetzt in die obige Gleichung ergibt sich die zu implementierende Gleichung:

$$\left.\frac{dv_{i_\alpha}}{dt}\right|_{phys.visc.} \overset{m_j=m, \eta_j=\eta\rho_j}{=} \underbrace{-m\eta \sum_j \overbrace{\left(\frac{\sigma_{i_{\alpha\beta}}}{\rho_j} + \frac{\sigma_{j_{\alpha\beta}}}{\rho_i}\right)}^{=:DemoWW::\_physvisc} \frac{\partial W_{ij}}{\partial x_\beta}}_{=:DemoTeilchen::\_dvdt\_physvisc} \tag{4.4}$$

mit der Abkürzung:

$$\sigma_{i_{\alpha\beta}} \overset{m_j=m}{=} \underbrace{m \sum_j \frac{1}{\rho_j} \overbrace{\left((\vec{v}_i - \vec{v}_j)_\alpha \frac{\partial W_{ij}}{\partial x_\beta} + (\vec{v}_i - \vec{v}_j)_\beta \frac{\partial W_{ij}}{\partial x_\alpha} - \frac{2}{3}\delta_{\alpha\beta}(\vec{v}_i - \vec{v}_j)_\gamma \frac{\partial W_{ij}}{\partial x_\gamma}\right)}^{=:DemoWW::\_sigma}}_{=:DemoTeilchen::\_sigma} \tag{4.5}$$

Die Implementierung dieser Gleichung teilt sich auf in die Memberfunktion „add_physvisc“ aus der Klasse „DemoAkkScheibe“, in die Memberfunktionen „sigma“und „add_physvisc“ aus der Klasse „DemoTeilchen“ und in die Memberfunktionen „sigma“ und „physvisc“ aus der Klasse „DemoWW“ und ist ein „vierstufiger“ Prozeß:

1. Für jedes „DemoWW“-Objekt wird die Membervariable „DemoWW::_sigma“ berechnet. Die Beziehung zu dem sym. Tensor $\sigma'_{i_{\alpha\beta}} := (\vec{v}_i - \vec{v}_j)_\alpha \frac{\partial W_{ij}}{\partial x_\beta} + (\vec{v}_i - \vec{v}_j)_\beta \frac{\partial W_{ij}}{\partial x_\alpha} - \frac{2}{3}\delta_{\alpha\beta}(\vec{v}_i - \vec{v}_j)_\gamma \frac{\partial W_{ij}}{\partial x_\gamma}$ ist dabei $(\_sigma_0, ..., \_sigma_5)|_i := (\sigma'_{i_{11}}, \sigma'_{i_{12}}, \sigma'_{i_{13}}, \sigma'_{i_{22}}, \sigma'_{i_{23}}, \sigma'_{i_{33}})$.

2. Für jedes „DemoTeilchen“ wird nun die Membervariable „DemoTeilchen::_sigma“ berechnet. Dabei gilt $DemoTeilchen::\_sigma := m \sum_j \frac{1}{\rho_j} DemoWW::\_sigma$.

3. Als nächstes wird für jedes „DemoWW“-Objekt die Membervariable „DemoWW::_physvisc“ berechnet. Dabei gilt $DemoWW::\_physvisc|_i := (\frac{\sigma_{i_{\alpha\beta}}}{\rho_j} + \frac{\sigma_{j_{\alpha\beta}}}{\rho_i})\frac{\partial W_{ij}}{\partial x_\beta}$.

4. Zuletzt wird für jedes „DemoTeilchen“ die Membervariable „DemoTeilchen::_dvdt_physvisc“ mit $DemoTeilchen::\_dvdt\_physvisc := m\eta \sum_j DemoWW::\_physvisc$ berechnet.

Der Source-Code dazu sieht so aus:

```
void DemoAkkScheibe::add_physvisc(DemoPhysSystem& sys)
{
  for (list<DemoWW>::iterator iter = _ww->begin(); iter != _ww->end(); ++iter)
    iter->sigma(sys);

  for (int i=0; i<_akkscheibe.size(); i++)
    _akkscheibe[i].sigma(sys);

  for (list<DemoWW>::iterator iter = _ww->begin(); iter != _ww->end(); ++iter)
    iter->physvisc(sys);

  for (int i=0; i<_akkscheibe.size(); i++)
    _akkscheibe[i].add_physvisc(sys);
}

void DemoWW::sigma(DemoPhysSystem& sys)
{
  double zd = 2.0 / 3.0;
  double vd = 4.0 / 3.0;

  _sigma[0] = vd * _v[0] * _dW_o_fak[0] - zd * ( _v[1] * _dW_o_fak[1] + _v[2] * _dW_o_fak[2] );   // xx
  _sigma[1] = _v[0] * _dW_o_fak[1] + _v[1] * _dW_o_fak[0];                                        // xy
  _sigma[2] = _v[0] * _dW_o_fak[2] + _v[2] * _dW_o_fak[0];                                        // xz
  _sigma[3] = vd * _v[1] * _dW_o_fak[1] - zd * ( _v[0] * _dW_o_fak[0] + _v[2] * _dW_o_fak[2] );   // yy
  _sigma[4] = _v[1] * _dW_o_fak[2] + _v[2] * _dW_o_fak[1];                                        // yz
  _sigma[5] = vd * _v[1] * _dW_o_fak[1] - zd * ( _v[0] * _dW_o_fak[0] + _v[1] * _dW_o_fak[1] );   // zz
}

void DemoTeilchen::sigma(DemoPhysSystem& sys)
{
  for (int i=0; i<6; i++) _sigma[i] = 0.0;

  for (int i=0; i<_ww.size(); i++)
    _sigma += _ww[i]->sigma(this);

  _sigma *= sys.kernel()->dW_fak() * sys.m();
}

valarray<double> DemoWW::sigma(DemoTeilchen* t)
{
  if (t == _teilchen1)
    return ( _sigma / _teilchen2->rho() );
  else if (t == _teilchen2)
    return ( _sigma / _teilchen1->rho() );
  else
    cout << "Angegebenes Teilchen gehört nicht zu dieser DemoWW !" << endl;
}

void DemoWW::physvisc(DemoPhysSystem& sys)
{
  valarray<double> hilf(6);   // Reihenfolge der Parameter: xx, xy, xz, yy, yz, zz

  hilf = _teilchen1->sigma() / _teilchen2->rho() + _teilchen2->sigma() / _teilchen1->rho();

  _physvisc[0] = hilf[0] * _dW_o_fak[0] + hilf[1] * _dW_o_fak[1] + hilf[2] * _dW_o_fak[2];
  _physvisc[1] = hilf[1] * _dW_o_fak[0] + hilf[3] * _dW_o_fak[1] + hilf[4] * _dW_o_fak[2];
  _physvisc[2] = hilf[2] * _dW_o_fak[0] + hilf[4] * _dW_o_fak[1] + hilf[5] * _dW_o_fak[2];
}

void DemoTeilchen::add_physvisc(DemoPhysSystem& sys)
{
  for (int i=0; i<3; i++) _dvdt_physvisc[i] = 0.0;
  for (int i=0; i<_ww.size(); i++)
    _dvdt_physvisc += _ww[i]->physvisc(this);
  _dvdt_physvisc *= sys.m() * sys.nue() * sys.kernel()->dW_fak();

  b( b() + _dvdt_physvisc );
}

valarray<double> DemoWW::physvisc(DemoTeilchen* t)
{
  if (t == _teilchen1)
    return _physvisc;
  else if (t == _teilchen2)
    return ( - _physvisc);
  else
    cout << "Angegebenes Teilchen gehört nicht zu dieser DemoWW !" << endl;
}
```

4.3.3 Paket: „Kraefte“

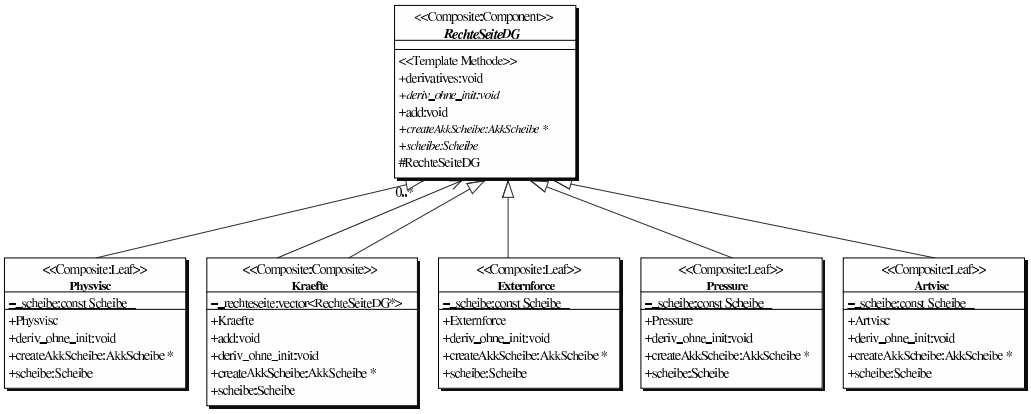


Abbildung 4.20: Das Paket der Kräfte in der Phase 3.

Das Paket „Kraefte“ (siehe Abb. 4.20) ist nach dem Composite-Pattern (siehe Abb. B.6) aufgebaut und nicht mehr wie in den vorigen Phasen nach dem Strategy-Pattern. Der Grund dafür ist, daß man mit dem Composite-Pattern beliebige Kombinationen von Kräften bilden kann, was mit dem Strategy-Pattern nicht möglich ist. Daß das Strategy-Pattern in den vorigen Phasen funktionierte, lag nur an der dort einzig zu behandelnden Kraft, nämlich der Gravitationskraft. Die Zusammenstellung der gewünschten Kräfte in diesem Paket geschieht mit Hilfe des Simulationspakets zur Laufzeit. Die „RechteSeiteDG“-Klasse entspricht der „Component“-Klasse, die „Kraefte“-Klasse der „Composite“-Klasse und die anderen Klassen sind „Leaf“-Klassen[4].

Interessant ist noch die Memberfunktion „createAkkScheibe“ aus der Klasse „RechteSeiteDG“, die in Abhängigkeit von den ausgewählten Kräften einen Pointer auf die passende Akkretionsscheibe zurückgibt. Diese Funktion wird von einem Objekt der „Simulationen“-Klasse aufgerufen. Wenn mehr als zwei Kräfte gewählt werden, so zeigt die folgende Definition der Memberfunktion „createAkkScheibe“ aus der „Kraefte“-Klasse, wie die Auswahl der Akkretionsscheibe erfolgt:

```
AkkScheibe* Kraefte::createAkkScheibe(InitAkkScheibe& iniakk, PhysSystem& sys)
{
  Scheibe sch = scheibe();
  switch(sch){
  case akk:
    return new AkkScheibe(iniakk, sys);
  case sphakk:
    return new SphAkkScheibe(iniakk, sys);
  case pressartviscakk:
    return new PressArtviscAkkScheibe(iniakk, sys);
  case physviscakk:
    return new PhysviscAkkScheibe(iniakk, sys);
  default:
    cout << "Fehler in Kraefte::createAkkScheibe !" << endl;
  }
}
```

Dabei enthält jede Kraft eine „private“ Membervariable aus der folgenden Menge:

```
enum Scheibe{ akk, sphakk, pressartviscakk, physviscakk };
```

Auch das Template-Pattern (siehe Abb. B.13) wird in diesem Paket verwendet. Die „Abstrakte Klasse“ ist hier die „RechteSeiteDG“-Klasse mit der Template-Methode „derivatives“. Die Definition dieser Templatemethode hat folgende Gestalt:

[4]Normalerweise müßte diese Beschreibung in der Abb. 4.20 als Kollaboration (siehe Abb. A.3) dargestellt werden. Dies kann Together 3.2 jedoch leider (noch ?) nicht leisten.

```
void RechteSeiteDG::derivatives(AkkScheibe &akk, PhysSystem &sys)
{
  akk.force_init(sys);
  deriv_ohne_init(akk, sys);
}
```

Wenn man diese Definition mit der Abbildung B.13 vergleicht, so entspricht die Memberfunktion „deriv_ohne_init“ einer abstrakten „primitiven Operation“. Diese Funktion hat die Aufgabe, die Kräfte zu berechnen, ohne daß der Speicherplatz, der das Ergebnis aufnehmen soll, vorher initialisiert wird. Diese Initialisierung geschieht durch Aufrufen der Funktion „force_init“ der Akkretionsscheibe. Damit wird erreicht, daß bei der Kräfteberechnung zuerst initialisiert wird und dann nachfolgend die einzelnen Kräfte aufsummiert werden!

4.3.4 Paket: „Integratoren“

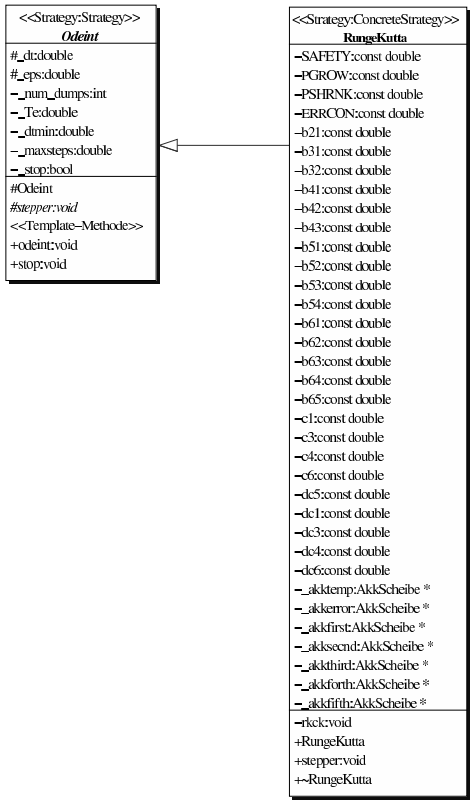


Abbildung 4.21: Das Paket der Integratoren in der Phase 3.

Das Paket „Integratoren“ (siehe Abb. 4.21) wurde gegenüber der zweiten Phase so angepasst, daß auch abgeleitete Akkretionsscheiben integriert werden können. Dies wird dadurch erreicht, daß der „RungeKutta“-Integrator die „public“-Memberfunktion „clone“ aus der abgeleiteten Akkretionsscheibe für seine Hilfs-Akkretionsscheiben aufruft. Diese Funktion gibt ein Pointer auf ein neu instanziiertes Objekt der eigenen Klasse zurück.

Weiterhin ist der Driver des „RungeKutta“-Integrators in die abstrakte Klasse „Odeint“ verlegt worden, da dieser Driver für zusätzliche Integratoren derselbe bleibt. Die restlichen Teile des „RungeKutta“-Integrators (Stepper und Algorithmus) wurden in eine Klasse zusammengefaßt.

4.3.5 Paket: „EinAusgabe“

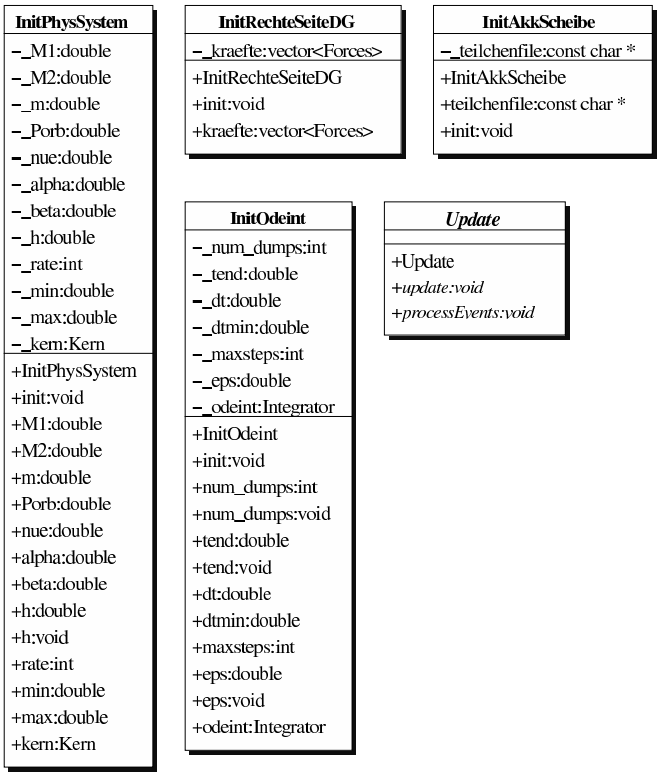


Abbildung 4.22: Das Paket der Ein-/Ausgabe in der Phase 3.

Das Paket „EinAusgabe“ (siehe Abb. 4.22) hat sich gegenüber der zweiten Phase nicht groß verändert, sondern es wurde nur den neuen Anforderungen angepaßt. Beispielsweise wurde die Elternklasse „Init“ weggelassen, da sie keine gemeinsamen Member-variablen bzw. -funktionen enthielt.

Neu hinzugekommen ist die abstrakte Klasse „Update“, die die Schnittstelle zwischen der Simulation und einer GUI ist. Die Kommunikation geschieht dadurch, daß die GUI eine konkrete Klasse von der Klasse „Update“ ableitet, die auf die eigenen Anforderungen zugeschnitten ist, und anschließend dessen Pointer der Akkretionsscheibe übergibt. Die Akkretionsscheibe kann nun über die Memberfunktion „update()“ der GUI mitteilen, daß sich die Akkretionsscheibe geändert hat. Diese Funktion wird in „Odeint::odeint“ aufgerufen, sobald eine neue Akkretionsscheibe berechnet wird. Über die Memberfunktion „processEvents()“ wird die Kontrolle kurz auf die GUI übertragen, damit sie empfangene Events abarbeiten kann. Dies geschieht z.B. vor jeder Kräfteberechnung in der Memberfunktion „RechteSeiteDG::derivatives“.

4.3.6 Paket: „Simulationen“

Das Paket „Simulationen“ enthält gegenüber der zweiten Phase einige neue Klassen, die die Steuerung der Simulation zusätzlich durch eine GUI ermöglicht (siehe Abb. 4.23). Damit gibt es also zwei verschiedenen Arten, die Simulation zu steuern:

1. Die Klasse „SimulationConsole“ steuert die Simulation in einem Terminal, wie es in der zweiten Phase schon beschrieben wurde.

2. Die Klasse „SimulationQt“ mit den Hilfsklassen „AkkDisplayQt“ und „UpdateQt“ steuert die Simulation mit Hilfe eines GUI (siehe Abb. 4.24). Die GUI wird im folgenden noch genauer beschrieben.

Da diese zwei Simulationsklassen gemeinsame Variablen besitzen, wurden diese in eine gemeinsame Elternklasse „Simulation“ verschoben.

4.3.6.1 Die GUI der Simulation

Das Ziel einer GUI ist die einfachere Benutzung eines Programmmes. Das GUI dieses Programms ist in Abb. 4.24 dargestellt und basiert auf der Qt-Library. Die Auswahlmöglichkeiten bzw. die Anzeigen des Programms werden im folgenden kurz beschrieben:

- ***File-Änderung***: Mit diesen zwei Push-Buttons, die jeweils einen File-Dialog aufmachen, wird das gewünschte Parameterfile bzw. die Akkretionsscheibe ausgewählt, bei der die Simulation beginnen soll.
- ***Kern***: Durch die Radio-Buttons kann man den gewünschten Kernel auswählen. Diese Auswahl („entweder-oder“) paßt zur Implementierung der Kernel durch das „Strategy-Pattern“. Da bis jetzt nur ein Kernel implementiert wurde, wird bei der Auswahl des „Kern1“ und des „Kern2“ jeweils der gleiche Kernel ausgewählt.
- ***Kräfte***: Alle Kombinationen der vier Kräfte kann man mit diesen Check-Boxes auswählen. Dies paßt zu der Implementierung der Kräfte durch das „Composite-Pattern“.
- ***Integrator***: Mit den Radio-Buttons kann der gewünschte Integrator ausgewählt werden. Wie bei der Auswahl des Kernels sind die Integratoren als „Strategy-Pattern“ implementiert und da bis jetzt nur ein Integrator implementiert ist, bewirkt die Auswahl von „RungeKutta1“ und „RungeKutta2“ genau dasselbe.
- ***Start/Stop der Simulation***: Mit diesem Push-Button wird die Simulation gestartet bzw. gestoppt. Beim Start der Simulation werden die Auswahl-Widgets deaktiviert, d.h. sie können nicht mehr angewählt werden. Weiterhin werden die ausgewählten Objekte des Kernels, des Integrator, der Kräfte und der zugehörigen Akkretionsscheibe erzeugt und die Simulation gestartet. Wird die Simulation gestoppt, so wird die Funktion „Odeint::stop()“aufgerufen, um den Integrator zu stoppen und die Auswahl-Widgets werden wieder aktiviert, damit man die nächste Simulation neu konfigurieren kann.
- ***Pause/Weiter***: Dieser Push-Button hat die Aufgabe die Simulation zu unterbrechen, damit man die Akkretionsscheibe durch die Slider flüssiger drehen kann.
- ***Reset***: Durch diesen Push-Button wird das unter „File-Änderung“ gewählte Parameterfile ausgelesen und deren Werte für die Simulation übernommen.
- ***Quit***: Zuerst wird die Simulation und danach das GUI beendet.
- ***Graphische Darstellung der Akkretionsscheibe***: Nach jeder neuen Berechnug der Akkretionsscheibe wird diese wie in Abb. 4.24 gezeigt graphisch dargestellt. Durch die zwei Slider hat man die Möglichkeit, die Ansicht durch Änderung der Winkel „phi“ und „theta“ zu ändern.
- ***phi/theta***: Die LCD-Anzeigen für „phi“ und „theta“ zeigen die durch die zwei Slider gewählten Winkel.
- ***Zeit***: Gibt die Zeit der Akkretionsscheibe aus.
- ***Teilchenzahl***: Gibt die aktuelle Teilchenzahl der Akkretionsscheibe aus.
- ***Anzahl Outputfiles***: Hier kann man die gewünschte Anzahl der Outputfiles angeben.

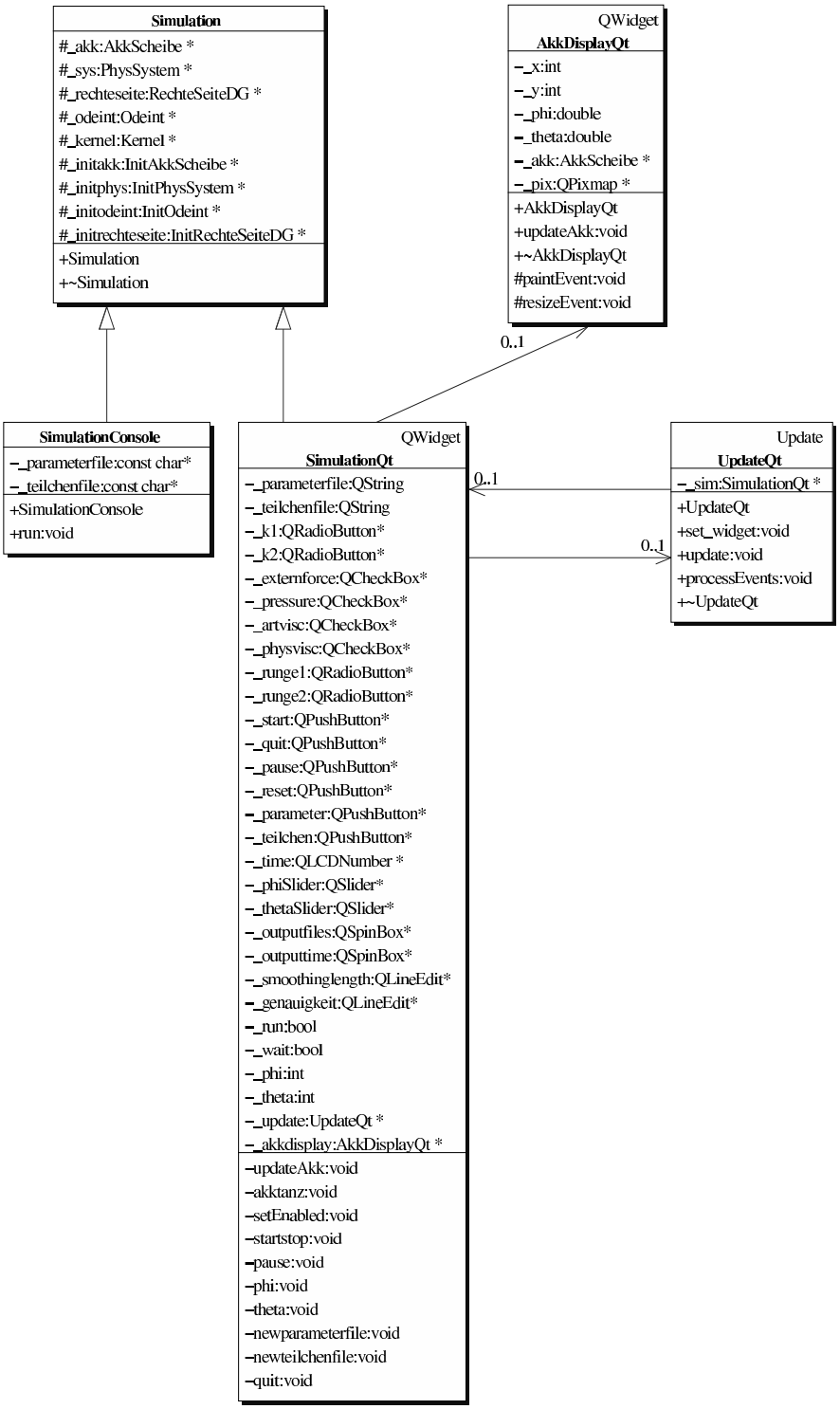


Abbildung 4.23: Das Paket der Simulationen in der Phase 3.

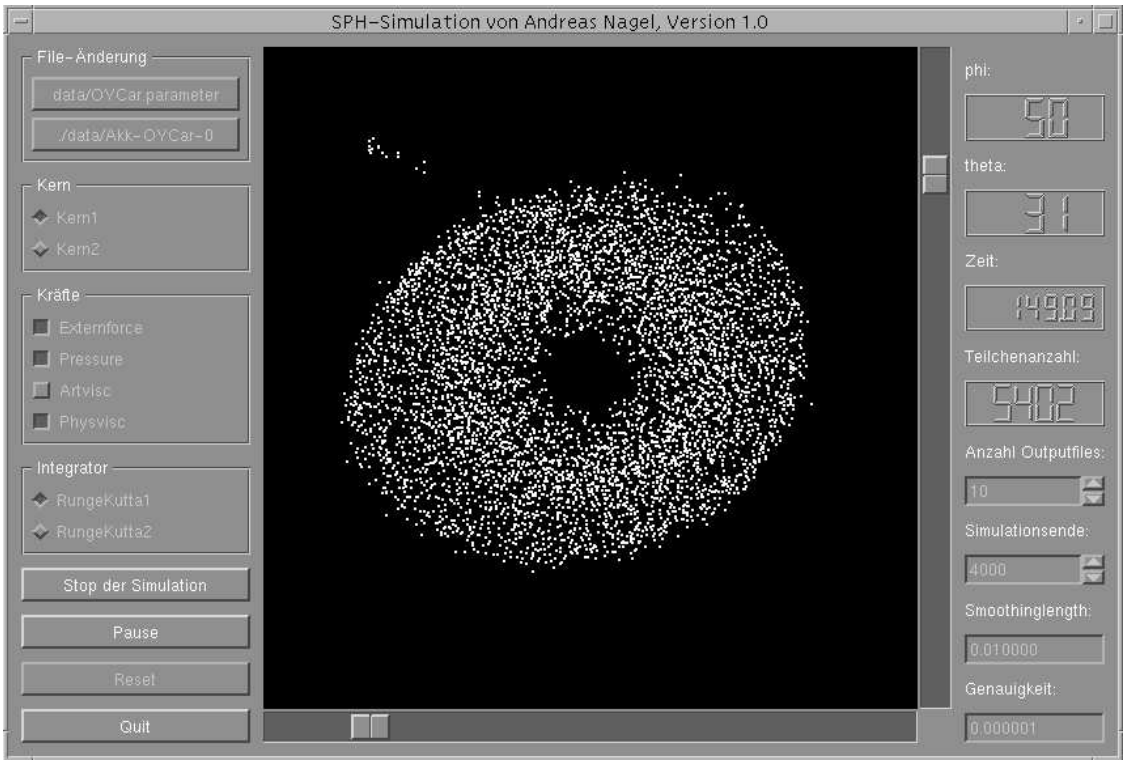


Abbildung 4.24: Das GUI von SPH++.

- ***Simulationsende***: Hier wird die gewünschte Endzeit der Simulation angegeben. Die zum Schluß ausgegebene Akkretionsscheibe kann natürlich wieder als Anfang einer neuen Simulation ausgewählt werden.
- ***Smoothinglength***: Hier kann man die gewünschte Smoothinglength auswählen. Da die Nachbarschaftssuche bis jetzt nur für eine Smoothinglength um die 0.01 gut funktioniert, sollte man diesen Wert nicht all zu stark verändern.
- ***Genauigkeit***: Hier wird die gewünschte Genauigkeit des Integrators angegeben.

Kapitel 5

Test von SPH++

Ein Arbeitsfluß des Software-Entwicklungsprozesses, der besonders am Ende einer Produktentwicklung eine große Rolle spielt, ist der ***Test*** (siehe Abb. C.5) des Produktes. Er besteht hauptsächlich aus der Überprüfung der am Anfang der Produktentwicklung geforderten Anforderungen bzw. der Anwendungsfälle.

Die Anforderungen des SPH++-Projektes wurden schon in der Einleitung genannt, nämlich daß die Funktionalität die gleiche wie beim C-Programm sein soll, gepaart mit den Vorteilen der Objektorientierung und von Designpatterns. Auch die Anwendungsfälle von SPH++ sind die gleichen wie die des C-Programms, nämlich die Berechnung der Simulation mit unterschiedlichen Ausgangsvariablen. Solch eine Auswahl der Ergebnisse einiger Simulationsläufe wird im folgenden Abschnitt 5.1 gezeigt.

Eine nichtfunktionale Anforderung an ein Programm ist die möglichst geringe Laufzeit des Programmes, d.h. es soll möglichst schnell sein. Dies gilt natürlich besonders für Simulationsprogramme wie z.B. SPH++, das ja in einer späteren Version auch auf parallelen Rechnern laufen soll. Deshalb wird im Abschnitt 5.2 die Geschwindigkeit von SPH++ mit dem C-Programm verglichen und im Abschnitt 5.3 die Stellen von SPH++ durch das Benutzen eines Profilers aufgezeigt, wo die meiste Rechenzeit verbraucht wird.

5.1 Berechnung von Akkretionsscheiben

In diesem Abschnitt werden einige mit SPH++ bzw. dem C-Programm berechneten Akkretionsscheiben gezeigt. Die Ausgangs-Akkretionscheibe zum Zeitpunkt 0 aller in diesem Abschnitt beschriebenen Simulationsläufe ist in Abbildung 5.1 zu sehen und stammt aus dem Paket des C-Programmes. Eine solche Ausgangs-Akkretionscheibe wird durch die Einfütterung von SPH-Teilchen in eine leere Ausgangskonfiguration mit einer konstanten Massenüberstromrate erzeugt bis sich die gewünschte Scheibe ausgebildet hat.

Auch die Konfigurationsdatei wurde für die Tests von SPH++ von dem vorgegebenen C-Programm weitgehend übernommen. In der Abbildung 5.2 sieht man auf der linken Seite die Konfigurationsdatei „parameter.config“ des C-Programms, die für die Simulation in der Abbildung 5.8 verwendet wurde. Auf der rechten Seite der Abbildung 5.2 ist die Konfigurationsdatei „OYCar.parameter“ von SPH++ angegeben, die für die Simulation der Abbildungen 5.3 bis 5.7 (nur Änderungen in der Auswahl der Kräfte) verwendet wurde. Die Parameter des Systems beschreiben dabei die Kataklysmische Variable „OYCar“, dessen Massenverhältnis $q = 0.1$ ist.

Zusätzliche Konfigurationsvariablen von SPH++ sind für die Auswahl des Kernels, des Inte-

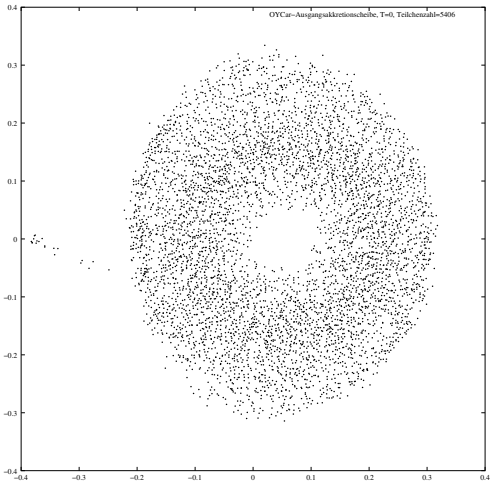


Abbildung 5.1: Die Akkretionsscheibe zum Zeitpunkt t=0.

C

```
#
# Parameter file für 3d mit sekundaerstern 1.1
#

# Das System:
Primaersternmasse 0.696
Sekundaersternmasse 0.069
Bahnperiode 5.4536544e3

# Physikalische Parameter
Viskositaet 3.0e-8
ArtVisk_alpha 0.1
ArtVisk_beta 0.2
Teilchenmasse 1.72815407e-17

#Numerische Parameter
Smoothinglength 0.01
Max_Teilchen 10000
Rate 0
MinAbstand 0.05
MaxAbstand 2.0

# Beginn und Ende der Simulation:
Eingabedatei 0
Outputfiles 100
Outputtime 5.4536544e3

# Die Integrationsparameter:
Anfangsschrittweite 2.5e0
MinSchrittweite 1.e-2
MaxSchritte 100000
Genauigkeit 1.0e-6

#Parallel Parameter
Grainsize 1

# DTS Thread Call Policy
#DTS_THREADTYPE DTS_LAZY
```

C++

```
#
# Parameter file für 3d mit sekundaerstern 1.1
#

# Das System:
Primaersternmasse 0.696
Sekundaersternmasse 0.069
Bahnperiode 5.4536544e3

# Physikalische Parameter
Viskositaet 3.0e-8
ArtVisk_alpha 0.1
ArtVisk_beta 0.2
Teilchenmasse 1.72815407e-17

#Numerische Parameter
Smoothinglength 0.01
Max_Teilchen 10000
Rate 0
MinAbstand 0.05
MaxAbstand 2.0

#Wahl des Kernels
Kern1

# Beginn und Ende der Simulation:
Eingabedatei
Eingabezeit 0.0
Outputfiles 100
Outputtime 5.4536544e5

# Die Integrationsparameter:
Anfangsschrittweite 2.5e0
MinSchrittweite 1.e-2
MaxSchritte 100000
Genauigkeit 1.0e-6
RungeKutta

# Kraefte fuer die rechte Seite der DG
Externforce
Pressure
Artvisc
Physvisc
```

Abbildung 5.2: Konfigurationsdateien des C- bzw. C++-Programms.

grators und der Kräfte eingeführt worden. Diese können während der Laufzeit des Programms jedoch verändert werden und stellen deshalb Default-Werte dar.

Durch die Konfigurationsvariablen „Outputfiles“ und „Outputtime“ wird bestimmt, wann Ausgabefiles erzeugt werden sollen. Dabei entspricht die Variablen „Outputtime“ bei dem C-Programm dem Zeitunterschied zwischen aufeinenderfolgenden Files, wohingegen sie bei SPH++ dem Endzeitpunkt der Simulation bestimmt.

Da die Einfütterung von SPH-Teilchen von SPH++ noch nicht möglich ist :-(, ist die Einfütterungsrate des C-Programms auch auf 0 gesetzt worden. Bei langen Simulationsläufen verringert sich deshalb die in der Simulation vorhandenen Teilchen, da solche, die außerhalb des Simulationsbereichs geraten, vernichtet werden.

Die Bahnperiode der Kataklysmische Variable „OYCar“ ist 5453.6544 sec, d.h. ca. 1,5 Stunden. Innerhalb dieser Zeit rotiert ein am äußeren Rand der Akkretionsscheibe gelegenes SPH-Teilchen ca. 2-Mal um den Weißen Zwerg (im mitrotierenden System) und ein am inneren Rand befindliches Teilchen ca. 30-Mal. Der Endzeitpunkt der im folgenden gezeigten Simulationen beträgt 100 Bahnperioden, dies entspricht ca. 6,3 Tage. Um sich eine Vorstellung über die notwendige Rechenzeit dieser Simulationen zu machen, sind diese in Tabelle 5.1 aufgeführt. Dies zeigt auch den Wunsch nach schnelleren Rechnern und der Parallelisierung von SPH++.

Abb.	In der Simulation verwendete Kräfte	Steps[a]	killed[b]	Zeit in sec.	Zeit/Step
5.3	Grav.	25612	400	24224.08	0.94
5.4	Grav.+Pressure	26004	1565	75415.73	2.90
5.5	Grav.+Pressure+Artvisc	27557	2008	100194.05	3.64
5.6	Grav.+Pressure+Physvisc	27594	2365	124822.23	4.52
5.7	Grav.+Pressure+Artvisc+Physvisc	29341	4108	123427.78	4.21
5.8	Grav.+Pressure+Artvisc+Physvisc	29398	4120	47150.56	1.60

Tabelle 5.1: Die Laufzeiten der in diesem Abschnitt gezeigten Simulationen auf einem Celeron(416Mhz).

[a]Die Steps sind effektive Steps, d.h. verworfene Steps (zur Korrektur der Schrittweite) sind darin nicht enthalten. Jedoch verbrauchen verworfene Steps fast genau soviel Rechenzeit wie normale Steps !

[b]Die Anzahl der aus dem Simulationsbereich gewanderten SPH-Teilchen.

Nun noch einige Anmerkungen zu den Abbildungen 5.3 bis 5.8:

- ***Abb. 5.3***: Da auf die SPH-Teilchen nur die Gravitationskraft wirkt, müssen deren Bahnen Kreise (oder genauer: Ellipsen) entsprechen. Mit Hilfe der GUI (siehe Abb. 4.24) kann man sich von der richtigen Implementierung dieser Kraft überzeugen. Daß sich am Ende der Simulation (unphysikalische ?) Ringe ausbilden, liegt entweder an SPH++ (Rundungsfehler,...) selber oder was wahrscheinlicher ist an der Ausgangs-Akkretionsscheibe (siehe Abb. 5.1), dessen Struktur sich während der Simulation verstärkt.

- ***Abb. 5.4***: Die zur Gravitation dazugekommene innere Druckkraft hat eine ähnliche Wirkung wie die Punktverwaschungsfunktion in der Optik. Im Gegensatz zur Simulation in Abb. 5.3 werden vorhandene Strukturen verwaschen bzw. gemittelt, was zur Folge hat, daß sich während der Simulation die Struktur der Akkretionsscheibe sich kaum verändert.

- ***Abb. 5.5***: Bei der Simulation dieser Abbildung wirkt auf jedes Teilchen zusätzlich zur Abb. 5.4 die künstliche Viskositätskraft. Wie in Kapitel 2.4 schon gesagt, wurde die künstliche

Viskosität eingeführt, um die numerische Auflösung von Schocks zu verbessern. Diese Viskosität ist in dieser Simulation schwächer als die physikalische Viskosität. Dies kann man durch den Vergleich mit Abb. 5.6 sehen bzw. auch durch die Anzahl der gekillten Teilchen in Tabelle 5.1.

- ***Abb. 5.6***: Statt der künstlichen Viskositätskraft wie in Abb. 5.5 wirkt hier die physikalische Viskositätskraft.

- ***Abb. 5.7 und Abb. 5.8***: Die Abbildung 5.7 zeigt die Simulation von SPH++, bei der alle Kräfte auf die SPH-Teilchen wirken. Die analoge Simulation des C-Programms ist in Abbildung 5.8 zu sehen. Wenn man die Abbildungen miteinander vergleicht, so erkennt man, daß sie nahezu identische Ergebnisse liefern, wie es ja auch sein soll. Allgemein nimmt die Viskosität in Richtung Scheibenmitte zu, da dort die Geschwindigkeitsgradienten größer werden. Dies bewirkt, daß umso näher sich die Teilchen dem Weißen Zwerg nähern, desto stärker werden sie abgebremst. Als Folge davon erhöht sich ihre Radialgeschwindigkeit, bis sie vom Weißen Zwerg akkretiert werden.

Zum Schluß dieses Abschnittes soll noch einmal auf die Wichtigkeit von Testläufen hingewiesen werden. Als begonnen wurde SPH++ zu testen und mit dem C-Programm zu vergleichen, ergab sich folgende Ausgangslage: Das C-Programm lief mit der Einstellung „double“ nur auf SPARC-Rechnern, während SPH++ das Problem hatte, daß es nach einiger Zeit mit einem Speicherfehler stehenblieb. Auch unterschieden sich die Ergebnisse der beiden Programme völlig. Da das C-Programm eigentlich die Referenz für SPH++ ist und die Entwicklung von SPH++ auf einer Intel-Maschine stattfand, mußte zuerst untersucht werden, warum das C-Programm auf dem Intel-Rechner Probleme hat. Durch Debuggerläufe des C-Programms wurden dann einige Fehler behoben, die zu völlig falschen Ergebnissen der künstlichen und physikalischen Viskosität geführt hatten (auf der beiliegenden CD befindet sich einerseits das ursprüngliche C-Programm „sph.tar.gz“ und andererseits die korrigierte Fassung „sph_2000.tar.gz“). Nach der Fehlerbeseitigung lief das C-Programm auch auf der Intel-basierten Maschine mit der Einstellung „double“, sodaß als nächstes die Berechnung der einzelnen Kräfte der beiden Programme verglichen werden konnte. Dabei wurde noch ein Fehler in SPH++ entdeckt, der ohne den Vergleich mit dem C-Programm unentdeckt geblieben worden wäre. Zum Schluß wurde auch noch der obige Fehler in SPH++ entdeckt und beseitigt, der nach längeren Durchläufen zu einem Speicherfehler geführt hatte.

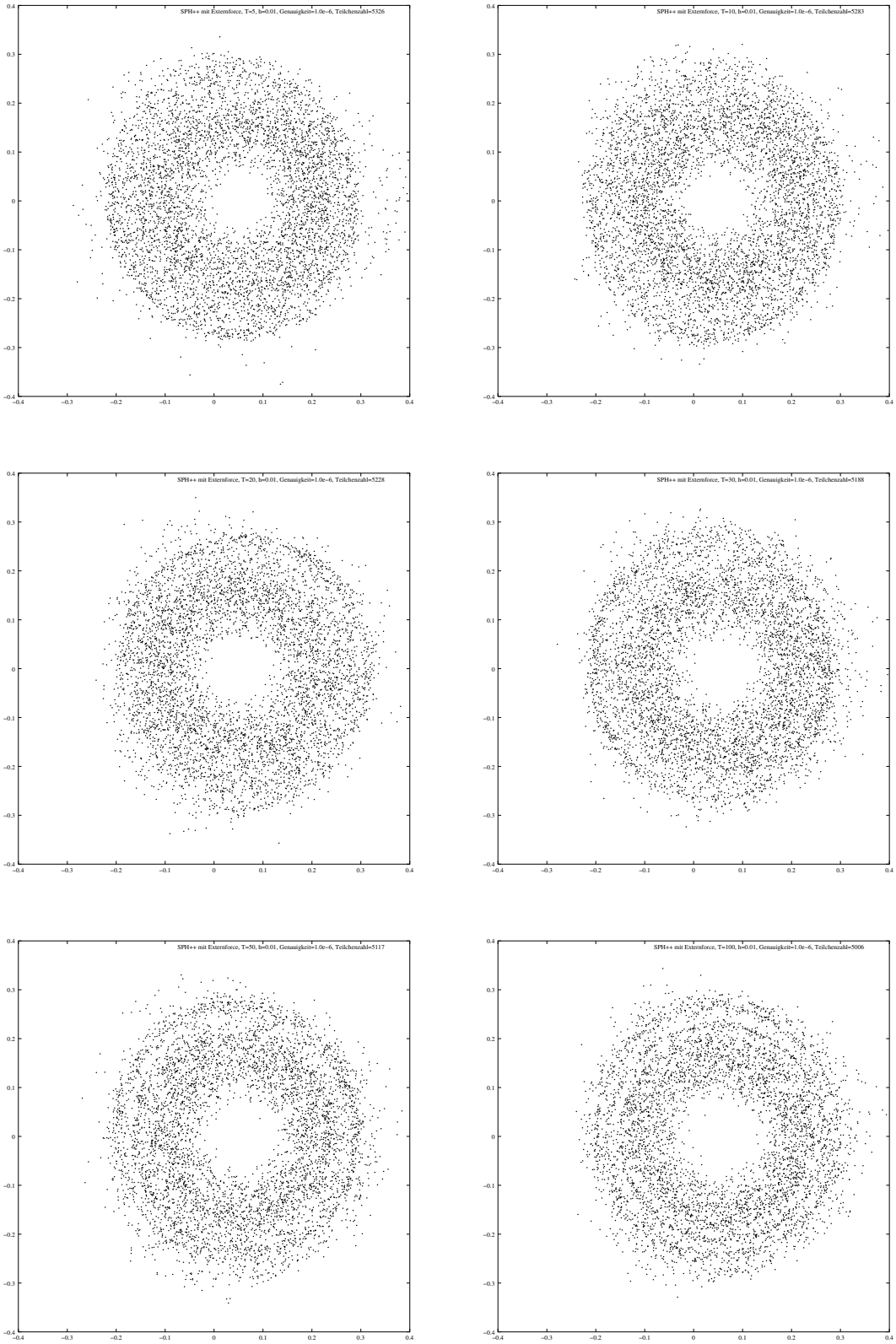


Abbildung 5.3: Mit SPH++ berechnete Akkretionsscheiben unter der Einwirkung der Gravitationskraft.

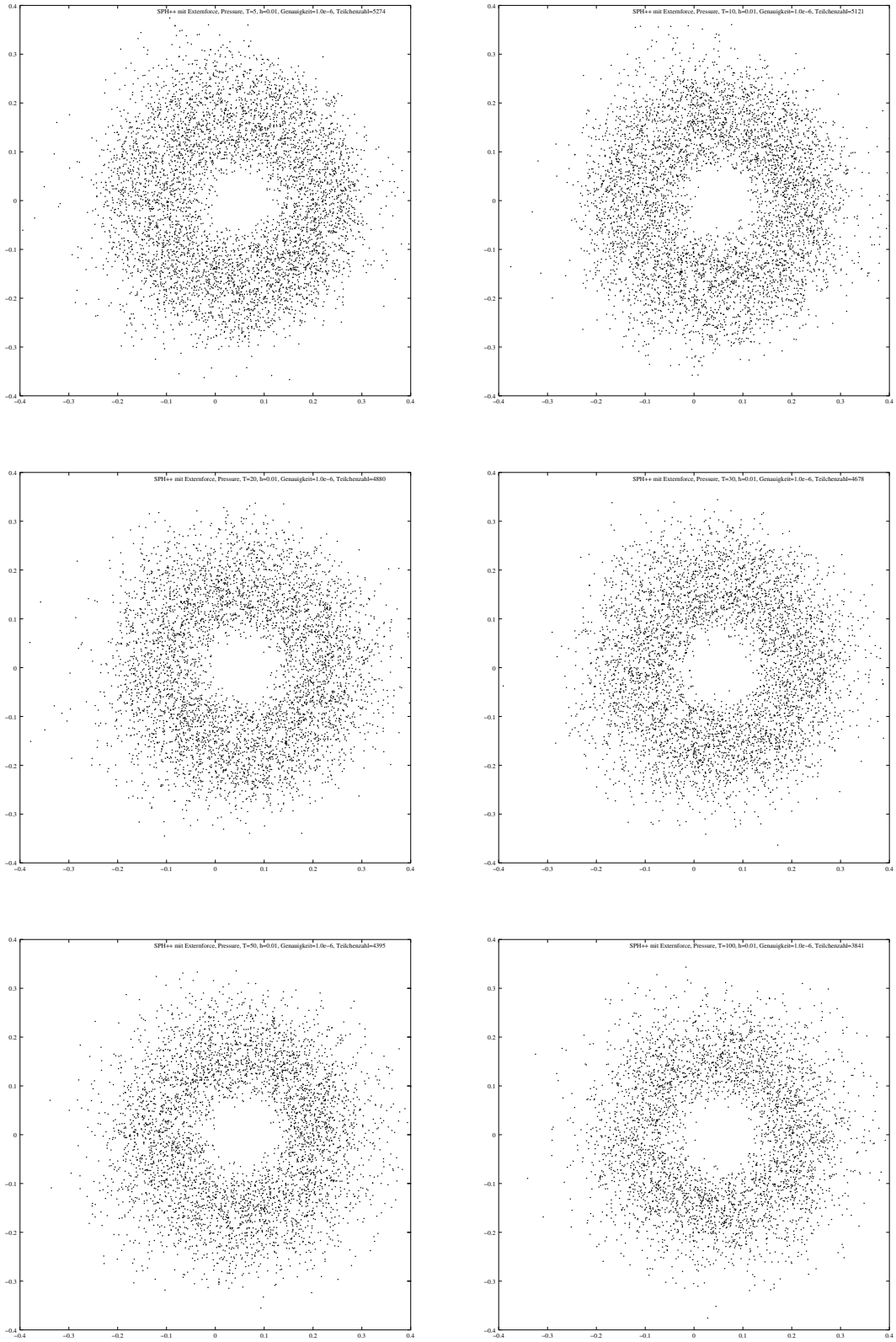


Abbildung 5.4: Mit SPH++ berechnete Akkretionsscheiben unter der Einwirkung der Gravitations- und der Druckkraft.

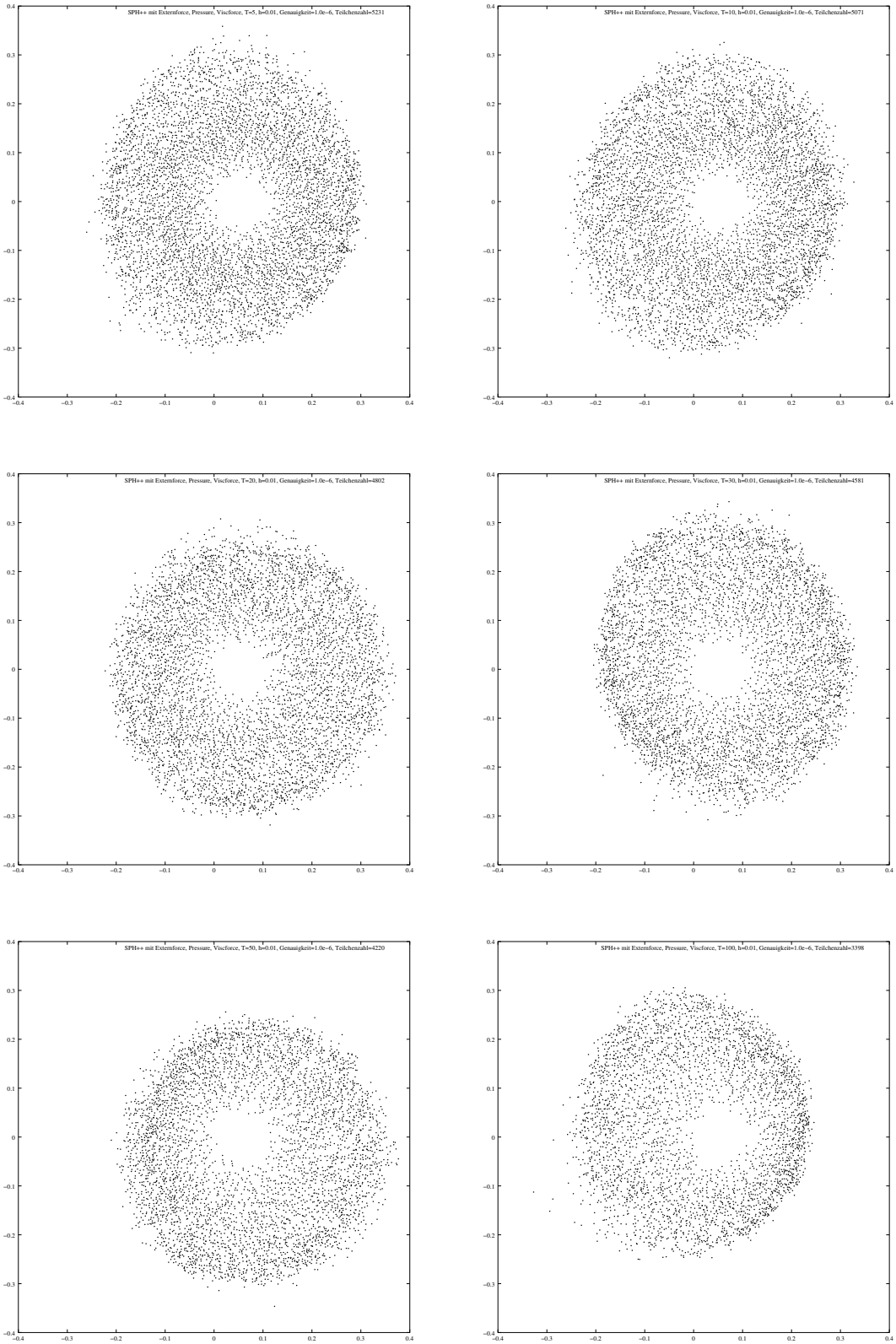


Abbildung 5.5: Mit SPH++ berechnete Akkretionsscheiben unter der Einwirkung der Gravitations-, der Druck- und der künstlichen Viskositätskraft.

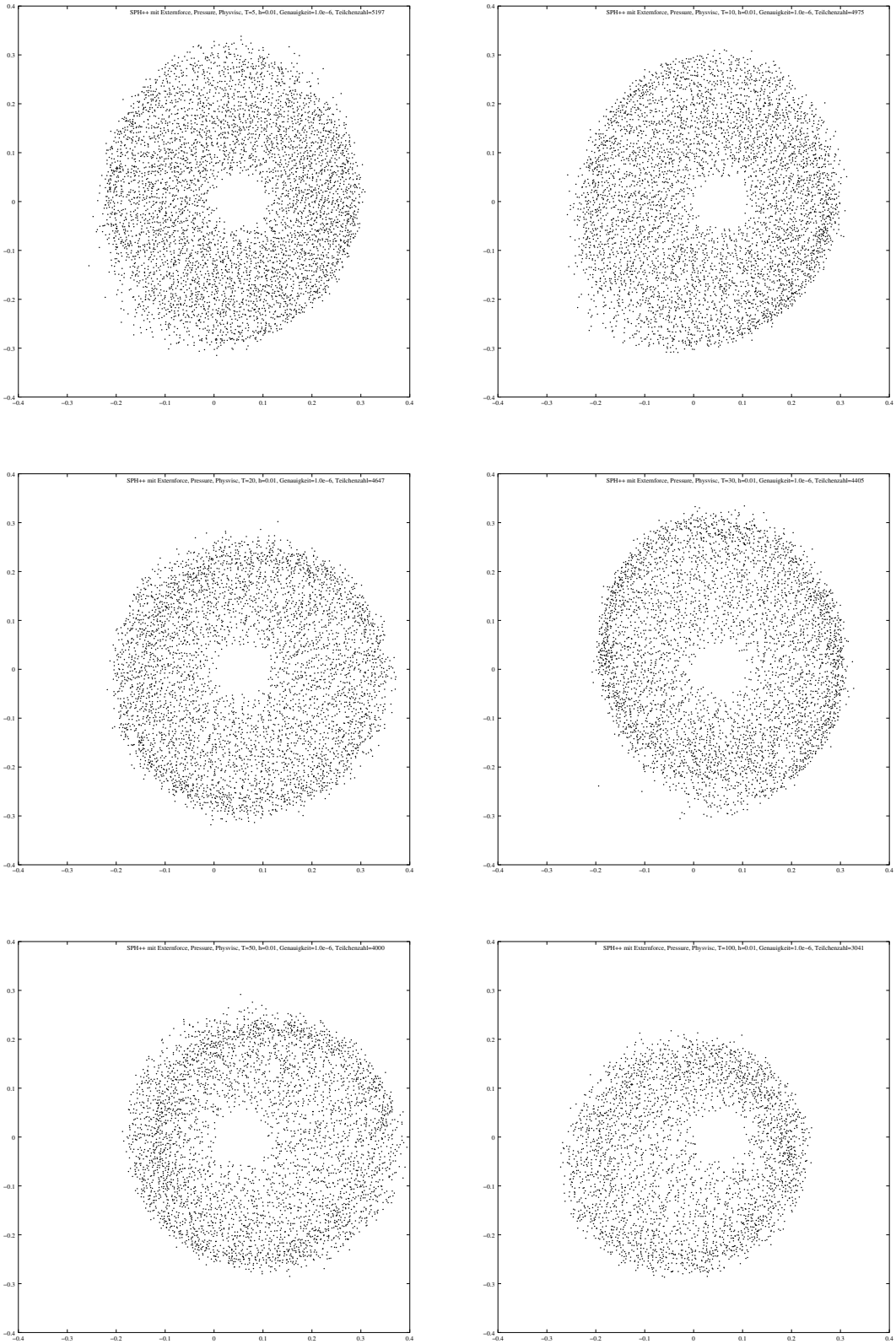


Abbildung 5.6: Mit SPH++ berechnete Akkretionsscheiben unter der Einwirkung der Gravitations-, der Druck- und der physikalischen Viskositätskraft.

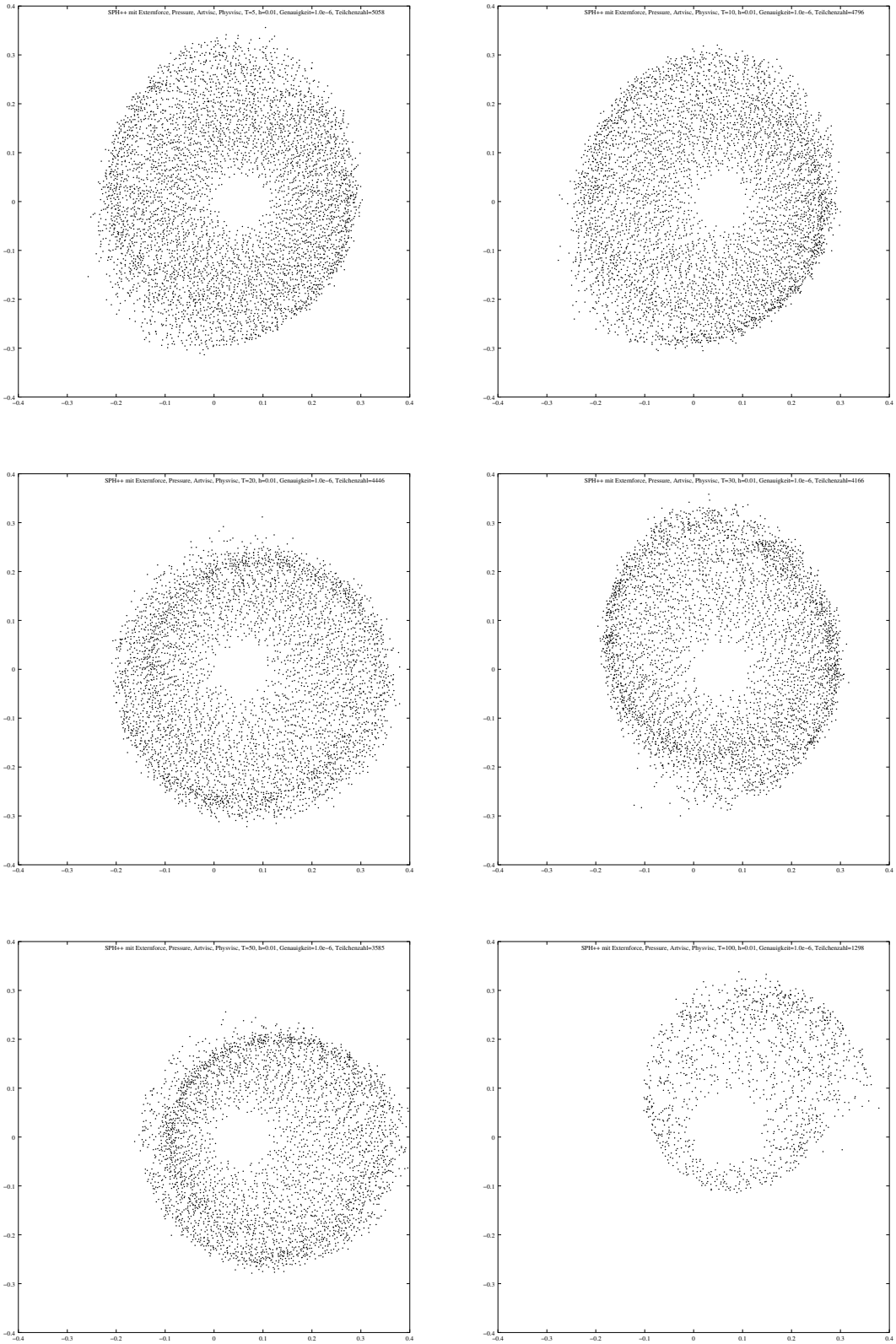


Abbildung 5.7: Mit SPH++ berechnete Akkretionsscheiben unter der Einwirkung der Gravitations-, der Druck-, der künstlichen Viskositäts- und der physikalischen Viskositätskraft.

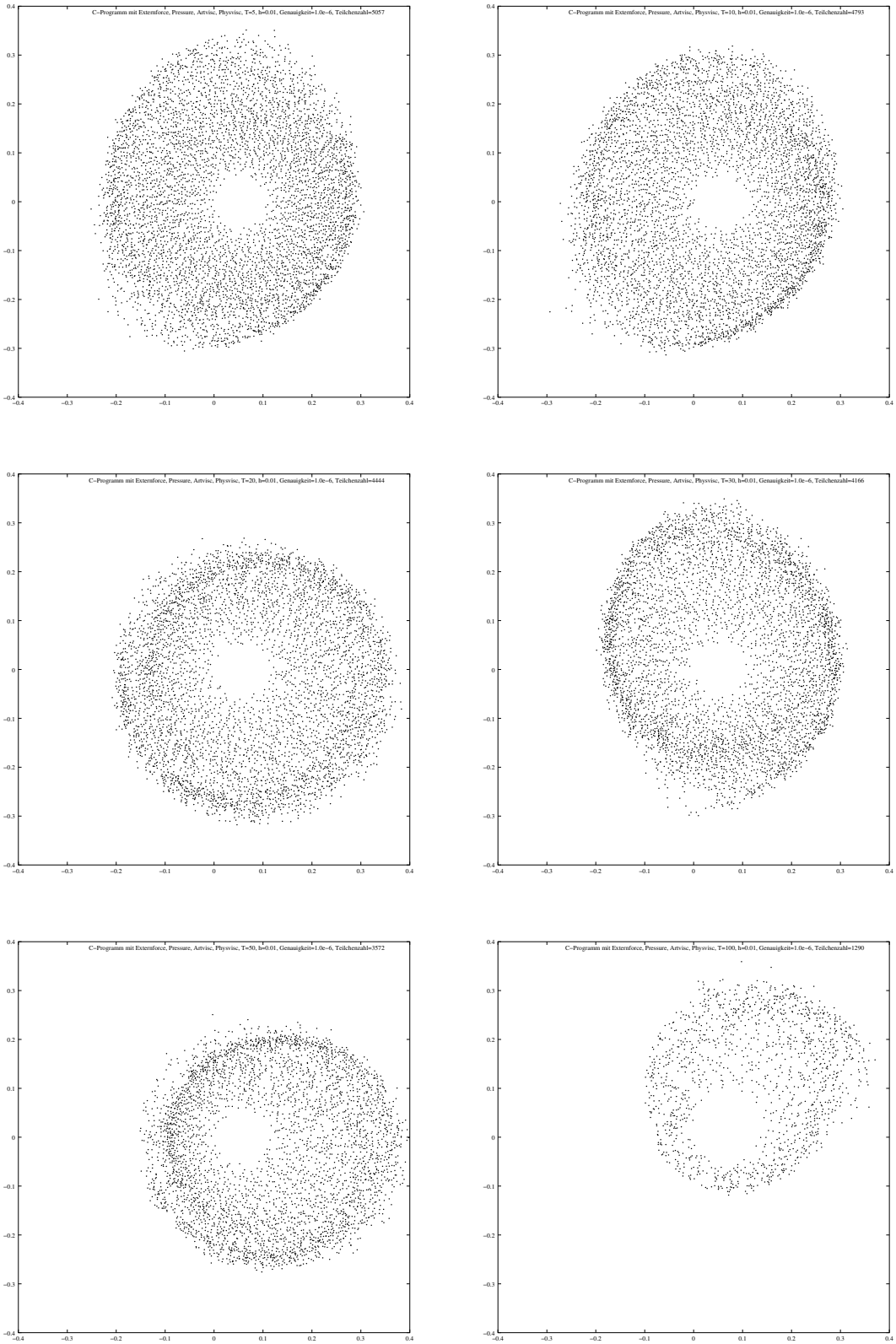


Abbildung 5.8: Die mit dem C-Programm berechnete Akkretionsscheiben unter der Einwirkung der Gravitations-, der Druck-, der künstlichen Viskositäts- und der physikalischen Viskositätskraft.

5.2 Laufzeitmessungen

Da SPH++ nun vernünftige Werte liefert (siehe vorigen Abschnitt), kann man sich nun der Frage zuwenden, wie effizient bzw. wie hoch die Geschwindigkeit von SPH++ ist. Erste Antworten bekommt man durch Laufzeitvergleiche zwischen SPH++ und dem C-Programm. In Tabelle 5.2 sind solche Laufzeitmessungen angegeben. Die Simulationsparameter sind die gleichen wie in Abb. 5.2, mit der Außnahme, daß „Outputfiles=1“ und „Outputtime=100“ gesetzt wurde.

Wie man aus der Tabelle 5.2 erkennen kann, ist SPH++ im Vergleich zum C-Programm mehr als doppelt so langsam. Um die Gründe dafür herauszufinden, wurden folgende Parameter variiert:

- ***Smoothinglength:*** Die Smoothinglength hat einerseits einen direkten Einfluß auf die Anzahl der Wechselwirkungspartner und andererseits ist sie gleich der Breite einer Gitterzelle. In dieses Gitter werden vor der Bestimmung der Wechselwirkungspartner die SPH-Teilchen eingeordnet, wie es in Abschnitt 4.3.1.3 beschrieben wird. Aus der Tabelle kann man erkennen, daß je kleiner h ist, desto langsamer wird SPH++ gegenüber dem C-Programm. Das deutet darauf hin, daß die Wechselwirkungssuche einer der Schwachstellen von SPH++ darstellt, wobei besonders die Kopplung von h an die Breite der Gitterzelle für kleine h‘s (d.h. viele Gitterelemente) viel Rechenzeit kostet.
- ***Teilchenanzahl:*** Je kleiner die Teilchenanzahl, desto stärker ist der Einfluß der Smoothinglength auf den Performance-Unterschied zwischen SPH++ und dem C-Programm.
- ***Unterschiedliche Architekturen:*** Die Laufzeitmessungen der Tabelle 5.2 sind auf folgenden Rechnern durchgeführt worden, um den Einfluß verschiedener Rechnerarchitekturen auf das Laufzeitverhalten von SPH++ zu untersuchen:
  - Ultra-Sparc 10/300MHz (256 kB 2-Level Cache off Chip) und 640 MByte Hauptspeicher mit Solaris7.
  - Intel Celeron/416MHz (128 kB 2-Level Cache on die) und 320 MByte Hauptspeicher mit Solaris8.
  - Intel PentiumIII/733MHz (256 kB 2-Level Cache on die) mit Suse-Linux 7.0.

  Wie man in der letzten Zeile der Tabelle 5.2 erkennen kann, läuft SPH++ im Verhältnis zum C-Programm auf der Ultra-Sparc-Maschine bei 5406 Teilchen ca. 50% langsamer als auf den beiden Intel-Maschinen. Da der 2-Level Cache der Sparc extern ist und deshalb nicht mit vollem Prozessortakt wie bei den 2 Intel-Prozessoren läuft, kann man vermuten, daß SPH++ starken Gebrauch vom 2-Level-Cache macht. Einer der Gründe ist die bei SPH++ nicht am Anfang festgelegte Speichermenge, in der die Daten der Teilchen bzw. der Wechselwirkungsteilchen abgelegt werden, wie das beim C-Programm der Fall ist, sondern die dynamische Erzeugung und Vernichtung dieser Objekte. Daß diese Vermutung richtig ist, wird im nächsten Abschnitt bestätigt.

Da weiter oben schon Anzeichen aufgetaucht sind, daß die schlechte Performance von SPH++ auch von der schlechten Implementierung der Wechselwirkungssuche herrührt, wurde Laufzeitmessungen von SPH++ mit verschiedenen Kräftekombinationen durchgeführt. Dies ist in Tabelle 5.3 aufgeführt. Wird nur die Gravitationskraft ausgewählt, so berechnet SPH++ ***keine*** Wechselwirkungsteilchen. Bei allen anderen Kräftekombinationen werden Wechselwirkungsteilchen benötigt. Wenn man zur Gravitation noch die innere Druckkraft hinzunimmt, so steigt die Laufzeit laut Tabelle um über 400%. Da die Druckkraft-Berechnung selber nicht viel Rechenzeit kosten kann, benötigt die Erzeugung,Vernichtung und Suche der Wechselwirkungsobjekt ca. 3 Sekunden pro Step. Dies entspricht fast der Hälte der Rechenzeit die SPH++ für die Berechnung aller Kräfte benötigt und der kompletten Rechenzeit des C-Programmes. Im nächsten Abschnitt wird mit Hilfe eines Profilers genauer untersucht, wo die meiste Rechenzeit von SPH++ verloren geht.

Programm	h	Teil.zahl[a]	WWP.[b]	Steps[c]	Arch.	Zeit	Zeit/Step	LV[d]
C-Prog.	0.005	5406 (-3)	2369	17(+3)	Sparc	40.57	2.03	1
"	"	"	"	"	Celeron	35.85	1.79	"
"	"	"	"	"	Pentium	20.25	1.01	"
SPH++	"	"	"	"	Sparc	180.79	9.04	4.46
"	"	"	"	"	Celeron	110.54	5.53	3.08
"	"	"	"	"	Pentium	68.28	3.41	3.37
C-Prog.	0.01	5406 (-4)	16433	12	Sparc	42.26	3.52	1
"	"	"	"	"	Celeron	37.65	3.14	"
"	"	"	"	"	Pentium	22.83	1.90	"
SPH++	"	"	"	"	Sparc	141.87	11.82	3.36
"	"	"	"	"	Celeron	86.58	7.21	2.30
"	"	"	"	"	Pentium	47.28	3.94	2.07
C-Prog.	0.02	5406 (-9)	74090	9	Sparc	99.86	11.10	1
"	"	"	"	"	Celeron	92.96	10.33	"
"	"	"	"	"	Pentium	56.31	6.26	"
SPH++	"	"	"	"	Sparc	355.62	39.51	3.56
"	"	"	"	"	Celeron	217.45	24.16	2.34
"	"	"	"	"	Pentium	123.74	13.75	2.20
C-Prog.	0.005	1000	53	6	Sparc	1.86	0.31	1
"	"	"	"	"	Celeron	1.36	0.03	"
"	"	"	"	"	Pentium	0.62	0.10	"
SPH++	"	"	"	"	Sparc	33.93	5.66	18.25
"	"	"	"	"	Celeron	20.55	3.42	15.11
"	"	"	"	"	Pentium	13.15	2.19	21.20
C-Prog.	0.01	1000	556	6	Sparc	2.25	0.37	1
"	"	"	"	"	Celeron	1.55	0.26	"
"	"	"	"	"	Pentium	0.77	0.13	"
SPH++	"	"	"	"	Sparc	8.51	1.42	3.78
"	"	"	"	"	Celeron	5.32	0.89	3.43
"	"	"	"	"	Pentium	3.19	0.53	4.14
C-Prog.	0.02	1000	2576	6	Sparc	3.83	0.64	1
"	"	"	"	"	Celeron	3.01	0.50	"
"	"	"	"	"	Pentium	1.60	0.27	"
SPH++	"	"	"	"	Sparc	10.95	1.82	2.86
"	"	"	"	"	Celeron	6.94	1.16	2.31
"	"	"	"	"	Pentium	3.70	0.62	2.31

Tabelle 5.2: Es werden die Laufzeiten des C-Programms bzw. von SPH++ in verschiedenen Konfigurationen angegeben. Die Integrationszeit ist dabei 100 sec. bei einer Genauigkeit von 10^{-6}.

[a] In der Klammer steht die Anzahl der während der Simulation vernichteten Teilchen.
[b] Anzahl der Wechselwirkungspartner beim Anfang des ersten Integrationsschrittes.
[c] In der Klammer steht die Anzahl der verworfenen Integrationsschritte.
[d] Laufzeitverlängerung von SPH++ gegenüber dem C-Programm.

Programm	In der Simulation verwendete Kräfte	Steps	Zeit	Zeit/Step
SPH++	Grav.	7	7.35	1.05
”	Grav.+Pressure	7	31.20	4.46
”	Grav.+Pressure+Artvisc	12	58.70	4.89
”	Grav.+Pressure+Physvisc	7	47.44	6.78
”	Grav.+Pressure+Artvisc+Physvisc	12	86.58	7.21
C-Prog.	Grav.+Pressure+Artvisc+Physvisc	12	37.65	3.14

Tabelle 5.3: Die Laufzeiten von SPH++ mit unterschiedlichen Kräften. Die Integrationszeit ist dabei 100 sec. bei einer Genauigkeit von 10^{-6}.

5.3 Profilemessungen

Im vorigen Abschnitt wurde empirisch festgestellt, daß bei der Wechselwirkungssuche und bei der dynamischen Erzeugung und Vernichtung von Objekten ein Großteil der Geschwindigkeit von SPH++ verloren geht. Mit einem Profiler kann man dies näher untersuchen. Als Profiler wurde Quantify auf Sparc/Solaris von der Firma Rational benutzt. In Abb. 5.9 sind die 20 Funktionen von SPH++ aufgeführt, die prozentual am meisten Rechenzeit benötigt haben. Die Konfigurationsvariablen sind dabei die gleichen wie in Abb. 5.2, wobei alle Kräfte ausgewählt wurden. Abb. 5.10 enthält diejenigen 20 Funktionen, dessen Laufzeit plus der Laufzeiten der von ihnen aufgerufenen Funktionen am größten ist. Mit Quantify kann man sich das Profile-Ergebnis auch graphisch darstellen lassen, wie man in Abb. 5.11 sehen kann. Dabei ist die Dicke der Linien proportional der verbrauchten Rechenzeit der Funktionen plus ihrer aufgerufenen Funktionen.

14.36%	_createww
8.96%	_free_unlocked
5.38%	_malloc_unlocked
4.46%	mutex_lock
4.46%	mutex_unlock
3.87%	__builtin_new
3.65%	realfree
3.62%	cleanfree
3.10%	malloc
3.07%	RungeKutta::rkck(AkkScheibe&,double,PhysSystem&,RechteSeiteDG&)
2.37%	free
2.28%	_init
2.26%	_add_press
1.98%	_return_zero
1.96%	_smalloc
1.90%	__builtin_vec_delete
1.85%	__valarray_copy
1.81%	Externforce::deriv_ohne_init(AkkScheibe&,PhysSystem&)
1.81%	__builtin_vec_new
1.74%	pow

Abbildung 5.9: Quantify: Top 20 functions that match '*', Function time (% of .root.).

Um herauszufinden, wo SPH++ die meisten Rechenzeit benötigt, ist Abb. 5.10 bzw. die analoge graphische Abb. 5.11 am Besten geeignet. Daß die Funktion „main“ (plus der abgeleiteten Funktionen) 100% der Rechenzeit benötigt, ist klar. Jedoch ist erstaunlich, daß die Funktion

100.00%	.root.
100.00%	_init[libc.so.1/1853]
100.00%	_start
100.00%	main
100.00%	SimulationConsole::run(int,char**)
94.78%	Odeint::odeint(AkkScheibe&,PhysSystem&,RechteSeiteDG&)
84.09%	RechteSeiteDG::derivatives(AkkScheibe&,PhysSystem&)
78.39%	RungeKutta::stepper(AkkScheibe&,PhysSystem&,RechteSeiteDG&,double*,double*)
78.16%	RungeKutta::rkck(AkkScheibe&,double,PhysSystem&,RechteSeiteDG&)
50.44%	PhysviscAkkScheibe::force_init(PhysSystem&)
50.44%	_force_init
49.24%	_createww
33.65%	Kraefte::deriv_ohne_init(AkkScheibe&,PhysSystem&)
29.48%	__builtin_vec_new
27.67%	__builtin_new
23.29%	malloc
19.54%	__builtin_vec_delete
17.64%	free
15.30%	Physvisc::deriv_ohne_init(AkkScheibe&,PhysSystem&)
15.30%	PhysviscAkkScheibe::add_physvisc(PhysSystem&)

Abbildung 5.10: Quantify: Top 20 functions that match '*', Function+descendants time (% of .root.).

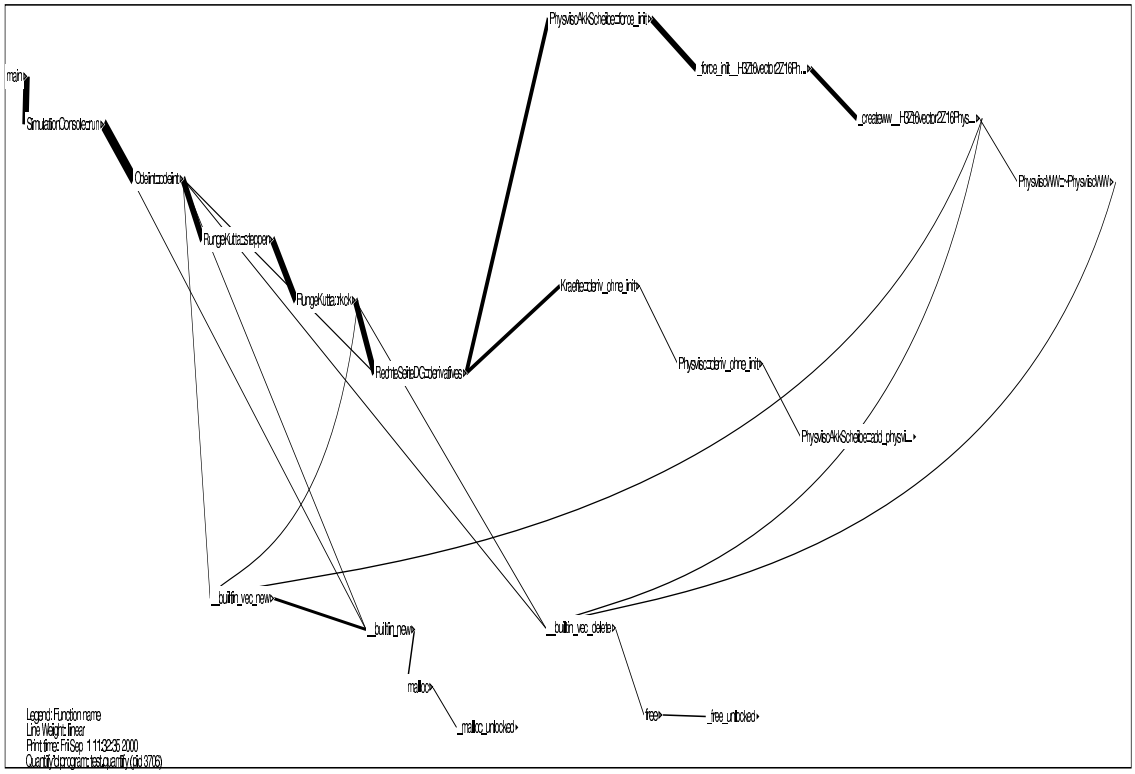


Abbildung 5.11: Die graphische Darstellung von Tabelle 5.10, generiert von Quantify.

„_createww“, d.h. die Wechselwirkungssuche plus der Erzeugung und Vernichtung der Wechselwirkungsobjekte, fast 50% der gesamten Rechenzeit verbraucht. Dieses Ergebnis bestätigt genau die im vorigen Abschnitt gemachte Abschätzung ! Viel Rechenzeit benötigen auch die Bibliotheksfunktionen „__builtin_vec_new“, „__builtin_new“, „malloc“, „__builtin_vec_delete“ und „free“, die für das Erzeugen und Vernichten von Objekten verantwortlich sind. Da diese Funktionen gemeinsame Unterfunktionen besitzen, kann man ihre Rechenzeit zwar nicht einfach aufaddieren, jedoch dürfte die Summe trotzdem über 50%Prozent liegen (wobei ca. 20% davon in _createww enthalten sind). In Abb. 5.9 kann man auch ablesen, daß die Bibliotheksfunktionen „_free_unlocked“, „_malloc_unlocked“, „mutex_lock“, „mutex_unlock“, „__builtin_new“, „realfree“, „cleanfree“ und „malloc“, die für die Erzeugung und Vernichtung von Objekten verantwortlich sind, zusammengenommen ca. 40% der Rechenzeit benötigen.

In Tabelle 5.12 wurde die verbrauchte Rechenzeit der Funktion „_createww“ in Abhängigkeit von der Smoothinglength h aufgetragen. Wie man erkennen kann, wird bei kleiner werdendem h fast die gesamte Rechenzeit durch die Funktion „_createww“ verbraucht, wohingegen auch bei größer werdendem h die von der Funktion benötigte Rechenzeit leicht ansteigt. Dies erklärt auch die Laufzeitmessungen, die im letzten Abschnitt in Tabelle 5.2 dargestellt sind.

h	WWPartner	Zeit	_createww	_createww (total)
0.002	142	376	92.82	95.38
0.005	2354	81	59.61	67.80
0.01	16425	143	14.36	49.24
0.02	74067	459	9.06	56.99
0.03	167948	964	8,83	59.24

Abbildung 5.12: 5406 Teilchen, Integrationszeit: 2.5.

Fazit : Wenn man das Laufzeitverhalten von SPH++ verbessern möchte, muß zuerst die Wechselwirkungssuche verbessert und die Erzeugung und Vernichtung der Wechselwirkungsobjekten minimiert werden. Um den Erfolg solcher Maßnahmen zu kontrollieren, ist es empfehlenswert, die Wechselwirkungssuche als eigene Klasse zu realisieren, wobei mittels dem Strategy-Pattern (siehe Anhang B.2.2) man leicht zusätzliche Wechselwirkungssuche-Algorithmen auswählen und testen kann.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Die vorigen Kapitel beschreiben die Entwicklung von SPH++ aus einem in C geschriebenes SPH-Programm von Stefan Hüttemann und Stefan Kunze. Dabei wurde bei der Entwicklung dieses C++-Programms ein großer Wert auf die gute Erweiterbarkeit, Wartung und Übersichtlichkeit gelegt, was durch die Verwendung von „Design-Pattern“ erreicht wurde. Ein weiterer Vorteil von SPH++ ist, daß zur Laufzeit die Konfiguration der Simulation verändert werden kann. Beispielsweise kann man während der Laufzeit einen anderen Kernel, einen anderen Integrator oder eine andere Kräftekombination auswählen. Um dies beim C-Programm zu erreichen, müssen im Sourcecode bestimmte Makros ein- bzw. auskommentiert und das Programm neu kompiliert werden.

Wenn man die Entwicklung dieser Software mit dem Unified Process vergleicht, so wurde der erste Zyklus der Software-Entwicklung beendet. Da nicht alle Anforderungen erfüllt wurden bzw. noch gar nicht bekannt sind, ist auf jeden Fall ein neuer Zyklus erforderlich, um diese restlichen Anforderungen zu erfüllen. Im nächsten Abschnitt wird darauf näher eingegangen.

Der Geschwindigkeitsvergleich von SPH++ mit dem C-Programm zeigt, daß SPH++ mehr als doppelt so langsam ist wie das C-Programm. Dies ist jedoch nicht auf die Programmiersprache C++ zurückzuführen, sondern auf die schlechte Implementierung der Wechselwirkungssuche und der häufigen Erzeugung und Vernichtung von Objekten.

In dieser Diplomarbeit wurden neue C++-Sprachmittel, wie valarray, Template-Funktionen oder RTTI (Run-Time Type Identification) angewendet, die teilweise vom gcc-Compiler 2.95.2 (noch) nicht fehlerfrei abgearbeitet werden.

6.2 Ausblick

SPH++ ist nun soweit gereift, daß es nahezu die gleichen Funktionalität und Stabilität wie das C-Programms besitzt. Jedoch sind während der Design- und besonders der Testphase einige Verbesserungsvorschläge aufgetaucht, die bei einem neuen Entwicklungszyklus von SPH++ berücksichtigt werden sollten:

- ***Weiterentwicklung des Entwicklungsprozesses***: Ein zukünftiger Entwicklungsprozeß von SPH++ sollte sich noch mehr in Richtung Unified Process entwickeln. Zusätzlich zu den Klassendiagrammen sollten dabei auch noch andere Diagramme (siehe Anhang A.2) eingesetzt werden.

- ***Parallelisierung von SPH++***: Um das Programm zu parallelisieren, muß es ganz neu strukturiert werden. Die Akkretionsscheibe muß in dieser neuen Struktur dabei in mehrere Gebiete zerteilt werden, die dann auf mehrere Prozessoren verteilt werden können. Zwischen den Gebieten können zur Berechnung der SPH-Kräfte statt Teilchenlisten Wechselwirkungsteilchen ausgetauscht werden. Die Vor- und Nachteile bei der Verwendung von Wechselwirkungsteilchen muß jedoch noch untersucht werden.

- ***Verbesserung des Designs von SPH++***:

  - ***Einfachere Erweiterung mit zusätzlichen SPH-Kräften***: Die Erweiterung von Kräften geschieht bis jetzt dadurch, daß eine neue Akkretionsscheibe von bestehenden abgeleitet wird. Da diese jedoch nur so von Template-Funktionen wimmeln, wäre es gut, wenn die neue Architektur des nächsten Zykluses weniger Template-Funktionen benutzen würde.
  - ***Die Trennung der Akkretionsscheibe von den Kräften soll aufgehoben werden***: Die Algorithmen der SPH-Kräfte wurden in die Akkretionsscheiben verlegt (Kapitel 4), da sich dort auch die Zwischenergebnisse der SPH-Kräfte befinden. Dadurch haben die Kräfte-Objekte nur noch die Aufgabe, die Ergebnisse aufzusummieren. Diese verbliebene Funktionalität der Kräfte-Objekte kann man jedoch auch ganz in die Akkretionsscheibe hinneinnehmen, was eher der physikalischen Realität entspricht, da zu jedem SPH-Teilchen eine Menge auf sie einwirkenden SPH-Kräften gehört. Ein weiterer Grund für diesen Schritt ist die Vermeidung des Problems, wie die Daten und Algorithmen zwischen der Akkretionsscheibe und den Kräften aufgeteilt werden sollen.
  - Die Wechselwirkungssuche soll mittels des Strategy-Patterns als eigene Klasse realisiert werden. Dadurch wird die Entwicklung von effizienten Algorithmen zur Wechselwirkungssuche vereinfacht.

- ***Einfütterung von Teilchen***: Dieses Feature kann eigentlich recht schnell in das Programm eingebunden werden. Dazu muß man jedoch die in den Akkretionscheiben enthaltenen „STL-Vektoren“ durch „STL-Listen“ ersetzen, da die Pointer, die jetzt noch auf die „STL-Vektoren“ zeigen, beim Wachstumsprozeß des Vektors irgendwann ins Nichts zeigen würden. Da jedoch eh ein neuer Zyklus notwendig ist, wurde diese Umgestaltung des Programms auf später verschoben.

- ***Verwendung eines besseren Compilers***: Zukünftig sollte ein Compiler benutzt werden, der dem ISO/ANSI C++ Standard[1] entspricht, der die neuen C++-Sprachmittel besser und stabiler unterstützt. Ein zukünftiger gcc (3.x) erfüllt diese Forderungen.

- ***Verwendung externer Bibliotheken***: Andere Bibliotheken, die auf Simulationen, lineare Algebra u.ä. optimiert sind (z.B. BLITZ++[2]), sollten statt eigener Lösungen verwendet werden.

[1] Dieser wurde 1998 verabschiedet.
[2] Siehe Homepage: http://www.oonumerics.org/blitz.

Anhang A

Überblick über die UML

A.1 Einführung

Mit Hilfe von Bildern oder von Modellen kann man sich die Struktur und die Wirkungsweise eines Programms besser veranschaulichen. Auch in anderen Bereichen, wie dem Maschinenbau oder dem Häuserbau hat sich die Erstellung von Plänen bzw. Modellen durchgesetzt. Da diese Branchen im Vergleich zur Software-Entwicklung sehr alt sind, gibt es dort schon sehr lange eine einheitliche Beschreibungssprache dieser Modelle, die den Vorteil der einfacheren Kommunikation zwischen den beteiligten Personen bietet. Auch kann man solche Pläne noch Jahre später leicht verstehen, da die verwendete Sprache ja bekannt ist. Deshalb versuchte man auch im Bereich der Software-Entwicklung eine standardisierte Modellierungssprache zu etablieren, damit die Kommunikation zwischen den Menschen, die in die Software-Entwicklung eingebunden sind, sich vereinfacht und damit man auch Jahre später sich einfach in ein Softwareprodukt einarbeiten kann.

Die Unified Modeling Language (UML) ist die Standard Modellierungssprache für die Software-Entwicklung, d.h. eine Sprache für die Visualisierung, Spezifizierung, Konstruktion und Dokumentation von Software. Die UML ermöglicht es einem Entwickler seine Arbeit als standardisierte Pläne oder Diagramme zu visualisieren. Zum Beispiel ist das charakteristische Symbol oder Icon der Bestimmung von Anforderungen eine Ellipse, die einen Anwendungsfall repräsentiert und ein Strichmännchen, das einen Benutzer symbolisiert, der diesen Anwendungsfall anwendet. Diese Icons stellen jedoch nur eine graphische Notation dar, hinter der sich eine standardisierte Semantik befindet. In den Büchern [UML-ReferencManual] bzw. [UML-UserGuide] wird näher auf diese Dinge eingegangen.

Im folgenden werden zuerst die Elemente von UML angegeben und wie man dieses Grundvokabular verfeinern und erweitern kann. Der Abschnitt A.2 gibt dann einen Überblick über die graphische Notation der UML.

A.1.1 Vokabular

Die UML stellt dem Entwickler ein Vokabular bereit, das in Abb. A.1 gezeigt wird. Wie man aus dieser Abbildung erkennen kann, gliedern sich die Sprachelemente der UML in drei Gruppen:

- ***Elemente***: Als Beispiel seien hier die „Klasse“, das „Packet“ und die „Anmerkung“ angegeben.
- ***Beziehungen***: Die Elemente der UML können verschiedene Beziehungen untereinander haben. Als Beispiel sei hier die „Ableitung“ von Klassen genannt.
- ***Diagramme***: Diagramme stellen die verschiedenen Ansichten der Software dar und bilden zusammen die Architektur der Software. Beispiele sind „Klassendiagramme“, die die sta-

tische Struktur der Software beschreiben im Gegensatz zu „Sequenzdiagrammen“, die eine Beschreibung des dynamischen Ablaufs der Software veranschaulicht.

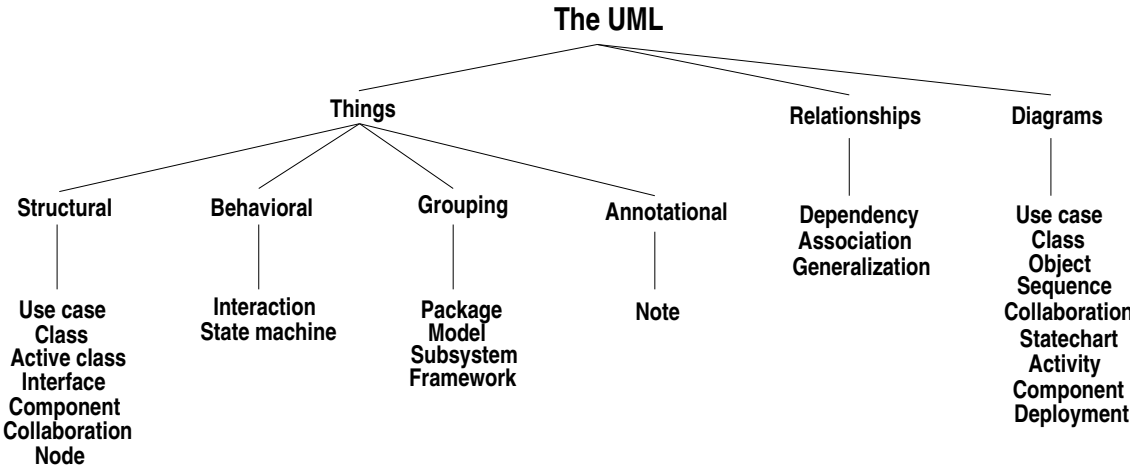


Abbildung A.1: Das Vokabular der UML.

A.1.2 Erweiterungsmechanismen

Die UML enthält einen ***Erweiterungsmechanismus***, der es seinen Benutzern ermöglicht die Syntax und die Semantik zu verfeinern bzw. zu erweitern. Dadurch kann die UML auf ein spezielles System, Projekt oder einen Entwicklungsprozeß zugeschnitten werden. Dieser Erweiterungsmechanismus enthält Stereotypen, Eigenschaftswerte (engl. tagged values) und Beschränkungen (engl. constraints). Durch Stereotypen kann man neue Elemente definiert, indem man die Semantik von schon existierenden Elementen, wie z.B. Dinge und Beziehungen, erweitert und verfeinert. Eigenschaftswerte definieren neue Eigenschaften von schon existierenden Elementen. Und zum Schluß enthalten Beschränkungen einschränkende Regeln, die auf Elemente und ihren Eigenschaften wirken.

A.2 Graphische Notation

Die Abbildungen dieses Kapitels stammen aus [UML-UserGuide] und geben einen graphischen Überblick über die UML.

A.2.1 Strukturelle Dinge

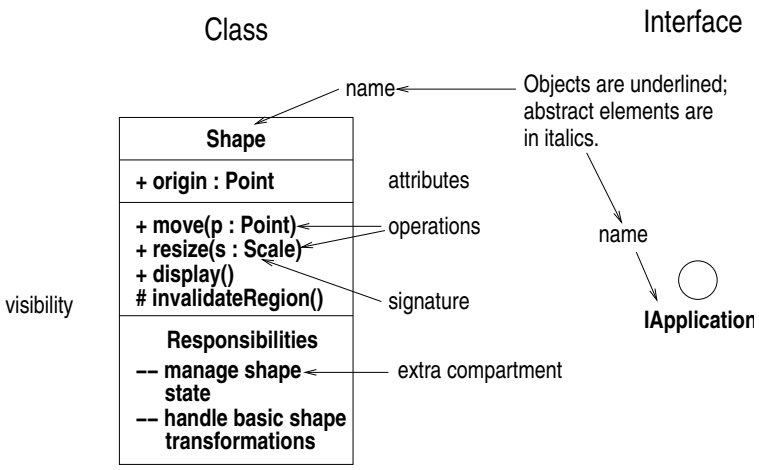


Abbildung A.2: Klassen und Schnittstellen.

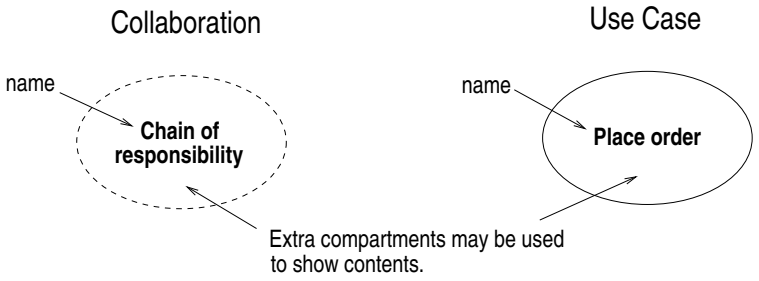


Abbildung A.3: Anwendungsfälle und Kollaborationen (Namen können auch außerhalb der Symbole stehen).

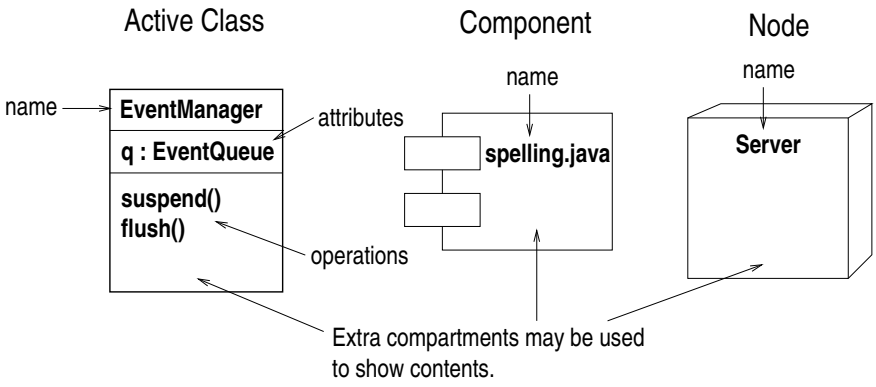


Abbildung A.4: Aktive Klassen, Komponenten und Knoten.

A.2.2 Verhalten

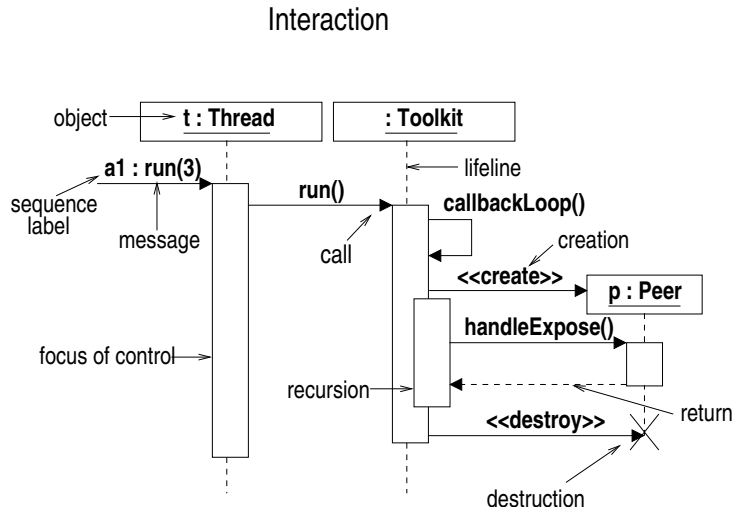


Abbildung A.5: Wechselwirkungen.

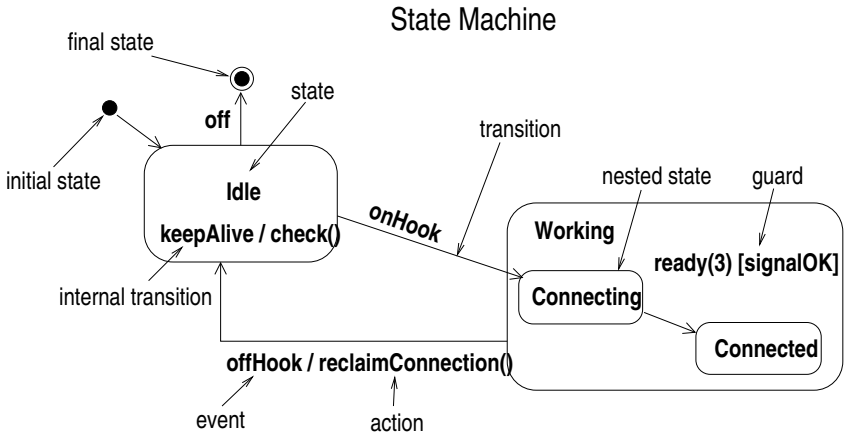


Abbildung A.6: Zustandsmaschinen.

A.2.3 Gruppierungen

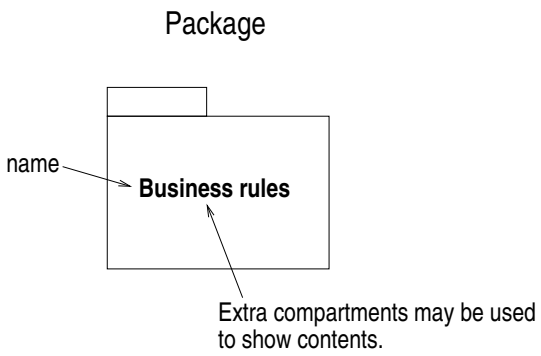


Abbildung A.7: Pakete.

A.2.4 Anmerkungen

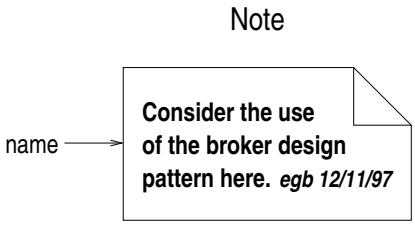


Abbildung A.8: Anmerkungen.

A.2.5 Abhängigkeitsbeziehungen

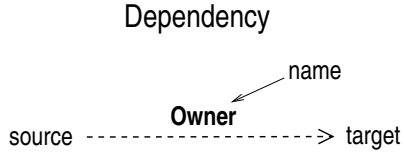


Abbildung A.9: Abhängigkeitsbeziehungen.

A.2.6 Assoziationsbeziehungen

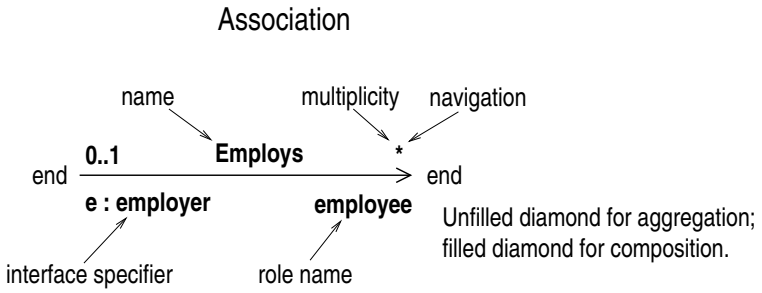


Abbildung A.10: Assoziationsbeziehungen.

A.2.7 Generalisationsbeziehungen

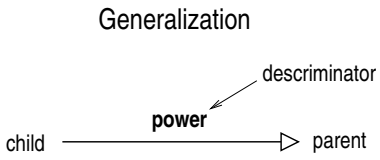


Abbildung A.11: Generalisationsbeziehungen.

A.2.8 Erweiterungsmechanismen

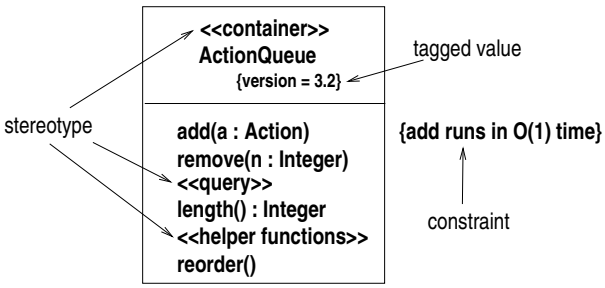


Abbildung A.12: Erweiterungsmechanismen.

Anhang B

Design Pattern Katalog

Die Idee von Design-Pattern ist die Wiederverwendung von erfolgreichen Strukturen um immer wiederkehrende Probleme schneller und effektiver zu lösen. Denn bei einem neuen Softwareprodukt ist der größte Teil nicht essentiell neu. Die meiste Zeit sollte dadurch für den innovativen Anteil am Entwurf genutzt werden. Design-Pattern sollen auch dem Anfänger sogenanntes „Expertenwissen“ zugänglich machen, ohne daß er sich dieses Wissen durch jahrelanges Üben selber aneignen muß.

Das wohl bekannteste Buch über Entwurfsmuster ist das Buch [DesignPatterns], dessen vier Autoren auch als die „Gang of Four (GoF)“ bezeichnet werden. Die unter anderem aus diesem Buch entstandene „Musterbewegung“ ist aber primär eine Internet-Bewegung[1], d.h. die aktuellen Muster sind meist nicht in Buchform verfügbar. Eine weiterführende Diskussion über Entwurfsmuster bzw. deren Dokumentation findet man in dem Buch [Quibeldey-Cirkel], in dem auch ein interessantes Kapitel über „Entwerfen und Dokumentieren von Mustern“ enthalten ist.

Die Abbildung B.1 zeigt eine Übersicht über all die Pattern, die in [DesignPatterns] enthalten sind und ihre Beziehung untereinander. Da die in [DesignPatterns] beschriebenen Design-Pattern noch nicht in UML sondern in OMT[2] beschrieben wurden, wird in Abschnitt B.1 zuerst die OMT eingeführt. Folgende, für diese Diplomarbeit wichtigen Pattern werden anschließend in Abschnitt B.2 beschrieben:

- ***Composite-Pattern***: Siehe B.2.1.
- ***Strategy-Pattern***: Siehe B.2.2.
- ***Template-Pattern***: Siehe B.2.3.
- ***Singleton-Pattern***: Siehe B.2.4.
- ***Observer-Pattern***: Siehe B.2.5.

[1] Einige interessante Sites:
http://ti.et-inf.uni-siegen.de/Entwurfsmuster/home/Entwurfsmuster-Homepage.htm,
http://www.hillside.net/patterns/patterns.html,
http://www.cetus-links.org/oo_patterns.html.

[2] OMT steht für „Object Modeling Technique“.

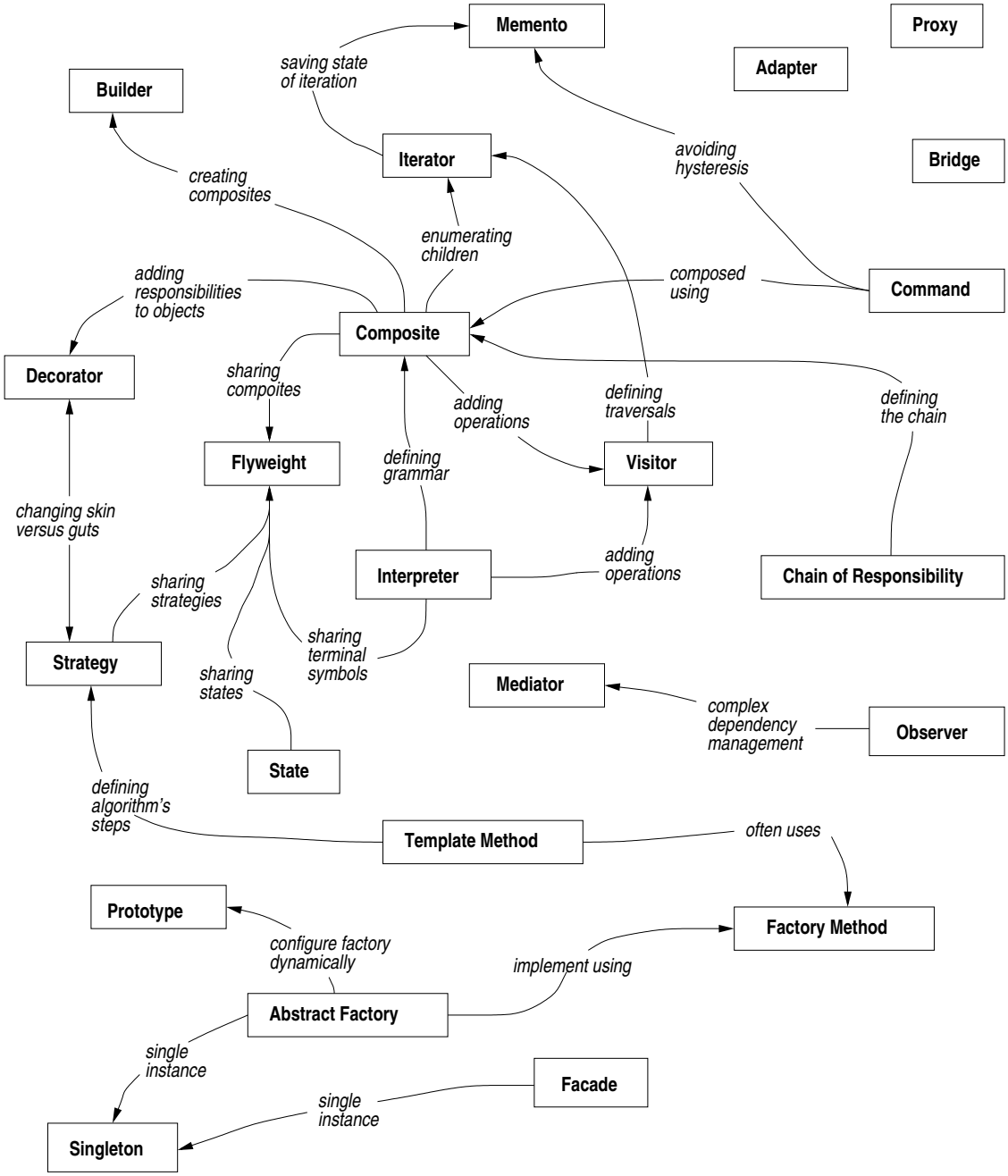


Abbildung B.1: Die Beziehungen zwischen Design Patterns. Aus [DesignPatterns].

B.1 Die OMT Notation

In [DesignPatterns] basieren die Klassen- und Objekt-Diagramme auf der OMT (Object Modeling Technique) von James Rumbaugh u.a., da bei der Entstehung des Buches sich der Standard UML noch nicht herausgebildet hatte. Damit man jedoch die Orginal-Diagramme der GoF-Pattern verstehen kann, wird die OMT im folgenden näher beschrieben:

- Die Abbildung B.2 zeigt die OMT-Notation für abstrakte (schräger Klassennamen) und konkrete Klassen. Typangaben sind optional und gehorchen der C++-Notation, d.h. sie werden vor dem Funktionsnamen angegeben. Auch die Client-Klassen erscheinen als normale Klassen.

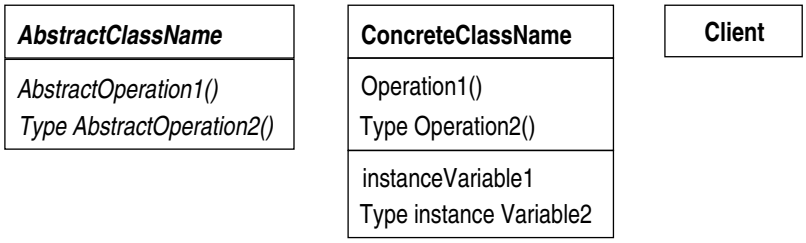


Abbildung B.2: Die OMT-Notation von Klassen.

- Zwischen den Klassen gibt es folgende Beziehungen (siehe Abb. B.3):

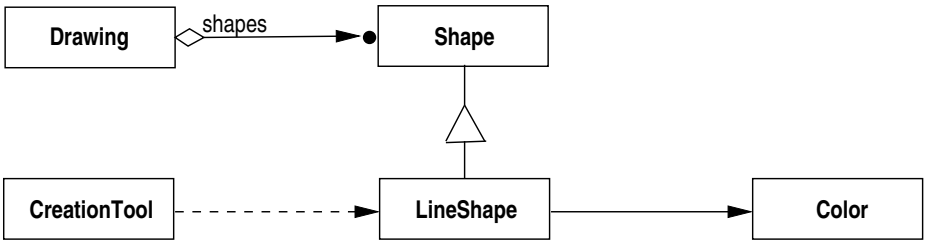


Abbildung B.3: Die OMT-Notation von Beziehungen.

  - Die OMT-Notation für die Ableitung von Klassen ist ein Dreieck zwischen der Elternklasse (hier:Shape) und der Subklasse (hier:LineShape).
  - Ein Pfeil mit einer Raute wird als „part-of“- oder Aggregation-Beziehung bezeichnet und stellt eine Objekt-Referenz dar.
  - Ein Pfeil ohne die Raute bezeichnet man als Bekanntschaft (engl.: acquaintance).
  - Der gestrichelte Pfeil bedeutet die Erzeugung einer Beziehung, d.h, er beschreibt welche Klasse eine andere Klasse instanziiert. Dieses Symbol ist eine hilfreiche Erweiterung der OMT Notation.

Pfeile kann man durch verschiedene Bezeichnungen voneinander unterscheiden. Befindet sich am Ende eines Pfeiles ein Punkt, so hat dies die Bedeutung von „mehr als Eins“, d.h. die Beziehung enthält mehrere Objekte der Klasse auf den der Pfeil zeigt.

- Die Abbildung B.4 verdeutlicht eine Pseudocode-Anmerkung.

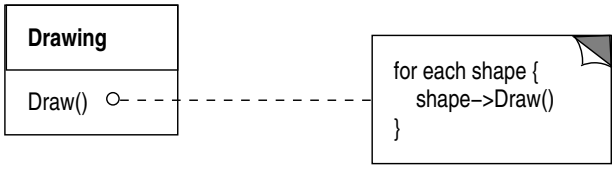


Abbildung B.4: Die OMT-Notation einer Pseudocode-Anmerkung.

- Mit Hilfe eines Objekt-Diagramms wird die Wechselwirkung von instanziierten Objekten untereinander betrachtet. Der Name eines Objektes unterscheidet sich von der zugehörigen Klasse durch ein vorangestellten kleinen Buchstaben. Beispielsweise gehört der Name „aSomething“ zu einem Objekt der Klasse „Something“. Das Symbol eines Objekts ist ein abgerundetes Rechteck, in der eine Linie den Objektnamen von den Objektgenerierungen trennt. Pfeile stellen die Objektgenerierung dar (siehe Abb.B.5).

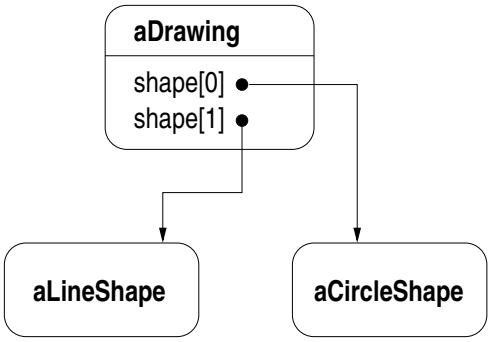


Abbildung B.5: Die OMT-Notation von Objekten.

B.2 Ausgewählte Design-Pattern

Die Beschreibung jedes der folgenden Pattern hat folgende Struktur:

- Aus dem Buch [DesignPatterns] ist die OMT-Struktur des Pattern entnommen.

- Der Beispielcode[3] des Patterns aus [DesignPatterns] ist mit dem Together-Produkt „reengeniert“ worden. Dadurch entstand ein UML-Diagramm des Patterns, das mit dem orginalen OMT-Diagramm verglichen werden kann. Der in C geschriebene Beispielcode der UML-Diagramme wurde auch mit angegeben, damit man sich ein Bild machen kann, wie die Implementierung eines Pattern aussieht.

[3] Auf der CD befinden sich die in C geschriebenen Beispielprogramme des Buches [DesignPatterns] im Verzeichnis /cdrom/data/src/DPCPP.tar.gz.

B.2.1 Composite-Pattern

Im „Composite“-Pattern werden Objekte so zu Baumstrukturen zusammengefügt, daß „Clients“ nicht wissen müssen, ob sie auf einzelne Objekte oder auf eine Zusammensetzung mehrerer Objekte zugreifen. In dieser Diplomarbeit wurde diese Struktur für die Anordnung der Kräfte benutzt (Abschnitt 4.3.3).

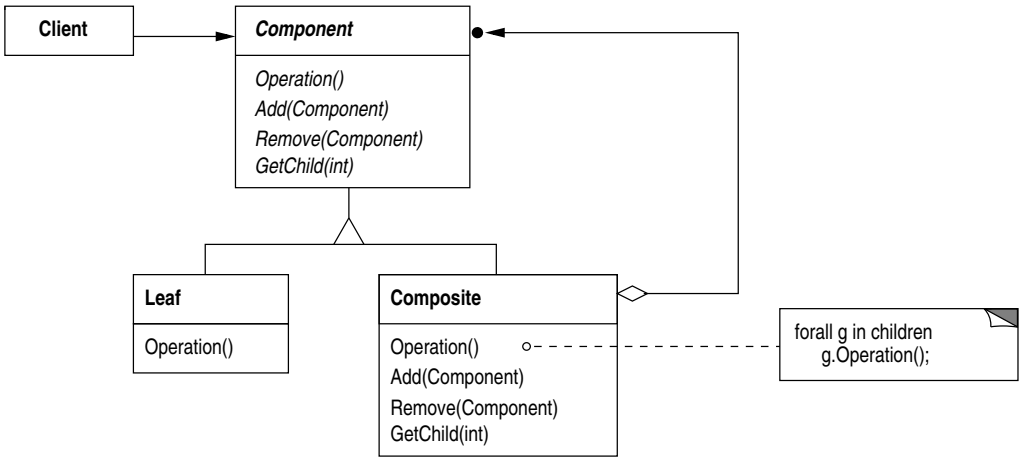


Abbildung B.6: Die OMT-Struktur des Composite-Patterns.

Eine typische Composite-Objekt-Struktur könnte wie in Abbildung B.7 aussehen.

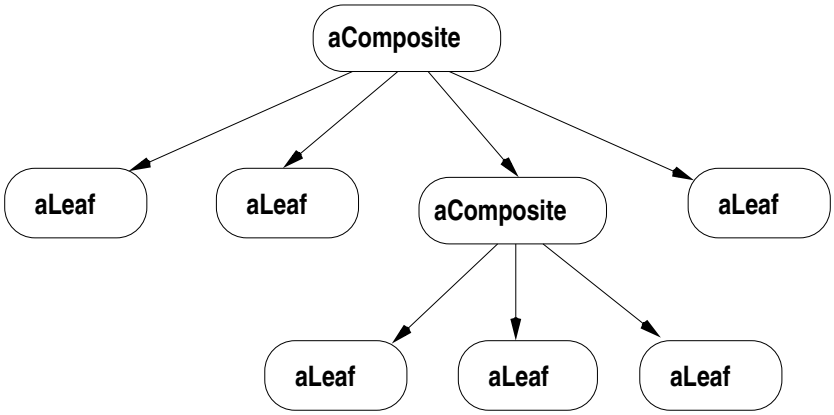


Abbildung B.7: Eine typische Composite-Objekt-Struktur.

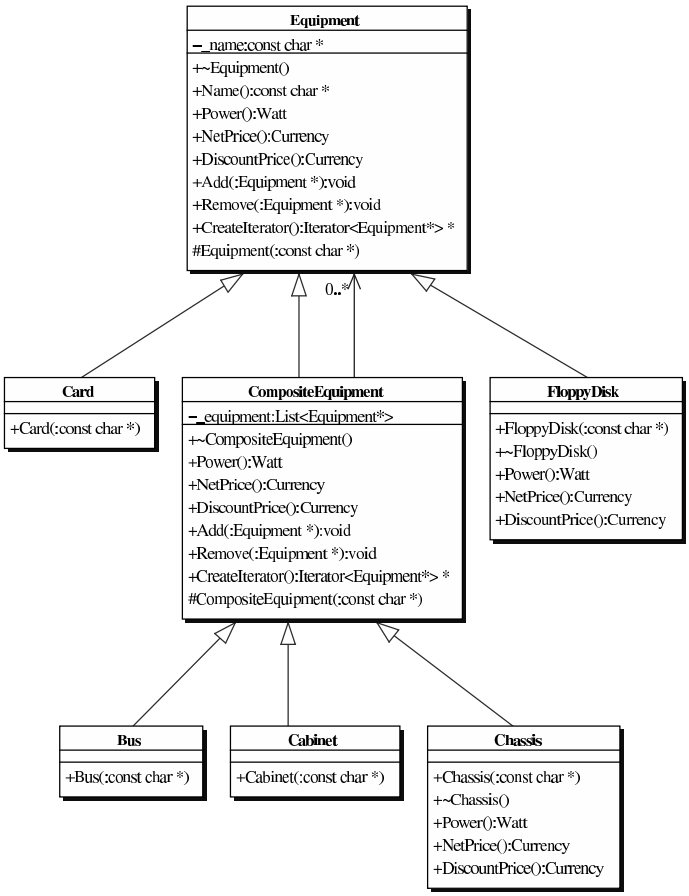


Abbildung B.8: Die UML-Struktur des Composite-Pattern-Beispielcodes.

```
#include "List.H"
#include "iostream.h"
typedef int Watt;
typedef int Currency;
typedef int Power;

class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};

class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};

class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List<Equipment*> _equipment;
};

class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};

class Cabinet  : public CompositeEquipment {
public:
    Cabinet(const char*);
};
class Bus  : public CompositeEquipment {
public:
    Bus(const char*);
};
class Card  : public Equipment {
public:
    Card(const char*);
};
```

Abbildung B.9: Der Composite-Pattern-Beispielcode.

B.2.2 Strategy-Pattern

Gegeben ist eine Familie von austauschbaren Algorithmen. Beim Strategy-Pattern können diese Algoritmen unabhängig von „Clients“ ausgetauscht werden. In dieser Diplomarbeit wird dieses Pattern einerseits für die Auswahl des Kernels (Abschnitt 4.3.2) und andererseits für die Auswahl des Integrators (Abschnitt 4.3.4) benutzt.

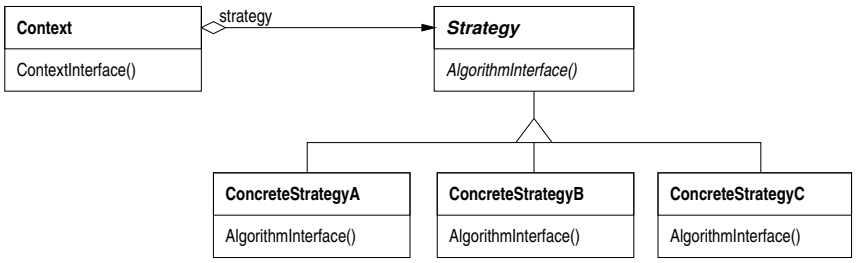


Abbildung B.10: Die OMT-Struktur des Strategy-Patterns.

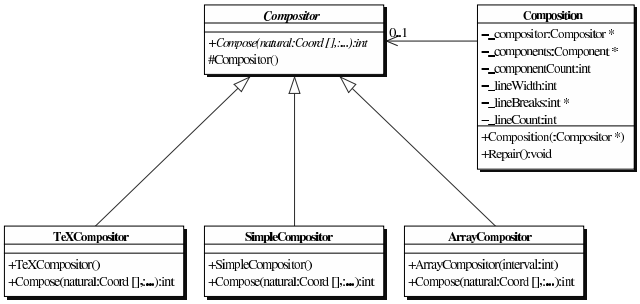


Abbildung B.11: Die UML-Struktur des Strategy-Pattern-Beispielcodes.

```
#include "Geom.H"

class Compositor;
class Component;

class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;    // the list of components
    int _componentCount;       // the number of components
    int _lineWidth;            // the Composition's line width
    int* _lineBreaks;          // the position of linebreaks
                               // in components
    int _lineCount;            // the number of lines
};

class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};


class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};

class TeXCompositor : public Compositor {
public:
    TeXCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};

class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Abbildung B.12: Der Strategy-Pattern-Beispielcode.

B.2.3 Template-Pattern

Beim Template-Pattern wird die Grundstruktur eines Algorithmus in der „Template Methode“ definiert und der Rest in Unterklassen verschoben. Dadurch werden bestimmte Teile des Algorithmus durch die Unterklassen neu definiert, ohne daß die Struktur des Algorithmus verändern wird. In dieser Diplomarbeit wird das Template-Pattern im Paket „Kraefte“ (Abschnitt 4.3.3) benutzt.

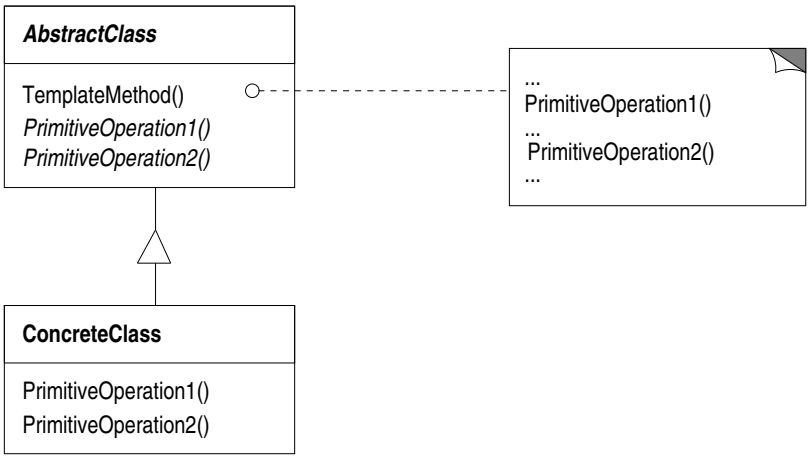
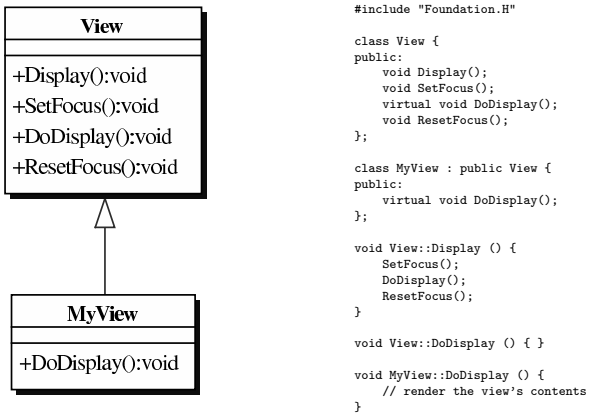


Abbildung B.13: Die OMT-Struktur des Template-Patterns.




```
#include "Foundation.H"

class View {
public:
    void Display();
    void SetFocus();
    virtual void DoDisplay();
    void ResetFocus();
};

class MyView : public View {
public:
    virtual void DoDisplay();
};

void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}

void View::DoDisplay () { }

void MyView::DoDisplay () {
    // render the view's contents
}
```

Abbildung B.14: Der Template-Pattern-Beispielcode und dessen UML-Struktur.

B.2.4 Singleton-Pattern

Das Singleton-Pattern stellt sicher, daß es nur eine Instanz einer Klasse gibt und daß auf dieses Objekt global zugegriffen werden kann. In dieser Diplomarbeit sollte dieses Pattern für die Klasse „PhysSystem“ benutzt werden, da es in der Simulation nur ein physikalisches System geben kann. Auch muß von überall auf ein Objekt dieser Klasse zugegriffen werden. Aus Performance Gründen wurde dieses Pattern jedoch nicht implementiert, da bei jedem Zugriff auf das Singleton-Pattern zuerst eine Abfrage erfolgt, ob dieses Objekt schon existiert oder nicht.

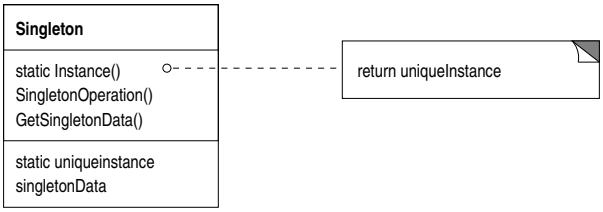
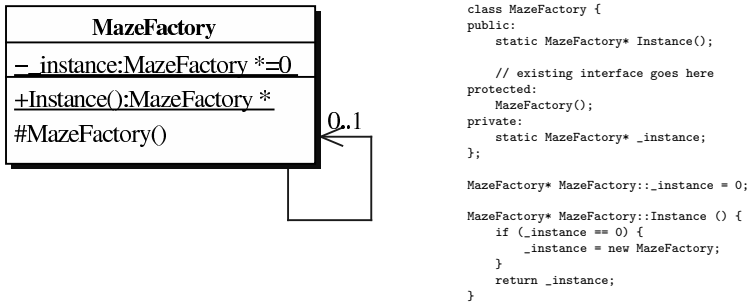


Abbildung B.15: Die OMT-Struktur des Singelton-Patterns.




```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};

MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Abbildung B.16: Der Singelton-Pattern-Beispielcode und dessen UML-Struktur.

B.2.5 Observer-Pattern

Das Observer-Pattern definiert eine „one-to-many“ Abhängigkeit zwischen Objekten. Das bedeutet, daß wenn sich der Zustand eines Objekts ändert, so werden die abhängigen Klassen benachrichtigt und automatisch aktuallisiert. Auch dieses Pattern ist in dieser Diplomarbeit nicht implementiert worden. Jedoch wird es in der Design-Phase vorgeschlagen (siehe Abb. 4.5), um eine GUI mit der Simulation möglichst lose zu koppeln.

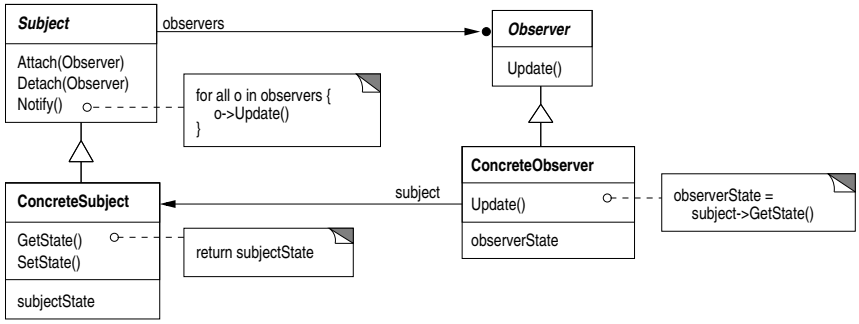


Abbildung B.17: Die OMT-Struktur des Observer-Patterns.

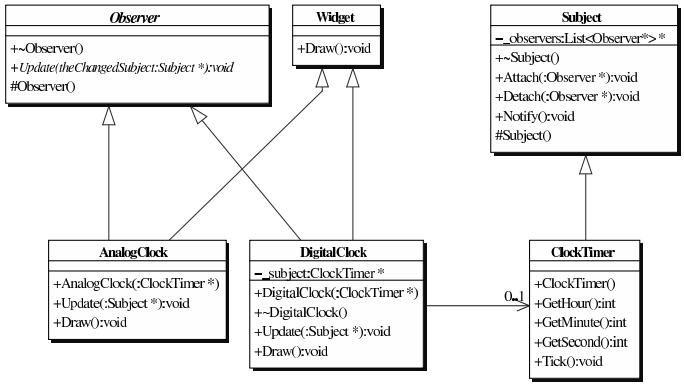


Abbildung B.18: Die UML-Struktur des Observer-Pattern-Beispielcodes.

```
#include "List.H"

class Subject;

class Observer {
public:
    virtual ~ Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};

class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}

class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};

void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```

```
class Widget {
public:
    virtual void Draw();
};

class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // draw the digital clock
}

class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};

ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Abbildung B.19: Der Observer-Pattern-Beispielcode.

Anhang C

Der Unified Process

C.1 Einführung in den Software-Entwicklungsprozeß

Da Computer immer schneller und leistungsfähiger[1] werden, erweitert sich auch deren Anwendungsbereich. Als Beispiel seien hier moderne Multimedia-Anwendungen, wie DVD-Player, Videoschnitt, u.s.w. genannt. Daraus folgt der Bedarf nach immer komplexeren Softwareprodukten, die natürlich im Internetzeitalter immer schneller und besser an die Benutzer angepaßt sein sollen.

Um diese erhöhten Anforderungen zu erfüllen, werden immer bessere Entwicklungstools, bessere Programmiersprachen und schnellere Hardware benutzt. Grundlegend für das Bauen von besser an den Benutzer angepasster Software ist jedoch ein guter Software-Prozeß. Allgemein definiert ein Prozeß, „***wer was wann***“ tut und „***wie***“ ein bestimmtes Ziel erreicht werden kann. Das Ziel eines Software-Prozesses ist die Herstellung eines Softwareproduktes oder die Verbesserung eines schon existierenden Softwareproduktes (siehe Abb. C.1). Ein guter Software-Prozeß beschreibt demnach eine Vorgehensweise, um effektiv hochwertige Software zu produzieren.

Abbildung C.1: Ein Software-Entwicklungsprozeß.

Die Wichtigkeit eines solchen Prozesses erkennt man z.B. durch Vergleich in einem ähnlichen Bereich, der Bauwirtschaft. Je größer und teuerer ein Gebäude werden soll, um so wichtiger ist die Bestimmung der Anforderungen und die Planung. Eine Hundehütte kann man noch aus ein paar Brettern und Nägeln erfolgreich zusammenbauen, jedoch geht diese Vorgehensweise beim Bau eines Hochhauses mit Sicherheit schief. Übertragen auf die Entwicklung von Software bedeutet dies, daß je größer und komplexer das zu bauende Softwareprodukt sein soll, desto wichtiger wird dessen Planung (Anforderungen, Finanzierung, ...).

Jedoch ist ein Softwareprozeß kein statisches Gebilde, sondern hängt dynamisch von folgenden vier Bereichen ab:

- der ***Technologie:*** Darunter fallen unter anderem Programmiersprachen, Netzwerke, Betriebssysteme und Entwicklungsumgebungen.
- den ***Tools:*** Sie erweitern die Möglichkeiten eines Prozesses. Zum Beispiel ist es mit modernen Modelling-Tools (wie Together) viel einfacher sich ein Bild von dem zukünftigen Software-

[1] vgl. mit ***Mooreschem Gesetz***: 1968 von Gordon Moore, dem damaligen Geschäftsführer von Intel, aufgestellte Gesetzmäßigkeit, nach der sich die Anzahl der Transistoren auf einem Prozessor etwa alle 18 Monate verdoppelt.

produkt zu machen. Überdies forciert natürlich ein Prozeß auch die Entwicklung von Tools, um häufig benutzte Teile des Prozesses zu vereinfachen.

- den ***Menschen:*** Ein Prozeß muß auf den Fähigkeiten der Entwickler aufbauen.
- der ***Organisationsstrukturen:*** Software-Entwickler, die an einem gemeinsamen Projekt arbeiten, können z.B. auf der ganzen Welt verteilt sein und aus freien Mitarbeitern, Angestellten und Zeitarbeitern bestehen.

Ein Softwareprozeß muß daher genauso wie ein Softwareprodukt entwickelt werden. Dies benötigt in der Regel einige Jahre um ein bestimmtes Maß an Stabilität und Reife zu erreichen.

Im folgenden werden die wichtigsten Software-Entwicklungsprozesse aufgezählt, die sich im Laufe der Zeit entwickelt haben:

- (Rational) Unified Process[2] der Firma Rational.
- FDD (Feature Driven Development) Prozeß der Firma Togethersoft[3].
- OPEN-Process von der OOSP[4] (Object Oriented Software Process).
- XP[5] (Extreme Programming).
- Catalysis[6].
- DSDM[7] (Dynamic System Development Method).

Von diesen Software-Prozessen soll im folgenden nur auf den Unified Process näher eingegangen werden, da dieser von denselben Leuten entwickelt wurde, die auch die UML entwickelt haben und deshalb der „Quasi-Standard“ in diesem Bereich ist. Die folgende Beschreibung des Unified Process ist größtenteils aus [The Unified Software Development Process] entnommen worden, wobei dieses Buch noch näher auf die Einzelheiten des Unified Process eingeht und deshalb als weiterführende Literatur empfohlen wird.

C.2 Einführung in den Unified Process

Der Unified Process ist kein bestimmter, spezieller Prozeß, sondern er ist eher ein Grundgerüst eines Prozesses, der an die verschiedenen Softwaresysteme, Anwendungsbereiche, Organisationsstrukturen und Projektgröße angepasst werden kann. Die beim Entwurf eines Softwareproduktes entworfenen Modelle werden mit Hilfe der ***Unified Modeling Language*** (siehe Anhang A) beschrieben, da wie vorher schon bemerkt wurde, die UML und der Unified Prozeß zusammen entwickelt wurden.

Es gibt drei Schlüsselbegriffe des Unified Process, die man auch als die ***Grundpfeiler*** desselben betrachten kann:

- ***Anwendungsfall gesteuert*** (engl.: use-case driven).
- ***Architektur zentriert*** (engl.: architecture-centric).
- ***Iterative und Inkrementelle*** Entwicklung.

In den folgenden drei Unterabschnitten werden diese drei Schlüsselbegriffe näher erläutert.

[2] siehe http://www.rational.com/products/rup.
[3] siehe http://www.togethersoft.com.
[4] siehe http://www.open.org.au.
[5] siehe http://www.ExtremeProgramming.org.
[6] siehe http://www.catalysis.org.
[7] siehe http://www.dsdm.org.

C.2.1 Der Unified Process ist Anwendungsfall gesteuert

Wie in vielen anderen Bereichen gilt auch in der Softwareentwicklung der Satz: „Der Kunde ist König“. Das bedeutet, daß die zu bauende Software den Anforderungen des Benutzers angepaßt werden soll und nicht umgekehrt. Diese Anforderungen müssen daher als erstes gefunden und analysiert werden.

Noch einige Worte zum Begriff „Benutzer“. Dieser Begriff bezieht sich nicht notwendigerweise auf einen menschlichen Benutzer, sondern kann sich z.B. auch auf ein anderes Computersystem beziehen. In diesem Sinne stellt der Begriff „Benutzer“ also jemanden oder etwas dar, der bzw. das mit dem zu entwickelten System wechselwirkt. Eine solche Wechselwirkung geschieht z.B. beim Abheben von Geld an einem Geldautomaten. Zuerst steckt der Bankkunde seine EC-Karte in den Automaten, beantwortet einige auf dem Display erscheinenden Fragen und erhält anschließend (hoffentlich) das gewünschte Geld. In Abhängigkeit von der EC-Karte und den Eingaben des Benutzers führt das System also eine Folge von ***Aktionen*** durch, die ein bestimmtes Resultat, hier das Abheben des gewünschten Geldbetrages, liefert.

Solch eine Wechselwirkung bezeichnet man als ***„Anwendungsfall“*** (engl.: use-case). Ein Anwendungsfall ist also ein Teil der Funktionalität des Systems, das dem Benutzer ein wichtiges Resultat liefert. Alle Anwendungsfälle zusammengenommen bilden das ***Anwendungsfall-Modell***, das die komplette Funktionalität des Systems beschreibt. Dieses Modell ersetzt die traditionelle funktionale Beschreibung des Systems, die aus den Antworten der Frage „Was soll das System tun?“ besteht. Ein Anwendungsfall kann man also als eine Antwort auf die erweiterte Frage: „Was soll das System ***für jeden Benutzer*** tun ?“ betrachten, wobei die drei zusätzlichen Worte die Bedeutung der Software auf die Bedürfnisse des Benutzers lenken soll und nicht in erster Linie auf seine Funktionalität.

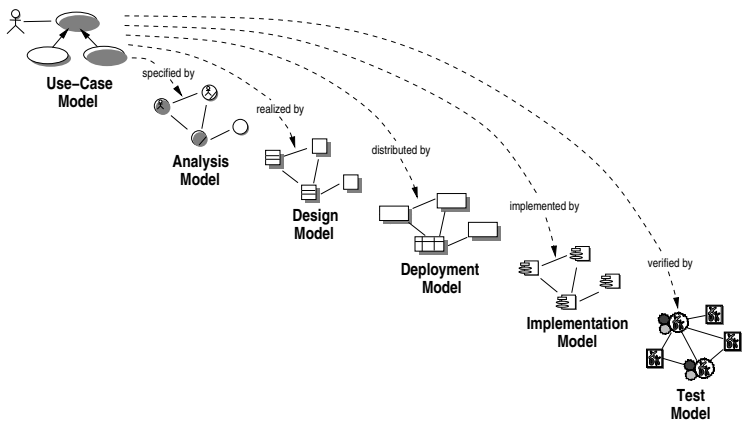


Abbildung C.2: Modelle des Unified Process. Zwischen den Modellen gibt es viele Abhängigkeiten. Als ein Beispiel sind die Abhängigkeiten zwischen dem Anwendungsfall-Modell und den anderen Modellen angegeben.

Das Anwendungsfall-Modell hat jedoch nicht nur die Aufgabe, die Anforderungen des Systems zu beschreiben, sondern es gibt auch die Richtung beim Design, der Implementierung und dem Test vor, d.h. es ***steuert den Entwicklungsprozeß*** (siehe Abb. C.2):

- Auf der Grundlage des Anwendungsfall-Modells wird eine Folge von Design- und Implementierungsmodellen von den Entwicklern gebaut, die die Anwendungsfälle realisieren.
- Die Entwickler überprüfen dann jedes dieser Modelle, ob es mit dem Anwendungsfall-Modell übereinstimmt.

- Die Tester testen daraufhin die Implementierung, um sicher zu stellen, daß die Komponenten des Implementationsmodells die Anwendungsfälle richtig implementieren.

Anwendungsfall gesteuert bedeutet also, daß der Entwicklungsprozeß einer Folge von Arbeitsflüssen (engl.: workflows) folgt, die durch die Anwendungsfälle gesteuert werden.

Anwendungsfälle werden jedoch nicht isoliert ausgewählt, sondern sie werden zusammen mit der Systemarchitektur, über die im nächsten Abschnitt diskutiert wird, entwickelt. Das bedeutet, die Anwendungsfälle steuern einerseits die Systemarchitektur und die Systemarchitektur hat andererseits Einfluß auf die Auswahl der Anwendungsfälle. Beide reifen während des Lebenszykluses des Softwareproduktes.

Da die Funktionalität und deshalb auch die Anwendungsfälle von SPH++ durch das C-Programm (Abschnitt 3.2) schon festgelegt worden sind, ist der in diesem Abschnitt beschriebene Grundpfeiler des Unified Process in dieser Diplomarbeit nur schwach sichtbar. Am Besten kann man den Schlüsselbegriff „Anwendungsfall gesteuert“ beim Testen von SPH++ (Kapitel 5) erkennen, bei der die verschiedenen Simulationsläufe (entspricht hier den Anwendungsfällen) von SPH++ mit dem des C-Programms verglichen wurden.

C.2.2 Der Unified Process ist Architektur fixiert

Am Anfang dieses Anhangs wurde schon auf die Analogie der Softwareentwicklung mit dem Bau eines Gebäudes hingewiesen. Diese Analogie gilt auch für die Architektur. Die Pläne, die ein Architekt für ein Gebäude erstellt, enthalten verschiedene Ansichten: die Vorderansicht des Gebäudes, der Verlauf der Versorgungsleitungen (z.B. Heizungsrohre, Wasserrohre, Elektrizität), u.s.w. . Erst diese Pläne, die zusammen die Architektur bilden, ermöglichen es den beteiligten Personen, sich ein Bild von dem zu bauenden Gebäude zu machen. Ähnlich soll die Architektur eines Softwaresystems die verschiedenen Ansichten der Software beschreiben.

Die Software-Architektur enthält also die wichtigsten statischen und dynamischen Aspekte des Systems. Wie schon im vorigen Abschnitt angedeutet, wird die Architektur hauptsächlich aus den Anforderungen der Benutzer, d.h. den Anwendungsfällen, gebildet. Jedoch wird die Architektur auch noch von vielen anderen Faktoren beeinflußt, wie der Plattform auf der die Software laufen soll (diese besteht aus der Computer-Architektur, dem Betriebssystem, dem Datenbankverwaltungssystem, den Netzwerkprotokollen, u.s.w.), den vorhandenen Bibliotheken (z.B. einer GUI- oder Mathematik-Bibliothek), den Lieferungsbedingungen, der notwendigen Kompatibilität mit älteren Programmen und von nichtfunktionalen Anforderungen wie der Geschwindigkeit und Zuverlässigkeit der Software. Die Architektur ist demnach eine Ansicht auf das gesamte Design, wobei die wichtigen Merkmale hervorgehoben und die Details ausgelassen werden. Diese Trennung ist jedoch subjektiv und hängt von der Erfahrung des Architekten ab, der deshalb eine Schlüsselrolle im Entwicklungsteam einnimmt.

Wie hängen nun die Anwendungsfälle und die Architektur miteinander zusammen? Nun, jedes Produkt hat eine Funktion und eine Form, die gut miteinander harmonieren müssen, damit das Produkt erfolgreich sein kann. Die Funktion entspricht dabei den Anwendungsfällen und die Form der Architektur. Die Entwicklung der Anwendungsfälle bzw. der Architektur ist also ein „Henne - Ei“-Problem: Auf der einen Seite bestimmen die Anwendungsfälle die Funktionalität der Architektur und andererseits bestimmt die Architektur, in welcher Form zukünftige Anwendungsfälle realisiert werden können. Anwendungsfälle und Architektur müssen deshalb parallel entwickelt werden.

Die Aufgabe des Architekten ist es also das System in eine ***Form*** zu gießen. Diese Form muß so designed werden, daß sie an zukünftigen Anforderungen angepasst werden kann. Um solch eine Form zu finden, muß der Architekt die für die Architektur wichtigen Anwendungsfälle von den un-

wichtigen unterscheiden. Diese wichtigen Anwendungsfälle heißen „***Schlüssel-Anwendungsfälle***“ und haben nur einen Anteil von 5% bis 10% aller Anwendungsfälle. Zusammenfassend hat der Architekt also folgende Aufgaben:

- Zuerst muß er eine grobe Struktur der Architektur bauen. Diese ist zwar noch unabhängig von irgendwelchen Anwendungsfällen, jedoch schon soweit vorbereitet, daß diese leicht eingebaut werden können.

- Als nächstes bestimmt er die „Schlüssel-Anwendungsfälle“, die nun detailliert beschrieben und als Subsysteme, Klassen und Komponenten implementiert werden.

- Anschließend entwickelt er die Architektur durch das Hinzufügen von weiteren Anwendungsfällen weiter bis sie eine gewisse Reife und Stabilität erreicht hat.

Die Entwicklung der Architektur von SPH++ wird in Kapitel 4 beschrieben und ist ein wichtiger Teil dieser Diplomarbeit. Jedoch besitzt die Architektur von SPH++ noch einige Nachteile, auf die in Abschnitt 6.2 genauer eingegangen wurde.

C.2.3 Der Unified Process ist iterativ und inkrementell

Die Entwicklung eines großen Softwareproduktes kann mehrere Monate oder sogar Jahre in Anspruch nehmen. Es ist daher geschickt, wenn man diese Arbeit in mehrere Teile oder ***Mini-Projekte*** zerlegt. Jedes Mini-Projekt ist dabei eine ***Iteration***, das zu einem ***Inkrement***, d.h. zu einem Zuwachs des Produktes, führt. Eine Iteration kann man dabei als eine Menge von einzelnen Schritten in den Arbeitsflüssen (siehe Abb. C.5) verstehen, wohingegen ein Inkrement einem Wachstumsschritt des ganzen Produktes entspricht. Je besser die Iterationen geplant und ausgeführt werden, desto effektiver ist der ganze Softwareprozess.

Die Frage, was nun in einer Iteration implementiert wird, hängt von zwei Faktoren ab. Als erstes wird eine Anzahl zu implementierender Anwendungsfällen ausgesucht, die die Funktionalität des Systems erweitern sollen. Als zweites werden die größten Risiken bestimmt und versucht diese zu vermeiden oder wenigsten zu verkleinern. Jede Iteration baut natürlich auf den Artefakten der vorhergegangenen Iteration auf. Sind die Anwendungsfälle bestimmt, ist die weitere Entwicklung (Analyse, Design, Implementierung und Test) schon vorgezeichnet, d.h. die Anwendungsfälle werden in einen ausführbaren Code überführt. Die Größe der Inkremente, beziehungsweise das Wachstum des Projektes in den einzelnen Iterationen, ist natürlich nicht notwendigerweise gleich groß. Speziell in der Anfangsphase kann in einer Iteration ein Design auch durch ein besseres Design ersetzt werden, wobei das Inkrement in diesem Fall gleich null ist. In späteren Iterationen sind die Inkrementen in der Regel ähnlich groß.

Der Ablauf jeder Iteration hat also ein festgelegtes Muster: Zuerst werden die relevanten Anwendungsfälle von den Entwicklern identifiziert und ausgewählt. Danach wird ein Design erstellt, das mit der gewählten Architektur übereinstimmt und implementiert. Zum Schluß wird diese Implementierung getestet, ob sie mit den am Anfang ausgewählten Anwendungsfällen übereinstimmt. Ist das der Fall, wird mit der nächsten Iteration fortgefahren, ansonsten muß die Iteration mit einem besseren Ansatz nochmals durchgeführt werden.

Der Softwareprozeß ist am effektivsten, wenn nur die Iterationen ausgeführt werden, die für das Endprodukt notwendig sind. Weiterhin versucht man, diese Iterationen in eine logische Reihenfolge zu bringen und sich daran auch möglichst genau zu halten. Unvorhergesehene Probleme können natürlich zusätzliche Iterationen oder eine Änderung der Reihenfolge notwendig machen, die zusätzliche Zeit und Arbeit benötigen. Diese unvorhergesehenen Probleme zu minimieren ist eins der Ziele der ***Risikominimierung***.

Welche Vorteile hat nun ein kontrolliert iterativer Prozeß? Einige dieser Vorteile sollen kurz aufgezählt werden:

- Kontrollierte Iterationen reduzieren die finanziellen Risiken auf die Ausgaben eines einzigen Inkrements. Muß eine Iteration wiederholt werden ist das immer noch besser als wenn das ganze Projekt wiederholt werden muß.
- Kontrollierte Iterationen reduzieren das Risiko, daß das Produkt nicht im geplanten Zeitpunkt auf den Markt kommt. Dadurch, daß die Risiken so früh wie möglich identifiziert werden, kann man früher an ihrer Lösung arbeiten und der entstandene Zeitverlust kann besser ausgeglichen werden. Beim „traditionellen“ Ansatz werden schwierige Probleme erst beim Testen des Systems aufgedeckt und die Zeit, die man für die Behebung dieses Problems benötigt, übersteigt gewöhnlich die restliche Zeit des Planes, d.h. der Auslieferungstermin muß fast immer verschoben werden.
- Kontrollierte Iterationen beschleunigen das Tempo der Softwareentwicklung, da Entwickler besser motiviert sind, wenn die Ziele klar definiert und zeitlich nicht weit weg liegen (vgl. auch den FDD-Prozeß der Firma Together).
- Kontrollierte Iterationen berücksichtigen die oft ignorierte Tatsache, daß die Anforderungen der Benutzer nicht von Anfang an vollständig bestimmt werden können. Diese kristallisieren sich erst während der folgenden Iterationen klar heraus.

Auch die Entwicklung von SPH++ verlief iterativ und inkrementell, wie man in Kapitel 4 an den drei Phasen erkennen kann. Diese drei Phasen bestanden wiederum aus mehreren Iterationen, wobei das Ende jeder Iteration aus einem ausführbaren Programm bestand. Ein weiterer Vorteil dieser Vorgehensweise war das leichtere Erlernen von C++, da während des ganzen Entwicklungszykluses von SPH++ der Schwerpunkt auf ein ausführbares Programm gelegt wurde.

Nachdem nun die Schlüsselkonzepte des Unified Process eingeführt wurden, wird im nächsten Kapitel das Lebens des Unified Process beschrieben.

C.3 Das Leben des Unified Process

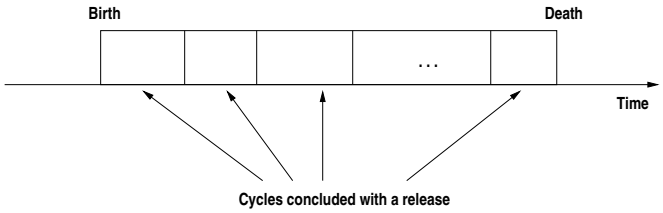


Abbildung C.3: Das Leben eines Prozesses besteht aus Zyklen.

Der Unified Process wiederholt sich bei jedem neuen Zyklus eines Softwaresystems solange, bis das Leben des Softwaresystems endet (siehe Abb. C.3). Jeder Zyklus wird dabei mit einem ***Release*** abgeschlossen, das an die Benutzer ausgeliefert wird. Was alles in solch einem Release enthalten sein soll, wird in Abschnitt C.3.1 genauer aufgeführt.

Jeder Zyklus wiederum besteht aus den folgenden vier Phasen (siehe Abb. C.4), auf die in Abschnitt C.3.2 näher eingegangen wird:

- Etablierung (engl.: inception).
- Entwurf (engl.: elaboration).

- Konstruktion (engl.: construction).
- Übergang (engl.: transition).

Diese vier Phasen unterteilt man wiederum in mehrere Iterationen (siehe Abb. C.4), über die in Abschnitt C.2.3 schon ausführlich diskutiert wurde.

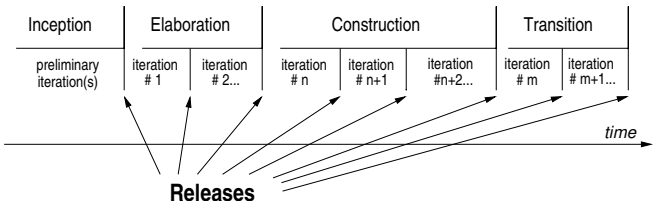


Abbildung C.4: Jede Iteration innerhalb einer Phase erzeugt ein ausführbares Release des Systems.

Die Entwicklung von SPH++ enthält auch diese vier Phasen, die im Hauptteil dieser Diplomarbeit beschrieben wurden. Demnach hat SPH++ einen kompletten Zyklus des Unified Process hinter sich gebracht und kann durch einen zukünftigen zweiten Entwicklungszyklus an neue Anforderungen angepasst werden. Auf diese zukünftige Entwicklung von SPH++ wird in Kapitel 6 näher eingegangen.

C.3.1 Das Software-Produkt

Das Ende jedes Zykluses liefert ein neues Release des Systems, das an die Kunden ausgeliefert werden kann. Dieses Release besteht aus dem Sourcecode, der kompiliert und ausgeführt werden kann, plus Manuals und anderen Dingen. Wichtig ist dabei anzumerken, daß ein Release nicht nur den Ansprüchen der Benutzer gerecht werden muß, sondern auch allen anderen beteiligten Personen, die an der Entwicklung des Produktes mitarbeiten. Das bedeutet, ein Softwareprodukt ist mehr als der ausführbare Maschinencode.

Das Endprodukt besteht also aus den Anforderungen, den Anwendungsfällen, den nichtfunktionalen Anforderungen und den Testfällen. Darüberhinaus gehört zum Endprodukt noch die Architektur und die visuellen Modelle, d.h. die Artefakte die mit Hilfe der UML modelliert wurden. Zusammengefaßt kann man deshalb sagen, daß alle Dinge, die vorher schon beschrieben wurden in dem Release enthalten sein müssen, da damit für alle an der Software Beteiligten, wie Kunden, Anwender, Analytiker, Designer, Implementierer, Tester und Manager, die Möglichkeit gegeben wird, das System zu beschreiben, zu designen, zu implementieren bzw. anzuwenden.

Ein weiterer Grund, daß der ausführbare Code allein nicht ausreicht, ist die sich mit der Zeit ändernde Umgebung des Systems, z.B. des Betriebssystems, der Datenbank, der Hardware, u.s.w.. Im Laufe der Zeit versteht man zwar die Aufgaben, die die Software erfüllen soll, immer besser, jedoch haben sich möglicherweise mit der Zeit die Anforderungen wieder geändert. Um dieser Tatsache Rechnung zu tragen, müssen die Entwickler eventuell einen neuen Zyklus starten. Dieser benötigt dann alle diejenigen Teile des Softwareproduktes, die im vorigen Entwicklungszykluses entwickelt wurden. Diese seien im Folgenden noch einmal aufgeführt (siehe auch Abb. C.2):

- Ein ***Anwendungsfall-Modell*** mit allen Anwendungsfällen und ihren Beziehungen zu den Benutzern.
- Ein ***Analyse-Modell***, das zwei Aufgaben hat: die Anwendungsfälle detaillierter auszuarbeiten und das gewünschte Verhalten des Systems auf eine Menge von Objekten zu übertragen.

- Ein ***Design-Modell***, das (a) die statische Struktur des Systems als Subsysteme, Klassen und Schnittstellen realisiert und (b) die Anwendungsfälle als ***Kollaborationen*** zwischen den Subsystemen, Klassen und Schnittstellen realisiert.
- Ein ***Implementierungsmodell***, welches die Komponenten, die den Source-Code repräsentieren, enthält und die Abbildung der Klassen auf diese Komponenten.
- Ein ***Einsatz-Modell***, das die physikalischen Knoten von Computern definiert und die Komponenten auf diese Knoten abbildet.
- Ein ***Test-Modell***, das die Testfälle beschreibt, welches die Anwendungsfälle verifiziert.
- Und natürlich eine Darstellung der ***Architektur***.

Das System enthält zusätzlich ein ***Business-Modell***, das die Geschäftsumgebung des Systems beschreibt.

Alle diese Modelle beziehen sich aufeinander und repräsentieren das ganze System. Elemente in einem Modell haben sog. ***Trace***Abhängigkeiten, die vorwärts oder rückwärts auf Elemente anderer Modelle gerichtet sind. Zum Beispiel kann ein Anwendungsfall (in dem Anwendungsfall-Modell) ein Trace zu einer Anwendungsfall-Realisation (in dem Design-Modell) oder zu einem Testfall (im Test-Modell) haben. Diese Trace-Abhängigkeiten fördern das Verständnis und die Weiterentwicklung des Systems.

Das Softwareprodukt „SPH++“ besteht also aus dem Source-Code, dieser Diplomarbeit und den anderen Dingen, die auf der beiliegenden CD enthalten sind. Da der Entwicklungsprozeß von SPH++ noch nicht alle Aspekte des Unified Process enthält, gilt dies natürlich auch für das Softwareprodukt „SPH++“. Beispielsweise enthält SPH++ noch keine Sequence- und Use-Case-Diagramme.

C.3.2 Die in einem Zyklus enthaltenen Phasen

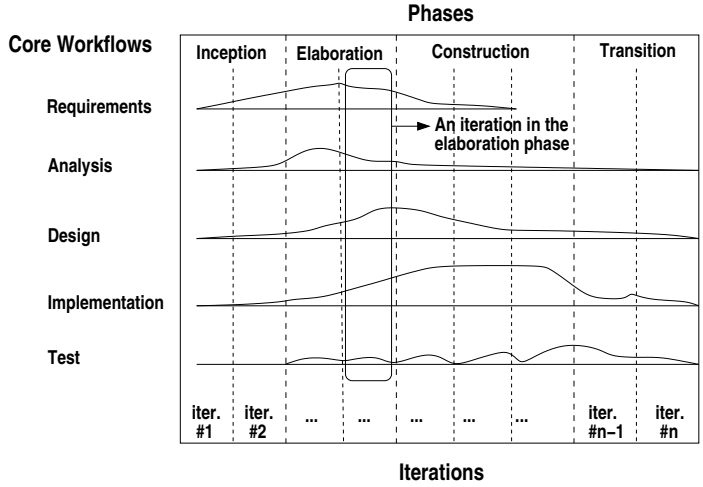


Abbildung C.5: Die fünf Arbeitsflüsse (Anforderungen, Analyse, Design, Implementation und Test) sind über die vier Phasen (Inception, Elaboration, Konstruktion und Transition) aufgetragen.

Die Zeit eines Zykluses wird in vier aufeinanderfolgende Phasen aufgeteilt (siehe Abb. C.5). Was in den einzelnen Phasen geschieht, kann man sich anhand der Fortschritte in den verschiedenen Modellen veranschaulichen. Jede Phase wird weiter in Iterationen zerlegt, die jeweils ein

Inkrement des Produktes zur Folge haben. Jede Phase wird durch einen sog. ***Meilenstein*** beendet. Ein Meilenstein wird als Vorhandensein einer Menge von Artifakten definiert, d.h. bestimmte Modelle und Dokumente müssen ein vorgeschriebenen Zustand erreicht haben.

Meilensteine dienen vielen Zwecken. Der entscheidenste ist, daß an diesem Punkt einige wichtige Entscheidungen getroffen werden müssen, bevor die Arbeit in die nächste Phase übergehen kann. Meilensteine erlauben es daher dem Management und natürlich auch den Entwicklern selber den Fortschritt der Arbeit zu kontrollieren. Darüberhinaus wird auch die Zeit und der Aufwand jeder Phase gespeichert, die dann hilfreich sind für die Abschätzung der erforderlichen Zeit und Mitarbeiteranforderungen in anderen Projekten.

Auf der linken Seite der Abb. C.5 sind die verschiedenen Arbeitsflüsse aufgetragen. Die dazugehörigen Kurven zeigen den geschätzten Arbeitsaufwand jedes Arbeitsflusses in Abhängigkeit von der Zeit, während der das Projekt entsteht. Diese Zeit wird weiter aufgeteilt in die früher schon erwähnten Phasen, die wiederum aus Iterationen bzw. Mini-Projekten bestehen. Eine typische Iteration geht dabei durch alle Arbeitsflüsse wie es für eine Iteration in der Elaboration-Phase in Abb. C.5 gezeigt wird.

Die einzelnen Phasen werden nun näher beschrieben:

- Während der ***Etablierungsphase*** wird eine gute Idee bis zu einer konkreten Vorstellung des Endproduktes entwickelt und der Einsatz dieses Produktes dargelegt (Businessfall). Im wesentlichen beantwortet diese Phase folgende Fragen:

  - Was soll das System für die meisten Benutzer hauptsächlich leisten?
  - Wie kann die Architektur für dieses System aussehen?
  - Wie sieht der Entwicklungsplan aus und was wird die Entwicklung des Produktes kosten?

  Die erste Frage wird durch ein vereinfachtes Anwendungsfall-Modell, das die wichtigsten Anwendungsfälle enthält, beantwortet. Zu diesem Zeitpunkt ist die Architektur noch nicht endgültig festgelegt, sondern ist typischerweise nur eine Skizze, die die wichtigsten Subsysteme enthält. In dieser Phase werden die größten Risiken identifiziert und hervorgehoben, die Elaboration-Phase wird im Detail geplant und das gesamte Projekt wird grob abgeschätzt.

- Während der ***Entwurfsphase*** werden die meisten Anwendungsfälle des Produktes bis ins Detail ausgearbeitet und die Architektur des Systems festgelegt. Die Architektur ist für das noch vollständig zu entwickelnde System besonders wichtig und elementar. Man kann sich die Beziehung der Architektur zu dem ganzen System durch eine Analogie veranschaulichen: Dabei entspricht die Architektur einem Gerippe, das nur aus Haut und Haaren besteht und das System einem daraus entwickelten, mit Muskeln versehenen Körper. Das Gerippe bzw. die Architektur hat die Aufgabe, den vollständigen Körper bzw. das System zu stabilisieren und die Grundlage für den Körper bzw. des Systems zu sein.

  Deshalb kann man die Architektur als Reduktion bzw. Vereinfachung aller Modelle, die ja zusammen das ganze System darstellen, auf das Wesentliche zusammengeschrumpft denken. Daraus folgt, daß es für das Anwendungsfall-Modell, das Analyse-Modell, das Design-Modell, das Implementation-Modell und das Einsatz-Modell ein entsprechendes, auf das wesentliche zusammengeschrumpftes Modell gibt. Das in der Architektur enthaltene reduzierte Implementationsmodell muß jedoch noch all die Komponenten enthalten, die für die Ausführbarkeit und dadurch für die Überprüfbarkeit der Architektur benötigt werden. In dieser Phase werden auch die Anwendungsfälle mit den größten Risiken, die in der Etablierungsphase

identifiziert wurden, implementiert. Das Ergebnis dieser Phase ist eine grundlegende Architektur (engl.: architecture ***baseline***), auf die die folgende Phase aufbauen wird.

Am Ende der Entwurfsphase hat der Projektmanager die Aufgabe, die weiteren Aktivitäten zu planen und die notwendigen Resourcen abzuschätzen, die noch benötigt werden, um das Projekt fertigzustellen. Die Schlüsselfrage an dieser Stelle lautet für die Verantwortlichen: Sind die Anwendungsfälle, die Architektur und die Pläne stabil genug, und sind die Risiken soweit unter Kontrolle, damit die komplette Softwareentwicklung des Produktes vertraglich festgelegt werden kann ?

- Während der ***Konstruktionsphase*** wird das Produkt gebaut, d.h. die Muskeln (die Software) werden dem Skelett (der Architektur) hinzugetan. In dieser Phase wächst die grundlegende Architektur zu einem vollständigen System, das so weit fertig ist, daß es an die Benutzer ausgeliefert werden kann. Während dieser Phase wird der größte Teil der benötigten Mittel aufgebraucht. Falls die Entwickler bessere Wege finden, um das System zu strukturieren, können noch kleine Änderungen an der stabilen Architektur dem Architekten vorgeschlagen werden. Am Ende dieser Phase beinhaltet das Produkt alle für dieses Release vorgesehenen Anwendungsfälle. Die restlichen Fehler, die in dieser Phase sich noch im Produkt befinden, werden in der Übergangsphase herausgefunden und behoben. Die Schlüsselfrage nach dieser Frage lautet: Stimmt das Produkt mit den Anforderungen der Benutzer genügend überein, daß man das Produkt als erste Version einigen Kunden ausliefern kann?

- Die ***Übergangsphase*** ist die Periode, in der das Produkt das Beta-Stadium erreicht. Das Beta-Release Produkt wird zuerst auf eine kleine Anzahl von erfahrenen Anwendern losgelassen, die die restlichen Fehler und Schwächen herausfinden sollen. Diese Fehler werden von den Entwicklern nun korrigiert und einige der vorgeschlagenen Verbesserungen werden in das endgültige Release noch integriert. Die Aktivitäten der Übergangsphase sind die Herstellung des Produktes, Einweisung von Kundenpersonal, Bereitstellung von Support und die Fehlerbereinigung von Fehlern, die während der Auslieferung gefunden wurden.

C.4 Zusammenfassung des Unified Process

Die UML ist eine normierte Beschreibungssprache für die Softwareentwicklung. Die Art und Weise, wie diese UML-Elemente beim Bau eines Softwareproduktes am Besten zusammenspielen, ist die Aufgabe eines Software-Entwicklungsprozesses wie dem Unified Process. Dieser hängt von den Schlüsselideen „Anwenderfall gesteuert“, „Architektur zentriert“ und „iterative und inkrementelle Entwicklung“ ab. Um diese Ideen in einen Prozeß zu integrieren, müssen folgende Dinge in dem Prozeß enthalten sein: Zyklen, Phasen, Arbeitsflüsse, Risikominimierung, Qualitätskontrolle, Projekt Management und Konfigurationskontrolle. Der Unified Process bildet daher einen Rahmen, der all diese Aspekte in sich integriert. Mit Hilfe dieses Rahmens können Tool-Hersteller und Entwickler Werkzeuge für die Automatisierung des Prozesses, die Unterstützung individueller Arbeitsflüsse, das Bauen all der verschiedenen Arbeitsmodell, u.s.w. entwickeln.

Literaturverzeichnis

[Accretion Power] Juhan Frank, Andrew King, Derek Raine: Accretion Power in Astrophysics. Second Edition. Cambridge University Press, 1992.

[Flebbe] O. Fleppe: Smoothed Particle Hydrodynamics: Modellierung von Superhump Lichtkurven. Dissertation, 1994.

[C++Primer] Stanley B.Lippman/Josée Lajoie: C++ Primer. Third Edition. Addison Wesley, 1998.

[DesignPatterns] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Addison Wesley, 1994.

[Hack & la Dous] M. Hack, C. la Dous: Cataclysmic Variables and Related Objects. NASA/CNRS Monograph Series on Non-Thermal Phenomena in Stellar Atmospheres. 1993.

[Kernighan] Brian W. Kernighan, Dennis M. Ritchie: Programmieren in C. Zweite Auflage. Prentice-Hall International Inc., London, 1990.

[Landau] L.D. Landau, E.M. Lifschitz: Lehrbuch der Theoretischen Physik VI (Hydrodynamik). Akademie Verlag, Leipzig, 1991.

[Monaghan] J.J. Monaghan, R.A. Gingold: Shock simulation by the particle method SPH. J. Comput. Phys., 52,374(1983).

[Numerical Recipes] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, Second Edition 1992.

[Quibeldey-Cirkel] Klaus Quibeldey-Cirkel: Entwurfsmuster: Design Patterns in der objektorientierten Softwaretechnik. Springer-Verlag, Berlin Heidelberg 1999.

[Speith] Roland Speith: Untersuchung von Smoothed Particle Hydrodynamics anhand astrophysikalischer Beispiele. Dissertation, Tübingen, 1998.

[Stroustrup] Bjarne Stroustrup: Die C++ Programmiersprache. Dritte Auflage. Addison Wesley, 1998.

[The Unified Software Development Process] Ivar Jacobson, Grady Booch, James Rumbaugh: The Unified Software Development Process. Addison Wesley, 1999.

[UML-ReferencManual] James Rumbaugh, Ivar Jacobson, Grady Booch: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[UML-UserGuide] Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide. Addison Wesley, 1999.

Danksagung

Ich möchte mich ganz herzlich bei Prof. Dr. Hanns Ruder und Prof. Dr. Wolfgang Rosenstiel bedanken, die mir diese Diplomarbeit ermöglicht haben und bei Stefan Hüttemann für die Betreuung dieser Diplomarbeit.