UNIKASSEL VERSITÄT

Informatik

SOFTWARE-PATTERN MIT C++ UND OPENMP

Abschlussarbeit zum Diplom 1

an der Universität Kassel

offiziell abgegeben am

23.08.2007

vorgelegt von

Florian Bachmann

Betreuung durch

Prof. Dr. Claudia Leopold

und

Dipl.-Inf. Michael Süß

Fachbereich Informatik

Research Group Programming Languages / Methodologies

Universität Kassel

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Kassel, den 23.08.2007

_____

Florian Bachmann

INHALTSVERZEICHNIS

1. EINFÜHRUNG

> *Wer parallele Programme schreibt, kann langsamer werden;*
> *wer keine parallelen Programme schreibt, ist schon zu langsam.*
>
> *(Bachmann, 2007)*

Das Schreiben paralleler Programme wird weitestgehend vermieden und führt ein Nischendasein. Dies erscheint merkwürdig, da das richtige Leben auch parallel abläuft.

Vom Standpunkt des Programmierers aus gesehen haben Computer in den letzten 20 Jahren fast ausschließlich nur sequenziell gearbeitet, in einer linearen Abfolge, mit einem Arbeitsschritt pro Zeitpunkt. Dies ist für Menschen eine eher unnatürliche Vorgehensweise. Im Vergleich dazu vermögen Menschen, komplexer vorzugehen. Sie sind in der Lage, mehrere Dinge auf einmal zu erledigen und wahrzunehmen. Schließlich müssen sie sich in einer simultan ablaufenden Welt zurechtfinden und in dieser leben. In dieser Welt geschehen nahezu alle Vorgänge simultan, wobei in dieser Arbeit jedoch der Begriff „parallel" vorgezogen wird [nach Mattson05].

Schon länger existieren große und kostspielige parallele „Supercomputer", die hauptsächlich für rechenintensive wissenschaftliche Berechnungen, wie z. B. in der Physik, bei Wetterberechnungen, auf der Suche nach außerirdischem Leben [Seti07], bei Matrixmultiplikationen oder als Gegner menschlicher Schachgroßmeister genutzt werden [Deep07]. Parallele Programme gewinnen immer mehr an Relevanz und viele Probleme sind allein von ihrem Umfang her betrachtet nur parallel zu lösen. Man denke nur an Hollywood, das einen immer größer werdenden Bedarf an parallelen Systemen hat, mit denen es für den Kinogänger noch schönere, noch realistischere Filme rendern kann [Manthey98]. Die Möglichkeit, mit diesen Rechnern arbeiten zu dürfen, blieb jedoch bisher nur einer kleinen Gruppe von Personen vorbehalten.

Dies hat sich in den letzten Jahren geändert. Die Entwicklung paralleler Hardware hat gerade im Heimanwenderbereich starke Fortschritte gemacht. „Dual Core" Prozessoren finden sich mittlerweile in fast jedem aktuell im Handel erhältlichen Computer. Multicore Prozessoren wurden mit der Einführung der Xbox360[1] und der Playstation 3[2] noch mehr in den Massenmarkt integriert [Rauber07, S. 100, 107]. Parallele Programmierung verspricht extreme Geschwindigkeitsverbesserungen gegenüber einer sequenziellen Implementierung – gerade bei sehr rechenintensiven Problemen.

---

[1] Die Xbox 360 besitzt drei Cores, auf denen jeweils zwei Threads laufen können.

[2] Die Playstation 3 ist mit insgesamt 9 Prozessoren, Cores genannt, ausgestattet.

> *„Multi cores are pointless and useless without "parallel programming".“*
> *(Butenhof, 2007 [Butenhof07])*

Jeder, der möchte und interessiert ist, kann mittlerweile auf parallele Hardware zugreifen und Software für diese entwickeln. Leider, wie eingangs schon angedeutet, wird diese Möglichkeit nur von sehr wenigen Personen wahrgenommen. Es stellt sich die Frage, warum das so ist.

Ein korrektes paralleles Programm zu schreiben, stellt eine völlig neue Herausforderung gegenüber einem „normalen“ sequenziellen Programm dar. Softwareentwicklung ist ein hochkomplexes Feld, bei dem viele Aspekte beachtet werden müssen und das mannigfaltige Herausforderungen und Schwierigkeiten in sich birgt.

> *„C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg!“*
> *(Stroustrup, 2002)*

Dieses für C++ gültige Zitat lässt sich leicht auf den Unterschied zwischen sequenzieller und paralleler Programmierung übertragen. Jedoch ist es bei paralleler Programmierung nicht unbedingt schwer, sich ‚das Bein wegzuschießen‘.

Der Programmierer – und er ist immerhin die wichtigste Person, wenn es darum geht, die parallele Hardware zum Leben zu erwecken - muss zusätzlich zu allen Herausforderungen, die er auch beim Schreiben sequenzieller Programme beachten muss, die parallelen Anforderungen berücksichtigen: Daten müssen aufgeteilt, Datenabhängigkeiten beachtet, Deadlocks vermieden, Raceconditions bekämpft, evtl. entstehender Overhead muss beachtet werden sowie Threadsicherheit gewährleistet sein, um nur einige Beispiele schlagwortartig aufzuführen, womit man sich bei der täglichen Programmierarbeit auseinandersetzen muss.

Dabei gerät man in Gefahr, sich auf eigentlich nebensächlichen Aufgabenfeldern zu verlieren, ohne sich wirklich um das hauptsächliche Problem kümmern zu können: nämlich das Lösen einer bestimmten Aufgabe! Gerade diese vielen Faktoren, die beachtet werden müssen, können vom eigentlichen Problem ablenken, es komplizieren oder es unnötig aufblähen. Dies schreckt viele Programmierer ab, parallele Programme zu entwickeln. Es müssen also Möglichkeiten gefunden werden, mehr Programmierer für das Schreiben paralleler Programme zu motivieren. Ein möglicher Weg, dies zu erreichen, wäre die Aufmerksamkeit des Programmierers zurück auf die eigentliche Problemstellung zu lenken. Programmierer müssten allein schon aus dem Grund parallele Programme schreiben, damit die gesamte zur Verfügung stehende parallele Hardware letztendlich genutzt werden kann.

Zieht man nun die vom eigentlichen Problem ablenkenden Faktoren heraus, erkennt man, dass bestimmte Strukturen immer wieder auftreten. Seien es die Aufteilung von Datenelementen, das Vermeiden von Datenabhängigkeiten oder Raceconditions. Diese Strukturen können als Pattern formuliert werden.

In der Softwarebranche ist seit längerem der Sinn und Nutzen sogenannter „Design Pattern“ bekannt. Design Pattern beschreiben eine abstrahierte Problemlösung. Diese Problemlösung ist nicht an irgendeine

Syntax gebunden. Somit sollte es möglich sein, sie in jeder Sprache[3] zu implementieren (wobei der Implementierungsaufwand stark von den gebotenen Möglichkeiten der verwendeten Sprache abhängt). Viele Software-Entwurfsentscheidungen ergeben sich nahezu automatisch, wenn das zu verwendende Pattern bekannt ist. Gerade Benutzer der Programmiersprache Java nutzen (vielleicht unbewusst) Pattern täglich und ganz selbstverständlich, da viele dort direkt durch den Sprachkern angeboten werden.

Diese Design Pattern lassen sich auch auf parallele Problembereiche übertragen. Sie sollen helfen, dem Entwickler das Erstellen paralleler Programme zu erleichtern. Sie bieten auf einer hohen Abstraktionsebene Lösungen an, die im besten Fall „Out-of-the-Box" genutzt werden können. Diese Lösungen stellen Erfahrungswerte dar, die bei immer wieder auftretenden Problemen und Aufgabenstellungen helfen, Design-Entscheidungen zu fällen. Dadurch wird das Erstellen guter und korrekter paralleler Programme erleichtert, da Entwickler auf bereits bewährte und getestete Lösungen zurückgreifen können.

Ziel dieser Arbeit ist es nicht, neue Pattern zu entdecken und diese in den mittlerweile sehr umfangreichen Pattern-Katalog einzureihen, sondern schon bestehende Pattern auf parallele Systeme zu übertragen und zu prüfen, wie leicht oder schwer sich diese Pattern mit dem Gespann C++ und OpenMP realisieren lassen.

So soll dazu beigetragen werden, die Aufmerksamkeit des Programmierers wieder auf die eigentliche Problemlösung zu lenken:

> *"…we are in the middle of a revolution right now! It is a parallel revolution, and this time it is for real."*
> *(Süß, 2006 [Süß06b])*

Die in dieser Arbeit vorgestellten Pattern sollen einen Teil des Handwerkszeugs darstellen, diese ‚Revolution' durchzuführen.

[3] Das Wort Sprache ist im Folgenden synonym mit Programmiersprache zu lesen.

1.1. STRUKTUR DER ARBEIT

Der Schwerpunkt von Kapitel 2 liegt auf den in der Arbeit verwendeten Grundbegriffen, die nötig sind, um den Rest dieser Arbeit zu verstehen. Dabei wird besonderes Augenmerk auf den Pattern-Begriff und seine verschiedenen Ausprägungen gelegt. Desweiteren werden die Grundbegriffe des parallelen Programmierens gestreift und vorgestellt welche Techniken mit C++, OpenMP und Template hinter den in dieser Arbeit vorgestellten Pattern stehen.

In Kapitel 3 folgt ein kurzer Exkurs zu der AthenaMP-Bibliothek. Sie soll das Schreiben paralleler Programme erleichtern, indem sie häufig benötigte Strukturen als einfach zu verwendende „Pattern“ anbietet. Alle in dieser Arbeit implementierten Design Pattern sollen sich nahtlos in diese Bibliothek integrieren und sie so erweitern und vervollständigen.

Den Hauptteil der Arbeit ab Kapitel 4 stellen die fünf für C++ und OpenMP umgesetzten Design Pattern dar. Jedes einzelne Pattern wird detailliert vorgestellt, zudem wird auf dessen Implementierung, seine Vor- und Nachteile und die Verwendung eingegangen.

Diese Arbeit schließt in Kapitel 5 mit einer kurzen Zusammenfassung und gibt einen Ausblick über den weiteren Sinn und Zweck paralleler Pattern.

Anmerkung:

Im Zweifel werde ich immer die englischen Fachbegriffe einer deutschen Übersetzung vorziehen, um eine genaue und eindeutige Terminierung zu ermöglichen. Sollte ein englischer Begriff unklar sein, wird beim ersten Auftreten des Wortes eine sinngemäße Übersetzung des Begriffs in Klammern angeboten.

Unklare Fachbegriffe werden, sofern sie nicht im Kontext erläutert werden, kurz im Glossar auf Seite 70 erklärt.

2. GRUNDLAGEN UND GRUNDBEGRIFFE

Im folgenden Kapitel werden Grundbegriffe und Ideen geklärt, um ein umfassendes Verständnis des „Pattern"-Begriffs zu ermöglichen.

2.1. DER BEGRIFF „PATTERN"

> *„Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen"*
> *(Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, & Angel, 1977)*

Ein Pattern (im Folgenden auch Entwurfsmuster oder Design Pattern genannt) beschreibt eine bewährte Lösung für ein Problem. Es stellt damit eine wiederverwendbare Vorlage zu einer Problemlösung dar.

Geprägt wurde der Pattern-Ausdruck durch den Architekten und Mathematiker Christopher Alexander in den 70er Jahren, der einen Musterkatalog für Architekten unter dem Titel „A *Pattern Language*" („*Eine Muster-Sprache*") zusammenstellte [Alexander77]. Das Ziel seiner formulierten Pattern war es, Elemente der Architektur in elementare Pattern und ihre Beziehungen untereinander aufzulösen. Die Basis dieser Pattern war, eine humane und menschengerechte Architektur zu schaffen, die die elementaren Bedürfnisse der Menschen optimal widerspiegelt und befriedigt.

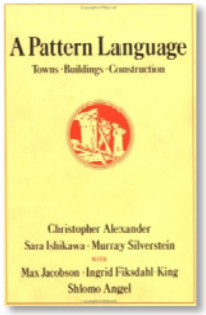


Abbildung 2.1.1: „A Pattern Language" von Alexander.

In seinem eigenen Berufsstand wurden die von Alexander und seinem Mitstreitern vorgestellten Pattern, es waren immerhin 253(!) verschiedene, bei weitem nicht so gut aufgenommen wie in der Softwareentwicklung. Viele Kritiker empfanden Alexanders Idee als zu banal, zu logisch und zu einfach klingend und warfen ihm vor, nur das Offensichtliche niedergeschrieben zu haben [nach Bienhaus; Design Pattern Vorlesung; Semester WS 2005/2006, Uni Kassel].

2.2. PATTERN NACH GAMMA

Alexanders Ideen wurden enthusiastisch von Erich Gamma aufgenommen, der sie auf die Informatik übertrug und damit die heutige Methodik der Softwareentwicklung wesentlich beeinflusste. Erich Gamma legte im Jahre 1995 zusammen mit seiner „Gang of Four" (kurz *GoF:* Viererbande), bestehend aus Richard Helm, Ralph Johnson, John Vlissides und ihm selbst, ein Standardwerk der Informatik vor. Es heißt „*Design Patterns - Elements of Reusable Object-Oriented Software*" [Gamma95]. Man kann guten Gewissens behaupten, dass dieses Werk die Basis der Arbeit darstellt.

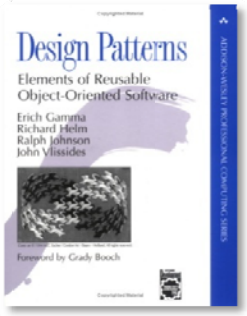


Abbildung 2.2.1: „Design Patterns" von Gamma et al..

Gamma erkannte, dass das in der Softwarebranche angesammelte Erfahrungswissen nicht an die Lehre weitergegeben wurde. Er führt an, dass Experten dadurch zu solchen werden, dass sie sich aufgrund ihrer Erfahrung Wissen angeeignet haben, mit dem sie beurteilen können, was funktioniert und was fehlschlägt. Hat jemand für sich eine funktionierende Lösung gefunden, wird er versuchen, diese auch als Grundlage für viele weitere und insbesondere ähnliche Probleme immer wieder anzuwenden, anstatt jedes Mal „das Rad neu zu erfinden“. Gamma schrieb diese Erfahrungen als Design Pattern nieder. Jedes – seiner damals 23 vorgestellten – Pattern benennt, erläutert, bewertet und katalogisiert einen wichtigen und wiederkehrenden Entwurf eines objektorientierten Systems.

Ein Pattern sollte nach Gamma *et al.* folgende Kriterien erfüllen:

- ein oder mehrere Probleme lösen
- ein erprobtes Konzept bieten
- über das rein Offensichtliche hinausgehen
- den Benutzer in den Entwurfsprozess einbinden
- Beziehungen aufzeigen, die tiefer gehende Strukturen und Mechanismen eines Systems umfassen

Man erkennt, dass das einführende Zitat Alexanders, indem er definiert, was Pattern ausmacht, auch für objektorientierte Pattern oder noch allgemeiner für Software-Pattern gilt. Pattern, wobei es keinen Unterschied macht, ob sie nach Gamma *et al.* oder nach Alexander *et al.* spezifiziert werden, sind als Problemlösungen für bestimmte Situationen zu verstehen.

Es wurde eben schon herausgearbeitet, dass ein Pattern eine abstrahierte Lösung eines Problems darstellt. Objektorientierte Pattern stellen typischerweise die Beziehungen und Interaktionen von Klassen oder Objekten dar, ohne dabei festzulegen, welche Klassen oder Objekte letztendlich im Source-Code implementiert werden. Algorithmen werden nach Gamma *et al.* und im Allgemeinen nicht als Design Pattern betrachtet, da sie meist dazu dienen, Berechnungen durchzuführen oder mathematische Probleme anstelle von Design-Problemen zu lösen. Was jedoch genau ein Pattern ist und was nicht, hängt von der Auffassung und Betrachtungsweise des Benutzers ab.

2.3. PARALLELES PROGRAMMIEREN

> *„You really just need to cast off your illogical and unnatural preconceptions that only one thing happens at a time (no child would last long in the real world with such ideas!) and embrace asynchrony.“*
> *(Butenhof, 2007 [Butenhof07])*

Mit der Massenmarkteinführung der MultiCore Systeme wurde die von David Butenhof erwähnte erzwungene Sequentialisierung bestehender Computersysteme zugunsten leichter verfügbarer Parallelität aufgelockert.

Das Ziel von multiplen Prozessoren ist es, Probleme in weniger Zeit zu lösen und/oder größere, komplexere Probleme, als es auf einer Ein-Prozessor-Maschine möglich wäre, zu lösen. Der Erfolg von parallelen Berechnungen beruht auf Nebenläufigkeit. Diese Nebenläufigkeit gilt es zu finden, aus der Problemstellung zu extrahieren und in ein parallel laufendes Programm zu „gießen“. Was sich auf dem Papier relativ einfach anhört, ist in der Praxis für den Programmierer ein schwieriges und komplexes Unterfangen.

Hierfür wird versucht, ein Problem in mehrere kleine Probleme zu partitionieren. Diese kleineren Teilprobleme sollten nach Möglichkeit voneinander unabhängig laufen können, also parallel, und dabei viele kleine Teillösungen liefern, die zusammengenommen die Lösung des Problems darstellen. Die Teilprobleme, die die Teillösungen liefern, nennt man auch Teilaufgaben. Diese Teilaufgaben bezeichnet man in der Literatur auch als Tasks. Die verschiedenen Tasks müssen nun als Programmcode implementiert und auf einem parallelen System ausgeführt werden, um zu verifizieren, ob die gewählte Problempartitionierung auch die gewünschte Lösung liefert. Dieses Verteilen der Tasks auf verschiedene Prozessoren wird in der Literatur als Mapping bezeichnet. Dies kann dabei statisch zur Compile-Zeit oder dynamisch zur Laufzeit erfolgen. Die Ausführungszeit des daraus resultierenden Programms müsste sich auf einem parallelen System (z. B. einem Mehrprozessorsystem) gegenüber einer sequenziellen Lösung des Problems verkürzen. Dieser eventuell erreichte Geschwindigkeitsgewinn wird in der Literatur als Speedup bezeichnet.

Parallele Programmierung birgt einzigartige Herausforderungen. Häufig besitzen die Tasks, die das Problem lösen, Abhängigkeiten untereinander, die identifiziert und korrekt gemanagt werden müssen. Diese Daten- und Kontrollabhängigkeiten bestimmen die Ausführungsreihenfolge, in der die Tasks bearbeitet werden können. Liegen keine Daten- und Kontrollabhängigkeiten vor, d. h., wenn die Task komplett unabhängig voneinander arbeiten können, bezeichnet man diese als „Embarrassingly Parallel“ (beschämend Parallel).

Für den Fall, dass Datenabhängigkeiten existieren, was für gewöhnlich der Fall ist, so besteht die Aufgabe des Programmierers darin, diese Abhängigkeiten durch Synchronization zu regeln und eventuell konkurrierende Datenzugriffe auf einen gemeinsamen Speicher zu steuern. Diese Synchronisation ist wichtig, da das Laufzeitverhalten der Tasks, bei den von uns verwendeten Architekturen mit gemeinsamem Adressraum als Threads bezeichnet, nicht deterministisch ist und vom Scheduler bestimmt wird, auf den wir keinen bis nur geringen Einfluss haben.

Es kann vorkommen, dass selbst korrekte parallele Programme den versprochenen Geschwindigkeitsgewinn entweder aufgrund bestimmter Abhängigkeiten oder durch Parallelisieren des falschen Bereichs nicht erfüllen können. Auch ist es oftmals schwer, die Arbeit gleichberechtigt unter den an der Berechnung teilnehmenden Prozessoren gleichmäßig zu verteilen („Load Balancing“). Dies ist nötig um ein Lastgleichgewicht zu erzeugen. Die Größe der einzelnen Teilaufgaben, Granularität genannt, hat dabei einen großen Einfluss auf eine gleichmäßige Lastenverteilung und somit auch auf eine gleichmäßige Auslastung der für die Aufgabe zur Verfügung stehenden Prozessoren. Dabei ist zu beachten, dass der vom Programm erzeugte Overhead, um die Parallelität zu verwalten, die eigentliche Berechnung nicht übersteigt.

Korrekte parallele Programme zu schreiben, erfordert deshalb eine große Sorgfalt seitens des Programmierers.

Eine gute und umfassende Einführung zu dem Thema bieten die Bücher „Introduction to parallel Computing“ von Grama *et al.* [Grama03] und „Parallele Programmierung“ von Rauber *et. al.* [Rauber07].

2.4. C++, OPENMP & TEMPLATES

Diese Diplomarbeit bedient sich der OpenMP-Erweiterung von C++ für paralleles Programmieren. OpenMP ist ein „Application Programming Interface“ (API). Es stellt einen Satz von Compilerdirektiven für die Programmierung von Shared-Memory-Systemen zur Verfügung. Zur Zeit existieren OpenMP-Spezifikationen für FORTRAN, C, und C++.

OpenMP wird häufig verwendet, um in inkrementeller Weise parallele Abschnitte zum sequenziellen Code hinzuzufügen. Wenn eine Compilerdirektive wie `#pragma omp parallel for` um eine for-Schleife geschrieben wird, kümmert sich der Compiler um alle Details wie Thread-Erzeugung und -Verwaltung, so dass diese for-Schleife parallel ausgeführt wird.

Eine interessante Eigenschaft von OpenMP ist der korrekte Programmablauf (abgesehen von Ausnahmen) trotz der fehlenden Unterstützung von OpenMP auf dem Zielsystem, da Compilerdirektiven in diesem Fall als Kommentare gewertet werden. Eine zuvor auf mehrere Threads aufgeteilte for-Schleife würde in diesem Fall nur von einem Thread durchlaufen werden.

OpenMP-Programme laufen sehr gut auf Architekturen mit gemeinsamem Hauptspeicher (SMP). Dadurch, dass das OpenMP zugrunde liegende Speichermodell keine Unterscheidung zwischen der Zugriffszeit von Non-Uniform-Speicher und Uniform-Speicher bietet, ist es weniger für NUMA oder Distributed-Memory Rechner geeignet.

Damit die in dieser Arbeit vorgestellten Pattern auf ein möglichst breites Anwendungsspektrum passen, wurde nach Möglichkeit der Einsatz von Templates forciert. Durch Templates (Schablonen) ist der Code nicht mehr an bestimmte Datentypen gebunden. Sie ermöglichen die Trennung von Datenstrukturen und Algorithmen und somit generische Programmierung. Generische Programmierung wird verwendet, um generische Komponenten zu erstellen. Diese helfen bei der Entwicklung wiederverwendbarer Software-Bibliotheken.

Unter Einsatz dieser Techniken wurden fünf Pattern, die in Kapitel 4 beginnend auf Seite 14 im Detail vorgestellt werden, realisiert. Diese Pattern sollen sich in die in Kapitel 3 vorgestellte AthenaMP-

Bibliothek integrieren. Auf diese Weise soll eine leicht wiederverwendbare Software-Bibliothek für parallele Programmierung entstehen.

2.5. PATTERN NACH MATTSON

Timothy Mattson führt in seinem Buch „Patterns for parallel Programming“ [Mattson05] zusammen mit Beverly Sanders und Berna Massingill eine erweitere Pattern-Definition gegenüber Gamma *et al.* ein. Der Pattern-Begriff bezeichnet bei ihm im Gegensatz zu dem von Gamma geprägten Pattern-Begriff nicht nur abstrakte Lösungen als Pattern, sondern auch konkrete Implementierungen häufig benötigter Algorithmen.

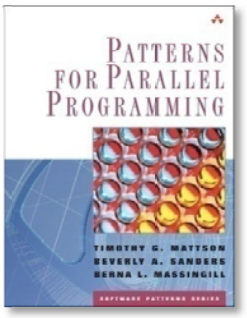


Abbildung 2.5.1: „Patterns for parallel Programming“ von Mattson et al..

Desweiteren führen Mattson *et al.* eine Pattern-Sprache ein, um dem Programmierer die Parallelisierung eines Problems zu erleichtern. Sie ist dem bekannten Wasserfallmodell ähnlich, spezialisiert sich jedoch auf das Erstellen paralleler Programme. Dafür gibt er eine Art „Kochrezept“ vor, dessen Anweisungen befolgt werden müssen, um ein Programm perfekt zu parallelisieren. Natürlich nur für den Fall, sofern es überhaupt parallelisierbar ist.

3. EXKURS ATHENAMP

> *„AthenaMP is a library to show some easy and powerful programming techniques in OpenMP and C++. It is intended as a resource that can be employed as is with the provided interfaces, but the user is also encouraged to look under the hood and change our easy to comprehend code coupled with to-the-point documentation.“*
> *(AthenaMP, 2007)*

3.1. EINFÜHRUNG

Objektorientierte Programmierung, Design Pattern und Frameworks bzw. Bibliotheken sind allgemein angewandte Techniken, um die Komplexität serieller Programme zu reduzieren. AthenaMP unternimmt den Versuch, diese Techniken auf parallele Systeme zu übertragen.

AthenaMP ist eine Bibliothek, die es Programmierern erleichtern soll, häufig wiederkehrende parallele Strukturen und Probleme einfacher zu lösen. Ihr Schwerpunkt liegt dabei auf der Verwendung von OpenMP 2.5 in Kombination von C++, Templates und objektorientierter Programmierung.

Der Name AthenaMP leitet sich von der griechischen Göttin Athene ab. Athene galt als die Göttin der Weisheit sowie der Kriegstaktik und Strategie und war außerdem die Schirmherrin der Künste und der Wissenschaften.

Abbildung 3.1.1 zeigt Athena. Der Ausschnitt ist ein Teil des Deckengemäldes des Göttweiger Klosters in Österreich von Paul Tröger.

Sie besaß also alle Eigenschaften, die man bei einer täglichen Programmierarbeit paralleler Systeme mehr als dringend benötigt. An dieser Stelle nun soll AthenaMP dem Programmierer unterstützend zur Seite stehen. AthenaMP bietet für einige der Schwierigkeiten und Komplexitäten, die es beim Schreiben paralleler Programme zu meistern gilt, Lösungen an, die einfach benutzt werden können. AthenaMP stellt eine Sammlung verschiedener paralleler Design Pattern dar und kann somit als Toolbox betrachtet werden, aus der der Programmierer ein für sein Problem nützlich erscheinendes Werkzeug wählen kann.

Ziele der AthenaMP-Bibliothek sind [ATHENA07]:

1. Sie soll zeigen, wie ausgewählte Pattern unter Verwendung von OpenMP / C++ threadsicher umgesetzt werden können.
2. Sie soll „Out-of-the-Box“ und ohne große Anpassungen nutzbar sein, ohne dass ein Nutzer jedes Implementierungsdetail kennen muss. Ihre offene Struktur (BSD Lizenz) bietet einem Nutzer aber die Möglichkeit, aufgrund ihrer dokumentierten API jedes Implementierungsdetail kennen-

zulernen und so OpenMP parallelisierte Programme besser zu verstehen und selbst anwenden zu können.

3. Sie soll durch ihren exzessiven Gebrauch von Templates und objektorientierter Programmierung helfen, die existierenden Compiler zu verbessern, indem sie einfach zu handhabende und leicht wiederholbare Test-Szenarios (CppUnit-Tests) für die angebotenen Pattern anbietet. In den Test-Szenarios wird getestet, ob die erwartete Grundfunktionalität eines Patterns garantiert ist.

AthenaMP sieht sich selbst als Forschungsprojekt, das helfen soll, die Stärken und Schwächen von OpenMP zu erkennen und dadurch die Sprache immer weiter zu verbessern, indem Funktionalität angeboten wird, die in dieser Form bisher nicht vorhanden oder schwierig zu implementieren ist.

AthenaMP ist keine Design Pattern Demonstrationsbibliothek. Sie soll parallele Programmierer unterstützen, leichter parallele Programme zu schreiben. Deshalb ist AthenaMP als eine Mischung aus angewandten Design Pattern und einer Template Bibliothek, ähnlich der STL-Bibliothek in C++, zu betrachten. Der Begriff Design Pattern kann Verwirrung stiften, da sie eine Idee und deren abstrakte Lösung, die über eine konkrete Implementierung hinausgeht, darstellen. In dem Buch „*Modern C++ Design – Generic Programming and Design Patterns Applied*“ [Alexandrescu06] ist, wie der Titel schon andeutet, Alexandrescu mit dem selben Problem konfrontiert. Er begründet, die konkrete Implementierung der Pattern damit, dass er die Erzeugung der Pattern-Implementierungen mithilfe generischer Programmierung automatisiert, statt selbst zu versuchen Pattern zu implementieren. Dieser Weg wird auch bei AthenaMP gegangen. AthenaMP implementiert Design Pattern also nicht im softwarehistorischen Kontext nach Gamma *et al.*, sondern richtet sich bei dem Pattern-Begriff eher nach Mattson *et al.* und folgt bei der Umsetzung Alexandrescu.

Das Ziel von AthenaMP besteht darin viele generische Komponenten zu schaffen, denen bereits implementierte Design Pattern zugrunde liegen und die sich somit durch Flexibilität, Wiederverwendbarkeit und leichte Benutzbarkeit auszeichnen [nach Alexandrescu06, Seite xii bis xviii].

Dabei soll AthenaMP eine gut erweiterbare und leicht zu nutzende Bibliothek sein, die dabei hilft, die relativ steile „Lernkurve“ abzuflachen, die sich beim Einstieg in die parallele Programmierung auftut.

3.2. STRUKTUR

Dieses Kapitel stellt die Struktur der AthenaMP-Bibliothek vor. AthenaMP ist modular und „leichtgewichtig“ (lightweight) aufgebaut. Dies bedeutet, dass nahezu jede Komponente einzeln verwendet werden kann, ohne dass starke Abhängigkeiten untereinander vorhanden sind.

Es wird zwischen fünf Kategorien von Pattern unterschieden. Dies hat den Sinn, dass sich die angebotenen Pattern leichter katalogisieren lassen. Das Ziel dieser Diplomarbeit besteht darin, die AthenaMP-Bibliothek um fünf Pattern zu erweitern. Auf diese fünf neuen Pattern wird im Detail in Kapitel 4 eingegangen. Der Schwerpunkt liegt dabei auf der Implementierung, da der Fokus dieser Arbeit darauf liegt, wie gut oder schlecht sich diese Pattern in der Kombination C++ und OpenMP umsetzen lassen.

Im Folgenden wird kurz auf die Schwerpunkte der einzelnen Kategorien eingegangen, und die ihnen in dieser Arbeit vorgestellten Pattern werden zugeordnet:

3.2.1. DATENPARALLELE PATTERN (DATA-PARALLEL)

Als datenparallele Pattern werden Pattern bezeichnet, wenn auf einer Menge von Daten parallel ein bestimmter Task ausgeführt wird. Ein Task bezeichnet eine Operations-Sequenz vergleichbar mit einer Methode. Die Daten werden dabei für gewöhnlich gleichmäßig unter allen partizipierenden Threads verteilt. Jeder Thread führt dann den Task auf den ihm zugeordneten Daten aus.

3.2.2. OBJEKTORIENTIERTE PATTERN (OBJECT-ORIENTED)

Als objektorientierte Pattern werden alle Pattern im Sinne von Gamma *et al.* (vgl. Kapitel 2.2 Seite 5) bezeichnet. Objektorientierte Pattern stellen typischerweise die Beziehungen und Interaktionen von Klassen oder Objekten dar. Der Fokus bei der Implementierung dieser Pattern wurde auf Threadsicherheit gelegt.

- Observer (Kapitel 4.3 ab Seite 29)

3.2.3. SYNCHRONISATIONSPATTERN (SYNCHRONIZATION)

Pattern werden als Synchronisationspattern bezeichnet, bei denen die Zugriffsreihenfolge von Threads auf gemeinsam genutzten Speicher bzw. zeitliche Abhängigkeiten unter den Threads (ordering constraints) eine Rolle spielen. Synchronisationspattern werden auch verwendet, wenn das Programm auf jeden Fall korrekt ablaufen soll, unabhängig davon, wie Threads vom Scheduler ausgeführt werden.

Im weitesten Sinne können die OpenMP-Compilerdirektiven `#pragma omp barrier` sowie `#pragma omp critical` als Synchronisationspattern bezeichnet werden.

- Shared Queue (Kapitel 4.2 ab Seite 22)

3.2.4. TASKPARALLELE PATTERN (TASK-PARALLEL)

Taskparallele Pattern bezeichnen Pattern, bei denen mehrere Tasks auf einer Datenmenge ausgeführt werden, im Gegensatz zu datenparallelen Pattern, bei denen jeder Thread einen Task auf einen Teilbereich der Daten ausführt. Das bedeutet, dass jeder Task auf die komplette Datenmenge, anstatt auf nur einen Teilbereich angewendet wird. Dabei ist es üblich, dass ein Task durch einen Thread abgebildet wird und jeder Task eine andere Aufgabe auf den Daten ausführt.

- Pipeline [not typesafe] (Kapitel 4.4.1 ab Seite 43)
- Pipeline [typesafe] (Kapitel 4.4.2 ab Seite 53)

3.2.5. WEITERE PATTERN (MISCELLANEOUS PATTERN)

Unter diese Kategorie fallen alle Pattern, die sich nicht eindeutig in die vier vorausgehenden Kategorien einordnen lassen.

- threadsichere Datencontainer [Deque, List und Vector] (Kapitel 4.1 ab Seite 15)

4. BESCHREIBUNG DER IMPLEMENTIERTEN PATTERN

> *Muster, es wird vergessen, sind bloß Kostproben von Möglichkeiten.*
>
> *(Emil Baschnonga, (*1941), Schweizer Schriftsteller)*

Nachdem in den vorherigen Kapiteln alle Begriffe und Grundlagen geklärt wurden, wird nun auf die in dieser Arbeit realisierten Pattern im Detail eingegangen. Dieses Kapitel stellt einen Pattern-Katalog dar, ähnlich dem Werk von Gamma *et al.* [Gamma95] oder Mattson *et al.* [Mattson05], wobei jedoch der Begriff des Pattern, wie zuvor schon angedeutet, im historischen Sinn nach Gamma nicht mehr ganz wörtlich genommen werden darf. Die in diesem Kapitel diskutierten Pattern sind bereits von mir in C++ als generische Komponenten [siehe Kapitel 3.1 und Alexandrescu06] implementiert worden, so dass sie ganz leicht in eigenen OpenMP-Programmen verwendet werden können, dabei werden ihre Implementierung, ihr Nutzen sowie ihre Vor- und Nachteile aufgezeigt.

Sofern Laufzeitmessungen angegeben sind, wurden diese auf einem Linuxsystem mit zwei Dual Core Opteron 270 Prozessoren mit jeweils 2 GHz und 2 Gigabyte Arbeitsspeicher durchgeführt. Alle Programme wurden mit dem Intel Compiler 10.0 übersetzt. Aufgeführte Zeitmessungen zeigen immer den besten Wert aus jeweils drei Programmdurchläufen.

Bei der Entwicklung wurde darauf geachtet, dass die neu entstandenen Pattern den Coding Guidelines der bereits bestehenden AthenaMP-Bibliothek folgen sowie jedes vorgestellte Pattern in seiner Kernfunktionalität durch CppUnit-Tests abgedeckt ist.

4.1. THREADSAFE STL-CONTAINER

KURZBESCHREIBUNG:

Dieses Kapitel stellt die aus der STL bekannten abstrakten Datentypen (ADT) List, Deque und Vector als threadsichere parallele Implementierungen vor [Stroustrup01, S. 471, S. 500, S. 504].

MOTIVATION/PROBLEM:

Haben mehrere Threads Zugriff auf einen gemeinsamen Speicher, so müssen diese Speicherzugriffe geschützt werden. Mattson [Mattson05. S. 15] empfiehlt Programmierern, nach Möglichkeit auf gemeinsamen Speicher zu verzichten und alle Speicherzugriffe ‚private' – also lokal – für jeden Thread zu gestalten. Jedoch ist dieses Vorgehen bei manchen Programmen nicht möglich, da die Benutzung eines gemeinsamen Speichers nahezu von der Problemlösung diktiert wird.

Angenommen, es existiert eine global genutzte Datenstruktur, die als Liste implementiert ist, so bietet diese Liste eine Funktion `takeFirstFromList()` an, die ein Element aus einer Liste zurückliefert und es aus dieser entfernt. Die angebotene Funktion `takeFirstFromList()` könnte sich aus folgenden Elementaroperationen zusammensetzen:

1. Das erste Element der Liste ausfindig machen und zwischenspeichern.
2. Die Referenz des ersten Elements auf das zweite Element der Liste setzen (das zweite Element wird so zur neuen Nummer Eins).
3. Das ursprüngliche erste Element löschen.
4. Die Größe der Liste aktualisieren.
5. Das in Schritt 1. gemerkte erste Element zurückliefern.

Sollten jetzt zwei Threads gleichzeitig die `takeFirstFromList()` Funktion ausführen, könnte es passieren, dass diese nicht geschützten – und somit nicht threadsicheren – Elementaroperationen nicht atomar ausgeführt werden. Eine nicht atomare Ausführungsreihenfolge führt schnell zu einer inkonsistenten Liste. Es können die verschiedensten Probleme auftreten, wenn mehrere miteinander konkurrierende Threads auf gemeinsam genutzte Datenstrukturen zugreifen.

In einem solchen Fall müssen die Zugriffe auf eine gemeinsam genutzte Datenstruktur in jedem einzelnen Thread geschützt werden. Threadsicherheit muss gewährleistet sein! Der OpenMP-Standard führt dazu folgende Bedingung auf:

> *„All library, intrinsic and built-in routines provided by the base language must be threadsafe in a compliant implementation."*
>
> *(OpenMP Standard 2.5, S. 13)*

Im Klartext heißt das, dass alle von der Basissprache angebotenen Datenstrukturen, in unserem Fall C++, threadsicher sein müssen. Dazu zählen auch die in der STL zur Verfügung gestellten elementaren Datenstrukturen wie Deque, Vector, List und Map.

Jedoch ist leider nicht – wie vom OpenMP-Standard gefordert – gewährleistet, dass die von der STL dem Benutzer zur Verfügung gestellten Datenstrukturen threadsicher sind. Die in diesem Kapitel vorgestellten Datencontainer versuchen nun, die geforderte Threadsicherheit nachzurüsten.

IMPLEMENTIERUNG:

Im Folgenden sollen threadsichere Implementierungen einer Deque, einer List und eines Vectors vorgestellt werden. Sie heißen `deque_ts`, `list_ts` und `vector_ts` Das abschließende „ts“ deutet darauf hin, dass es sich um threadsichere (Threadsafe) Implementierungen handelt. Hierfür werden die bestehenden STL-Implementierungen dieser drei Container [Louis03 und Stroustrup01] durch Anwendung des von Gamma *et al.* vorgestellten Decorator Design-Patterns [Gamma95, S. 199] gekapselt und die Methodenaufrufe durch Locks geschützt.[4]

```
01  template<class Type, class Ax_ = std::allocator<Type> >
02  class list_ts {
03  private:
04      std::list<Type> elems_; //Datenstruktur, die alle Daten enthält
05      ...
06  }
```

Listing 4.1.1 Beispiel anhand `list_ts`, wie die korrespondierende STL-List gekapselt ist

Das Sourcecode-Beispiel zeigt am Beispiel der AthenaMP `list_ts` wie die STL-List gekapselt wird (Listing 4.1.1). Der Template-Parameter `Type` gibt dabei den Datentyp der in der Liste zu verwaltenden Objekte an. Intern werden immer die von der STL bereitgestellten jeweilig korrespondierenden Datencontainer genutzt. (Der zweite Template-Parameter `Ax_` wird nur der Vollständigkeit halber benötigt, sofern die Liste mit einem eigenen Speicher-Allokator initialisiert werden soll.)

Anstatt auf die von OpenMP angebotenen Locks zurückzugreifen, wurde der Lock-Mechanismus der AthenaMP-Bibliothek verwendet. AthenaMP bietet Lock-Adapter an, die das Interface des vom OpenMP angebotenen Sprachkerns in einer leichter zu nutzenden Klasse kapseln (siehe AthenaMP-Dokumentation [Athena07]). Listing 4.1.2 zeigt am Beispiel der `resize()`-Methode wie diese durch von AthenaMP zur Verfügung gestellte Locks umschlossen und in einer `resize()`-Methode von `list_ts` gekapselt wird, um Threadsicherheit zu erreichen.

---

[4] *Anmerkung:* Alle drei Datenstrukturen besitzen viele Gemeinsamkeiten und basieren in ihrer Realisierung auf den gleichen Grundideen. Deshalb wird der ihnen zugrunde liegende Aufbau nur einmal allgemein und beispielhaft erörtert, so dass diese Arbeit keine unnötigen Redundanzen erhält.

```
01  void resize(size_type count_) {
02      lock_write_.set ();
03      elems_.resize(count_);
04      lock_write_.unset ();
05  }
```

Listing 4.1.2: Nutzung von Locks anhand des `resize()`-Befehls.

Als zweites unterstützendes Pattern der AthenaMP-Bibliothek wird das Guard-Objekt genutzt (Listing 4.1.3). Für gewöhnlich werden wie im vorherigen Beispiel `set()` und `unset()`-Lock direkt aufgerufen. In manchen Fällen ist es nicht möglich, den Lock mit `unset()` wieder zu lösen, wenn zum Beispiel mit **`return`** aus dem aktuellen Scope gesprungen wird. Das Guard-Objekt beendet den Lock valide, obwohl mit **`return`** aus dem Scope gesprungen wird, indem es während des Destruktor-Aufrufes den Lock mit `unset()` wieder entsperrt.

```
01  iterator erase(iterator where) {
02      guard my_guard (lock_write_);
03      return elems_.erase (where);
04  }
```

Listing 4.1.3: Beispiel für die Nutzung des Guard-Objekts..

Dies schafft threadsichere Datenstrukturen, die analog zu den in der STL angebotenen Datenstrukturen genutzt werden können.

ANWENDUNGSBEISPIEL:

Um bei den STL-Datencontainern Threadsicherheit zu garantieren, muss gewährleistet sein, dass nur ein Thread auf einmal zugreifen darf. Damit dies ermöglicht werden kann, müsste der Programmierer alle Zugriffe „per Hand“, also nicht automatisiert, durch Locks schützen, um die geforderte Threadsicherheit zu erlangen. Der direkte Vergleich des STL-Vectors und des AthenaMP `vector_ts‘` zeigt den Implementierungsunterschied:

```
01  #include <vector>
02  omp_lock_t lock_;
03  std::vector <int> stl_vec_;
04  ...
05  // ab hier parallele Region:
06  // nachfolgender gemeinsam genutzter Speicher muss geschützt sein
07  omp_set_lock (&lock_);
10  stl_vec_.push_back(42);
11  omp_unset_lock (&lock_);
```

Listing 4.1.4 STL-Vector.

```
01  #include "athenamp.hpp"
02
03  vector_ts<int> ts_vec_;
04  ...
05
06
07
10  ts_vec_.push_back(42);
11
```

Listing 4.1.5 AthenaMP Vector TS.

VOR- UND NACHTEILE/NUTZEN:

Man erkennt, dass der Aufwand, den – in Listing 4.1.4 genutzten – STL-Vector threadsicher zu machen, den Code relativ stark aufbläht. Die in AthenaMP zur Verfügung gestellten Implementierungen der STL-Container erleichtern den Umgang mit Datenstrukturen, die threadübergreifend in parallelen Programmen genutzt werden, erheblich (Listing 4.1.5). Der Programmierer muss nun nicht mehr darauf achten, dass alles threadsicher ist und alle Locks richtig gesetzt und wieder gelöst werden. Zusätzlich bieten die

hier vorgestellten Container alle bekannten Vor- und Nachteile, die von den STL-Containern bekannt sind und die auch bei Stroustrup [Stroustrup01, S. 465] nachgelesen werden können.

Da alle AthenaMP-Container auf den STL-Containern beruhen, gelten für sie auch dieselben Einschränkungen und Bedingungen: Für Vector und Deque gilt, im Gegensatz zu List, dass bei Einfügen und Löschen, die im Container enthaltenen Elemente verschoben werden. Das hat zur Folge, dass ein Iterator danach auf ein anderes oder auf gar kein Element verweist. Der Iterator wird somit ungültig und das Verhalten ist undefiniert [vgl. Stroustrup01, S. 483]. Bei Einsatz von Iteratoren liefert deshalb nur STL-List und infolgedessen auch AthenaMP `list_ts` korrekte und definierte Zustände.

Ein Nachteil der Nutzung des Patterns, besteht darin, dass jeder Operationsaufruf auf den AthenaMP-Containern automatisch das Setzen eines Locks hinter sich herzieht und somit wiederum Zeit kostet. Dies kann im schlimmsten Fall zu einer Sequentialisierung des Programms führen und evtl. den durch Parallelität gewonnenen Geschwindigkeitszuwachs wieder zunichte macht, liefert dafür aber auf jeden Fall korrekte Ergebnisse.

Intel bietet eine kostenpflichtige Alternative zu den drei hier vorgestellten Datenstrukturen in seiner Threading Building Blocks Library an. Die von Intel vorgestellten Klassen namens `concurrent_vector` und `concurrent_queue` sind nicht im Speziellen auf OpenMP spezialisiert, bieten aber eine ähnliche Funktionalität an. Ähnlich den hier vorgestellten Datenstrukturen, `list_ts` ausgenommen, werden ihre Iteratoren auch ungültig, wenn neue Elemente hinzugefügt oder alte Elemente entfernt werden.

4.1.1. LIST_TS

Die in der AthenaMP angebotene `list_ts` ist eine dynamische Datenstruktur, die ihre Daten in einer sequenziellen Reihenfolge speichert. Einfügen und Löschen von Elementen erfolgt für `list_ts` schneller als für `deque_ts` oder `vector_ts`, da diese Operationen nicht mit dem Verschieben der nachfolgenden Elemente verbunden sind. Dafür fehlt `list_ts` der indizierte Zugriff auf Elemente an einer bestimmten Position. AthenaMP `list_ts` arbeitet äquivalent zu der in der STL angebotenen `list`, d. h., `list_ts` bietet alle Methoden und Konstruktoren an, die auch STL-List anbietet.

`List_ts` ist durch Templates realisiert. Der erste Template-Parameter `Type` gibt den Datentyp der zu verwaltenden Objekte an (Listing 4.1.6):

```
01 template<class Type, ... >
02 class list_ts {    ...    };
```

Listing 4.1.6: Klassensignatur von `list_ts`.

Listing 4.1.7 zeigt die Erzeugung einer `list_ts` Datenstruktur mit `int` als Datenelement.

```
01 list_ts<int> ts_list;
```

Listing 4.1.7: Erzeugung einer `list_ts` mit Datentyp int auf dem Stack.

In die Liste können neue Elemente mittels `insert()`, `push_front()`, `push_back()`, `resize()`, `assign()` oder `merge()` eingefügt werden, wobei bestehende Elemente durch `erase()`, `pop_front()`, `pop_back()`, `resize()`, `clear()`, `unique()` oder `remove()` gelöscht werden können.

Auf einzelne Elemente kann über die von `front()` und `back()` zurückgelieferten Referenzen zugegriffen werden, außerdem ist der Zugriff auch über zurückgelieferte Iteratoren mittels `begin()`, `end()`, `rbegin()` oder `rend()` möglich.

Mit `reverse()` kann die Reihenfolge aller in der Liste enthaltenden Elemente umgekehrt werden. Außerdem können alle Elemente durch Verwendung von `sort()` neu sortiert werden.

4.1.2. DEQUE_TS

Deque steht für Double-Ended QUEue. `Deque_ts` ist eine dynamische Datenstruktur, die üblicherweise auf einer Verkettung von Array-Blöcken basiert und deren Elemente in einer sequenziellen Reihenfolge angeordnet sind. Es handelt sich um eine Datenstruktur, bei der die Daten sowohl am Anfang als auch am Ende in konstanter Zeit eingefügt oder entfernt werden können. Das Einfügen oder Entfernen von Elementen an einer beliebigen Position der `deque_ts` ist im Vergleich zu `list_ts` recht zeitaufwändig, da intern die Elemente der `deque_ts` umkopiert werden müssen. AthenaMP `deque_ts` arbeitet äquivalent zu der in der STL angebotenen Deque, d. h., `deque_ts` bietet alle Methoden und Konstruktoren an, die auch STL-Deque anbietet.

`Deque_ts` ist durch Templates realisiert. Der erste Template-Parameter `Type` gibt den Datentyp der zu verwaltenden Objekte an (Listing 4.1.8):

```
01 template<class Type, ... >
02 class deque_ts {    ...    };
```

Listing 4.1.8: Klassensignatur von `deque_ts`.

Listing 4.1.9 zeigt die Erzeugung einer `deque_ts` Datenstruktur mit **`float`** als Datenelement.

```
01 deque_ts<float> ts_deque;
```

*Listing 4.1.9: Erzeugung einer `deque_ts` mit Datentyp **`float`** auf dem Stack.*

Es können neue Elemente in die deque_ts mittels `insert()`, `push_front()`, `push_back()`, `resize()` oder `assign()` eingefügt werden. `erase()`, `pop_front()`, `pop_back()` oder `clear()` entfernen Elemente aus der `deque_ts`.

Auf einzelne Elemente der `deque_ts` kann über Iteratoren mithilfe von `begin()`, `rbegin()`, `end()` und `rend()` zugegriffen werden, außerdem bietet die Deque auch direkten Zugriff auf einzelne Elemente mit dem Operator `[]`, `at()`, `front()` und `back()`an.

4.1.3. VECTOR_TS

Die Klasse `vector_ts` bietet wie ein Array einen schnellen und direkten Zugriff auf einzelne Elemente sowie Einfügen und Löschen an beliebigen Positionen in linearer Zeit. Am Ende des `vector_ts` können neue Elemente jedoch mit konstanter Zeit hinzugefügt oder entfernt werden. Im Vergleich zu `list_ts` ist das Einfügen und Löschen an mittleren Positionen zeitaufwendiger, jedoch hat man im Gegensatz zur `list_ts` einen bequemen indizierten Zugriff auf alle Elemente im `vector_ts`. Im Gegensatz zur `deque_ts` ist `vector_ts` nur für das Einfügen und Entfernen am Ende der Datenstruktur optimiert. Athe-

naMP `vector_ts` arbeitet äquivalent zum STL `Vector`, d. h., `vector_ts` bietet alle Methoden und Konstruktoren an, die auch STL Vector anbietet.

Der Vollständigkeit halber wird nun auch noch auf die Details von `vector_ts` eingegangen, der in der Art seiner Implementierung analog der beiden Vorgänger ist.

`Vector_ts` ist durch Templates realisiert. Der erste Template-Parameter `Type` gibt den Datentyp der zu verwaltenden Objekte an (Listing 4.1.10):

```
01 template<class Type, ... >
02 class vector_ts {    ...    };
```

Listing 4.1.10: Klassensignatur von `vector_ts`.

Listing 4.1.11 zeigt die Erzeugung einer `vector_ts` Datenstruktur mit **`double`** als Datenelement.

```
01 vector_ts<double> ts_vector;
```

*Listing 4.1.11: Erzeugung einer `vector_ts` mit Datentyp **`double`** auf dem Stack.*

Es stehen folgende Operationen zur Verfügung, um Elemente in den `vector_ts` einzufügen: `insert()`, `push_back()`, `resize()` und `assign()`. Elemente werden mit `erase()`, `pop_back()`, `resize()` oder `clear()` entfernt.

Der Zugriff auf einzelne Elemente erfolgt mithilfe von `begin()`, `rbegin()`, `end()`, `rend()` zurückgelieferter Iteratoren oder über die von dem Operator `[]`, `at()`, `front()` und `back()` zurückgelieferten Referenzen.

LAUFZEITMESSUNGEN:

Die Laufzeitmessungen der Datencontainer beschränken sich für alle Datencontainer auf die Methoden `push()` und `pop()`. Dabei werden die Methoden sequenziell und parallel getestet:

Sequenziell:

Jeder threadsichere Datencontainer wird gegen sein Äquivalent aus der STL-Bibliothek getestet (`list_ts` gegen `std::list`, `deque_ts` gegen `std::deque` und `vector_ts` gegen `std::vector`). Dabei wird als Referenzwert die Zeit gemessen, die benötigt wird, um 50millionenmal einen int-Werte mittels `push()` auf den Datencontainer zu legen und anschließend diesen mittels `pop()` wieder komplett zu leeren. Diese Zeit ist für jeden der drei Datencontainer in der Tabelle 4.1.1 unter „STL“ zu sehen. Der zweite Test, in Tabelle 4.1.1 als „STL MIT CRITICAL“ bezeichnet, misst wie viel Zeit benötigt wird, wenn `push()` und `pop()` jeweils durch eine kritische Sektion geschützt sind (mittels `#pragma omp critical`). Für den, als „STL MIT LOCK-ADAPTER“ bezeichneten, dritten Test wurden die durch criticals geschützten kritischen Sektionen durch einen von AthenaMP angebotenen Lock-Adapter geschützt (vergleiche Seite 17). Der letzte sequenzielle Test befasst sich direkt mit den in diesem Kapitel vorgestellten

Datenstrukturen und führt auf jeder von ihnen die entsprechende Anzahl von `push()` und `pop()`-Operationen aus (siehe „THREADSAFE CONTAINER“).

Wie Tabelle 4.1.1 zeigt, schneiden die threadsicheren Datencontainer logischerweise im direkten Vergleich zu den STL-Datencontainern sehr schlecht ab. Jedoch hinkt dieser Vergleich, da die STL-Datencontainer nicht threadsicher sind und somit auch keine korrekt ablaufenden Programme garantieren. Dieser Test diente dazu, insbesondere da er nur sequenziell abläuft, um den durch die Locks verursachten Overhead zu messen. Interessant ist, dass ein mit critical geschützter kritischer Abschnitt signifikant schneller als der verwendete Lock-Adapter ist. Mit critical geschützte Abschnitte sind jedoch nicht möglich, da einige Methoden, die Werte zurückliefern nur mit einem Guard-Objekt geschützt werden konnten (vergleiche Seite 17).

		Vector	List	Deque
STL	`push()`	0,84	4,20	0,53
	`pop()`	0,02	1,55	0,31
	Gesamt:	0,86	5,77	0,84
STL MIT CRITICAL	`push()`	2,51	5,41	2,15
	`pop()`	1,75	4,05	2,02
	Gesamt:	4,27	9,47	4,18
STL MIT LOCK ADAPTER	`push()`	5,52	7,57	5,17
	`pop()`	4,78	6,39	5,03
	Gesamt:	10,31	13,96	10,21
THREADSAFE CONTAINER (TS)	`push()`	5,50	8,88	7,36
	`pop()`	4,75	7,72	6,71
	Gesamt:	10,26	16,60	14,07

Tabelle 4.1.1: Die sequenziellen Laufzeiten für 50 Millionen Datenelemente, aufgeschlüsselt nach push(), pop() und Gesamtlaufzeit, gemessen in Sekunden.

Parallel:

Die parallelen Laufzeittests funktionieren nach dem Producer / Consumer – Prinzip (Erklärung siehe Abbildung 4.2.4 auf Seite 26). Der Producer-Thread fügt 50 Millionen int-Werte in den Datencontainer, während der Consumer-Thread solange läuft, bis er diese 50 Millionen Werte aus dem Datencontainer entfernt hat (siehe Tabelle 4.1.2).

	`vector_ts`	`list_ts`	`deque_ts`
ZEIT:	37,75	63,74	35,31

Tabelle 4.1.2: Die Laufzeiten des Producer / Consumer-Tests, gemessen in Sekunden.

Tabelle 4.1.2 zeigt, dass `list_ts` deutlich langsamer als `vector_ts` und `deque_ts` ist, dies deckt sich auch mit dem im Seriellen gemessenen Laufzeitverhalten der zugrunde liegenden STL-List.

4.2 SHARED QUEUE

KURZBESCHREIBUNG:

Die Shared Queue ist ein threadsicherer, abstrakter Datentyp in Form einer Queue. Sie ermöglicht einen komplett lock-freien Datenaustausch in einer Richtung zwischen zwei Threads.

Hinweis:

Die Shared Queue ist aufgrund des OpenMP Speichermodells ohne Locks nicht portabel. Praktische Tests ergaben, dass sie auf x86-Architekuren dennoch funktioniert. Details siehe „Vor- und Nachteile“.

MOTIVATION/PROBLEM:

Im Idealfall bietet die zugrunde liegende Programmiersprache threadsichere Datencontainer an. Dies ist in unserem Fall leider nicht gegeben. Alle in Kapitel 4.1. vorgestellten Datencontainer machen häufigen Gebrauch von Locks, die zu einer Sequentialisierung des Programms führen können. Als Alternative wird in diesem Kapitel die Shared Queue vorgestellt.

Die Shared Queue:

- ist ein ADT,
- ist threadsicher,
- kommt komplett ohne Locking aus (für genau zwei Threads).

Die Shared Queue stellt eine Spezialisierung einer normalen Queue dar und soll helfen, einer – aufgrund zu vieler Locks – drohenden Sequentialisierung zu entgehen. Sie ordnet sich in die Kategorie der Synchronisationspattern ein (vgl. Kapitel 3.2.3 Seite 12).

Die hier vorgestellte Shared Queue basiert auf einer von Mattson *et al.* [Mattson05, S. 183] vorgestellten Idee.

IMPLEMENTIERUNG:

Das wichtigste Methodenpaar einer Queue sind `push()` und `pop()`. Dabei ist es üblich, dass `push()` neue Daten in die Queue einfügt, während `pop()` diese wieder entfernt.

Die Grundidee hinter der hier vorgestellten lockfreien Shared Queue besteht darin, dass diese beiden Methoden unabhängig voneinander arbeiten können. „Unabhängig“ heißt in diesem Fall, dass `push()` und `pop()` nicht auf denselben Speicher zugreifen. Würden `push()` und `pop()` auf denselben Speicher zugreifen, müsste der Speicherbereich durch Criticals oder Locks geschützt werden. Um dies zu verdeutlichen, wird `push()` als `push_back()` und `pop()` als `pop_front()` bezeichnet.

Queues sind für gewöhnlich als durch Pointer miteinander verkettete Nodes (Knoten) realisiert (siehe Abbildung 4.2.1 auf der nächsten Seite). Die Nodes speichern die in die Queue eingefügten Datenelemen-

te. Ein Node besitzt dabei immer eine Referenz auf seinen Nachfolger oder auf `NULL`, wenn es der letzte Node in der Queue ist. Desweiteren besitzt die Shared Queue noch je einen Zeiger vom Typ Node auf ihren Anfang (`head_`) und auf ihr Ende (`tail_`). Wird eine neue Instanz der Shared Queue erzeugt, zeigt `head_` auf `tail_`, oder anders ausgedrückt der Nachfolger beider Nodes zeigt auf `NULL`. Dies ist auch der Fall, wenn eine Liste Elemente enthält, die komplett mittels `pop_front()` entfernt werden.

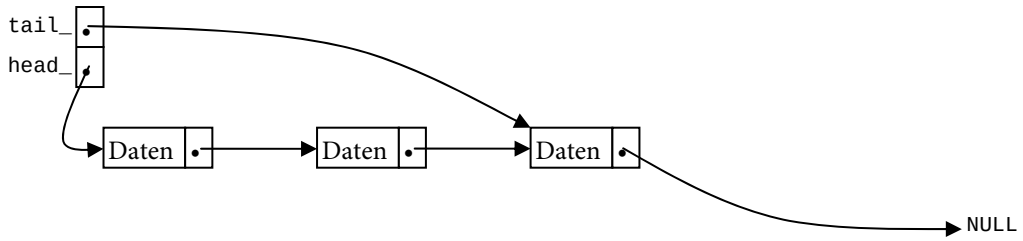


Abbildung 4.2.1: Queue mit drei als Nodes realisierten Datenelementen

Die `push_back()`-Methode arbeitet nur auf dem von `tail_` referenzierten aktuell letzten Node der Queue, sowie auf `tail_` selbst. Wird ein neues Element eingefügt, so wird per `tail_` der letzte Node ermittelt. An diesen wird ein neuer Node angehängt, der nun der neue letzte Node ist. Die Referenz von `tail_` wird dahingehend verändert, dass `tail_` auf diesen neu eingefügten, nun letzten Node zeigt (Abbildung 4.2.2). Gut zu erkennen ist, dass nur der letzte Node und `tail_` von den Änderungen betroffen ist. (Anmerkung: Rot gefärbte Elemente markieren, dass eine Sache entfernt wird. Während grün gefärbte Elemente markieren, dass etwas neu erstellt wurde.)

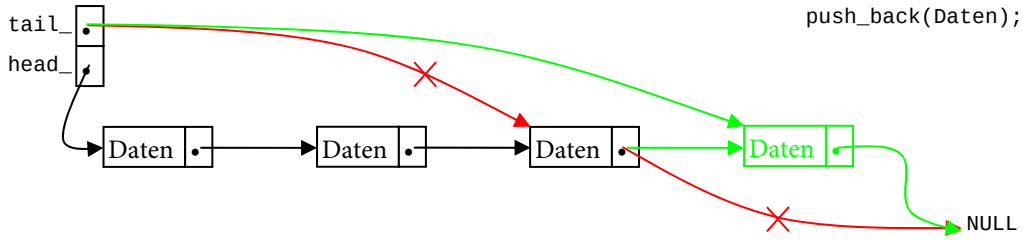


Abbildung 4.2.2: `push_back()` neuer Daten auf die Queue.

Die `pop_front()`-Methode arbeitet nur auf `head_` sowie dem ersten Node der Queue, also komplett unabhängig von `push_back()`. Wird ein Element gelöscht, so wird die neue `head_` Referenz auf den Nachfolge-Node des ersten sich in der Queue befindlichen Node gesetzt, welcher dann anschließend entfernt werden kann (siehe Abbildung 4.2.3).

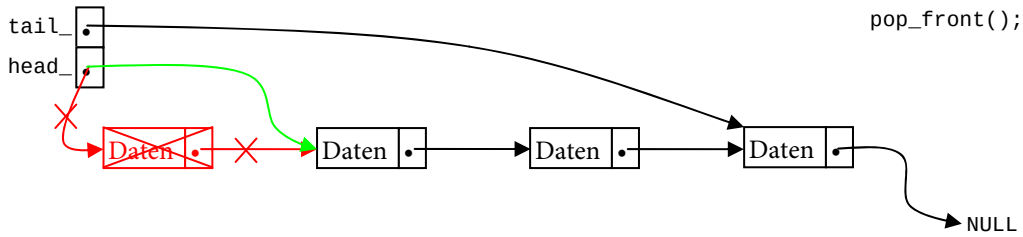


Abbildung 4.2.3: `pop_front()` entfernt den ersten Node aus der Queue. Änderungen finden nur auf dem ersten Node in der Queue und auf `head_` statt.

`push_back()` und `pop_front()` stellen zwei komplett voneinander unabhängige Methoden dar, selbst wenn sich nur ein Element in der Queue befindet, da im Falle eines `pop_front()`-Aufrufes wieder die Start-Situation gilt. Sie haben jeweils ihren eigenen Speicher, auf dem sie arbeiten. Dadurch können sie sich nicht gegenseitig stören. Die Shared Queue kommt ohne Locks aus, wenn sie von genau zwei Threads genutzt wird. Dabei muss darauf geachtet werden, dass ein Thread nur die `push_back()`-Methode nutzt, während dem anderen Thread ausschließlich die Benutzung von `pop_front()` vorbehalten ist (inklusive `front()`, um eine Referenz auf das erste, sich in der Queue befindliche Element zurückgeliefert zu bekommen).

Die Shared Queue besitzt zwei Template-Parameter, durch die sie konfiguriert werden kann:

```
01 template<class Type, bool HasConstSizeRunTime = false>
02 class shared_queue { ... }
```

Listing 4.2.1 zeigt die Template-Parameter der Shared Queue.

Der erste Parameter namens `Type` gibt wie gewöhnlich den Typ der in der Shared Queue gespeicherten Datenelemente an. Der zweite Parameter namens `HasConstSizeRunTime` ist optional, aber auch interessanter. Durch ihn wird das Laufzeitverhalten der Klasse beeinflusst (Policy-Based Class Design nach Alexandrescu [Alexandrescu01]). Muss eine Anwendung, die die Shared Queue nutzt, wissen, wie viele Elemente in ihr enthalten sind, kann dies mit der Methode `size()` abgefragt werden. `HasConstSizeRunTime` beeinflusst nun, ob das Laufzeitverhalten von `size()` im schlimmsten Fall linear ist. Wird `HasConstSizeRunTime` mit `false` angegeben, werden bei einem Aufruf von `size()` alle Elemente in der Shared Queue durchlaufen und aufsummiert, um die Anzahl der Elemente in der Liste zu bestimmen. Dies hat bei N in der Liste enthaltenen Elementen logischerweise eine Laufzeit von O(N).

Gibt man `HasConstSizeRunTime` mit `true` an, wird bei jedem Aufruf von `push_back()` oder `pop_front()` intern eine Variable hoch- bzw. heruntergezählt, die immer die aktuelle Anzahl aller momentan in der Shared Queue enthaltenen Elemente mitprotokolliert. Da diese interne Zählvariable jedoch sowohl in `push_back()` als auch in `pop_front()` vorkommt, ist der Zugriff auf sie – gemäß den OpenMP-Konventionen – durch ein `#pragma omp atomic` geschützt. Dieses `atomic` geht auf die Kosten der Geschwindigkeit, die sich aber amortisieren, wenn in einer Anwendung viele `size()`-Abfragen benötigt werden. Die Shared Queue besitzt in diesem Fall für einen `size()`-Aufruf eine konstante Laufzeit von O(1).

Als Besonderheit wurde bei der Umsetzung der Shared Queue darauf geachtet, dass, sollte sie mit linearer Laufzeit (also `HasConstSizeRunTime = false`) initialisiert sein, die intern von `size()` genutzte Zählvariable vom Compiler wegoptimiert wird. Um dies zu erreichen, wurde eine Technik namens Empty Base Optimization (kurz EBO) angewendet [Meyers01, Seite 34]. EBO beruht auf den Prinzipien privater Vererbung:

```
01 template<class Type, bool HasConstSizeRunTime = false>
02 class shared_queue : private size_wrapper< HasConstSizeRunTime >
```

Listing 4.2.2 zeigt den Aufbau der Shared Queue unter Verwendung von EBO.

Die Shared Queue erbt in diesem Fall von einer Struct namens `size_wrapper`. Die Aufgabe dieser Struct ist es, die interne Zählvariable – für die Anzahl der Elemente – in der Shared Queue zu speichern. `size_wrapper` ist ein spezialisiertes Template, das aufgrund des ihm übergebenen Parameters zur Compile-Zeit entscheidet, welcher Spezialisierungstsyp gewählt ist. Wird `size_wrapper` mit `false` aufgerufen, fungiert es nur als ein Set von Dummy-Methoden, die keinen Code enthalten, also komplett leer sind

(siehe Listing 4.2.3). Weil Member-Methoden im Gegensatz zu Member-Variablen keinen Speicher benötigen, kann so die dadurch nicht vorhandene interne Zählvariable wegoptimiert werden. Wird `size_wrapper` dagegen mit `true` aufgerufen, werden die zuvor leergelassenen Methoden implementiert, so dass die interne Zählvariable bei jedem Aufruf von `push_back()` bzw. `pop_front()` korrekt mitgezählt wird (siehe Listing 4.2.4).

```
01 template
02 <bool HasConstSizeRunTime=false>
03 class size_wrapper {
04 // keine Variable
05 // für die Größe der Queue
06
07
10 public:
11     ...
12
13     // update der Größe
14     // (Dummy)
15     void size_update(int add) {}
16
17
18
19 };
```

Listing 4.2.3 `size_wrapper` bei linearer Laufzeit ohne eine interne Zählvariable.

```
template
<>
class size_wrapper<true> {
private:
    //interne Zählvariable
    int size_;

public:
    ...

    // update der Größe
    // (implementiert)
    void size_update(int add) {
        #pragma omp atomic
        size_+=add;
    }
};
```

Listing 4.2.4 `size_wrapper` bei konstanter Laufzeit mit einer internen Zählvariablen.

Dies hat den Vorteil, dass so erzeugte Instanzen der Shared Queue weniger Speicher benötigen. Auf den von mir für Testzwecke zur Verfügung stehenden Intel-Architekturen hatte die Shared Queue bei konstanter Laufzeit eine durch `sizeof()` ermittelte Größe von 12, die Shared Queue besaß dagegen bei linearer Laufzeit eine Größe von 8, da die interne Zählvariable durch Anwendung von EBO wegoptimiert wurde.

Wird nun in der `push_back()`-Methode `size_wrapper :: size_wrapper< HasConstSizeRunTime >::size_update( 1 )` aufgrund von `HasConstSizeRunTime` entschieden, ob die `size()`-Methode in linearer oder konstanter Laufzeit durchlaufen soll und ob infolgedessen eine interne Zählvariable (Listing 4.2.4 Zeile 06) korrekt durch `#pragma omp atomic` geschützt werden muss. Analog dazu gilt das Gleiche für die `pop_front()`-Methode, allerdings ändert sich der `size_update()` zu `size_update( -1 )`.

ANWENDUNGSBEISPIEL:

Die hier vorgestellte Shared Queue eignet sich hervorragend, um das Producer / Consumer-Problem leicht zu lösen. Das Producer / Consumer-Problem kommt ursprünglich aus der Betriebssystemtheorie [Tanenbaum03, S. 125], wird aber auch gerne im parallelen Programmieren aufgegriffen. Auf unsere Zielsprache bezogen, existieren genau zwei Threads, die über eine gemeinsam genutzte Datenstruktur kommunizieren und Daten austauschen müssen. Dabei schreibt der Produzent (Producer) Daten in die gemeinsam genutzte Datenstruktur, während der Verbraucher (Consumer) diese Daten wieder ausliest. Klassisch wird dieses Problem meist unter Zuhilfenahme von Semaphoren gelöst. Ich will hier jedoch eine Lösung mithilfe der Shared Queue skizzieren (siehe Abbildung 4.2.4).

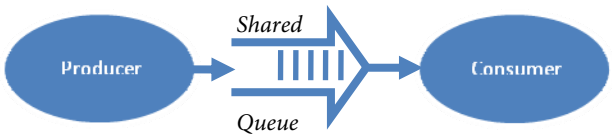


Abbildung 4.2.4 zeigt die schematische Darstellung des Producers und Consumers, die durch eine Shared Queue miteinander verbunden sind. Die Shared Queue wird durch einen Pfeil zwischen Producer und Consumer repräsentiert. Die Striche in diesem Pfeil spiegeln zwischengespeicherte Tasks wider.

Zunächst wird eine neue Shared Queue, die den selbstdefinierten Datentyp ‚`task`' aufnehmen kann, erzeugt:

```
01  shared_queue<task, true> s_queue_;
```

Danach wird noch festgelegt, mit wie vielen Threads OpenMP arbeiten soll, und die parallele Region wird eröffnet:

```
01  omp_set_num_threads(2); //nutze genau 2 Threads
02  #pragma omp parallel
03  {
04  //diese öffnende Klammer muss natürlich auch wieder geschlossen werden.
```

Jetzt kommt der relevante und eigentliche Teil des Problems:

```
01  // der Producer
02  // stellt nur Daten in die
03  // s_queue. (wird von
04  // Thread 0 ausgeführt)
05  for (int i = 0; i < N; ++i)
06  {
07      ...
08      // hier irgendwas arbeiten
09      // den erzeugten Task jetzt
10      // in die shared_queue
11      // einfügen
12      s_queue_.push_back(task);
13      ...
14      // hier weitere
15      // wichtige Dinge
16  }
17  s_queue_.push_back(POISON_TASK);
18
19
20
21
```

Listing 4.2.5: Code für den Producer.

```
// der Consumer
// nimmt nur Daten aus der s_queue.
// (wird von Thread 1 ausgeführt)
do
{
    // wenn keine Arbeit vorhanden,
    // skip
    if (s_queue_.empty())
        continue();

    // Arbeit vorhanden!
    task = s_queue_.front();
    s_queue_.pop_front();

    if (task != POISON_TASK)
    {
        ...
        // bearbeite den Task
        ...
     }
} while (task != POISON_TASK);
```

Listing 4.2.6: Code für den Consumer.

Man erkennt, dass dieses Beispiel komplett ohne Locks auskommt. Producer (Listing 4.2.5) und Consumer (Listing 4.2.6) können simultan auf der Datenstruktur arbeiten, ohne sich gegenseitig zu stören, oder– durch ein evtl. ungünstiges Scheduling – eine inkonsistente Datenstruktur zu hinterlassen.

Im vorliegenden Fall wurde der Consumer in Listing 4.2.6 mit Busy-Waiting realisiert, d. h., sollte die Shared Queue leer sein, wartet der Thread solange, bis wieder Elemente in die Liste des Producers eingefügt wurden. `empty()` liefert **`true`** zurück (Zeile 08), wenn der `head_`-Node auf `NULL` als nächstes Element verweist, ansonsten wird **`false`** zurückgeliefert. Der Consumer führt seine Arbeit solange aus, bis er vom Producer eine sogenannte Poison-Pill erhält. Die Poison-Pill ist ein spezieller Task, der angibt, dass alle Arbeit getan ist. Im obigen Beispiel heißt die Poison-Pill `POISON_TASK`. Der `POISON_TASK` wird vom Producer der Einfachheit halber in die gemeinsam genutzte Shared Queue `s_queue` mittels `push_back()` eingefügt, nachdem er N Elemente erzeugt hat.

Holt sich der Consumer nun seinen `task` ab, wird zuerst überprüft, ob dieser Task die Poison-Pill ist. Ist dies nicht der Fall, führt er auf normale Weise seine Arbeit mit diesem Task oder auf ihm aus, ansonsten beendet er die `while()`-Schleife.

Man sieht, dass sich das Producer / Consumer-Problem durch die hier vorgestellte Shared Queue mit relativ wenigen Zeilen Code komplett ohne Locks realisieren lässt. Dabei ist – wie zuvor schon angedeutet – wichtig, dass die Shared Queue nur mit genau zwei Threads arbeitet, wobei ein Thread immer Daten in die Queue hereinstellt und der andere Thread immer Daten aus eben dieser herausnimmt.

Ein weiteres konkretes Anwendungsbeispiel der Shared Queue findet sich in Kapitel 4.4 im dort vorgestellten Pipeline Pattern.

LAUFZEITMESSUNGEN:

Um die Geschwindigkeit der Shared Queue zu testen, wurde wieder das Producer / Consumer-Beispiel herangezogen. Der Producer-Thread fügt 50 Millionen int-Werte in den Datencontainer ein, während der Consumer-Thread solange läuft, bis er diese 50 Millionen Werte aus dem Datencontainer entfernt hat (siehe Tabelle 4.1.2). Dabei wird die Shared Queue je einmal mit **`HasConstSizeRunTime`** = **`true`** und einmal mit **`HasConstSizeRunTime`** = **`false`** initialisiert.

	shared_queue<false>	shared_queue<true>
ZEIT:	18,56	21,13

Tabelle 4.2.1: Die Laufzeiten des Producer / Consumer-Tests der Shared Queue, gemessen in Sekunden.

Die mit `false` initialisierte Shared Queue ist schneller als die mit `true` initialisierte. Dies liegt aber auch hauptsächlich daran, dass der Producer / Consumer-Test keinen Nutzen von der `size()`-Methode der Shared Queue macht, die sich bei `false` mit linearer Laufzeit bemerkbar macht.

VOR- UND NACHTEILE/NUTZEN:

Die Vorteile der Shared Queue liegen klar auf der Hand: Ein lockfreier Datenaustausch zwischen zwei Threads wird ermöglicht, allerdings mit der zuvor schon erwähnten Einschränkung, dass die Aufgabe eines jeden Threads genau definiert sein muss (also entweder nur `push_back()` oder nur `pop_front()`) und im Laufe der Anwendung nicht wechseln darf.

Die Shared Queue verträgt sich jedoch in der Theorie nicht mit dem verwendeten Speichermodell von OpenMP [Details siehe OpenMP Application Program Interface 2.5, Seite 10]. Das Speichermodell garantiert nicht, dass sich der momentane Inhalt („Temporary View“) einer Variablen eines Threads mit dem wirklichen Inhalt einer Variablen deckt. Ändert ein Thread den Inhalt einer gemeinsam genutzten Variable (Shared Variables) kann es passieren, dass die anderen Threads noch auf einem gecachten und somit veralteten Inhalt der Variable arbeiten.

Dies kann auch bei der Shared Queue passieren. Die Pointer `head_` (zugehörig zu `push_back()`) und `tail_` (zugehörig zu `pop_front()` und `front()` sind gemeinsam genutzte Variablen. `head_` und `tail_` zeigen bei einer leeren Shared Queue auf denselben Speicher. Man stelle sich zwei Threads und eine leere Shared Queue vor. Sind keine Elemente in der Queue enthalten, zeigt `tail_` auf `null`. Thread A fügt Daten in die leere Queue ein. Thread B versucht diese Daten mittels `front()` auszulesen. Der `tail_`-Pointer zeigt nun nicht mehr auf `null`, sondern auf das eben von Thread A eingefügte Datenelement. Es besteht nun die Gefahr, dass Thread B noch mit einem veralteten Inhalt `tail_`‘s arbeitet und somit nie realisiert, dass neue zu bearbeitende Elemente in der Shared Queue eingefügt wurden.

Die Flush Operation synchronisiert auf Kosten der Laufzeit den Speicher mit dem momentanen Wert einer Variablen, jedoch gilt: „*When a thread executes a flush, no later memory operation by that thread for a variable involved in that flush is allowed to start until the flush completes.*“ [OpenMP05, S. 12], sonst kann es passieren, dass der Variablen-Wert für einen lesenden Thread undefiniert ist *(„[…] the value seen by any reading thread is unspecified.*“ [OpenMP05, S. 11]). Dies kann nach dem Speichermodell OpenMPs nur durch den Aufruf einer expliziten Barriere korrigiert werden.

Auf x86-Prozessoren funktioniert dieses dennoch durch einfachen Aufruf von flush. Deshalb zieht jeder `push_back()`- und `pop_front()`-Aufruf ein explizites `flush` nach sich.

Jeder `push()`-Aufruf zieht implizit einen `new`-Aufruf nach sich. Dieser reserviert Speicher für den Node, der neu in die Shared Queue eingefügt werden soll. Messungen haben ergeben, dass dieser Aufruf einen großen Anteil an den Laufzeitkosten der Shared Queue besitzt. Diese Kosten könnten verringert werden, wenn der Shared Queue ein Speichermanager spendiert würde, ähnlich des `reserve()`-Aufrufs bei den STL-Datencontainern. Weiß man schon ungefähr die Anzahl der Elemente, die maximal in der Shared Queue enthalten sind, könnte `reserve()` ausreichend Speicher reservieren, so dass dieser bei jedem `push()`-Aufruf nur noch zugewiesen statt allokiert werden müsste und so die Geschwindigkeit nochmals beschleunigt werden würde.

4.3 OBSERVER

KURZBESCHREIBUNG:

Das Observer-Pattern ermöglicht die automatische Weitergabe von Änderungen eines Objekts an abhängige Objekte.

MOTIVATION/PROBLEM:

Das Observer-Pattern ist ein Design Pattern im klassischen Sinne nach Gamma *et al.* [Gamma95, S. 287] und ordnet sich in die Kategorie der objektorientierten Pattern ein (vgl. Kapitel 3.2.2 Seite 12). Es beschreibt die Idee, bestimmte Objekte auf Veränderungen ihres Zustands zu überwachen. Beim Observer-Pattern wird zwischen einem zu überwachenden Subject und den Observern (Beobachtern) unterschieden. Ein Subject kann viele Observer kennen. Es besteht also eine 1-zu-N Relation zwischen Subject und Observern.

Das Subject informiert die ihm bekannt gemachten Observer über evtl. Änderungen. Um die Objekte nicht eng aneinander zu koppeln, bietet das Pattern die Möglichkeit, an einem Subject gemachte Änderungen an alle registrierten Observer automatisch weiterzumelden. Dabei braucht das geänderte Subject die Observer nicht zu kennen. Es reicht aus, wenn Objekte vom Typ Observer ein gemeinsames Interface besitzen. Auf diese Weise werden die Veränderungen der Daten und die daraus resultierenden Aktionen entkoppelt (Loose Coupling). Das Observer-Pattern ist eines der in Java am häufigsten verwendeten Design Pattern (dort ist es unter dem Namen „Listener“ bekannt).

Ein Beispiel aus dem richtigen Leben für das Observer-Pattern ist das Zeitungsabonnement (siehe Abbildung 4.3.1). Jedoch kein Zeitungsabonnement im klassischen Sinn, dass regelmäßig Zeitungen an die Abonnenten liefert, sondern nur immer dann eine Zeitung ausliefert, wenn sich wirklich etwas Neues ereignet hat (ähnlich einem Newsletter).

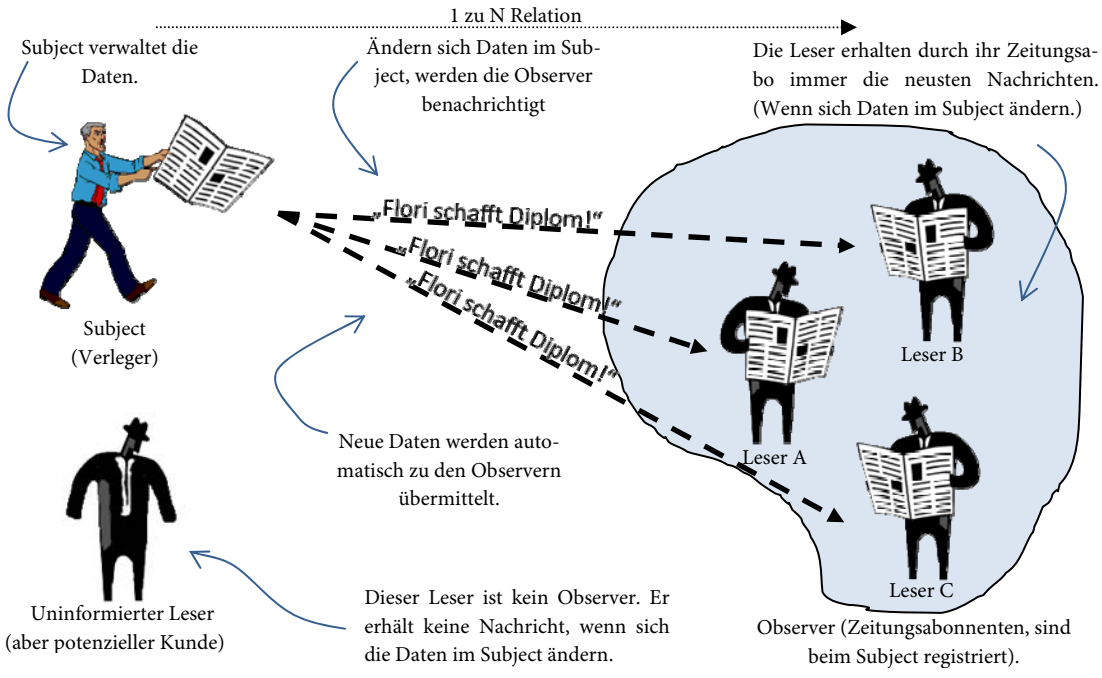


Abbildung 4.3.1: Beispiel des Observer Patterns anhand eines Zeitungsabonnements.

Ein Zeitungsverleger, wir nennen ihn „Subject“, gibt eine Zeitung heraus. Wird man Abonnent dieser Zeitung heißt man ab sofort „Observer“. Der Verleger gibt eine neue Zeitung heraus, wenn sich etwas an dem Zeitungsobjekt ändert. Der registrierte Leser erfährt dies nun automatisch, oder um in der Analogie der Zeitungsbranche zu bleiben: Die Zeitung wird direkt vor die Haustür des Abonnenten geliefert.

IMPLEMENTIERUNG:

Beim Observer-Pattern wird in der Literatur zwischen Push- und Pull-Variante unterschieden [Gamma95, S. 294].

- *Push-Variante*: Das Subject schickt den Observern detaillierte Informationen über die Änderungen, ob Interesse besteht oder nicht.
- *Pull-Variante*: Das Subject schickt nur minimale Informationen, woraufhin die Observer sich benötigte Daten selbst vom Subject abholen müssen.

Im Rahmen dieser Arbeit wurde das Observer-Pattern mit der Pull-Variante implementiert. Sollte jedoch einmal die Push-Variante benötigt werden, so ist es ein Leichtes das bestehende Pattern aus der AtheNaMP-Bibliothek zu variieren und an die Anforderungen anzupassen.

Wie in Abbildung 4.3.2 zu sehen ist, gestaltet sich das Pattern recht übersichtlich. Das Klassendiagramm besteht im Grunde genommen aus zwei Interfaces, die im Gegensatz zu Gamma *et al.* [Gamma95, S. 289] jedoch direkt als Klassen angeboten werden.

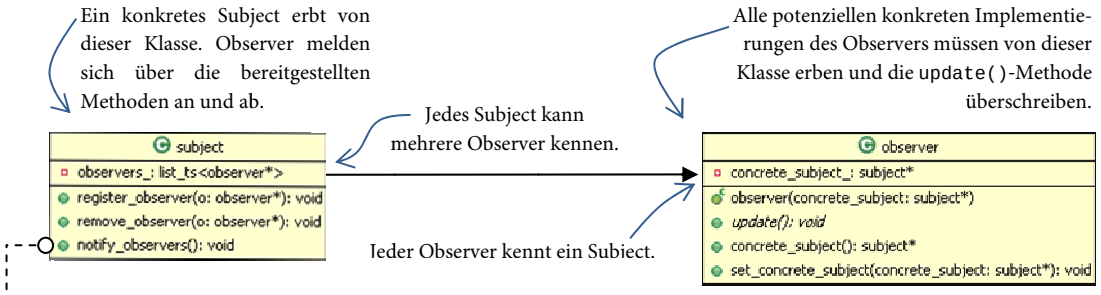


Abbildung 4.3.2: Klassendiagramm des Observer-Patterns.

```
for (list_ts< observer*>::iterator iterator = observers_.begin();
     observers_.end() != iterator;
     ++iterator
    )
{
    observer* obs = *iterator;
    obs->update();
}
```

Listing 4.3.1: Die Implementierung der Methode `notify_observers()` wird immer aufgerufen, wenn sich etwas am Zustand eines konkreten Subjects ändert.

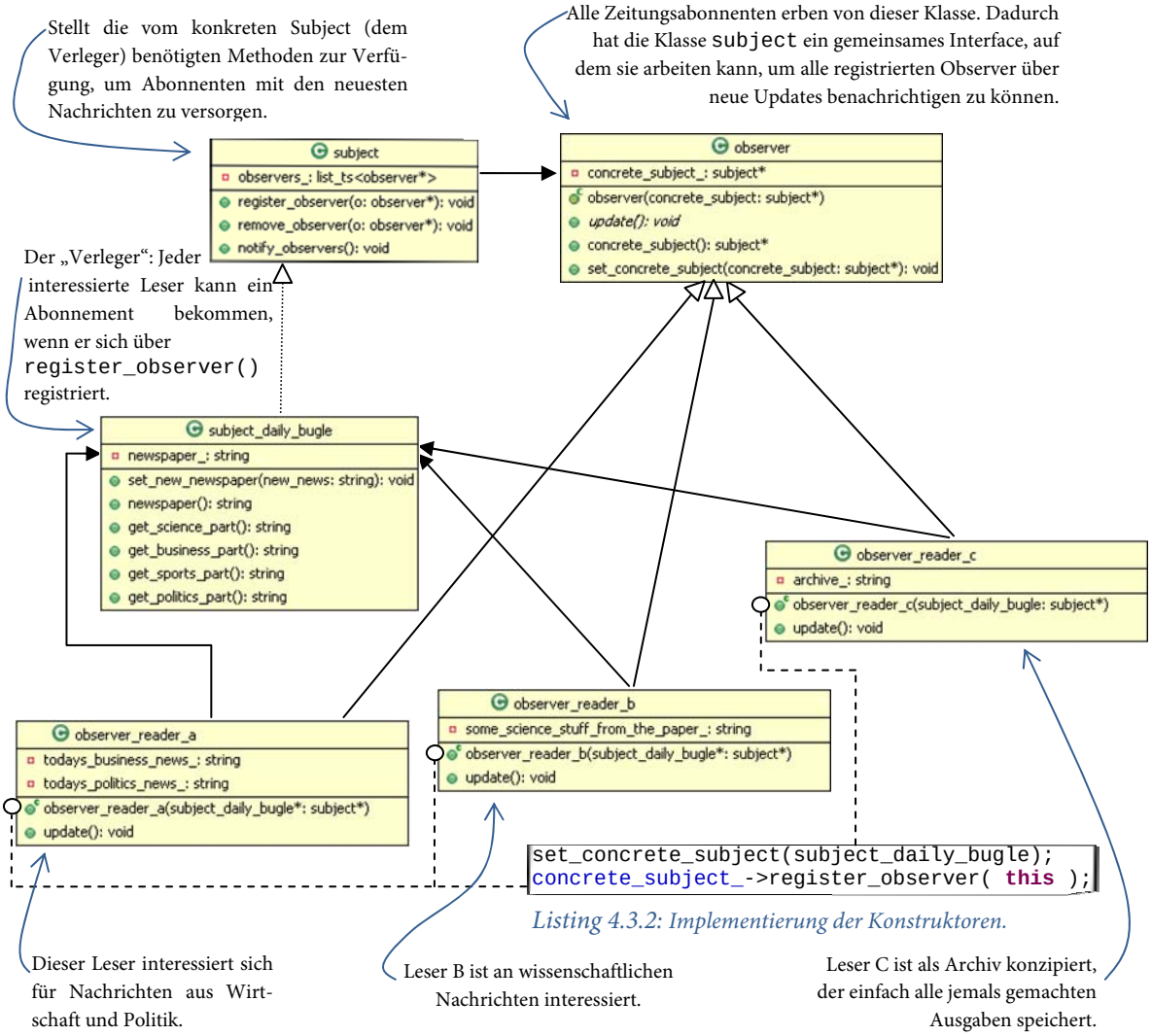
Die Klasse `observer` ist – wie schon erwähnt – eigentlich ein Interface, besitzt aber zusätzlich noch einen Pointer auf die Klasse `subject`, so dass mögliche konkrete Implementierungen der Klasse auf die Daten eines konkreten Subjects (z. B. die Zeitung eines Zeitungsverlegers) zugreifen können. (*Anmerkung*: Mit der angebotenen Lösung besitzt jeder Observer einfachheitshalber nur ein Subject. Dies kann jedoch bei Bedarf angepasst werden.) Desweiteren wird noch die virtuelle Methode `update()` angeboten. Listing 4.3.1 zeigt, dass `update()` durch die Methode `notify_observers()` in der Klasse `subject`

aufgerufen wird. Um alle interessierten Observer zu speichern, bedient sich die Klasse `subject` der in Kapitel 4.1.1 vorgestellten threadsicheren Datenstruktur `list_ts`. Dies bietet– wie in Kapitel 4 erwähnt – auch bei den in Listing 4.3.1 verwendeten Iteratoren noch ein korrektes threadsicheres Laufzeitverhalten, selbst wenn sich während eines „Update"-Vorgangs bereits registrierte Observer abmelden (`remove_observer()`) oder neue Observer an das Subject anmelden (`register_observer()`) sollten.

Es bleibt den einzelnen Klassen, die von der Klasse `observer` erben, überlassen, wie sie ihre `update()`-Methode implementieren und welche Informationen sie sich von dem Subject holen möchten (ein Beispiel hierfür ist unter „Anwendungsbeispiel" gegeben.

ANWENDUNGSBEISPIEL:

Exemplarisch wird auf das in der Motivation des Observer-Patterns vorgestellte Beispiel eines Zeitungsabonnements zurückgegriffen (siehe Abbildung 4.3.1), das unter Zuhilfenahme eben dieses Patterns implementiert werden soll (siehe Abbildung 4.3.3).



Listing 4.3.2: Implementierung der Konstruktoren.

Abbildung 4.3.3: Klassendiagramm des Observer Patterns anhand eines Zeitungsabonnements.

Der Verleger aus Abbildung 4.3.1 ist ein konkretes Subject und wird durch die Klasse `subject_daily_bugle` repräsentiert. Die Klasse `subject_daily_bugle` erbt von der Klasse `subject` aus der AthenaMP-Bibliothek und verfügt über die von diesen angebotenen Methoden gemäß den allgemein gültigen Vererbungsregeln.

Wollen nun die drei konkreten Observer namens `observer_reader_[a|b|c]` (siehe Abbildung 4.3.3) immer bei neuen Nachrichten sofort vom Verleger benachrichtigt werden,, so müssen sie sich jeweils durch Aufruf der Methode `register_observer()` einmalig bei diesem anmelden (siehe Listing 4.3.3.).

```
01  subject_daily_bugle db;  // Verleger-Objekt.
02
03  // Verknüpfung der Objekte „per Hand“
04  observer_reader_a r1;    // Lesers A.
05  db.register_observer( r1 );
06  r1.set_concrete_subject( db );
07
08  // Alternativ automatische Verknüpfung.
09  observer_reader_b r2( &db );   // Lesers B
10  observer_reader_c r3( &db );   // Lesers C
```

Leser A (`r1`) bei Verleger registrieren.

Verleger bekommt neuen Leser (Zeitungsabonnent).

Aufruf von `register_observer()` und `set_concrete_subject()` im jeweiligen Konstruktor (siehe Listing 4.3.2).

Listing 4.3.3: Zeigt den Registrierungsvorgang der drei konkreten Observer am Subject.

Ändert sich nun intern der Zustand unseres Verleger - Objektes (in Listing 4.3.3 mit `db` bezeichnet), d. h., in diesem Beispiel kommen neue wichtige Nachrichten aus der Welt herein, die sofort allen registrierten Lesern publik gemacht werden müssen, so ruft das Verleger-Objekt die geerbte Methode `notify_observers()` auf (siehe Listing 4.3.4).

```
01  void set_new_newspaper(string new_news) {
02      newspaper_ = new_news;
03
04      // Here the magic happens! (:
05      // Alle registrierten Observer benachrichtigen
06      notify_observers(); // Code siehe Listing 4.3.1
07  }
```

Listing 4.3.4: Die Implementierung der Setters `set_new_newspaper(…)` der Klasse `subject_daily_bugle`.

Jeder der drei Abonnenten (`observer_reader_[a|b|c]`) muss nur die in der Klasse `observer` als virtuell markierte Methode `update()` implementieren, da diese – wie in Listing 4.3.1 zu sehen – von `notify_observers()` aufgerufen wird. Eine mögliche `update()`-Methode, exemplarisch für alle drei, am Beispiel der Klasse (`observer_reader_a`) aufgezeigt, könnte folgendermaßen aussehen:

```
01  //Leser A, ein waschechter Observersprössling
02  void update() {
03      todays_business_news_ =
04          dynamic_cast<subject_daily_bugle*>
05              (concrete_subject_)->get_business_part();
06      todays_politics_news_ =
07          dynamic_cast<subject_daily_bugle*>
08              (concrete_subject_)->get_politics_part();
09      // verarbeite die neuen Daten
10  }
```

Liefern jeweils nur den gewünschten Teil einer Zeitung zurück.

Listing 4.3.5: Die Implementierung der `update()`-Methode für die Klasse `observer_reader_a`.

Leser A ist nach Abbildung 4.3.3 sehr an politischen und wirtschaftlichen Geschehen interessiert. Da das hier vorgestellte Observer-Pattern nach der Pull-Methode arbeitet, pickt sich Leser A über die entsprechenden Getter der Klasse `subject_daily_bugle` die Nachrichtensparten heraus, die ihn interessieren. Dies macht er auch nur dann, wenn er durch `notify_observers()` benachrichtigt wurde, dass neue Nachrichten vorhanden sind.

PARALLELE IMPLEMENTIERUNG:

Es wird ein Beispiel konstruiert, das zeigen soll, dass das Observer-Pattern insbesondere auch im parallelen Bereich seine Berechtigung hat:

```
01 //Das konkrete Subject erzeugen -> gibt Zeitungen heraus.
02 subject_daily_bugle db;
03 #pragma omp parallel
04 {
05     int thread_id = omp_get_thread_num();
06     //THREAD 0 -> Der Zeitungsverleger
07     if (thread_id == 0)
08     {
09         // warten bis mindestens ein Abonnent vorhanden
10         do {
11             wait(FOR_SOME_READERS);  // einfach nur ein wenig warten
12         } while (db.count_observers() <= 0);
13         // die Zeitung hat mindestens einen Leser
14         int newspaper_issue = 0;
15         do {
16             wait(FOR_THE_NEW_NEWS);
17             ++newspaper_number;
18
19             db.set_new_newspaper("Newspaper #"+ newspaper_issue);
20
21         } while (db.count_observers() > 0);
22         // Zeitung hat keine Leser mehr, also wird sie eingestellt.
23     } //Ende von Thread 0, dem Verleger
```

Sendet eine Benachrichtigung an alle Abonnenten, dass eine neue Ausgabe / Nachricht existiert!

Listing 4.3.6.: Erzeugung des konkreten Subjects in Form eines Zeitungsverlegers.

```
01 //THREAD 1 bis N, steht exemplarisch für einen Zeitungsabonnenten.
02 if (thread_id==1) //exemplarisch für Thread 1
03 {
04     observer_reader_a r1(&db); //Leser erzeugen.
05     do {
06         wait(DO_WHAT_YOU_WANT);
07     } while(r1->paper_count()<=50);
08 } //Ende von Thread 1/Leser A
```

Steht für „Dinge“, die der Leser XYZ gerne macht. In diesem Fall warten.

Listing 4.3.7 zeigt eine mögliche Implementierung eines Zeitungsabonnenten.

Dieses ‚parallelere‘, jedoch recht stark konstruiert wirkende Beispiel, besteht aus einem konkreten Subject, hier wieder in Form eines Zeitungsverlegers (siehe Listing 4.3.6) und einem konkreten Observer, der exemplarisch für alle konkreten Observer in diesem Beispiel steht (siehe Listing 4.3.7). Der einzige Nutzen des Beispiels liegt darin, eine mögliche Idee aufzuzeigen, wie das Pattern parallel zu nutzen ist. Probleme, die auftreten können und deren mögliche Lösungen werden unter Vor- und Nachteile erörtert.

In Listing 4.3.6 wird der Zeitungsverleger namens `db` erzeugt (Zeile 02). Es wurde willkürlich festgelegt, dass der Verleger im Thread 0 arbeiten soll (Zeile 07). Nachdem sich mindestens ein Abonnent beim Verleger angemeldet hat (wird in Zeile 12 getestet), nimmt er seine Arbeit auf und schickt immer wenn neue Nachrichten vorhanden sind (Zeile 16) eine neue Zeitung an alle registrierten Lesern (Zeile 20). In der

Methode `set_new_newspaper()` wird die vom `subject` geerbte Methode `notify_observers()` aufgerufen (siehe Listing 4.3.1).

Listing 4.3.7 zeigt einen konkreten Observer. Dieser konkrete Observer ist als Zeitungsabonnent realisiert. Er steht exemplarisch für alle möglichen Zeitungsabonnenten. In dem gezeigten Beispiel macht er gar nichts, außer warten, bis er 50 Zeitungsausgaben erhalten hat. D. h., er bleibt solange in der **`while`**-Schleife hängen, bis seine `update()`-Methode 50mal durch `notify_observers()` aufgerufen wurde. Der Wert 50 wurde dabei willkürlich gewählt.

Im Vergleich zur seriellen Version des Observer-Patterns [vgl. Gamma95, S. 289] fällt auf, dass es nahezu keinen Implementierungsunterschied gibt. Im Gegensatz zu der STL-List des Originals wird hier auf die in Kapitel 4.1 vorgestellte `list_ts` zurückgegriffen, um an Zustandsänderungen interessierte Observer zu speichern.

Das nicht-deterministische Laufzeitverhalten, sobald Threads ins Spiel kommen, stellt ein großes Problem bei der Parallelisierung dieses Patterns dar. Sequenziell ausgeführt, läuft es deterministisch ab. Im Parallelen muss nun mit den von OpenMP gebotenen Mitteln versucht werden, diesen vorerst nicht vorhandenen Determinismus wieder künstlich herzustellen. Im Folgenden werden, einschließlich der bereits vorgestellten Lösung, drei leicht verschiedene Implementierungsstrategien mit ihren spezifischen Vor- und Nachteilen vorgestellt.

```
01  void set_new_value(string new_value) {
02      value_ = new_value;
03      notify_observers(); //siehe Listing 4.3.1 Seite 30
04  }
```

Listing 4.3.8: Die Methode `set_new_value()` eines konkreten Subjects.

Listing 4.3.8 zeigt die bereits in Listing 4.3.4 vorgestellte `set_new_newspaper(…)`-Methode in allgemeinerer Form, die sich nicht auf Zeitungen beschränkt. Die Methode ist Teil eines konkreten Subjects. Jedoch ist der gezeigte Code nicht threadsicher. Rufen mehrere Threads gleichzeitig die Methode `set_new_value()` auf (und somit implizit auch `notify_observers()`, kann es passieren, dass die Liste `list_ts`, die alle Observer enthält, geändert wird. Die Datenstruktur `list_ts` garantiert ein konsistentes Verhalten, wenn während des Iterationsvorgangs (Listing 4.3.1 Zeile 02) ein Observer aus der Liste gelöscht wird (z. B. durch einen Aufruf der Methode `remove_observer()`). Jedoch kann es bei dem Aufruf der `update()`-Methode (Listing 4.3.1 Zeile 04) des gerade entfernten Observers zu einer Exception kommen, wenn zuvor dessen Destruktor aufgerufen wurde. Diese Exception könnte zu einem Programmabsturz und somit zu dem wenig beliebten „Core Dump“ führen. Da Exceptions in OpenMP momentan noch stiefmütterlich unterstützt werden [Süß06a, Süß06b], sollte man sehr genau abwägen, auf welches dünne Eis man sich begibt. Wird bei einer Problemstellung nie die Funktionalität benötigt, einmal registrierte Observer zu entfernen, stellt diese Art der Implementierung eine gute Wahl dar.

Ein Anwender des hier vorgestellten Patterns kann durch die Art der Instanziierung mittels eines Template-Parameters bestimmen, welche Update-Strategie er im Falle von `notify_observers()` benutzen möchte. Der in Listing 4.3.9 gezeigte Template-Parameter `Strategy` muss dabei durch ‚`none`‘, ‚`the_big_lock`‘ oder ‚`mixed`‘ ersetzt werden. Diese Update-Strategien bestimmen das Laufzeitverhalten der Klasse (Policy-Based Class Design nach Alexandrescu [Alexandrescu01]). Sie werden auf den folgenden Seiten im Detail mit ihren „Vor- und Nachteilen“ vorgestellt und abschließend noch einmal in Tabelle 4.3.1 auf Seite 38 übersichtlich dargestellt.

```
01 // eine Möglichkeit, ein neues konkretes Subject zu erzeugen
02 class concrete_subject : public subject< Strategy >
03 { ... }
```

Listing 4.3.9: Eine Möglichkeit, ein konkretes Subject zu erzeugen.

PARALLELE IMPLEMENTIERUNG – NONE STRATEGY:

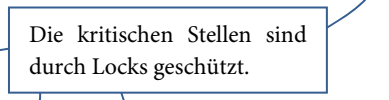
Glücklicherweise macht die Methode `register_observer()` keine Probleme. Das Hinzufügen eines neuen Observers aufgrund der Verwendung von `list_ts` und den damit vorhandenen Eigenschaften der STL-List ist in jedem Fall threadsafe. Es besteht lediglich die vernachlässigbare Gefahr, dass, sollte momentan schon ein `notify_observers()`-Vorgang aktiv sein, erst beim nächsten Durchlauf der gerade hinzugefügte neue Observer mit berücksichtigt wird. Würde man eine andere Datenstruktur als die verwendete nehmen, müsste auch der Aufruf von `notify_observers()` in Kombination mit `register_observer()` geschützt werden. In der Literatur geschieht dies üblicherweise durch Anlegen einer Kopie der Datenstruktur bevor in der Methode `notify_observers()` über alle registrierten Observer iteriert wird [Lea99, Kap. 3.5.2.]. Dies schützt leider nicht vor den eben erwähnten Dateninkonsistenzen in Kombination mit `remove_observer()`.

Der Code ist an einer anderen Stelle ebenfalls noch nicht völlig korrekt bzw. nicht deterministisch. Angenommen zwei Threads, Thread A und B, rufen die Methode `set_new_value()` gleichzeitig auf, wird der von Thread A gesetzte neue Wert nun implizit per `notify_observers()` an alle registrierten Observer propagiert. Dasselbe geschieht bei Thread B. Wäre das Verhalten deterministisch, hätten nach Abschluss der Operationen alle Observer den gleichen Wert. In diesem Fall aber kann es passieren, dass während des Update-Vorgangs eine Art Race-Condition auftritt. Observer A erhält erst den Wert von Thread A, danach den Wert von B, während Observer B erst den Wert von Thread B und danach den Wert von Thread A annimmt. Dateninkonsistenzen sind die Folge. Es ist nicht definiert, welcher Observer welchen Wert hat, d. h., jeder Observer kann einen anderen Wert besitzen. Eine mögliche Lösung sei hier aufgeführt, die jedoch nur theoretisch angedacht werden soll: Jeder Nachricht wird ein Timestamp oder eine Zählvariable mitgegeben, so dass der Observer zwischen alten und neuen Nachrichten unterscheiden kann. Dies setzt natürlich voraus, dass immer die letzte Nachricht die aktuelle und somit die zu verwendende Nachricht darstellt. Dabei wird jedoch eine korrekte Reihenfolge aller Nachrichten außer acht gelassen, außerdem wird stillschweigend angenommen, dass Nachrichten verlorengehen könnten.

PARALLELE IMPLEMENTIERUNG – THE BIG LOCK STRATEGY:

Die einfachste Lösung wäre nun, die kritischen Stellen in `notify_observers()` und `remove_observer()` durch einen von der AthenaMP-Bibliothek angebotenen Lock-Adapter zu schützen (siehe Listing 4.3.10 Zeile 01).

```
01  omp_lock_ad lock_notify_; //der Lock-Adapter
02  ...
03  void notify_observers() {
04      lock_notify_.set();
05      list_ts<observer*>::iterator it;
06      for( it = observers_.begin(); observers_.end() != it; ++it ) {
07          observer* obs = *it;
08          obs->update();
09      }
10      lock_notify_.unset();
11  }
12
13  void remove_observer( observer* o ) {
14      lock_notify_.set ();
15      observers_.remove(o);
16      lock_notify_.unset ();
17  }
```



Listing 4.3.10: Erste alternative Implementierung des Listing 4.3.1.

Dadurch könnten die zuvor aufgeführten Dateninkonsistenzen eingeschränkt werden. Sogar die Race-Condition wäre behoben und es wäre immer gewährleistet, dass die richtigen Werte in der Reihenfolge, in der sie gesetzt werden, bei allen gleich ankommen. Jedoch führt diese Lösung zu einer Sequentialisierung des Codes, die aber nicht weiter ins Gewicht fällt, wenn nur wenige Observer beim Subject registriert sind. Hat ein Subject eine große Anzahl von flukturierenden Observern, d. h., es melden sich häufig neue Observer an und ebenso auch wieder beim Subject ab, bietet sich diese Implementierungsweise von `notify_observers()` an, da auf diese Weise keine Exceptions zu erwarten sind.

PARALLELE IMPLEMENTIERUNG – MIXED STRATEGY:

Um dieser drohenden Sequentialisierung zu entgehen, bietet sich noch eine dritte Alternative an: Man gestattet den Aufruf der Methode `remove_observer()` nur, wenn gerade kein `notify_observers()`-Vorgang aktiv ist. Um dieses Verhalten zu erreichen, wird auf den von AthenaMP angebotenen Reader-Writer Lock (kurz RW-Lock) zurückgegriffen. Wird zum Aufrufzeitpunkt von `remove_observer()` über alle registrierten Observer iteriert, wartet die `remove_observer()`-Methode, bis die `notify_observers()`-Methode durchgelaufen ist (siehe Listing 4.3.11 Zeile 02) und entfernt erst anschließend den Observer (Listing 4.3.9 Zeile 06).

```
01  ...
02  // Erzeugt einen RW-Lock
03  rw_lock<omp_lock_ad, is_writer_preferred> rw_lock_;
04  ...
05  void notify_observers() {
06      rw_lock_.set_r();
07      list_ts<observer*>::iterator it;
08      for( it = observers_.begin(); observers_.end() != it; ++it ) {
09          observer* obs = *it;
10          obs->update();
11      }
12      rw_lock_.unset();
13  }
14
15  ...
16
17  void remove_observer( observer* o ) {
18      rw_lock_.set_w();
19      observers_.remove(o);
20      rw_lock_.unset();
21  }
```

Die kritischen Stellen sind durch RW-Locks geschützt.

Listing 4.3.11: Zweite alternative Implementierungen der Methode `remove_observer()`.

Die Methode `notify_observers()` ist dabei nahezu identisch zu Listing 4.3.1 mit der einzigen Ausnahme, dass ein zu entfernender Observer nur gelöscht wird, wenn gerade kein `notify_observers()`-Vorgang aktiv ist. Dies wird durch den von der AthenaMP-Bibliothek angebotenen RW-Lock erreicht (Zeile 02). Bei Eintritt in die Methode wird der RW-Lock `rw_lock_` lesend gesetzt (Zeile 06) und beim Verlassen wieder entsperrt (Zeile 12). Beim Aufruf von `remove_observer()` wird der RW-Lock zu Beginn schreibend gesetzt (Zeile 17) und beim Verlassen wieder entsperrt (Zeile 20). Dies bietet gegenüber dem einfachen Lock den Vorteil, dass die `notify_observers()`-Methode gleichzeitig von mehreren Threads aufgerufen werden kann. Hat ein Subject eine geringe Anzahl von flukturierenden Observern, d. h., es melden sich nur selten bis nie Observer ab, bietet sich diese Implementierungsweise von `notify_observers()` an, da auf diese Weise keine Exceptions zu erwarten sind. Jedoch treten so die zuvor erwähnten Dateninkonsistenzen wieder auf.

VOR- UND NACHTEILE/NUTZEN:

Das hier vorgestellte Observer-Pattern lässt sich durch die angebotenen Update-Strategien leicht und flexibel auf viele Problemstellungen anwenden und ist somit auch in parallelen Anwendungsfeldern ein probates Mittel, um Daten-Änderungen eines Objekts an andere interessierte Objekte weiterzuleiten. Tabelle 4.3.1 listet die drei Möglichkeiten mit ihren Vor- und Nachteilen auf.

Strategy:	Bemerkung:
`none`	siehe Listing 4.3.1 *Pro*: Schnell, gut und einfach, wenn nie `remove_observer()` aufgerufen wird. *Contra*: Keine Locking-Strategie, `remove_observer()` kann Exceptions verursachen, kann zu Dateninkonsistenzen bei mehrmaligen gleichzeitigen Aufruf von `notify_observers()` bei registrierten Observern führen.
`the_big_lock`	siehe Listing 4.3.10 *Pro*: Lock um jeden kritischen Abschnitt, keine Dateninkonsistenzen. *Contra*: Lock um jeden kritischen Abschnitt, kann zu Serialisierung führen.
`mixed`	siehe Listing 4.3.11 *Pro*: Zwischending aus ‚`none`' und ‚`the_big_lock`', nutzt Reader-Writer Lock. Keine Exceptions. *Contra*: Kann, ähnlich wie ‚`none`', zu Dateninkonsistenzen führen.

Tabelle 4.3.1: Zeigt die Template-Parameter für die drei möglichen Strategien und deren Vor- und Nachteile, mit denen die Klasse `subject` *seine registrierten Observer benachrichtigen kann.*

4.4 PIPELINE PATTERN

KURZBESCHREIBUNG:

Eine Pipeline ist eine Verkettung von Arbeitsstationen, die so zusammenhängen, dass die Ausgabe einer Station die Eingabe der nachfolgenden ist; jede ist auf einen Fertigungsschritt spezialisiert.

MOTIVATION/PROBLEM:

Eine Pipeline lässt sich analog zur Fließbandfertigung betrachten. Die Fließbandfertigung wird in der Regel so organisiert, dass sich der Produktionsprozess auf viele kleine Fertigungsschritte verteilt. Ein Produkt wird durch mehrere Stationen (bzw. Arbeitsplätze) geschleust. Jede Station ist dabei auf einen speziellen Fertigungsschritt spezialisiert. Die Stationen ordnen sich nach der Reihenfolge der Fertigungsschritte.

Das hier vorgestellte Pipeline Pattern basiert auf einer von Mattson *et al.* vorgestellten Idee [Mattson05, S. 103] und ordnet sich in die Kategorie der taskparallelen Pattern ein (vgl. Kapitel 3.2.4 Seite 13).

Das prominenteste Beispiel des Pipeline Patterns ist die von Henry Ford im Jahr 1913 für die Autoproduktion eingeführte Fließbandfertigung. Aber auch im digitalen Bereich finden Pipeline Pattern Verwendung:

- *Instruction Pipeline bei CPUs*: Moderne CPUs bestehen aus mehreren Stationen. Jede Station hat eine bestimmte und genau determinierte Aufgabe, wie z. B.: Instruktionen holen, decodieren, ausführen usw.
- *Signal Verarbeitung*: Um Echtzeitdaten aus dem Bereich der Signalverarbeitung weiterzuverarbeiten, kann der Datenstrom durch mehrere Filter geschickt werden. Jeder Filter ist eine Station, die wiederum pipelineartig angeordnet ist.
- *Bildverarbeitung*: Soll ein Set von Operationen wie z. B. „Ändere Größe!“, „Passe Farbwerte an!“, „Entferne rote Augen!“ usw. auf eine große Anzahl von Bildern angewendet werden, können diese Operationen auch als Stationen einer Pipeline modelliert werden.
- *UNIX-Shell Programme:* Befehle in der Unix Shell können über den sogenannten Pipe-Operator zu einer Pipeline verknüpft werden.

Alle aufgeführten Beispiele haben gemein, dass auf jedem durch die Pipeline laufenden Datenelement nacheinander eine zuvor definierte Sequenz von Operationen ausgeführt wird (siehe Listing 4.4.1).

```
01 while (Aufgabe noch nicht erledigt)
02 {
03     empfange Daten von der Vorgänger-Station
04     führe Task auf Daten aus
05     sende Daten an die Nachfolger-Station weiter
06 }
```

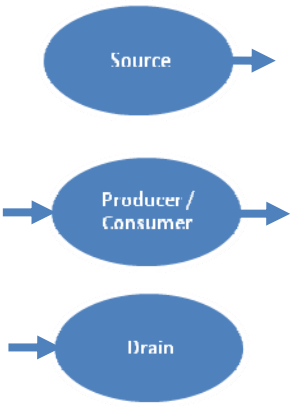
Listing 4.4.1: Allgemeine Struktur einer Pipeline Station in Pseudo-Code [nach Mattson04 S. 103].

Um das Pipeline Pattern anwenden zu können, wird eine Aufgabe genommen und diese in viele Teilaufgaben, sogenannte Tasks, zerlegt. Diese Tasks werden nun den verschiedenen Stationen zugewiesen. Jede Station erledigt dabei immer genau einen Task; d. h., sie erledigt immer dieselbe Aufgabe mit jeweils neu-

en Daten (siehe Zeile 04, Listing 4.4.1). Eine Station empfängt dabei Daten von ihrer vorhergehenden Station (siehe Zeile 03, Listing 4.4.1) und die Daten werden, nachdem ein Task auf diese angewendet wurde, an die nachfolgende Station weitergereicht (siehe Zeile 05, Listing 4.4.1).

Wenn die Anzahl der zu bearbeitenden Daten bekannt ist, kann jede Station die Anzahl der bereits abgearbeiteten Elemente mitzählen und ihre Tätigkeit einstellen, wenn alle Arbeit erledigt ist. Ist die Anzahl der zu bearbeitenden Elemente hingegen nicht bekannt, arbeitet jede Station für gewöhnlich solange, bis sie eine Poison-Pill erhält (vgl. Seite 27).

Je nach Position innerhalb der Pipeline kann eine Station unterschiedliche Ausprägungen besitzen:



- Source (Quelle): Diese Station besitzt nur einen Nachfolger. Sie hat keinen Vorgänger und kann somit keine Eingabe-Daten von außerhalb verarbeiten.
- Lineare Stationen haben für gewöhnlich einen Vorgänger und einen Nachfolger.
- Drain (Abfluss). Diese Station hat nur einen Vorgänger. Sie hat keinen Nachfolger.

Abbildung 4.4.1 zeigt mögliche Spezialisierungen der Station einer Pipeline [nach Lea99].

Für Stationen vom Typ Source gilt die Einschränkung, dass sie nur am Anfang der Pipeline eingesetzt werden können. Im Gegenzug gilt für Stationen vom Typ Drain die Einschränkung, dass sie nur am Ende der Pipeline eingesetzt werden können. Diese Einschränkung gilt nur, wie in unserem Fall, bei einer linearen Pipeline, bei nichtlinearen Pipelines ist sie aufgehoben.

Zeitliche Abhängigkeiten unter den Aufgaben definieren die Reihenfolge, in der die Tasks abgearbeitet werden müssen (ordering constraints) und somit definieren sie auch die Reihenfolge der Stationen in der Pipeline.

Im Falle der Fließbandfertigung nach Henry Ford könnte sich die Aufgabe, Automobile herzustellen, aus folgenden Tasks zusammensetzen: Motor einbauen, Windschutzscheibe montieren, Karosserie auf Fahrgestell montieren, in einer gewünschten Farbe lackieren, Türen einsetzen, Radio verdrahten usw. Bei der Anordnung der Stationen muss auf zeitliche Abhängigkeiten geachtet werden: So kann dass Radio erst verdrahtet werden, nachdem die Karosserie auf dem Fahrgestell montiert ist, um nur ein Beispiel aufzuführen (siehe Abbildung 4.4.2).

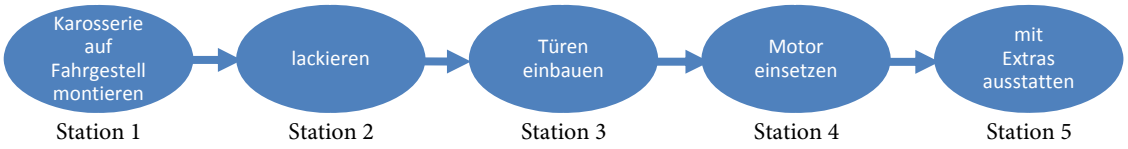


Abbildung 4.4.2: Beispiel einer linearen Pipeline mit fünf Stationen zur Herstellung eines Automobils.

Das Pattern funktioniert am besten, wenn jeder Task pro Station ungefähr gleichviel Arbeitszeit benötigt. Wenn eine Station signifikant mehr oder weniger Arbeitszeit als eine andere braucht, entwickeln sich langsamere Stationen zu einem Flaschenhals (Bottleneck) für die Pipeline, so dass das ganze System aus-

gebremst wird. Ist dies der Fall, muss ein zu rechenintensiver und somit auch zeitintensiver Task evtl. auf andere Stationen aufgeteilt werden. Im Gegenzug bietet es sich an, mehrere kleine nicht sehr rechenintensive Tasks, deren Kommunikationsoverhead größer als der Rechenaufwand ist, zu einem größeren Task zusammenzufassen. Dabei muss jedoch immer auf zeitliche Abhängigkeiten Rücksicht genommen werden. Der Grad der Parallelität dieses Patterns wird durch die Anzahl der Stationen bestimmt. Dies lässt sich auf folgende einfache Formel reduzieren:

Je mehr Stationen, desto mehr Parallelität.

Jedoch müssen die Daten auch zwischen den einzelnen Stationen transportiert bzw. verschickt werden. Dies verursacht wieder den zuvor bereits erwähnten Kommunikationsoverhead. Dabei sollte darauf geachtet werden, dass der Kommunikationsoverhead immer kleiner als der zeitliche Rechenaufwand einer Station ist. Bei OpenMP wird der Kommunikationsoverhead als Speicherzugriffszeit gemessen.

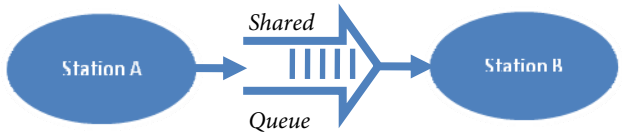


Abbildung 4.4.3: Zwei Stationen, Station A und Station B, sind durch eine Shared Queue (siehe Kapitel 4.2 auf Seite 22), die als Datenbuffer dient, miteinander verbunden (ähnlich der Abbildung 4.2.4 auf Seite 26).

Da Stationen in den seltensten Fällen perfekt synchronisiert sind und die Bearbeitungszeit der Daten pro Station schwankt, sollte ein Datenspeicher zwischen den Stationen geschaltet und der Datenaustausch geordnet sein. Die Shared Queue bietet sich als Datenspeicher zwischen den einzelnen Stationen an, um Schwankungen in der Bearbeitungszeit aufzufangen (siehe Abbildung 4.4.3). Dabei können zwei benachbarte Stationen in einer Pipeline als Producer und Consumer angesehen werden (bekannt aus Kapitel 4.2). Station A produziert nur Daten und Station B konsumiert Daten. Schaltet man nun sehr viele Producer / Consumer in Reihe, ergibt sich eine lockfreie Pipeline.

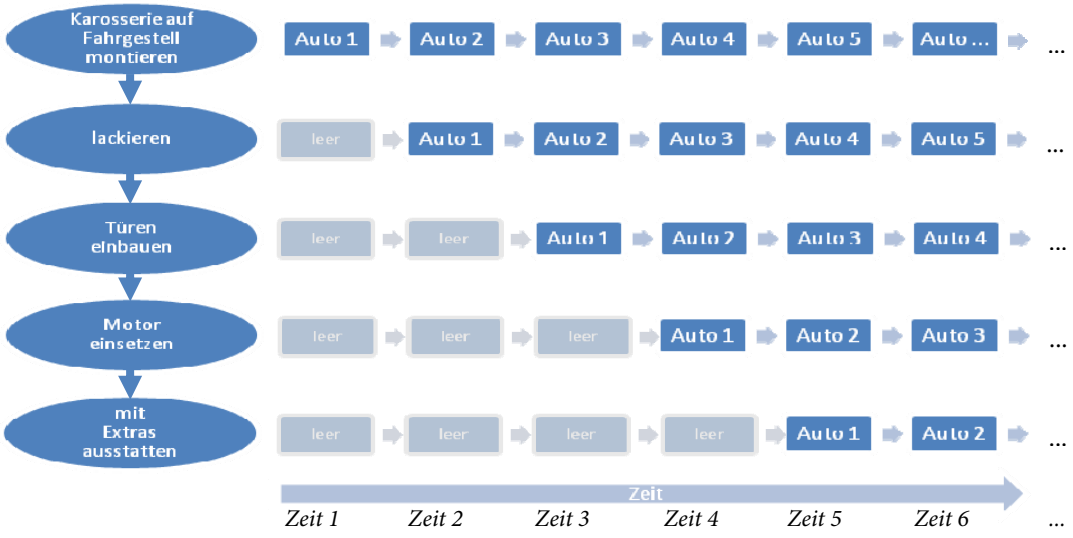


Abbildung 4.4.4 zeigt die Auslastung der einzelnen Stationen der Automobil-Pipeline zu Beginn ihrer Inbetriebnahme. Dies wird auch als Füll-Zeit der Pipeline bezeichnet.

Weiterhin muss berücksichtigt werden, dass die Pipeline eine gewisse Füll- und Leer-Zeit benötigt. Der Begriff Füll-Zeit bezeichnet, wie lange die Pipeline braucht, bis sie mit Daten gefüllt ist (*Filling the Pipeline*). Diese Zeit wird von der Anzahl der Stationen beeinflusst. Mit jeder in der Pipeline enthaltenen Station verlängert sich die Füll-Zeit; sie sollte deshalb vergleichsweise klein im Gegensatz zu der Gesamtlaufzeit der Pipeline sein (siehe Abbildung 4.4.4 vorherige Seite). Die Leer-Zeit ist dabei analog zur Füll-Zeit zu betrachten, bezeichnet jedoch, wie lange es dauert, bis die Pipeline keine Daten mehr enthält (*Draining the Pipeline*) [nach Wilkinson05, Seite 143].

Im Folgenden sollen zwei verschiedene Pipeline-Implementierungen vorgestellt werden. Kapitel 4.4.1 stellt eine nicht-typsichere Pipeline vor, während sich Kapitel 4.4.2 mit einer typsicheren Pipeline befasst. Dabei geht jedes Unterkapitel im Speziellen auf die in ihm vorgestellte Pipeline-Variation und deren Implementierung ein. Abschließend werden die beiden Pipeline-Variationen noch miteinander verglichen.

4.4.1 PIPELINE (NON-TYPESAFE)

Das „non-typesafe“ steht für „nicht-typsicher“. Nicht-typsicher bedeutet, dass mit Void-Pointern (void*) gearbeitet wird, d. h., es findet keine Typprüfung der in den einzelnen Stationen enthaltenen Tasks statt. Auf Kosten der Laufzeitsicherheit wird so eine Flexibilität erkauft, die die Handhabung dieses Patterns vereinfacht, da die einzelnen Stationen der Pipeline automatisch miteinander verknüpft werden. Die nicht vorhandene Typsicherheit kann aber auch schwer zu debuggende Fehler mit sich bringen.

IMPLEMENTIERUNG (PIPELINE NON TYPESAFE):

Die Pipeline besteht hauptsächlich aus zwei Klassen, die alles weitere regeln: Die Klasse `pipeline` selbst und die Klasse `pipe_stage_auto` (siehe Abbildung 4.4.5). Zwischen ihnen besteht eine 1 zu N Beziehung. Eine Pipeline kann mehrere `pipe_stage_auto` enthalten. Die beiden Klassen wurden als Friend-Klassen konzipiert, dies ermöglicht, nahezu alle Methoden als privat zu deklarieren. Dadurch konnten alle für den internen Ablauf der Pipeline benötigten Methoden als **`private`** deklariert und somit vor dem Nutzer versteckt werden, wodurch sich eine leichtere und übersichtlichere Benutzung ergibt. Der Zweck der anderen drei gezeigten Klassen (`pipe_stage_auto_task_interface`, `task1` und `task2`) ist, die Benutzung des Patterns im weiteren Verlauf des Kapitels zu verdeutlichen.

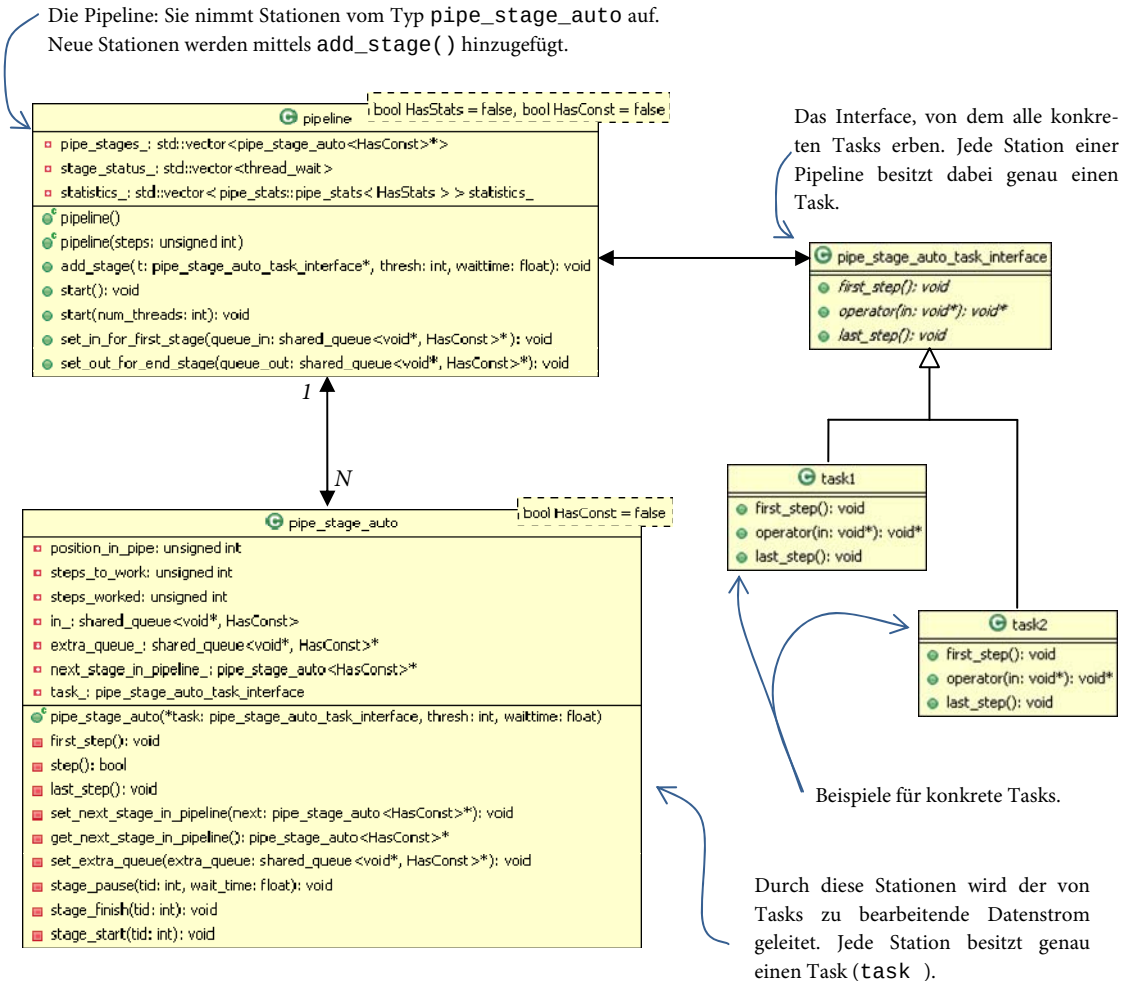


Abbildung 4.4.5: Das Klassendiagramm (vereinfacht) der nicht-typsicheren Pipeline.

Die Grundidee ist nun, dass jeder potenzielle Task einer Station von der Klasse `pipe_stage_auto_task_interface` erbt und deren virtuelle Methoden implementiert (Details siehe im Anwendungsbeispiel Seite 50). Diese konkreten Tasks werden anschließend über die Methode `add_stage(…)` in die Pipeline eingehängt (Listing 4.4.2. Zeile 02 und 03). Jede per `add_stage(…)` hinzugefügte Station wird nun intern von der Pipeline mit seiner Vorgängerstation durch eine Shared Queue verknüpft. Dabei entscheidet der Template-Parameter **`HasConst`** der Klasse `pipeline`, ob die genutzte Shared Queue ihre Größe in linearer Laufzeit ermitteln soll oder nicht (vergleiche **`HasConstSizeRunTime`** Kapitel 4.2). (Der zweite Template-Parameter namens **`HasStats`** wird im Detail im Abschnitt „Implementierung des Schedulers und der Statistiken:“ auf Seite 59 erläutert, da er für beide Pipelines gleich ist.)

Standardmäßig wird davon ausgegangen, dass die als erste eingefügte Station eine Daten erzeugende Station ist (siehe Abbildung 4.4.1 Source) und die als letzte eingefügte Station nur Daten konsumiert (siehe Abbildung 4.4.1 Drain). Werden die Daten nicht von der Pipeline erzeugt, sondern liegen sie bereits vor, kann mittels der Methode `set_in_for_first_stage(…)` optional eine Shared Queue mit der Pipeline verknüpft werden. Diese Shared Queue reicht Daten in die Pipeline und somit in die erste Station hinein (Listing 4.4.2. Zeile 05). Sollen von der Pipeline erzeugte oder bearbeitete Daten im späteren Programmablauf weiterbenutzt werden, so besteht die Möglichkeit mithilfe der Methode `set_out_for_last_stage(…)` eine Shared Queue mit der Pipeline zu verknüpfen (Listing 4.4.2. Zeile 06). Wird keine dieser beiden Methoden benutzt, ist es nicht ohne weiteres möglich, Daten aus dem Program in die Pipeline zu geben oder welche aus ihr herauszubekommen. Eine mögliche Alternative wäre, dass die erste Station in der Pipeline Daten aus einer Datei liest, während die letzte Station der Pipeline Daten in eine (andere) Datei schreibt.

Der Konstruktor der Pipeline besitzt einen optionalen Parameter. Er ist vom Typ **`unsigned int`** und gibt die Anzahl der Arbeitsschritte jeder Station an. Hat jede Station diese Anzahl von Arbeitsschritten erledigt, wird sie beendet. Wird ein Objekt der Pipeline ohne Parameter erzeugt, wird der Standardkonstruktor aufgerufen, der davon ausgeht, dass eine Station solange arbeitet, bis ein zuvor definiertes Poison-Element gefunden wird (Listing 4.4.2. Zeile 01).

```
01 pipeline<HasStats, HasConst> pl;
02 pl.add_stage(new task1);
03 pl.add_stage(new task2);
04 ...
05 pl.set_in_for_first_stage(&queue_in); // optionale Queue
06 pl.set_out_for_end_stage(&queue_out); // optionale Queue
07 ...
08 pl.start();        // Start der Pipeline.
```

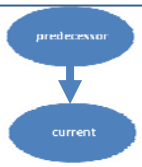
Fügt exemplarisch zwei neue Stationen hinzu. `task1` und `task2` erben von der Klasse `pipe_stage_auto_task_interface` (siehe Abbildung 4.4.5)

Listing 4.4.2: Zeigt exemplarisch die Konstruktion einer Pipeline mit mehreren Stationen.

Doch bevor genauer auf das Herzstück der Pipeline die `start()`-Methode eingegangen wird, soll noch die `add_stage(…)`-Methode betrachtet werden:

```
01 template< bool HasStats, bool HasConst >
02 void pipeline<HasStats, HasConst>::add_stage(
03 pipe_stage_auto_task_interface * t, unsigned int thresh, float waittime)
04 {
05     pipe_stage_auto<HasConst> *current =
06         new pipe_stage_auto<HasConst>(t, thresh, waittime);
07
08     if (pipe_stages_.size() != 0)
09     {
10         // Station mit Vorgänger-Station verknüpfen.
11         pipe_stage_auto<HasConst>* predecessor = pipe_stages_.back();
12         predecessor->set_next_stage_in_pipeline(current);
13     }
14     // Der Station eine Kante zur Pipeline ziehen und ihr mitteilen,
15     // welche Position sie innerhalb der Pipeline besitzt.
16     current->set_pipeline(this, pipe_stages_.size());
17
18     // Wird die Pipeline mit einer festen Arbeitsschrittanzahl
19     // aufgerufen, wird an dieser Stelle jeder Station mitgeteilt,
20     // wie viele Schritte sie arbeiten muss, bevor sie beendet wird.
21     if (steps_ > 0) current->steps(steps_);
23
24     // Anschließend wird die aktuelle Station der Pipeline auf einen
25     // Vector gepushed, der alle Stationen dieser Pipeline enthält.
26     pipe_stages_.push_back(current);
27
28     // Hilfsinformationen zur aktuellen Station initialisieren.
29     // Der stage_status_ - Vektor enthält Informationen bzgl. des Status
30     // einer Station.
31     stage_status_.push_back(thread_wait());
32 }
```

Erstellt neue Station und übergibt ihr ihren Task (`t`).



Listing 4.4.3 zeigt die Implementierung der Methode `add_stage(…)`.

Eine neue Station (`current`) wird in Zeile 05 erstellt. Dieser Station wird der Task `t` übergeben. Die Besonderheit der nicht-typsicheren Pipeline zeigt sich aber in den Zeilen 11 und 12. In ihnen wird die aktuell erstellte Station (`current`) mit der zuletzt eingefügten Station (`predecessor`) automatisch verknüpft. Dies stellt sicher, dass die Ausgabe der vorherigen Station gleich der Eingabe der aktuellen Station ist. Zeile 08 sorgt dafür, dass dieser Verknüpfungsvorgang nicht für die erste Station der Pipeline vorgenommen wird, da die erste Station immer vom Typ Source sein muss. Damit eine Station während des Programmlaufes der Pipeline Anweisungen geben kann, wird direkt im Anschluss noch eine Verbindung von der aktuellen Station zur Pipeline gezogen (Zeile 16) und ihr ihre aktuelle Position innerhalb der Pipeline mitgeteilt (`position_in_pipe_`). Eine mögliche Anweisung wäre z. B., die Vorgänger-Station „schlafen zu legen“, damit keine neue Arbeit nachproduziert wird, und um so zunächst einmal alle noch ausstehenden Arbeitsschritte abzuarbeiten. In Zeile 26 wird die neu hinzugefügte Station (`current`) auf einen Vektor gepushed, der alle in der Pipeline enthaltenen Stationen speichert. In Zeile 31 werden noch diverse Hilfsinformationen, die für den späteren Ablauf der Pipeline von Relevanz sind, in einem Vektor namens `stage_status` gespeichert (Details siehe Listing 4.4.4).

```
01 struct thread_wait {
02     // Enum, spiegelt den Status einer Station wider.
03     pipe_stage_strategy thread_what_to_do_;
04
05     // Gibt die Zeit in Sekunden an, die eine Station
06     // schläft, sofern sie schlafen gelegt wird.
07     float thread_forced_to_wait_time_;
08     // Hat die aktuelle Station den Scheduler?
09     bool scheduling_;
10 };
```

Der Enum, kann folgende Werte annehmen:

stopped
Station ist gestoppt.

work
Station arbeitet.

sleep
Station schläft.

finished
Station ist beendet.

Listing 4.4.4 zeigt die Hilfsstruktur `thread_wait`; `thread_wait` enthält Informationen über den Thread-Status einer jeden Station.

Nachdem die einzelnen Stationen erzeugt und mit der Pipeline verknüpft sind, kann die Pipeline mittels `start()` gestartet werden (Listing 4.4.2. Zeile 09). Wichtig ist, dass `start()` erst aufgerufen werden darf, nachdem die Pipeline komplett konstruiert wurde. Strukturelle Änderungen zur Laufzeit, wie z. B. das Hinzufügen oder Entfernen neuer Stationen, sind nicht gestattet und führen zu undefiniertem Verhalten.

Betrachten wir nun das Herzstück selbst, die `start()`-Methode. Die Betrachtung bezieht sich jedoch auf eine stark vereinfachte Version ohne Scheduler und ohne die Möglichkeit Stationen „schlafen zu legen“, um mit der Idee hinter der Methode vertraut und nicht durch unnötige Details abgelenkt zu werden (Listing 4.4.5). Der Scheduler wird im Detail im Abschnitt „Implementierung des Schedulers und der Statistiken:“ auf Seite 59 erläutert, da er für beide Pipelines gleich ist.

Die Methode lässt sich in drei Abschnitte zerlegen, die zur besseren Übersicht farblich voneinander abheben:

1. *Initialisierungsarbeiten (inklusive Aufruf der Methode* `first_step()`*)*
2. *Hauptteil*
3. *Abschlussarbeiten (inklusive Aufruf der Methode* `last_step()`*)*

```
01  template< bool HasStats, bool HasConst >
02  void pipeline<HasStats, HasConst>::start(unsigned int num_threads)
03  {
04      // Anzahl, der in der Pipeline befindlichen Stationen.
05      current_working_stages_ = pipe_stages_.size();
06
07      for (int i = 0; i < pipe_stages_.size(); ++i) {
08          stage_status_[i].thread_what_to_do_ = work;
09          pipe_stages_[i]->first_step();
10      }
11
12
13  #pragma omp parallel
14      {
15          int stage_id = omp_get_thread_num();
16          bool stage_finished;
17          // Schleife läuft bis alle Stationen
18          // ihre Arbeit beendet haben.
19          while (current_working_stages_!=0)
20          {
21              stage_finished = false;
23
24              // Hauptteil:
25              if (stage_status_[stage_id].thread_what_to_do_ == work)
26              {
27                  stage_finished = pipe_stages_[stage_id]->step();
28              }
29              // Hat eine Station ihre Arbeit
30              // erledigt (true), wird sie beendet.
31              if (stage_finished)
32              {
33                  stage_status_[stage_id].thread_what_to_do_ = finished;
34  #pragma omp atomic
35                  --current_working_stages_; // eine Station weniger!
36              }
37          }//while
38      }//pragma omp parallel
39
40      for (int i = 0; i < pipe_stages_.size(); ++i) {
41          pipe_stages_[i]->last_step();
42      }
43  }//void start(…)
```

Initialisiert alle Stationen mit dem ‚*work*‘-Status und führt den ersten Schritt einer jeden aus (`first step()`).

Solange der ‚*work*‘-Status einer Station gesetzt ist, führt sie ihren Arbeitsschritt aus. Hat eine Station ihre Arbeit komplett erledigt, wird `true` zurückgegeben.

Ruft die `step()`-Methode einer konkreten Station auf.

Abschließend wird für jede Station `last_step()` ausgeführt.

Listing 4.4.5 zeigt die `start()`*-Methode der Pipeline in Auszügen (ohne Scheduler und die Möglichkeit, Threads „schlafen zu legen“).*

Der Ablauf der Pipeline wird durch eine parallele Region bestimmt (Listing 4.4.5, Zeile 13 bis 38). In dieser wird für jede Station ein Thread erzeugt. Jeder dieser Threads führt eine Methode der Klasse `pipe_stage_auto` namens `step()` aus (Listing 4.4.5, Zeile 27). Sie ist dafür verantwortlich, dass der virtuelle `operator()`-Aufruf ausgeführt wird. Dieser Aufruf wird von dem zugewiesenen Task der Station bestimmt. Er wird in Listing 4.4.6 auf Seite 48f detailliert erörtert. Liefert `step()` `true` zurück, so wird innerhalb des if-Blocks die Station beendet (Zeile 31 bis 36) und die Anzahl der aktiven, arbeitenden Stationen (`current_working_stages_`) um eins vermindert (Listing 4.4.5 Zeile 35). Sind keine Stationen der Pipeline mehr aktiv (der Fall `current_working_stages_` gleich 0), werden die while-Schleife (Zeile 19) und somit die parallele Region verlassen. Dies beendet die Pipeline.

Abgerundet wird die `step()`-Methode durch die Möglichkeit, in den konkreten Stationen `first_step()` und `last_step()` der Klasse `pipe_stage_auto` auszuführen. `first_step()` wird einmalig pro Station als erster Arbeitsschritt ausgeführt (Listing 4.4.5 Zeile 09). `last_step()` funktioniert analog, wird jedoch erst ausgeführt, nachdem die Pipeline komplett durchgelaufen ist (Listing 4.4.5 Zeile 41). Dieses Methodenpaar bietet sich an, um in den einzelnen Stationen einmalig benötigte Initialisierungsarbeiten vorzunehmen, wie z. B. das Öffnen eines Dateihandles und dessen ordnungsgemäßes Schließen nach erledigter Arbeit.

Bei Verwendung der Klasse `pipe_stage_auto` muss – im Gegensatz zu der in Kapitel 4.4.2 vorgestellten typsicheren Pipeline – nicht umständlicherweise zwischen dem Typ einer Station unterschieden werden (siehe Abbildung 4.4.1), dies geschieht automatisch.

Die letzte Methode der nicht-typsicheren Pipeline, die für die Implementierung von Interesse ist, ist die in Listing 4.4.6 (vereinfachte) gezeigte `step()`-Methode. Sie lässt sich, abhängig vom Typ der Pipeline-Station, in drei Abschnitte zerlegen. Der Typ der Pipeline-Station gibt an, welcher Abschnitt der Methode ausgeführt werden soll. Diese Abschnitte sind zur besseren Übersicht farblich voneinander abgehoben:

1. *Source* (Zeile 8 bis 53)
2. *Producer / Consumer* (Zeile 58 bis 76)
3. *Drain* (Zeile 82 bis 109)

```
01  template<bool HasConst>
02  bool athenamp::pipe_stage_auto<HasConst>::step()
03  {
04     void* output = NULL;
05     // Stationen von Typ „Source“ (Abschnitt 1)
06     if (position_in_pipe_ == 0)
07     {
08        output = (*task_)( NULL );
09
10        if (output == NULL)
11        {
12           next_stage_in_pipeline_->in_.push_back( NULL );
13           return true;
14        }
15
16        ++steps_worked_;
17
18        next_stage_in_pipeline_->in_.push_back( output );
19     }//„Source“
```

Der zugewiesene Task wird ausgeführt.

Wird ein Poison-Element zurückgeliefert, schließt die Station.

Das Ergebnis wird an die nachfolgende Station weitergereicht.

```
20  // Stationen vom Typ „Producer/Consumer“ (Abschnitt 2)
21     else if (next_stage_in_pipeline_ != NULL && !in_.empty())
23     {
24        if (in_.front()==NULL)
25        {
26           next_stage_in_pipeline_->in_.push_back( NULL );
27           return true;
28        }
29
30        output = (*task_)( in_.front() );
31        in_.pop_front();
32
33        ++steps_worked_;
34        next_stage_in_pipeline_->in_.push_back( output );
35     }//„Producer/Consumer“
36
37     // Stationen vom Typ „Drain“ (Abschnitt 3)
38     else if (next_stage_in_pipeline_ == NULL && !in_.empty())
39     {
40        if (in_.front()==NULL)
41        {
42           return true;
43        }
44
45        output = (*task_)( in_.front() );
46        in_.pop_front();
47
48        ++steps_worked_;
49     }//„Drain“
50
51     // Allgemeiner Teil:
52     if (step_ && steps_to_work_ == steps_worked_)
53     {
54        return true;
55     }
56
57     return false;
58  }
```

Poison-Element – analog zu Zeile 10 bis 14).

Der zugewiesene Task wird ausgeführt.

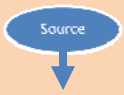
Poison-Element – analog zu Zeile 10 bis 14).

Der zugewiesene Task wird ausgeführt.

Gelangt ein Durchlauf bis hierhin, bedeutet es entweder, dass keine Arbeit im aktuellen Durchlauf gefunden wurde oder dass noch weitere Arbeit auf die Station wartet. .

Listing 4.4.6 zeigt die (vereinfachte) `step()`-Methode der Klasse `pipe_stage_auto`.

Abschnitt 1:



Eine Station vom Typ Source befindet sich immer an Position 0 (siehe Zeile 06). Würde eine Source-Station an einer anderen Stelle eingefügt werden, wäre der Datenfluss innerhalb der Pipeline blockiert. Der Abschnitt für Stationen vom Typ Source erstreckt sich über die Zeilen 06 bis 19.

Die eigentliche Funktionalität für Stationen dieses Typs befindet sich in Zeile 08. Stationen vom Typ Source produzieren nur Ausgabe-Daten und enthalten keinen Eingabe-Parameter. Dort wird der vom Nutzer übergebene, als Funktor implementierte Task (`task_`) ausgeführt. Die produzierten Daten werden in Zeile 18 an die nachfolgende Station weitergereicht.

Stationen vom Typ Source enden, wie alle Stationen einer Pipeline, wenn sie ein Poison-Element erhalten. Das Poison-Element wird bei der nicht-typsicheren Pipeline durch den `NULL`-Pointer repräsentiert. Ob ein `NULL`-Pointer zurückgeliefert wurde, wird in den Zeilen 10 bis 14 getestet. Dieser Fall ist wichtig, sofern der Pipeline statt eines Poison-Elements eine feste Schrittzahl übergeben wurde.

Weiterhin existiert ein hier nicht gezeigter Spezialfall. Bisher ging man davon aus, dass die erste Station der Pipeline Daten produziert, die dann weitergereicht werden. Für den Fall, dass die Daten schon vorher existieren, also nicht mehr erzeugt werden müssen, bietet es sich an, die Daten direkt in die Pipeline hereinzureichen, so dass sie in dieser Station bearbeitet werden können. Dadurch ändert sich streng genommen der Typ der Station von Source in eine Art Producer/Consumer Typ. Dieser Fall greift, wenn bei der Konstruktion der Pipeline mittels `set_in_for_first_stage(…)` eine Shared Queue, die bereits Daten enthält, mit der Pipeline verknüpft wurde (siehe Listing 4.4.2 Zeile 5).

Abschnitt 2:

Besitzt eine Station einen Nachfolger und befindet sie sich nicht an der ersten Stelle innerhalb der Pipeline, wird ihr Verhalten in den Zeilen 21 bis 35 gesteuert. Der als Funktor übergebene Task nimmt Daten seiner Vorgänger-Station entgegen (`in_.front()`), erledigt die vom Benutzer definierte Aufgabe und liefert das Ergebnis zurück (Zeile 30). Dieses wird in die Eingabe Queue der nachfolgenden Station eingetragen (Zeile 34).
Die Zeilen 24 bis 28 prüfen die in die Station kommenden Daten auf ein Poison-Element, den `NULL`-Pointer. In diesem Fall, wird analog zu Abschnitt 1 in Zeile 10, die Station von der Pipeline beendet, indem **`true`** zurückgeliefert wird.

Abschnitt 3:

In Abschnitt 3, der sich über die Zeilen 38 bis 49 erstreckt, werden Stationen vom Typ Drain behandelt. Drain-Stationen stellen die Negation von Source-Stationen dar. Sie können Daten bekommen und verarbeiten, allerdings liefern sie keine Daten zurück. Sie funktionieren nach dem gleichen Prinzip wie die zwei vorausgehenden Stationstypen. In Zeile 46 wird der Task-Funktor ausgeführt. Dieser liefert dennoch Ergebnisse in eine Variable namens `output` zurück (Zeile 45). Wird die Drain-Station in ihrer normalen Ausprägung verwendet, wird dieser `output`-Parameter nicht benötigt und verworfen.

Wurde aber bei der Konstruktion der Pipeline mittels der Methode `set_out_for_last_stage(…)` eine Shared Queue, die produzierte Daten aufnehmen kann, mit der Pipeline verknüpft (siehe Listing 4.4.2 Zeile 6), greift ein hier nicht weiter im Detail gezeigter Sonderfall. Bei diesem wird das in `output` zurückgelieferte Ergebnis in die so verknüpfte Ausgabe-Queue gepushed. So wird es ermöglicht, von der Pipeline produzierte Daten auch innerhalb des Programms weiter zu nutzen.

Die Überprüfung auf ein Poison-Element findet nicht statt. Stattdessen wird die Station beendet, wenn festgestellt wird, dass die erste Station der Pipeline beendet ist und sie ebenso viele Arbeitsschritte wie diese durchlaufen hat.

Alle drei Stationen aus Listing 4.4.6 besitzen noch einen gemeinsamen Codeblock. Dieser befindet sich in den Zeilen 52 bis 57. Wird eine Pipeline ohne Poison-Element ausgeführt bzw. angewiesen, nach einer festen Schrittzahl zu stoppen, so wird in diesem Abschnitt geprüft, ob diese feste Schrittzahl von einer Station erreicht wurde. Ist dies der Fall, wird der Pipeline **`true`** zurückgeliefert (Zeile 54) und die Station somit beendet.

Greift keiner der oben genannten Fälle oder es befindet sich keinerlei Arbeit in der Eingabe-Queue, so wird `false` zurückgeliefert (Zeile 57) und die Pipeline führt einen weiteren Aufruf der `step()`-Methode durch, bis jede Station `true` zurückliefert.

ANWENDUNGSBEISPIEL (PIPELINE NON TYPESAFE):

Benutzer des Pipeline-Patterns müssen den Task implementieren, der auf einer konkreten Station ausgeführt werden soll. Jede zur Pipeline mittels `add_stage(…)` hinzugefügte Station muss einen Task, der vom gemeinsamen Interface `pipe_stage_auto_task_interface` erbt, übergeben bekommen. Ein Klassenrohbau sieht wie folgt aus, wobei es dem Nutzer überlassen bleibt, was die einzelnen Methoden im Detail erledigen:

```
01 class task_at_station1 : public pipe_stage_auto_task_interface
02 {
03 public:
04     task_at_station1() {}
05
06     // Wird einmalig zu Beginn ausgeführt (vgl. Listing 4.4.5 Zeile 09).
07     void first_step() {}
08
09     // Der eigentliche Arbeitsschritt.
10     void* operator ()(void* value) // Erhalte Daten.
11     {
12         ...
13         // Erledige Arbeit auf den Daten.
14         ...
15         return value;  // Gib Daten zurück.
16     }
17
18     // Wird einmalig am Ende ausgeführt (vgl. Listing 4.4.5 Zeile 41).
19     void last_step() {}
20 };
```

Wird implizit durch die `start()`-Methode der Pipeline aufgerufen (siehe Listing 4.4.5 Zeile 27). Die `step()`-Methode ruft wiederum die nun konkrete Implementierung dieses virtuellen `operator()`-Aufrufs auf (vgl. Listing 4.4.6 Zeile 09, 31 und 46).

Listing 4.4.7 Blaupause eines Tasks an einer konkreten Station.

Das Kernstück einer jeden konkreten Station stellt der `operator()`-Aufruf dar (Listing 4.4.7 Zeile 10). Der `operator()`-Aufruf ist der Task, der auf jeder Station ausgeführt wird. Diese Methode ist in der Klasse `pipe_stage_auto_task_interface` als virtuell markiert und muss implementiert werden. Gleiches gilt für das Methodenpaar `first_step()` und `last_step()` (Zeile 07 und 19), das ebenfalls implementiert werden muss. Sollte es nicht benötigt werden, reicht es, die Methoden ohne Inhalt hinzuschreiben. Dies ist nötig, damit ihr Aufruf in der `start()`-Methode der Klasse `pipeline` keinen Linker-Fehler erzeugt (Listing 4.4.5 Zeile 13 und 97).

Um die Funktionsweise der Pipeline zu verdeutlichen, soll das einführende Automobil-Beispiel herangezogen werden. Die Idee dieses Beispiels basiert auf der Aufgabenstellung der Veranstaltung Parallelverarbeitung I der Universität Kassel im WS 05/06. Es soll eine digitale Autofabrik simuliert werden.

Ein Auto besitzt folgende fünf willkürlich gewählte Eigenschaften:

1. ID: Ein Auto besitzt eine eindeutige fortlaufende Identifikationsnummer
2. Farbe: Die Lackierung des Autos, z. B. rot, grün, blau, …
3. Türen: Die Anzahl der Türen (ein Wert zwischen 2 und 5)
4. Motor: Die Leistung des Autos gemessen in PS
5. Extras: Diverse Extras eines Autos, wie ABS, Radio, …

Listing 4.4.8 zeigt die blanke Datenstruktur, die als Grundlage eines jeden Autos dienen soll:

```
01 struct car
02 {
03     long id;          // Eigenschaft 1, die ID.
04     char color[20];   // Eigenschaft 2, die Farbe.
05     int doors;        // Eigenschaft 3, die Anzahl der Türen.
06     int motor;        // Eigenschaft 4, die Leistung in PS.
07     char* extras;     // Eigenschaft 5, die Extras.
08 };
```

Listing 4.4.8: Die Datenstruktur eines Autos.

Autos werden für gewöhnlich per Fließbandfertigung hergestellt. Da das hier vorgestellte Pattern die digitale Analogie eines Fließbands darstellt, liegt es auf der Hand, dass die gewünschte Autofabrik durch eine Pipeline mit fünf Stationen dargestellt wird, fünf Stationen, da sich in diesem Beispiel ein Auto aus fünf Eigenschaften zusammensetzt (ähnlich Abbildung 4.4.2). Zuerst wird gezeigt, wie die Pipeline mit ihren Stationen konstruiert wird. Hierfür wird das bereits auf Seite 44 gezeigte Listing 4.4.2 adaptiert und so erweitert, dass es zur geforderten Autofabrik passt.

```
01 pipeline<false, true> pl;
02
03 station0_source         task0;
04 station1_identification task1;
05 station2_spraying       task2;
06 station3_doors          task3;
07 station4_motor          task4;
08 station5_extras         task5;
09
10 pl.add_stage(&task0);
11 pl.add_stage(&task1);
12 pl.add_stage(&task2);
13 pl.add_stage(&task3);
14 pl.add_stage(&task4);
15 pl.add_stage(&task5);
16
17 shared_queue<void*, true> queue_out;
18 pl.set_out_for_last_stage(&queue_out);
19
20 pl.start();  // Startet die Pipeline.
```

Erzeugt neue Pipeline. Der erste Template-Parameter (`false`) gibt an, dass keine Statistiken gesammelt werden sollen, während der zweite Parameter (`true`) angibt, wie der Parameter `HasConstSizeRunTime` der intern verwendeten Shared Queue initialisiert wird.

siehe Listing 4.4.10

Fügt neue Stationen hinzu. Diese Stationen müssen von der Klasse `pipe_stage_auto_task_interface` erben.

Zusätzliche Queue. Die letzte Station legt ihre Daten in dieser Queue ab.

Listing 4.4.9: Erweiterung von Listing 4.4.2; zeigt, wie eine Pipeline konstruiert wird

Exemplarisch wird eine mögliche Implementierung der ersten Station aufgeführt:

```
01 class station0_source : public pipe_stage_auto_task_interface {
02     int steps;
03 public:
04     station0_source() {}
05     void first_step() { steps = 0; }
06
07     void* operator ()(void* value) {
08         if (steps < PRODUCE_N_CARS) {
09             steps++;
10             return new car;
11         } else {
12             return NULL;
13         }
14     }
15     void last_step() {}
16 };
```

Der eigentliche Task der Station: Produziere eine bestimmte Anzahl von Autos. Ist diese Anzahl erreicht, wird das Poison-Element zurückgeliefert (`NULL`).

Listing 4.4.10 zeigt eine Station, die Autos produziert.

Die in Listing 4.4.9 in aufgeführten weiteren Stationen 1 bis 5 werden hier nicht mehr gezeigt, da sie analog zu der in Listing 4.4.10 gezeigten Station funktionieren und sich nur in der Implementierung des `operator()`-Aufrufs unterscheiden, der sich jedoch für die Benutzung des Pipeline-Patterns als nicht relevant erweist. (Der vollständige Source-Code dieses Beispiels findet sich als CPP-Unit Test in der AthenaMP-Bibliothek wieder.)

Listing 4.4.10 zeigt eine Station vom Typ Source. Für ihre Ausführung ist Abschnitt 1 aus Listing 4.4.6 zuständig (ohne den dort aufgeführten Sonderfall, da keine Eingabe-Queue gesetzt wurde). Bewusst wurde keine Implementierung einer konkreten Station vom Typ Drain aufgeführt. Eine Station dieses Typs würde sich implementierungstechnisch nicht stark von der in Listing 4.4.10 gezeigten Station unterscheiden, da die Klasse `pipe_stage` entscheidet, um welchen Typ es sich handelt. Auf diese Weise werden viele Fehlerquellen, die der Benutzer machen kann, von vornherein ausgeschlossen.

In Listing 4.4.9 wird Gebrauch von der Besonderheit gemacht, eine Extra-Queue namens `queue_out` anzulegen und mittels `set_out_for_last_stage(&queue_out)` mit der Pipeline zu verknüpfen (Zeile 18). Dies hat den Effekt, dass Daten, nachdem sie alle Pipeline-Stationen durchlaufen haben, in dieser Shared Queue gespeichert werden, so dass sie im späteren Programmverlauf noch genutzt werden können.

4.4.2 PIPELINE (TYPESAFE)

Diese Variante der Pipeline unterscheidet sich von der in Kapitel 4.4.1 vorgestellten nicht-typsicheren Pipeline nur in der Benutzung und in der Implementierung. Die Idee hinter den beiden Pipelines, nämlich der Umsetzung einer parallelen Fließbandfertigung, bleibt unverändert erhalten.

Das „typesafe“ steht für „typsicher“, d. h. es findet eine Typprüfung der Daten statt, die in die Tasks der einzelnen Stationen gereicht werden sollen. Da dies schon während der Compile-Zeit geschieht, beugt diese Implementierung undefiniertem Verhalten aufgrund falscher Zuweisungen vor.

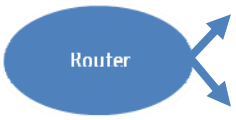
Die Typsicherheit wird durch den Einsatz von Templates erkauft. Während die nicht-typsichere Pipeline automatisch ihre Stationen mit Shared Queues untereinander verknüpft, muss der Verknüpfungsvorgang bei der typsicheren Pipeline per Hand, also durch den Benutzer erfolgen.

Im Gegensatz zur nicht-typsicheren Pipeline können die Stationen der typsicheren Pipeline mit einem beliebigen threadsicheren Datencontainer verknüpft werden, dessen Interface folgende Funktionen anbietet: `push_back()`, `pop_front()`, `front()` und `empty()`. Die hier vorgestellte Implementierung wurde erfolgreich mit folgenden Datentypen getestet:

- Shared Queue (siehe Kapitel 4.2 Seite 22)
- `list_ts` (siehe Kapitel 4.1.1 Seite 18)
- `deque_ts` (siehe Kapitel 4.1.2 Seite 19)

Wichtig ist, dass ein Datencontainer immer genau zwei Stationen miteinander verknüpft.

Stationen der typsicheren Pipeline können zusätzlich zu den in Abbildung 4.4.1 auf Seite 40 vorgestellten Ausprägungen Source, Producer/Consumer und Drain noch zwei weitere Ausprägungen besitzen:



- Router: Diese Station hat zwei Nachfolger. Sie schickt ihre Daten an den Nachfolger, der weniger Datenelemente in seiner Eingabe-Queue hat.

- Collector: Diese Station hat zwei Vorgänger und sammelt die Daten von beiden Vorgängern.

Abbildung 4.4.6 zeigt weitere mögliche Spezialisierungen, die bei einer typsicheren Pipeline genutzt werden können [nach Lea99].

Mit den zusätzlichen Stationen vom Typ Router und Collector ist es nun möglich, den starren linearen Ablauf aufzusplitten. Benötigt eine Station signifikant mehr Arbeitszeit als die restlichen Stationen, bietet es sich an, einen Router in die Pipeline zu setzen. Der Router verteilt den Datenstrom auf zwei „Fließbänder“. Die Daten werden nun von zwei Nachfolger-Stationen bearbeitet, anstatt nur von einer. Dadurch können auf elegante Art drohende Flaschenhälse einer Pipeline von vornherein ausgeschlossen werden. Der aufgeteilte Datenstrom wird anschließend über einen Collector wieder zusammengeführt. Abbildung 4.4.7 stellt eine Erweiterung der bereits bekannten linearen Pipeline aus Abbildung 4.4.2 von Seite 40 dar und soll den Einsatz von Router und Collector veranschaulichen:

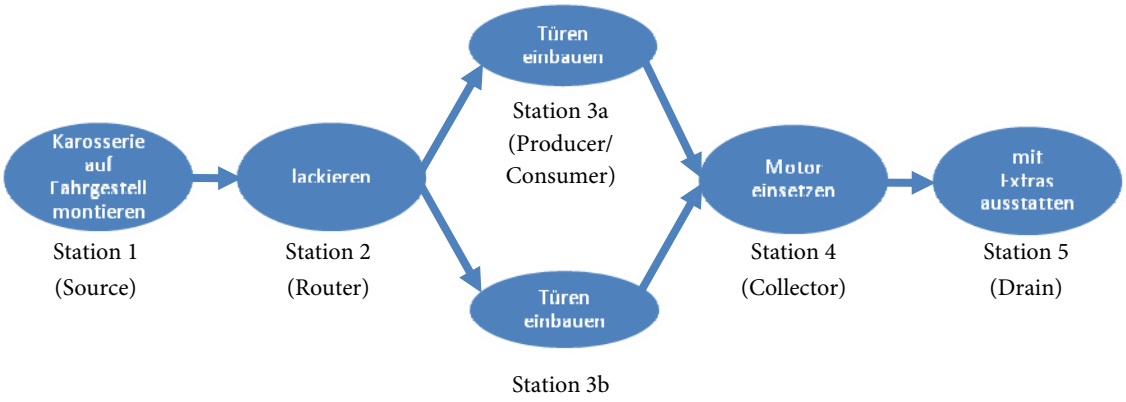


Abbildung 4.4.7: Beispiel einer nicht linearen Pipeline. Geht man davon aus, dass die Arbeit in Station 3 im Vergleich zu den anderen Stationen unverhältnismäßig lange dauert, bietet es sich an, Station 3 in zwei Stationen zu splitten.

Es ist zu beachteten, dass die Daten an Station 2 aufgeteilt werden, d. h., Daten laufen entweder nur durch Station 3a oder nur durch Station 3b. Ist es nötig, dass jedes Auto, das durch die Pipeline aus Abbildung 4.4.7 läuft, eine Tür bekommt, muss der „Tür einbauen"-Task in der Station 3a und 3b vorkommen. Wird dies nicht beachtet, wird ca. die Hälfte der Autos ohne Türen ausgeliefert.

IMPLEMENTIERUNG (PIPELINE TYPESAFE):

Im Gegensatz zu der nicht-typsicheren Pipeline entfällt bei der typsicheren die automatische Bestimmung des Stations-Typs. Bei der nicht-typsicheren Pipeline war es ausreichend einfach, per `add_stage(…)` eine neue Station unter Zuhilfenahme der Klasse `pipe_stage_auto` hinzuzufügen. Der Typ der Station wurde automatisch, abhängig von der Position innerhalb der Pipeline, bestimmt. Bei der typsicheren Pipeline werden Stationen auch per `add_stage(…)` eingefügt. Jedoch steht dem Benutzer nun ein ganzes Paket von Klassen zur Verfügung, aus dem er den Typ der einzufügenden Station selbst bestimmen muss. Die theoretische Unterscheidung, die in Abbildung 4.4.1 und Abbildung 4.4.6 zwischen den einzelnen konkreten Stationstypen vorgenommen wurde, muss nun vom Benutzer explizit angegeben werden. Dabei stehen ihm folgende fünf Klassen zur Verfügung:

- `pipe_stage_source` entspricht einer Station vom Typ *Source*,
- `pipe_stage` entspricht einer Station vom Typ *Producer / Consumer*,
- `pipe_stage_drain` entspricht einer Station vom Typ *Drain*,
- `pipe_stage_router` entspricht einer Station vom Typ *Router*,
- `pipe_stage_collector` entspricht einer Station vom Typ *Collector*.

Abbildung 4.4.8 auf Seite 55 zeigt das Klassendiagramm der typsicheren Pipeline. Es soll helfen, die Zusammenhänge der einzelnen Klassen besser zu visualisieren.

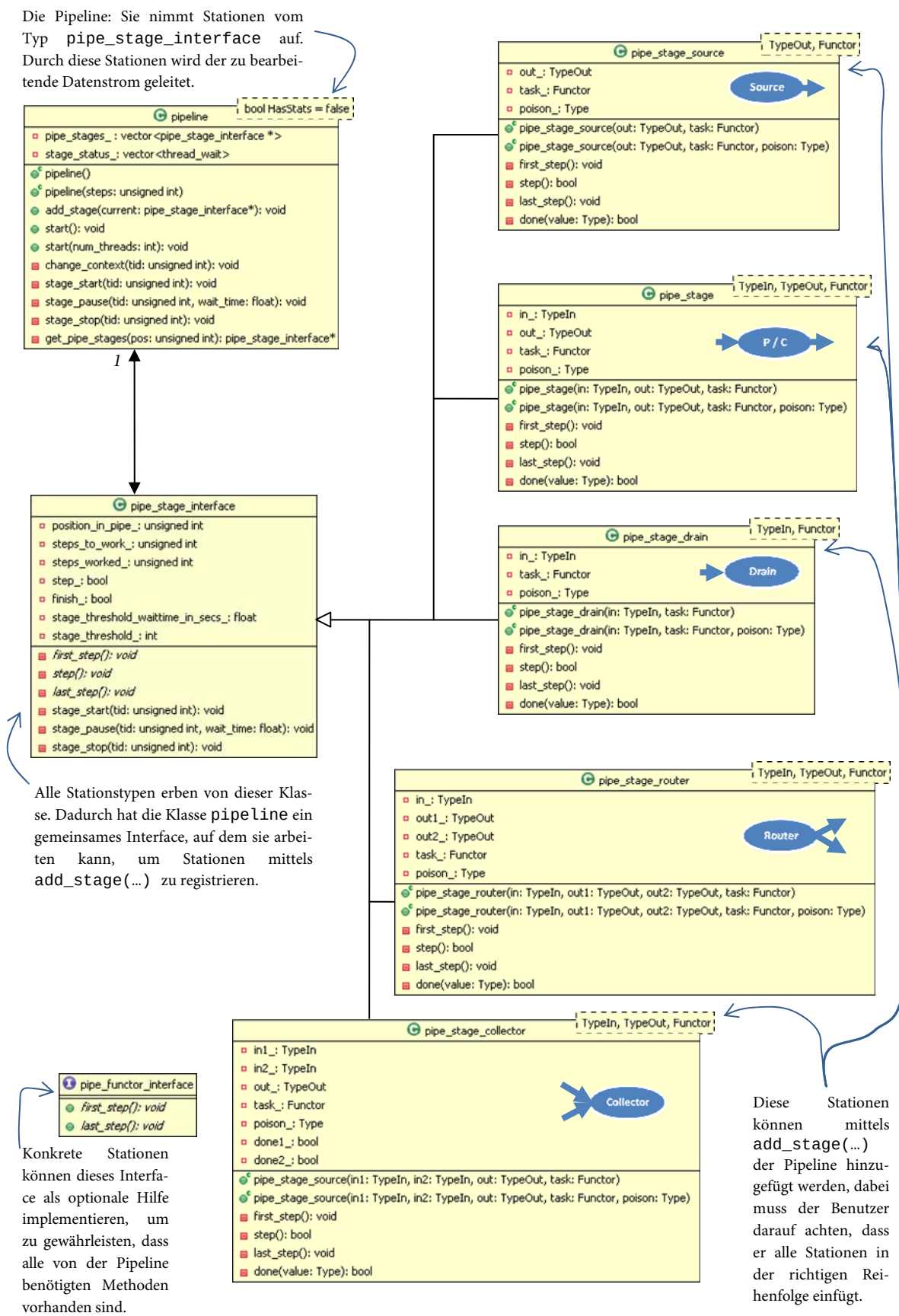


Abbildung 4.4.8: Das Klassendiagramm der typsicheren Pipeline (vereinfacht). Die Klasse `pipeline` kann beliebig viele Stationen aufnehmen, die alle durch die gemeinsame Klasse `pipe_stage_interface` repräsentiert werden.

Die Vielfalt der vielen konkreten Stationstypen mag im ersten Moment verwirrend erscheinen, ist aber nur logisch, da jede Klasse genau einem Pipeline-Typ entspricht. Dabei muss der natürliche Typ beim Einfügen einer Station in die Pipeline beachtet werden. Eine Station vom Typ Source muss als erste, eine Station vom Typ Drain als letzte eingefügt werden muss. Jeder durch einen Router geteilte Datenfluss muss wieder von einem Collector zusammengefügt werden. Wird dies nicht beachtet, kann der Datenfluss nicht korrekt durch die Pipeline fließen. Infolgedessen kann es zu undefiniertem und schwer zu debuggendem Verhalten kommen.

Die fünf zur Verfügung stehenden Klassen erben alle von einer gemeinsamen Klasse namens `pipe_stage_interface`. Dieses „Interface“ gewährleistet, dass alle vorgestellten Stationstypen mittels der `add_stage(…)`-Methode bei der Pipeline registriert werden können. Jeder der fünf konkreten Stationstypen besitzt zwei bis drei Template-Parameter. Alle Stationstypen besitzen die Gemeinsamkeit, einen Funktor-Parameter namens `Functor` zu besitzen. Konkrete Tasks einer Station werden als Funktor in diese hereingereicht. Ein Funktor muss dabei der in Listing 4.4.11 gezeigten Struktur entsprechen. Er kann jedoch von dieser abweichen, da Stationen vom Typ Task keinen Eingabe-Parameter besitzen, während Stationen vom Typ Drain in der Regel keinen Ausgabe-Parameter aufweisen.

```
01  class functor_task
02  {
03  public:
04      functor_task() {}
05
06      void first_step(){}
07
08      TYP_AUSGABE operator() (TYP_EINGABE value)
09      {
10          return value;
11      }
12
13      void last_step() {}
14  };
```

Daten aus der Eingabe-Queue werden hier entgegengenommen; entfällt bei Stationen vom Typ Source

Daten werden an die Ausgabe Queue weitergereicht; entfällt bei Stationen vom Typ Drain.

Listing 4.4.11 Allgemeine Struktur eines Tasks, der in einer Station bearbeitet werden soll und als Funktor realisiert ist.

Je nach Ausprägung einer Pipeline-Station besitzt sie eine unterschiedliche Anzahl an Template-Parametern. Der Template-Parameter `TypeIn` steht für den Datencontainer der Eingabe-Queue und der Template-Parameter `TypeOut` definiert die Ausgabe-Queue einer Station. Als Datencontainer kann, wie eingangs schon erwähnt, unter den Folgenden drei gewählt werden: Shared Queue (siehe Kapitel 4.2 Seite 22), `list_ts` (siehe Kapitel 4.1.1 Seite 18) und `deque_ts` (siehe Kapitel 4.1.2 Seite 19).

Der *`NULL`*-Pointer kann bei der typsicheren Pipeline nicht als Poison-Element verwendet werden, da *`NULL`* nur bei Pointern definiert ist. Stattdessen muss der Nutzer selbst ein Element als Poison-Element definieren und es beim Erzeugen einer Station im Konstruktor an diese übergeben. In Abbildung 4.4.8 ist in Kombination mit dem Poison-Element der ominöse Datentyp `Type` zu sehen. Der als `Type` bezeichnete Typ des Poison-Elements stellt eine abkürzende Schreibweise für **`typename`** `type_traits::get_type<` **`TypeOut`** `>::value_type` dar (Exemplarisch für den Template-Parameter **`TypeOut`**>). Dieses Konstrukt bestimmt mithilfe des in Listing 4.4.12 gezeigten Type-Traits automatisch den Typ des Poison-Elements. Dadurch wird dem Benutzer erspart, den Typ des Poison-Elements als zusätzlichen Template-Parameter mit anzugeben, und somit auch eine mögliche Fehlerquelle ausgeschlossen.

```
01 namespace type_traits {
02    template< typename T >
03    struct get_type {
04       typedef typename T::value_type value_type;
05    };
06    template< typename T >
07    struct get_type<T*> {
08       typedef typename get_type<T>::value_type value_type;
09    };
10 }
```

Liefert den Typ eines Datencontainers auf dem Stack.

Liefert den Typ eines Datencontainers auf dem Heap.

Listing 4.4.12: Type-Trait zur automatischen Bestimmung des Typs eines Datencontainers.

Lief die Konstruktion der nicht-typsicheren Pipeline nahezu voll-automatisiert ab, bieten sich bei der typsicheren Pipeline großzügigere Konfigurationsmöglichkeiten, angefangen vom Stationstyp, über die Wahl des Datencontainers bis hin zur Verknüpfung der einzelnen Stationen. Diese Konfigurationsmöglichkeiten liefern mehr Freiheit, allerdings auch mehr Fehlermöglichkeiten.

Folgendes Beispiel zeigt die Konstruktion einer linearen typsicheren Pipeline, die drei Stationen besitzt, die durch den Datencontainer `deque_ts` vom Typ `int` miteinander verknüpft wurden:

```
01 // Pipeline mit einer festen Schrittanzahl erzeugen.
02 pipeline< false > pl(STEPS_TO_RUN);
03
04 task0_source     task0;    // Task an Station 0
05 task1_in_and_out task1;    // Task an Station 1
06 task2_drain      task2;    // Task an Station 2
07
08 // Die Shared Queues, die die einzelnen Stationen miteinander verbinden.
09 deque_ts<int> q1;          // Ausgabe 1 -> Eingabe 2
10 deque_ts<int> q2;          // Ausgabe 2 -> Eingabe 3
11
12 pl.add_stage(
13    new pipe_stage_source<
14       deque_ts<int>,                task0_source >         (&q1, task0));
15 pl.add_stage(
16    new pipe_stage<
17       deque_ts<int>, deque_ts<int>, task1_in_and_out> (&q1, &q2, task1));
18 pl.add_stage(
19    new pipe_stage_drain<
20       deque_ts<int>,                task2_drain >     (&q2,      task2));
21
22 pl.start();
```

Listing 4.4.13: Konstruktion einer linearen typsicheren Pipeline..

Stationen werden durch die Methode `add_stage(…)` hinzugefügt. Die `add_stage(…)`-Methode der typsicheren Pipeline unterscheidet sich nicht sonderlich von der bereits auf Seite 45 in Listing 4.4.3 vorgestellten gleichnamigen Methode der nicht-typsicheren Pipeline. Jedoch ist ihre Implementierung viel einfacher, da die in Listing 4.4.3 vorgenommene, automatische Verknüpfung mit der Vorgänger-Station entfällt. Dieser Verknüpfungsvorgang muss, wie bereits in Listing 4.4.13 zu sehen ist, nun „per Hand“ vorgenommen werden. Der einzige Nutzen der `add_stage()`-Methode liegt darin, die hinzugefügte Station der Pipeline bekannt zu machen und sie in einem Vektor zu speichern (vergleiche Listing 4.4.13 Zeile 27). Ansonsten werden noch ein paar Initialisierungsarbeiten getätigt, die sich allerdings mit jenen in Listing 4.4.13 decken (Zeile 17, 23 und 34).

Auf die `start()`-Methode der Klasse `pipeline` soll hier nicht im Detail eingegangen werden, da sie sich zu 100% mit der in Kapitel 4.4.1 vorgestellten `start()`-Methode der nicht-typsicheren Pipeline auf Seite 46 deckt.

Damit die Implementierung der `step()`-Methode keine überladene Code-Bombe ist, wird im Folgenden auf die verwendeten Ideen und Prinzipien eingegangen, anstatt den Quelltext aufzuführen. Interessierte Leser können ihre Implementierung im Anhang dieser Arbeit finden.

Im Prinzip funktioniert die `step()`-Methode der Klassen `pipe_stage_source`, `pipe_stage_drain`, `pipe_stage`, `pipe_stage_router` und `pipe_stage_collector` ähnlich der in Listing 4.4.6 auf Seite 48 gezeigten `step()`-Methode. Der augenscheinlichste Unterschied zu ihrem nicht-typsicheren Äquivalent besteht darin, dass sich die `step()`-Methode auf die eben erwähnten Klassen verteilt, anstatt durch einen Anweisungsblock zu entscheiden, welche Typ-Ausprägung die aktuelle Station besitzt. So entspricht z. B. die `step()`-Methode der Klasse `pipe_stage` dem auf Seite 48 gezeigten ‚2. Abschnitt' des dort angeführtem Listings.

Die Kernfunktionalität der `step()`-Methode ist es, einen Task auf Daten auszuführen. Dies funktioniert immer nach dem gleichen Prinzip: Sie holt Daten aus der Eingabe-Queue, führt den Task auf diesen Daten aus und reicht sie an die Ausgabe-Queue weiter:

```
out_->push_back(task_(in_->front()));
```

Entfällt bei Drain-Stationen. | Entfällt bei Source-Stationen.

Je nach Ausprägung einer Station kann ein Schritt wegfallen, da Stationen vom Typ Source beispielsweise keine Eingabe-Daten besitzen.

Doch bevor der Task auf den Datenstrom angewendet wird, muss, sofern die Pipeline-Station mit einem Poison-Element initialisiert wurde, geprüft werden, ob sich das Poison-Element in der Eingabe-Queue befindet (außer bei Stationen vom Typ Source, da diese keine Eingabe-Queue besitzen). Dies erledigt die in Abbildung 4.4.8 auf Seite 55 von jedem Stationstyp implementierte Methode `done()`. Diese Methode vergleicht das aktuelle Element der Eingabe-Queue (`in_->front()`) mit dem bei der Konstruktion der Station übergebenen und in der Variable `poison_` gespeicherten Element. Sie liefert **`true`** zurück, wenn sich beide Elemente gleichen, was zur Folge hat, dass diese Station von der Pipeline beendet wird, ansonsten wird **`false`** zurückgeliefert und der Task auf dem Datenelement ausgeführt.

Alternativ kann die Arbeit in einer Stationen auch beendet werden, wenn der Konstruktor der Pipeline mit einer festen Schrittzahl aufgerufen wird (`step_ = true` in Abbildung 4.4.8). Nachdem der Task auf dem aktuellen Datenelement ausgeführt wurde, wird eine Zählvariable hochgezählt, die angibt, wie viele Arbeitsschritte bereits in der aktuellen Station bearbeitet wurden (`steps_worked_`). Im nächsten Schritt wird dann geprüft, ob die aktuelle Schrittzahl der vorher festgelegten Schrittzahl (`steps_to_work_`) entspricht. Ist dies der Fall, wird die Station beendet, indem sie **`true`** an die Pipeline zurückliefert.

Für alle Stationen gilt, dass sie von der Pipeline beendet wird, sobald sie an irgendeiner Stelle **`true`** zurückliefert. Die Stationen liefern dagegen immer **`false`** zurück, wenn sie noch nicht beendet werden sollen oder wenn sich keine Arbeit in der Eingabe-Queue befindet.

IMPLEMENTIERUNG DES SCHEDULERS UND DER STATISTIKEN:

Die in der AthenaMP-Bibliothek befindliche Implementierung der `start()`-Methode besitzt für beide Pipeline-Typen die Möglichkeit, Stationen „schlafen zu legen“, einen Scheduler zu nutzen und Statistiken zu sammeln.

- *Die Möglichkeit Stationen „schlafen zu legen“:*

  Die Klasse `pipe_stage_auto` der nicht-typsicheren Pipeline und die Klasse `pipe_stage_interface` der typsicheren Pipeline besitzen zwei Variablen namens `stage_threshold_` und `stage_threshold_waittime_in_secs`. Sie ermöglichen den Vorgänger einer Station, diese für eine bestimmte Zeit „schlafen zu legen“, sofern diese mehr Daten produziert, als die aktuelle Station verarbeiten kann. Dies ist der Fall, wenn die Anzahl der von einem Task zu bearbeitenden Daten in der Eingabe-Queue den von `stage_threshold_` angegeben Schwellwert übersteigt. Die beiden Parameter lassen sich für jede Station beim Aufruf von `add_stage()` durch optionale Parameter individuell setzen. Wird der optionale Parameter nicht angegeben, nehmen sie vordefinierte Default-Werte an.

  Um die Vorgänger-Station „schlafen zu legen“, wird über die Methode `stage_pause()` der Status Pipeline von ‚***work***‘ auf ‚***sleep***‘ gesetzt (siehe Listing 4.4.4 Seite 45):

  ```
  stage_status_[Id_der_Vorgänger_Station].thread_what_to_do_= sleep;
  ```

  Ist der Status einer Station auf ‚***sleep***‘ gesetzt, wird statt der `step()`-Methode einer konkreten Station per Busy-Waiting für die in `stage_threshold_waittime_in_secs` angegebene Zeit „schlafen gelegt“ und anschließend der Status wieder auf ‚***work***‘ zurückgesetzt.

- *Der Scheduler:*

  Beide Pipeline-Implementierungen besitzen einen internen Scheduler. Der Scheduler ist nötig, wenn eine Pipeline mehr Stationen besitzt, als Threads vom Laufzeitsystem zur Verfügung gestellt werden können. Im optimalen Fall existiert für jede Station der Pipeline ein Thread, der diese bearbeitet. Existieren mehr Threads als Stationen, so bleiben diese ungenutzt. Für den Sonderfall, dass weniger Threads als Stationen vorhanden sind, wird ein interner Scheduler nach dem Round-Robin Verfahren bemüht, der die einzelnen Stationen abwechselnd von den vorhandenen Threads ausführt. Übersteigt die Anzahl der Datenelemente in der Eingabe-Queue einen zuvor definierten Schwellwert, wird der aktuellen Station der Thread entzogen und einer zuvor ‚threadlosen‘ Station zugewiesen. Der Scheduler soll hier nicht im Detail betrachtet werden, da sich seine Funktionsweise darauf beschränkt, die ID einer ‚threadlosen‘ Station zurückzuliefern, also von einer Station, die momentan von keinem Thread bearbeitet wird und noch nicht beendet ist.

- *Die Statistiken:*

  Beide Pipline-Klassen besitzen den Template-Parameter **`HasStats`**; er definiert über einen Boolean-Wert das Laufzeitverhalten und gibt an ob bei dem aktuellen Durchlauf statistische Daten über die Auslastung der einzelnen Stationen gesammelt werden sollen (Policy-Based Class Design

nach Alexandrescu [Alexandrescu01]; vergleiche auch das spezialisierte Template `size_wrapper` der Shared Queue auf Seite 25).

Beide Pipelines besitzen jeweils eine Methode namens `get_statistics()`, die einen Vektor mit einer Struktur namens `pipe_stats` für jede Station der Pipeline zurückliefert. In dieser werden folgende Werte für jede Station protokolliert:

- `total_work_count_` speichert, wie oft ein Task in einer Station ausgeführt wurde.
- `total_step_count_` speichert, wie oft die `step()`-Methode einer Station aufgerufen wurde: dieser Wert ist unabhängig davon, ob ein Task auf dem aktuellen Datenelement ausgeführt wurde oder nicht.
- `total_sleep_times_` speichert, wie oft eine Station „schlafen gelegt" wurde.
- `max_in_queue_size_` speichert die Anzahl der Elemente, die sich maximal in der Eingabe-Queue einer Station befanden.

	Station 0	Station 1	Station 2	Station 3
Durchläufe mit Arbeit:	5.000	5.000	5.000	5.000
Durchläufe ohne Arbeit:	0	1	5.075.970	7.240.110
Gesamtzahl an Durchläufen:	5.000	5.001	5.080.970	7.245.110
Durchschnittliche Auslastung:	100%	99%	0%	0%
„Schlafen gelegt":	0	0	0	0
max. Elemente in Eingabe-Queue:	5.000	5.000	1.686	3.314

Tabelle 4.4.1 zeigt eine mögliche Ausgabe der gesammelten Statistiken einer 4stufigen Pipeline.

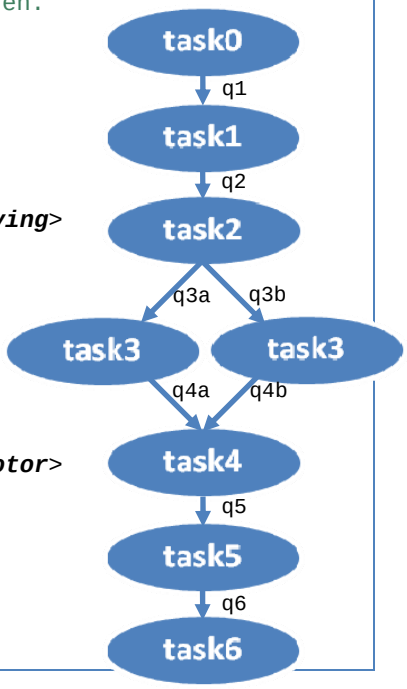
Station 2 und 3 der in Tabelle 4.4.1 gezeigten Statistiken verdeutlichen mit dem Punkt „Durchläufe ohne Arbeit" sehr schön die als „*Filling the Pipeline*" bezeichnete Zeit, bis alle Station der Pipeline gefüllt sind (siehe Abbildung 4.4.4).

ANWENDUNGSBEISPIEL (PIPELINE TYPESAFE):

Als Anwendungsbeispiel wird wieder das bereits in Kapitel 4.4.1 auf Seite 50 vorgestellte Automobil-Beispiel herangezogen. Es soll die fünf Stationen angeben, die sich jeweils um eine Eigenschaft des Automobils kümmern. Als Besonderheit wird eine nicht lineare Pipeline konstruiert (Listing 4.4.14).

Es wird willkürlich davon ausgegangen, dass die Station, die die Türen montiert, unverhältnismäßig viel Zeit benötigt. Deshalb wird der Task, der die Türen an das Auto schraubt, in zwei Stationen eingefügt, die von einer Router-Station (Zeile 34) abwechselnd ihre Arbeit zugewiesen bekommen (siehe Zeile 37 und 40). Anschließend werden die Daten der beiden Stationen wieder durch einen Collector zusammengefügt (Zeile 43).

```
01  pipeline pl;  //Erzeugt eine neue Pipeline auf dem Stack.
02
03  poison_car *p_car = new poison_car; //Das Poison-Element!
04
05  // Der Task, der in einer Station ausgeführt wird.
06  // Jeder Task wird durch einen Funktor repräsentiert.
07  task0_only_out     task0(p_car); // Task an Station 0
08  task1_id           task1;        // Task an Station 1
09  task2_spraying     task2;        // Task an Station 2
10  task3_doors        task3;        // Task an Station 3
11  task4_motor        task4;        // Task an Station 4
12  task5_extras       task5;        // Task an Station 5
13  task6_only_in      task6;        // Task an Station 6
14
15  // Die Shared Queues, die die einzelnen Stationen miteinander verbinden.
16  shared_queue<car*> q1;  // 1  out
17  shared_queue<car*> q2;  // 1  out -> 2  in
18  shared_queue<car*> q3a; // 2  out -> 3a in
19  shared_queue<car*> q3b; // 2  out -> 3b in
20  shared_queue<car*> q4a; // 3a out -> 4a in
21  shared_queue<car*> q4b; // 3b out -> 4b in
23  shared_queue<car*> q5;  // 4  out -> 5  in
24  shared_queue<car*> q6;  //           5  in
25
26  // Die Pipe-Stationen bei der Pipeline registrieren.
27  pl.add_stage(
28      new pipe_stage_source<car*, task0_only_out>
29          (&q1, func0, p_car));
30  pl.add_stage(
31      new pipe_stage<car*, car*, task1_id>
32          (&q1, &q2, task1, p_car));
33  pl.add_stage(
34      new pipe_stage_router<car*, car*, task2_spraying>
35          (&q2, &q3a, &q3b, task2, p_car));
36  pl.add_stage(
37      new pipe_stage<car*, car*, task3_doors>
38          (&q3a, &q4a, task3, p_car));
39  pl.add_stage(
40      new pipe_stage<car*, car*, task3_doors>
41          (&q3b, &q4b, task3, p_car));
42  pl.add_stage(
43      new pipe_stage_collector<car*, car*, task4_motor>
44          (&q4a, &q4b, &q5, task4, p_car));
45  pl.add_stage(
46      new pipe_stage<car*, car*, task5_extras>
47          (&q5, &q6, task5, p_car));
48  pl.add_stage(
49      new pipe_stage_drain<car*, task6_only_in>
50          (&q6, task6, p_car));
```



Listing 4.4.14: Zeigt die Konstruktion einer nicht linearen typsicheren Pipeline.

Die Skizze der Pipeline neben dem Listing 4.4.14 soll den Aufbau der Pipeline vor Augen führen. Das Listing zeigt, dass man bei der Verknüpfung der Stationen unbedingt darauf achten muss, dass die richtigen Stationen miteinander verknüpft werden, da sonst der Datenfluss durch die Pipeline nicht mehr gewährleistet ist.

LAUFZEITMESSUNGEN:

Für beide Pipelines existieren folgende vier Test-Szenarios:

- *BASIC POISON:*
  Konstruktion einer linearen Pipeline mit 4 Stationen (Source, P/C, P/C, Drain). Es werden 5 Millionen Integer-Elemente in Station 0 erzeugt und durch die Pipeline geleitet. Jede Station hat den Task bekommen, das aktuelle Datenelement um eins zu erhöhen. Eine Station wird beendet, sobald sie einen vorher definierten Integer-Wert als Poison-Element erhält (Wird von Station 0 nach 5 Millionen Elementen produziert).
- *BASIC STEP:*
  Konstruktion einer linearen Pipeline mit 4 Stationen (Source, P/C, P/C, Drain). Es werden 5 Millionen Integer-Elemente in Station 0 erzeugt und durch die Pipeline geleitet. Jede Station hat den Task bekommen, das aktuelle Datenelement um eins zu erhöhen. Eine Station wird beendet, sobald sie eine zuvor definierte Schrittanzahl von 5 Millionen Arbeitsschritten erreicht hat.
- *COMPLEX POISON:*
  Entspricht dem Beispiel der Automobil-Fabrik aus Abbildung 4.4.2 auf Seite 40. Konstruktion einer linearen Pipeline mit 5 Stationen (alle vom Typ P/C). Es werden 5 Millionen Elemente der in Listing 4.4.8 auf Seite 51 gezeigten Datenstruktur erstellt und durch die Pipeline geleitet. Die Tasks einer Station entsprechen denen der auf Seite 40 in Abbildung 4.4.2 gezeigten Tasks. Eine Station wird beendet, sobald sie eine vorher definierte Datenstruktur vom Typ `car` als Poison-Element erhält. (Wird nach 5 Millionen Elementen in die Eingabe-Queue gepackt).
- *COMPLEX STEP:*
  Entspricht dem Beispiel der Automobil-Fabrik aus Abbildung 4.4.2 auf Seite 40. Konstruktion einer linearen Pipeline mit 5 Stationen (alle vom Typ P/C). Es werden 5 Millionen Elemente der in Listing 4.4.8 auf Seite 51 gezeigten Datenstruktur erstellt und durch die Pipeline geleitet. Die Tasks einer Station entsprechen denen der auf Seite 40 in Abbildung 4.4.2 gezeigten Tasks. Eine Station wird beendet, sobald sie eine zuvor definierte Schrittanzahl von 5 Millionen Arbeitsschritten erreicht hat.

Nicht-typsichere Pipeline:

Für die nicht-typsichere Pipeline ergeben sich für jedes Test-Szenario aufgrund ihrer Template-Parameter vier Test-Konfigurationen. Tabelle 4.4.2 zeigt alle möglichen Kombinationen untereinander. Die `true`- und `false`-Werte in Klammern hinter dem Datencontainer geben an, mit welchen Einstellungen die Pipeline konfiguriert wurde: `(bool HasConstSizeRunTime, bool HasStats)`. Die Shared Queue, die genutzt wird, um einzelne Stationen miteinander zu verknüpfen, kann mit linearer oder konstanter Laufzeit konfiguriert werden (`HasConstSizeRunTime`) und die Pipeline selbst besitzt einen Template-Parameter, der festlegt, ob Statistiken (`HasStats`) gesammelt werden sollen (vergleiche Ta-

belle 4.4.1 auf Seite 60). Tabelle 4.4.2: Die Laufzeiten für 5 Millionen Datenelemente, die durch eine einfache nicht-typsichere Pipeline geleitet werden, gemessen in Sekunden.

	Basic Poison	Basic Step	Complex Poison	Complex Step
shared_queue (false,false)	6,56	6,75	20,81	20,72
shared_queue (false,true)	7,04	6,80	20,19	20,80
shared_queue (true,false)	11,56	10,69	21,53	21,16
shared_queue (true,true)	11,57	10,90	22,75	21,24

Tabelle 4.4.2: Die Laufzeiten für 5 Millionen Datenelemente, die durch eine einfache nicht-typsichere Pipeline geleitet werden, gemessen in Sekunden.

Typsichere Pipeline:

Bei der typsicheren Pipeline ergeben sich mehr Testfälle, da sie neben der Shared Queue noch mit den Datencontainern `list_ts` und `deque_ts` betrieben werden kann. Tabelle 4.4.3 zeigt alle möglichen Kombinationen untereinander. Wird die typsichere Pipeline mit der Shared Queue betrieben, ergeben sich analog zu der Laufzeitmessung der nicht-typsicheren Pipeline wieder vier Test-Konfigurationen. Der erste Wert in Klammern konfiguriert wieder den Template-Parameter **`HasConstSizeRunTime`** und der zweite Wert wieder **`HasStats`** (vergleiche Tabelle 4.4.1 auf Seite 60).

Die Datencontainer `list_ts` und `deque_ts` besitzen keinen Konfigurationsparameter, deshalb gibt der Wert in Klammern nur an, ob Statistiken gesammelt werden sollen (**`bool HasStats`**). Tabelle 4.4.2: Die Laufzeiten für 5 Millionen Datenelemente, die durch eine einfache nicht-typsichere Pipeline geleitet werden, gemessen in Sekunden.

	Basic Poison	Basic Step	Complex Poison	Complex Step
shared_queue (false,false)	5,67	4,96	20,26	19,88
shared_queue (false,true)	6,90	5,35	21,02	19,33
shared_queue (true,false)	7,52	5,72	21,80	21,65
shared_queue (true,true)	8,73	6,79	22,51	21,24
list_ts (false)	9,77	8,99	43,17	37,69
list_ts (true)	10,20	9,60	43,53	156,84
deque_ts (false)	5,28	5,54	31,23	32,07
deque_ts (true)	5,78	5,11	29,67	26,24

Tabelle 4.4.3: Die Laufzeiten für 5 Millionen Datenelemente, die durch eine einfache typsichere Pipeline geleitet werden, gemessen in Sekunden.

Vergleicht man die Test-Ergebnisse beider Pipelines miteinander, so fällt auf, dass keine signifikant schneller oder langsamer als die jeweils andere ist. Die Geschwindigkeit wird einzig von dem Stationsverknüpfenden Datencontainer bestimmt. Überraschend ist, dass die `deque_ts` trotz exzessiven Gebrauchs von Locks der Shared Queue nahezu ebenbürtig ist, sie sogar bei dem Nutzen von skalaren Typen schlägt (vergleiche Test „Basic Poison“). Dies liegt darin begründet, dass die Shared Queue ihre Daten intern in den einzelnen Nodes speichert, die als komplexe Datentypen implementiert sind.

VOR- UND NACHTEILE/NUTZEN:

Der Durchsatz einer Pipeline ist die Zeit, die Daten brauchen, um die Pipeline komplett zu durchlaufen. Er wird zu einem von der Anzahl der Stationen bestimmt. Diese bestimmt die Anzahl der Threads im parallelen Abschnitt. Für den Sonderfall, dass die Laufzeitumgebung weniger Threads als vorhandene Stationen zurückliefert, greift der Scheduler, es steigt der Verwaltungsaufwand und es sinkt der Durchsatz. Zum anderen wird der Durchsatz auch vom schwächsten Glied in der Kette bzw. in unserem Fall von der schwächsten Station in der Pipeline bestimmt. Dies gilt ebenso für eine Pipeline, deren Stationen im Parallelen wie im Sequenziellen ausgeführt werden. Die langsamste Station wird immer einen Flaschenhals darstellen.

Bei der typsicheren Pipeline bot sich mit den Klassen `pipe_stage_router` und `pipe_stage_collector` die Möglichkeit, die lineare Struktur aufzubrechen, um so drohende Flaschenhälse von vornherein zu kompensieren (Abbildung 4.4.9). Dieses Vorgehen ist bei der nicht-typsicheren Pipeline leider nicht möglich, da diese bisher keine Funktionalität für Router und Kollektoren anbietet. Jedoch ist es auch möglich, diese „Nicht-Linearität“ bei der nicht-typsicheren Pipeline zu erzeugen, indem man die Stationen einfach linear anordnet (Abbildung 4.4.10) und in der `step()`-Methode der Station angibt, wie Daten behandelt werden sollen oder ob ein Datenpaket einfach weitergereicht werden soll.

Abbildung 4.4.9: Nicht-lineare Pipeline.

Abbildung 4.4.10: Nicht-lineare Pipeline linear angeordnet.

Das in Abbildung 4.4.10 gezeigte Vorgehen ist für gewöhnlich dem Vorgehen aus Abbildung 4.4.9 vorzuziehen. Werden Router und Kollektoren genutzt, splittet sich der Datenstrom immer weiter auf und man bekommt, neue parallele Unter-Regionen in bereits parallelen Bereichen. Diese Regionen sind schwer zu kontrollieren und noch schwerer zu debuggen. Man hat keine wirkliche Kontrolle und Übersicht mehr darüber, welchen Weg die Daten durch die Pipeline nehmen. Man verliert teuer erkauften Determinismus und Daten können am Ende der Pipeline somit in einer anderen Reihenfolge als ihrer ursprünglichen vorliegen.

Existiert eine Station, die sich als Flaschenhals herausstellt, muss der Datenstrom einer Pipeline nicht unbedingt gesplittet werden, um ihn zu lösen. Es gibt viele andere Möglichkeiten, die Bearbeitungszeit der Daten einer Station zu verkürzen, indem man die Station selbst parallelisiert oder in ihr einen Task-Pool [Wirz06], wie z. B. den von der AthenaMP-Bibliothek zur Verfügung gestellten Task-Pool verwendet.

Die nicht-typsichere Pipeline bietet gegenüber der typsicheren Pipeline den Vorteil, dass sie relativ leicht zu benutzen ist. Sie hat nach außen hin nur einen Stationstyp (`pipe_stage_auto`), der sich je nach Position innerhalb der Pipeline automatisch an die erforderlichen Gegebenheiten anpasst. Bei der typsicheren Pipeline kann das „Sammelsurium“ an Stationen gerade neue Benutzer dieses Patterns mit der Fülle der Möglichkeiten nahezu erschlagen.

Beim Vergleich der typsicheren Pipeline zur nicht-typsicheren fällt auf, dass ihr Arbeitsprinzip das gleiche ist, dass sie sich jedoch von der Implementierung her unterscheiden und dadurch auch leichte Unterschiede in der Benutzung auftreten.

Der Vorteil der typsicheren Pipeline gegenüber der nicht-typsicheren Pipeline, die namensgebende Typsicherheit, wird allein schon aus dem Grund hinfällig, dass die Verknüpfung der einzelnen Stationen per Hand zu erfolgen hat. Sind Stationen falsch miteinander verknüpft, hat dies zur Folge, dass Daten in einer nicht gewünschten und somit auch nicht korrekten Reihenfolge durch die Pipeline fließen oder im schlimmsten Fall nicht durch alle Stationen geleitet werden. Dies kann zur Folge haben, dass die Pipeline zu früh oder gar nicht terminiert. Die nicht vorhandene Typsicherheit der nicht typsicheren Pipeline straft Nutzer, sofern ein anderer Datentyp als erwartet bearbeitet wird, mit einer Runtime-Exception oder undefiniertem Laufzeitverhalten (vgl. Seite 43). Letztendlich ist es dem Anwender überlassen, mit welcher Fehlerart er besser zurechtkommt.

Die typsichere Pipeline bietet dem Nutzer mehr Konfigurationsmöglichkeiten über den Typ einer Station, da genau festgelegt werden kann, welcher Stationstyp an welcher Position einer Pipeline stehen soll. Diese Kontrolle gibt man bei der nicht typsicheren Pipeline zugunsten einer leichteren Benutzung auf. Die typsichere Pipeline besitzt aber den Vorteil, dass sie einfacher als die nicht-typsichere, um neue Stationstypen ergänzt werden kann. Dazu müssen neue potenzielle Stationstypen einfach von der Klasse `pipe_stage_interface` abgeleitet und um die gewünschte Funktionalität erweitert werden.

Klassen der typsicheren Pipeline und der nicht-typsicheren Pipeline sind nicht kompatibel und können somit auch nicht kombiniert werden.

Abschließend wird ein Vergleich zwischen den zwei Pipeline-Pattern aufgeführt. Es wurde bewusst darauf verzichtet, bestimmte Punkte als Vor- oder Nachteil zu kennzeichnen, da sich die einzelnen Vor- bzw. Nachteile, oftmals erst aus dem verwendeten Kontext ergeben.

nicht typsichere Pipeline:	typsichere Pipeline:
• keine Typsicherheit	• Typsicherheit
• nur eine Stationsklasse: `pipe_stage_auto`	• drei verschiedene Stationsausprägungen `pipe_stage_source`, `pipe_stage`, `pipe_stage_drain`
• nur linearer Datenfluss möglich	• die Möglichkeit nicht lineare Datenflüsse zu gen (mit `pipe_stage_router` und `pipe_stage_collector`)
• automatische Verknüpfung der einzelnen Stationen durch Shared Queues	• die einzelnen Stationen werden nicht automatisch verknüpft. Der Nutzer muss dies selbst vornehmen und darauf achten, dass ein korrekter Datenfluss gewährleistet ist (außerdem ist der verwendete Datencontainer frei wählbar).
• leichter zu bedienen, aber weniger Konfigurations-Freiheit	• schwerer zu bedienen, aber mehr Konfigurations-Freiheit

Tabelle 4.4.4 zeigt die Unterschiede zwischen der nicht-typsicheren und der typsicheren Pipeline.

Beide Pipeline-Varianten bieten eine elegante Möglichkeit, viele Aufgaben parallel zu lösen, ohne sich als Nutzer Gedanken über eine mögliche Parallelisierung machen zu müssen. Ein großer Vorteil liegt darin, dass der Datenfluss beider Pipelines immer deterministisch ist, sofern man auf Router und Kollektoren als Pipeline-Stationen verzichtet.

5 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit sind insgesamt fünf generische Pattern-Implementierungen vorgestellt worden. Drei threadsichere Datencontainer, `list_ts`, `deque_ts` und `vector_ts` wurden in Kapitel 4.1 erläutert. Im Hinblick auf die Datencontainer ist hervorzuheben, dass jeder sein von der STL angebotenes Äquivalent mithilfe des Decorator-Patterns kapselt und die Methodenaufrufe durch Locks schützt.

Die Shared Queue, der in Kapitel 4.2 vorgestellte Datencontainer, kommt dagegen komplett ohne Locks aus. Er funktioniert allerdings nur für genau zwei Threads und ist aufgrund des von OpenMP verwendeten Speichermodells nicht auf allen Plattformen lauffähig. OpenMP garantiert nur nach dem Aufruf einer expliziten Barriere, dass der momentane Inhalt einer Variablen sich mit dem wirklichen Inhalt einer Variablen deckt. Durch diese Inkonsistenz kann es auf manchen Architekturen vorkommen, dass ein Thread immer mit einem veralteten Wert einer Variablen arbeitet und somit Änderungen nie registriert. Würde man eine Barriere einbauen, wäre die Shared Queue nicht mehr lock-frei. Tests ergaben jedoch, dass die Shared Queue auf allen x86-Architekturen problemlos und fehlerfrei ihren Dienst verrichtet.

Als drittes Pattern wird in Kapitel 4.3 auf das durch Gamma *et al.* berühmt gewordene Observer-Pattern eingegangen. Es überwacht bestimmte Objekte auf Zustandsänderungen und benachrichtigt zuvor registrierte Objekte, sofern dieser Fall eintritt. Die Benachrichtigungsmethoden und die damit verbundenen An- und Abmelde-Verfahren des Patterns wurden unter Verwendung von Templates auf drei verschiedene Arten angeboten. Jede Art, Strategie genannt, besitzt dabei ihre ganz spezifischen Vor- und Nachteile. Die Strategie ‚`none`' entspricht dem Pattern-Entwurf Gammas *et al.*; sie kann jedoch zu undefinierten Laufzeitverhalten führen, sofern während eines Benachrichtigungsvorgangs Objekte vom Observer abgemeldet werden. Die Strategie ‚`the_big_lock`' beugt diesem vor, indem alle kritischen Regionen durch Locks geschützt werden. Die dritte Alternative namens ‚`mixed`' nutzt einen von AthenaMP angebotenen Reader-Writer Lock, um die Gefahr einer drohenden Serialisierung der vorherigen Strategie abzudämpfen.

Abschließend werden in Kapitel 4.4 zwei verschiedene Implementierungen einer Pipeline vorgestellt. Es wird zwischen einer nicht-typsicheren (Kapitel 4.4.1) und einer typsicheren Pipeline (Kapitel 4.4.2) unterschieden. Die Grundidee hinter beiden Pipelines, die Umsetzung einer parallelen Fließbandfertigung, ist bei beiden gleich. Bei der typsicheren Pipeline ist der Datentyp einer jeden Station durch Template-Parameter definiert, im Gegensatz zu ihrem nicht-typsicheren Pendant, das mit Void-Pointern (void*) arbeitet.

Diese hier vorgestellten Pattern sollen generische Komponenten darstellen, die sich durch Flexibilität, Wiederverwendbarkeit und leichte Benutzbarkeit auszeichnen. Ziel dieser Arbeit war die Untersuchung des Pattern-Verhaltens unter bestimmten Bedingungen, d. h., wie leicht bzw. wie schwer sie sich mit dem Gespann C++ und OpenMP realisieren lassen.

Eine Schwäche von OpenMP, die während der Umsetzung dieser Pattern deutlich wurde, ist das Fehlen eines Mechanismus', der es erlaubt, einen Thread anzuhalten, bis eine bestimmte Bedingung erfüllt ist. Findet ein Thread einer Pipeline-Station eine leere Eingabe-Queue vor, so muss er warten, bis neue Arbeit eingefügt wurde. Die einzige Lösung ist aktives Warten, d. h., der Thread prüft ständig, ob diese Bedingung erfüllt ist und verschwendet dabei Prozessor-Leistung. Um diesen Missstand zu beheben, bietet sich das Observer-Pattern an. Java-Threads besitzen die Möglichkeit, Threads schlafen zu legen, bis eine bestimmte Bedingung eintritt. Ist dies der Fall, kann dort ein `notify()`-Aufruf abgesetzt werden, der ver-

anlasst, dass der „schlafengelegte“ Thread wieder aufwachen soll (analog zu der `notify_observers()`-Methode des in Kapitel vorgestellten Observer-Patterns) [Goetz06].

Eine andere Schwäche von OpenMP hat sich bei der Implementierung der Shared Queue offenbart und betrifft das zugrunde liegende Speichermodell (vergleiche Seite 28). Es führt explizite Flush-Aufrufe, abgesehen für Boolsche-Variablen, ohne explizite Barrieren, (die jeweils immer ein implizites Flush nach sich ziehen), ad absurdum. Ob das Flush dennoch in der Praxis funktioniert, hängt von der zugrunde liegenden Systemarchitektur ab. Diese Tatsache bedeutet einen empfindlichen Schlag gegen die Portabilität, gerade wenn das System für eine universell einsetzbare Bibliothek entwickelt wird, wie es bei AthenaMP der Fall sein soll.

Die Kombination C++ und OpenMP arbeitet im Grunde sehr gut, jedoch gibt es an einigen Stellen noch Optimierungsbedarf. In OpenMP ist es nicht möglich, Exceptions aus einer parallelen Region zu werfen (vergleiche Seite 34). Das Observer-Pattern ist geradezu prädestiniert, Exceptions zu „schmeißen“, wenn im aktuellen `notify()`-Durchlauf ein Objekt benachrichtigt werden soll, dessen Thread sich aber gerade vom Subject abgemeldet hat und evtl. gar nicht mehr existiert. Gerade in einer Bibliothek wäre es von Vorteil, Exceptions nutzen zu können, damit Fehlersituationen elegant abgefangen werden können.

Die in dieser Arbeit verwendeten Pattern, insbesondere die typsichere Pipeline, machen starken Gebrauch von Templates. Templates werden genutzt, um bei der Pipeline, wie der Name schon vermuten lässt, Typsicherheit zu gewährleisten. Da jeder Stationstyp eine andere Anzahl von Eingabe- bzw. Ausgabe-Queues besitzt, macht sich dieser Faktor auch in den Template-Parametern bemerkbar, über die die Queues definiert werden. In C++ war es mir jedoch nicht möglich, die Verknüpfung der einzelnen Stationen, deren Parameter von Templates bestimmt werden, vollautomatisch zu gestalten. Erstellt und verknüpft man die Stationen einer Pipeline per Hand – ähnlich dem in Listing 4.4.9 auf Seite 51 gezeigten Beispiel der Konstruktion einer Automobilfabrik – so wird für gewöhnlich der Typ des Datencontainers über den Template-Parameter während der Initialisierung der Station in die Klasse hereingereicht. In einer Station könnte sich folgender Code befinden: `deque_ts < TypeIn >`. Ein Template-Parameter namens **`TypeIn`** definiert den Typ des Datencontainers. Um die Konstruktion der Pipeline zu vereinfachen, wäre es von Nutzen, wenn der Template-Parameter **`TypeIn`** abhängig vom Ausgabe-Typ der Vorgänger-Station bestimmt werden könnte. Die Ein- und Ausgabe-Parameter, die in dem Datencontainer verwendet werden sollen, sind durch den zugewiesen Task schon bekannt und müssten nur genutzt werden. Es müsste also eine Anweisung wie folgt möglich sein:

```
deque_ts< Vorgänger_Station.get_Ausgabe_Typ() >
```

oder alternativ:

```
deque_ts < aktuelle_Station.get_Typ_des_Tasks() >.
```

Dadurch würde auch das im Buch „The Pragmatic Programmer“ postulierte DRY-Prinzip (DRY - Don’t Repeat Yourself) [Hunt99, S. 27, S. 29, S. 42], das in C++ leider so oft nicht beachtet wird (als prominentes Beispiel die Trennung von Header und Source-Dateien), Anwendung finden. Der Benutzer wäre so nicht gezwungen, Typen doppelt anzugeben (einmal für den Task und einmal für die Queue). Der positive Nebeneffekt wäre, dass die Stationen selbst, ähnlich der nicht-typsicheren Pipeline, automatisch miteinander verknüpft werden könnten. Es wäre dann auch möglich, den Typ des Datencontainers per Template-Parameter an die Pipeline zu übergeben, ähnlich wie es bei der bisherigen Implementierung der Fall ist:

```
TYP_DES_CONTAINERS < aktuelle_Station.get_Typ_des_Tasks() >.
```

Dies würde eine vollkommen flexible und leicht zu konfigurierende typsichere Pipeline ermöglichen. Da diese Probleme von mir nicht gelöst werden konnten, wurde die nicht-typsichere Pipeline implementiert, die mit ihren Void-Pointern, abgesehen von der nicht vorhandenen Typsicherheit, alle diese Vorteile besitzt.

Jedoch verspricht der für das Jahr 2009 anvisierte neue C++ Standard, namens C++0x, alle angesprochenen Probleme leicht zu lösen. Geplant ist, dass die Anzahl der Template-Argumente variabel gestaltet werden kann. Dies würde bei der typsicheren Pipeline die explizite und künstlich wirkende Unterscheidung zwischen den fünf verschiedenen Stationstypen überflüssig machen. Die typsichere Pipeline könnte auch von den beiden neuen Sprachergänzungen `auto` und `decltype` profitieren. Die Ergänzung `auto` bestimmt automatisch den Typ einer Variablen. Zum einen könnte der in Listing 4.4.12 auf Seite 57 vorgestellte Type-Trait zur Bestimmung eines Typs durch `auto` ersetzt werden und zum anderen ließen sich damit Methoden erstellen, die ähnlich den bei der nicht-typsicheren Pipeline verwendeten Methoden `set_in_for_first_stage(…)` und `set_out_for_last_stage(…)` einfach eine Ein- bzw. eine Ausgabe-Queue mit der Pipeline verknüpfen könnten (siehe Listing 4.4.2. Zeile 05 und 06). Der besondere „Clou“ wäre jedoch die Verwendung des neuen Schlüsselwortes `decltype`. Es könnte genutzt werden, um während der Compile-Zeit den Typ einer Variablen festzustellen. Damit wäre nun folgendes Code-Konstrukt gültig:

```
TYP_DES_CONTAINERS < decltype(aktuelle_Station.get_Typ_des_Tasks()) >.
```

Durch zu erwartende Erweiterungen des neuen C++-Sprachstandards könnten die beiden Pipeline-Varianten zu einer Version zusammengefügt werden, ohne dabei auf irgendwelche Template-Hacks zurückgreifen zu müssen, die den Code für Lernende nur unnötig komplizieren und von der eigentlichen parallelen Problemstellung ablenken würden.

Sprachen wie Java oder C# bieten gegenüber C++ den Vorteil, dass sie dem Benutzer umfassende Bibliotheken standardmäßig mitliefern und auch gerade für parallele Problemstellungen viele Methoden anbieten, die einfach „Out-of-the-Box“ genutzt werden können. C++ bietet diese Möglichkeit nicht. Jedoch ist es möglich, Bibliotheken wie Boost [Karlsson05] oder Loki [Alexandrescu06] zu nutzen, die die Funktionalität von C++ erweitern, aber entweder nur threadsichere Datencontainer (Loki) oder nur rudimentäre Thread-Unterstützung bieten (Boost). Die gebotene Funktionalität beschränkt sich allerdings meist auf irgendwelche ‚Low-Level‘-Funktionen. Komplexe Kontrollstrukturen muss der Nutzer meist selbst beisteuern. Intel hat diese Lücke erkannt und hat mit seiner Threading Building Blocks-Library (TBB) eine kostenpflichtige Bibliothek für parallele ‚Low-‘ als auch ‚High-Level‘-Funktionen auf den Markt gebracht.

Die in dieser Arbeit vorgestellten „Pattern“ sollen nun helfen, eine Bibliothek mit Ansprüchen, ähnlich denen der TBB-Bibliothek speziell auf OpenMP zugeschnitten, zu liefern.

> *„I remember a conversation with Jon Blossom at a Computer Game Developer conference in which he asked me if we using STL. I said, „No, we've pretty much internalized how to do a linked list.“ The very next week I wrote a linked list with a stupid bug. The next day I switched to STL.“*
>
> *(Jamie Fristom talking about „Die by the Sword“. [Rollings04, S. 558])*

Steht ein Programmierer vor einer Problemstellung, vergleichbar mit den in Kapitel 4.4 unter „Motivation/Problem“ aufgeführten Anwendungsgebieten einer Pipeline, so hat er zwei Möglichkeiten. Er implementiert zum einen selbst eine Pipeline und ärgert sich eventuell Tage und Wochen mit nebensächlichen Problemen herum, die ihn von seinem eigentlichen Aufgabenfeld, nämlich dem Lösen einer Problemstellung, abhalten (vergleiche Zitat von Jamie Fristom).

Die zweite Möglichkeit wäre, eine bereits existierende Pipeline aus einer Bibliothek zu verwenden. Diese würde unter Umständen nicht ganz die Performance einer individuell auf die Problemlösung zugeschnittene Pipeline erreichen. Sie würde aber helfen, Entwicklungszeit zu sparen, da wohlgetestete Komponenten verwendet würden. Ein Programmier könnte im Falle der Pipeline auf eine vollständig parallelisierte Komponente zurückgreifen, ohne eine Zeile parallelen Code schreiben zu müssen. Viele der in der Einleitung angesprochenen Herausforderungen paralleler Programmierung würden unmittelbar ihren „Schrecken“ verlieren. Viele Nachteile wären behoben: Es gäbe keine Deadlocks, keine Datenabhängigkeiten, keine Raceconditions usw.

Der hier vorgestellte Anwendungsfall der Pipeline lässt sich auf alle in dieser Arbeit und von AthenaMP zur Verfügung gestellten Pattern übertragen. Das Schreiben paralleler Programme wird unter Verwendung der vorgestellten Pattern erleichtert, da sie als Werkzeuge ohne große Mühen verwendet werden können. Die Aufmerksamkeit des Programmierers kann sich so den eigentlichen Problemstellungen widmen und zudem wird die zur Verfügung stehende parallele Hardware genutzt. Dies erhöht die Produktivität und senkt die Fehlerraten.

Neben den hier vorgestellten fünf Pattern existieren noch viele weitere Pattern, die sich für die Benutzung paralleler Systeme als nützlich erwiesen haben und implementiert werden könnten. Auch die hier vorgestellten Pattern stellen nur eine mögliche Implementierung dar und können gegebenenfalls eine bessere Performance erzielen, wenn sie anders umgesetzt werden.

Es wäre wünschenswert, wenn Bibliotheken wie AthenaMP, TBB oder Cops unter Java in verstärktem Maße dazu beitragen würden, das Interesse der Programmierer, die es bisher vermieden haben, parallele Programme zu schreiben, zu wecken, indem die Benutzung paralleler Hardware erleichtert würde und Benutzern so von Geschwindigkeitsgewinnen gegenüber einer sequenziellen Implementierung profitierten.

I. GLOSSAR

ADT, abstrakter Datentyp:	Als ADT gilt eine Menge von Werten auf denen eine Menge von Operationen arbeitet, vergleichbar einer Klasse.
BSD Lizenz:	Eine Lizenz für freie Software. BSD steht für Berkeley Software Distribution. Die BSD Lizenz erlaubt, Software frei zu kopieren, zu ändern und zu verbreiten.
Core Dump:	Ein „Core Dump“ bezeichnet üblicherweise das Speicherabbild, das zum Zeitpunkt eines Programmabsturzes gemacht wird.
Deadlock:	Das Locken gemeinsam genutzter Ressourcen kann eine Sperre verursachen, wenn zwei Prozesse gegenseitig auf die Freigabe von ihnen gesperrter Ressourcen warten.
Dual-Core:	Dual-Core Prozessoren setzen sich aus zwei physikalischen Prozessoren (Cores) zusammen, die zusammen auf einem Schaltkreis untergebracht sind. Jeder der beiden Cores hat seine eigenen Ressourcen (Register usw.).
Funktor:	Funktoren bezeichnen Objekte, die genauso aufgerufen werden können wie Funktionen, aber trotzdem alle Eigenschaften von Objekten besitzen. In C++ sind Funktoren Referenzen oder Pointer zu Funktionen oder Klassen mit dem durch `operator()` überladenen Funktionsoperator ().
Kante	Eine Kante bezeichnet eine Aggregation zwischen zwei Objekten.
Multi-Core:	Ein Multi-Core-Prozessor setzt sich aus N Hauptprozessoren (Cores) zusammen. Er stellt eine Erweiterung des Dual-Core Prozessors dar (siehe Dual-Core).
NUMA:	Non-Uniform Memory Access (NUMA) ist eine Computer-Speicher-Architektur für Multiprozessorsysteme, bei denen die Zugriffszeiten auf den Speicher vom Ort des Speichers abhängen. Ein Prozessor kann auf seinen eigenen, als lokal zugewiesenen Speicher schneller zugreifen als auf den von anderen Prozessoren im gleichen System verwalteten Speicher (Shared Memory System).
Poison-Pill:	Die Poison-Pill ist ein spezieller Task, der angibt, dass alle Arbeit getan ist. (auch Poison-Task oder Poison-Element).
Race-Condition:	Als Race-Condition werden Programmabläufe bezeichnet, bei denen das Ergebnis von dem zeitlichen Verhalten der teilnehmenden Threads beeinflusst wird.
Round-Robin:	Round-Robin wird zur Lastverteilung verwendet, mit dem Ziel, mehrere Ressourcen möglichst gleichmäßig zu verteilen (*load balancing*).
Runtime-Exception:	Runtime-Exception (Laufzeitfehler) machen sich erst während der Programmausführung bemerkbar. Sie werden nicht vom Compiler erkannt. Laufzeitfehler führen zum Absturz des Programms oder zu unbestimmtem Verhalten durch inkonsistente Daten.
SMP:	Symmetrisches Multiprozessorsystem (SMP). Mehrere Prozessoren besitzen einen gemeinsamen Adressraum.
Speedup:	Speedup bezeichnet den Faktor, die ein paralleler Algorithmus schneller als die perfekte sequenzielle Lösung ist. $Speedup = \frac{Zeit\ Sequenziell}{Zeit\ Parallel}$
STL	Die Standard Template Library (STL) bezeichnet eine in C++ verfasste Bibliothek, deren Schwerpunkt auf generischen Datenstrukturen und Algorithmen liegt.
Task:	Ein Task bezeichnet eine Aufgabe. Hauptsächlich führt ein Thread einen Task aus.
Thread:	Leichtgewichtiger Prozess.
Threadsafe (Threadsicherheit):	Ein Programmabschnitt gilt als threadsicher, wenn er von mehreren Threads parallel durchlaufen werden kann und unabhängig von der Thread-Anzahl korrekte Ergebnisse liefert.
Wasserfallmodell:	Das Wasserfallmodell bezeichnet ein Vorgehensmodell in der Softwareentwicklung, bei dem der Softwareentwicklungsprozess in Phasen organisiert wird. Dabei gehen die Phasenergebnisse wie bei einem Wasserfall immer als bindende Vorgaben für die nächst tiefere Phase ein [Wiki07Wa].

II. ABBILDUNGSVERZEICHNIS

III. TABELLENVERZEICHNIS

IV. LISTINGVERZEICHNIS

V. LITERATURVERZEICHNIS

[Alexander77]	C. ALEXANDER, S. ISHIKAWA, M. SILVERSTEIN, M. JACOBSON, I. FIKSDAHL-KING & S. ANGEL. *A pattern Language.* New York: Oxford University Press, 1977
[Alexandrescu06]	ANDREI ALEXANDRESCU. *Modern C++ Design – Generic Programming and Design Patterns Applied.* Addison-Wesley, 2006
[Athena07]	ATHENAMP. *Statement AthenaMP README. readme.txt* Stand: 2007.
[Butenhof07]	D. BUTENHOF. *Ten Questions with David Butenhof about Parallel Programming and POSIX Threads.* http://www.thinkingparallel.com/2007/04/11/ten-questions-with-david-butenhof-about-parallel-programming-and-posix-threads/#more-102 Abgerufen am 21. Juli 2007.
[Coding07]	JEFF ATWOOD. *Folding: The Death of the General Purpose CPU.* http://www.codinghorror.com/blog/archives/000823.html Abgerufen am 23. Juli 2007.
[Deep07]	DEEP BLUE. http://www.research.ibm.com/deepblue/ Abgerufen am 15. Juni 2007.
[Folding07]	FOLDING@HOME ON THE PS3. http://folding.stanford.edu/FAQ-PS3.html Abgerufen am 5. Juni 2007.
[Goetz06]	B. GOETZ, T. PEIERLS, J. BLOCH, J. BOWBEER, D. HOLMES & D. LEA. *Java Concurrency in Practice.* Addison-Wesley, 2006.
[Grama03]	A. GRAMA, A. GUPTA, G. KARYPIS & V. KUMAR. *Introduction to Parallel Computing, Second Edition.* Pearson, 2003.
[Hunt99]	ANDREW HUNT, DAVID THOMAS. The *Pragmatic Programmer – from journeyman to master.* Addison-Wesley, 1999.
[Karlsson05]	BJÖRN KARLSSON. *Beyond the C++ Standard Library. An Introduction to Boost.* Addison-Wesley, 2005.
[Lea99]	DOUG LEA *Concurrent Programming in Java. Design Principles and Patterns.* Addison-Wesley, 1999.
[Louis03]	D. LOUIS *C/C++ - Die praktische Referenz.* Markt+Technik Verlag, 2003.
[Manthey98]	DIRK MANTHEY *Making of... Wie ein Film entsteht: Making of 2: Band 2.* Rowohlt Tb., 1998.
[Mattson05]	T. MATTSON, B. SANDERS & B. MASINGILL. *Patterns for parallel Programming.* Addison-Wesley, 2005.

[Meyers07]	SCOTT MEYERS. *Effective C++ - Third Edition.* Addison-Wesley, 2007.
[OpenMP]	OPENMP STANDARD 2.5. http://www.openmp.org
[Rauber07]	THOMAS RAUBER & GUNDULA RÜNGER *Parallele Programmierung.* Springer, 2007
[Rollings04]	A. ROLLINGS & D. MORRIS. *Game Architecture and Design – A new Edition* New Riders, 2004.
[Seti07]	SETI@HOME http://setiathome.ssl.berkeley.edu/ Abgerufen am 18. August 2007.
[Stroustrup01]	B. STROUSTRUP. *Die C++-Programmiersprache.* Addison-Wesley, 2001.
[Süß06]	MICHAEL SÜß. *Why I love parallel programming.* http://www.thinkingparallel.com/2006/07/28/why-i-love-parallel-programming/ Abgerufen am 25. Juni 2007.
[Süß06a]	MICHAEL SÜß. *Exceptions in OpenMP and C++ - what's the state of affairs today?* http://www.thinkingparallel.com/2006/10/07/exceptions-in-openmp-and-c-whats-the-state-of-affairs-today/ Abgerufen am 14. August 2007.
[Süß06b]	MICHAEL SÜß. *Making Exceptions Work with OpenMP - Some Tiny Workarounds.* http://www.thinkingparallel.com/2006/11/30/making-exceptions-work-with-openmp-some-tiny-workarounds/ Abgerufen am 14. August 2007.
[Tanenbaum03]	ANDREW TANENBAUM. *Moderne Betriebssysteme.* Pearson, 2003.
[Top07]	TOP500 SUPERCOMPUTER SITES. *BlueGene/L* http://www.top500.org/system/7747 Abgerufen am 15. August 2007.
[Vandevoorde06]	D. VANDEVOORDE & N. JOSUTTIS. *C++ Templates – The complete Guide.* Addison-Wesley, 2007.
[Wiki07Wa]	WASSERFALLMODELL http://de.wikipedia.org/wiki/Wasserfallmodell Abgerufen am 15. August 2007.
[Wilkinson05]	BARRY WILKINSON & MICHAEL ALLEN. *Parallel Programming – Techniques and Applications Using Networked Workstations and Parallel Computers.* Pearson, 2005.