Objektorientierte Modellierung einer Simulationsumgebung mit Patterns

Jan Peter Riegel
Martin Schütze
Gerhard Zimmermann

09/1996

Sonderforschungsbereich 501

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-67653 Kaiserslautern

Inhaltsverzeichnis

Kapitel 1 Einleitung

Ziel dieser Arbeit ist es, eine Methode zur Verfügung zu stellen, mit der ein Simulator für gebäudespezifische Aufgaben modelliert werden kann. Die Modellierung muß dabei so angelegt sein, daß sowohl einfache als auch sehr komplexe Simulatoren für spezielle Gebäude entworfen werden können. Aus dem erstellten Modell ist es anschließend möglich, mit Hilfe von Generatoren automatisch ein Programm zu erzeugen. Dadurch kann ein Entwerfer ohne spezielle Kenntnisse auf dem Gebiet der Simulation einen Gebäude-Simulator entwickeln.
Simuliert werden sollen im Gebäudebereich auftretende Gegebenheiten, also hauptsächlich physikalische Größen wie Raumtemperatur, Luftfeuchtigkeit oder Luftdruck. Diese Größen werden quasi-kontinuierlich[1] berechnet; es können dabei sowohl kontinuierliche Vorgänge (z.B. Änderung der Lufttemperatur) als auch atomare Ereignisse (Tür wird geöffnet) berücksichtigt werden. Der Simulator kann jeweils so konfiguriert bzw. modelliert werden, daß er für ein spezielles Gebäude die momentan interessanten Größen berechnet. Weiterhin kann der Simulator sukzessive verfeinert werden: wird beispielsweise in einem frühen Stadium der Simulation nur eine grobe Abschätzung der Raumtemperatur benötigt, so kann diese später durch genauere Berechnungsformeln verfeinert werden.
Eine weitere Anforderung an den Simulator ist, daß einzelne Teile durch reale Hardware ersetzt werden können (Hardware-In-The-Loop). So können zum Beispiel die Messwerte eines Temperaturfühlers benutzt werden, um die Lufttemperatur zu bestimmen, und die gemessenen Werte können herangezogen werden, um die Luftfeuchtigkeit zu simulieren. Um die Integration von Hardware in den Simulator zu ermöglichen, muß dieser echtzeitfähig sein.

Der SFB-Bericht beschreibt eine Möglichkeit zur Modellierung (und Generierung) einer flexiblen Simulationsumgebung für Gebäude mit deren Hilfe Gebäudesteuerungen ausgetestet werden können (siehe [SFB94]). Die hier vorgestellte Modellierungstechnik geht von einem sehr eingeschränkten Entwurfsbereich (Domäne) für Applikationen aus: es sollen nur Gebäudesimulatoren modelliert werden. Für diese Domäne werden typische Entwurfs- und Programmuster gesucht und in Patterns[2] zusammengefaßt. Mit diesen Patterns kann dann ein Simulator modelliert und anschließend generiert werden.

1. siehe dazu Kapitel 3.1 oder [MaM89].
2. Patterns werden in Kapitel 3 und in [GHJ95] und [Pre95] beschrieben.

Das 2. Kapitel behandelt den Software-Entwurf mit Patterns. Dieser wird zunächst allgemein beschrieben, und anschließend wird gezeigt, wie Patterns zur Modellierung und Generierung eines Gebäude-Simulators benutzt werden können. Im Kapitel 3 wird auf die Simulation und das Simulator-Modell eingegangen. Zusätzlich wird kurz die Steuerung der Simulation besprochen, und ein Pufferungs-Konzept wird vorgestellt, mit dem die während der Simulation anfallenden Integrations- und Interpolationsprobleme gelöst werden können. Der Ausblick in Kapitel 4 schließt den Hauptteil der Arbeit ab.
Im Anhang A wird ein Pattern-Katalog vorgestellt, mit dem ein Gebäude-Simulator modelliert werden kann. Der Katalog enthält speziell auf den Entwurfsbereich „Gebäude-Simulation“ zugeschnittene Patterns. Wird ein Pattern aus diesem Katalog im Text referenziert, so erfolgt dieses unter Angabe des Pattern-Namens (*kursiv* gedruckt) und der Seitennummer, auf der das Pattern beschrieben ist. Dadurch ist das Wiederfinden der Patterns im Katalog einfacher.

Kapitel 2 Modellierung mit Patterns

2.1 Software-Entwurf

Im folgenden wird ein Ansatz zum Software-Entwurf vorgestellt, der sich zur Modellierung von Anwendungen eines speziellen Entwurfsbereiches eignet. Dabei soll aus den aufgestellten Modellen ein Großteil der Applikation automatisch generiert werden. Der Software Entwurfsprozeß ist genauer in [SFB96] erläutert.
In einem ersten Schritt der Software-Erstellung muß die Problembeschreibung sorgfältig analysiert werden, um einen Anforderungskatalog für die zu erstellende Software aufzustellen. Dabei werden die in der „realen Welt“ vorkommenden Begriffe abstrahiert und strukturiert, um einen systematischen Zugang zur Aufgabenstellung zu erhalten. Als Ergebnis der Analyse erhält man ein Basismodell, mit dem Vorgänge und Zustände aus dem Problembereich modelliert werden können. Zusätzlich erhält man zu den einzelnen Anwendungsprojekten Charakteristiken, die das spezielle Projekt näher spezifizieren. Die Charakteristiken sind also Verfeinerungen und Erweiterungen des Basismodells. Sie erlauben, die Aufgabenstellungen des Anwendungs-Projektes adäquat zu beschreiben. Der Anforderungskatalog an die zu entwerfende Software umfaßt das Basismodell mit seinen Charakteristiken und die Beschreibung des ursprünglichen Problems im Umfeld dieser Modelle. Während der Entwurfs-Phase werden nun Teile der Basismodelle und die Charakteristik für eine Applikation zusammengeführt und verfeinert. Als Ergebnis erhält man ein Applikationsmodell, in dem alle applikationsspezifischen Daten und Algorithmen enthalten sind. Aus diesem Applikationsmodell kann dann die endgültige Software möglichst vollständig generiert werden.

Der Software-Entwurfsprozeß ist noch einmal in Abbildung 3 (siehe [SFB96]) dargestellt.

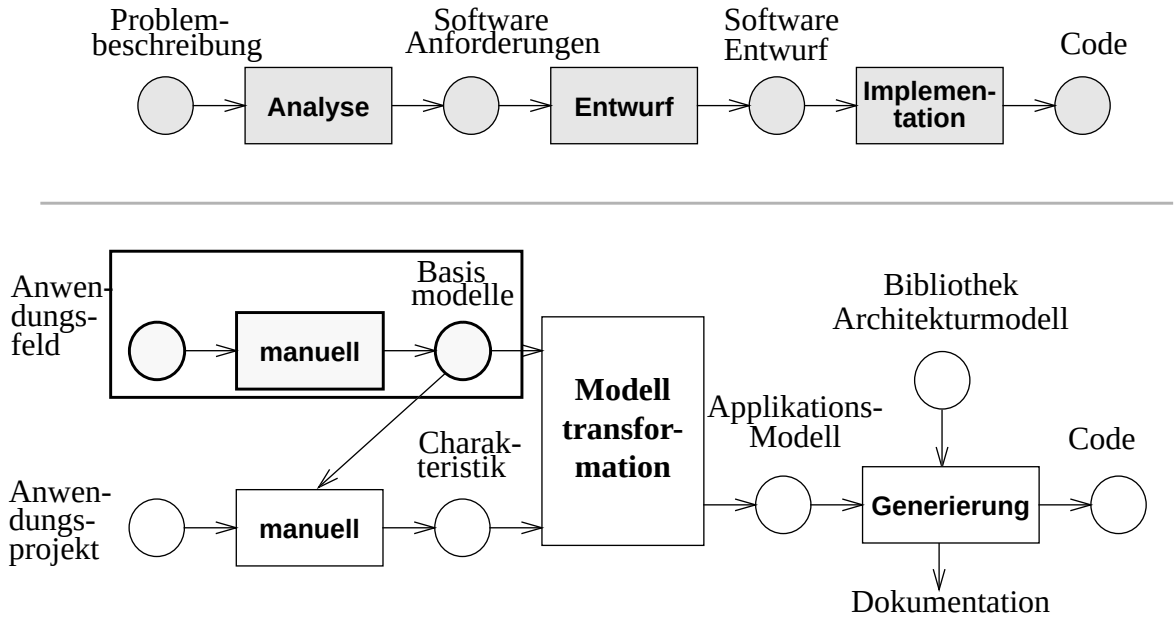


Abb. 3: Allgemeiner Software-Entwurfsprozeß

Als Beispiel sei hier die Programmierung einer Heizungssteuerung erwähnt. Bei der Modellierung der Heizungsanlage treten so allgemeine Begriffe wie „Raum“, „Heizkörper“, „Kessel“ oder auch „Raumtemperatur“ auf. Die hinter diesen Begriffen stehenden Konzepte und Objekte werden in möglichst optimaler Form (das heißt kurz, aussagekräftig und vollständig) in einem geeigneten Basismodell zusammengefaßt. Spezielle Eigenschaften der Steuerung (zum Beispiel Begriffe wie „Kesseltemperatur“, „Ventilsteuerung“ etc.) bilden die Charakteristik der Heizungssteuerung. Wird zusätzlich noch ein weiteres Anwendungs-Projekt geschrieben, zum Beispiel ein Simulator zur Überprüfung der Steuerung, so kann dieses auf demselben Basismodell operieren; in der Charakteristik des Simulators treten dann zusätzlich neue Begriffe wie „Wärmeübergangskoeffizient“ oder ähnliches auf.
Um den Analyseprozeß zu unterstützen, kann auf ein breites Spektrum von Modellierungstechniken und Notationen zurückgegriffen werden (siehe [RBP91], [Boo90]). Bei sorgfältig durchgeführten Problemanalysen können Teilergebnisse und Analyseverfahren aus anderen Projekten übernommen werden („reuse of design“).
Anschließend an die Analysephase erfolgt der Entwurf. Ziel der Entwurfs-Phase ist ein Applikationsmodell, das die Problemlösung beschreibt. In diesem Modell ist also erklärt, wie die einzelnen Teilaspekte gelöst werden können und wie alle Teile des Programmes zusammenspielen. Das Applikationsmodell stellt eine formale Spezifikation der Software dar. Dadurch kann die eigentliche Implementierung des Programmes derart von Generatoren unterstützt werden, daß eine Programmierung „von Hand“ auf ein Mindestmaß reduziert wird. Die Generatoren greifen auf eine Bibliothek von fertigen Software-Bausteinen zurück, die anhand des Applikationsmodelles zum fertigen Programm zusammengestellt werden. Auf diese Art und Weise können, korrekte Bibliotheken vorausgesetzt, Flüchtigkeitsfehler und unsaubere Programmiermethoden vermieden werden.

Ist das Programm fertig erstellt beziehungsweise generiert worden, so muß durch Tests die Konsistenz mit der ursprünglichen Anforderung überprüft werden. Zusätzlich sollte abschließend der gesamte Erstellungsprozeß nach Verfahren und Teilaspekten durchsucht werden, die in einem späteren Entwurf wiederverwendet werden können.

Zur Modellierung von Applikationen gibt es mehrere Beschreibungsmethoden, die sich jeweils für unterschiedliche Gesichtspunkte besonders eignen. In dieser Arbeit werden vor allem erweiterte Entity-Relationship-Diagramme verwendet. Die darin benutzte Notation ist von der Object-Modelling-Technique (OMT, siehe [RBP91]) und der Software-Entwicklungsumgebung MOOSE (siehe [SSA94]) übernommen und wird kurz im folgenden Kapitel beschrieben. Anschließend wird eine über EER-Diagramme hinausgehende Modellierungsmethode mit Patterns vorgestellt.

2.2 Notation der EER-Diagramme

Zentraler Bestandteil der Entity-Relationship Diagramme ist natürlich das „Entity“. Bei der objektorientierten Modellierung kann ein Entity als Objekttyp oder Objektklasse aufgefaßt werden. Ein Objekttyp wird im folgenden als Rechteck aufgeschrieben und die Beziehungen zwischen einzelnen Objekttypen werden durch Linien repräsentiert. Mögliche Relationentypen sind die Generalisierung oder „Is-A Relation“ (repräsentiert durch einen Kreis am Ende der Relation), die Aggregation oder „Part-Of Relation“ (repräsentiert durch einen Pfeil) und eine unspezifische Relation. An den einfachen Relationen stehen Kardinalitäten, um anzuzeigen wieviele Objekte eines Typs mit wievielen Objekten des anderen Typs in Verbindung stehen können.
Zur Gruppierung von Objekttypen können diese im Schemata zusammengefaßt werden und die Schemata selbst können hierarchisch angeordnet werden, um Abhängigkeiten einzelner Objektgruppen zu modellieren. Ein Schema wird durch ein Rechteck mit ausgefüllten Ecken dargestellt.
Die folgende Abbildung zeigt ein einfaches EER-Diagramm in MOOSE-Notation. Dargestellt sind vier Objekttypen „Raum“, „Raumteiler“, „Wand“ und „Fenster“ die zum Schema „Gebäude“ gehören. Ein Raum aggregiert dabei mehrere Raumteiler (d.h. er „besteht“ aus

Raumteilern). Ein Raumteiler kann entweder eine Wand oder ein Fenster sein. Ein Fenster ist genau einer Wand zugeordnet, während in einer Wand mehrere Fenster seien können.

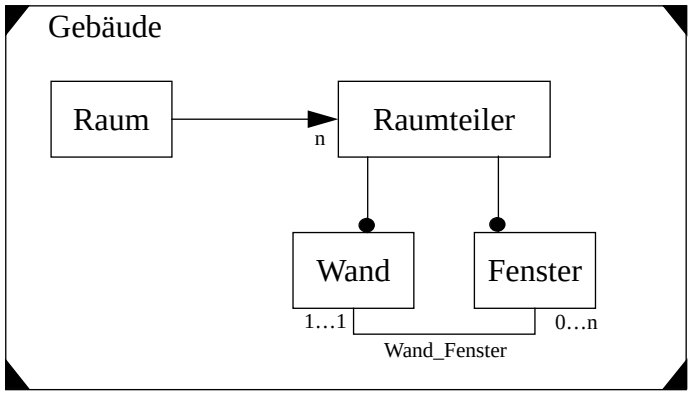


Abb. 4: EER-Diagramm in MOOSE-Notation

Um die Objekttypen weiter zu spezifizieren, können auch noch Attribute und Methoden der Objekte mit in die Notation hinzugenommen werden. Die Namen der Attribute und Methoden werden einfach mit in das Rechteck aufgenommen, das den Objekttyp darstellt. Der Raum in Abbildung 5 hat beispielsweise die Attribute „T“ und „V“ und eine Methode „calculate()“. Kommentare können an beliebigen Stellen eingefügt werden und stehen in grauen Kästchen.

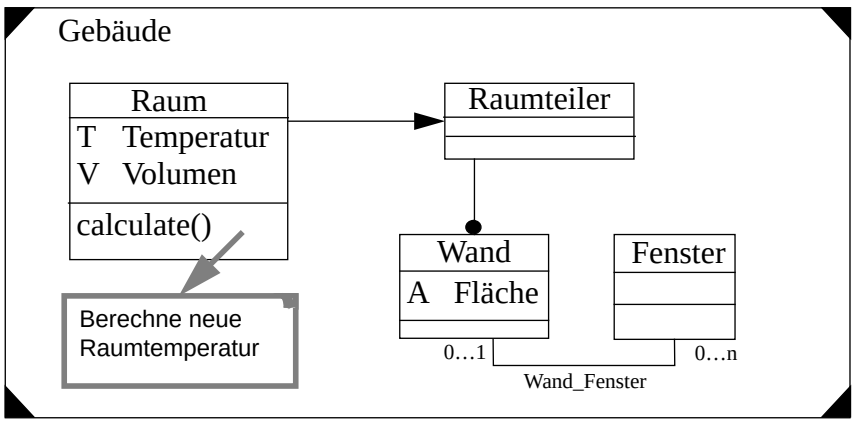


Abb. 5: Erweiterte MOOSE-Notation

2.3 Patterns

Ein Pattern (engl. „Muster“) beschreibt sowohl einen Teil eines Systems, als auch, wie dieser Teil erzeugt werden kann. Software-Patterns beschreiben normalerweise Programmiermuster und Abstraktionen, die von erfahrenen Programmierern in ihrer Software benutzt werden. Patterns kombinieren und vereinigen andere Abstraktionen wie zum Beispiel Objekte und Prozeduren und fügen diese zu einer umfassenden Beschreibung eines kleinen Teils der Software zusammen.

Der Begriff „Pattern“ als Entwurfsmethode wurde erstmals von dem Architekten Christopher Alexander (siehe [AIS77]) verwendet. In den 60er Jahren untersuchten Architekten Möglichkeiten eines automatisierten und computerisierten Gebäude-Designs. Gesucht wurden Regeln und Algorithmen, um Anforderungen in eine Konfiguration von vorgefertigten Gebäudeteilen umzusetzen. Alexander bemerkte, daß mit dieser Methode keine guten Architekturen gebaut werden konnten, sondern daß vielmehr alle Teile eines Gebäudes individuell und von Hand zusammengesetzt werden müssen. Zur Unterstützung dieses kreativen Prozesses stellte er Beschreibungen und Regeln zum Entwurf „guter“ Architektur auf, die er „Patterns“ nannte. Zum Beispiel ist „Fenster auf zwei Seiten jedes Raumes“ eine solche Beschreibung. Sie gibt eine Regel für einen Teil eines guten Entwurfs wieder, sagt jedoch nichts darüber aus, wie groß die Fenster sein sollen oder aus welchem Material der Fensterrahmen besteht. Zusammengefaßt ergeben diese Patterns eine Sprache, auf deren Basis Entwürfe gemacht werden können.
Im Bereich des Software-Engineerings wurde der Begriff „Pattern“ übernommen und bezeichnet dort Regeln und Anleitungen zum guten Software Entwurf.
Patterns können grob in *generative* und *beschreibende* Patterns gruppiert werden. Beschreibende Patterns geben wieder, wie ein bereits bestehendes System realisiert wurde (siehe [GHJ95]). Sie sind rein deskriptiv und passiv. Generative Patterns hingegen geben Regeln und Hinweise an, wie ein System erzeugt werden kann. Sie gehen also über den rein beschreibenden Charakter hinaus und abstrahieren soweit, daß sie an verschiedenen Stellen wiederverwendet werden können. Generative Patterns sind also erklärend und aktiv im dem Sinne, daß sie zur Erzeugung von Systemen benutzt werden können.
Der in dieser Arbeit vorgestellte Modellierungsansatz verwendet ausschließlich generative Software-Patterns und formalisiert diese soweit, daß eine *automatische* Generierung von Software möglich wird.

2.4 Wiederverwendung

Patterns werden beim Software-Engineering eingesetzt, um ein möglichst hohes Maß an Wiederverwendung (reuse) zu erreichen. In frühen Stadien des Software-Engineerings fand Wiederverwendung nur auf der Basis von Prozeduren statt. Diese können von einem Programm in ein anderes kopiert werden und müssen dort entsprechend angepaßt werden. Um diese Art der Wiederverwendung zu erleichtern, werden fertige und meist sehr allgemeine Prozeduren zu Programmbibliotheken zusammengefaßt. Teile solcher Bibliotheken können recht einfach aus Programmen referenziert und benutzt werden. Die eigentliche Wiederverwendung findet aber nur auf Code-Ebene statt.
Eine höhere Ebene der Wiederverwendung wird durch objektorientierte Programmiersprachen möglich. Hier können bei sorgfältiger Planung ganze Objekte in mehreren Programmen verwendet werden. Ein Objekt beinhaltet dabei sowohl Instanzenvariablen (seinen Zustand), als auch Methoden, um diesen Zustand manipulieren zu können. Durch die sinnvolle Zusammenfassung mehrerer Objekte zu einem Framework kann ein hoher Grad an Wiederverwendung sowohl auf Code- als auch auf Design-Ebene erreicht werden. Unter dem Begriff „*Framework*“ wird im folgenden eine Menge von Objekten und Klassen, ihre Funktionalität und ihr Zusammenspiel aufgefaßt (siehe [Che94]). Ein typisches Beispiel für solch ein Framework ist

das Model-View-Controller (MVC) Framework, das zur Modellierung von graphischen Benutzeroberflächen benutzt werden kann (siehe [KrP88]). In dem Modell wird angegeben, wie die einzelnen Teile einer Anwendung (Anzeige, Datenbasis und Eingabe) miteinander zusammenspielen. Zusätzlich zu diesen Entwurfsinformationen sind auch bereits konkrete Code-Informationen im MVC-Framework enthalten.

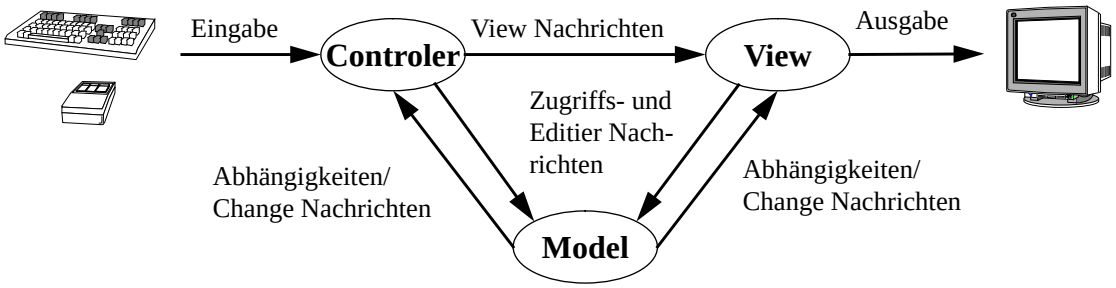


Abb. 6: Model-View-Controller Framework

Ein Entwerfer, der eine Benutzeroberfläche modellieren will und das MVC-Framework kennt, kann durch einfache Mechanismen wie Vererbung oder Delegation die Funktionalität des Frameworks nutzen. Vor allem erhält er durch die vom Framework vorgegebene Strukturierung schon Hinweise, wie er seine Applikation in Teile partitionieren kann und welche Funktionen die einzelnen Teile haben müssen.
Eine noch höhere (das heißt abstraktere) Form der objektorientierten Modellierung kann mit Patterns erfolgen. Patterns verfolgen hauptsächlich die Idee der Wiederverwendung auf der Ebene des Designs. Im Gegensatz zu Frameworks, die immer einen recht komplexen Gesamtüberblick eines Modells liefern, konzentrieren sich Patterns auf die Modellierung von kleineren Teilproblemen. Patterns beschreiben jeweils eine Problemsituation, wo sie auftritt und wie sie gelöst werden kann. Dadurch sind sie in vielen verschiedenen Bereichen einsetzbar. Im Gegensatz dazu sind Frameworks eher starre und komplexe Gebilde, die nur in wenigen unterschiedlichen Applikationen eingesetzt werden können.
Patterns werden häufig eingesetzt, um Frameworks zu dokumentieren. Ist die „Pattern-Sprache“ bekannt, so kann schnell ein Einblick in das vorgegebene Framework erhalten werden. Jedes Pattern beschreibt dabei das Zusammenspiel zwischen einigen Einzelkomponenten des Frameworks. Mit ihnen kann Schritt für Schritt das gesamte Modell beschrieben werden. Eine kurze Gegenüberstellung von Patterns und Frameworks findet sich in [GHJ95] auf den Seiten 26 bis 28.

2.5 Pattern-Notation

Es gibt verschiedene Notationen, um Patterns aufzuschreiben; diese unterscheiden sich jedoch nicht wesentlich voneinander. Im folgenden wird die Notation von Gamma, Helm, Johnson und Vlissides (siehe [GHJ95]) vorgestellt.

- **Name des Patterns**
  Der Name soll in kurzer Form den Inhalt des Patterns charakterisieren. Die Wahl eines ein-

fachen, griffigen und selbsterklärenden Namens ist eine wichtige Entscheidung, da diese Namen im Vokabular der Entwerfer auftreten und eine schnelle Methode zur Kommunikation der Entwerfer untereinander bilden.

Beispiel (siehe *Einfache Indirektion (67)*):

Einfache Indirektion

- **Zweck des Patterns**
  An dieser Stelle wird kurz beschrieben, wozu das Pattern da ist, welches Problem oder welche Design-Entscheidung es löst und welchen Zweck es erfüllt.

  Das Pattern dient zur näheren Beschreibung einer Gruppe von Objekten. Alle Eigenschaften, die mehrere Objekte gemeinsam haben, werden dabei in eine extra Klasse ausgelagert.

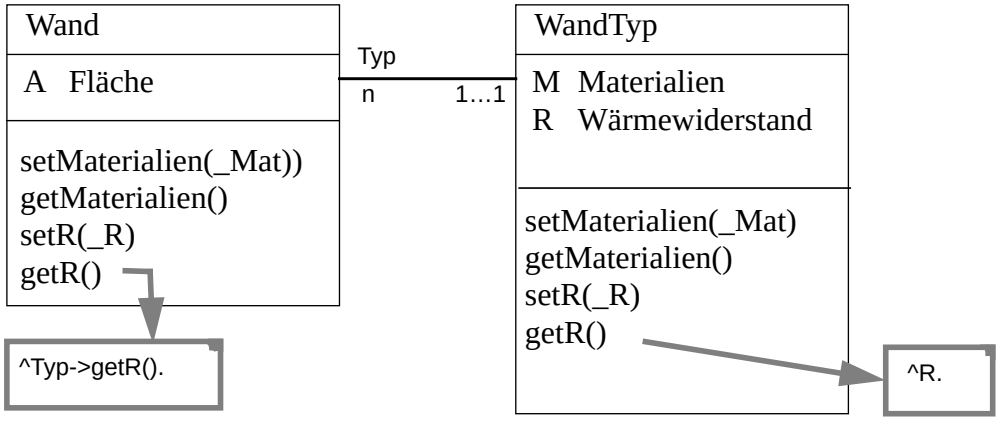
- **Bekannt unter**
  Hier werden andere bekannte Namen des Patterns angegeben, falls solche vorhanden sind. Es sollte auch angegeben werden, wo das Pattern unter dem anderen Namen benutzt wird.

  Item Description (siehe [Coa92], Seite 153)

- **Motivation**
  Hier wird an einer Beispielsituation erklärt, wann das Pattern eingesetzt werden kann und wie es das Problem in dem Beispiel löst. Diese Beispielsituation soll helfen, die eher abstrakte Strukturbeschreibung des Patterns leichter zu verstehen.

  Bei einem Gebäude werden häufig nur wenige verschiedene Wandtypen verwendet. Eine Außenwand beispielsweise besteht immer aus denselben Materialien und hat immer den gleichen Schichtaufbau. Andererseits hat auch jede Außenwand ihre individuellen Eigenschaften wie Lage, Größe oder Raumzugehörigkeit. Die gemeinsamen Attribute Materialien und Schichtaufbau einer Wand können also in einer neuen Klasse „Wandtyp“ gespeichert werden.

  Häufig wird bereits in der Motivation die Beispielsituation graphisch dargestellt, um schneller einen Überblick über die Problemsituation zu vermitteln.



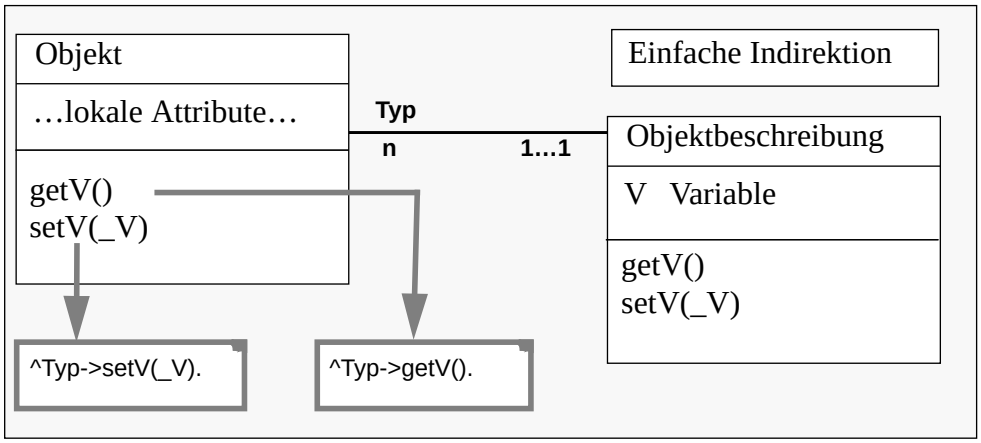
- **Anwendbarkeit**
  Dieser Abschnitt soll Anworten zu folgenden Fragen geben:
  In welcher Situation kann das Pattern angewendet werden? Welche schlechten Entwürfe können durch dieses Pattern verbessert werden? Wie erkennt man eine Situation, in der das Pattern angewendet werden kann?

  Die Einfache Indirektion kann angewendet werden, wenn mehrere unterschiedliche Objekte eine Eigenschaft miteinander teilen. Ebenso kann sie verwendet werden, wenn Daten entlang einer Relation zwischen zwei Objekten ausgetauscht werden sollen.

- **Struktur**
  Hier folgt nun der eigentliche Aufbau des Patterns. Die Modellnotation, in der das Pattern präsentiert wird, ist prinzipiell beliebig. Wichtig ist, daß die Grundprinzipien des Patterns deutlich werden und daß genug Informationen angegeben werden, damit das Pattern angewendet werden kann. Da die Patterns in dieser Arbeit zur automatischen Generierung von Software herangezogen werden, muß ihre Struktur in einer formalen Form vorliegen. Dazu wird die OMT-Notation benutzt. Sie beschreibt den Aufbau des jeweiligen Patterns und enthält zugleich den zu generierenden Code. Zur Verdeutlichung der dynamischen Strukturen werden eventuell zusätzlich noch andere Modellnotationen benutzt.

  Das Pattern beruht darauf, daß es zu den individuell unterschiedlichen Objekten (hier in der Klasse *Objekt* zusammengefaßt) eine weitere Klasse *Objektbeschreibung* gibt.
  Diese beiden Klassen sind mit einer n:1 Relation verknüpft. Die Zugriffsfunktionen der gemeinsamen Attribute (also Instanzenvariablen) werden bei den Objekten nachgebildet, so daß auf diese Attribute genau wie auf lokale Attribute zugegriffen werden kann. Die einzelnen Objekte

merken also gar nicht, wenn auf eine gemeinsame Instanzenvariable zugegriffen wird.



- **Mitwirkende Objekte**
  Anschließend an die Strukturbeschreibung erfolgt eine Erläuterung der an dem Pattern mitwirkenden Objekte bzw. Klassen und den Relationen. Es werden die Verantwortlichkeiten der einzelnen Objekte beschrieben und Hinweise gegeben, wie die Objekte im späteren Entwurf verwendet werden können.

  Die Klasse *Objekt* wird mit der Klasse *Objektbeschreibung* verbunden. Gemeinsame Attribute werden als Instanzenvariablen bei der Klasse *Objektbeschreibung* angelegt. Zusätzlich werden bei beiden Klassen Zugriffsfunktionen auf die Instanzenvariablen angelegt. Die Zugriffsfunktionen der Klasse *Objekt* verweisen dabei auf die entsprechenden Methoden ihrer Objektbeschreibung.

- **Zusammenarbeit**
  Wie arbeiten die einzelnen Objekte zusammen, um ihre Aufgabe zu erfüllen?

  Die Instanzen der Klasse Objekt arbeiten auf den gemeinsamen Attributen so, als ob sie lokal wären. Die Zugriffsfunktionen sind so definiert, daß keine lokalen Attribute verändert oder gelesen werden, sondern es wird der Zugriff an die Objektbeschreibung weitergeleitet.

- **Konsequenzen**
  Wie gut erfüllt das Pattern seine Aufgabe? Welche Zugeständnisse an den restlichen Entwurf müssen gemacht werden? Welche Teile des Patterns sind fest vorgegeben und wo liegen die Teile, die bei jedem Entwurf individuell anzupassen sind?

  Die Anwendung des Patterns hat sowohl Konsequenzen für den (schreibenden) Zugriff auf gemeinsame Attribute, als auch für die Instanziierung.
  Beim Schreiben eines gemeinsamen Attributes muß immer bedacht werden, daß eventuell andere Objekte auch noch denselben Attributwert haben. Das Schreiben eines Attributes beeinflußt also immer eine ganze Gruppe von

Objekten.
Bei der Instanziierung der Objekte muß darauf geachtet werden, daß die passenden Objektbeschreibungen auch immer richtig mit den Objekten verknüpft werden.

- **Implementation**
  Falltüren, Hinweise und Techniken, wie das Pattern effektiv angewendet werden kann. Hinweise auf sprachspezifische Konstrukte.

  Dieser Abschnitt ist für einen Entwerfer, der mit den in dieser Arbeit vorgestellten Patterns umgeht, wenig interessant, da die Implementierungsarbeit ja von einem Generator übernommen wird. Für den Generator allerdings kann an dieser Stelle eingetragen werden, wie das Pattern am besten implementiert wird.

- **Beispielimplementierung und Benutzung**
  Quelltexte, die eine Anwendung des Patterns zeigen. Üblicherweise sind diese Beispiele in Smalltalk oder C++ geschrieben. Die Beispiele in dieser Arbeit sind in Visual Works (Smalltalk-Dialekt, siehe [Vis95]) oder in C++ geschrieben.

```
Klasse Objekt:
getV
        „Gibt Wert des Attributes V einer
        Objektbeschreibung zurück“
        ^Typ getV.
…
```

- **Bekannte Anwendungen**
  In diesem Abschnitt können bekannte Anwendungen des Patterns angegeben werden.

  Dieser Teil ist für den Umgang mit den hier vorgestellten Patterns von keiner großen Bedeutung, da sie alle im selben Kontext angewendet werden. Von Interesse wären an dieser Stelle Anwendungen des Patterns aus unterschiedlichen Gebieten, um Hinweise zu geben, an welchen Stellen das Pattern überall benutzt werden kann.

- **Verwandte Patterns**
  Zuletzt werden Verweise auf ähnliche Patterns angegeben. Kann ein spezielles Pattern in einem Entwurf nicht benutzt werden, so sollten in diesem Abschnitt andere Patterns aufgeführt werden, die eine ähnliche Aufgabe erfüllen. Auch sollte kurz auf Unterschiede zwischen den Patterns eingegangen werden.

  Komplexe Indirektion (69) - Verfolgen einer 1:n Relation.

2.6 Software-Entwurf mit Patterns

Der Software-Entwurf mit Patterns erfolgt durch sukzessive Anwendung der entsprechenden Patterns. Dazu wird das ursprüngliche Problem in Teilprobleme aufgespalten und diese wer-

den, wenn möglich, durch die einzelnen Pattern gelöst. Im Gegensatz zum reinen „Divide and Conquer“ Entwurf stehen die Patterns jedoch nicht ganz für sich alleine, sondern sie arbeiten zusammen und ergänzen sich gegenseitig. Im folgenden wird eine Software-Modellierungsmöglichkeit vorgestellt, die es erlaubt, die korrekte Anwendung der Patterns zu automatisieren, um die Software-Entwicklung für einen eingeschränkten Bereich (der Gebäudesimulation) zu vereinfachen.
Patterns spiegeln Techniken und Beispiele wieder, wie gute Software geschrieben werden kann. Werden sie mit Bedacht und kreativ eingesetzt, so können sie in einer Vielzahl von Anwendungen verwendet werden. Eingeschränkte Patterns, die nur zur Lösung eines speziellen Problems dienen, machen für den Entwurf beliebiger Programme wenig Sinn. Schränkt man jedoch den Einsatzbereich der Patterns ein (beispielsweise auf die Simulation von Gebäuden), so liegt in diesen „Spezialpatterns“ eine große Einsatzmöglichkeit. Spezielle Patterns können die Probleme, die sie lösen, viel genauer eingrenzen und erläutern, und ihre Implementierung (d.h. Anwendung) kann automatisch erfolgen.
Der Entwerfer eines Simulators braucht daher kein Spezialist auf dem Gebiet der Simulation zu sein. Er sucht einfach zu den in der Simulation auftretenden Problemen ein passendes Pattern heraus. Welche Seiteneffekte und Konsequenzen die Anwendung des Patterns hat, kann in der Pattern-Beschreibung nachgelesen werden.
Die Patterns, so wie sie im folgenden benutzt werden, stellen jeweils eine eigene Funktionalität (auf Code-Ebene) zur Verfügung und bilden ein Bindeglied zwischen einzelnen Komponenten aus unterschiedlichen Modellen. Durch die Bindung der Patterns an Modell-Komponenten wird die Rolle der Patterns quasi umgekehrt: statt der reinen Beschreibung eines Frameworks dienen sie jetzt zur Modellierung und Verfeinerung von Modellen.
Als Vorgabe für den Entwurf eines Simulators gibt es zum einen den Pattern-Katalog, und zum anderen dienen schon bereits vorgefertigte Modelle als Eingabe für den Entwurf. Die Benutzung der Patterns bewirkt nun, daß einzelne Komponenten der Eingabe-Modelle ausgewählt und mit zusätzlicher Funktionalität versehen werden. Alle diese Teile werden dann zu einem komplexen Simulator-Modell zusammengesetzt, das den Gebäude-Simulator komplett beschreibt.

Der Entwurf eines Simulators wird in Abbildung 7 beschrieben (vergleiche Abbildung 3 auf Seite 6).

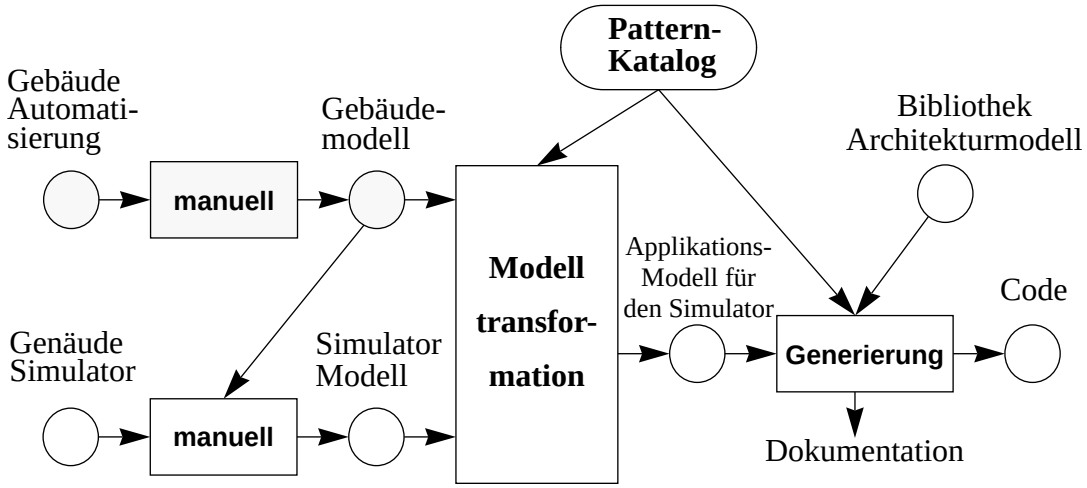


Abb. 7: Entwurf eines Gebäude-Simulators

Das aktuelle Anwendungsfeld des Sonderforschungsbereiches 501 ist die Gebäudeautomatisierung. Es sollen Programme zur Steuerung, Simulation und Konstruktion von Gebäuden entwickelt werden. Um die in der Gebäudeautomatisierung anfallenden Begriffe zu erklären und um Zusammenhänge zwischen den Begriffen zu erläutern, wurde ein Lexikon geschrieben (siehe [Sah96]). Dem Lexikon liegt ein Gebäudemodell zu Grunde, in dem die prinzipielle Struktur eines Gebäudes und die Beziehungen zwischen den einzelnen Gebäudekomponenten beschrieben sind. Es kann daher als Grundlage für sämtliche Applikationen im Gebäudebereich dienen.
Das spezielle Anwendungsfeld, das im folgenden behandelt werden soll, ist ein Simulator für Gebäude, die mit dem Gebäudemodell modelliert werden können. Hier könnte man sich auch andere Anwendungen, wie zum Beispiel eine Heizungssteuerung, vorstellen, die auch auf ähnliche Weise behandelt werden können.
Bei einem Simulator treten einige Besonderheiten auf. Dazu zählen sowohl eigene Objekte und Algorithmen, wie zum Beispiel Scheduling oder Event-Verarbeitung, als auch spezielle Probleme wie Echtzeitfähigkeit und Verteilung. Diese Besonderheiten können in Patterns und speziellen Datenmodellen codiert werden und machen die Charakteristik des Anwendungsprojektes „Simulator“ aus. Jedes Pattern beschreibt und löst ein spezielles Problem des Simulators und bildet zusammen mit allgemeinen Patterns die Entwurfsgrundlage für den Simulator. Die Patterns sind also speziell an das Anwendungsgebiet „Gebäude-Simulator“ angepaßt (domain-modelling).
Der Entwurf des Simulator-Gesamtmodells besteht nun aus der Zuordnung dieser Patterns zu Objekten aus dem Gebäudemodell. Um zum Beispiel die Temperatur in den Räumen des Gebäudes zu simulieren, wird das Pattern „Simulation thermischer Masse“ der Klasse Raum aus dem Gebäude-Modell zugeordnet. Die Konsequenzen, die die Anwendung des Patterns hat, sind in dem Pattern beschrieben, ebenso, was außerdem modelliert werden muß. Sämtliche Patterns sind in einem Pattern-Katalog enthalten (siehe beispielsweise Anhang A). Die

einzelnen Patterns aus diesem Katalog können zur Modellierung des Simulators herangezogen werden. Ebenso enthalten die Patterns Hinweise, wie sie von einem Generator implementiert werden können. Zusammen mit Hinweisen, wie das Architekturmodell in Programm-Code umgesetzt werden kann, ist es möglich, den Simulator anhand der Modellierung zu generieren.

2.7 Beispielentwurf

In diesem Kapitel soll der Software-Entwurf mit Patterns verdeutlicht werden. Die verwendeten Patterns befinden sich im Anhang A, eine genaue Erklärung der Möglichkeiten und Anwendungsarten der Patterns steht in Kapitel 2.8.
Als Beispiel soll die Temperatur zweier Räume eines einfachen Hauses simuliert werden (siehe Abbildung 8). Die beiden Türen können während der Simulation geöffnet werden, und die Temperaturen der Räume passen sich dann entsprechend an. Der Heizkörper in Raum 1 heizt zunächst konstant mit einer Leistung von 0,5 KW, die Heizleistung kann aber auch dynamisch von einer Simulator-Steuerung angepaßt werden.

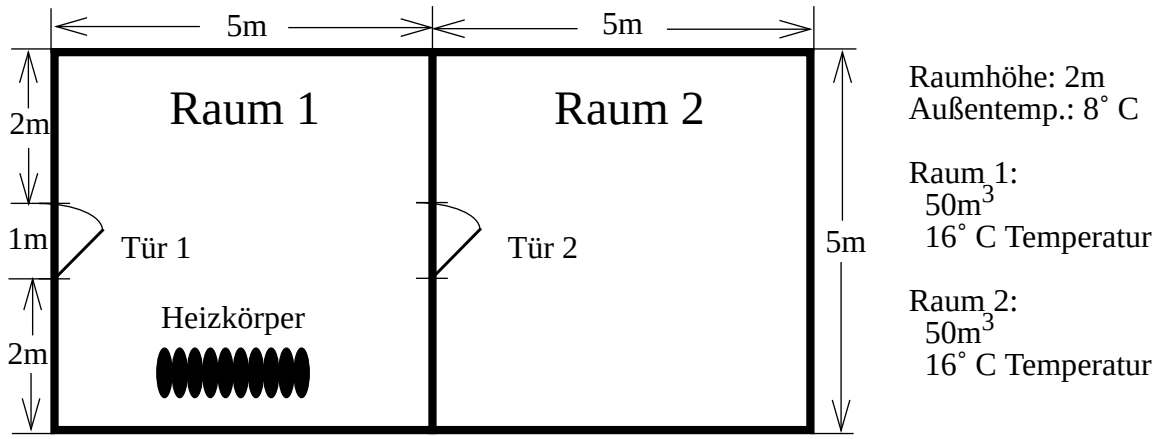


Abb. 8: Beispielhaus

Zur Simulation werden offensichtlich die Objekte „Raum“, „Raumteiler“ (Wand und Tür), „Heizkörper“ und „Umgebung“ benötigt. Der entsprechende Ausschnitt des Gebäudemodells sieht vereinfacht folgendermaßen aus:

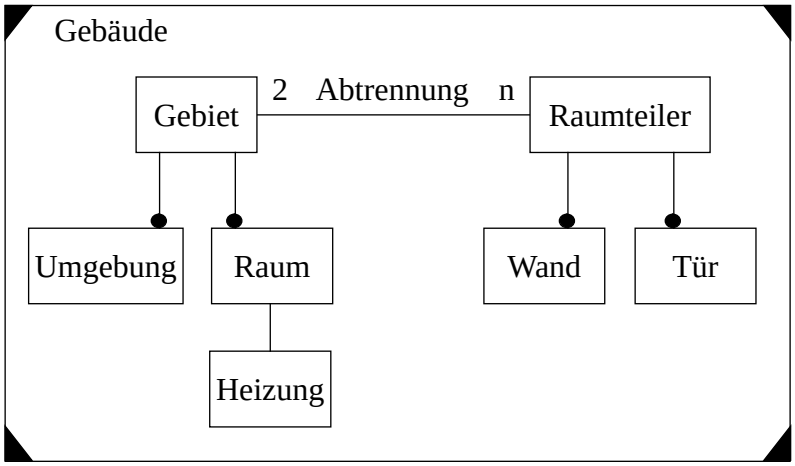


Abb. 9: Ausschnitt aus dem Gebäudemodell

Ein Gebiet kann entweder ein Raum oder die Umgebung sein. Je 2 Gebiete sind durch Raumteiler getrennt. Ein Raumteiler ist entweder eine Wand oder eine Tür.
In dem Beispiel soll die Temperatur der Räume simuliert werden. Dazu kann das Pattern „Simulation thermischer Masse“ eingesetzt werden. Die thermische Masse ist in diesem Fall die Luft in den Räumen. Die Verbindung zwischen zwei Räumen geschieht über Raumteiler. Die Bindung des Patterns an das Gebäudemodell ist der Graphik aus Abbildung 10 zu entnehmen.

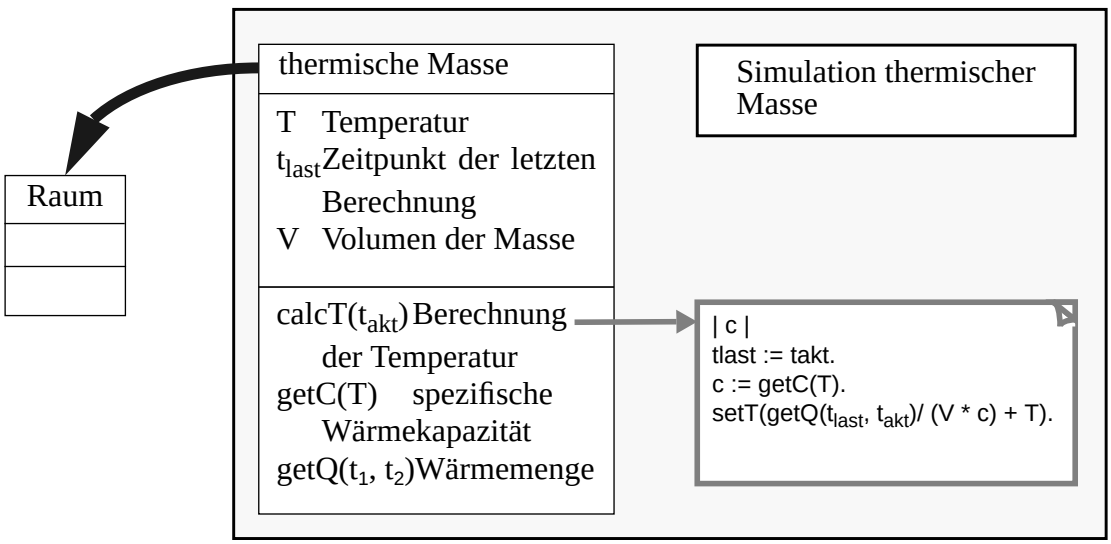


Abb. 10: Anwendung des Patterns „Simulation thermischer Masse“

Bindungen:

Klassen:

• thermische Masse -> Raum

Durch die Bindung des Patterns an die Klasse *Raum* aus dem Gebäudemodell werden die in dem Pattern angegebenen Instanzenvariablen und Funktionen automatisch für alle Räume angelegt. Gleichzeitig mit der Bindung auf Klassenebene können auch Umbenennungen und bei Funktionen Parameteranpassungen vorgenommen werden. Näheres dazu ist in Kapitel 2.8 beschrieben. Das Pattern *Simulation thermischer Masse (45)* gibt für die Funktion *calcT()* gleichzeitig das Interface und die Implementierung vor. Das heißt, daß für jeden Raum die Temperatur mit derselben Formel $\left(T_{i+1} = \frac{Q(t_{last}, t_{akt})}{V \cdot C(t_{akt})} + T_i \right)$[3] berechnet wird[4]. Die spezifische Wärmekapazität eines Raumes und die Wärmemenge, die auf ihn während des letzten Berechnungsintervalles eingewirkt hat, müssen getrennt modelliert werden. Das Interface der entsprechenden Funktionen ist allerdings durch das Pattern schon vorgegeben, so daß erkennbar ist, daß sie noch entworfen werden müssen.

Bei der Instanziierung des Patterns, das heißt bei der eigentlichen Erzeugung der Simulationsobjekte, muß noch für jede Instanzenvariable, die im Pattern vorkommt, ein passender (Initial–) Wert eingetragen werden. Im Beispiel wäre das also:

Raum 1: T := 16, v:= 50, tlast := 0.
Raum 2: T := 16, v:= 50, tlast := 0.

Als nächstes müssen die Funktionen *getC()* und *getQ()* modelliert werden, da im Pattern nur ihr Interface vorgegeben wurde. Die spezifische Wärmekapazität der Raumluft kann man der Einfachheit halber als konstant annehmen. Also kann an die Funktion *getC()* das Pattern *Konstanter Wert (63)* gebunden werden.

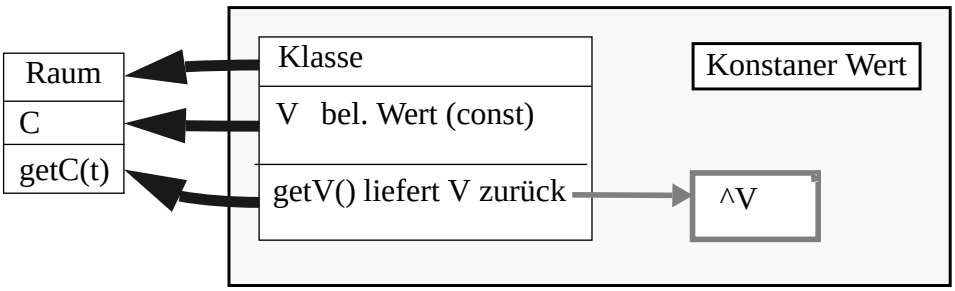


Abb. 11: Konstante Wärmekapazität der Raumluft

Die Anbindung des Patterns an eine (oder mehrere) Klassen aus dem Gebäudemodell wird graphisch durch die fettgedruckten Pfeile dargestellt. Dabei geht jeweils ein Pfeil von einer im Pattern definierten Klasse zu einem Objekttypen aus einem der bisherigen Modelle. Dadurch

3. Die meisten Formeln, die in dieser Arbeit erwähnt werden, stammen aus [LJK94] und [PDr80] und werden dort ausführlich beschrieben.
4. Gibt es unterschiedliche Raumtypen, die verschieden berechnet werden sollen, so müssen diese über den Vererbungsmechanismus im Ausgangsmodell separat modelliert werden.

werden die Klassen ausgewählt, für die neue Funktionalität modelliert werden soll. Die Pfeile auf Variablen- oder Funktionenebene geben jeweils an, welche Instanzenvariablen oder Funktionen korrespondieren. Dabei sind Namensumbenennungen möglich. Wird eine Variable oder Funktion unmodifiziert aus dem Pattern übernommen, so ist ein entsprechender Pfeil überflüssig und kann weggelassen werden.

Bindungen:

Klassen:
• Klasse -> Raum
Funktionen:
• getV() -> Raum::getC(t)
Initialisierung:
V := 1,007

Als nächstes soll die Berechnung der Wärmemenge (Funktion getQ(t_1, t_2) des Raumes) modelliert werden. Diese setzt sich aus der Transmissionswärmemenge und der Strahlungswärmemenge der Heizung zusammen ($Q_{ges} = Q_T + Q_S$). Diese Summe kann einfach mit dem Formel-Pattern gebildet werden, wobei als Formel die Addition zweier Werte eingetragen wird (siehe Pattern *Funktion (61)* und Abbildung 12).

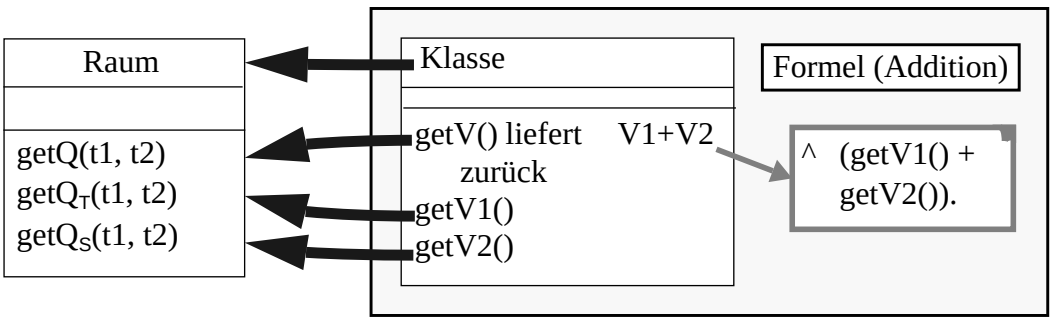


Abb. 12: Addition der Wärmemengen-Anteile

Die Funktion *getQ(t1, t2)* wird implementiert als Summe der Rückgabewerte der (neuen) Funktionen *getQ$_T$(t1, t2)* und *getQ$_S$(t1, t2)*. Die Parameter t1 und t2 werden also aus dem bereits bestehendem Interface der Funktion *getQ(t1, t2)* übernommen:

Bindungen:

Klassen:
• Klasse -> Raum
Funktionen:
• getV() -> Raum::getQ(t1, t2)
• getV1() -> Raum::getQ$_T$(t1, t2)
• getV2() -> Raum::getQ$_S$(t1, t2)

Als nächstes müssen die beiden Wärmemengen Q_T und Q_S modelliert werden. Die Strahlungswärmemenge aller Heizungen ist einfach die Summe der Wärmemengen der einzelnen Hei-

zungen eines Raumes. Diese indirekte Summe (der Raum benötigt die Summe von Werten seiner Heizungen) läßt sich mit dem Pattern *Komplexe Indirektion (69)* berechnen.

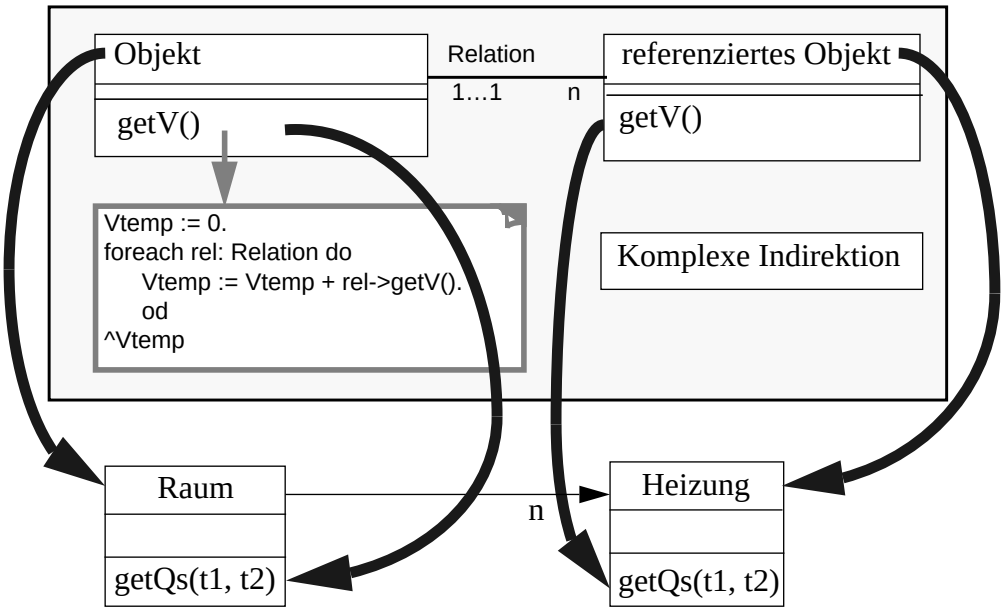


Abb. 13: Aufsummieren der einzelnen Wärmekapazitäten

Bindungen:

Klassen:
- Objekt -> Raum
- referenziertes Objekt -> Heizung

Relationen:
- Relation -> partOf (Raum, Heizung)

Funktionen:
- Objekt::getV() -> Raum::getQ_S(t1, t2)
- referenziertes Objekt::getV() -> Heizung::getQ_S(t1, t2)

Die Strahlungswärmemenge (Q) einer Heizung berechnet sich nun durch das Integral über dem Wärmestrom der Heizung im aktuellen Zeitintervall $\left(Q(t_1, t_2) = \int_{t_1}^{t_2} \Phi(t)\,dt\right)$. Solche Integrale können leicht mit einem Puffer (siehe 3.3) berechnet werden. Unter der Annahme, daß sich der Wärmestrom nicht stark ändert, kann ein einfacher Puffer (z. B. Lineare Pufferung) verwendet werden. Dazu wird das Pattern *Einfache Indirektion (67)* zur Verbindung der Heizung mit einem linearen Puffer verwendet (Abbildung 14). Die Puffer stammen aus einem separaten Modell (siehe Kapitel 3.3), so daß der verwendete Puffer sehr einfach (durch die

Neubindung der Klasse *Objektbeschreibung* an einen andern Puffer) ausgetauscht werden kann.

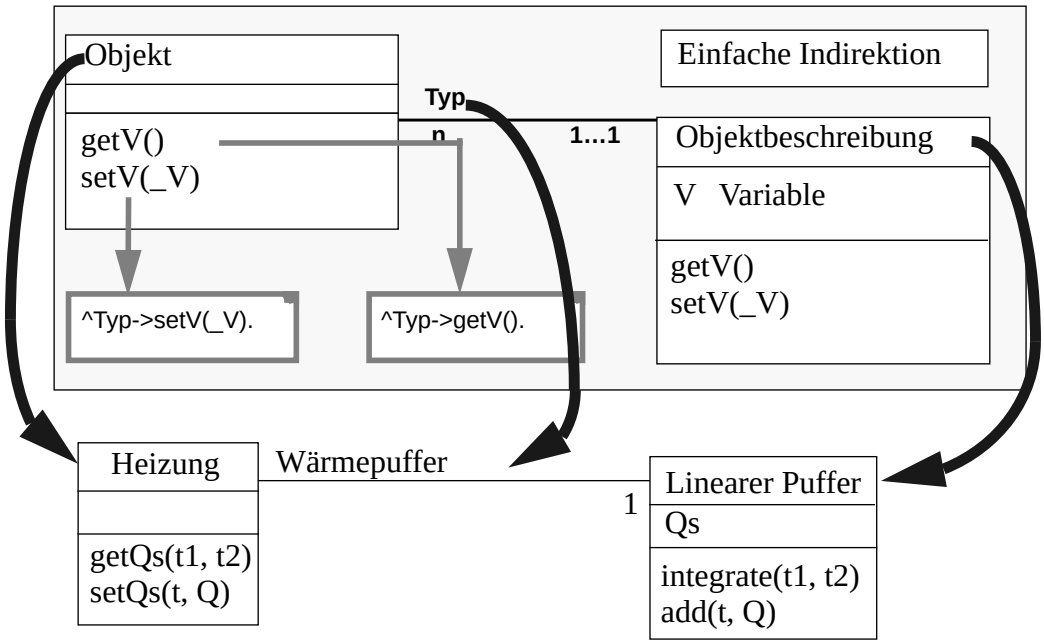


Abb. 14: Thermische Verbindung zwischen Gebieten

Bindungen:

Klassen:
- Objekt -> Heizung
- Objektbeschreibung -> Linearer Puffer

Relationen:
- Typ -> Wärmepuffer (Heizung, Linearer Puffer)

Funktionen:
- Objekt::getV() -> Heizung::getQ_S(t1, t2)
- Objekt::setV(t) -> Heizung::setQ_S(t, Q)
- Objektbeschreibung::getV() -> Linearer Puffer::integrate(t1, t2)
- Objektbeschreibung::setV(t) -> Linearer Puffer::add(t, Q)

Als letztes fehlt zur Heizungssimulation noch der von der Heizung produzierte Wärmestrom. Dieser soll von einer Gebäudesteuerung oder durch manuelle Eingaben in den Simulator gesteuert werden. Aus der Sicht des Simulators stellt der Wärmestrom also einen Aktuator

(Steuerglied) dar. Mit dem Pattern *Aktuator (55)* wird die entsprechende Funktionalität zur Verfügung gestellt. Abbildung 15 zeigt die Struktur des Patterns.

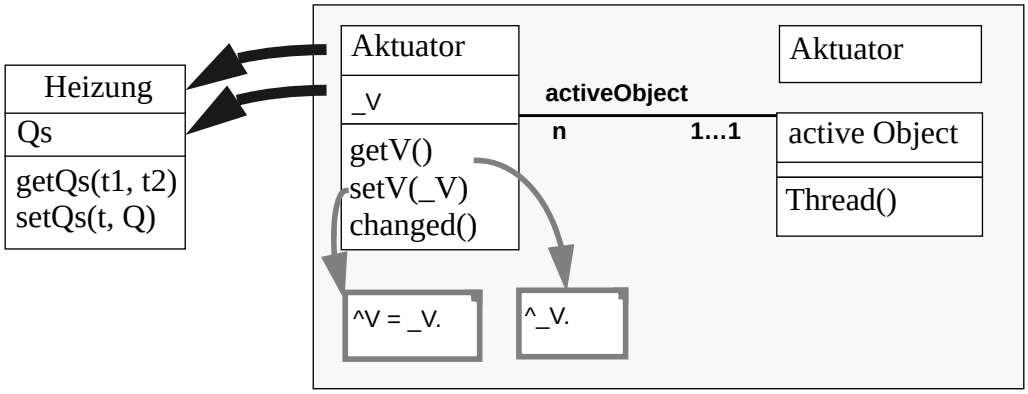


Abb. 15: Pattern *Aktuator (55)*

Bindungen:

Klassen:
- Aktuator -> Heizung

Relationen:
- activeObject -> activeObject (Heizung, ActiveObject)

Funktionen:
- Aktuator::getV() -> Heizung::getQ_S(t1, t2)
- Aktuator::setV(_V) -> Heizung::setQ_S(t, Q)

Das *active Object* dient zur Anbindung der Heizung an die Steuerung. Dadurch ist eine Heizung in der Lage, Events von Simulator zu bearbeiten (siehe dazu auch Kapitel 3.2). Das active Objekt ist in der Lage, Nachrichten von der Simulator-Steuerung zu empfangen und entsprechend zu bearbeiten. Um beispielsweise die Leistung einer Heizung zu verändern, kann die Instanzesvariable Qs der Heizung von der Steuerung direkt gesetzt werden. Diese Änderung macht es jedoch eventuell erforderlich, daß sofort Neuberechnungen einiger Simulationsgrößen stattfinden müssen. Dazu kann an das Active Object der Heizung eine changed-Nachricht geschickt werden, die dann die Funktion *changed()* der Heizung aufruft. Daraufhin werden alle abhängigen Simulationsgrößen neu berechnet.

Was jetzt noch fehlt, ist die Berechnung der Transmissionswärmemenge durch die an einen Raum angrenzenden Wände. Dazu dient das Pattern *Simulation thermischer Verbindung (49),* das an die Klassen *Raum* und *Raumteiler* gebunden wird (Abbildung 16).

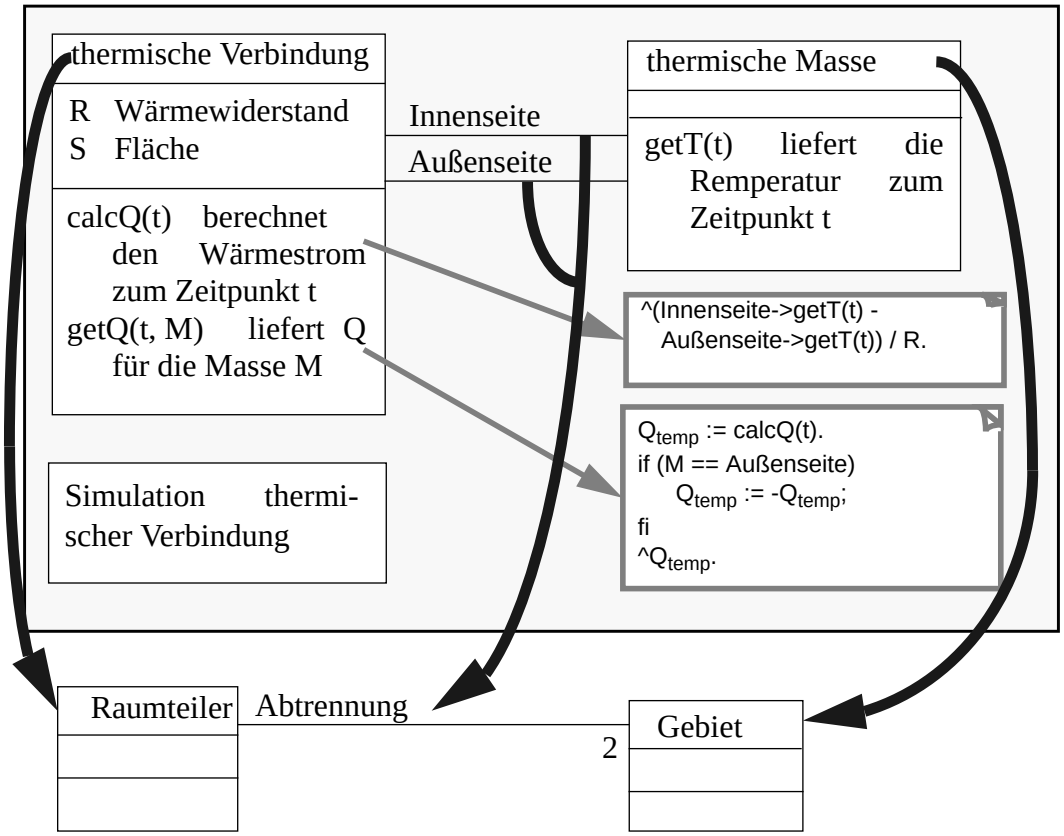


Abb. 16: Pattern *Simulation thermischer Verbindung (49)*

Bindungen:

Klassen:
- thermische Verbindung -> Raumteiler
- thermische Masse -> Gebiet

Relationen:
- Innenseite -> Abtrennung[0](Gebiet, Heizung)
- Außenseite -> Abtrennung[1](Gebiet, Heizung)

Die Methode *calcQ(t)* berechnet jeweils den aktuellen Wärmestrom durch eine Wand. Die Funktion *getQ(t, M)* liefert diese dann vorzeichenrichtig an eine thermische Masse zurück. Um den Wärmestrom durch eine thermische Verbindung berechnen zu können, muß bei der thermischen Masse die Funktion *getT(t)* vorhanden sein. Sie soll die Temperatur zum Zeitpunkt t zurückliefern. Dazu wird ein weiterer Puffer benötigt. Dieser wird an die Klasse Gebiet des Gebäudemodells gebunden (Pattern *Einfache Indirektion (67),* vgl. Abbildung 17) und speichert die berechneten Temperaturen ab. Ist das Gebiet die Umgebung, so wird der Puffer einfach mit einem konstanten Wert (Pattern *Konstanter Wert (63)*) gefüllt und nicht weiter aktualisiert (Die Außentemperatur wird als konstant angenommen). Im Falle eines Raumes, wird bei jeder Aktualisierung der Temperatur die Funktion *setT (T)* (vgl. Abbildung 10)

aufgerufen. Diese wird mit dem Pattern *Einfache Indirektion (67)* so überschrieben, daß die aktuelle Temperatur in dem Puffer abgelegt wird.

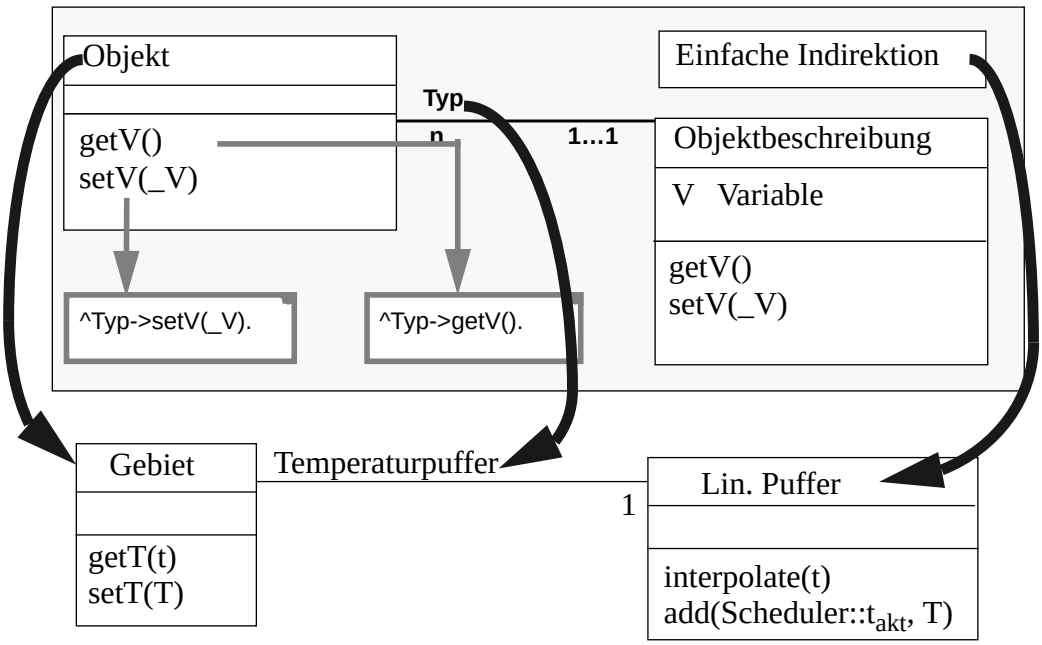


Abb. 17: Pufferung der Temperaturen

Bindungen:

Klassen:
- Objekt -> Gebiet
- Objektbeschreibung -> Linearer Puffer

Relationen:
- Typ -> Temperaturpuffer (Gebiet, Trapezregel)

Funktionen:
- Objekt::getV() -> Gebiet::getT(t)
- Objekt::setV(t) -> Gebiet::setT(T)
- Objektbeschreibung::getV() -> Linearer Puffer::interpolate(t)
- Objektbeschreibung::setV(t) -> Linearer Puffer::add(Scheduler::t_{akt}, T)

Da die aktuelle Zeit, zu der die Temperatur berechnet wurde, eine für alle Simulationsobjekte globale Variable ist, kann diese vom Scheduler abgefragt und als Parameter an die Funktion *add* des Puffers übergeben werden (siehe 2.8.3).

Was jetzt noch fehlt, ist ein Puffer, der die berechneten Wärmeströme zwischenspeichert und die vom Raum benötigte Wärmemenge daraus berechnet. Das Pattern *Einfache Indirektion (67)* wird dazu an den Raumteiler und einen Puffer gebunden. Die Bindungen sehen wie folgt aus:

Bindungen:

Klassen:
- Objekt -> Raumteiler
- Objektbeschreibung -> Trapezregel

Relationen:
- Typ -> Gebiete (Gebiet, Raumteiler)

Funktionen:
- Objekt::getV() -> Raumteiler::getQ(t1, t2)
- Objekt::setV(t) -> Raumteiler::setQ(Scheduler::t_{akt}, Q)
- Objektbeschreibung::getV() -> Trapezregel::integrate(t1, t2)
- Objektbeschreibung::setV(t) -> Trapezregel::add(t, Q)

Damit ist die Modellierung des Simulators fast fertig. Es fehlt lediglich noch die kontinuierliche Simulation der Raumteiler und der Räume. Diese wird dadurch erreicht, daß das Pattern *Kontinuierliche Simulation (53)* an die entsprechenden Objekte gebunden wird. Das vollständige Gesamtmodell ist in Abbildung 18 zu sehen. Die Patterns sind in diesem Bild als Wolken dargestellt, die über Pfeile an die Objekttypen gebunden sind. Der Simulator konnte durch die Anwendung von 14 Patterns (davon 8 unterschiedliche) modelliert werden.

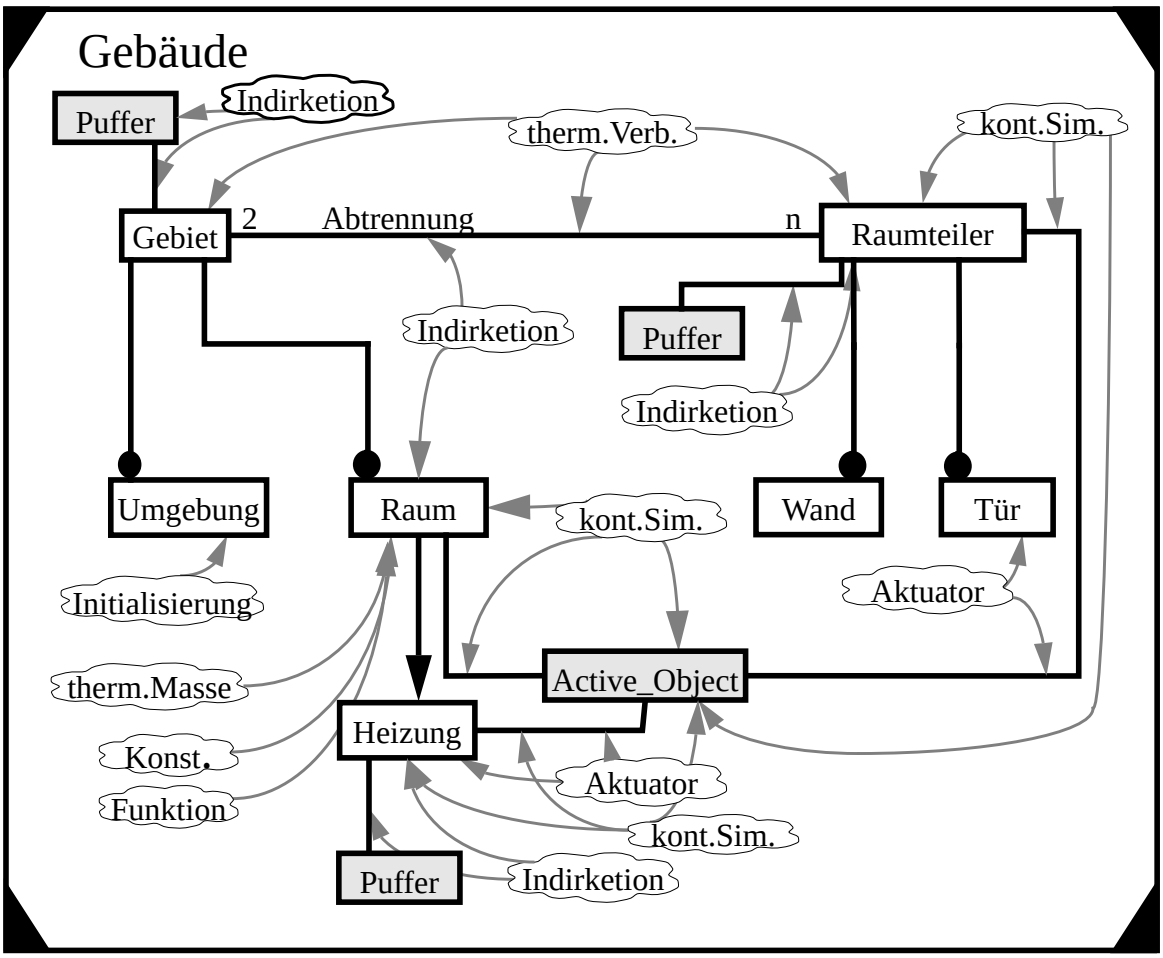


Abb. 18: Komplettes Modell mit Patterns

Ein einmal angefertigtes Modell kann im Bedarfsfall recht einfach abgewandelt oder verfeinert werden. Soll beispielsweise eine Wand aus mehreren Schichten bestehen und der Wärmedurchgangskoeffizient („k-Wert“) mehrschichtiger Wände berechnet werden, so kann die Modellierung dahingehend angepaßt werden. Dazu muß eine neue Modell-Klasse „Wandschicht“ aufgenommen werden. Jede Wandschicht kennt ihren Wärmewiderstand und ihre Dicke. Um daraus den Wärmedurchgangswiderstand der gesamten Wand zu berechnen, genügt es, die einzelnen Schichten mit dem Pattern *Einfache Indirektion (67)* an eine Wand anzubinden und durch eine *Funktion (61)* den gewünschten Wert zu berechnen.
Soll zusätzlich noch jede einzelne Wandschicht als eigene thermische Masse betrachtet werden, ist eine kleine Umstrukturierung der bereits gebundenen Patterns notwendig. Der Wärmeübergang vom Raum auf die erste Wandschicht wird über eine Zwischenschicht modelliert. Die neue Modell-Klasse „Zwischenschicht“ dient also als thermische Verbindung zwischen den Klassen „Raum“ und „Wandschicht“. Das Pattern *Simulation thermischer Verbindung (49)*, das vorher an den Raum und die Wand gebunden war, muß jetzt an die Zwischenschicht gebunden werden. Der Wärmewiderstand solch einer Zwischenschicht hängt von der Beschaffenheit der äußersten Wandschicht ab und kann in Tabellen nachgelesen werden (siehe [LJK94] auf der Seite 131). Jetzt fehlt nur noch, die einzelnen Wandschichten als thermische Massen zu betrachten. Dies geschieht durch die Bindung des Patterns *Simulation thermischer Masse (45)* an die Modell-Klasse Wandschicht. Zuletzt sorgt die *Kontinuierliche Simulation (53)* für eine quasi-kontinuierliche Aktualisierung der Temperatur einer Wandschicht (siehe Abbildung 19).

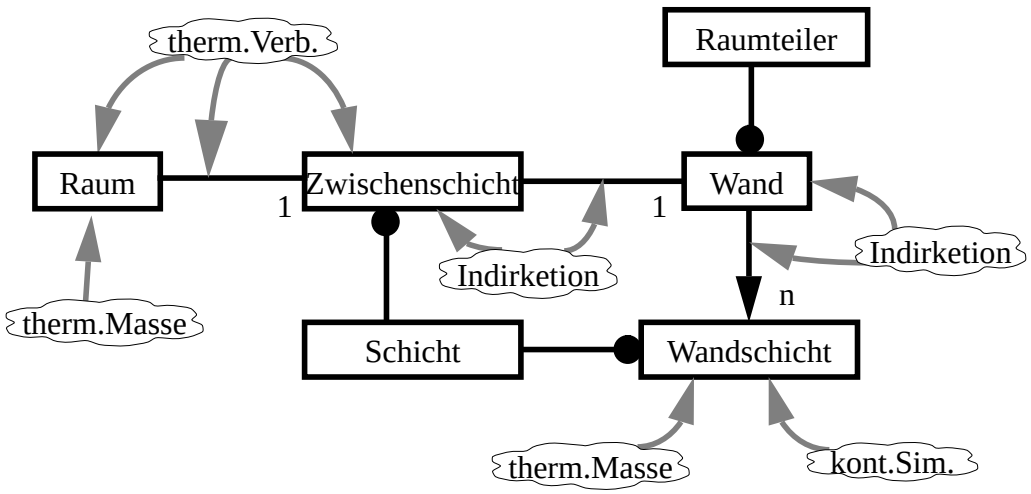


Abb. 19: Verfeinerter Ausschnitt aus Abbildung 18

2.8 Bindung der Patterns

Wie im vorigen Beispiel gesehen, besteht die Modellierung eines Simulators hauptsächlich aus der Bindung von Patterns an bereits bestehende oder neu anzulegende Modellkomponen-

ten. Den Ausgangspunkt bilden ein oder mehrere Objektmodelle, die durch Anbindung an Patterns verfeinert werden. Die einzelnen Objektklassen erhalten dabei die in den Patterns spezifizierte Funktionalität. Die Bindung eines Patterns an Objekte bzw. Klassen geschieht in mehreren Ebenen, die im folgenden näher erläutert werden:

2.8.1 Bindung auf Klassenebene

In diesem Schritt wird jeder Klasse, die in dem Pattern vorkommt, eine Klasse aus einem der bereits bestehenden Objektmodelle zugeordnet. Dies geschieht durch einfaches Auswählen einer Klasse aus einem der „Eingabe“-Modelle (Gebäude-Modell, Simulator-Modell, etc., siehe Abbildung 7 auf Seite 16). Eventuell kann es vorkommen, daß eine benötigte Klasse in keinem dieser Modelle existiert. Wenn es sich dabei nur um ein selten benötigtes Hilfsobjekt handelt, so kann an dieser Stelle auch eine neue, „leere“ Klasse angelegt werden, die ihre erste Funktionalität durch das aktuelle Pattern bekommt. Wird diese neue Klasse jedoch an mehreren Stellen im Simulator benötigt, so ist es eventuell besser, sie in eines der Eingabe-Modelle aufzunehmen, damit der Sinn der Klasse und ihr Zusammenspiel mit anderen Objekten deutlich wird.
Durch die Bindung der Patterns auf Klassenebene werden also alle Objektklassen ausgewählt bzw. angelegt, die durch das Pattern erweitert werden sollen.

2.8.2 Bindung der Funktionen und Instanzenvariablen

Sind alle Klassen ausgewählt, auf die das Pattern angewendet werden soll, so müssen in einem zweiten Schritt die Instanzenvariablen und die Funktionen, die im Pattern definiert sind, angepaßt werden. Ist keine Anpassung notwendig (d.h. eine Instanzenvariable oder Funktion soll ohne Änderung aus dem Pattern übernommen werden), so braucht an dieser Stelle nichts unternommen zu werden. Häufig sind jedoch Anpassungen nötig, da die Funktionen im späteren Programm andere Namen haben sollen, als im Pattern vorgegeben. In diesen Fällen kann der vorgegebene Name geändert werden. Dabei kann der ursprüngliche (im Pattern definierte) Name durch einen beliebigen, gültigen[5] Namen ersetzt werden.
Instanzenvariablen können an bereits im Modell vorgegebene Variablen gebunden oder - falls keine passenden Variablen im Modell enthalten sind - neu angelegt werden. Im letzteren Fall wird für die entsprechende Modell-Klasse eine neue Variable mit dem gewünschten Namen erzeugt und es werden zusätzlich Zugriffsfunktionen auf diese Variable vorgesehen (siehe Pattern *Instanzenvariable (58)*). Nachdem die Instanzenvariable angelegt wurde, kann sie über die Zugriffsfunktionen auch von anderen Patterns aus gelesen oder neu gesetzt werden.
Funktionen treten in den Patterns in zwei Formen auf. Entweder wird in dem Pattern nur das Interface, also die Schnittstelle der Funktion, beschrieben, oder es wird zusätzlich noch die Implementierung derselben vorgegeben. Ist die Funktion vollständig im Pattern beschrieben (also Interface und Implementierung), so wird diese Funktion bei der Generierung komplett erzeugt. Ein vorgegebenes Interface kann mit einer beliebigen Funktions-Implementierung verknüpft werden. Das Pattern *Komposition (72)* beispielsweise beschreibt, wie eine Operation auf einer Menge von Objekten ausgeführt werden kann. Angenommen, eine Menge von Räumen soll aufgefordert werde, sich neu zu berechnen. Dies geschehe durch den Aufruf der

5. „gültige“ Namen für Instanzenvariablen und Funktionen sind alle diejenigen, die auch in der Ziel-Programmiersprache gültig sind.

Funktion *calculate()* der Klasse Raum. Eine Klasse *Haus* beinhaltet Verweise auf alle Räume, die den Funktionsaufruf von *calculate()* erhalten sollen. Konsequenterweise wird also die Pattern-Klasse *Menge* an die Modell-Klasse *Haus* gebunden. Die Klasse *Komponente* wird dann mit dem Raum identifiziert. Für eine *Menge* wird im Pattern die Funktion *operation()* komplett vorgegeben. Bei der Bindung dieser Funktion an die Klasse *Haus* findet also nur eine Umbenennung statt, so daß im späteren Programm diese Methode unter dem Namen *calculate()* aufgerufen werden kann. Für einen Raum ist nur das Interface der *operate()* Funktion vorgegeben. Also wird nur vermerkt, daß der Raum (nach einer Umbenennung) eine Funktion mit dem Namen *calculate()* haben muß. Die eigentliche Funktionalität dieser Methode kann durch ein anderes Pattern spezifiziert werden. In diesem anderen Pattern muß dann eine Funktion komplett spezifiziert sein, die sich durch Umbenennung an die *calculate()* Methode binden läßt.
Durch diese Aufspaltung der Methoden in Interface und Implementation können also mehrere Pattern zusammenarbeiten. Ein Pattern gibt die Struktur einer Funktion vor, und ein anderes implementiert dann die eigentliche Funktionalität.

2.8.3 Parameteranpassung

In einigen Fällen kann es vorkommen, daß die Parameter einer Funktion, wie sie im Pattern vorgesehen sind, nicht mit denen übereinstimmen, die im späteren Modell benötigt werden. Dabei kann sowohl vorkommen, daß mehr Parameter als vorgegeben benötigt werden, als auch, daß Parameter weggelassen werden können (siehe folgende Beispiele).

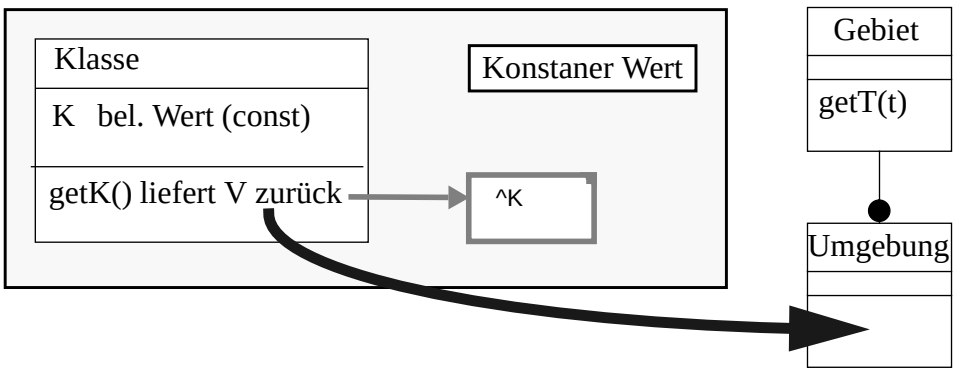


Abb. 20: Parameter hinzufügen

In Abbildung 20 soll für die Klasse Umgebung die Funktion *getT* modelliert werden. Sie soll die konstante Außentemperatur zurückliefern. Dazu wird das Pattern *Konstanter Wert (63)* an die Klasse Umgebung gebunden. Ohne eine Parameteranpassung würde eine neue, parameterlose Funktion erzeugt werden. In diesem Fall soll aber die durch die Oberklasse *Gebiet* bereits vorgegebene Funktion *getT(t)* überladen werden. Daher ist es zusätzlich zu der Umbenennung der Funktion getK aus dem Pattern notwendig, einen neuen Parameter *t* einzuführen. Dieser wird dann in der Funktion selbst nicht berücksichtigt und dient nur zur Anpassung des Methodennamens. Wird im Pattern nur das Interface einer Funktion vorgegeben und dieses soll an eine bereits bestehende Klassenmethode angepaßt werden, so sind eventuell komplexere Parameteranpassungen nötig. Dann kann es nämlich vorkommen, daß der zusätzliche Parameter nicht einfach ignoriert werden kann, sondern es wird ein sinnvoller Wert von dieser Funktion

erwartet. In diesem Fall kann angegeben werden, welcher Wert als Parameter übergeben werden soll. Möglich sind dabei Konstanten, Instanzenvariablen des aufrufenden Objektes, Funktionsparameter der aufrufenden Methode oder ein Verweis auf die aufrufende Instanz.

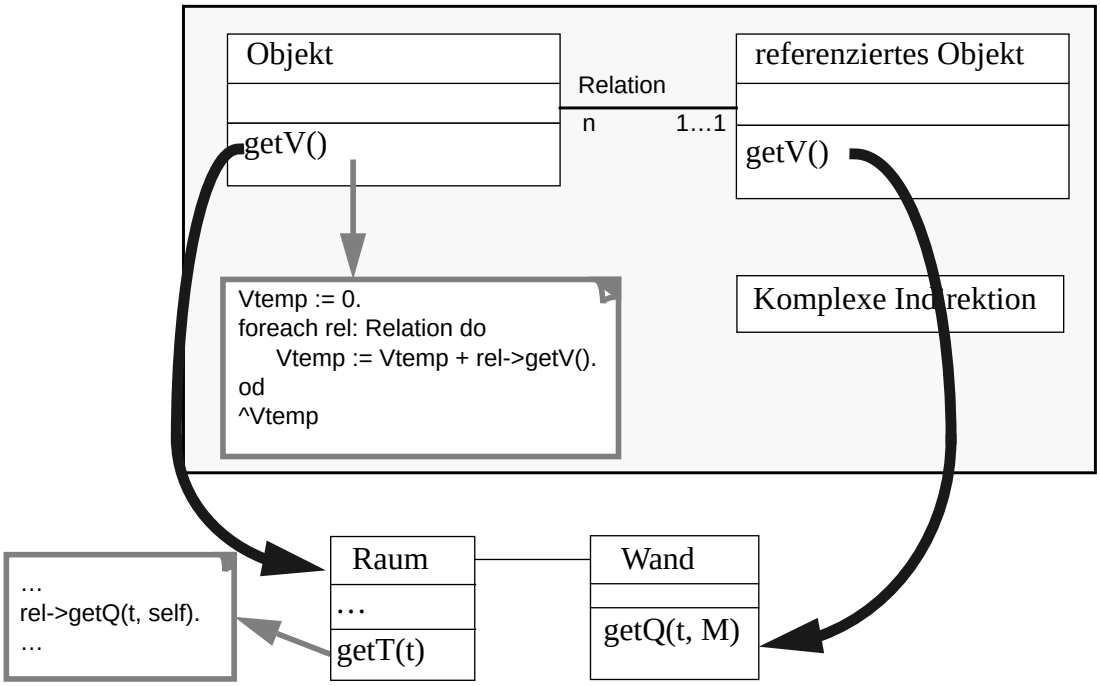


Abb. 21: Zusätzliche Aufrufparameter einführen

In dem Beispiel in Abbildung 21 soll die Funktion *getV()* des referenzierten Objektes auf die Funktion *getQ(t, M)* der Wand abgebildet werden. Als Parameter erwartet diese Methode den aktuellen Zeitpunkt der Berechnung und einen Zeiger auf das aufrufende Objekt (vgl. 2.7). Zur Bindung der Funktion *getV()* an die Funktion *getT(t)* des Raumes wurde bereits der Parameter t neu eingeführt. Dieser kann zum Aufruf von *getQ(t, M)* benutzt werden. Zusätzlich wird ein Zeiger auf den aktuellen Raum mit übergeben.

2.9 Implementierung der Patterns

Unter dem Begriff Implementierung wird im folgenden der Weg verstanden, wie man von den eher abstrakten Patterns zu einem konkreten Programm kommt. Prinzipiell gibt es mehrere Wege, die gebundenen Patterns in Quelltext zu verwandeln. Entweder wird für jede Funktion des Patterns spezieller Code generiert, oder es werden die Patterns selbst als eigenständige Klassen generiert und über Vererbung oder Delegation mit dem restlichen Modell verknüpft.

Für jeden beim Software-Entwurf im Modell benutzten Objekttyp wird im Programm eine eigene Klasse benötigt. Diese Modell-Klassen müssen zusätzlich zu ihrer Grundfunktionalität (z.B. Initialisierungsmethoden oder Ein-/Ausgabe) noch die Funktionalität erhalten, die sie bei der Modellierung durch die Bindung an Patterns zugewiesen bekommen haben. Die folgenden drei Abschnitte beschäftigen sich mit Möglichkeiten, diese Funktionalität zu implementieren.

2.9.1 Implementierung durch Delegation

Die Implementierung der Patterns über Delegation wird unter anderem in [Sou95] beschrieben. Jedes Pattern wird dabei im Programm durch eine eigene Klasse repräsentiert. Die Bindungen der Patterns an die Modellierungs-Objekte (in diesem Fall also an das Gebäude-Modell) erfolgen über Relationen. Der Vorteil dieser Implementierungstechnik ist, daß im fertigen Programm die Patterns und das Objektmodell nach wie vor getrennt vorliegen und dadurch unterscheidbar bleiben, und es kann auch sehr einfach erkannt werden, welches Objekt durch welches Pattern beeinflußt wird.
Bei dieser Implementierung der Patterns werden sämtliche Funktionsaufrufe an die entsprechenden Funktionen der Pattern-Klassen weitergeleitet. Dies ist in Abbildung 22 am Beispiel des Patterns *Komposition (72)* verdeutlicht.

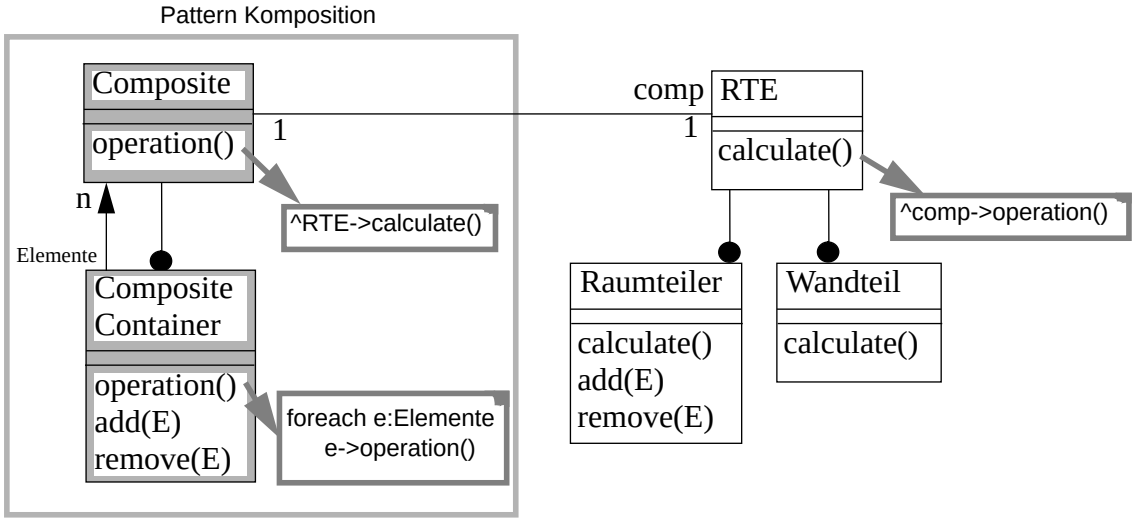


Abb. 22: Pattern-Implementierung durch Delegation

Mit dem Kompositions-Pattern können mehrere Objekte in einer baumartig strukturierten Menge gruppiert werden. In diesem Beispiel wurden mehrere Wandabschnitte zu einem Raumteiler zusammengefaßt. Der Aufruf der Methode *calculate()* eines Raumteilers soll an alle Wandteile dieses Raumteilers weitergeleitet werden.
In der Implementierung wird das Kompositions-Pattern durch die zwei Klassen *Composite* und *CompositeContainer* repräsentiert. Die Wände und Raumteiler sind über die 1:1 Relation *comp* mit diesen Klassen verbunden (Die Klasse Raumteiler agiert als Kontainer-Klasse, wird also mit Objekten aus der Klasse CompositeContainer verbunden). Die Funktion *calculate()* eines Raumteilerelementes (RTE) ist nun so implementiert, daß sie die Funktion *operation()* der Pattern-Klassen aufruft. Die *operation()* Methode eines CompositeContainers sorgt nun dafür, daß für sämtliche Objekte in dem Kontainer die jeweilige *operation()* Funktion aufgerufen wird. Für ein Composite-Objekt ruft *operation()* einfach die Funktion *calculate()* der Klasse RTE auf. Für die Blätter des Kontainer-Baumes (in diesem Fall also für die Klasse Wandteil) muß die Methode *calculate()* also überladen werden, um die gewünschte Funktionalität zu erreichen.
Wird jetzt *calculate()* bei einem Raumteiler aufgerufen, so wird der Aufruf an die Pattern-Klasse CompositeContainer weitergeleitet. Dort wird für alle Elemente, die sich in dem Con-

tainer befinden, die Methode *operation()* aufgerufen. Gelangt der Algorithmus an ein Blatt der Baumstruktur (also an ein Objekt der Klasse Composite), so wird wieder der Rückschritt von den Pattern-Klassen zu den ursprünglichen Modell-Klassen vorgenommen und die dort (für eine Wand) überladene Methode *calculate()* aufgerufen.
Die Vorteile dieses Verfahrens, die Patterns als eigenständige Klassen zu implementieren, sind hauptsächlich die gute Strukturierung des Programmes (jedes Pattern kann eindeutig den Modell-Klassen zugeordnet werden) und die Einfachheit der Implementierung (die Pattern-Klassen können fest vorgegeben werden, und es sind nur kleine Anpassungen an die Modell-Klassen notwendig). Sämtliche Funktionalität der Patterns ist bei diesem Ansatz in den Pattern-Klassen implementiert. Nur die Anpassung der Modell-Klassen an die Patterns muß noch (durch Überladen der entsprechenden Methoden) implementiert werden.
Ein Nachteil ist jedoch die komplizierte Aufrufhierarchie der Methoden. Es muß genau geplant werden, welche Methoden bei welchen Klassen überladen werden können oder müssen. Außerdem sind die Pattern-Klassen naturgemäß sehr allgemein gehalten. Da eventuell Objekte derselben Pattern-Klasse ihre Funktionalität für unterschiedliche Modell-Objekte zur Verfügung stellen müssen, können Optimierungen für spezielle Modell-Klassen nicht durchgeführt werden.

2.9.2 Implementierung durch Vererbung

Eine ähnliche Implementierungsmethode der Patterns funktioniert mit Hilfe des Vererbungsmechanismus. Dabei sind die Patterns im Programm auch als eigenständige Klassen vorgegeben, werden jedoch nicht über Relationen, sondern über Vererbung an die Modell-Klassen gebunden.

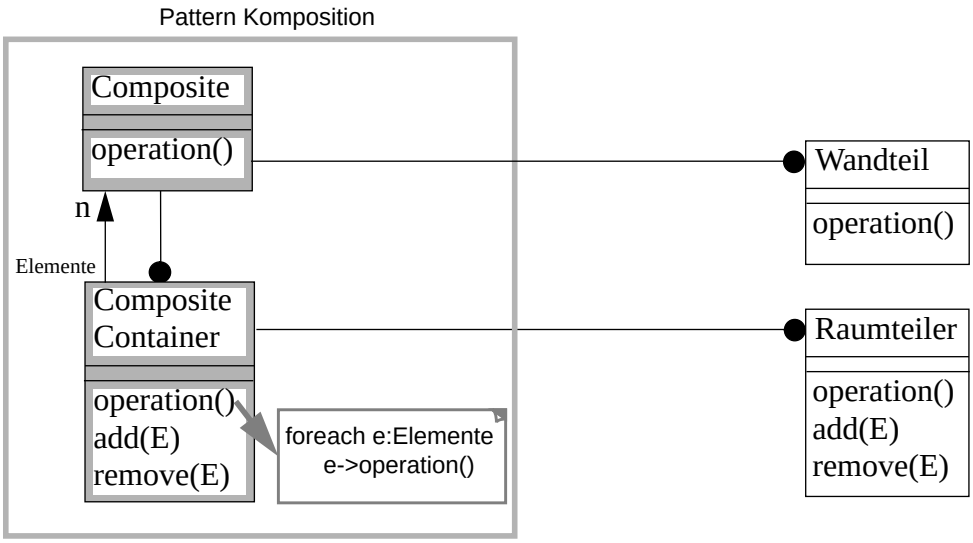


Abb. 23: Pattern-Implementierung durch Delegation

Wie in Abbildung 23 dargestellt, erben die Klassen Wandteil und Raumteiler ihre Funktionalität von den entsprechenden Pattern-Klassen. Dadurch wird die Aufruf-Hierarchie der Methode *operation()* wesentlich einfacher. Die Klasse Wandteil überläd nach wie vor diese Funktion, damit sie die gewünschte Aufgabe durchführt.

Nachteilig bei dieser Implementierungs-Methode ist, daß die im Pattern vorgegebenen Funktionen nicht umbenannt werden können. Dadurch wird der Quelltext der Modell-Klassen unleserlich. Ein zweites Problem ist, daß bei diesem Ansatz sehr leicht Mehrfach-Vererbung auftreten kann. Diese ist in Smalltalk nicht vorhanden und bereitet auch in anderen Programmiersprachen wie C++ Probleme.
Generell muß bei dieser Implementierungs-Methode sehr darauf geachtet werden, welche Objekte von welchen anderen Objekten Eigenschaften erben. Bestehen im ursprünglichen Modell bereits Vererbungs-Hierarchien (was ja auch durchaus sinnvoll zur Beschreibung dieser Modelle ist), so können durch die Patterns Mehrfachvererbung oder im Extremfall sogar zyklische Vererbungen eingeführt werden. Ob diese Implementierungstechnik eingesetzt werden kann, muß also im Einzelfall genau überdacht werden.

2.9.3 Implementierung durch spezielle Generatoren

Die weitaus flexibelste Methode Patterns zu implementieren besteht darin, für jedes Pattern speziell auf die Modell-Klassen angepaßten Code zu generieren. Ein Ansatz wie das geschehen kann wird in [BFV96] beschrieben[6]. Verwendet man spezielle Generatoren, so tauchen die Patterns nicht mehr (unbedingt) als eigenständige Klassen im Programm-Code auf, sondern sind vielmehr in die Modell-Klassen eingebettet. Dadurch geht der direkte Zusammenhang zwischen den Patterns und den an sie gebundenen Modell-Klassen etwas verloren, dafür werden eine große Flexibilität und Optimierungsmöglichkeiten gewonnen.
So kann dasselbe Pattern für unterschiedliche Modell-Klassen auf verschiedene Arten implementiert werden, um Geschwindigkeits-Optimierungen durchzuführen oder einfach nur besser an das Modell angepaßten Code zu erhalten.
Die Aufruf-Hierarchie der Methoden bleibt bei diesem Ansatz übersichtlich und einfach. Nachteilig ist nur, daß bei mehrfacher Verwendung desselben Pattern jedesmal wieder derselbe (oder zumindest ähnlicher) Code im Programm steht. Dies führt zu größeren Programmen und dadurch höheren Speicherplatzverbrauch.
Voraussetzung für eine sinnvolle Anwendung dieses Implementierungs-Ansatzes ist ein Programm-Generator, der die Modellierung in ein fertiges Programm übersetzen kann. Der Generator kopiert dazu bereits im Pattern vorgegebenen Programm-Code und führt dabei Anpassungen an die jeweilige Modell-Klasse durch.

Jede der drei hier vorgestellten Implementierungstechniken für Pattern hat ihre Vor- und Nachteile auf konzeptioneller oder Code-Ebene. Ein „intelligenter“ Generator könnte im Einzelfall entscheiden, welche Methode angewendet werden soll. Dabei kann je nach Programmiersprache, Pattern und benutztem Modell eine andere Technik verwendet werden.

6. In dieser Arbeit wird ein Werkzeug beschrieben, mit dem die Patterns aus [GHJ95] instanziieren kann. Dabei wird allerdings jedes Pattern für sich alleine betrachtet und der erzeugte Code muß von Hand zu dem Gesamtprogramm zusammenkopiert werden.

Kapitel 3

Simulation

3.1 Simulationsmethoden

Es gibt eine Vielzahl von Einsatzgebieten für Simulatoren. Je nach dem, was mit Hilfe eines Simulators untersucht werden soll, eignet sich die eine oder andere Simulationsmethode besser, um die zu Grunde liegenden physikalischen Größen zu untersuchen. Die folgende Abbildung ist aus [MaM89] entnommen und zeigt eine Klassifikation unterschiedlicher Simulationsmethoden.

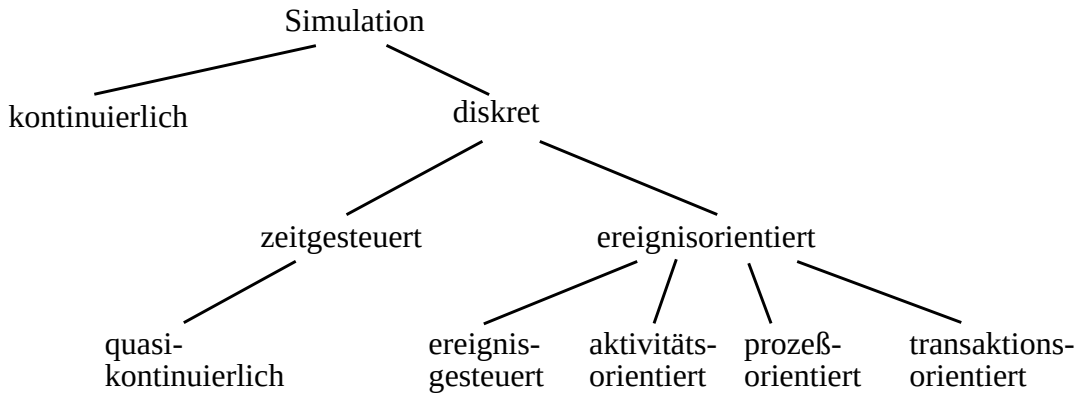


Abb. 24: Klassifikation von Simulationsmethoden

Bei der kontinuierlichen Simulation wird davon ausgegangen, daß sich der Zustand des Systems kontinuierlich im Laufe der Zeit ändert. Das bedeutet, daß alle Simulationsgrößen jederzeit neuberechnet werden müssen.
Im Gegensatz dazu finden bei der diskreten Simulation Änderungen der Simulationsgrößen nur zu bestimmten, diskreten Zeitpunkten statt. Die Änderung einer Simulationsgröße wird durch ein Ereignis (Event) hervorgerufen. Diese Events sind bei der ereignisorientierten Simulation atomar, daß heißt, sie finden zu einem bestimmten Zeitpunkt statt und „verbrauchen“ keine eigene Zeit. So ein Ereignis könnte zum Beispiel sein: „Tür 2 wird um 12:15 Uhr geöffnet“. Direkt nach 12:15 Uhr ist die Tür dann offen und kann entsprechend behandelt werden. Da sämtliche Ereignisse atomar sind, braucht die Zeit zwischen zwei aufeinander folgenden Events nicht betrachtet zu werden (während dieser Zeit kann ja keine Zustandänderung statt-

finden). Ein ereignisgesteuerter Simulator bearbeitet also immer den Event, der als nächstes gerechnet werden soll. Löst die Bearbeitung dieses Events weitere Events aus (beispielsweise kann das Öffnen einer Tür die Auslösung der Alarmanlage zur Folge haben), so werden diese in die Liste aller noch ausstehender Events einsortiert. Ist ein Event abgearbeitet, so wird die Simulationsuhr einfach auf den nächsten Event vorgestellt und dieser wird dann als nächstes bearbeitet. Die einzelnen Ausprägungen der ereignisorientierten Simulation (ereignis-, aktivitäts-, prozeß- oder transaktionsgesteuert) unterscheiden sich darin, wie die einzelnen Events aussehen und unter welchen Bedingungen sie den Zustand des Systems manipulieren. Näheres dazu steht in [MaM89].

Der Simulationsvorgang für den Gebäude-Simulator ist prinzipiell kontinuierlicher Art da kontinuierlichen Simulationsgrößen wie „Raumtemperatur“ oder „Luftfeuchtigkeit“ auftreten. Bei Digitalrechnern ist eine solche kontinuierliche Berechnung allerdings nicht möglich. Da jeder Computerbefehl zu einem diskreten Zeitpunkt stattfindet, können die Simulationsgrößen auch nur zu diskreten Zeitpunkten aktualisiert werden. Zur Aktualisierung wird jeweils die Änderung der Simulationsgröße seit der letzten Berechnung bestimmt. Werden diese Zeitintervalle genügend klein gewählt, kann eine gute Approximation an den realen Verlauf erreicht werden.

Diese Art der Simulation heißt quasi-kontinuierlich. Prinzipiell werden dabei die Simulationsobjekte zu diskreten Zeitpunkten berechnet, allerdings werden sie häufig genug aktualisiert, so daß jederzeit eine brauchbare Annäherung an den realen Wert zur Verfügung steht. Abhängigkeiten zwischen einzelnen Simulationsgrößen müssen teilweise vereinfacht werden, um sie berechnen zu können.

Beispielsweise hängt die Raumtemperatur von der Wärmemenge ab, die in den Raum einströmt. Die Transmissionswärmemenge, die durch eine Wand fließt, ist umgekehrt aber abhängig von den Temperaturen der angrenzenden Räume[7]. Um solche zweiseitigen Abhängigkeiten berechnen zu können, muß eine der beiden Berechnungsformeln vereinfacht werden. So kann zum Beispiel zur Berechnung der Transmission durch eine Wand angenommen werden, daß sich die Temperaturen der angrenzenden Räume unabhängig von der Transmissionswärme verändert haben. Durch diesen Trick kann zu diskreten Zeitpunkten die aktuelle Transmissionswärmemenge auf Grund von den letzten bekannten Raumtemperaturen berechnet werden. Ebenso wird zu (eventuell unterschiedlichen) Zeitpunkten die Raumtemperatur auf Grund der im letzten Zeitintervall berechneten Wärmemenge aktualisiert. Könnte man bei dieser Art der Berechnung die Zeitintervalle, zu denen Neuberechnungen stattfinden, unendlich klein machen, so wären die berechneten Ergebnisse immer korrekt. Die Hoffnung ist nun, daß solcherlei Vereinfachungen auch schon bei „hinreichend kleinen“ Zeitintervallen Ergebnisse innerhalb gewisser Toleranzgrenzen liefern. Bei der Raumtemperatur ist es wahrscheinlich überflüssig, die Ergebnisse im Milligrad-Bereich genau zu berechnen.

Welches Zeitintervall „hinreichend klein“ für eine bestimmte Simulationsgröße ist, hängt sehr stark von den Simulationsobjekten selbst ab. Eine häufige Neuberechnung der Simulationsgrößen führt zu einem erheblichen Rechenaufwand, liefert dafür aber genauere Ergebnisse.

---

7. Es gilt für die Raumtemperatur $T_{i+1} = T_i + \int_{t_{last}}^{t_{akt}} \frac{Q(t)}{v \cdot c(t)} dt$ und $Q(t) = \frac{\Delta T(t)}{R}$, siehe Patterns *Simulation thermischer Masse (45)* und *Simulation thermischer Verbindung (49).*

Bei zu großen Berechnungsintervallen kann daß System leicht ins Schwingen geraten und falsche Ergebnisse liefern. Um das zu verhindern, können Schranken vorgegeben werden, wie groß ein Berechnungsintervall maximal werden darf.
Wird Hardware in den Simulator integriert, so muß die Simulation zusätzlich synchron zur Echtzeit ablaufen. Bei einer reinen Software-Simulation kann auch in komprimierter Echtzeit gerechnet werden. Durch das hier verwendete eventbasierte Scheduling können diese beiden Bedingungen eingehalten werden.
Ein eigener Programmteil, der Scheduler, kümmert sich darum, daß alle Berechnungen zum richtigen Zeitpunkt stattfinden. Dazu wird, ähnlich wie bei der ereignisorientierten Simulation, eine Event-Liste verwaltet. Das Ereignis, daß eine Simulationsgröße neu berechnet werden muß, ist ein besonderer Event, der zum richtigen Zeitpunkt beim Simulationsobjekt eintreffen muß. Durch die Verwendung von Event-Listen können also kontinuierliche Simulationsgrößen angenähert und auch atomare Ereignisse berücksichtigt werden.

3.2 Simulator-Modell

Die eigentliche Funktionalität des „Simulierens“, das heißt, der quasi-kontinuierlichen Berechnung der einzelnen Simulationsgrößen, wird durch ein gesondertes Modell ausgedrückt. Dieses Simulator-Modell ist prinzipiell unabhängig von einem bestimmten Simulator (zum Beispiel einem Simulator für Gebäude) und kann so sehr flexibel eingesetzt werden. Das Simulator-Modell besteht aus 6 Klassen mit bereits fertig vorgegebenen Funktionen (siehe [Hei96] und Abbildung 25).

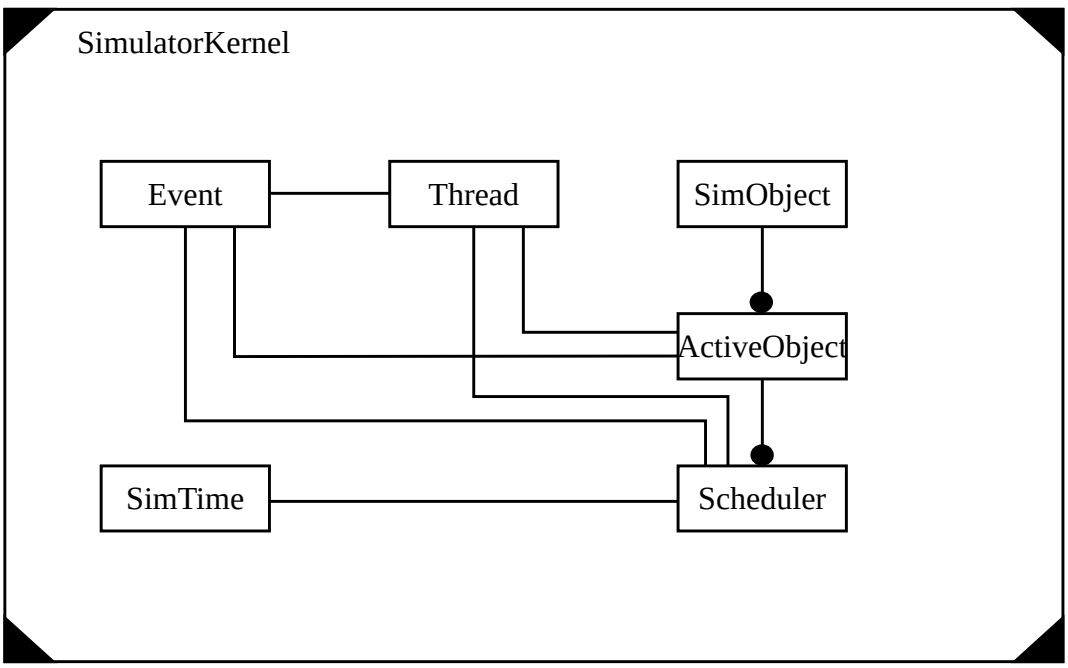


Abb. 25: Simulator-Modell

Um diese Funktionen für einen speziellen Simulator nutzen zu können, werden einige Klassen aus diesem Modell mit entsprechenden Patterns (*Kontinuierliche Simulation (53)* und *Aktuator (55)*) referenziert und dadurch mit in das Applikationsmodell des zu entwerfenden Simulators aufgenommen. Zusätzlich wird automatisch ein Scheduler (also eine Instanz der Klasse Scheduler) erzeugt, der sich um die Zuteilung von Rechenzeiten an die Simulationsobjekte kümmert. Der Scheduler hat die Aufgabe, die einzelnen Simulationsobjekte zu bestimmten Zeitpunkten aufzufordern, sich selbst (das heißt ihre Simulationsgrößen) neu zu berechnen, und er kümmert sich zudem um die Einhaltung der Echtzeit, das heißt, er überprüft Deadlines und kann Prioritäten für unterschiedliche Ereignisse berücksichtigen. Aus der Sicht der Simulationsobjekte braucht die genaue Arbeitsweise des Schedulers also gar nicht bekannt zu sein - sie stellen nur ihre Berechnungsformeln zur Verfügung und diese werden dann vom Scheduler aufgerufen. Mit dem Pattern *Kontinuierliche Simulation (53)* kann dafür gesorgt werden, daß in regelmäßigen Abständen ein Simulationsobjekt zur Neuberechnung aufgefordert wird.
Die Kommunikation der Simulationsobjekte untereinander und auch die Kommunikation mit dem Scheduler funktionieren über Events. Ein Event ist eine Nachricht, die zwischen zwei Objekten verschickt wird und die in einem bestimmten Zeitintervall beim Zielobjekt ankommen muß. Der Inhalt einer solchen Nachricht kann zum Beispiel sein, eine Simulationsgröße neu zu berechnen. Der Scheduler ist dafür verantwortlich, daß alle Events pünktlich bei ihren Zielobjekten ankommen. Kann diese Zeitplanung nicht eingehalten werden (dadurch, daß die vorgesehenen Ankunftszeiten zu knapp bemessen sind oder daß zu viele Events gleichzeitig verschickt werden sollen), so gibt der Scheduler eine Fehlermeldung aus und kann in Spezialfällen eine Fehlerbehandlung starten (eventuell können einige Events gelöscht oder verzögert werden). Für die Echtzeitfähigkeit des Simulators ist also der Scheduler (relativ unabhängig von den Simulationsobjekten) verantwortlich. Er steuert die einzelnen Berechnungen so, daß sie zum angeforderten Zeitpunkt stattfinden. Wird zu einem Zeitpunkt keine Berechnung gefordert, so wartet der Scheduler solange, bis das nächste Event eintrifft oder versendet werden soll.
In Abbildung 26 ist die Funktionsweise des Schedulers skizziert. Der Scheduler verwaltet eine sortierte Liste aller Events, die versendet werden müssen. Jeweils das Event, das zum frühesten Zeitpunkt eintreffen soll, steht in dieser Warteschlange ganz oben. Solange die vorgesehene Ankunftszeit dieses Events noch nicht erreicht ist, wartet der Scheduler entsprechend lange. Dazu wird ein Timer gestellt und nach Ablauf des Timers wird das oberste Event in der Warteschlange verschickt. Wird der Scheduler aufgefordert, ein neues Event zu verschicken, so wird der Wartezustand unterbrochen, das neue Event in die Warteschlange einsortiert und

der Timer neu gestartet. Dadurch wird jeweils das Event mit der frühesten Ankunftszeit zuerst versendet (shortest deadline first).

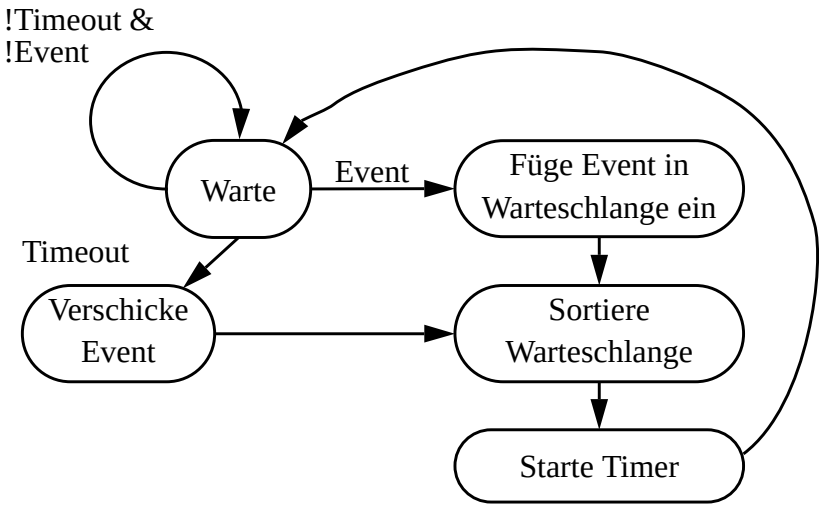


Abb. 26: Zustandsdiagramm des Schedulers

3.2.1 Events

Ein Event wird versendet, indem eine Instanz der Klasse Event angelegt wird. Beim Anlegen dieses Objektes werden alle relevanten Daten angegeben. Dazu zählen der Absender und das Zielobjekt, der Typ des Events und zusätzliche übergebbare Daten, sowie Timestamp und Deadline, also die Zeitpunkte, zu denen das Event ankommen soll und bis wann es vollständig bearbeitet sein muß. Um solch ein Event empfangen zu können, muß das Simulationsobjekt Funktionalität aus der Klasse ActiveObject erben. Dieses führt eventuell zu der Notwendigkeit, Mehrfachvererbung zu benutzen. Da jedoch Mehrfachvererbung in Smalltalk problematisch ist, wird hier ein anderer Weg gegangen. Für jedes Objekt, das Events senden und empfangen will, wird eine Instanz der Klasse ActiveObject angelegt, die dann vom Ursprungs-Objekt referenziert wird. Sämtliche simulationsrelevanten Befehle werden an dieses Objekt delegiert.
Bei der Modellierung eines Gebäude-Simulators muß keine besondere Rücksicht auf die Struktur oder Implementierung des Schedulers oder der ActiveObjects genommen zu werden. Um beispielsweise die Temperatur eines Raumes alle 10 Sekunden neu zu berechnen, kann einfach das Pattern *Kontinuierliche Simulation (53)* an den Raum gebunden werden. Dadurch wird für jeden Raum ein ActiveObject angelegt, das dafür sorgt, daß die Funktion zur Berechnung der Raumtemperatur rechtzeitig aufgerufen wird. Die Periodendauer wird bei der Instanziierung der einzelnen Objekte eingetragen und kann bei Bedarf auch dynamisch angepaßt werden[8]. Um die quasi-kontinuierliche Simulation durchzuführen, wird ein neuer Thread angelegt, der auf das Eintreffen eines Events zur Neuberechnung der Temperatur wartet. Trifft

8. Die Periodendauer wird als Instanzenvariable bei den Simulationsobjekten angelegt (siehe Pattern *Kontinuierliche Simulation (53)*). Diese kann (zum Beispiel durch das *Aktuator (55)* Pattern) beeinflußt werden. Außerdem kann durch das Versenden eines entsprechenden Events das Simulationsobjekt unabhängig von seiner Periodendauer aufgefordert werden, sich neu zu berechnen.

dieses Event ein, so wird die entsprechende Berechnungsformel beim Raum aufgerufen. Zur Neuberechnung der Raumtemperatur kann es notwendig sein, die Werte von abhängigen Simulationsobjekten wie zum Beispiel von Wänden oder Heizkörpern abzufragen. Dieses kann direkt durch Prozeduraufrufe bei den abhängigen Objekten geschehen. Müssen bei den abhängigen Simulationsobjekten vorher noch aufwendige Berechnungen durchgeführt werden, so ist es sinnvoller, diese unabhängig durchzuführen und die Abfrage der berechneten Werte durch schnelle Funktionsaufrufe (eventuell gekoppelt mit einer Zwischenpufferung) durchzuführen, damit keine Deadline-Verletzungen auftreten. Um die Berechnung der abhängigen Simulationswerte kümmern sich die entsprechenden Simulationsobjekte selbst. Während der Berechnung der Simulationsgrößen kann der Scheduler (quasi-)parallel dazu weiterarbeiten, da er in einem separaten Prozeß läuft. Abbildung 27 zeigt noch einmal die Aufrufhierarchie zur Event-Verarbeitung.

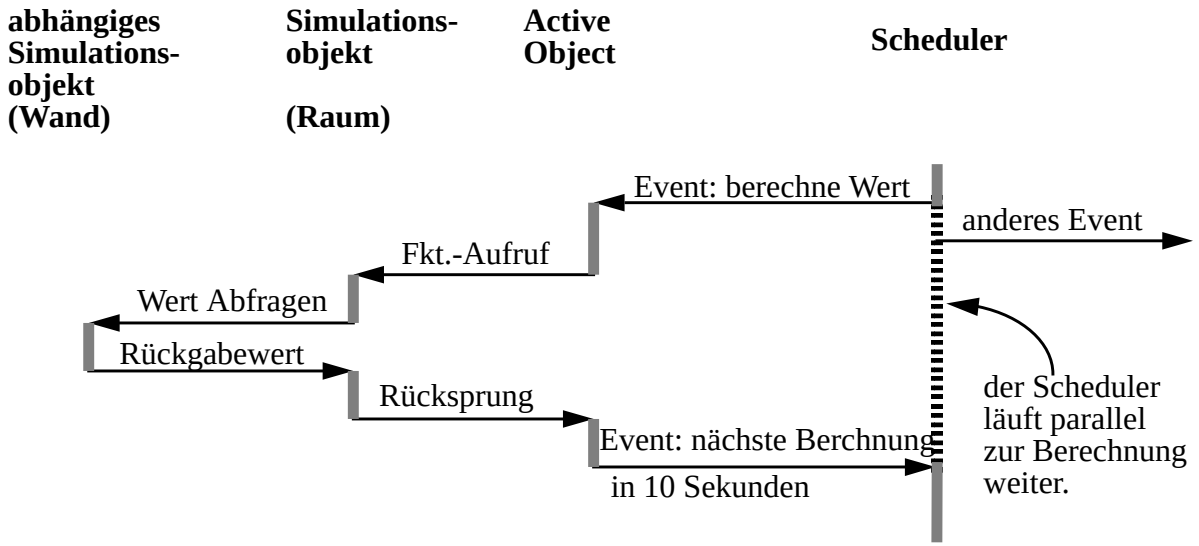


Abb. 27: Aufrufreihenfolge bei der Berechnung von Simulationsgrößen

3.2.2 Steuerung des Simulators

Bisher wurde noch nicht behandelt, wie der Simulator überhaupt vom Benutzer gesteuert werden kann und wie Simulationsgrößen für einen Benutzer visualisiert werden können.
Die Steuerung des Simulators ist prinzipiell von der Modellierung desselben unabhängig. Für den Simulators ist es unwichtig, ob und wie die berechneten Simulationsgrößen angezeigt werden oder durch welche Aktionen er stimuliert wird.

Um den Wert einer Simulationsgröße abzufragen, genügt es, bei den Simulationsobjekten eine entsprechende Methode aufzurufen. Ebenso kann der Simulator durch das Setzen von Simulationsgrößen stimuliert werden. Die Simulatorsteuerung und -anzeige kann also „von außen“ das eigentliche Simulations-Modell beeinflussen, ohne daß dieses bemerkt, wodurch es beeinflußt wird. Das Simulations-Modell kann also zunächst vollkommen unabhängig von seiner Steuerung entworfen werden. Die Steuerung wird erst zum Schluß an das Modell angepaßt.

Die Modellierung einer Simulator-Steuerung kann sogar auf die einfache Konfigurierung vorgefertigter Visualisierungs- und Eingabe-Komponenten reduziert werden[9]. Diese Komponenten werden einfach mit den Simulationsobjekten verknüpft und arbeiten dann anschließend eigenständig und unabhängig von der Berechnung.

3.3 Pufferung

Durch die quasi-kontinuierliche Simulation werden Simulationsgrößen, die sich eigentlich kontinuierlich verändern, nur noch zu diskreten Zeitpunkten berechnet. Um die dadurch entstehenden Rechenfehler so klein wie möglich zu halten, muß die korrekte zeitliche Änderung dieser Größen so genau wie möglich (bzw. so genau wie nötig) approximiert werden.
Dazu werden die berechneten Werte der Simulationsgrößen mit ihren Berechnungszeitpunkten zwischengespeichert. Ein spezielles Puffer-Objekt kann dann automatisch Zwischenwerte interpolieren oder Integrale bilden. Je nach dem, wie genau ein Wertverlauf angenähert werden muß eignen sich unterschiedliche Puffer zur Interpolation bzw. Integration. Im einfachsten Fall reicht es, sich den zuletzt berechneten Wert zu merken. Aber auch aufwendigere Verfahren wie die Integration mit der Simpson-Regel sind eventuell notwendig.

9. Eine Bibliothek solcher Anzeige- und Eingabekomponenten wurde bereits in der AG „VLSI Entwurf und Architektur“ der Universität Kaiserslautern von Daniel Bolender implementiert.

Um eine einfache Austauschbarkeit zwischen verschiedenen Puffern zu erhalten, sind mehrere Puffer und die dazugehörigen Berechnungsmethoden in einem eigenen Modell zusammengefaßt (siehe Abbildung 28).

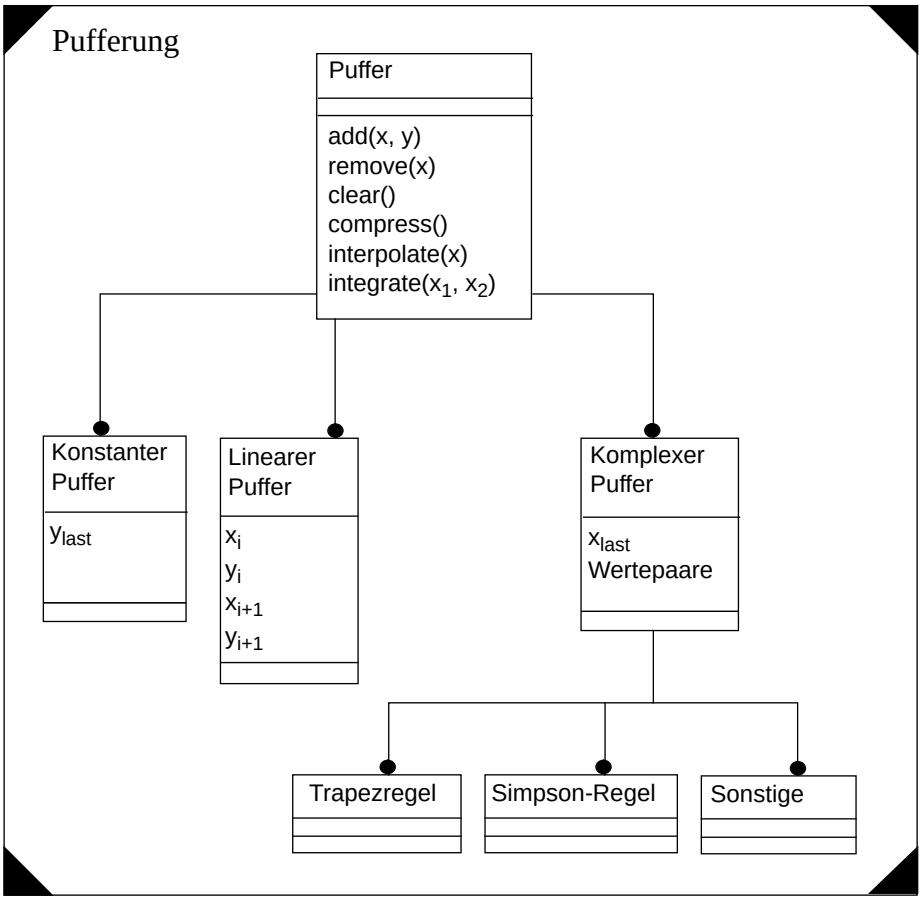


Abb. 28: Klassendiagramm der Puffer

Sämtliche Puffer haben dieselbe Funktionsschnittstelle. Es können neue Werte in den Puffer aufgenommen werden, der Puffer kann gelöscht werden, und man kann Integrale über einem Intervall oder Funktionswerte an bestimmten Punkten berechnen. Unterschiedlich sind die Puffer nur in ihrer Implementierung. Hauptsächlich unterscheiden sie sich in ihrer Genauigkeit und der Berechnungsgeschwindigkeit. Einfachere Algorithmen sind schneller als aufwendige, die dafür in der Regel genauere Ergebnisse liefern. Durch die Aufstellung eines eigenen Puffer-Modells (im Gegensatz zur Kapselung der Puffer in ein spezielles Pattern) kann sehr leicht mit unterschiedlichen Puffern experimentiert werden. Sogar unterschiedliche Puffer für Objekte derselben Klasse sind möglich.

Kapitel 4 Ausblick

4.1 Zusammenfassung

Die Modellierung eines Gebäude-Simulators erfordert eine flexible, ausdrucksstarke aber dennoch einfach benutzbare Entwurfsmethode. Patterns scheinen hier ein geeignetes Mittel zu sein, um Entwerfern, die sich nicht im Bereich der Simulation auskennen, Methoden an die Hand zu geben, mit denen ein Simulator entwickelt werden kann.
Einem Entwerfer werden dazu bereits vorgefertigte Modelle bereitgestellt, die jeweils einen kleinen Ausschnitt aus dem Gebäude-Simulator beschreiben. Zu diesen Modellen gehört das Gebäude-Modell, das den prinzipiellen Aufbau eines Gebäudes und den Zusammenhang zwischen den einzelnen Gebäude-Teilen beschreibt, das Simulator-Modell, in dem der Kern eines Simulators zur quasi-kontinuierlichen Simulation vorgegeben ist, sowie möglicherweise weitere Modelle (z.B. zur Pufferung, vgl. 3.3). Diese Modelle beschreiben recht allgemein jeweils einen Teil des Gesamtsystems. Durch spezielle Patterns können nun einzelne Teile dieser Modelle zusammengebracht und mit zusätzlicher Funktionalität versehen werden.
Jedes Pattern beschreibt die Lösung eines Problems, das bei der Gebäude-Simulation auftreten kann. Durch die Bindung der Patterns an einzelne Modell-Klassen werden diese um die entsprechende Funktionalität erweitert. Am Ende kommt dabei ein komplexes Applikationsmodell heraus, das den Simulator darstellt. Durch geeignete Generatoren kann aus diesem Modell ein fertiges Programmpaket erzeugt werden. Dazu muß nur die Funktionalität der einzelnen Patterns auf geeignete Weise in Programm-Fragmente umgesetzt werden (siehe 2.9).
Was nach dieser Modellierung auf Klassen-Ebene noch fehlt, ist eine konkrete Instanziierung der Klassen, um einzelne Objekte zu erhalten, die simuliert werden sollen. Dieser Instanziierungs-Schritt kann (größtenteils) automatisch erfolgen, wenn die einzelnen Objekte bereits in einem an das Gebäude-Modell angepaßten Format vorliegen. Es ist geplant, die zu simulierenden Häuser in der CAD-Software Speedikon[10] einzugeben. Ein geeigneter Transformator kann die in Speedikon vorliegenden Daten an das Gebäude-Modell anpassen. Da das Applikationsmodell auf das Gebäude-Modell aufsetzt, kann so also automatisch ein Großteil der zur Simulation benötigten Daten gewonnen werden. Der gesamte Entwurfsprozeß ist in Abbildung 29 zusammengefaßt.

10. Speedikon ist ein eingetragenes Warenzeichen der Firma IEZ AG.

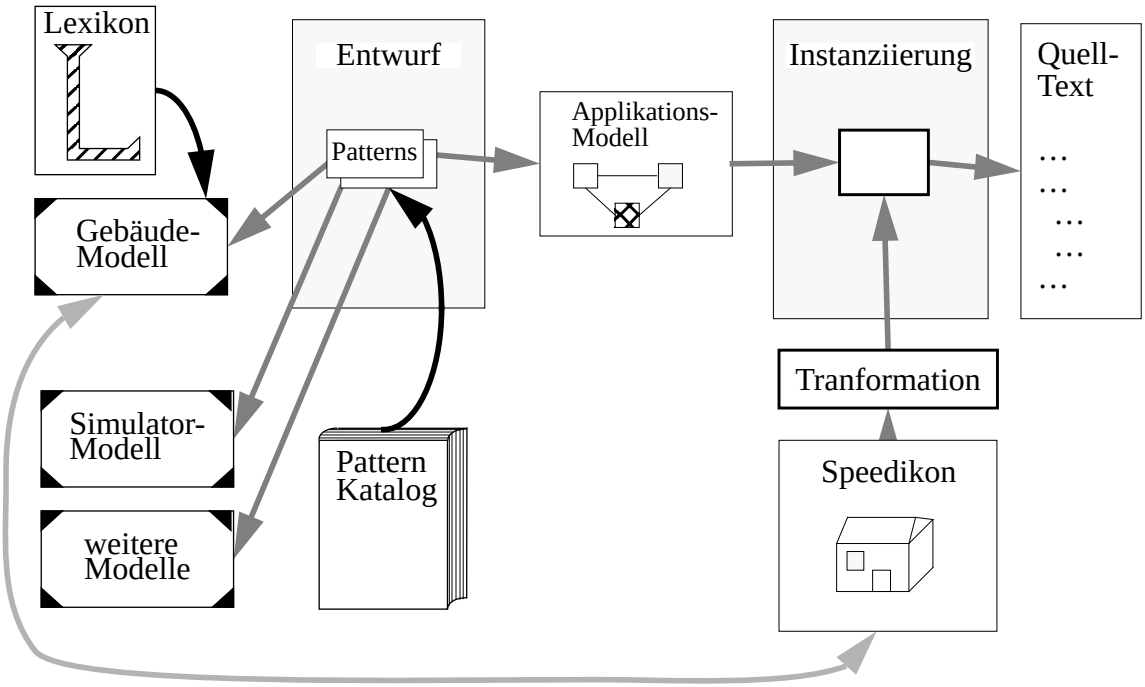


Abb. 29: Entwurfsprozeß mit Patterns

4.2 Vor- und Nachteile des patternbasierten Entwurfs

Durch die Patterns erhält man eine sehr flexible und ausdrucksstarke Modellierungsmöglichkeit. Da jedes Pattern eine Problemlösung für einen Teilbereich des zu modellierenden Systems liefert, kann das Modell Schritt für Schritt durch sukzessives Anwenden der Patterns aufgebaut werden. Vorteilhaft an den Patterns ist dabei, daß sie neben der Problemlösung auch ausführlich die Problemsituation beschreiben, bei denen sie verwendet werden können. Das erleichtert den Umgang mit Patterns und ermöglicht auch Nichtspezialisten auf dem Gebiet der Gebäudesimulation, einen Simulator zu erstellen.
Sind die Patterns einem Entwerfer bekannt, so können sie gut als Entwurfsvokabular benutzt werden. Dies erleichtert die Kommunikation zwischen mehreren Entwerfern. Durch einfache Angabe eines Pattern-Namens wird bereits eine bestimmte Problem-Situation beschrieben und ein Lösungsvorschlag gemacht.
Ein weiterer Vorteil der Modellierung mit Patterns ist, daß andere Modelle und Ideen verwendet werden können. So können separat vom Gebäude-Simulator bereits Modelle aufgestellt werden, die den Aufgabenbereich charakterisieren. Das Gebäude-Modell ist beispielsweise solch ein Modell. In ihm ist beschrieben, wie ein Haus aufgebaut ist und welche Eigenschaften es hat. Dieses Modell kann in vielen Bereichen zum Einsatz kommen. Beispielsweise kann eine Steuerung oder ein CAD-Programm genauso auf das Gebäude-Modell aufbauen wie der Gebäude-Simulator. Daher ist es sinnvoll, solcherlei Modelle separat zu modellieren und dann für das aktuelle Anwendungsprojekt adäquat zu verwenden. Der hier vorgeschlagene Modellierungsansatz kann verschiedene Modelle integrieren und fügt durch die Patterns zusätzliche Funktionalität hinzu.

Da die Patterns problemorientiert arbeiten, eignen sie sich auch sehr gut zur Dokumentation des erstellten Modells. Durch einen speziellen Generator könnte aus einem Applikationsmodell automatisch eine Dokumentation erstellt werden, in der beschrieben wird, wie die einzelnen Teilprobleme beim Entwurf gelöst wurden.
Voraussetzung für einen guten Entwurf mit Patterns sind natürlich gute Patterns. Die Patterns müssen einerseits so flexibel sein, daß sie an mehreren Stellen eingesetzt werden können, und andererseits muß jedes Pattern auch konkret genug sein, damit aus ihnen automatisch Code generiert werden kann. Dazu kommt noch, daß genug Patterns vorhanden sein müssen, um eine durchgängige Modellierung mit Patterns zu ermöglichen. Zu viele Patterns wiederum verkomplizieren den Simulator-Entwurf, da länger nach einem anwendbaren Pattern gesucht werden muß.
Um die Modellierung mit Patterns einfach und übersichtlich zu gestalten, sollte sie von entsprechenden Werkzeugen unterstützt werden. Ein spezieller Editor könnte die Auswahl der benötigten Patterns erleichtern, und mit ihm könnten auch die Patterns an Modellklassen gebunden werden. Zusätzlich sind mit dem Editor Konsitenzüberprüfungen möglich.
Auf jeden Fall muß die Modellierung in computer-lesbarer Form vorliegen, um eine automatische Codegenerierung zu ermöglichen.

4.3 Weitere Arbeiten

Der im Anhang A vorgestellte Pattern-Katalog ist sicherlich nicht vollständig. Er kann zur Modellierung einfacher Simulatoren herangezogen werden, muß aber beim Entwurf konkreter Gebäudesimulatoren noch angepaßt und erweitert werden. Derzeit wird im Rahmen einer Projektarbeit überprüft, wie praktikabel der hier vorgestellte Modellierungsansatz ist und ob die vorgestellten Patterns ausreichen, um einen einfachen Gebäudesimulator zu generieren.

Weiterhin ist die Verwendung der Patterns, vor allem unter dem Gesichtspunkt der *automatischen* Generierung eines Programmes, nur mit Hilfe geeigneter Werkzeuge möglich. Was fehlt ist also eine passende Repräsentation der Patterns in einem Computerprogramm, das die Bindung der Patterns an Modellklassen, wie in Kapitel 2.8 beschrieben, erlaubt. Auf dieses Modellierungs-Programm muß dann ein Generator aufgesetzt werden, der anhand der vorgenommenen Pattern-Bindungen und aufgrund entsprechenden Hintergrundwissens in der Lage ist, einen Gebäude-Simulator zu generieren.
Ein solcher Generator wird momentan in einer Diplomarbeit entwickelt. Dazu werden die Patterns selbst und die Bindungen der Patterns weiter formalisiert. Der Generator liest eine formale Beschreibung der Pattern-Bindungen ein und generiert daraus, zusammen mit den Eingabemodellen, den fertigen Simulatorcode. Der Generator selbst ist dabei generisch aufgebaut, so daß er Code für unterschiedlichste Patterns generieren kann und leicht erweiterbar ist.

Ein längerfristiges Ziel ist es, eine integrierte Entwicklungsumgebung zu erstellen, mit der Modelle und Patternbindungen eingegeben werden können, um anschließend daraus einen Gebäudesimulator zu generieren.

Anhang A Pattern Katalog

Im folgenden werden Patterns zur Modellierung eines Gebäude-Simulators vorgestellt. Der Katalog ist in seiner jetzigen Form sicherlich nicht vollständig und muß bei Bedarf noch erweitert und angepaßt werden.
Der Katalog ist in zwei Teile aufgeteilt: Kapitel A.1 beinhaltet Patterns, die sich speziell mit der Simulation beschäftigen. Im anschließenden Kapitel werden dann allgemeinere Patterns beschrieben, die sich um das Zusammenspiel der einzelnen Komponenten eines Frameworks kümmern.

A.1 Simulations-Patterns

Simulations-Patterns kümmern sich zum einen um die korrekte Berechnung der einzelnen Simulationsgrößen (Kapitel A.1.1) und zum anderen um Probleme des Scheduling (A.1.2). Bei der Modellierung eines Simulators sollte sich zunächst um die Modellierung der Simulationsobjekte (also um die Berechnungsformeln und um das Zusammenspiel unterschiedlicher Objekte) gekümmert werden. Die zeitliche Abfolge, wann welches Objekt berechnet werden soll, kann in einem zweiten Schritt einfach hinzumodelliert werden.

A.1.1 Berechnungsformeln

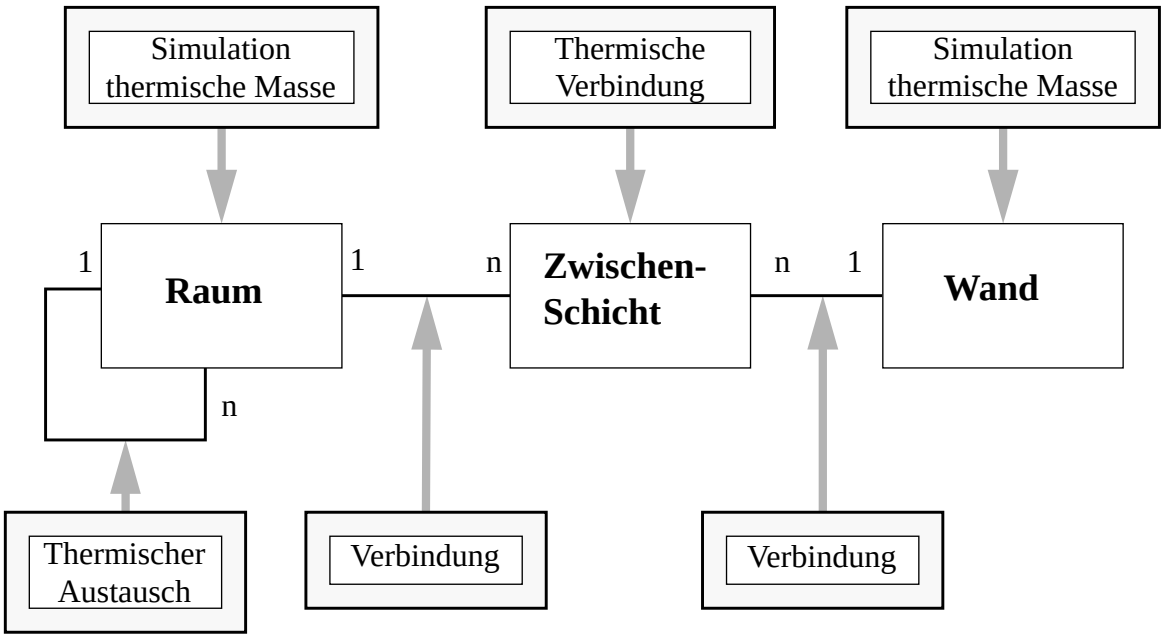
Die folgenden Patterns beschäftigen sich hauptsächlich mit der thermischen Simulation. Ähnliche Patterns für andere Simulationsgrößen (Luftdruck, Luftfeuchtigkeit, Licht, etc.) können aber sehr leicht neu in den Katalog aufgenommen werden.

Simulation thermischer Masse

- **Zweck**
  Dieses Pattern dient dazu, die Temperatur einer thermischen Masse in Abhängigkeit von auf die Masse einwirkenden Wärmemengen zu berechnen. Als thermische Masse kann die Luft in einem Raum angesehen werden, ebensogut können aber auch Wände oder Zwi-

schendecken als Wärmemasse betrachtet werden. Die spezifischen Eigenschaften einer Masse sind ihr Volumen und ihre Wärmekapazität. Die Wärmekapazität ist abhängig von der Temperatur der thermischen Masse, wird aber in der Regel als konstant angesehen.
Auf eine thermische Masse wirken Wärmemengen ein. Diese entstehen entweder durch Heizkörper, durch direkte Sonneneinstrahlung oder durch Transmission von angrenzenden Gebieten (Wärmeaustausch).

- **Motivation**
  Um die Temperatur an einer (beliebigen) Stelle eines Gebäudes zu messen, muß diese als thermische Masse agieren. Bei diesem Pattern wird angenommen, daß die Temperatur überall innerhalb der thermischen Masse gleich hoch ist (die Wärmeleitung innerhalb der Masse ist also unendlich groß). Typischerweise werden die Räume des Gebäudes als thermische Masse aufgefaßt. Aber auch Wände oder Decken und Fußböden können als eine eigene thermische Masse betrachtet werden. Ausschlaggebend für eine thermische Masse ist, daß diese eine gewisse Wärmemenge speichert und (eventuell) an umliegende Bereiche abgibt. Der Wärmeaustausch zwischen den einzelnen Massen geschieht entweder direkt (siehe Pattern *Thermischer Austausch (51)*) oder über eine Zwischenschicht (Pattern *Simulation thermischer Verbindung (49)*) und muß extra modelliert werden.
  Das folgende Bild zeigt, wie ein Teil des Simulators für ein einfaches Haus modelliert werden kann:

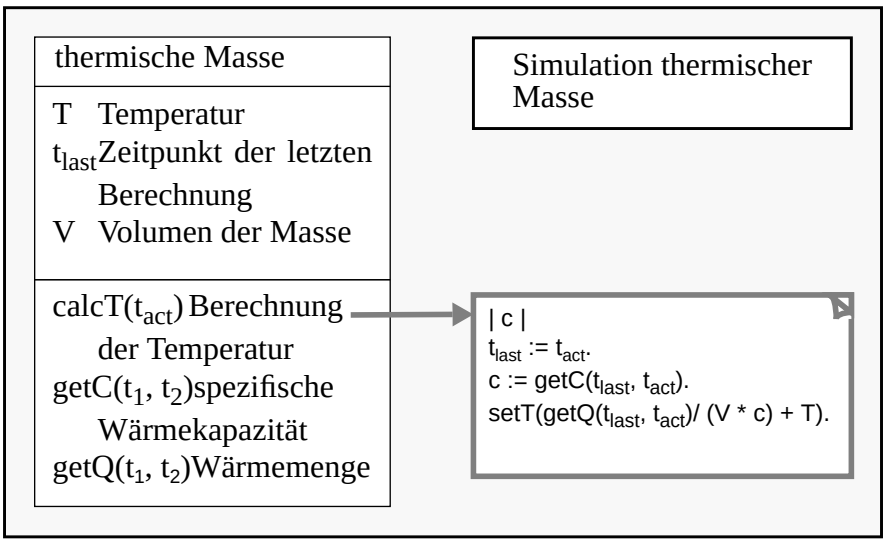


Bei dieser Modellierung werden Räume und Wände als thermische Masse aufgefaßt. Ein Wärmeaustausch zwischen diesen findet über Zwischenschichten statt, die jeweils eine Wand mit einem Raum verbinden. Dies entspricht einer Luftschicht, die sich vor den Wänden befindet und in der der größte Teil des Wärmeüberganges stattfindet. Zusätzlich zum Wärmeaustausch über Wände tauschen die Räume auch direkt, zum Beispiel über Zwischentüren, Wärmemengen aus.

Da die Art der Relationen zwischen den Objekten Raum, Wand und Zwischenschicht von Gebäudemodell zu Gebäudemodell verschieden sein kann, muß diese explizit über ein Verbindungspattern (siehe A.2.3) modelliert werden.

- **Anwendbarkeit**
  Dieses Pattern kann an eine beliebige thermische Masse gebunden werden. Typische Möglichkeiten sind Räume und dickere Wände oder Decken. Aber auch eine Nachtspeicherheizung kann man als thermische Masse auffassen. Wichtige Eigenschaft einer thermischen Masse ist, daß sie eine gewisse Wärmemenge speichern kann. Die Speicherfähigkeit ist abhängig von der spezifischen Wärmekapazität des Materials (oder Gases), aus dem die Masse besteht, und ihrem Volumen. Zusätzlich zu diesen Kennwerten muß bekannt (bzw. berechenbar) sein, welche Wärmemenge in die thermische Masse einfließt.

- **Struktur**
  Dieses Pattern beinhaltet nur Funktionalität für eine Klasse, nämlich die thermische Masse selbst. Die Berechnungsformel zur inkrementellen Berechnung der Temperatur lautet $T_{act} = T_{last} + \frac{Q(t_{last}, t_{act})}{V \cdot c(t_{last}, t_{act})}$ mit $Q(t_{last}, t_{act})$ = Wärmemenge, die im Zeitintervall (t_{last}, t_{act}) in die thermische Masse eingeflossen ist, V = Volumen der Masse, $c(t_{last}, t_{act})$ = durchschnittliche spezifische Wärmekapazität im Intervall (t_{last}, t_{act}). Durch die Anbindung dieses Patterns an ein Objekt, das als thermische Masse agieren soll, kann die jeweilige Temperaturänderung in Abhängigkeit der beeinflussenden Wärmemenge und der spezifischen Wärmekapazität (Materialkonstante) berechnet werden.



- **Mitwirkende Objekte**

  Klassen:

  thermische Masse: Die Masse, für die die Temperatur simuliert werden soll. Als Kennwerte der thermischen Masse muß das Volumen und die spezifische Wärmekapazität bekannt sein. Typische Beispiele für eine thermische Masse sind ein Raum oder auch dickere Wandschichten.

Instanzenvariablen / Konstanten:

T Beinhaltet die jeweils zuletzt berechnete Temperatur.

t_{last} Zeitpunkt, an dem die Temperatur T zuletzt berechnet wurde.

V Volumen der thermischen Masse (konstant)

Funktionen:

calcT(t_{act}) Funktion zur Berechnung der Temperatur. Als Übergabeparameter wird die aktuelle Zeit übergeben.

getC(T) Liefert die spezifische Wärmekapazität der thermischen Masse zurück. Die Wärmekapazität ist abhängig von der aktuellen Temperatur; sie wird jedoch meist als konstant angenommen. Die Berechnungsformel für die Wärmekapazität muß explizit über ein weiteres Pattern (*Konstanter Wert (63)* oder *Funktion (61)*) angegeben werden.

getQ(t_1, t_2) Die Temperatur in einer Masse wird durch die Wärmemenge beeinflußt, die in sie einströmt. Um also die Temperaturänderung im letzten Zeitintervall berechnen zu können, muß die Wärmemenge ermittelt werden, die im Intervall (t_{last}, t_{act}) auf die thermische Masse eingewirkt ist. Wie die Berechnung dieser Wärmemenge aussieht, muß durch andere Patterns modelliert werden.

- **Zusammenarbeit**
  Dieses Pattern hängt hauptsächlich von einer sinnvollen Berechnung der Wärmemenge ab, die in die thermische Masse einfließt. Üblicherweise setzt sich die Wärmemenge aus drei Komponenten zusammen: der Transmissionswärmemenge (Wärmedurchgang durch Wände etc.), der Strahlungswärmemenge von Heizkörpern oder Sonneneinstrahlung und dem Wärmeaustausch. Sämtliche Wärmemengen, die auf eine thermische Masse einwirken, müssen separat modelliert werden.

- **Konsequenz**
  Zur Anwendung des Patterns müssen das Volumen und die spezifische Wärmekapazität der thermischen Masse bekannt sein. Nach der Bindung dieses Patterns an eine Modell-Klasse muß vor allem noch modelliert werden, woher welche Wärmemengen in die Masse einfließen. Dazu werden in der Regel die entsprechenden abhängigen Objekte über eines der Pattern *Einfache Indirektion (67)* oder *Komplexe Indirektion (69)* an die thermische Masse gebunden (Ein Raum als thermische Masse kann beispielsweise die Klassen „Wand“, Tür“, „Fenster“ und „Heizkörper“ als Wärmequellen haben).

- **Beispielimplementierung und Benutzung**
  Siehe Abbildung 10 auf Seite 18.

- **Verwandte Patterns**
  Thermischer Austausch (51) zur Berechnung des direkten Wärmeaustausches.

Simulation thermischer Verbindung

- **Zweck**
  Eine thermische Verbindung ist ein Trennelement zwischen zwei thermischen Massen, die Wärme untereinander austauschen. Dieses Pattern dient dazu, den Wärmestrom zwischen den zwei thermischen Massen zu berechnen.

- **Motivation**
  Im einfachsten Fall kann eine Wand als Trennelement zweier Räume aufgefaßt werden. Haben beide Räume unterschiedliche Temperaturen, so fließt zwischen ihnen ein Wärmestrom. Dieser Wärmestrom ist abhängig vom Wärmewiderstand der thermischen Verbindung und der Temperaturdifferenz der beiden angrenzenden Massen ($Q = \frac{\Delta T}{R}$).

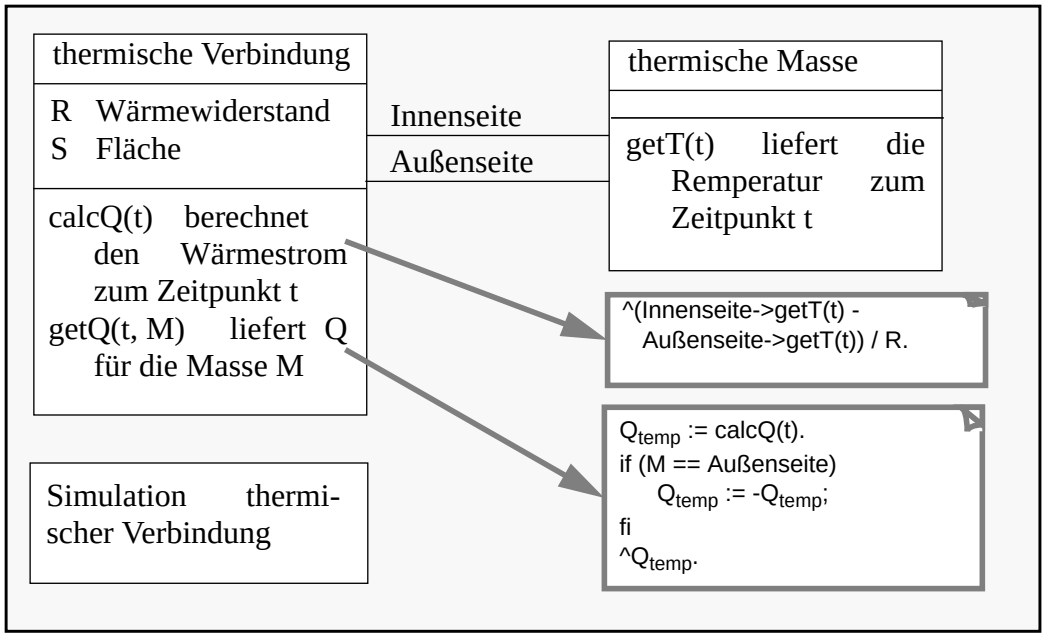
  Dieses Pattern stellt zwei Funktionen zur Verfügung: eine zur Berechnung des aktuellen Wärmestroms und eine zur vorzeichenrichtigen Abfrage desselben von einer thermischen Masse aus.

- **Anwendbarkeit**
  Bei jeder thermischen Verbindung. Der Wärmewiderstand der Verbindung muß bekannt oder zumindest berechenbar sein.

- **Struktur**
  Eine thermische Verbindung verbindet zwei thermische Massen miteinander. Durch eine thermische Verbindung fließt, abhängig von der Temperaturdifferenz der beiden thermischen Massen, ein Wärmestrom. Dieser Wärmestrom berechnet sich durch die Formel $Q = \frac{\Delta T}{R}$, wobei R den Wärmewiderstand[1] der thermischen Verbindung beschreibt. Der Wärmewiderstand einer thermischen Verbindung ist in der Regel konstant und wird entweder bei der Instanziierung der Klassen eingegeben oder durch eine Formel berechnet (Pattern *Funktion (61)*). Zusätzlich zu der Berechnungsformel für den Wärmestrom stellt dieses Pattern eine Funktion zur Abfrage desselben bereit. Dies ist notwendig, damit eine vorzeichenrichtige Verarbeitung des Wärmestroms möglich ist. Dazu wird als zusätzlicher Para-

1. Je nach Art der Verbindung wird der Wärmewiderstand unterschiedlich berechnet. Folgende vier Fälle sind möglich:
   i) Widerstand eines homogenen Wärmefeldes: $R_W = \frac{\delta}{\lambda \cdot S}$ mit δ = Schichtdicke, λ = Wärmeleitfähigkeit (Materialkonstante) und S = Schnittfläche
   ii) Wärmeübergangswiderstand für den Übergang an der Oberfläche eines festen Stoffes auf ein Fluid (z.B. Gas, Wasser, Dampf): $R_U = \frac{1}{\alpha \cdot S}$ mit dem Wärmeübergangskoeffizienten α.
   iii) Der Wärmedurchgangswiderstand beschreibt den Übergang von einem Fluid durch eine Wand auf ein anderes Fluid. Er berechnet sich durch $R_d = \frac{1}{k \cdot S}$ mit dem Wärmedurchgangzkoeffizienten $k = \frac{1}{\alpha_1} + R_W \cdot S + \frac{1}{\alpha_2}$ (α_i = Wärmeübergangskoeffizienden der Fluids).
   iv) Für die Wandung eines geraden Rohres mit kreisförmigen Querschnitt (Außendurchmesser d_2, Innendurchmesser d_1) gilt: $R = \frac{\ln (d_2/d_1)}{2\pi \cdot \lambda}$.

meter die thermische Masse an die Funktion *getQ* übergeben, die momentan als Innenseite agiert.
Die beiden Relationen Innenseite und Außenseite beschreiben die Nachbarschaft der thermischen Verbindung zu ihren thermischen Massen. Sie können im Modell auch zu einer zweistelligen, geordneten Relation zusammengefaßt werden.



- **Mitwirkende Objekte**

Klassen:

thermische Verbindung: Beschreibt die Verbindung zweier thermischer Massen. Die Verbindung selbst besteht aus einem festen Trennmaterial, so daß durch die angegebene Formel der Wärmedurchgang durch dieses Material berechnet wird.

thermische Masse: Die beiden thermischen Massen, die durch die Verbindung miteinander verbunden werden.

Instanzenvariablen / Konstanten:

R Der Wärmedurchgangswiderstand der Verbindung. Er ist während einer Simulation normalerweise konstant und setzt sich bei mehrschichtigen Wänden aus der Summe der Wärmewiderstände der einzelnen Schichten und den beiden Wärmeübergängen von den Randschichten in die benachbarte thermische Masse zusammen.

S Die Größe der Schnittfläche der thermischen Verbindung.

Funktionen:

thermische Verbindung::calcQ(t) Dient zur Berechnung des Wärmestromes durch die Verbindung zum Zeitpunkt t.

thermische Verbindung::getQ(t, M) Liefert den Wärmestrom zum Zeitpunkt t zurück, wobei die thermische Masse M als Innenseite angenommen wird.

thermische Masse::getT(t) Gibt die Temperatur der thermischen Masse zum Zeitpunkt t zurück.

- **Zusammenarbeit**
  Eine thermische Verbindung verbindet immer genau zwei Massen miteinander. Ist eine thermische Masse mit mehreren anderen Massen verbunden, so muß jede dieser Nachbarschaften durch ein separates Objekt vom Typ thermische Verbindung ausgedrückt werden. Um beispielsweise die Beziehung zwischen Räumen (= thermische Masse) und Wänden auszudrücken, lohnt es sich, einen neuen Objekttypen „Raumteilerelement“ einzuführen. Ein Raumteilerelement verbindet immer genau zwei Räume miteinander und kann so als thermische Verbindung aufgefaßt werden. Das komplexere Gebilde einer Wand würde in diesem Fall dann mehrere Raumteilerelemente aggregieren (siehe Abbildung 30).

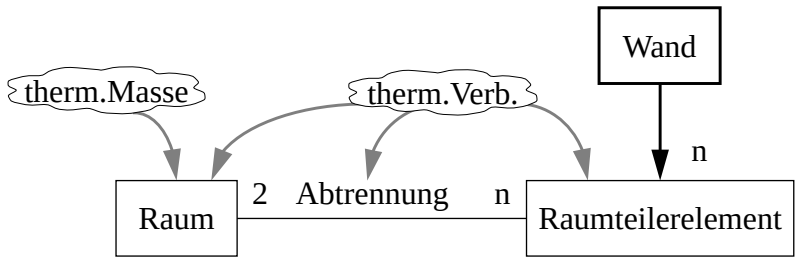


Abb.30: Eine Wand wird zusammengesetzt aus mehreren Raumteilerelementen

- **Konsequenz**
  Um dieses Pattern anwenden zu können, muß der Wärmewiderstand der Verbindung bekannt beziehungsweise berechenbar sein. Dieser muß bei der Initialisierung des Verbindungs-Objektes in die Instanzenvariable R eingetragen werden.
  Zusätzlich muß es möglich sein, die Temperatur einer thermischen Masse zu einem Zeitpunkt t abzufragen. Um dieses zu bewerkstelligen, müssen sämtliche oder zumindest die letzten Temperaturen der thermischen Massen zwischengepuffert werden. Ein weiterer Puffer ist eventuell notwendig, um die zuletzt berechneten Wärmeströme aufzunehmen, damit bei Bedarf daraus die Wärmemenge (= Integral über den Wärmestromverlauf) in einem Zeitintervall berechnet werden kann. Siehe dazu auch das Beispiel in Kapitel 2.7.

- **Verwandte Patterns**
  Simulation thermischer Masse (45) zur Berechnung der Temperatur einer Masse in Abhängigkeit des Wärmestromes.
  Das Pattern *Thermischer Austausch (51)* beschreibt die direkte Mischung zweier Gase.

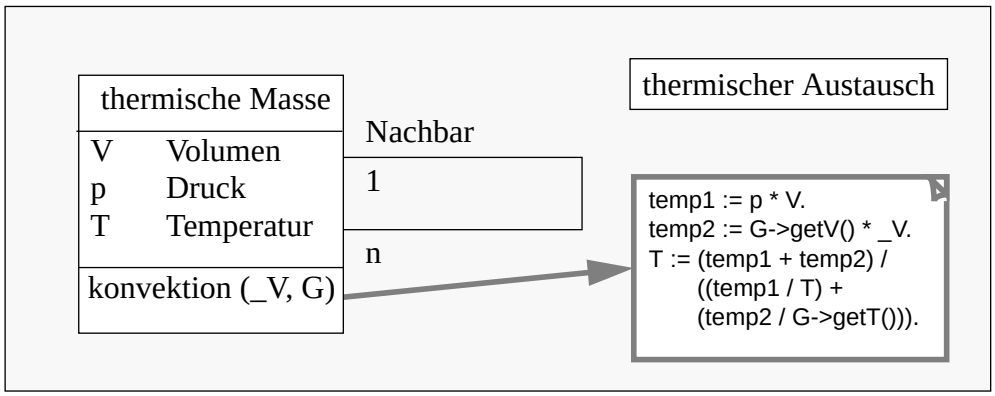
Thermischer Austausch

- **Zweck**
  Häufig tauschen sich verschiedene Gase direkt miteinander aus, es findet also kein Wärmeübergang statt, sondern eine direkte Vermischung der beiden (Luft-)Massen. Dies geschieht zum Beispiel durch ein offenes Fenster in einem Raum, durch das sich die Luft im Raum mit der Außenluft vermischt. Dieses Pattern stellt die nötigen Funktionen zur Verfügung, um die aus der Vermischung resultierenden Temperaturen zu berechnen.

- **Motivation**
  Durch ein offenes Fenster oder eine offene Tür strömt Luft, so daß sich die Luftmassen zu beiden Seiten des Fensters vermischen. Dadurch gleichen sich die Temperaturen der beiden Luftmassen allmählich aus. Die resultierende Temperatur ist abhängig vom Volumen der ausgetauschten Luftmengen und den Luftdrücken.

- **Anwendbarkeit**
  Das Pattern wird angewendet, um die aus der Vermischung zweier Gase entstehende Temperaturveränderung zu berechnen. Die angegebene Berechnungsformel liefert exakte Ergebnisse für gleichatomige Gase und kann als gute Annäherung für verschiedenatomige Gase benutzt werden.

- **Struktur**
  Zwei thermische Massen sind mit einer Relation verbunden, die die Nachbarschaft ausdrückt. Um zu berechnen, wie sich die Temperatur einer Gasmenge verändert, wenn eine andere Menge sich mit ihr vermischt, wird die Funktion *konvektion(V, G)* bereitgestellt. Als Übergabeparameter erhält sie das Volumen der ausgetauschten Gasmenge und einen Zeiger auf das einströmende Gas. Das Volumen und der Druck der Masse, in die das zweite Gas einströmt, muß auch bekannt sein. Die Berechnungsformel für die Vermischung zweier Gase lautet: $T = \frac{p_1 V_1 + p_2 V_2}{\frac{p_1 V_1}{T_1} + \frac{p_2 V_2}{T_2}}$. Sie kann auch für die Vermischung von mehr als zwei Gasen erweitert werden.



- **Mitwirkende Objekte**

  Klassen:

  thermische Masse: Objekte dieser Klasse repräsentieren die Gase, die sich vermischen.

  Instanzenvariablen / Konstanten:

  thermische Masse::V Volumen des Gases in m^3.
  thermische Masse::p Gasdruck in Pascal.

thermische Masse::T Temperatur in Grad Celsius.

Funktionen:

thermische Masse::konvektion(_V, G) Dient zur Anpassung der Temperatur, wenn ein Volumen _V des Gases G in das aktuelle Gas einströmt.

- **Zusammenarbeit**
  Die thermischen Massen sind über die 1:n Relation *Nachbar* miteinander verbunden.
- **Verwandte Patterns**
  Simulation thermischer Verbindung (49) zur Berechnung des thermischen Austausches durch ein Zwischenmedium (Wand etc.).

A.1.2 Scheduling

Die folgenden Patterns beschäftigen sich mit dem Problem des Scheduling, also mit dem Aufruf bestimmter Funktionen zu einem festen Zeitpunkt und in der gewünschten Reihenfolge. Die Funktionsweise des Scheduler ist in Kapitel 3.2 und in [Hei96] beschrieben.

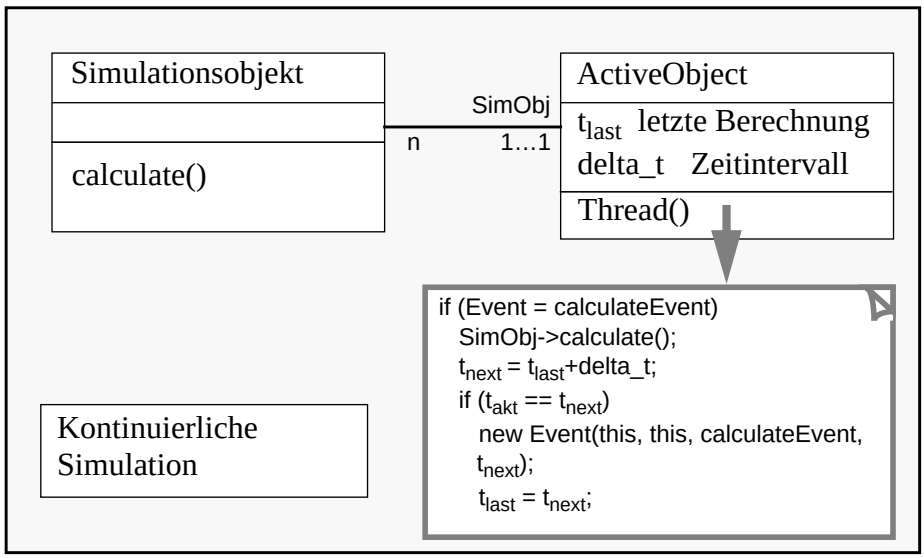
Kontinuierliche Simulation

- **Zweck**
  Dieses Pattern sorgt dafür, daß das Simulationsobjekt, an das es gebunden wird, kontinuierlich zur Berechnung seiner Simulationsgröße aufgefordert wird. Durch die Bindung des Patterns wird auch ein Event-Typ erzeugt, mit dem das Simulationsobjekt jederzeit (also auch außerhalb der periodischen Berechnungen) aufgefordert werden kann, seine Simulationsgröße zu aktualisieren.
- **Motivation**
  Die Modellierung der Berechnungsformeln einer Simulationsgröße alleine reicht nicht aus, um einem Simulator „Leben einzuhauchen“. Das Simulationsobjekt muß auch in bestimmten Abständen aufgefordert werden, seine Berechnungen durchzuführen. Bei den meisten Objekten ist dies sinnvoll, wenn sie sich in regelmäßigen Abständen neu berechnen, um jederzeit eine gute Annäherung an die nachgeahmte physikalische Größe zu haben. Die Periodendauer ist je nach Simulationsobjekt unterschiedlich und kann sich auch während der Simulation noch ändern. Bei der Raumtemperatur beispielsweise reicht es aus, wenn sie jede Minute neu berechnet wird. Öffnet jedoch jemand die Tür zu diesem Raum, kann sich die Temperatur sehr rasch ändern. Das macht dann eine häufigere Berechnung (z.B. alle 5 Sekunden) notwendig.
  Um eine kontinuierliche Berechnung durchführen zu können, muß das Simulationsobjekt eine Anbindung an den Scheduler haben. Diese Kopplung geschieht über *Active Objects* (vgl. 3.2). Durch dieses Pattern wird das Simulationsobjekt mit einem Active Object verbunden (falls das nicht bereits durch ein anderes Pattern geschehen ist). In diesem Active Object wird ein neuer Thread (= unabhängig von der restlichen Simulation laufender Prozeß) erzeugt. Dieser Thread wird aufgerufen, wenn er über den Scheduler ein entsprechen-

des Event zugesendet bekommt. Daraufhin veranlaßt er die Neuberechnung der Simulationsgröße (durch den Aufruf der Metode *calculate()* beim Simulationsobjekt). Anschließend verschickt der Thread einfach ein Event an sich selbst und gibt darin an, daß es erst zu einem späteren Zeitpunkt (nämlich genau zum Beginn der nächsten Periode) ankommen soll. Dadurch treffen dann zu den entsprechenden Zeitpunkten Events mit der Aufforderung zur Neuberechnung an.

- **Anwendbarkeit**
  Das Pattern kann prinzipiell an jede Klasse gebunden werden. Beachtet werden muß allerdings, daß eine zu häufige Berechnung einer Simulationsgröße zu einem erheblichen Rechenaufwand führen kann, so daß im Extremfall die Echtzeitfähigkeit des Simulators nicht aufrecht erhalten werden kann. Andererseits führt eine zu langsame Berechnung zu mathematischen Ungenauigkeiten, die das Ergebnis der Simulation verfälschen können.

- **Struktur**
  Durch die Bindung des Patterns Kontinuierliche Simulation an eine Klasse wird diese mit einem Active Object verbunden. Die Simulations-Klasse braucht von dieser Ankopplung nichts zu wissen, da die Kommunikation nur in der anderen Richtung stattfindet: das Active Object ruft in periodischen Zeitabständen die Methode *calculate()* des Simulationsobjektes auf. Zusätzlich zu der Anbindung an ein Active Object wird in diesem noch ein neuer Thread erzeugt, und es wird ein neuer Typ von Events vorgesehen, der die Simulationsobjekte zum Rechnen auffordern soll.



Der Thread reagiert auf das Eintreffen eines calculate-Events. Daraufhin ruft er eine Methode zur Neuberechnung der Simulationsgröße (*calculate()*) auf. Zuletzt schickt er einen neuen calculate-Event an sich selbst, der erst zur nächsten Periode eintreffen soll.

- **Mitwirkende Objekte**

  Klassen:

  Simulationsobjekt: Das Objekt, deren Simulationsgröße quasi-kontinuierlich neu berechnet werden soll.

  ActiveObject: Die entsprechende Klasse aus dem Simulator-Modell.

Instanzenvariablen / Konstanten:

ActiveObject::t_{last} Zeitpunkt der letzten Berechnung (genauer: des letzten Periodenanfanges)

ActiveObject::delta_t Dauer einer Periode. Kann im Bedarfsfall an die momentane Situation angepaßt werden (z.B. kürzere Periode nach Öffnen eines Fensters).

Funktionen:

Simulationsobjekt::calculate() Methode zur Neuberechnung der Simulationsgrößen.

ActiveObject::Thread() Der Thread (= zeitlich unabhängiger Programmteil), der sich um die Kontinuität kümmert.

- **Zusammenarbeit**
  Hauptsächlich kümmert sich der Scheduler darum, daß die Neuberechnung der Simulationsgrößen immer zum richtigen Zeitpunkt angestoßen wird. Dazu wird dem Active Object ein Event geschickt. Bei Ankunft dieses Events fordert das Active Object sein Simulationsobjekt auf, sich neu zu berechnen. Anschließend schickt es sich selbst ein neues Event, das aber erst zu einem späteren Zeitpunkt wieder eintreffen soll. Der Scheduler merkt sich dieses Event und schickt es an das ActiveObject zurück, sobald der entsprechende Zeitpunkt gekommen ist.

- **Konsequenz**
  Prinzipiell kann dieses Pattern an jede Modell-Klasse gebunden werden. Da es jedoch einen gewissen Overhead mit sich führt (für das Active Object muß ja ein eigener Thread angelegt werden), sollte es sparsam eingesetzt werden. Jedes Simulationsobjekt braucht nur mit maximal einem ActiveObject verbunden zu werden. Der Thread des ActiveObjects kann auch durchaus mehrere unterschiedliche Events verarbeiten.

- **Verwandte Patterns**
  Mit dem Pattern *Aktuator (55)* kann ein Simulationsobjekt zu einer einmaligen (asynchronen) Berechnung seiner Simulationsgrößen aufgefordert werden.

Aktuator

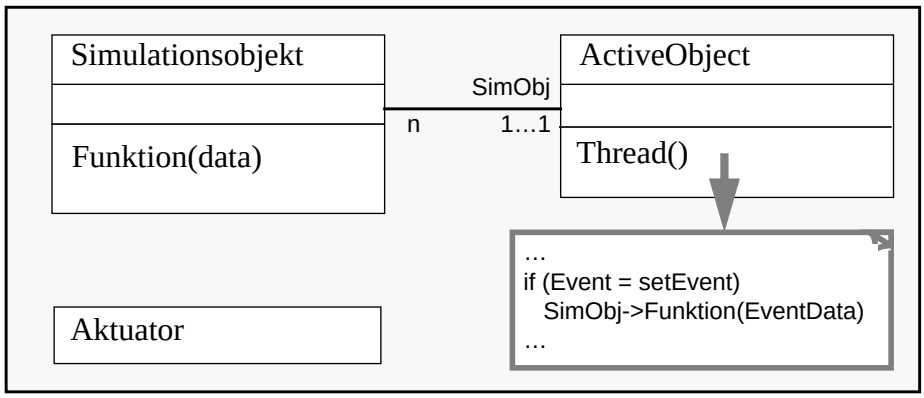
- **Zweck**
  Ein Aktuator ist aus Sicht des Simulators eine Stellgröße, die von außen gesetzt wird. Diese Größe kann dann zur Berechnung weiterer Simulationswerte herangezogen werden. Durch dieses Pattern wird einem Simulationsobjekt prinzipiell die Möglichkeit gegeben, von außerhalb der Simulation (Simulator-Steuerung oder Hardware-In-The-Loop) verändert zu werden.

- **Motivation**
  Die Heizleistung eines Radiators ist ein typisches Beispiel, bei dem ein Aktuator verwendet werden kann. Die Heizleistung ist normalerweise ein fester Wert, der sich ändert, wenn ein Benutzer das Heizungsventil öffnet oder schließt. Diese Benutzereingabe verläuft prinzipiell asynchron zur eigentlichen Simulation.

Um eine Simulationsgröße derart beeinflussen zu können, muß eine Verbindung zwischen dem Simulationsobjekt und dem Scheduler hergestellt werden. Dann kann über entsprechende Events jederzeit - asynchron zur Simulation - der Zustand des Simulationsobjektes abgefragt werden.

- **Anwendbarkeit**
  Dieses Pattern braucht nur eingesetzt zu werden, wenn Änderungen der Simulationsgröße zu einem bestimmten Zeitpunkt stattfinden sollen. Um ein Simulationsobjekt sofort zu ändern, kann auch direkt die entsprechende Instanzenvariable modifiziert werden (über die Zugriffsfunktionen). Um jedoch Aktionen für die Zukunft einzuplanen (Beispiel: um 7:00 Uhr soll die Heizung eingeschaltet werden), muß das Objekt, das angesprochen werden soll (in diesem Fall also die Heizung), in der Lage sein, Events zu empfangen. Dann kann nämlich ein Event verschickt werden, das genau in einem vorherbestimmten Zeitintervall beim Ziel eintrifft. Mit diesem Pattern wird die Fähigkeit modelliert, Events empfangen zu können.

- **Struktur**
  Um ein Event empfangen zu können, muß ein Objekt vom Typ Active Object sein. Zur Vermeidung der Mehrfachvererbung, wird mit diesem Pattern für ein Simulationsobjekt extra ein ActiveObject angelegt und über Delegations-Mechanismen angesprochen. Das Active Object legt einen Thread an, der auf das Eintreffen von Events wartet. Trifft ein solches Event ein, wird eine Funktion beim Simulationsobjekt aufgerufen. Diese kann eine Zugriffsfunktion auf eine Instanzenvariable sein, kann aber auch genau so gut eine andere Funktionalität besitzen (beispielsweise zur Neuberechnung einer Simulationsgröße auffordern).



- **Mitwirkende Objekte**

  Klassen:

  SimulationsObjekt: Die Modell-Klasse, die Events zu bestimmten Zeitpunkten empfangen können soll.

  ActiveObject: Die Klasse ActiveObject ist bereits im Simulator Modell vorgegeben (siehe 3.2). Durch dieses Pattern wird nur ein Thread des Active Objects um die Funktionalität erweitert, auf ein spezielles Event zu reagieren.

Funktionen:

ActiveObject::Thread() Diese Methode wird unabhängig vom übrigen Programmfluß ausgeführt und wartet kontinuierlich auf das Eintreffen eines Events. Beim Eintreffen eines Events wird eine dem Event-Typ entsprechende Aktion beim Simulationsobjekt ausgelöst, und anschließend wird wieder auf neue Events gewartet.

SimulationsObjekt::Funktion(data) Die Funktion, die ausgeführt werden soll, wenn ein Event eintrifft. Als Übergabeparameter können die im Event mitgegebenen Daten dienen.

- **Zusammenarbeit**
  Jedes Simulationsobjekt braucht nur mit einem Active Object verbunden zu sein. Ist eine Modell-Klasse bereits mit einem Active Object verbunden, so wird durch dieses Pattern einfach nur der Thread dieses Objektes erweitert. Die Kommunikation verläuft immer ausgehend von einem Event über das Active Object zum eigentlichen Simulationsobjekt. Dieses braucht sich ansonsten nicht weiter um das Active Object zu kümmern.

- **Konsequenz**
  Prinzipiell kann dieses Pattern an jede Modell-Klasse gebunden werden. Da es jedoch einen gewissen Overhead mit sich führt (für das Active Object muß ja ein eigener Thread angelegt werden), sollte es sparsam eingesetzt werden. Jedes Simulationsobjekt braucht nur mit maximal einem ActiveObjekt verbunden zu werden. Der Thread des Active Objects kann auch durchaus mehrere unterschiedliche Events verarbeiten.

- **Verwandte Patterns**
  Kontinuierliche Simulation (53)

A.2 Framework-Patterns

Die folgenden Patterns können zur Modellierung beliebiger Frameworks herangezogen werden. Die impliziten Patterns in Kapitel A.2.1 können von einem Generator automatisch verwendet werden. Dadurch erhält selbst ein ansonsten „leeres“ Framework (d.h. eines, das außer Instanzenvariablen und Relationen noch keine Funktionalität hat) bereits eine gewisse Grundfunktionalität. Dadurch, daß diese Funktionalität einheitlich für alle Klassen generiert wird, erleichtert sich der Umgang mit dem Framework. Zur Verfeinerung der einzelnen Modell-Klassen dienen die Patterns in Kapitel A.2.2. Im folgenden Kapitel werden Methoden vorgestellt, wie man Relationen zwischen Klassen verfolgen kann. Zum Schluß werden noch Strukturierungsmethoden der Modell-Klassen vorgestellt.

A.2.1 Implizite Patterns

Die Patterns in diesem Abschnitt können von einem Generator automatisch angewendet werden, um eine einheitliche Zugriffsschnittstelle auf alle Objekte des Gesamtmodells zu erhalten. Bei der Modellierung eines Frameworks kann daher davon ausgegangen werden, daß die folgenden Patterns bereits auf jede Modell-Klasse angewendet worden sind.

Instanzenvariable

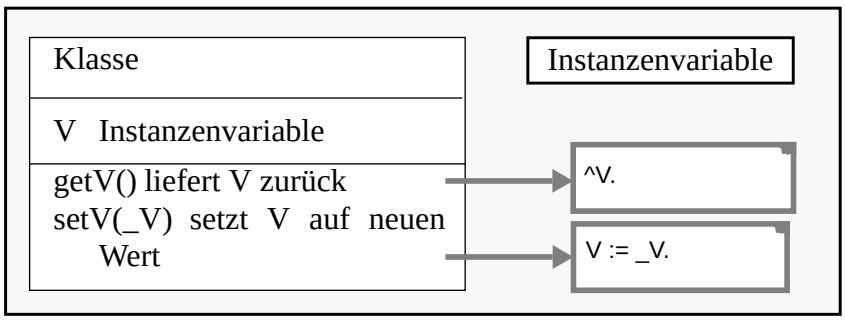
- **Zweck**
  Dient zur automatischen Kapselung von Instanzenvariablen. Für sämtliche Instanzenvariablen, für die durch andere Patterns noch keine Zugriffsfunktionen erstellt wurden, werden entsprechende Zugriffsmethoden erzeugt. Der Zugriff auf eine Instanzenvariable, ausgehend von einem Objekt einer anderen Klasse, sollte nur über diese Funktionen geschehen.

- **Bekannt unter**
  Attribut

- **Motivation**
  Jedes Objekt sollte bei einem objektorientierten Programm nur direkten Zugriff auf seine eigenen, lokalen Daten haben. Auf Daten anderer Objekte wird über Zugriffsfunktionen operiert. Durch diese Indirektion ist es möglich, zusätzliche Sicherheitsabfragen (zum Beispiel Bereichsüberschreitungen oder Schutzverletzungen) in die jeweiligen Zugriffsfunktionen einzubauen, um somit die Daten eines Objektes konsistent zu halten. Wenn auf sämtliche Instanzenvariablen über eigene Funktionen zugegriffen werden kann und darüberhinaus die Namenskonvention dieser Funktionen gleichbleibend ist, so kann auf einfache Art und Weise der Zustand jedes Objektes gelesen und verändert werden.

- **Anwendbarkeit**
  Dieses Pattern wird vom Generator implizit angewendet. Es braucht daher nicht explizit an Objekte gebunden zu werden. Im generierten Programm kann davon ausgegangen werden, daß für sämtliche Instanzenvariablen Zugriffsfunktionen vorhanden sind.

- **Struktur**
  Die Struktur des Patterns ist sehr einfach. Jede Instanzenvariable erhält eine Lesefunktion und eine Schreibfunktion. Die Lesefunktion heißt dabei genau so, wie die Variable im Modell selbst, erhält nur den zusätzliche Präfix *get*; bei der Schreibfunktion (Präfix *set*) wird ein zusätzlicher Parameter übergeben.



- **Mitwirkende Objekte**

  Klassen:

  Klasse: Die Klasse, für die Zugriffsfunktionen generiert werden.

  Instanzenvariablen / Konstanten:

  V Die entsprechende Variable. Um Namenskonflikte zu vermeiden, kann sie im Quelltext umbenannt werden (z.B. _V).

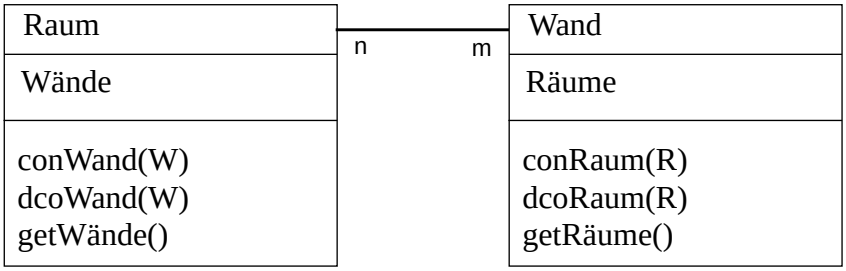
Funktionen:

getV() Methode, um den Wert der Variablen abzufragen.
setV(_V) Methode zum Setzen der Instanzenvariable.

- **Konsequenz**
  Der Zugriff auf Instanzenvariablen von außerhalb eines Objektes sollte nur über die Zugriffsfunktionen geschehen.
- **Implementierung/Bindung**
  Die Instanziierung dieses Patterns erfolgt automatisch durch den Generator.
- **Verwandte Patterns**
  Funktion (61) zur Modellierung einer neuen Funktion. Instanzenvariablen sind „nach außen“ hin nur über ihre Zugriffsfunktionen sichtbar. Es ist also auch möglich, über die Nachbildung dieser Zugriffsfunktionen eine Instanzenvariable nachzubilden oder ihr zusätzliche Funktionalität zu geben.

Relation

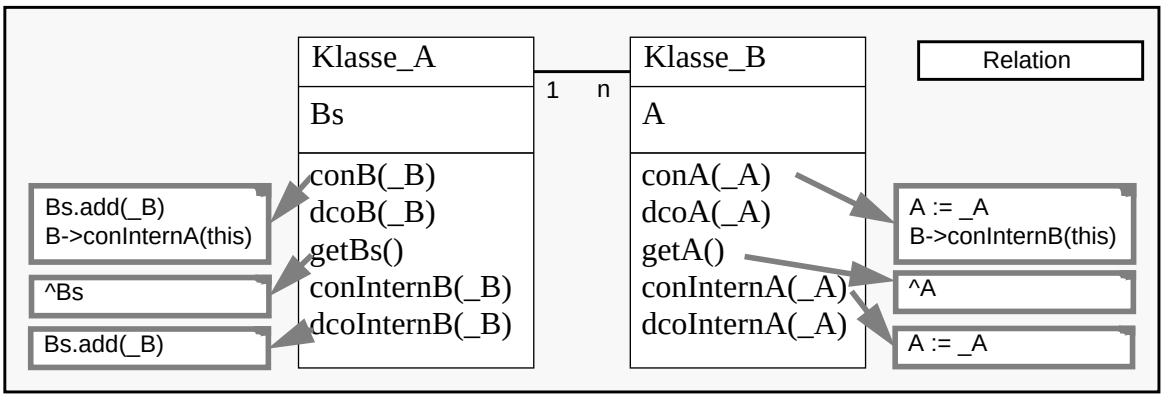
- **Zweck**
  Dieses Pattern generiert automatisch Zugriffsmethoden für Relationen. Für eine zweiseitgie Relation werden Funktionen zum Verbinden zweier Objekte und zum Auflösen einer bestehenden Verbindung angelegt. Dabei werden immer beide Richtungen berücksichtigt; verbindet man also Objekt A mit einem Objekt B, so kann auch von B aus auf A zugegriffen werden.
- **Motivation**
  Zum einen dient dieses Pattern zur Datenkapselung. Auf den Instanzenvariablen, die die Relation bilden, soll nur über Zugriffsfunktionen operiert werden. Zum anderen wird dafür gesorgt, daß die Relationen auf beiden Seiten immer konsistent sind.
  Besteht beispielsweise eine Relation zwischen einem Raum und dessen Wänden, so werden für den Raum die Funktionen *conWand()* und *dcoWand()* zum Verbinden einer Wand mit einem Raum generiert (die Präfixe *con* und *dco* stehen für connect bzw. disconnect). Gleichzeitig wird eine Zugriffsfunktion (*getWände()*) angelegt, um alle mit einem Raum verbundenen Wände zu erhalten.



Hat man in obigen Beispiel eine Wand W1 und einem Raum R1, so können diese mit dem Aufruf *R1.conWand(W1)* miteinander verbunden werden. Dasselbe hätte *W1.conRaum(R1)* bewirkt. Mit der Funktion *Wände()* wird die Menge von Wänden zurückgegeben, die mit dem aktuellen Raum verbunden sind. Bei einer 1:1 oder n:1 Relation wird jeweils nur eine Wand zurückgegeben.

- **Anwendbarkeit**
  Das Pattern wird vom Generator automatisch für jede Relation verwendet. Eine explizite Anwendung ist nicht nötig.

- **Struktur**
  Für jede Relation zwischen zwei Objekten werden auf beiden Seiten connect und disconnect Methoden angelegt. Können von einem Objekt aus mehrere andere Objekte referenziert werden (ein Raum wird zum Beispiel von mehreren Wänden umgeben), so werden alle diese Objekte in einer Menge gespeichert. Handelt es sich dagegen um eine 1:1 Relation, so kann *das* refenzierte Objekt auch in einer effizienteren Form (z.B. als Zeiger) abgelegt werden.
  Wird bei einer Klasse die connect-Methode aufgerufen, so wird in dieser Klasse die entsprechende Referenz eingetragen und gleichzeitig eine connectIntern-Methode beim referenzierten Objekt aufgerufen. Diese interne Methode sorgt dafür, daß alle Referenzen jeweils auf beiden Seiten der Relation eingetragen werden.



In der Abbildung sind aus Platzgründen nur die connect-Methoden aufgeführt. Die disconnect-Methoden funktionieren aber analog.

- **Mitwirkende Objekte**

  Klassen:

  Klasse_A: Beliebige Klasse, die eine 1:n Relation zu einer anderen Klasse hat.
  Klasse_B: Die Klasse B soll in diesem Beispiel eine 1:1 Relation zur Klasse A haben.

  Instanzenvariablen / Konstanten:

  Klasse_A::Bs Menge aller referenzierten Objekte der Klasse B
  Klasse_B::A Zeiger auf das referenzierte A-Objekt

Funktionen:

Klasse_A::conB(_B) Verbindet ein Objekt der Klasse A mit einem der Klasse B (und umgekehrt)

Klasse_A::dcoB(_B) Löst eine vorhandene Verbindung wieder auf.

Klasse_A::Bs() Methode zum Abfragen aller referenzierten Objekte.

Klasse_A::conInternB(_B) Diese Methode verbindet nur einseitig ein B-Objekt mit einem Objekt der Klasse A. Sie wird von der Methode KlasseB::conA(_A) aufgerufen und sollte ansonsten nicht direkt verwendet werden.

Klasse_A::dcoInternB(_B) Zum einseitigen lösen einer Verbindung.

Die entsprechenden Methoden der Klasse B besitzen dieselbe Funktionalität.

- **Zusammenarbeit**
  Beim Ein- oder Austragen einer Relation wird jeweils die entsprechende interne Methode der anderen Klasse aufgerufen, damit die Relationen auf beiden Seiten konsistent sind.
- **Verwandte Patterns**
  Um eine Relation zu verfolgen und Methoden bei den referenzierten Objekten aufzurufen, können die Patterns *Einfache Indirektion (67)* und *Komplexe Indirektion (69)* benutzt werden.

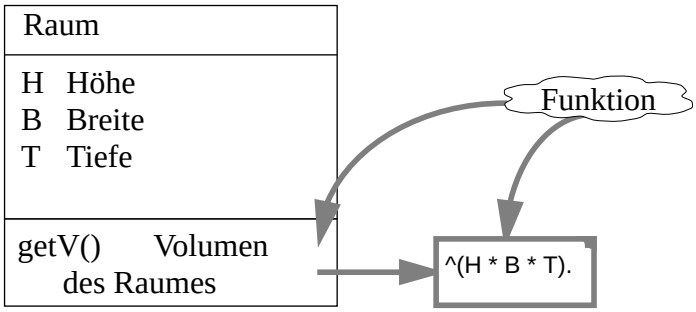
A.2.2 Klassen

Die Patterns in diesem Abschnitt dienen zur Verfeinerung einzelner Klassen. Statt sich um das Zusammenspiel zwischen Klassen zu kümmern, fokussieren sie hauptsächlich die Funktionalität einzelner Klassen.

Funktion

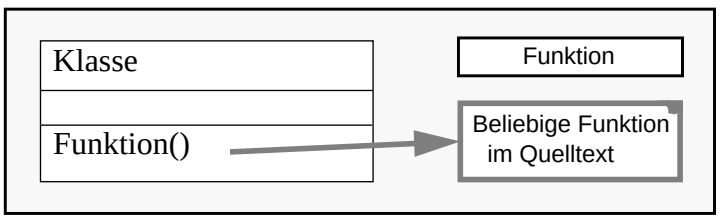
- **Zweck**
  Bei der Modellierung werden häufiger Berechnungsformeln benötigt, die nicht durch ein Pattern ausgedrückt werden können. Um diese Formeln ausdrücken zu können, kann in dieses Pattern die entsprechende Funktion eingesetzt werden.
- **Motivation**
  Angenommen, es sind im Gebäudemodell die Maße (Breite, Höhe und Tiefe) für einen Raum eingetragen. Um die Temperatur dieses Raumes zu berechnen, wird jedoch das Volu-

men (= Breite * Höhe * Tiefe) benötigt. Diese Berechnungsformel kann einfach über dieses Pattern an den Raum gebunden werden.



- **Anwendbarkeit**
  Jederzeit, wenn eine zusätzliche Berechnungsformel oder ein Prozeduraufruf, der nicht durch andere Patterns abgedeckt werden kann, benötigt wird. Zur Anwendung dieses Patterns muß der *Quellcode*, der generiert werden soll, eingegeben werden. Die Benutzung dieses Patterns ist also von der Ziel-Programmiersprache abhängig und erfordert Kenntnisse dieser Sprache. Daher sollte dieses Pattern möglichst spärlich eingesetzt werden.
  Alternativ wäre auch denkbar, den Quelltext der Formel aus einer allgemeinen Beschreibungssprache heraus zu generieren. In diesem Fall könnte bei der Benutzung des Patterns die gewünschte Formel in einer speziellen Sprache eingegeben werden und bei der Generierung dann in entsprechenden Quelltext transformiert werden. Dadurch wird das Modell unabhängig von der Ziel-Programmiersprache.

- **Struktur**
  Die Struktur des Patterns ist sehr einfach. Es wird für die Klasse, an die das Pattern gebunden wird, eine neue Funktion erzeugt, die die geforderten Berechnungen ausführt.



- **Mitwirkende Objekte**

  Klassen:
  Klasse: Die Klasse, für die die Funktion angelegt werden soll.

  Funktionen:
  Funktion() Die Funktion, die erzeugt werden soll. Bei der Bindung des Patterns muß der Quelltext eingegeben werden.

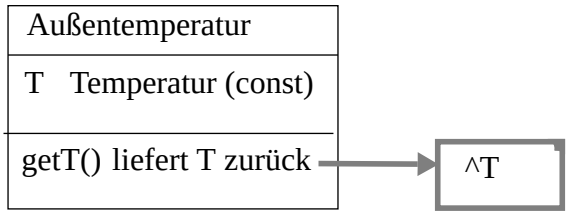
- **Konsequenz**
  Bei der Bindung des Patterns muß die gewünschte Funktion im *Quelltext* der Ziel-Programmiersprache eingegeben werden. Dadurch wird das Pattern von der Programmiersprache

abhängig. Ein Generator kopiert während der Generierung den eingegebenen Quelltext einfach an die entsprechende Stelle im Programm hinein.

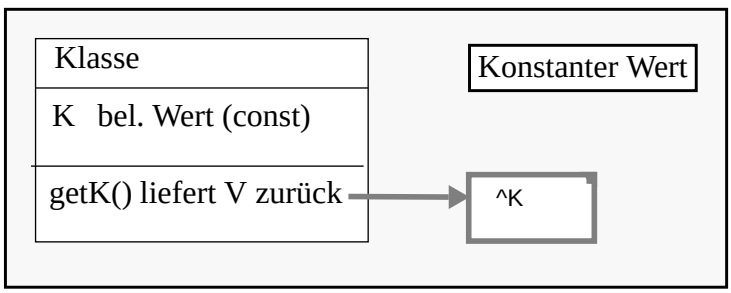
- **Verwandte Patterns**
  Tabelle (65) um Funktionswerte aus einer Tabelle auszulesen oder zu interpolieren.
  Konstanter Wert (63) zur Modellierung einer Konstante.

Konstanter Wert

- **Zweck**
  Liefert einen konstanten Wert zurück. Dieser Wert ist für alle Objekte einer Klasse gleich.
- **Motivation**
  Viele Algorithmen hängen von Konstanten ab, die sich zwar von Anwendung zu Anwendung ändern, während einer Anwendung jedoch konstant bleiben,
  Legt man bei der Simulation beispielsweise ein vereinfachtes Wettermodell zugrunde, so ändert sich die Außentemperatur während eines Simulationslaufes nicht.



- **Anwendbarkeit**
  Überall dort, wo mit konstanten Werten gerechnet wird. Kann es vorkommen, daß sich der Wert während einer Simulation ändert, so sollte eine Variable benutzt werden (Pattern *Instanzenvariable (58)*).
- **Struktur**
  Die Struktur des Patterns ist sehr einfach. Das Pattern wurde nur in den Pattern-Katalog aufgenommen, um eine durchgängige Modellierung mit Patterns zu ermöglichen.



- **Mitwirkende Objekte**

  Klassen:

  Klasse: Die Klasse, für die der konstante Wert gelten soll. Die Klasse muß bereits existieren.

  Instanzenvariablen / Konstanten:

  K Die verwendete Konstante. Für den Wert der Kostante wird kein zusätzlicher Speicherplatz benötigt; vielmehr wird der Wert direkt in den Programmtext hineinkopiert.

  Funktionen:

  getK() Methode, um den Wert der Konstanten abzufragen. Um eine transparente Datenkapselung zu erhalten, sollte nur mit dieser Funktion auf die Konstante zugegriffen werden.

- **Konsequenz**

  Ist ein Wert als konstant festgelegt, so kann er nachträglich nicht geändert werden. Eine Änderung erfordert in der Regel die Neugenerierung des gesamten Programmes.

- **Implementierung/Bindung**

  Klasse -> beliebige, bereits existierende Klasse.

  K -> beliebiger Variablenname, der Wert wird direkt eingesetzt.

  getK() -> beliebige Funktion. Die Funktion wird vollständig angelegt (Struktur und Implementierung).

  Quelltext:

```
Klasse
…
getK
    "Liefert konstanten Wert zurück"
    "generiert aus Pattern: Konstanter Wert"
    ^K
```

- **Beispielimplementierung und Benutzung**

  Bei der Einfachheit des Patterns sollte die Beschreibung unter Struktur und Motivation ausreichen.

- **Verwandte Patterns**

  Tabelle (65)- Werte aus einer Tabelle auslesen.

  Funktion (61)- (Beliebige) Funktionen berechnen.

  Instanzenvariable (58) - Um Speicherplatz für eine Variable zu reservieren und Zugriffsfunktion bereitstellen.

Tabelle

- **Zweck**
  Eine Tabelle bietet die Möglichkeit, zu gegebenen Eingabewerten vorher abgespeicherte Ausgabewerte zurückzuliefern. Bei numerischen Tabelleneinträgen besteht auch die Möglichkeit, Werte zu interpolieren.

- **Motivation**
  Viele physikalische Größen wie Materialkennwerte sind in der Regel nur schwer zu berechnen. Daher werden Meßreihen aufgestellt, wie sich diese Kennwerte bei unterschiedlichen Randbedingungen verändern. Die Dichte von Luft ist beispielsweise von ihrer Temperatur abhängig. Um die Dichte bei einer beliebigen Temperatur anzunähern, kann eine Meßreihe aufgestellt (oder berechnet) werden, in der Dichtewerte bei einzelnen Temperaturen vermerkt sind. Durch Interpolation dieser Werte kann die gesuchte Dichte dann ermittelt werden.

Tabelle 1: Dichte von Luft in Abhängigkeit der Temperatur

ν [°C]	-40	-20	0	20	40	60	80	100	120
ρ [kg/m^3]	1,49	1,37	1,27	1,18	1,11	1,04	0,99	0,93	0,88

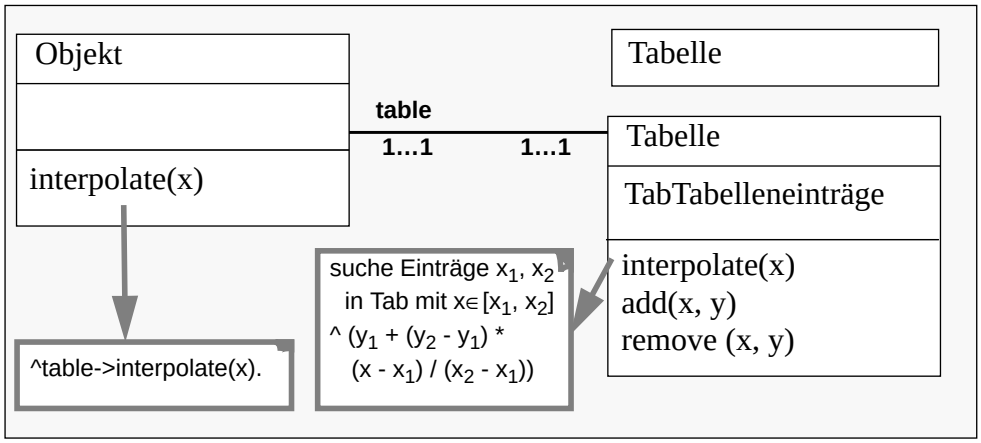
Die Dichte der Luft bei 25 °C ergibt sich bei linearer Interpolation zu 1,16 kg/m^3 (aus der Tabelle können die Werte $\nu_1 = 20$, $\nu_2 = 40$, $\rho_1 = 1{,}18$, $\rho_2 = 1{,}11$ abgelesen werden. Mit diesen Eckdaten ergibt sich $\rho(\vartheta) = \rho_1 + \frac{(\rho_2 - \rho_1)}{(\vartheta_2 - \vartheta_1)} \cdot (\vartheta - \vartheta_1)$, also $r(25°C) \approx 1,16 kg/m^3$).

- **Anwendbarkeit**
  Dieses Pattern kann zum Auslesen und Interpolieren numerischer Tabellen benutzt werden. Diese Tabellen können fest vorgegeben sein oder auch während des Programmlaufes verändert werden.

- **Struktur**
  Eine Tabelle wird durch ein eigenes Objekt repräsentiert. Dieses speichert die Tabelleneinträge in einem sortierten Dictionary ab[2]. Zur Interpolation wird dieses Dictionary durchsucht, um das kleinste Intervall zu finden, in dem der zu interpolierende Wert liegt. Danach wird linear interpoliert.

2. Ein Dictionary ist eine Menge von Wertepaaren der Form (Schlüssel, Wert).

Eine Tabelle wird über die Relation *table* an einen Objekttyp angebunden. Für diesen wird auch lokal eine interpolate() Methode zur Verfügung gestellt, die die entsprechende Methode bei dem Tabellen-Objekt aufruft.



- **Mitwirkende Objekte**

  Klassen:

  Objekt: Für diese Klasse soll eine Tabelle zur Verfügung stehen.

  Tabelle: Stellt die eigentliche Tabellen-Funktionalität zur Verfügung.

  Instanzenvariablen / Konstanten:

  Tabelle::Tab Ein sortiertes Dictonary, das die Tabelleneinträge beinhaltet.

  Funktionen:

  Objekt::interpolate(x) Diese Funktion dient zum vereinfachten Zugriff auf die Tabelle und ruft die gleichnamige Methode bei der Tabelle auf.

  Tabelle::interpolate(x) Zum linearen Interpolieren des y-Wertes an der übergebenen x-Position.

  Tabelle::add(x, y) Diese Funktion dient zum Hinzufügen von Wertepaaren in die Tabelle. Dabei sollte die Tabelle aus Geschwindigkeitsgründen sortiert gehalten werden. Aus Platzmangel wurde die Implementation dieser Funktion nicht in der Graphik des Patterns aufgeführt.

  Tabelle::remove(x, y) Löscht ein Wertepaar aus der Tabelle.

- **Zusammenarbeit**

  Die Klasse Tabelle speichert selbst alle Informationen, die zum Auslesen von Werten nötig sind. Dadurch beschränkt sich die Zusammenarbeit zwischen einer Tabelle und einem Objekt, das die Tabelle benutzen will, auf das simple Aufrufen von Funktionen.

- **Konsequenz**

  Um die Funktionalität einer Tabelle zu erweitern (beispielsweise eine genauere Interpolationsmethode), kann ein eigenes Modell aufgestellt werden, in dem mehrere Tabellen-Typen beschrieben sind. Bei der Bindung dieses Patterns kann dann der Objekttyp Tabelle an die gewünschte Tabelle aus dem Modell gebunden werden.

- **Verwandte Patterns**
  Mit dem Pattern *Funktion (61)* kann eine Tabelle auch nachmodelliert werden.

A.2.3 Relationen

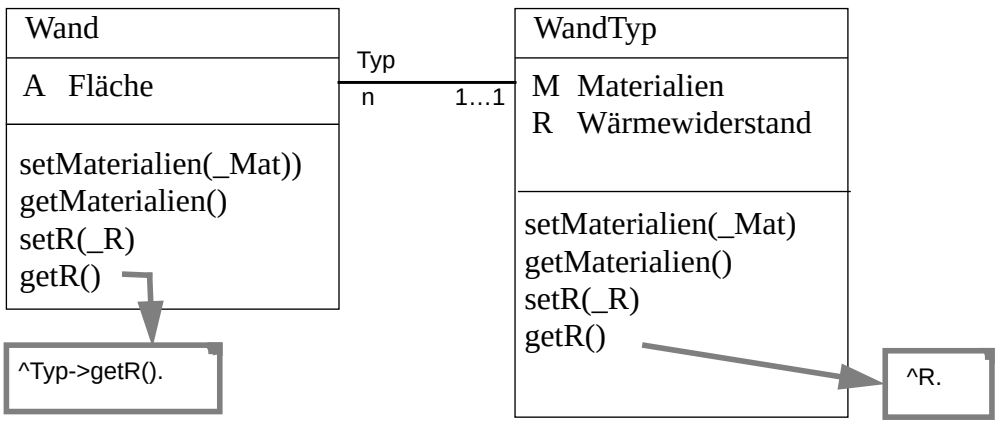
In diesem Abschnitt sind Patterns zusammengestellt, die es erlauben, Relationen zu verfolgen und Methoden bei referenzierten Objekten aufzurufen.

Einfache Indirektion

- **Zweck**
  Dieses Pattern dient hauptsächlich zur näheren Beschreibung einer Gruppe von Objekten. Jedes Objekt kann individuelle Eigenschaften (Attribute) haben und hat darüber hinaus eventuell auch noch andere, die es mit anderen Objekten teilt. Diese gemeinsamen Eigenschaften werden dann vom eigentlichen Objekt ausgelagert und in einer eigenen Klasse beschrieben. Zugriffsmethoden beim ursprünglichen Objekt sorgen für einen transparenten Zugriff auf die ausgelagerten Attribute.

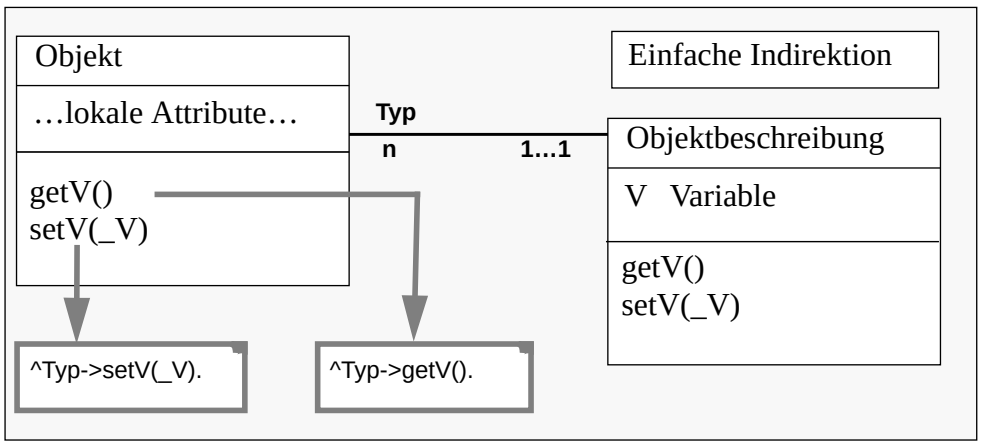
- **Bekannt unter**
  Item Description (siehe [Coa92], Seite 153)
  Dieses Pattern kann auch als Spezialisierung des *Bridge* Patterns ([GHJ95], Seite 151) aufgefaßt werden.

- **Motivation**
  Häufig ist es sinnvoll, Gruppen von Objekten zusammenzufassen. Zum Beispiel werden in einem Gebäude viele Wände vom gleichen Typ gebaut. Diese Wände bestehen alle aus den gleichen Materialien (Stein, Putz, Dämmstoff, etc.) und haben den gleichen Schichtaufbau, allerdings hat jede Wand eine individuell unterschiedliche Größe. Es ist also sinnvoll, verschiedene Wände gleichen Typs zusammenzufassen und die gemeinsamen Attribute innerhalb dieser neuen Klasse abzulegen. Graphisch läßt sich das folgerndermaßen ausdrücken:



- **Anwendbarkeit**
  Das Pattern ist anwendbar, wenn mehrere Objekte einige Attribut*werte* gemeinsam haben. Ebenso kann es angewendet werden, wenn Objekte in einer Beziehung zueinander stehen (d.h. mit einer Relation verbunden sind) und ein einfacher Datenaustausch entlang dieser Relation gewünscht ist (Stichwort: *Datenvererbung)*. Dabei kann theoretisch auch ein zyklischer oder rekursiver Datenaustausch auftreten. In diesem Fall muß noch für geeignete Abbruchmechanismen gesorgt werden.
  Die Zugriffsfunktionen des Wandtyps werden vom Generator automatisch erzeugt (siehe Pattern *Instanzenvariable (58)*). Durch die einfache Indirektion werden nur Methoden für die Wand implementiert, um auf die ausgelagerten Attribute des Wandtyps zuzugreifen. Diese Methoden rufen die entsprechenden Zugriffsfunktionen des Wandtyps auf.

- **Struktur**
  Das Pattern beruht darauf, daß es zu den individuell unterschiedlichen Objekten (hier in der Klasse *Objekt* zusammengefaßt) eine weitere Klasse *Objektbeschreibung* gibt.
  Diese beiden Klassen sind mit einer 1:1 Relation (Typ) verknüpft. Die Zugriffsfuktionen der gemeinsamen Attribute (also Instanzenvariablen) werden bei den Objekten nachgebildet, so daß auf diese Attribute genau wie auf lokale Attribute zugegriffen werden kann. Die einzelnen Objekte merken also gar nicht, wenn auf eine gemeinsame Instanzenvariable zugegriffen wird. Beim Ändern einer Objektbeschreibung ist besondere Vorsicht geboten (siehe Punkt Konsequenz).



- **Mitwirkende Objekte**

  Klassen:

  Objekt: Die ursprüngliche Klasse, für die eine Beschreibung vorliegt, die mehrere Instanzen dieser Klasse teilen.

  Objektbeschreibung: Die Beschreibung der ursprünglichen Objekte. Hier werden die Instanzenvariablen abgelegt, die von mehreren Objekten gemeinsam benutzt werden. Dies kann auch eine neue Klasse sein.

  Instanzenvariablen / Konstanten:

  V Das beschreibende Attribut.

Funktionen:

getV(), setV(_V) Zugriffsfunktionen auf die Variable V. setV() verändert eine Instanz der Objektbeschreibung, was bedeutet, daß dadurch eventuell mehrere Objekte betroffen sind.

- **Zusammenarbeit**
  Die Zugriffsfunktionen auf Attribute der Klasse Objekt sind so benannt, daß es keinen Unterschied macht, ob auf ein lokales oder ein gemeinsames Attribut zugegriffen wird. Funktionsaufrufe, die gemeinsame Attribute betreffen, werden einfach an die Objektbeschreibung weitergeleitet.
  Die Objektbeschreibung ihrerseits braucht keine Kenntnis davon zu haben, für welche Objekte sie als Beschreibung dient.

- **Konsequenz**
  Benutzen mehrere Objekte dieselben Attributwerte (das heißt, die Objekte sind vom selben Typ), so können die Attribute in eine eigene Klasse ausgelagert werden. Dadurch brauchen diese Attribute nicht bei jedem Objekt desselben Typs abgespeichert zu werden; Redundanzen in der Datenbasis werden also vermieden. Vorsicht ist jedoch beim Ändern dieser gemeinsamen Attribute geboten: bei einer Änderung wird immer die gesamte Objektbeschreibung geändert, was Auswirkungen auf andere Objekte desselben Typs hat. Beim Schreiben auf ein gemeinsames Attribut muß also immer bedacht werden, was damit erreicht werden soll. Soll die Objektbeschreibung an sich geändert werden (zum Beispiel weil *alle* Außenwände eine zusätzliche Isolierung bekommen), so kann dieses direkt geschehen. Ändert sich hingegen nur ein Objekt (*eine* Außenwand wird verstärkt) so muß zunächst für dieses Objekt die Objektbeschreibung kopiert werden und kann dann erst abgeändert werden.
  Zusätzliche Vorsicht ist bei der Instanziierung der Objekte geboten. Es muß dafür gesorgt werden, daß alle Objekte mit der richtigen Objektbeschreibung verbunden werden. Eventuell kann es auch sinnvoll sein, mehrere gleiche Objektbeschreibungen zu haben. Dadurch können gleichartige Objekte in Clustern zusammengefaßt werden, und die Änderung einer Clusterbeschreibung wirkt sich dann nur auf diesen aus.

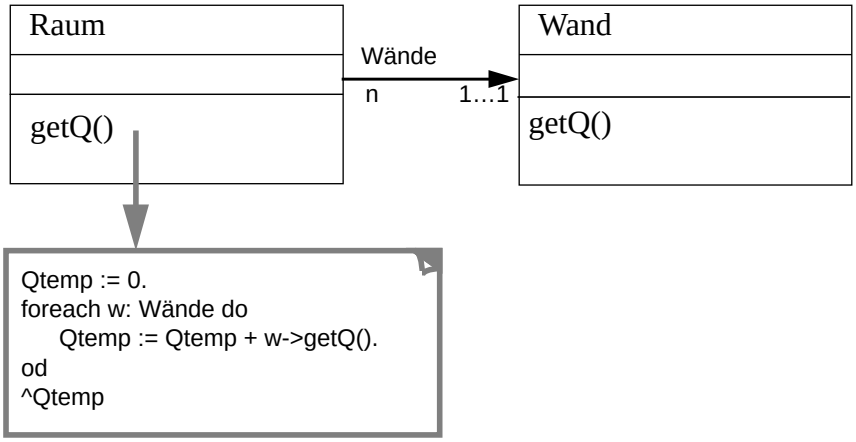
- **Verwandte Patterns**
  Komplexe Indirektion (69) - Verfolgen einer 1:n Relation.

Komplexe Indirektion

- **Zweck**
  Oftmals haben einzelne Objekte Beziehungen zu mehreren anderen Objekten einer Klasse. Dieses Pattern ermöglicht den Zugriff auf einzelne Attribute der referenzierten Objekte und verrechnet diese zu einem Gesamtwert (Summe, Durchschnitt, etc.).

- **Motivation**
  1:n Relationen treten an vielen Stellen eines Modells auf. Zum Beispiel grenzen an einen Raum mehrere Wände. Zur Simulation der Temperatur in einem Raum muß die Gesamt-

wärmemenge ermittelt werden, die in (bzw. aus) dem Raum fließt. Diese Gesamtwärmemenge ist einfach die Summe der Wärmemengen, die durch die einzelnen Wände fließen.
Mit Hilfe der komplexen Indirektion können für alle Wände eines Raumes diese Wärmemengen berechnet werden (durch den Aufruf einer entsprechenden Funktion), und anschließend werden die erhaltenen Werte aufsummiert.

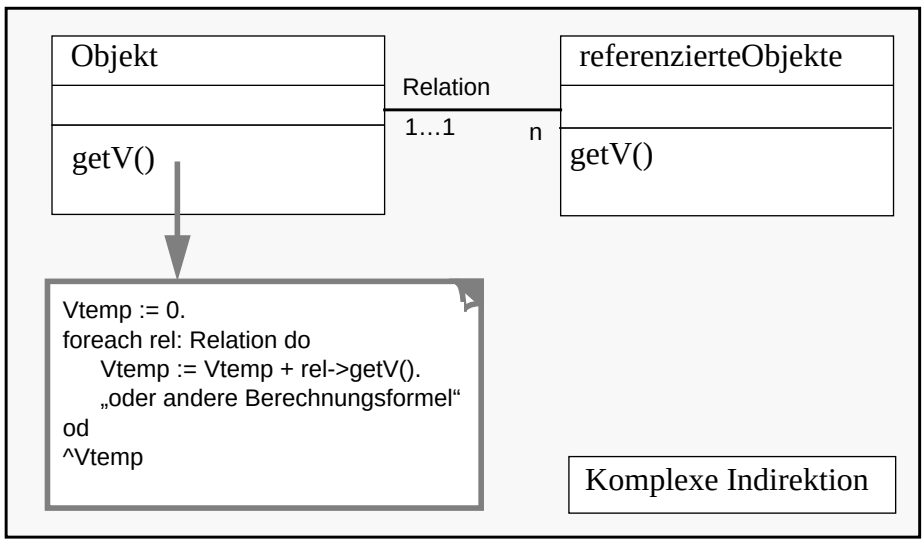


Die Funktion *getQ()* des Raumes (s. Abbildung) sorgt dabei für die Abfrage der einzelnen Wärmemengen der referenzierten Wände und summiert diese Werte auf.
Die Funktion *getQ*() der Wand wird in diesem Pattern nicht näher beschrieben, da es in der Verantwortung der Wände steht, wie sie ihre Wärmemengen ermitteln (durch Berechnungsformeln, Instanzenvariablen o.ä.).

- **Anwendbarkeit**
  Das Pattern kann zur Verfolgung beliebiger 1:n Relationen herangezogen werden. Mögliche Varianten zur Berechnung des resultierenden Wertes sind *Summe*, *Produkt*, *Mittelwert*, *Standardabweichung* und *Varianz*.

- **Struktur**
  Dieses Pattern ist sehr einfach aufgebaut. Ein Objekt referenziert mehrere Objekte einer anderen (oder auch derselben) Klasse. Bei diesem Objekt wird eine Funktion zur Verfügung gestellt, mit der bei allen referenzierten Objekten eine bestimmte Methode aufgerufen werden kann. Hat diese Methode einen Rückgabewert, so können die erhaltenen Werte zu einem Gesamtwert verrechnet werden. Welche Berechnungsformel dabei zum Einsatz kommt, wird bei der Bindung des Patterns festgelegt. Möglich sind dabei Formeln wie Summe, Durchschnitt, Produkt oder Standardabweichung.

Die Methode, die bei den referenzierten Objekten aufgerufen wird, muß separat modelliert werden.



- **Mitwirkende Objekte**

  Klassen:

  Objekt: Das „Ursprungsobjekt“ von dem aus die 1:n Relation verfolgt werden soll.

  referenzierteObjekte: Die referenzierten Objekte. Werden diese beiden Klassen an dieselbe Modell-Klasse gebunden, so muß darauf geachtet werden, daß keine Zyklen bei der Referenzierung auftreten.

  Funktionen:

  Objekt::getV() Diese Methode wird aufgerufen, um alle verbundenen Objekte zu durchlaufen.

  referenzierteObjekt::getV() Bei jedem referenzierten Objekt wird die Funktion getV() aufgerufen. Diese muß an geeigneter Stelle genauer spezifiziert werden.

- **Zusammenarbeit**

  Die Relation zwischen den beiden Objekttypen wird nur in einer Richtung verfolgt. Die Objekte werden in einer beliebigen Reihenfolge - je nach Implementierung der Relation durch das Pattern *Relation (59)* - durchlaufen.

- **Konsequenz**

  Vorsicht ist geboten, falls zyklische Referenzierungen auftreten können. Dann muß für Abbruchkriterien gesorgt werden, damit keine Endlos-Schleifen auftreten.

- **Verwandte Patterns**

Einfache Indirektion (67) - Verfolgen einer 1:1 Relation.

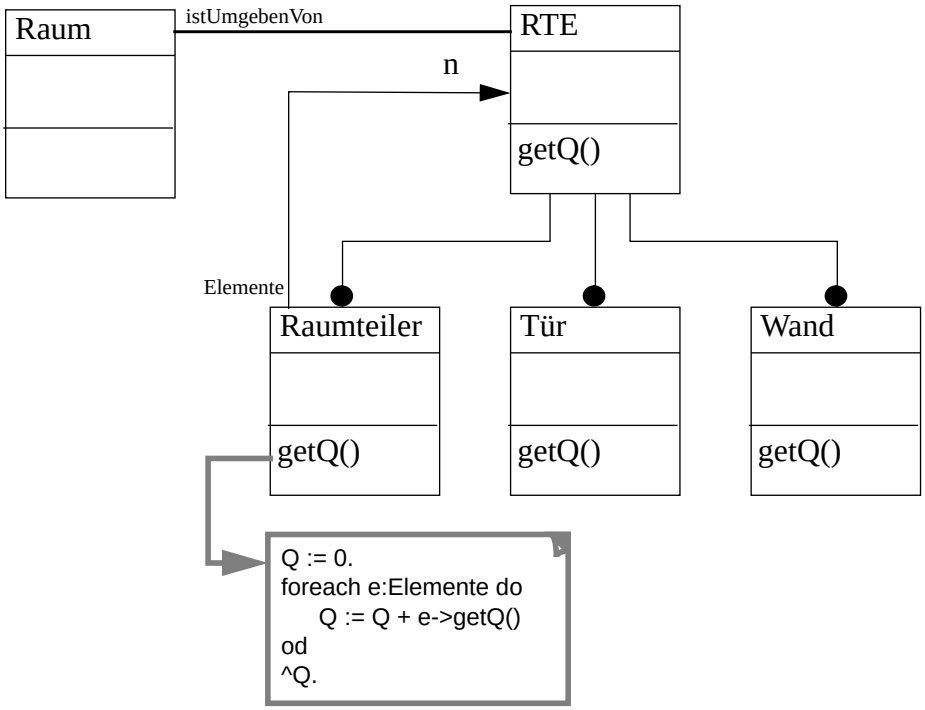
A.2.4 Strukturierungen

Diese Patterns fassen unterschiedliche Modell-Klassen zusammen und gruppieren sie in bestimmte Strukturen.

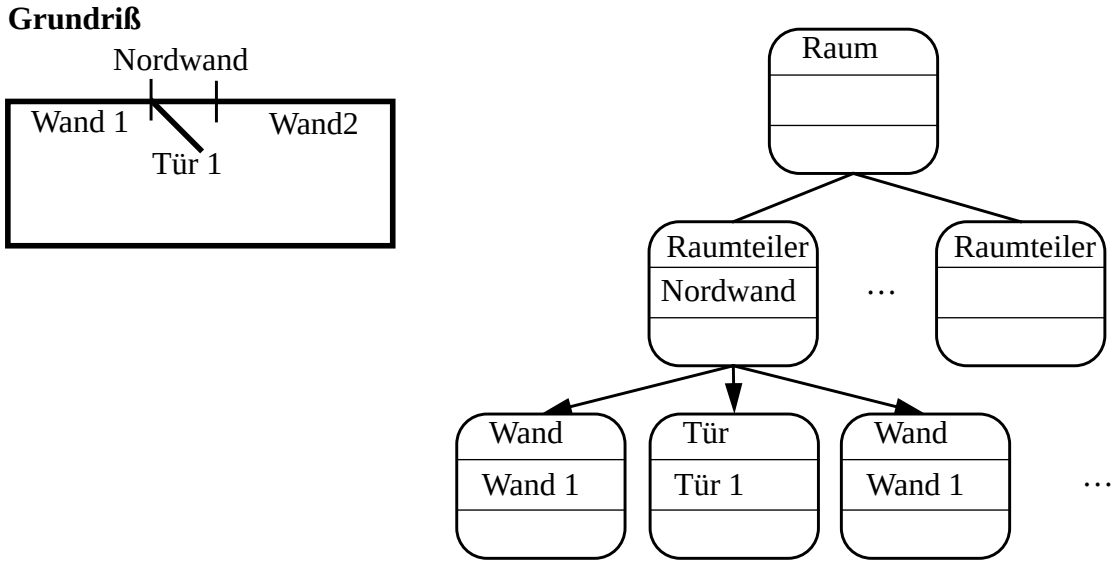
Komposition

- **Zweck**
  Gruppiert unterschiedliche Objekte in einer Baumstruktur.
- **Bekannt unter**
  Composite [GHJ95] und in leicht abgewandelter Form als
  Rekursive Aggregation [RBP91].
- **Motivation**
  Häufig treten in Applikationen heterogene Mengen auf. Dabei sollte auf die einzelnen Objekte in der Menge mit gleichen Methoden zugegriffen werden. Jedes Objekt der Menge kann dann diese Methoden unterschiedlich abarbeiten (Polymorphie).
  Zum Beispiel wird der Wärmedurchgang durch Wände und geschlossene Türen gleich berechnet. Ist eine Tür hingegen offen, so treten zusätzlich noch andere Effekte auf (Austausch von Luftmassen). Der Wärmedurchgang durch eine Tür muß also gesondert berechnet werden. Einen Raum, der von Wänden und Türen umgeben ist, interessiert nur das Ergebnis der Wärmedurchgangsberechnung; die genauen Berechnungsformeln sollten nur der Wand bzw. der Tür bekannt sein. Um diese Situation in den Griff zu bekommen, kann man eine abstrakte Klasse Raumteilerelement (RTE) definieren. Diese Klasse stellt eine Funktionsschnittstelle zur Berechnung des Wärmedurchgangs zur Verfügung. Die Klassen *Wand* und *Tür* sind von der Klasse *RTE* abgeleitet und implementieren die entsprechenden Berechnungsformeln.

Um jetzt eine Menge von Wänden und Türen zusammenfassen zu können, braucht man eine weitere Klasse *Raumteiler*. Ein Raumteiler ist selbst wieder ein *RTE* und aggregiert zusätzlich mehrere Raumteilerelemente (siehe folgendes Bild).

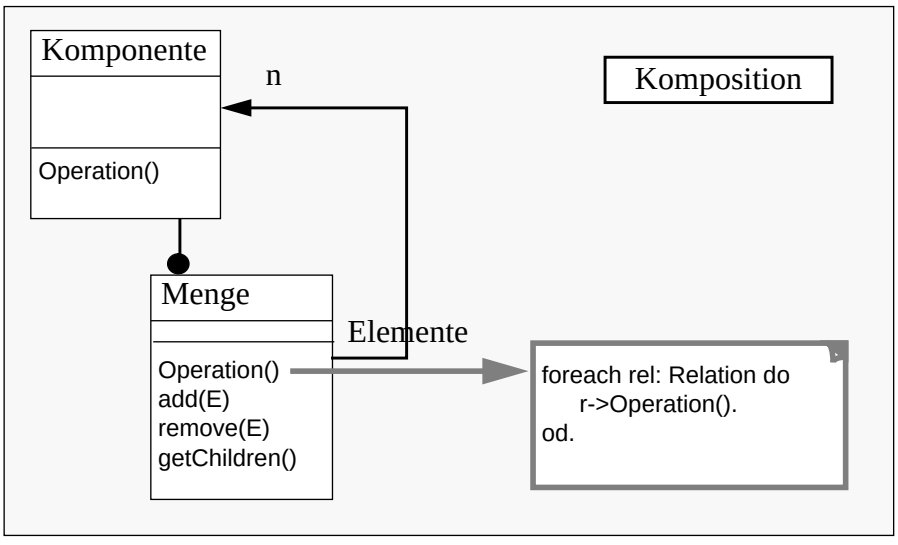


Wird ein Raumteiler aufgefordert, einen Wärmedurchgang zu berechnen, so leitet er diese Aufforderung weiter (d.h. er ruft die entsprechenden Funktionen seiner aggregierten Teile auf) und liefert die Summe dieser Teilergebnisse zurück. Zur Verrechnung der Teilergebnisse können auch andere Formeln verwendet werden (z.B Produkt, Mittelwert, etc.).
Jetzt ist ein Raum nicht mehr von mehreren unterschiedlichen Objekten umgeben. Jedes Objekt, das den Raum abgrenzt, reagiert auf dieselben Nachrichten. Insbesondere können jetzt auch Wände und Türen beliebig hierarchisch geschachtelt sein (siehe das Objektdiagramm in der nächsten Abbildung).



- **Anwendbarkeit**
  Das Pattern wird angewendet, um Unterschiede zwischen einzelnen Objekten einer heterogenen Menge zu vereinheitlichen. Alle Objekte einer solchen Menge können gleichartig angesprochen werden.

- **Struktur**
  Das Pattern beruht darauf, daß es zu den individuell unterschiedlichen Objekten (hier in der Klasse *Objekt* zusammengefaßt) eine weitere Klasse *Objektbeschreibung* gibt.
  Diese beiden Klassen sind mit einer 1:1 Relation verknüpft. Die Zugriffsfuktionen der gemeinsamen Attribute (also Instanzenvariablen) werden bei den Objekten nachgebildet, so daß auf diese Attribute genau wie auf lokale Attribute zugegriffen werden kann. Die einzelnen Objekte merken also gar nicht, wenn auf eine gemeinsame Instanzenvariable zugegriffen wird.



Die Methode *Menge::Operation()* kann mehrere unterschiedliche Ausprägungen haben. Neben dem Aufrufen von Prozeduren können auch Funktionen mir Rückgabewerten aufgerufen und die erhaltenen Werte anschließend verrechnet werden. Mögliche Berechnungen sind *Summe*, *Produkt*, *Mittelwert*, *Standardabweichung* und *Varianz* (vergleiche Pattern *Komplexe Indirektion (69)*).

- **Mitwirkende Objekte**

  Klassen:

  Komponente: Diese abstrakte Klasse dient als Oberklasse für alle Objekte, die in der Menge aufgenommen werden sollen. Im wesentlichen stellt sie die Methode *operation()* zur Verfügung, die für alle Objekte einer konkreten Menge aufgerufen werden kann.

  Menge: In einer Menge können mehrere Objekte der Klasse *Komponente* abgelegt werden. Dies können zum einem weitere Mengen sein oder es handelt sich um Blätter der Baumstruktur.

  Funktionen:

  Komponente::Operation() Abstrakte Methode, die bei allen Objekten der Menge vorhanden ist. Sie muß überladen werden, um den Objekten die gewünschte Funktionalität zu geben.

  Menge::Operation() Bei einer Menge wurde die Funktion *Operation()* bereits so überladen, daß automatisch die jeweiligen *Operate()* Funktionen aller Objekte in der Menge aufgerufen werden.

  Menge::add(E) Nimmt ein neues Objekt in eine Menge auf. Dies kann auch wiederum eine weitere Menge sein.

  Menge::remove(E) Löscht ein Objekt aus einer Menge.

  Menge::getChildren() Liefert alle *direkten* Kinder einer Menge. Handelt es sich dabei um weitere Mengen, so muß für diese wiederum die *getChildren()* Funktion aufgerufenwerden, um *alle* Kinder der Menge zu erhalten.

- **Zusammenarbeit**

  Die Instanzen der Klasse Objekt arbeiten auf den gemeinsamen Attributen so, als ob sie lokal wären. Die Zugriffsfunktionen sind derart definiert, daß keine lokalen Attribute verändert oder gelesen werden, sondern es wird der Zugriff an die Objektbeschreibung weitergeleitet.

- **Konsequenz**

  Die Anwendung des Patterns hat sowohl Konsequenzen für den (schreibenden) Zugriff auf gemeinsame Attribute, als auch für die Instanziierung.

  Beim Schreiben eines gemeinsamen Attributes muß immer bedacht werden, daß eventuell andere Objekte auch noch denselben Attributwert haben. Das Schreiben eines Attributes beeinflußt also immer eine ganze Gruppe von Objekten.

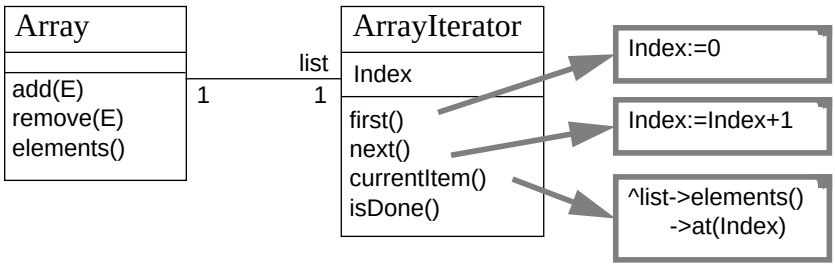
  Bei der Instanziierung der Objekte muß darauf geachtet werden, daß die passenden Objektbeschreibungen auch immer richtig mit den Objekten verknüpft werden.

- **Verwandte Patterns**

  Iterator (76) zum Iterieren über eine Menge.

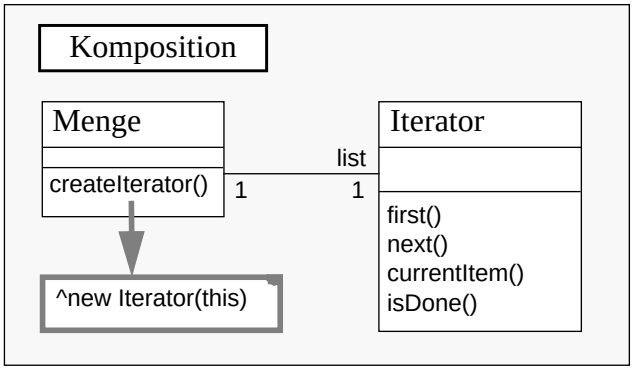
Iterator

- **Zweck**
  Dieses Pattern dient zur schrittweisen Verfolgung der Objekte in einer Menge. Die Reihenfolge, in der die Objekte abgearbeitet werden, kann dabei in einem speziellen Iterator festgelegt werden.
- **Bekannt unter**
  Iterator aus [GHJ95].
- **Motivation**
  Häufig müssen Objekte einer Menge in einer bestimmten Reihenfolge verfolgt werden. Würde man die Reihenfolge der Objekte in der Menge bei dem Mengen-Objekt vorgeben, so wird das Interface dieser Menge sehr schnell recht komplex und unübersichtlich. Statt dessen kann man eine neue (abstrakte) Klasse *Iterator* einführen, die sich um die schrittweise Abarbeitung der Objekte kümmert. Ein spezieller Iterator kümmert sich um die gewünschte Reihenfolge. So können für ein und dieselbe Menge mehrere Iteratoren existieren, die die Elemente der Menge in unterschiedlichen Reihenfolgen abarbeiten. Um die Elemente eines Arrays zu verfolgen, kann folgendes modelliert werden:



- **Anwendbarkeit**
  Das Pattern lohnt sich überall dort zu verwenden, wo in unterschiedlichen Reihenfolgen auf Objekte einer Menge zugegriffen werden soll. Wird eine Menge immer in derselben Reihenfolge durchlaufen, so kann dafür auch eine einfache Funktion bei der Menge selbst zur Verfügung gestellt werden. Wenn die Menge durch eine 1:n Relation repräsentiert wird, kann sie alternativ auch mit dem Pattern *Komplexe Indirektion (69)* durchlaufen werden.
- **Struktur**
  Das Pattern besteht aus einer Klasse, die die Menge repräsentiert, und einer abstrakten Iterator-Klasse. Da die Iterations-Reihenfolge erst im konkreten Iterator festgelegt werden soll, muß solch ein konkreter Iterator von der Iterator-Klasse dieses Pattern abgeleitet und an anderer Stelle separat modelliert werden. Auch die konkrete Implementation der Menge wird mit diesem Pattern nicht beschrieben, da sie prinzipiell vom Iterator unabhängig sein sollte.
  Beim Anlegen eines neuen Iterators muß die Menge, über die iteriert werden soll, bereits vollständig bekannt sein. Daher stellt die Klasse *Menge* eine Funktion *createIterator()* zur

Verfügung, die einen neuen Iterator anlegt. Dem Konstruktor des Iterators wird dabei die aktuelle Menge als Parameter übergeben.



- **Mitwirkende Objekte**

  Klassen:

  Menge: Die Menge, über die iteriert werden soll.

  Iterator: Kapselt die eigentliche Iteration und liefert Schritt für Schritt das jeweils nächste Element der Menge zurück (über die Funktionen *next()* und *currentItem()*).

  Funktionen:

Menge::createIterator()	Legt einen neuen Iterator über der aktuellen Menge an.
Iterator::first()	Setzt den Iterator auf das erste Element der Menge zurück.
Iterator::next()	Liefert das jeweils nächste Element der Menge.
Iterator::currentItem()	Gibt das aktuelle Element aus der Menge zurück.
Iterator::isDone()	Gibt an, ob alle Elemente der Menge durchlaufen worden sind.

- **Zusammenarbeit**

  Der Iterator merkt sich jeweils das aktuelle Element der Menge und kann das jeweils nachfolgende Element berechnen.

- **Konsequenz**

  Durch die spezielle Iterator-Klasse wird das Interface einer Menge vereinfacht. Die Aufgabe das jeweils nächste Element der Menge zu berechnen, wird dem Iterator überlassen. Dadurch ist es möglich, eine Menge mit mehreren verschiedenen Iteratoren in unterschiedlichen Reihenfolgen zu durchlaufen.

- **Verwandte Patterns**

  Komplexe Indirektion (69) zum einfachen Durchlaufen einer 1:n Relation.

Anhang B

Literaturverzeichnis

[ACL96] Alencar, P.S.C.; Cowan, D. D.; Lichtner, K. J.; Lucena, C. J. P., Nova, L. C. M.: „Tool Support for Design Patterns“, University of Waterloo, Waterloo, 1996 (http://csg.uwaterloo.ca/~stafford/ADV/theroy.html)

[AIS77] Alexander, C.; Ishikawa, S.; Silverstein, M.: „A Pattern Language“, Oxford University Press, 1977

[Boo90] Booch, G.: „Object-Oriented Design with Applications“, Redwood City, CA: Benjamin-Cummings, 1990

[BFV96] Budinsky, F.J.; Finnie, M. A.; Vlissides, J.M.; Yu, P.S.: „Automatic code generation from design patterns“, IBM Systemn Journal, Vol. 35, No. 2, 1996 (http://www.almaden.ibm.com/journal/sj/budin/budinsky.html)

[Che94] Chen, D.J.; Chen, D.T.K.: „An experimental study of using reusable software design frameworks to archieve software reuse“, Journal Of Object-Oriented Programming, 7(2), 1994, Seiten 56-67

[Che76] Chen, P. P.: „The entity-relationship model: Toward a unified view of data”. In „ACM Transactions on Database Systems, Vol. 1”, Seiten 9-36, 1976.

[Coa92] Coad, P.: „Object-Oriented Patterns“, Communications of the ACM, Vol. 35, No. 9, 1992, Seiten 152-158.

[Geu95] Geuer, E.: „Prototypische Implementierung einer flexiblen, generierbaren Simulationsumgebung“, Diplomarbeit, Universität Kaiserslautern, 1995

[GHJ95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: „Design Patterns“, Addison-Wesley, 1995

[Hei96] Heister, F.: „Simulator-Kernel-Dokumentation“, Interne Arbeit der AG „VLSI Entwurf und Architektur“, Universität Kaiserslautern, 1996

[KrP88] Krasner, G. E.; Pope, S. T.: „A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80“, Journal of Object-Oriented Programming, 1(3), August/September 1988, Seiten 26-49

[LJK94] Lutz, P.; Jenisch, R.; Klopfer, H.; Freymuth, H.; Krampf, L.; Petzold, K.: „Lehrbuch der Bauphysik“, Teubner Verlag, Stuttgart, 1994

[MaM89] Mattern, F.; Mehl, H.: „Diskrete Simulation - Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung“, Informatik Spektrum ‘89, Seiten 198-210, 1989

[Nee87] Neelamkavil, F.: „Computer Simulation and Modelling“, John Wiley & Sons Ltd., Großbritannien, 1987

[Pre95] Pree, W.: „Design Patterns for Object-Oriented Software Development“, ACM Press, Addison-Wesley, 1995

[PDr80] Puschmann, Drath: „Technische Wärmelehre“, 25. Auflage, Darmstadt, Fikentscher, 1980

[RBP91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: „Object-Oriented Modeling and Design“, Prentice Hall, Englewood Cliffs, N.J., 1991

[Sah96] Sahler, A.: „Analyse und Realisierung einer Notation zur Erstellung von Gebäudemodellen“, Diplomarbeit, Universität Kaiserslautern, 1996

[SFB94] Sonderforschungsbereich 501, Finanzierungsantrag 1995-1996-1997 „Entwicklung großer Systeme mit generischen Methoden“, Universität Kaiserslautern, 1994

[SFB96] Altmeyer, J.; Schürmann, B.; Schütze, M.; Zimmermann, G.: „Generator-Based Reuse of Common Models“, Bericht 3/1996 im Sonderforschungsbereich 501, Universität Kaiserslautern, 1996

[Sou95] Soukup, J.: „Pattern Languages of Program Design; Chapter 20: Implementing Patterns“, Addison-Wesley, 1995
(http://www.codefarms.com/papers/patterns.html)

[SSA94] Schütze, M.; Schürmann, B.; Altmeyer, J.: „Generating Abstract Datatypes with Remote Acceess Capabilities“, Proceedings EDAF 1994, Porto Alegre, 1994

[Vis95] „Visual Works 2.5 Cookbook“, ParcPlace-Digitalk Inc., Sunnyvale, CA, 1995