Studienarbeit
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Automatische Generierung von UML-Zustandsdiagrammen aus Java-Klassen

Matthias Witt

Betreuer:
Prof. Dr.-Ing. Heinz Züllighoven

Inhaltsverzeichnis

1. Einleitung

Das Verhalten von Klassen in objektorientierten Software-Systemen lässt sich durch Zustandsdiagramme (Statecharts) beschreiben. Zustandsdiagramme sind Teil der Unified Modeling Language[1], wo sie zur Modellierung der dynamischen Aspekte eines Systems vorgesehen sind. In dieser Arbeit ist das zu untersuchende System immer eine Klasse. Das Zustandsdiagramm dieser Klasse zeigt an, welche Zustände ein Objekt einnehmen kann, das Exemplar der Klasse ist, und welche Zustandsübergänge beim Aufruf einer Methode erfolgen.
UML-Diagramme werden vorwiegend in der Modellierungsphase eingesetzt. Im Regelfall werden zunächst diverse UML-Diagramme gezeichnet und danach wird mit der Programmierung begonnen. Bezogen auf Zustandsdiagramme heißt das, dass erst das Diagramm entworfen und anschließend die zugehörige Klasse implementiert wird.[2]
In dieser Arbeit soll aber der umgekehrte Weg gegangen werden: Wir gehen von einer fertig programmierten Klasse aus und wollen zu dieser ein Zustandsdiagramm erzeugen. Dieses Diagramm kann dann dazu dienen, das Verhalten der Klasse zu überprüfen.
Zu diesem Vorhaben äußern sich die Entwickler der UML folgendermaßen:

> *Reverse engineering* (the creation of a model from code) is theoretically possible, but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams.[3]

Sie gehen also davon aus, dass es keine Möglichkeit gibt zu entscheiden, was ein sinnvoller Zustand ist. Die übliche Definition des Objektzustands ist die momentane Belegung der Attribute. Da in diesem Fall bei weitem zu viele Zustände entstehen, ist eine geeignete Zusammenfassung von Belegungen unumgänglich. Die Schwierigkeit besteht nun darin, herauszufinden, welche Zusammenfassungen sinnvoll sind.
Eine weitere Hürde ist die Bestimmung des Zustands, in dem sich das Objekt nach der Ausführung einer Methode befindet. Hierzu muss herausgefunden werden, welche Belegungen die Attribute nach der Methodenausführung haben können. Bei einer Sprache mit Turing-Mächtigkeit ist allerdings nicht entscheidbar, was eine beliebige Methode berechnet.[4]
Auch wenn diese Probleme behoben werden können, wenn Termination stillschweigend vorausgesetzt wird, so wäre eine Bestimmung aller möglichen Attributbelegungen zu aufwändig und würde das Problem der Zustandsbestimmung nicht lösen.
Wenn also keine weiteren Voraussetzungen gemacht werden, ist die automatische Generierung eines sinnvollen Zustandsdiagramms in der Tat unmöglich. Wir müssen demnach davon ausgehen, dass der Programmierer uns nähere Hinweise über das Zustandsverhalten der Methoden gibt.
Genau diese Hinweise liefern die Zusicherungen, die im Rahmen des Vertragsmodells angegeben werden.[5] Der Programmierer formuliert zu jeder Methode Vor- und Nachbedingungen, die Aufschluss darüber geben, in welchen Zuständen sich das Objekt vor bzw. nach Ausführung der Methode befinden kann. Nehmen wir diese Zusicherungen als

---

[1] UML, siehe [BRJ 00]
[2] In der Praxis werden Zustandsdiagramme jedoch eher selten eingesetzt. Klassen- und Anwendungsfalldiagramme kommen weit häufiger zum Einsatz.
[3] [BRJ 00], S. 339
[4] siehe [Schöning 01]
[5] siehe [Meyer 92] oder [Meyer 97]

Grundlage, so kann es uns gelingen, ein sinnvolles und aussagekräftiges Zustandsdiagramm zu generieren.
Die Programmiersprache, die in dieser Arbeit Verwendung findet, ist Java.[6] Diese Wahl ist jedoch nicht von entscheidender Bedeutung, da im Wesentlichen die Zusicherungen der Methoden betrachtet werden, die, wie wir noch sehen werden, lediglich in Kommentaren stehen. Eine Übertragung der Beispiele auf andere objektorientierte Programmiersprachen ist leicht möglich; auch das im 4. Kapitel entwickelte Verfahren zur automatischen Generierung von Zustandsdiagrammen setzt keine bestimmte Programmiersprache voraus. Die Syntax der Zusicherungen ist allerdings sprachabhängig, da es sich um boolesche Ausdrücke handelt.
Der Aufbau dieser Arbeit ist wie folgt: Das 2. Kapitel gibt eine Einführung in Zustandsdiagramme. Dabei wird sowohl auf deren Syntax als auch auf die Art der Verwendung in unserem Kontext eingegangen.
In Kapitel 3 werden Zusicherungen vorgestellt, und zwar sowohl die allgemeinen Konzepte als auch die programmiersprachliche Formulierung, die in dieser Arbeit vorausgesetzt wird. Hierbei wird bereits angedeutet, wie Zustände und Zustandsübergänge durch Zusicherungen beschrieben werden. Die Zusicherungen werden in verschiedene Kategorien unterteilt, was im weiteren Verlauf von Nutzen ist.
Das 4. Kapitel ist das zentrale Kapitel dieser Arbeit. Es wird ein Verfahren vorgestellt, mit dessen Hilfe sich aus den gegebenen Zusicherungen ein Zustandsdiagramm erzeugen lässt. Dazu werden die Zusicherungen mathematisch modelliert. In einigen Schritten werden dann mathematische Beschreibungen der Zustände und Zustandsübergänge entwickelt.
Auf die praktische Anwendung dieses Verfahrens wird in Kapitel 5 eingegangen. Zunächst wird das Programm class2statechart beschrieben, das im Rahmen dieser Arbeit entstanden ist und die automatische Generierung eines Zustandsdiagramms zu einer beliebigen Java-Klasse durchführt. Anschließend wird untersucht, wie die erzeugten Zustandsdiagramme von bereits bestehenden Klassen aussehen, um so den Nutzen des Verfahrens abzuschätzen. Hierzu wurde das JWAM-Rahmenwerk[7] (Version 1.6.0) untersucht, das am Arbeitsbereich Softwaretechnik im Fachbereich Informatik der Universität Hamburg entstanden ist. In JWAM werden Zusicherungen durchgehend verwendet (aus diesem Grund werden auch in Kapitel 3 viele Beispiele aus JWAM angeführt).
Das 6. Kapitel bietet schließlich einen Ausblick auf weitere mögliche Fragestellungen. Es werden einige Aspekte von Zusicherungen und Zustandsdiagrammen genannt, die bisher nicht verwendet wurden, sich aber für die Integration in das entwickelte Verfahren anbieten.

[6] siehe [GJSB 00]
[7] ein Rahmenwerk in Java nach dem Werkzeug-Automat-Material-Ansatz (http://www.jwam.de; zum WAM-Ansatz siehe [Züllighoven 98])

2. Zustandsdiagramme

Dieses Kapitel gibt eine Einführung in Zustandsdiagramme (Statecharts). Es wird erläutert, was Zustandsdiagramme sind und wie sie in der objektorientierten Programmierung verwendet werden. Außerdem wird noch kurz darauf eingegangen, wie Zustände charakterisiert werden.

2.1 Einführung in Zustandsdiagramme

Zustandsdiagramme wurden von David Harel entwickelt um komplexe Systeme beschreiben zu können. In [Harel 87] erklärt er die Syntax von Zustandsdiagrammen und nennt die zugrunde liegenden Ideen.
Dabei ist das Grundprinzip das des endlichen Automaten. Es gibt Zustände, in denen das System verweilen kann, und Zustandsübergänge (Transitionen).
Zustandsdiagramme sind endliche Automaten mit einigen Erweiterungen. Reine endliche Automaten eignen sich nicht um komplexere Systeme zu beschreiben, da sie schwer lesbar und schlecht erweiterbar sind.
Die wesentlichen Erweiterungen sind:

- hierarchische Zustände
- orthogonale Zustände
- Auslösen von Ereignissen
- History-Funktionen

Die Zustandsübergänge können durch verschiedene Ereignisse hervorgerufen werden und selbst wieder Ereignisse auslösen.
Harel dachte zunächst nicht an objektorientierte Programmierung. Der Titel von [Harel 87] („Statecharts: A visual formalism for complex systems“) verdeutlicht, worum es ihm ging: Zustandsdiagramme waren erstens ein visueller Formalismus (also leicht vom Menschen lesbar) und zweitens ein Mittel zum Beschreiben komplexer Systeme. Das steht ein wenig im Gegensatz zu unseren Zielen. Wir verwenden Zustandsdiagramme um Java-Klassen zu beschreiben. Java-Klassen sind aber im Allgemeinen nicht sehr komplex aufgebaut. Wenn wir also Zustandsdiagramme aus Java-Klassen generieren, dann reicht es aus, sich auf flache Zustandsdiagramme[8] zu beschränken, damit unser Verfahren nicht unnötig kompliziert wird. Wir werden später sehen, dass die generierten Zustandsdiagramme im Regelfall nicht zu komplex werden.

2.2 Die Syntax von Zustandsdiagrammen

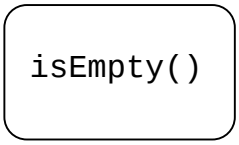
Zustandsdiagramme gibt es in mehreren Varianten. Für die Objektorientierung gibt es beispielsweise die so genannten Objectcharts[9] oder Zustandsdiagramme im Rahmen von OMT.[10]

[8] Unter flachen Zustandsdiagrammen versteht man solche, die weder hierarchische noch orthogonale Zustände enthalten.
[9] siehe [CHB 92] und Abschnitt 3.2.1
[10] siehe [Rumbaugh 95]

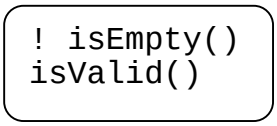
Wir verwenden hier Zustandsdiagramme, wie sie in der UML vorkommen.[11] Die UML hat sich mittlerweile zur universellen Beschreibungssprache in der Objektorientierung und darüber hinaus entwickelt.
Im Folgenden werden diejenigen Bestandteile von Zustandsdiagrammen vorgestellt, die im weiteren Verlauf dieser Arbeit Verwendung finden. Im Wesentlichen handelt es sich hierbei um die elementaren Bausteine.

2.2.1 Zustände

Zustände werden durch abgerundete Rechtecke dargestellt. Die Zustände können Namen bekommen, die dann innerhalb des abgerundeten Rechtecks stehen:



Das Objekt, welches durch das Zustandsdiagramm beschrieben wird, befindet sich zu jedem Zeitpunkt in genau einem Zustand. Befindet sich das Objekt im oben gezeichneten Zustand, so bedeute dies, dass zu diesem Zeitpunkt `isEmpty()` gilt.
Mehrere Ausdrücke, die in einem Zustand gelten, schreiben wir untereinander:

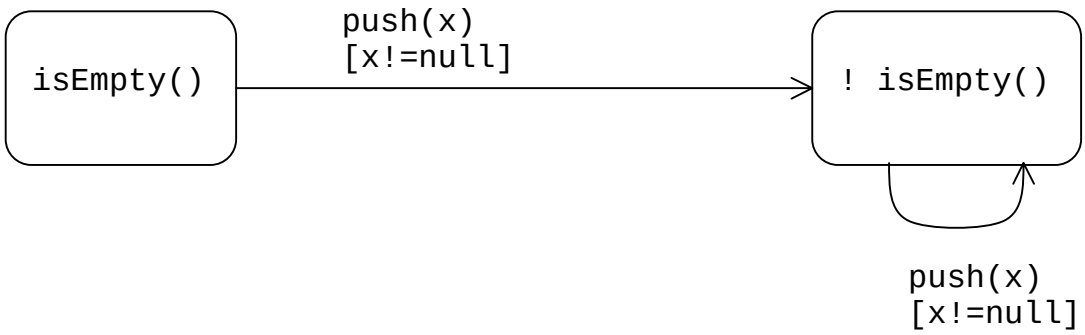


In diesem Zustand gilt sowohl `!isEmpty()` als auch `isValid()`. Man hätte auch eine einzige Zeile mit einem und-verknüpften Ausdruck verwenden können; die obige Schreibweise ist jedoch praktikabler, da bei längeren Ausdrücken beide Dimensionen genutzt werden.

2.2.2 Transitionen

Transitionen beschreiben Zustandsübergänge. Sie führen von einem Ursprungszustand zu einem Zielzustand (welche auch identisch sein können) und werden durch einen Pfeil dargestellt. Die Transition wird mit einem auslösenden Ereignis beschriftet, welches bei uns eine Methode der Klasse ist. Zusätzlich kann noch in eckigen Klammern eine so genannte „Guard-Bedingung“ angegeben werden. Das ist ein boolescher Ausdruck, der wahr sein muss, damit der Zustandsübergang stattfinden kann.

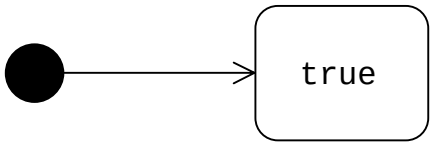
[11] siehe [BRJ 00]



In diesem Beispiel führt der Aufruf der Methode `push` mit dem Parameter `x` zu einem Zustandsübergang in den Zustand `!isEmpty()`, und zwar unabhängig vom Zustand vor dem Aufruf. In jedem Fall gibt es eine Guard-Bedingung, die besagt, dass der Zustandsübergang nur möglich ist, wenn `x!=null` gilt.

2.2.3 Anfangszustände

Der erste Zustand, den das Objekt einnimmt, ist der Anfangszustand. Das ist ein ausgezeichneter Zustand, der nicht Zielzustand einer Transition sein kann. Der Anfangszustand wird durch einen kleinen ausgefüllten Kreis dargestellt. Hier sehen wir einen Anfangszustand mit einer Transition zu einem gewöhnlichen Zustand:



(Wenn neben dem Anfangszustand nur ein einziger Zustand existiert, können wir diesen mit `true` bezeichnen; das ist die Bedingung, die immer erfüllt ist.)
Führt der Anfangszustand zu mehreren anderen Zuständen, so werden wir mehrere Anfangszustände mit jeweils einer Transition zeichnen. Dies erhöht die Übersicht im Diagramm, weil sich viele lange Pfeile, die sich überschneiden, schwer verfolgen lassen.
In der UML sind außerdem noch Endzustände definiert. Endzustände verwenden wir jedoch nicht, da es in Java keine Destruktoren gibt.[12] Objekte lassen sich nicht explizit löschen; wird ein Objekt nicht mehr benötigt, so entfernt man einfach die Referenzen auf dieses, was in jedem Zustand möglich ist. Von daher machen Endzustände in unserem Modell keinen Sinn.
Obwohl Zustandsdiagramme noch weitaus komplexere Elemente enthalten können als die hier vorgestellten, bestehen die Diagramme, die wir generieren werden, lediglich aus den obigen Elementen. In Kapitel 6 wird dennoch kurz darauf eingegangen werden, wie sich auch Diagramme mit höheren Darstellungselementen (beispielsweise hierarchischen Zuständen) erzeugen lassen könnten.

[12] Eine Ausnahme bildet die Methode `finalize`, die jedoch implizit von der Laufzeitumgebung aufgerufen wird, und das zu einem völlig undefinierten Zeitpunkt (es wird nicht einmal garantiert, dass sie überhaupt aufgerufen wird).

2.3 Die Beschreibung von Klassen durch Zustandsdiagramme

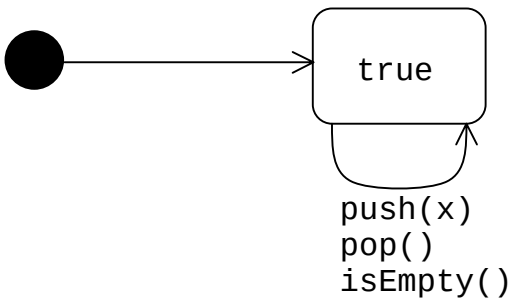
Es soll nun dargelegt werden, wie Zustandsdiagramme von Java-Klassen prinzipiell aufgebaut sind. Dabei wird die Identifikation der Zustände auf Kapitel 4 verschoben.

2.3.1 Ein einfaches Beispiel

Als Beispiel sehen wir uns die Schnittstelle einer Klasse an, die einen Stack implementiert:

```
public class Stack
{

     public Stack();

     public void push(Object x);

     public Object pop();

     public boolean isEmpty();

}
```

Das zugehörige Zustandsdiagramm sieht folgendermaßen aus:



Vor dem Aufruf des Konstruktors befindet sich ein Objekt der Klasse `Stack` im Anfangszustand. Die Transition zum Zustand `true` entspricht dem Aufruf des Konstruktors. Diese Transition bleibt unbeschriftet, weil klar ist, dass nur der Konstruktor sie auslöst (dabei ist es unerheblich, wie viele Konstruktoren die Klasse besitzt). Das Objekt bleibt nun unabänderlich im Zustand `true`, wo die drei Methoden `push`, `pop` und `isEmpty` immer aufgerufen werden dürfen.

2.3.2 Zustände in Klassen

Häufig findet man die Definition, der Zustand eines Objekts sei beschrieben durch die momentanen Attributwerte des Objekts. Diese Definition bezieht sich auf eine sehr technische Ebene. Wir werden allerdings die Zustände auf einer etwas höheren Abstraktionsebene ansiedeln, und zwar aus folgenden Gründen:

- Nimmt man jede erdenkliche Kombination von Attributwerten als eigenen Zustand, entstehen viel zu viele Zustände. Viele davon unterscheiden sich konzeptionell kaum voneinander.
- Auch Attribute aus anderen Objekten, die aus dem eigenen heraus referenziert werden, können für den Objektzustand relevant sein.
- Attribute haben in der Regel privaten Zugriff, weshalb sich von außerhalb der Klasse der Zustand nicht überprüfen lässt.

Wir beschränken uns also nicht auf Attributwerte, sondern sehen auch die Ergebniswerte von bestimmten Funktionen als zustandsbeschreibend an. Im obigen Beispiel kann die Funktion `isEmpty` so betrachtet werden, dass sie Zustände beschreibt. Je nachdem, ob diese Funktion `true` oder `false` liefert, befindet sich das Objekt im Zustand `isEmpty()` bzw. `!isEmpty()`.[13] Letzten Endes wird natürlich auch hier der Zustand durch Attributwerte charakterisiert, da `isEmpty` in irgendeiner Weise auf ein Attribut zugreift; aber wir schaffen es so, die Zustände mithilfe der Schnittstelle zu beschreiben, nicht mithilfe der Implementierung.
Wie man die zustandsbeschreibenden Funktionen und Attribute findet, wird im Verlauf dieser Arbeit noch ausführlich erläutert werden.

2.3.3 Funktionen und Zustandsänderungen

In Zusammenhang mit dem Objektzustand unterscheidet man oft zwischen Funktionen und Methoden ohne Rückgabewert. Manche Autoren vertreten die Meinung, dass Funktionen (also Methoden mit Rückgabewert) den Objektzustand nicht verändern dürfen.[14] Solche Funktionen stünden in Zustandsdiagrammen nur an Selbsttransitionen (Schleifen) und die einzige Begründung dafür, sie überhaupt einzuzeichnen, wäre klarzustellen, in welchen Zuständen sie aufgerufen werden dürften.
Trotzdem kommt es selbst bei einigen Klassen aus den Java-Standardpaketen vor, dass zustandsändernde Methoden einen Rückgabewert liefern, der eine Rückmeldung über die Operation gibt. Beispielsweise gibt die Methode `add` des Interface `java.util.Collection` einen Boolean-Wert zurück, der `true` ist, wenn die Sammlung durch den Aufruf verändert wurde. Ein weiteres Beispiel ist die Methode `pop` aus Abschnitt 2.3.1, die ebenfalls sowohl den Objektzustand verändert (nicht im dort gezeichneten Diagramm, aber konzeptionell) als auch einen Wert zurückgibt. Auch wenn dies nicht der empfehlenswerte Weg ist, so müssen wir dennoch damit rechnen, dass auch Funktionen den Zustand verändern können.

[13] vgl. dazu das Beispiel aus Abschnitt 2.2.2
[14] siehe z. B. [Züllighoven 98]

3. Zusicherungen

Die Charakterisierung von Zuständen und Zustandsübergängen erfolgt mithilfe von Zusicherungen. Dieses Kapitel gibt eine allgemeine Einführung in das Konzept der Zusicherungen (das so genannte Vertragsmodell) und erläutert außerdem, was in Zusammenhang mit Zustandsdiagrammen relevant ist. Zum Abschluss wird noch gezeigt, wie man die Zusicherungen um eventuell vorhandene Klasseninvarianten ergänzen kann.
Teilweise wird bereits darauf eingegangen, wie die hier besprochenen Konzepte mit dem Zustandsmodell zusammenhängen. Das erfolgt an dieser Stelle jedoch nur, soweit es zum Verständnis notwendig ist. Eine detaillierte Darstellung der Zusammenhänge zwischen Zusicherungen und Zuständen erfolgt in Kapitel 4.

3.1 Einführung in Zusicherungen

Das Fundament für Zusicherungen ist das Konzept des Programmierens durch Vertrag.[15] Hierbei wird der Aufruf einer Methode als Dienstleistung angesehen, der bestimmte Vereinbarungen (der Vertrag) zugrunde liegen. Kunde und Lieferant sind jeweils Klassen, und zwar ist die aufrufende Klasse der Kunde und die Klasse, welche die Methode zur Verfügung stellt, der Lieferant. Beide Klassen gehen ein Vertragsverhältnis ein und müssen sich an bestimmte Regeln halten, damit der Vertrag zustande kommt. Diese Regeln werden als Zusicherungen bezeichnet und im Wesentlichen in Vor- und Nachbedingungen aufgeteilt. Vorbedingungen sind die Regeln, die der Kunde einzuhalten hat, während der Lieferant garantiert, dass die Nachbedingungen gelten.
Während die Programmiersprache Eiffel Zusicherungen standardmäßig unterstützt, ist das bei Java leider nicht der Fall. Hier muss man sich damit begnügen, die Zusicherungen in Kommentare zu packen, oder das Vertragsmodell in einer eigenen Klasse selbst implementieren, was jedoch das Modell in mancher Hinsicht einschränkt, insbesondere im Hinblick auf Vererbung. Wir betrachten hier nur die erste Lösung, da sich so die Zusicherungen recht einfach aus der Klasse auslesen lassen. Für eine Überprüfung der Zusicherungen zur Laufzeit wäre hingegen die zweite Lösung unumgänglich. In diesem Rahmen ist das jedoch nicht notwendig.

3.1.1 Vorbedingungen

Eine Vorbedingung ist ein boolescher Ausdruck, der aussagt, wann eine Methode aufgerufen werden darf. Nur wenn dieser Ausdruck zum Aufrufzeitpunkt der Methode zu „wahr“ ausgewertet wird, ist der Aufruf erlaubt. Im anderen Fall hat der Aufrufer (der „Kunde“) den Vertrag gebrochen und die Methode (als „Lieferant“) ist an keine Verpflichtung mehr gebunden. Sie darf Beliebiges machen; im Regelfall wird hier aber ein Programmabbruch mit Fehlermeldung stattfinden.
Vorbedingungen werden innerhalb des Dokumentationskommentars[16] der Methode durch das Tag `@require` eingeleitet. Dies ist kein Java-Standard-Tag, wurde aber im JWAM-Rahmenwerk durchgehend verwendet (ebenso wie die später vorgestellten Tags `@ensure` und `@invariant`). Es sind beliebig viele Vorbedingungen erlaubt.

---

[15] Vertragsmodell, siehe [Meyer 92] oder [Meyer 97]
[16] der Kommentar vor der Methodendeklaration, der in /** ... */ eingeschlossen ist und von Javadoc ausgewertet wird um eine Beschreibung der Methode zu erzeugen

Als Beispiel sei die Definition einer Methode `rename` angegeben, welche eine einfache Umbenennung vornimmt. Als Parameter wird der neue Name übergeben.

```
/**
 * @require newName != null
 * @require newName.length() > 0
 */
public void rename (String newName);
```

(de.jwam.handling.thing.ThingImpl)[17]

Diese Methode hat zwei Vorbedingungen:

1. Der Parameter darf nicht `null` sein (d. h. darf nicht die leere Referenz enthalten).
2. Die Länge des neuen Namens muss größer als 0 sein.

Beide Vorbedingungen hätten auch zu einer zusammengefasst werden können, indem sie durch logisches Und verknüpft worden wären.

3.1.2 Nachbedingungen

Eine Nachbedingung ist ein boolescher Ausdruck, der nach der Ausführung einer Methode immer erfüllt ist. Der Aufrufer kann sich darauf verlassen, dass nach der Ausführung der Methode sämtliche Nachbedingungen erfüllt sind. Ist dies nicht der Fall, so hat der Lieferant den Vertrag gebrochen.
Nachbedingungen werden durch das Tag `@ensure` eingeleitet, wobei auch das Schlüsselwort `result` erlaubt ist um etwas über den Rückgabewert auszusagen. Auch hier sind beliebig viele Nachbedingungen erlaubt.
Es folgt das Beispiel einer Methode, die einen Namen zurückliefert.

```
/**
 * @ensure result != null
 * @ensure result.length() > 0
 */
public String name ();
```

(de.jwam.handling.thing.ThingImpl)

Beide Nachbedingungen sagen etwas über den Rückgabewert aus:

1. Der Rückgabewert ist nicht `null`.
2. Die Länge der zurückgegebenen Zeichenkette ist größer als 0.

Auch hier wäre es möglich gewesen, beide Nachbedingungen durch logisches Und zu einer einzigen zusammenzufassen.
Eine weitere Möglichkeit Nachbedingungen zu formulieren bietet die Old-Schreibweise. Während für die Attribute und Methoden, die in Nachbedingungen vorkommen,

[17] Hierbei wie an allen folgenden Stellen handelt es sich um ein Zitat aus einer Klasse des JWAM-Rahmenwerks (Version 1.6.0).

standardmäßig die Werte *nach* Beendigung der Methodenausführung eingesetzt werden, kann auf die entsprechenden Werte *vor* der Methodenausführung zugegriffen werden, indem man das Schlüsselwort `old` vor einen Teilausdruck (meist ein Attribut) setzt. Damit könnte z. B. die Hinzufügeoperation einer Behälterklasse zusichern, dass sich die Größe des Behälters bei Aufruf dieser Operation um genau 1 erhöht:

```
@ensure size = old size + 1
```

So eine Nachbedingung lässt sich in Eiffel leicht formulieren, aber in Java ist eine entsprechende Simulation recht umständlich.[18]

3.1.3 Klasseninvarianten

Eine Klasseninvariante ist ein boolescher Ausdruck, der zu jedem Zeitpunkt wahr ist, zu dem keine Methode der Klasse ausgeführt wird. Der erste Zeitpunkt, zu dem die Klasseninvariante gültig sein muss, ist nach Ausführung des Konstruktors. Weiterhin ist sie vor und nach jedem Methodenaufruf erfüllt, jedoch nicht zwingend während der Ausführung einer Methode. Klasseninvarianten sind wie Nachbedingungen Zusicherungen, die der Lieferant an die aufrufende Klasse macht.
Invarianten werden durch `@invariant` eingeleitet und können im Dokumentationskommentar einer Klasse, eines Interface oder eines Attributs stehen.
Beispielsweise könnte eine Behälterklasse zusichern, dass die Größe zu jedem Zeitpunkt nichtnegativ ist:

```
@invariant size >= 0
```

Es sei noch angemerkt, dass innerhalb von Dokumentationskommentaren streng genommen HTML-Code stehen muss. Eine Bedingung wie:

```
! s.equals("") && s.length < 8
```

müsste also korrekt formuliert werden:

```
! s.equals(&quot;&quot;) &amp;&amp; s.length &lt; 8
```

Wir verwenden wegen der besseren Lesbarkeit dennoch die erste Schreibweise; die gängigen Browser haben auch keine Probleme damit, sie korrekt anzuzeigen.

3.2 Ansätze zur Beschreibung von Zuständen und Zustandsänderungen

In [Meyer 97] werden vier Hauptgründe für das Schreiben von Zusicherungen genannt[19]:

- Unterstützung beim Schreiben korrekter Software
- Dokumentationshilfe
- Unterstützung von Testen, Debugging und Qualitätssicherung
- Unterstützung von Softwarefehler-Toleranz

[18] Dies gilt für den Fall, dass man Code zur Überprüfung der Zusicherungen schreibt.
[19] S. 389

Keiner dieser Gründe steht jedoch in Zusammenhang mit der Dokumentation von Zustandsänderungen. Diese Art der Verwendung erwähnt Meyer nicht.

3.2.1 Objectcharts

Ein Ansatz zur Charakterisierung von Zustandsdiagrammen mithilfe von Zusicherungen findet sich in [CHB 92] unter der Bezeichnung Objectcharts. Dieser Ansatz hat jedoch einen entscheidenden Nachteil: Die Zusicherungen sind nicht an die Methoden gebunden, sondern an die Transitionen. Im Programmtext müssen die Zusicherungen aber an Methoden gebunden sein, weil die Transitionen dort nicht explizit auftauchen.
Zur Verdeutlichung wird hier ein Beispiel aus [CHB 92] angeführt[20]:

$$\text{open} \rightarrow \text{open} : \{true\}\ type(c)\ \{contents = \overleftarrow{contents}\,.\,c \wedge display = contents\}$$

Dabei steht der übergestellte Pfeil für die Old-Schreibweise und der Punkt für die Konkatenation. Die Formel bedeutet, dass für eine Transition von open nach open die Methode `type` mit dem Parameter `c` aufgerufen werden kann. Die Vorbedingung ist `true`, d. h. es gibt keine Einschränkungen; die Nachbedingung sagt aus, dass `c` an `contents` angehängt wird und `display` auf `contents` gesetzt wird.
Die Formel sagt allerdings nicht aus, wann die Methode `type` aufgerufen werden darf und in welchem Zustand das Programm sich danach befindet. Zwar erkennt man, dass es möglich ist, mit `type` vom Zustand open zum Zustand open zu kommen, aber es könnte noch andere Zustandsübergänge durch `type` geben. Viel sinnvoller wäre es, die Zustände selbst in die Zusicherungen der Methode `type` einzubauen, was sich auch unmittelbar im Programmtext niederschreiben ließe. Im zitierten Beispiel ist die Vorbedingung von `type` jedenfalls nicht `true`, wie sich aus dem auf der gleichen Seite des zitierten Artikels abgebildeten Zustandsdiagramm leicht ablesen lässt: Ein Aufruf von `type` ist nur im Zustand open möglich.

3.2.2 Der Ansatz von Harel und Gery

Einen anderen Ansatz vertreten Harel und Gery in [HG 96]. Sie benutzen das volle Spektrum an Möglichkeiten bei Zustandsdiagrammen und beziehen sich nicht auf Zusicherungen. Ohne näher auf ihren Ansatz einzugehen sei bemerkt, dass ihr Ansatz aus zwei Gründen für unsere Zwecke ebenfalls unbrauchbar ist:

1. Die Wahl der Zustände ist willkürlich.
2. Die Kommunikation zwischen den Objekten wird zu stark betont.

Der erste Punkt besagt, dass der Programmierer im Wesentlichen selbst entscheidet, welche Zustände es gibt. Was die Zustände dabei charakterisiert, ist nicht formalisierbar und verhindert daher eine automatische Generierung von Zustandsdiagrammen.
Nicht minder gewichtig ist der zweite Punkt. Er widerspricht dem Lokalitätsprinzip. Die Autoren gehen davon aus, dass eine Klasse für einen ganz bestimmten Verwendungszweck geschrieben wird. Die Objekte des Systems kommunizieren miteinander und diese Kommunikation schlägt sich in den Zustandsdiagrammen der Objekte nieder. Bei dieser

[20] S. 14

Methode macht es keinen Sinn, das Zustandsdiagramm einer einzelnen Klasse für sich allein zu betrachten. Gerade das ist aber das Ziel dieser Arbeit.

3.3 Unterteilung der Zusicherungen

Im Folgenden soll der in dieser Arbeit verwendete Ansatz vorgestellt werden.
Der Aufruf einer Methode kann zu einem Zustandswechsel innerhalb des Zustandsdiagramms führen. In jedem Fall findet eine Transition von einem Ursprungs- zu einem Zielzustand statt (welche auch identisch sein können).
Die Zusicherungen können nun dazu verwendet werden, diesen Zustandsübergang zu dokumentieren. Vereinfacht gesagt beschreiben die Vorbedingungen die möglichen Ursprungszustände und die Nachbedingungen die möglichen Zielzustände.
Allerdings beschreiben nicht alle Zusicherungen einen Zustand. Sie können auch etwas über die Parameter der Methode oder den Rückgabewert aussagen. Eine Unterteilung der Zusicherungen in verschiedene Typen ist daher an dieser Stelle sinnvoll. Diese Typen werden im Folgenden näher beschrieben.

3.3.1 Parameter-Zusicherungen

Parameter-Zusicherungen sagen etwas über einen Parameter der Methode aus. Sehr häufig trifft man eine Vorbedingung an, die aussagt, dass ein Parameter nicht die leere Referenz enthalten darf:

```
@require newName != null
```

Ein weiterer Fall ist der, dass eine Methode des übergebenen Objekts nur bestimmte Werte liefern darf:

```
@require newName.length() > 0
```

Ferner gibt es den Fall, dass eine bestimmte Methode, aufgerufen mit dem Parameter, gewisse Werte liefern muss:

```
@require isValid(s)
```

(de.jwam.lang.domainvalue.DomainValueImpl)

Bei Werteparametern ist auch eine Einschränkung des Wertebereichs möglich.
Parameter-Zusicherungen treten fast ausschließlich bei Vorbedingungen auf; in seltenen Fällen kann es sie aber auch in Form von Nachbedingungen geben:

```
/**
 * @ensure creator().equals(user)
 */
public void setCreation (dvUserIdentificator user);
```

(de.jwamx.handling.registry.RegisterableImpl)

Diese Arten von Zusicherungen können durchaus so interpretiert werden, dass hier ein Zustand beschrieben wird. Solche Zustände sind jedoch von komplexer Natur, da nicht von vornherein ersichtlich ist, welche Belegungen für die Parameter überhaupt möglich sind. Im Prinzip müsste für jede mögliche Belegung ein eigener Zustand eingeführt werden, was nicht mehr handhabbar wäre. Weiterhin flößen so die Zustände anderer Objekte in das Zustandsdiagramm mit hinein. Von daher ist es sinnvoll, Parameter-Zusicherungen generell nicht zum Bestimmen von Zuständen einzusetzen.

3.3.2 Rückgabewert-Zusicherungen

Rückgabewert-Zusicherungen sind Nachbedingungen, die etwas über den Rückgabewert einer Funktion aussagen. Sie sind daran leicht zu erkennen, dass sie das Schlüsselwort `result` enthalten.
In Abschnitt 3.1.2 finden sich Beispiele für Rückgabewert-Zusicherungen.
Auch diese Zusicherungen eignen sich nicht dazu, Zustände zu beschreiben. Abgesehen von der entstehenden Komplexität[21] macht es in den allermeisten Fällen auch gar keinen Sinn, da Rückgabewert-Zusicherungen selten etwas über das eigene Objekt aussagen. Deshalb sollten auch Rückgabewert-Zusicherungen nicht zum Bestimmen von Zuständen verwendet werden.
Es sei noch bemerkt, dass es Zusicherungen gibt, die sowohl Rückgabewert- als auch Parameter-Zusicherungen sind:

```
/**
 * @ensure result.equals(value)
 * @ensure result.getClass() == value.getClass()
 */
protected DomainValue registeredValue (DomainValue value);
```

(de.jwam.lang.domainvalue.DomainValueImpl)

3.3.3 Zustands-Zusicherungen

Alle anderen Zusicherungen sind Zustands-Zusicherungen. Hier wird also weder auf die Parameter noch auf den Rückgabewert der Methode Bezug genommen.
Folgende Möglichkeiten gibt es:

- Es wird auf ein oder mehrere Attribute Bezug genommen.

  Beispiel:

  ```
  /**
   * @require _form != null
   */
  protected void createSubFpsForNewMaterial();
  ```

  (de.jwambeta.handling.formtools.fpFormEditor)

[21] siehe Abschnitt 3.3.1

Diese Art von Zusicherung ist recht selten und auch nicht empfehlenswert, da Attribute in der Regel nicht von außen sichtbar sind und somit auch der Zustand nicht von außerhalb überprüfbar ist. Stattdessen sollte man die Klasse um eine Methode erweitern, die das gewünschte Attribut abfragt, und die Zusicherung mithilfe dieser Methode formulieren.

- Es wird auf eine oder mehrere Methoden Bezug genommen.

  Beispiel:

  ```
  /**
   * @require hasLock()
   * @ensure !isLocked()
   */
  public void unlock (KeyRing ring);
  ```

  (de.jwam.handling.accesscontrol.LockableThingImpl)

  Dies ist der bei weitem häufigste Fall von Zustands-Zusicherungen. In den meisten Fällen handelt es sich bei den angegebenen Methoden wie im obigen Beispiel um parameterlose Prädikate. Weniger häufig trifft man auf die Abfrage, ob eine bestimmte Methode den Wert `null` zurückliefert. Methoden, die Integer- oder Character-Werte zurückgeben, sind zwar prinzipiell auch möglich, treten aber in der Praxis äußerst selten in Zustands-Zusicherungen auf.

Die in der Zusicherung abgefragte Methode muss nicht unbedingt in der eigenen Klasse liegen, sondern kann sich auch in einem Objekt befinden, das aus dem eigenen heraus referenziert wird:

```
/**
 * @ensure floorPlan().entryCount() > 0
 */
public BuildingImpl();
```

(de.jwamalpha.handling.corporateroom.BuildingImpl)

Da es sich bei Zusicherungen um boolesche Ausdrücke handelt, können diese durch logische Operatoren verknüpft werden. Hierbei findet die Und-Verknüpfung die bei weitem häufigste Anwendung. Dies schlägt sich auch darin nieder, dass die Zusicherungen oft aufgeteilt werden, was einer Und-Verknüpfung entspricht. Eine Oder-Verknüpfung hingegen kommt sehr selten vor.[22]

3.3.4 Old-Zusicherungen

Wenn wir Nachbedingungen mit dem Schlüsselwort `old` erlauben – die wie schon erwähnt nicht leicht überprüfbar sind – können wir diese als Old-Zusicherungen einstufen. Wir werden jedoch im weiteren Verlauf dieser Arbeit darauf verzichten, unter anderem deshalb, weil Old-Zusicherungen im JWAM-Rahmenwerk so gut wie nicht vorkommen. Ihre einzige Anwendung finden sie in den Account-Klassen aus den Paketen unter

[22] vgl. Abschnitt 5.3.1

de.jwamexample.cookbook, wo sie den korrekten Saldo nach dem Ein- und Auszahlen zusichern:

```
/**
 * @ensure balance() = old balance() + amount
 */
public void deposit (float amount);
```

(de.jwamexample.cookbook.step01_material.Account)

Bei diesen Zusicherungen handelt es sich jedoch ebenso um Parameter-Nachbedingungen, sodass es nicht nötig ist, einen neuen Typ von Zusicherungen einzuführen. Old-Zusicherungen, die keinem der drei anderen Typen von Zusicherungen entsprechen, kommen in JWAM nicht vor. Wenn man sie dennoch verwenden würde, würden sie als Zustands-Zusicherungen eingestuft, was zwar korrekt wäre, aber zu falschen Ergebnissen führen würde. Von daher lassen wir Old-Zusicherungen, die keine Parameter- oder Rückgabewert-Zusicherungen sind, nicht zu.[23]

3.4 Die Beschreibung von Zuständen durch Zusicherungen

Die Zustands-Zusicherungen sind gemäß Abschnitt 3.3 diejenigen, die tatsächlich einen Zustandsübergang beschreiben. Aber wie lassen sich die Zustände auf einfache Weise charakterisieren?
Dazu führen wir zunächst den Begriff des Observers ein.

3.4.1 Observer

Ein **Observer** ist ein in einer Zustands-Zusicherung vorkommendes Attribut, eine Methode oder ein Ausdruck mit genau einem relationalen Operator.

Insbesondere beinhaltet ein Observer keine booleschen Verknüpfungen, auch nicht die Negation.
Diese Definition ist an den Observer-Begriff aus [CHB 92] angelehnt, wobei dort unter einem Observer eine sondierende Operation verstanden wird, was ein Spezialfall dieser Definition ist, und zwar der wichtigste.
Wichtig beim Observer sind seine verschiedenen Belegungen. Die konkreten Belegungen aller Observer ist gerade das, was einen Zustand ausmacht.
Nehmen wir ein paar Beispiele um zu verdeutlichen, was ein Observer ist:

```
@require hasLock() && ! isLocked()
```

Hier gibt es zwei Observer: `hasLock()` mit der Belegung `true` und `isLocked()` mit der Belegung `false`.
Dieser Fall, also parameterlose Prädikate als Observer, ist der bei weitem häufigste.
Das nächste Beispiel ist nicht so eindeutig:

```
@require _form != null
```

[23] Die Old-Schreibweise macht in Rückgabewert-Zusicherungen allerdings wenig Sinn, da Letztere nur bei Funktionen auftreten und dort keine Zustandsänderung erfolgen sollte.

In diesem Beispiel kann das Attribut `_form` als Observer angesehen werden. Seine konkrete Belegung ist `!=null`. Es ist aber auch möglich, den gesamten Ausdruck `_form!=null` als Observer mit der Belegung `true` zu interpretieren.
Weitere Fälle für Observer erhält man dadurch, dass man das obige Attribut durch einen Methodenaufruf ersetzt oder das Ungleichheitssymbol mit einem anderen relationalen Operator vertauscht. Die Mehrdeutigkeiten bleiben hierdurch erhalten.

3.4.2 Observer, die in Nachbedingungen nicht vorkommen

Der Zustandsübergang einer Methode wird nun folgendermaßen beschrieben:

- Die Vorbedingungen geben an, in welchen Zuständen die Methode aufgerufen werden darf.
- Die Nachbedingungen geben an, in welchen Zuständen das Programm sich nach Ausführung der Methode befinden kann.

Der erste Punkt ist noch unproblematisch. Der Programmierer weiß, welche Bedingungen erfüllt sein müssen, damit die Methode läuft. Diese Bedingungen sind eben die möglichen Ursprungszustände für Transitionen.
Mit den Nachbedingungen verhält es sich aber anders. Nehmen wir das folgende Beispiel:

```
/**
 * @require hasLock()
 * @ensure isLocked()
 */
public void lock (KeyRing ring);
```

(de.jwam.handling.accesscontrol.LockableThingImpl)

Während die Vorbedingung klar die Zustände wiedergibt, in denen die Methode aufrufbar ist, sagt die Nachbedingung nur etwas über den Observer `isLocked()` aus. Ob im Zielzustand nun `hasLock()` die Belegung `true` oder `false` hat, ist nicht spezifiziert, und somit müssten beide Belegungen als Alternativen für Zielzustände angesehen werden.
Aber es kommt noch schlimmer: Was ist mit Observern, die in den Zusicherungen dieser Methode überhaupt nicht erwähnt sind, aber bei anderen Methoden derselben Klasse auftreten? Wenn man keine näheren Angaben hat, müsste man davon ausgehen, dass solche Observer ihre Belegung ändern könnten, und dementsprechend würde sich die Anzahl der Zielzustände weiter vermehren. Man stelle sich eine Methode ohne Nachbedingung vor (d. h. die Nachbedingung ist `true`). In diesem Fall wären sämtliche Zustände Zielzustände der Methode, weil ja keine Angaben zur Belegung der Observer nach Ausführung der Methode gemacht werden.
Man kann sich leicht vorstellen, dass die Zustandsdiagramme, die aus solchen Annahmen entstehen, ziemlich schnell sehr unübersichtlich werden. Bereits bei wenigen Observern entstehen derartig viele Transitionen, dass das Zustandsdiagramm praktisch nicht mehr zu gebrauchen ist.
Glücklicherweise gibt es einen Ausweg aus diesem Dilemma. In den meisten Fällen ändert nämlich eine Methode gar nicht die Observer, die in den Nachbedingungen nicht angegeben sind. Im obigen Beispiel hieße das, dass `hasLock()` einfach `true` bleibt. Wenn ein Observer sich ändert, so ist meist seine Belegung nach jedem Aufruf der Methode dieselbe,

und die wird dann in der Nachbedingung angegeben. Der Fall, dass ein Observer bei verschiedenen Aufrufen derselben Methode unterschiedliche Belegungen annehmen kann, ist eher selten.
Dieser Fall kommt aber vor, wie man an folgendem Beispiel sehen kann, das aus der Implementierung eines Stacks stammen könnte:

```
/**
 * @require ! isEmpty()
 */
public void pop ();
```

Die Methode `pop`, die das oberste Element vom Stack entfernt, hat die Vorbedingung, dass der Stack nicht leer sein darf. Ob er allerdings hinterher leer oder nicht leer ist, kann unterschiedlich sein.
Hier stehen wir also vor einem Problem: Wir können nicht einfach annehmen, dass Observer, die in der Nachbedingung nicht angegeben sind, ihre Belegungen behalten. Andererseits werden unter diesen Umständen die Zustandsdiagramme nicht mehr überschaubar.
Der Grund, warum [Meyer 97] dieses Problem nicht sieht, ist klar: Er hat einfach eine ganz andere Sicht auf Zusicherungen. Für ihn sind Zusicherungen Bestandteil eines Vertrages zwischen Kunde und Lieferant, nicht ein Mittel um Zustände zu beschreiben. Auch ist sein Blickwinkel ein anderer: Nach seinem Verständnis kann die Methode selbst entscheiden, was sie zusichert und was nicht. In unserem Kontext entwickeln wir jedoch ein Zustandsmodell der ganzen Klasse; somit tritt die Methode in den Dienst der Klasse und hat zu beschreiben, wie sie sich darin wiederfindet.
In der ersten Auflage von [Meyer 97] findet sich dennoch ein Konstrukt, das unser Problem teilweise löst: der Ausdruck *Nochange*. Das ist ein boolescher Ausdruck, der „genau dann wahr liefert, wenn kein Attribut des aktuellen Objekts seit dem Aufruf den Wert geändert hat“.[24] *Nochange* ist nur in Nachbedingungen erlaubt und soll nützlich sein, wenn man aussagen möchte, dass eine Methode unter bestimmten Umständen keinen Effekt hat. Anscheinend ist *Nochange* aber doch nicht so nützlich, denn in der zweiten Auflage wird es nicht mehr erwähnt.
Für unsere Zwecke ist *Nochange* aber zu wenig. Es reicht nicht zu spezifizieren, dass sich gar nichts ändert, sondern man müsste angeben können, dass sich ganz bestimmte Observer nicht ändern. Eine mögliche Lösung für unser Problem bestünde also darin, ein zusätzliches Tag `@nochange` einzuführen, das angibt, welche Observer von der Methode nicht geändert werden (durch Kommata getrennt oder in verschiedenen Anweisungen). Im ersten Beispiel dieses Abschnitts würde man also noch hinzufügen:

```
@nochange hasLock()
```

Hier ist allerdings Vorsicht geboten: Sollte es in der Klasse noch andere Observer geben, müssen diese ebenfalls in einem `@nochange`-Ausdruck angegeben werden. Da später noch neue Methoden und somit neue Observer zur Klasse hinzugefügt werden können, liegt hier eine gefährliche potenzielle Fehlerquelle!
Eine andere Lösung unseres Problems wäre genau den umgekehrten Fall explizit anzugeben: Wir führen ein Tag `@canchange` ein, das angibt, welche Observer von der Methode geändert werden können, wobei nicht von vornherein klar ist, welche Belegungen sie am Ende annehmen. Das hieße, dass beim obigen Stack-Beispiel noch zusätzlich anzugeben wäre:

[24] 1. Auflage von [Meyer 97] (deutsche Übersetzung) S. 124

```
@canchange isEmpty()
```

Aus folgenden Gründen ist diese Lösung der Nochange-Lösung vorzuziehen:

1. Der Canchange-Fall tritt wesentlich seltener auf als der Nochange-Fall. Es ist daher eine sehr große Erleichterung für den Programmierer, den Nochange-Fall als Standard anzusehen, den er nicht noch explizit hinschreiben muss.
2. Da zusätzliche Observer, an die der Programmierer nicht gedacht hat, standardmäßig als unverändert angesehen werden, besteht hier kein Risiko, dass der Programmierer etwas übersieht.

In dieser Arbeit wird also davon ausgegangen, dass Observer, die von einer Methode geändert werden können, mittels eines `@canchange`-Tags angegeben werden (durch Kommata getrennt oder in mehreren Tags). Wenn natürlich klar ist, welche Belegungen sie nach Ausführung der Methode haben, sollen die Belegungen stattdessen in einem `@ensure`-Tag niedergeschrieben werden.
Statt des `@canchange`-Tags hätten wir auch folgende Nachbedingung angeben können:

```
@ensure isEmpty() || ! isEmpty()
```

Diese Bedingung wäre nicht, wie man auf den ersten Blick vermuten könnte, identisch mit `true`, weil in `true` überhaupt keine Observer erwähnt werden und somit keine Zustandsänderung stattfindet. Diese Nachbedingung sagt lediglich aus, dass die Belegung von `isEmpty()` nicht feststeht, was äquivalent zu `@canchange isEmpty()` ist. Wir verwenden dennoch die Formulierung mithilfe des `@canchange`-Tags, weil sie klarer ist und sich leichter programmtechnisch verarbeiten lässt.

3.4.3 Eine Anmerkung zu Funktionen

Ob man davon ausgeht, dass Funktionen generell den Objektzustand nicht ändern, oder nicht, spielt in diesem Modell keine wesentliche Rolle. Funktionen können Zustands-Vorbedingungen haben, wenn sie nicht in allen Zuständen aufgerufen werden dürfen. Wenn es keine Zustands-Nachbedingungen gibt, wird wie oben beschrieben davon ausgegangen, dass der Zustand sich nicht ändert. Sieht man das als Normalfall für Funktionen an, so wird genau richtig gehandelt; und wenn man dennoch zustandsändernde Funktionen schreiben möchte, so kann man einfach entsprechende Nachbedingungen angeben. In diesem Modell ist also keine explizite Unterscheidung zwischen Funktionen und Methoden ohne Rückgabewert notwendig.

3.5 Klasseninvarianten im Zusammenspiel mit Vor- und Nachbedingungen

Zum Abschluss soll noch erläutert werden, wie Vor- und Nachbedingungen um Klasseninvarianten ergänzt werden können.
Dazu gehen wir zur Vereinfachung davon aus, dass es nur eine einzige Klasseninvariante gibt. Dies können wir ohne Beschränkung der Allgemeinheit annehmen, denn falls es mehrere Invarianten gibt, kann man sie einfach und-verknüpfen, ohne dass sich die Semantik ändert.
Die Klasseninvariante ist vom Prinzip her genauso aufgebaut wie eine Zustands-Zusicherung. Auch die Invariante beschreibt die Belegungen von Observern und schränkt diese Belegungen

für alle Zustände des Objekts ein. In allen einnehmbaren Zuständen müssen die Observer-Belegungen der Invariante genügen. Daher muss es erlaubt sein, die Invariante zu sämtlichen Zusicherungen hinzuzufügen (also durch „und“ zu verknüpfen).
Nehmen wir z. B. an, die Klasseninvariante sei folgende:

```
@invariant ! isHandled() || ! isCancelled()
```

(de.jwam.handling.toolconstruction.Request)[25]

Weiterhin sei eine Methode `cancel` folgendermaßen deklariert:

```
/**
 * @ensure isCancelled()
 */
public void cancel();
```

(de.jwam.handling.toolconstruction.Request)[26]

Wegen der Invariante könnte man als Vorbedingung hinzufügen:

```
@require ! isHandled() || ! isCancelled()
```

Das ist natürlich insofern unnötig, als dass ohnehin kein Zustand eingenommen werden kann, der nicht dieser Bedingung entspricht.
Bei der Nachbedingung sieht das allerdings schon anders aus. Die Invariante führt zu folgender erweiterter Nachbedingung:

```
@ensure isCancelled() && ( ! isHandled() || ! isCancelled() )
```

Dieser Ausdruck lässt sich vereinfachen zu:

```
@ensure isCancelled() && ! isHandled()
```

Hier haben wir in der Tat eine Einschränkung, die wir ohne Berücksichtigung der Invariante nicht gefunden hätten: Im Zustand, den das Objekt nach Ausführung der Methode einnimmt, gilt neben `isCancelled()` auch noch `!isHandled()`.
Allgemein lässt sich also sagen, dass die Invariante zu den Vorbedingungen hinzugefügt werden kann (wenn auch nicht muss, da die nicht der Invariante entsprechenden Zustände ohnehin nicht erreichbar sind), während sie zu den Nachbedingungen generell hinzugefügt werden muss. In vielen Fällen wird man allerdings die Nachbedingungen schon so formulieren, dass die Invariante dort auftritt.
Im weiteren Verlauf dieser Arbeit wird zur Vereinfachung davon ausgegangen, dass Invarianten stets in den Nachbedingungen hingeschrieben werden. Dass das nicht, wie man zunächst vermuten könnte, zu einem erheblichen Mehraufwand führt, kann man sich dadurch

[25] Die Invariante ist dort nicht explizit angegeben.
[26] Die Nachbedingung ist hier für unsere Zwecke angepasst worden. Zusätzlich wird noch `!isHandled()` zugesichert, was wir aber anhand der Invariante herleiten.

klar machen, dass der Programmierer sich eher die vollständige Nachbedingung überlegt als die Invariante implizit anzunehmen. Auch im obigen Beispiel wurde in der Originalklasse die Invariante nicht angegeben; stattdessen wurde die vollständige Nachbedingung `isCancelled() && !isHandled()`formuliert.

4. Von den Zusicherungen zum Zustandsdiagramm

In diesem Kapitel wird ausführlich erläutert, wie man von der Java-Klasse zum Zustandsdiagramm kommt. Aus den Zusicherungen werden zunächst die Observer bestimmt, woraufhin ein mathematisches Modell der Zusicherungen sowie der Zustände entwickelt wird. Dieses Modell liefert sowohl die Zustände als auch die Transitionen.
Der komplette Vorgang wird anhand eines Beispiels nachvollzogen. Dazu verwenden wir wiederum die Schnittstelle einer Stack-Klasse, die zwar verhältnismäßig klein ist, aber dennoch alle Typen von Zusicherungen enthält. Um das zu erzeugende Diagramm zu vereinfachen kommen jedoch nur Prädikate als Observer vor; auf die Behandlung von andersartigen Observern wird im Text näher eingegangen.
Die Schnittstelle unseres Beispiels mit den zugehörigen Dokumentationskommentaren sieht folgendermaßen aus:

```
public class Stack
{

	/**
	 * @ensure isEmpty()
	 * @ensure ! isFull()
	 */
	public Stack();

	/**
	 * @require x != null
	 * @require isEmpty() || ! isFull()
	 * @ensure ! isEmpty() && ! isFull()
	 */
	public void push(Object x);

	/**
	 * @require ! isEmpty()
	 * @ensure ! isFull()
	 * @canchange isEmpty()
	 */
	public void pop();

	/**
	 * @require ! isEmpty()
	 * @ensure result != null
	 */
	public Object top();

	public boolean isEmpty();

	public boolean isFull();

}
```

4.1. Erkennen der Zustands-Zusicherungen

Ein Programm zur automatischen Generierung eines Zustandsdiagramms müsste zuerst zu jeder Methode die Zusicherungen extrahieren, und zwar getrennt nach `require`, `ensure` und `canchange`.
Da im Folgenden nur die Zustands-Zusicherungen benötigt werden, müssen die Parameter- und Rückgabewert-Zusicherungen entfernt werden. Parameter-Zusicherungen erkennt man daran, dass ein Parameter der entsprechenden Methode darin vorkommt. In unserem Beispiel ist die Vorbedingung `x!=null` bei `push` eine Parameter-Vorbedingung. Rückgabewert-Zusicherungen erkennt man ebenso leicht am Schlüsselwort `result`. Die Nachbedingung `result!=null` bei `top` ist eine Rückgabewert-Nachbedingung.
Bleibt nun bei einer Methode keine Vor- bzw. Nachbedingung mehr übrig, so ist `true` als entsprechende Zusicherung zu setzen. Dies gilt für die Nachbedingung von `top` sowie für Vor- und Nachbedingungen von `isEmpty` und `isFull`. Der Konstruktor hingegen hat keine Zustands-Vorbedingung.
Falls eine Methode jetzt noch mehrere Vor- bzw. Nachbedingungen hat, fassen wir diese zu einer einzigen zusammen, indem wir sie und-verknüpfen. Damit haben wir es geschafft, dass jede Methode genau eine Zustands-Vorbedingung (bis auf den Konstruktor) und eine Zustands-Nachbedingung hat. Im weiteren Verlauf werden wir nur noch von der Vor- und Nachbedingung einer Methode sprechen, wenn wir die Zustands-Zusicherungen meinen, da die übrigen Zusicherungen nicht mehr verwendet werden.
Die Methoden des Beispiels besitzen also folgende Vor- und Nachbedingungen:

Methode	**Vorbedingung**	**Nachbedingung**
(Konstruktor)		`isEmpty() && ! isFull()`
`push`	`isEmpty() || ! isFull()`	`! isEmpty() && ! isFull()`
`pop`	`! isEmpty()`	`! isFull()`
`top`	`! isEmpty()`	`true`
`isEmpty`	`true`	`true`
`isFull`	`true`	`true`

4.2 Bestimmung der Observer

4.2.1 Identifikation der Observer

Laut Abschnitt 3.4.1 ist ein Observer ein in einer Zustands-Zusicherung vorkommendes Attribut, eine Methode oder ein Ausdruck mit genau einem relationalen Operator. Somit könnten wir jeden Ausdruck zwischen den booleschen Operatoren der Zusicherungen als Observer ansehen. In unserem Fall wären das `isEmpty()` und `isFull()`. Wenn wir generell Ausdrücke mit einem relationalen Operator (z. B. `size>0`) als Observer betrachten, so erhalten wir ausschließlich boolesche Observer. Sehen wir aber in solchen Ausdrücken die Methoden bzw. Attribute auf der „linken“ Seite der Relation als Observer an, so bekommen wir auch anderweitig getypte Observer, deren Belegung jeweils auf der „rechten“ Seite angegeben ist. Das ermöglicht komplexere Belegungen. Gibt es beispielsweise die Zusicherungen `size>0` und `size<10`, so kann man das Attribut `size` als Observer betrachten. Die relevanten Belegungen wären:

```
1. size <= 0
2. 0 < size < 10
3. size >= 10
```

Betrachtet man dagegen die beiden Ausdrücke als zwei (boolesche) Observer, so sind die Belegungen:

```
1. size > 0
2. ! (size > 0)
```

und:

```
1. size < 10
2. ! (size < 10)
```

Bei dieser Interpretation geht allerdings der Zusammenhang zwischen den beiden Ausdrücken verloren. Es ist nicht mehr erkennbar, dass `!(size>0)` und `!(size<10)` gleichzeitig unmöglich ist. Bei der Interpretation mit `size` als Observer ist dieser Fall von vornherein ausgeschlossen, da der Wertebereich korrekt aufgeteilt wurde.
Beide Interpretationen sind prinzipiell möglich. Man sollte jedoch die einmal gewählte Strategie beibehalten und sich nicht von Fall zu Fall umentscheiden.
Nachdem wir die Observer identifiziert haben, nummerieren wir sie und bezeichnen sie als o_1, o_2 usw. bis o_n. n sei die Anzahl der Observer.

Die Anzahl der Observer wird mit n bezeichnet. Die Observer seien $o_1, o_2, \ldots, o_n$. $num(o)$ liefere die Nummer des Observers o. Es gilt: $num(o_i) = i$ für $i \in \{1, \ldots, n\}$

In unserem Beispiel setzen wir o_1 für `isEmpty()` und o_2 für `isFull()`. Diese Zuordnung ist willkürlich.

4.2.2 Relevante Werteteilmengen der Observer

Wir müssen jetzt herausfinden, welche Belegungen der Observer relevant sind, d. h. wir fassen Belegungen zusammen, die identische Zustände beschreiben. Jeder Observer ist ein programmiersprachlicher Ausdruck, dessen Werte auf einen bestimmten Wertebereich eingeschränkt sind. Wir führen eine Mengenfunktion ein, die den Wertebereich eines Observers liefert:

Für einen Observer o sei $w(o)$ die Menge der Werte, die o annehmen kann.

Da die Wertebereiche aller Ausdrücke beschränkt sind – auch die von Fließkomma-Ausdrücken – ist $w(o)$ in jedem Fall endlich. Wichtiger ist jedoch die Eigenschaft, dass sich $w(o)$ in endlich viele Mengen partitionieren lässt.
Im einfachsten Fall, wenn o ein boolescher Observer ist, gilt $w(o) = \{$`true`, `false`$\}$. Im Beispiel trifft das bei beiden Observern zu. Hat o einen numerischer Typ, so ist $w(o)$ eine Menge von Zahlen, und zwar eine Teilmenge der natürlichen Zahlen, wenn o einen ganzzahligen Typ hat, oder eine Teilmenge der rationalen Zahlen, wenn o einen Fließkomma-Typ hat.

Wenn *o* einen Referenztyp hat, sprechen wir im programmiersprachlichen Gebrauch nicht von Werten, sondern von Referenzen. Wir werden hier dennoch die verschiedenen Referenzen, die *o* annehmen kann, als seine Werte bezeichnen, da eine Unterscheidung in diesem Kontext nicht notwendig ist. Dies sind ebenfalls nur endlich viele Werte, da es nur endlich viele Objekte im System geben kann.
Wir teilen nun die Wertebereiche der Observer in die relevanten Teilmengen auf. Dabei werden die Belegungen, die bei allen Vor- und Nachbedingungen zu denselben Ergebnissen führen, zusammengefasst. In Abschnitt 4.2.1 findet sich bereits ein Beispiel, wie man dem Observer `size` drei relevante Werteteilmengen zuweisen kann. Streng genommen müsste man noch die Einschränkung des gesamten Wertebereichs von `size` berücksichtigen, da beliebig große und beliebig kleine Werte nicht angenommen werden können, aber diese Tatsache muss nicht explizit angegeben werden; es genügt, wenn man sie im Hinterkopf behält.
Bei Observern mit Referenztypen verfährt man folgendermaßen: Alle Belegungen, die in einer Zusicherung explizit angegeben sind, erhalten eine eigene Werteteilmenge; am Schluss definiert man eine weitere Teilmenge, die sämtliche anderen Belegungen enthält (die Restmenge bezüglich des Wertebereichs).

Die relevanten Werteteilmengen eines Observers *o* seien $w_1(o)$, $w_2(o)$, ... , $w_k(o)$.
Die Menge aller relevanten Werteteilmengen eines Observers *o* sei $W(o)$.

Es gilt: $W(o) = \{w_1(o), w_2(o), \dots , w_k(o)\}$

Weiterhin: $\forall i \in \{1, \dots, n\} : w(o_i) = \bigtimes_{k=1}^{|W(o_i)|} w_k(o_i)$

Und: $\forall i \in \{1, \dots, n\} : \forall j,k \in \{1, \dots, |W(o_i)|\} : w_j(o_i) \cap w_k(o_i) \neq \emptyset \Rightarrow j = k$

Die letzten beiden Formeln sagen aus, dass der gesamte Wertebereich die Vereinigung aller Werteteilmengen ist und dass die Werteteilmengen eines Observers zueinander disjunkt sind.
In unserem Beispiel besitzen beide Observer die relevanten Werteteilmengen {`true`} und {`false`}. Damit erhalten wir insgesamt folgende Zuordnungen:

o_1	`isEmpty()`
o_2	`isFull()`
$w(o_1)$	{`true`, `false`}
$w(o_2)$	{`true`, `false`}
$w_1(o_1)$	{`true`}
$w_2(o_1)$	{`false`}
$w_1(o_2)$	{`true`}
$w_2(o_2)$	{`false`}

Die Zuordnung von {`true`} jeweils zu w_1 ist wiederum willkürlich; ebenso gut hätten wir {`false`} w_1 zuordnen können.

4.3 Modellierung der Zusicherungen

Ähnlich den Observern werden wir die Zusicherungen mathematisch modellieren. Dies wird nötig sein um später die Zustände und Transitionen präzise anzugeben. Wir werden zunächst die Vor- und Nachbedingungen mithilfe der relevanten Werteteilmengen beschreiben und

anschließend die Mengen *canchange* und *nochange* bestimmen, die ausdrücken, wie diejenigen Observer sich verhalten, die in der Nachbedingung einer Methode nicht auftreten.

4.3.1 *pre*(*m*) und *post*(*m*)

Wir modellieren die Vor- und Nachbedingung einer Methode als Menge von *n*-Tupeln (*n* ist die Anzahl der Observer). Jedes Element dieser Menge gibt dabei eine Belegung der Observer an, die dieser Bedingung genügt. Das *i*-te Element eines Tupels ist eine Vereinigung von relevanten Werteteilmengen des *i*-ten Observers, die die Belegung dieses Observers anzeigt. Beispielsweise steht das Tupel $(w_2(o_1), w(o_2), w_1(o_3) \cup w_3(o_3))$ für eine Belegung, in der o_1 einen Wert aus seiner 2. Teilmenge hat, o_2 einen beliebigen Wert (aus seiner gesamten Wertemenge) und o_3 einen Wert aus seiner 1. oder 3. Teilmenge.

Zu einer Methode *m* beschreiben *pre*(*m*) und *post*(*m*) die Vor- bzw. Nachbedingung.
pre(*m*) existiere nur, wenn *m* kein Konstruktor ist.
Es gelte:

$$pre(m) \subseteq \{(v_1,...,v_n) \mid \forall i \in \{1,...,n\}: (v_i \neq \emptyset \wedge v_i \subseteq w(o_i) \wedge \forall j \in \{1,...,n\}: v_i \supseteq w_j \vee v_i \cap w_j = \emptyset)\} \supseteq post(m)$$

Falls $n \geq 1$, gelte zusätzlich:

$$| pre(m) | \geq 1 \leq | post(m) |$$

Die erste Formel besagt, dass *pre*(*m*) und *post*(*m*) Mengen von *n*-Tupeln sind, wobei die Mengen an *i*-ter Position des Tupels Teilmengen des Wertebereichs des *i*-ten Observers und gleichzeitig entweder Obermengen der Werteteilmengen oder disjunkt zu diesen sind. Damit garantieren wir, dass die Werteteilmengen nur vollständig oder gar nicht dort auftreten, jedoch keine Teilmengen von diesen.
Wir müssen weiterhin garantieren, dass keine leeren Mengen innerhalb des Tupels auftreten; dann wäre die entsprechende Bedingung nicht erfüllbar. Ebenso wäre sie nicht erfüllbar, wenn *pre*(*m*) oder *post*(*m*) leer wären. Dies verbietet die zweite Formel. Wenn es allerdings keine Observer gibt, können wir $pre(m) = post(m) = \emptyset$ setzen.
Um eine Vor- oder Nachbedingung in diese Form zu bringen ist es zunächst erforderlich, sie in die disjunktive Normalform umzuwandeln, d. h. in eine Disjunktion von Mintermen. Die Anzahl der Minterme entspricht der Mächtigkeit von *pre*(*m*) bzw. *post*(*m*). Jeder Minterm ist eine Konjunktion von Observer-Belegungen, die mithilfe der relevanten Werteteilmengen ausgedrückt werden können und sich somit als Tupel niederschreiben lassen.
In unserem Beispiel liegen die Zusicherungen bereits in disjunktiver Normalform vor.[27] Die Modellierung der Zusicherungen sieht wie folgt aus:

Methode *m*	***pre*(*m*)**	***post*(*m*)**
(Konstruktor)		$\{ (w_1(o_1), w_2(o_2)) \}$
`push`	$\{ (w_1(o_1), w(o_2)), (w(o_1), w_2(o_2)) \}$	$\{ (w_2(o_1), w_2(o_2)) \}$
`pop`	$\{ (w_2(o_1), w(o_2)) \}$	$\{ (w(o_1), w_2(o_2)) \}$
`top`	$\{ (w_2(o_1), w(o_2)) \}$	$\{ (w(o_1), w(o_2)) \}$
`isEmpty`	$\{ (w(o_1), w(o_2)) \}$	$\{ (w(o_1), w(o_2)) \}$
`isFull`	$\{ (w(o_1), w(o_2)) \}$	$\{ (w(o_1), w(o_2)) \}$

[27] In der Praxis kommt es äußerst selten vor, dass dies nicht der Fall ist.

4.3.2 *canchange*(*m*) und *nochange*(*m*)

Nachdem die Vor- und Nachbedingung einer Methode mathematisch modelliert worden sind, muss noch ausgedrückt werden, was mit den Observern geschieht, die in der Nachbedingung nicht vorkommen. Für derartige Observer gibt es genau zwei Möglichkeiten: Entweder ihr Wert kann von der Methode beliebig geändert werden (Canchange-Fall) oder ihr Wert bleibt erhalten (Nochange-Fall).
Die Menge *canchange*(*m*) enthält die Observer, die in der Nachbedingung nicht vorkommen und deren Wert sich ändern kann:

Für eine Methode *m* sei *canchange*(*m*) die Menge der Observer, deren Wert nach Ausführung der Methode nicht festgelegt ist.
Es gelte: $\forall o \in canchange(m) : (v_1,...,v_n) \in post(m) \Rightarrow v_{num(o)} = w_{num(o)}$

Die Formel besagt, dass alle Observer aus *canchange*(*m*) mit ihrem gesamten Wertebereich in der Nachbedingung auftreten müssen, d. h. der Wertebereich darf nicht eingeschränkt sein.
Die Elemente von *canchange*(*m*) zu finden ist einfach, da sie im `canchange`-Tag explizit angegeben werden. Man braucht nur die Observer, die in diesem Tag genannt werden, in die Menge aufzunehmen.
Die Menge *nochange*(*m*) enthält die in der Nachbedingung nicht vorkommenden Observer, deren Werte erhalten bleiben:

Für eine Methode *m* sei *nochange*(*m*) folgendermaßen definiert:
$nochange(m) = \{o \mid o \notin canchange(m) \wedge \forall (v_1,...,v_n) \in post(m) : v_{num(o)} = w_{num(o)}\}$

Man beachte, dass *nochange*(*m*) nicht notwendigerweise alle Observer beinhalten muss, deren Wert unverändert bleibt: Es kann Observer geben, die in der Vor- und Nachbedingung mit derselben Belegung vorkommen. Es geht hier ausschließlich um die Observer, die in der Nachbedingung nicht angegeben sind, und für diese gibt es genau zwei Möglichkeiten: Entweder sie sind in *canchange*(*m*) enthalten oder in *nochange*(*m*).
Die Bestimmung der Elemente von *nochange*(*m*) kann dadurch geschehen, dass man die Observer heraussucht, die weder in der Nachbedingung noch in *canchange*(*m*) vorkommen.
Für unser Beispiel ergibt sich Folgendes:

Methode *m*	***canchange*(*m*)**	***nochange*(*m*)**
(Konstruktor)	$\varnothing$	$\varnothing$
`push`	$\varnothing$	$\varnothing$
`pop`	$\{o_1\}$	$\varnothing$
`top`	$\varnothing$	$\{o_1, o_2\}$
`isEmpty`	$\varnothing$	$\{o_1, o_2\}$
`isFull`	$\varnothing$	$\{o_1, o_2\}$

4.4 Modellierung der Zustände und Transitionen

In diesem Abschnitt werden wir die Elemente des Zustandsdiagramms modellieren, also im Wesentlichen die Zustände und Transitionen. Dabei werden wir das Zustandsdiagramm unseres Beispiels Schritt für Schritt aufbauen.

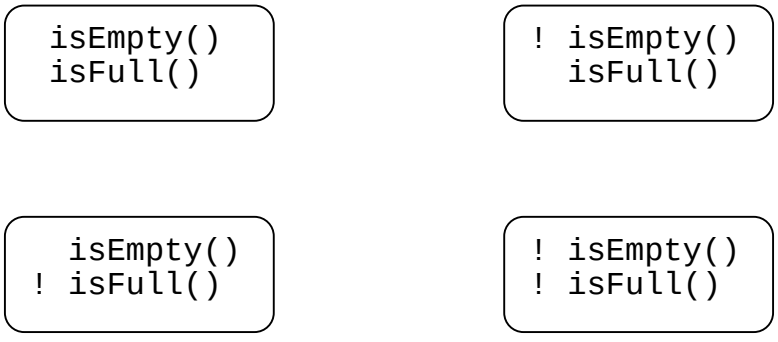
4.4.1 Die Zustände

Die aktuelle Belegung der Observer beschreibt den Objektzustand. Entscheidend ist beim einzelnen Observer, in welcher relevanten Werteteilmenge sein aktueller Wert liegt, da alle Werte innerhalb einer Werteteilmenge denselben Zustand beschreiben. Dies führt zu folgender Definition:

Die Menge der Zustände ist: $Z = \bigtimes_{i=1}^{n} W(o_i)$, falls $n \geq 1$. Ansonsten ist $Z = \{\texttt{true}\}$.

Damit ist die Anzahl der Zustände: $|Z| = \prod_{i=1}^{n} |W(o_i)|$

In Worten ausgedrückt ist die Zustandsmenge das Kreuzprodukt der relevanten Werteteilmengen aller Observer. Wenn es keine Observer gibt, nennen wir den einzigen Zustand `true`. – Der Anfangszustand ist in dieser Definition nicht enthalten; er existiere unabhängig davon in jedem Fall.
Die Existenz dieser Zustände garantiert nicht, dass alle Zustände auch eingenommen werden können. Es handelt sich lediglich um die prinzipiell möglichen Zustände des Objekts. Wir werden später sehen, wie man nicht erreichbare Zustände ausschließen kann.
Für unser Beispiel ergibt sich die Zustandsmenge Z = { ({`true`}, {`true`}), ({`false`}, {`true`}), ({`true`}, {`false`}), ({`false`}, {`false`}) }. Wir können also beginnen das Zustandsdiagramm zu entwerfen, indem wir die Zustände zeichnen. Der besseren Lesbarkeit wegen schreiben wir die Namen der Observer in die Zustände und geben jeweils ihre konkrete Belegung an.



4.4.2 Die Transitionen

Eine Transition vom Zustand z_1 zum Zustand z_2 bei Aufruf der Methode *m* existiert, wenn in z_1 die Vorbedingung und in z_2 die Nachbedingung von *m* gilt. Dabei hängt z_2 nicht von z_1 ab, da wir in der Nachbedingung keinen Bezug auf die Vorbedingung erlaubt haben. Hätten wir das zugelassen (beispielsweise durch die Old-Schreibweise), so müssten wir bei der Bestimmung der Zielzustände einer Methode die jeweiligen Ursprungszustände berücksichtigen.
Zur mathematischen Modellierung benötigen wir noch die Tupel-Projektion:

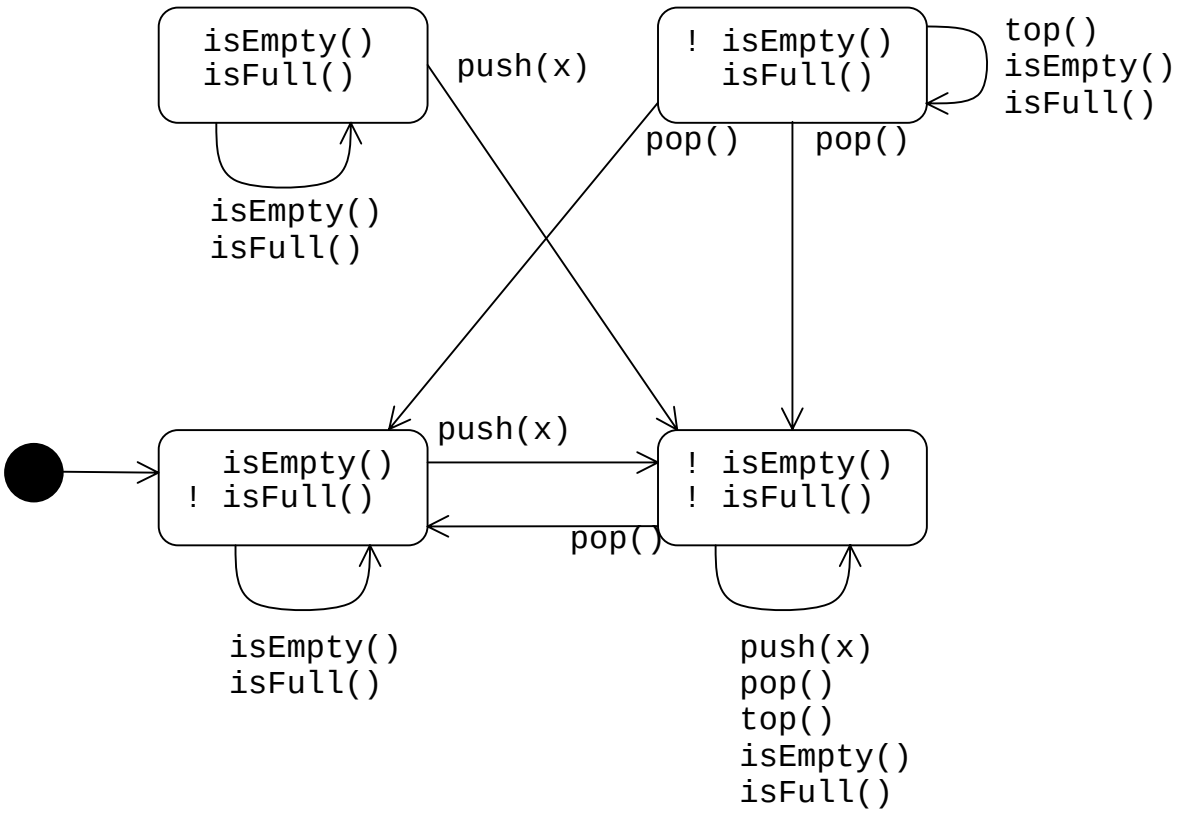
Zu einem Tupel $t = (t_1, \ldots, t_i, \ldots, t_n)$ beschreibe $\pi_i(t)$ die *i*-te Projektion, d. h. $\pi_i(t) = t_i$.

Nun können wir die Transitionen mathematisch formulieren:

Eine durch die Methode *m* ausgelöste Transition vom Zustand z_1 zum Zustand z_2 existiert genau dann, wenn folgender Ausdruck wahr ist:
$\forall i \in \{1,...,n\}: (\exists p \in pre(m): \pi_i(z_1) \subseteq \pi_i(p) \wedge \exists p \in post(m): \pi_i(z_2) \subseteq \pi_i(p) \wedge (o_i \in nochange(m) \Rightarrow \pi_i(z_1) = \pi_i(z_2)))$

Eine Transition vom Anfangszustand zum Zustand *z* existiert genau dann, wenn für mindestens einen Konstruktor *k* folgender Ausdruck wahr ist:
$\forall i \in \{1,...,n\}: \exists p \in post(k): \pi_i(z) \subseteq \pi_i(p)$

In den Formeln wird jeweils geprüft, ob die Belegungen der Observer den in den Zusicherungen angegebenen Belegungen entsprechen. Außerdem ist bei einem Observer, der in *nochange* vorkommt, zu beachten, dass seine Belegung beim Zustandswechsel erhalten bleiben muss.
Jetzt können wir die Transitionen für unser Beispiel ermitteln. Auf die mathematische Ausformulierung wird hier verzichtet; sie kann bei Bedarf vom Leser selbst nachvollzogen werden. Stattdessen wird das resultierende Zustandsdiagramm gezeigt:



Damit ist die mathematische Modellierung des Zustandsdiagramms abgeschlossen. Das generierte Diagramm lässt sich aber noch in mancher Hinsicht verbessern. An obigem Beispiel sieht man, dass nicht alle Zustände erreichbar sein müssen. Im folgenden Abschnitt wird daher darauf eingegangen, wie sich das Diagramm systematisch verbessern lässt.

4.5 Verbesserungen am generierten Zustandsdiagramm

4.5.1 Entfernen unerreichbarer Zustände und Finden neuer Invarianten

Das Auffinden und Entfernen von unerreichbaren Zuständen ist ein graphentheoretisches Problem.[28] Man kann das Zustandsdiagramm als Graphen ansehen, wobei die Zustände die Knoten und die Transitionen gerichtete Kanten repräsentieren. Dann lässt sich folgender Standardalgorithmus anwenden um die unerreichbaren Zustände zu entfernen:

1. Setze Z auf die Menge der Zustände, die Zielzustände eines Anfangszustands sind.
2. Markiere alle Zustände aus Z.
3. Setze $Z_{neu} \leftarrow \emptyset$.
4. Füge alle Zustände, die Zielzustände eines Zustands aus Z und nicht markiert sind, zu Z_{neu} hinzu.
5. Setze $Z \leftarrow Z_{neu}$.
6. Falls $Z \neq \emptyset$, gehe zu Schritt 2.
7. Alle Zustände, die nicht markiert wurden, sind nicht erreichbar und können entfernt werden. Ebenso können alle Transitionen entfernt werden, deren Ursprungszustand nicht markiert wurde.

Unerreichbare Zustände hängen eng mit Invarianten zusammen. Ein Zustand, der niemals eingenommen wird, entspricht ja einer Bedingung, die nie wahr ist. Die Negation dieser Bedingung ist somit eine Invariante. Wir können also neue Invarianten formulieren, indem wir die Observer-Belegungen in einem unerreichbaren Zustand und-verknüpfen und die so entstandene Bedingung negieren. Diese Invarianten lassen sich zu einer einzigen zusammenfassen, indem sie alle und-verknüpft werden.
Im generierten Zustandsdiagramm aus dem Beispiel sind – wie schon direkt ersichtlich – die oberen beiden Zustände nicht erreichbar. Das bedeutet, dass wir als Invarianten `!(isEmpty()&&isFull())` und `!(!isEmpty()&&isFull())` formulieren können. Eine einzige Invariante erhalten wir, indem wir diese beiden und-verknüpfen. Das ergibt:

```
!(isEmpty() && isFull()) && !(!isEmpty() && isFull())
```

Der Ausdruck kann vereinfacht werden zu:

```
! isFull()
```

Wir haben somit als Invariante erhalten, dass `isFull()` immer `false` liefert.

4.5.2 Parameter-Vorbedingungen als Guard-Bedingungen

Bisher haben wir Parameter-Zusicherungen nicht für die Generierung des Zustandsdiagramms verwendet. Zumindest die Parameter-Vorbedingungen haben dennoch ein Pendant im Zustandsdiagramm: Da sie Bedingungen repräsentieren, die beim Aufruf der zugehörigen Methode gelten müssen, sind sie nichts anderes als Guard-Bedingungen der entsprechenden Transition. Wir können sie also wie folgt in das Diagramm einbeziehen:

[28] siehe z. B. [Turau 96]

- Alle Parameter-Vorbedingungen einer Methode werden und-verknüpft.
- Die entstandene Bedingung wird als Guard-Bedingung an alle Transitionen angefügt, die von dieser Methode ausgelöst werden.

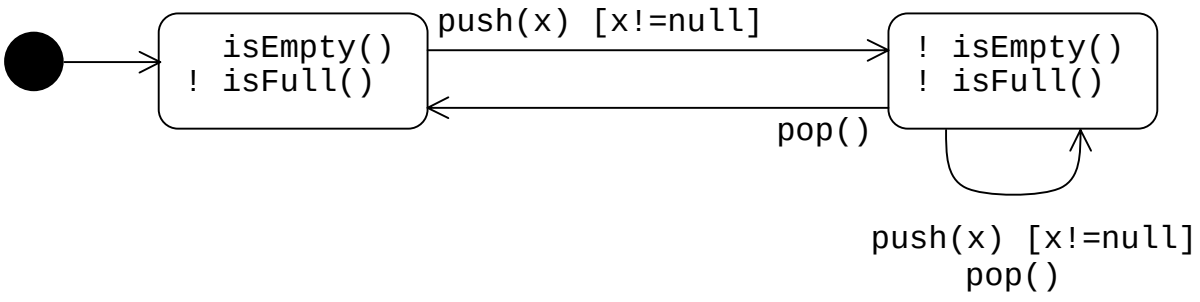
Im Beispiel können wir `[x!=null]` an alle Transitionen von `push(x)` schreiben.

4.5.3 Verzicht auf die Einbeziehung von Funktionen

Bei Klassen mit vielen Methoden kann das Diagramm unübersichtlich werden. Nun wurde bereits in Abschnitt 2.3.3 angemerkt, dass Funktionen im Allgemeinen als nicht zustandsändernd angesehen werden. Behält man dies konsequent bei, so stehen Funktionen ausschließlich an Selbsttransitionen (gleicher Ursprungs- und Zielzustand) und die einzige Information, die man bekäme, wäre, in welchen Zuständen ein Aufruf der Funktion erlaubt ist. Liegt das Hauptaugenmerk aber auf den zustandsändernden Methoden, dann stören die vielen Funktionen nur. Zeichnet man keine Funktionen in das Diagramm, so kann es sehr an Übersicht gewinnen.
Dabei muss jedoch bedacht werden, dass zustandsändernde Funktionen in der Praxis durchaus erlaubt sind. Es ist daher zu empfehlen, dass ein Programm zur automatischen Generierung von Zustandsdiagrammen zumindest eine Option besitzt, welche die Einbeziehung von Funktionen ein- oder ausschaltet.

Wir wenden uns jetzt ein letztes Mal unserem Beispiel zu und berücksichtigen dabei auch die Verbesserungen aus den letzten Abschnitten. Die Klasse enthält keine zustandsändernden Funktionen, was daran zu erkennen ist, dass die Funktionen keine Zustands-Nachbedingungen besitzen. Die Funktionen können daher im Diagramm weggelassen werden; dadurch geht allerdings die Information verloren, dass `top` nur in Zuständen aufgerufen werden darf, in denen `!isEmpty()` gilt.

Das generierte Zustandsdiagramm sieht letztendlich so aus:



5. Praktische Anwendung

Nachdem wir in den vorigen Kapiteln ein Verfahren kennen gelernt haben, mit dem man Zustandsdiagramme aus Java-Klassen generieren kann, widmen wir uns in diesem Kapitel der praktischen Anwendung dieses Verfahrens. Zunächst wird das Programm behandelt, das im Rahmen dieser Studienarbeit entstand. Anschließend werden die Klassen des JWAM-Rahmenwerks im Hinblick auf ihre Zustandsdiagramme näher untersucht. Außerdem wird diskutiert, inwieweit die mit unserem Verfahren generierten Zustandsdiagramme in der Praxis von Nutzen sind.

5.1 Das Programm class2statechart

Im Rahmen dieser Studienarbeit wurde das Programm class2statechart entwickelt, das den Quelltext einer Java-Klasse einliest und das zugehörige Zustandsdiagramm generiert und anzeigt. Die Java-Klasse kann entweder in der Kommandozeile übergeben werden oder in einem File-Dialog ausgewählt werden. Das Programm wird im Folgenden näher beschrieben.

5.1.1 Die Optionen

Beim Aufruf des Programms lassen sich verschiedene Optionen angeben, die Einfluss auf den Generierungsprozess sowie teilweise auf das dargestellte Zustandsdiagramm haben. Die Optionen beziehen sich größtenteils auf Sonderbehandlungen. Wird keine Option angegeben, so wird das Standardverfahren angewendet, welches im Allgemeinen die brauchbarsten Ergebnisse liefert.
Die Optionen werden durch ein Minuszeichen eingeleitet. Mehrere Optionen können entweder nach einem einzigen Minuszeichen oder jede Option nach einem eigenen Minuszeichen stehen. Die Optionen `c` und `f` können also durch „`-cf`“ oder durch „`-c -f`“ gesetzt werden.
Folgende Optionen sind möglich:

- `c` („canchange“): Bei einem Observer, der in der Nachbedingung einer Methode nicht vorkommt, wird standardmäßig der Nochange-Fall angenommen. Es wird also davon ausgegangen, dass sein Wert nach Ausführung der Methode erhalten bleibt (sofern der Observer nicht in einem `@canchange`-Tag vorkommt). Will man dagegen den Canchange-Fall als Standard setzen, so kann man dies mit der Option `c` tun. In diesem Fall wird bei einem Observer, der in der Nachbedingung einer Methode nicht vorkommt, davon ausgegangen, dass er anschließend alle möglichen Werte angenommen haben kann. Dies gilt allerdings nur für Methoden ohne Rückgabewert, nicht für Funktionen. Bei Funktionen wird weiterhin der Nochange-Fall angenommen (sofern nicht die Option `p` gesetzt wird). – Das Einschalten dieser Option führt jedoch in der Regel dazu, dass das Zustandsdiagramm viele Transitionen enthält und dadurch unübersichtlich wird.
- `f` („show functions“): Standardmäßig werden im Zustandsdiagramm keine Funktionen angezeigt.[29] Viele Zustandsdiagramme werden dadurch wesentlich übersichtlicher. Bei eingeschalteter Option `f` werden auch die Funktionen im Diagramm angezeigt.
- `p` („functions like procedures“): Da in vielen Kreisen die Meinung herrscht, dass Funktionen den Objektzustand nicht ändern sollen, geht das Programm standardmäßig von dieser Annahme aus. Selbst wenn die Option `c` aktiviert ist, wird bei Funktionen immer

[29] vgl. Abschnitt 4.5.3

noch vom Nochange-Fall ausgegangen. Sollen aber Funktionen wie alle anderen Methoden behandelt werden, so ist die Option `p` anzugeben. Diese Option hat nur bei gleichzeitig eingeschaltetem `c` und `f` eine Wirkung. Die gleichzeitige Aktivierung dieser drei Optionen führt zu den größten Zustandsdiagrammen, die sich mit dem Programm erzeugen lassen. Sie sind im Regelfall weder lesbar noch sinnvoll.

- `t` („time information“): Diese Option führt dazu, dass auf der Standardausgabe angezeigt wird, wie viel Zeit für das Parsen der Datei und für die Generierung des Zustandsdiagramms benötigt wurde. Im Normalfall liegen beide Werte weit unter einer Sekunde.
- `v` („verbose information“): Bei dieser Option wird die Anzahl der Methoden, der Observer, der Zustände und der Transitionen angezeigt.

Beim Diagramm selbst gibt es außerdem eine Checkbox „show methods of self-transitions“, mit der man ein- und ausschalten kann, ob Selbsttransitionen dargestellt werden sollen. Manchmal gibt es sehr viele Selbsttransitionen, die Teile des Diagramms verdecken.
Um die Größenordnungen zu demonstrieren, die das Einschalten dieser Optionen bewirken kann, wird hier die Anzahl der generierten Elemente für die JWAM-Klasse de.jwam.handling.toolconstruction.request.Request bei unterschiedlichen Optionen angegeben. Die Klasse hat einen Konstruktor und 6 weitere Methoden (davon 3 Funktionen).

Optionen	**Zustände (ohne Anfangszustand)**	**Transitionen**
(keine)	3	7
`-c`	5	21
`-f`	6	32
`-cf`	10	112
`-cfp`	16	770

5.1.2 Einschränkungen

Das Programm beherrscht nicht alle Möglichkeiten, die das besprochene Verfahren bietet. Um es zu vereinfachen besitzt das Programm einige Einschränkungen, die im Folgenden erläutert werden. In Abschnitt 5.3 werden wir sehen, dass diese Einschränkungen keine großen Konsequenzen mit sich ziehen. Für die meisten Klassen generiert das Programm genau die Zustandsdiagramme, die auch das vollständige Verfahren liefert.
Die Einschränkungen des Programms sind im Einzelnen:

- **Es werden ausschließlich boolesche Observer unterstützt.** Dabei muss es sich nicht zwangsläufig um parameterlose Prädikate handeln, sondern es sind auch Ausdrücke wie beispielsweise `sender!=null` erlaubt. Entscheidend ist nur, dass die Ausdrücke Boolean-Werte liefern. Durch diese Einschränkung entfällt das Bestimmen der relevanten Werteteilmengen, denn diese sind in jedem Fall {`true`} und {`false`}. Die beiden Belegungen der Observer erhält man dadurch, dass man einmal den Observer direkt hinschreibt und einmal ein Ausrufezeichen davor. Im genannten Beispiel wäre die zweite Belegung `!sender!=null` (was selbstverständlich etwas anderes ist als `sender==null`, da dies ein anderer Observer wäre). Letztendlich gibt es durch diese Einschränkung keine Ausdrücke, die nicht verarbeitet werden, denn jede Zusicherung besteht aus booleschen Ausdrücken.
- **Nur und-verknüpfte Zusicherungen werden unterstützt.** Ausdrücke, in denen Oder- bzw. andere Verknüpfungen vorkommen, werden ignoriert. Somit bleibt es dem

Programm erspart, die Ausdrücke zunächst in die disjunktive Normalform umzuwandeln, denn diese liegt bereits vor. Die Ausdrücke sind also so aufgebaut, dass zwischen den Und-Symbolen die Observer stehen, wobei die jeweilige Belegung an einem eventuell vorhandenen vorangestellten Ausrufezeichen abzulesen ist.

- **Eine Methodendeklaration muss in einer einzigen Zeile stehen.** In den meisten Fällen trifft das zu, aber bei Methoden mit vielen Parametern kann es durchaus vorkommen, dass die Methodendeklaration über zwei Zeilen geht. Dies wird vom Programm derzeit noch nicht unterstützt.

5.1.3 Das Vorgehen

In diesem Abschnitt wird die Vorgehensweise des Programms kurz beschrieben. Dabei werden lediglich die einzelnen Schritte des Generierungsprozesses aufgezeigt; eine detaillierte Beschreibung des Programms erfolgt hier nicht. Es soll nur dargelegt werden, wie das Programm im Prinzip vorgeht.
Grob gesagt lässt sich der Programmablauf in folgende Schritte unterteilen:

1. **Die Datei wird geparst.** Dabei werden die einzelnen Methoden herausgesucht. Zu jeder Methode wird der Dokumentationskommentar durchsucht und die `@require`-, `@ensure`- und `@canchange`-Tags gespeichert. Hier werden auch gleich die Zustands-Zusicherungen herausgefiltert. Parameter-Vorbedingungen werden der Methode[30] als Guard-Bedingungen hinzugefügt. Die Zustands-Zusicherungen werden ebenfalls den Methoden übergeben, wobei gleichzeitig die Observer bestimmt und gesammelt werden. Am Ende dieses Schrittes stehen also die Methoden mitsamt ihren Zusicherungen sowie die Observer fest.
2. **Die Zustände und Transitionen werden bestimmt.** Dazu werden der Reihe nach alle Methoden der eingelesenen Klasse durchgegangen. Nun werden anhand der Vor- und Nachbedingung und der Canchange-Observer die Transitionen generiert, die zur jeweiligen Methode gehören. Alle Zustände, die der Vorbedingung genügen, sind Ursprungszustände von Transitionen. Bei Zuständen, die der Nachbedingung genügen, müssen die Observer betrachtet werden, die in der Nachbedingung nicht vorkommen. Kommt ein derartiger Observer in einem `@canchange`-Tag vor, so ist jeder passende Zustand Zielzustand einer Transition; kommt er dort nicht vor, so ist er nur bei denjenigen Transitionen Zielzustand, in deren Ursprungszustand seine Belegung dieselbe ist wie im Zielzustand. – Die zugehörigen Zustände werden bei Bedarf erzeugt.
3. **Die nicht erreichbaren Zustände werden entfernt.** Das geschieht mithilfe des in Abschnitt 4.5.1 vorgestellten Algorithmus. Die nicht erreichbaren Transitionen werden dabei gleich mit entfernt, da in einem Zustand auf die ausgehenden Transitionen verwiesen wird.
4. **Das Zustandsdiagramm wird angezeigt.** Vom Modell des Zustandsdiagramms, das bisher im Speicher aufgebaut wurde, wird nun ein darstellbares Modell erzeugt, d. h. für die einzelnen Zustände und Transitionen werden Bildschirmkoordinaten berechnet. Das so generierte Diagramm wird dargestellt.

[30] Gemeint ist die Methode der eingelesenen Klasse. Diese Methode wird im Programm durch ein eigenes Objekt repräsentiert.

5.2 Folgen fehlerhafter Zusicherungen

Die Zusicherungen werden in Kommentaren angegeben. Kommentare werden vom Compiler ignoriert. An keiner Stelle findet eine Überprüfung der Zusicherungen statt, nicht einmal auf syntaktische Korrektheit. Das führt zwangsläufig dazu, dass es bei der Eingabe von Zusicherungen zu Fehlern kommt, und fehlerhafte Zusicherungen ziehen mit sich, dass das besprochene Verfahren falsche Zustandsdiagramme generiert. In diesem Abschnitt wird untersucht, wie schwerwiegend unterschiedliche Arten von Fehlern sind und was sich dadurch am Zustandsdiagramm ändert. Dazu werden die möglichen Fehler in verschiedene Kategorien unterteilt und jeweils Beispiele aus dem JWAM-Rahmenwerk angeführt. Außerdem werden Möglichkeiten zur Fehlerbehebung aufgezeigt.

5.2.1 Schreibfehler bei Observern

Der erste mögliche Fehler ist, dass man sich bei einem Observer verschreibt. Dies führt natürlich nur dann zu einem falschen Diagramm, wenn der Observer mal falsch und mal richtig geschrieben ist.
Ein Beispiel aus JWAM ist:

```
/**
 * @require hasLock()
 * @require !isLocked()
 * @ensure !hasLock
 * @ensure hasRemovedLock()
 */
public void removeLock ();
```

(de.jwam.handling.accesscontrol.LockableThingImpl)

Hier wurden bei der ersten Nachbedingung die Klammern hinter `!hasLock` vergessen. Dieser Umstand führt im Generierungsprozess dazu, dass ein weiterer Observer eingeführt wird. Es gibt nun also die Observer `hasLock` und `hasLock()`, obwohl nur ein einziger Observer gemeint war.
Im Diagramm ergibt sich durch den zusätzlichen Observer eine Verdoppelung der Zustände und ein recht undurchsichtiges Verhalten, und zwar nicht nur bei der Methode `removeLock`. Am Diagramm ist allerdings sofort ersichtlich, dass ein derartiger Fehler aufgetreten ist, da in allen Zuständen die beiden fast identischen Observer stehen. Man braucht lediglich den falsch geschriebenen Observer in seinem Programm zu finden um den Fehler beheben zu können.
Prinzipiell wäre es jedoch möglich, dass das Programm diesen Fehler aufdeckt. Durch Vergleich der Bedingungen mit den Attributen und Methoden der Klasse könnte festgestellt werden, dass `hasLock` kein Attribut der Klasse ist und somit ein Fehler vorliegt. Zu dieser Art von Prüfung müssten allerdings sämtliche Oberklassen einbezogen werden, sodass der Generierungsprozess umständlicher und länger wäre.

5.2.2 Schreibfehler bei Parameter-Zusicherungen

Parameter-Zusicherungen werden nicht zum Bestimmen von Zuständen verwendet.[31] Sie werden dadurch erkannt, dass in ihnen ein Parameter der zugehörigen Methode auftritt. Ist

[31] vgl. Abschnitt 3.3.1

dieser Parameter in der Zusicherung jedoch falsch geschrieben, so wird die Zusicherung nicht als Parameter-Zusicherung eingestuft, sondern als Zustands-Zusicherung.
Hier ein Beispiel:

```
/**
 * @require newAccountNr != null
 */
public void changeAccountNumber(dvAccountNumber newAccountNo);
```

(de.jwamexample.cookbook.step04_domainvalue.Account)

Der Parameter ist in der Vorbedingung falsch geschrieben. Sie wird deshalb als Zustands-Vorbedingung interpretiert und es wird ein Observer `newAccountNr!=null` eingeführt.
Wie im vorigen Abschnitt ist auch dies sofort im Diagramm ersichtlich und der Fehler muss nur noch lokalisiert werden. Auch ein programmseitiges Auffinden des Fehlers wäre möglich, da `newAccountNr` kein Attribut ist.

5.2.3 Schreibfehler bei Rückgabewert-Zusicherungen

Auch bei der Formulierung von Rückgabewert-Zusicherungen kann es zu Fehlern kommen. Diese Zusicherungen charakterisiert ja das Schlüsselwort `result`. Wird dieses falsch geschrieben, so wird die Zusicherung als Zustands-Zusicherung eingestuft.
Auch hierzu ein Beispiel:

```
/**
 * @require hasRecipient()
 * @ensure return != null
 */
public dvTransportAddress recipient ();
```

(de.jwamalpha.handling.processfolder.material.ProcessFolderAdapter)

Hier wurde fälschlich `return` statt `result` geschrieben. Die Folge ist auch hier, dass ein neuer Observer `return!=null` eingeführt wird, was aber ebenso leicht zu beheben ist wie in den vorigen Abschnitten (sowohl vom Programmierer als auch vom Programm).

5.2.4 Schreibfehler beim Namen eines Tags

Es kann sogar vorkommen, dass der Name eines Tags falsch geschrieben ist. Im JWAM-Rahmenwerk findet sich folgendes Beispiel:

```
/**
 * @rquire frameContext != null
 */
public void equipWithFrame(FrameContext frameContext);
```

(de.jwam.handling.toolconstruction.basicconstruction.ToolImpl)

Dieser Fehler führt dazu, dass das Tag nicht erkannt wird und damit unberücksichtigt bleibt. Im obigen Beispiel hat das zwar keine nennenswerten Folgen, da es sich um eine Parameter-Zusicherung handelt, die ohnehin nur als Guard-Bedingung eingestuft worden wäre, aber

wenn es sich um eine Zustands-Zusicherung gehandelt hätte, wäre sie nicht in den Generierungsprozess einbezogen worden und die Methode hätte andere Transitionen.
Diese Art von Fehlern ist nicht so einfach zu entdecken. Erst ein Vergleich des Zustandsdiagramms mit dem intendierten Verhalten liefert einen Hinweis darauf, dass bei den Zusicherungen ein Fehler stecken muss.

5.2.5 Vergessene Zusicherungen

Schließlich kann es vorkommen, dass bestimmte Zusicherungen zwar gelten, aber nicht formuliert werden. Konzeptionell ist dies nur bei Vorbedingungen problematisch; dort wird es auch kaum auftreten. Vorbedingungen beschreiben die Voraussetzungen für einen korrekten Methodenaufruf, und die gibt der Programmierer im Allgemeinen an, damit sie vor Ausführung der Methode geprüft werden.
Bei Nachbedingungen ist man oft nachlässiger. Häufig macht der Programmierer sich über die Nachbedingungen weniger Gedanken als über die Vorbedingungen und formuliert nicht alle Nachbedingungen, die tatsächlich gelten. – Diese Einstellung ist im Sinne von [Meyer 97] durchaus legitim. Meyer vertritt die Position, dass eine Methode für sich selbst entscheiden darf, was sie zusichert und was nicht. Für ein korrektes Zustandsdiagramm ist es jedoch erforderlich, dass das, was gilt, auch wirklich zugesichert wird.
Ein Fehlen von Nachbedingungen findet sich häufig (auch in JWAM) bei Konstruktoren. In den meisten Fällen landet ein neu erzeugtes Objekt nach Ausführung des Konstruktors konzeptionell in einem ganz bestimmten Zustand, d. h. sämtliche Observer haben eine genau festgelegte Belegung. Das sollte dazu führen, dass vom Anfangszustand nur eine Transition zu einem einzigen Zustand führt. In vielen Fällen ist beim Konstruktor jedoch gar keine Nachbedingung angegeben, sodass vom Anfangszustand Transitionen zu sämtlichen Zuständen führen. Ein unübersichtliches und falsches Zustandsdiagramm ist das Resultat, da auch Zustände eingezeichnet sind, die in der Praxis gar nicht erreicht werden können.
Wenn man das generierte Diagramm jedoch sieht, erkennt man das obige Problem leicht, weil es zu viele Anfangszustände gibt. Man braucht lediglich in den Nachbedingungen des Konstruktors die anfänglichen Observerbelegungen anzugeben um den richtigen Anfangszustand zu beschreiben.
Bei fehlenden Nachbedingungen in anderen Methoden ist die Sache schon schwieriger. Hier ist das Verhalten des generierten Zustandsdiagramms genauer zu prüfen.

5.3 Zustände in den JWAM-Klassen

In diesem Abschnitt soll untersucht werden, was für Zustandsdiagramme das besprochene Verfahren für die Klassen des JWAM-Rahmenwerks liefert. In JWAM sind die Vor- und Nachbedingungen wie von uns vorausgesetzt in `@require`- und `@ensure`-Tags angegeben. Dort gibt es allerdings kein `@canchange`-Tag, welches in dieser Arbeit erst eingeführt wurde. Dennoch ist JWAM derzeit das einzige größere Programm, in dem die Zusicherungen im von uns erwarteten Format vorliegen.
Natürlich wurden die Zusicherungen in JWAM ursprünglich nicht zu dem Zweck angegeben, Zustandsdiagramme zu beschreiben, sondern um das Vertragsmodell von Meyer anzuwenden. Aber vielleicht lohnt sich gerade deshalb die Untersuchung von JWAM um herauszufinden, wie brauchbar unser Verfahren wirklich ist.
JWAM hat in der Version 1.6.0 insgesamt 1530 Klassen und Interfaces; darunter fallen 518 Testklassen (mit der Endung „_Test“), die nicht betrachtet werden. Die restlichen 1012 Klassen und Interfaces werden nun im Hinblick auf die generierten Zustandsdiagramme untersucht. Dabei werden wir generell von „Klassen“ sprechen, wohl wissend, dass damit sowohl echte Klassen als auch Interfaces gemeint sind.

5.3.1 Anzahl der unterschiedlichen Zusicherungen

Zunächst wollen wir die JWAM-Klassen im Hinblick auf die Unterscheidung der verschiedenen Arten von Zusicherungen untersuchen. Zu diesem Zweck wurde das Programm dahingehend abgeändert, dass die JWAM-Verzeichnisstruktur rekursiv durchsucht wird und alle Java-Dateien (bis auf die Testklassen) vom Parser eingelesen werden, der dann die Arten der Zusicherungen ermittelt und zählt.
Bei einem anderen solchen Testlauf wurde gezählt, dass JWAM insgesamt 8428 Methoden enthält; darunter fallen 1020 Konstruktoren und 3575 Funktionen.
Die folgende Tabelle zeigt die Verteilung der Zusicherungsarten unter Berücksichtigung aller Methoden:

	Anzahl gesamt	**nicht unterstützt**	**Parameter-Zus.**	**Rückgabewert-Zus.**
require	5193	9	4021	–
ensure	2473	1	419	1622

Dazu ist zu erwähnen, dass bei Nachbedingungen zuerst geprüft wurde, ob es eine Rückgabewert-Zusicherung ist, und erst dann auf Parameter-Zusicherung. Nachbedingungen, die sowohl Rückgabewert- als auch Parameter-Zusicherungen sind, wurden somit als Rückgabewert-Zusicherungen eingestuft.
Die nicht unterstützten Bedingungen sind diejenigen, die Oder-Verknüpfungen enthalten. Es sind nur sehr wenige.
Die Zusicherungen, die nicht in eine der drei obigen Kategorien fallen, sind die Zustands-Zusicherungen, die tatsächlich zur Identifikation der Zustände herangezogen werden. Dies sind also 1163 Vorbedingungen (22,4 %) und 431 Nachbedingungen (17,4 %).
Lässt man die Funktionen außen vor (was ja bei class2statechart standardmäßig der Fall ist), so ergibt sich folgendes Bild:

	Anzahl gesamt	**nicht unterstützt**	**Parameter-Zus.**	**Rückgabewert-Zus.**
require	3200	7	2645	–
ensure	743	1	412	11

Die 11 Rückgabewert-Nachbedingungen sind auf Programmierfehler zurückzuführen, da die hier untersuchten Methoden gar keinen Rückgabewert liefern.
Bei 548 Vorbedingungen (17,1 %) und 319 Nachbedingungen (42,9 %) handelt es sich um Zustands-Zusicherungen.
Dieses Ergebnis zeigt, dass die Zustands-Zusicherungen insgesamt stark in der Unterzahl sind. Die meisten Zusicherungen werden also nicht für die Charakterisierung der Zustände benutzt. An dieser Stelle kann man sich fragen, inwieweit die Parameter-Zusicherungen tatsächlich nur etwas über die Parameter aussagen oder ob viele von ihnen doch Auskunft über Zustandsänderungen geben.[32] Da wir uns aber entschieden haben, nur die Zustands-Zusicherungen zum Beschreiben von Zuständen einzusetzen, werden wir dieser Frage hier nicht weiter nachgehen.

5.3.2 Anzahl der Observer

Wir ermitteln nun, wie viele Observer das Programm class2statechart jeweils findet. Dazu wurde wiederum die JWAM-Verzeichnisstruktur rekursiv durchsucht. Die Ergebnisse wurden in eine Ausgabedatei geschrieben.

[32] vgl. Abschnitt 3.3.1

Folgende Verteilung ergab sich, wobei Zusicherungen von Funktionen nicht berücksichtigt wurden:

Anzahl Observer	Klassen absolut	Klassen prozentual
0	719	71,0 %
1	181	17,9 %
2	44	4,3 %
3	38	3,8 %
4	15	1,5 %
5	6	0,6 %
6	4	0,4 %
7	0	0 %
8	4	0,4 %
9	1	0,1 %

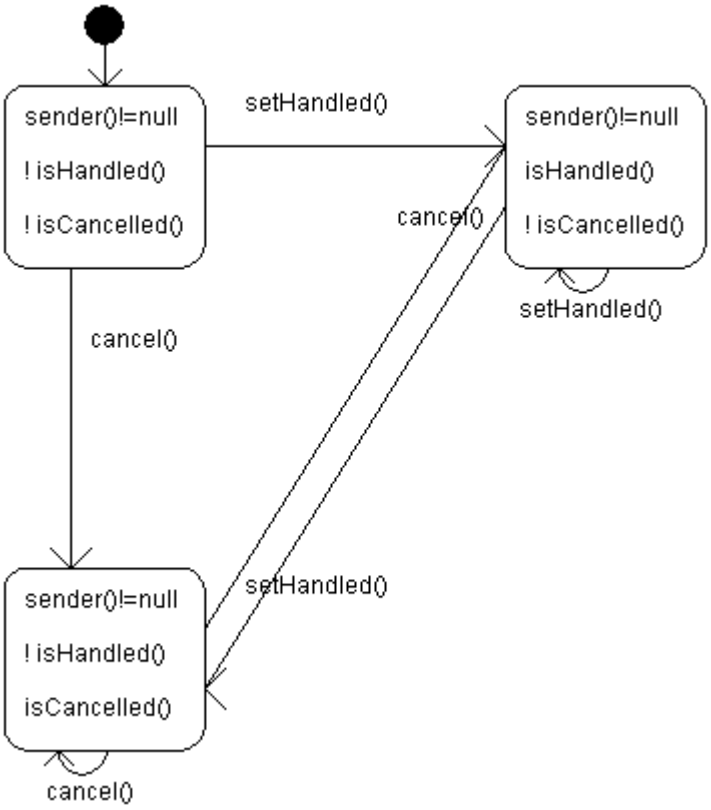
Fast 90 % der Klassen haben weniger als zwei Observer, was zu einem Zustandsdiagramm mit maximal zwei Zuständen führt. Die meisten Klassen haben überhaupt keine Observer; das zugehörige Zustandsdiagramm hat also neben dem Anfangszustand nur einen einzigen Zustand. Das ist durchaus nicht als negativ zu betrachten; es bedeutet, dass alle Methoden zu jedem Zeitpunkt aufgerufen werden dürfen. Für viele Klassen ist das ausreichend und erleichtert die Benutzung.
Bei den Klassen mit vielen Observern ist zu beachten, dass durch Schreibfehler und durch Methodendeklarationen, die über mehrere Zeilen gehen, manchmal zusätzliche Observer generiert werden, die eigentlich nicht beabsichtigt sind. Konzeptionell haben die Klassen also eher weniger Observer als in der Tabelle angegeben.
Um der Frage nachzugehen, wie die Verteilung aussieht, wenn die Observer korrekt ermittelt würden, wurden die Klassen mit mehr als drei Observern mit den generierten Diagrammen verglichen. Dabei wurde versucht festzustellen, welche Observer durch Fehler hereingekommen sind. Die Verteilung bei korrekter Identifikation der Observer sieht dann so aus (für Klassen mit mehr als drei Observern):

Anzahl Observer	Klassen absolut	Klassen prozentual
4	3	0,3 %
5	4	0,4 %
6	1	0,1 %
7	0	0 %
8	2	0,2 %
9	0	0 %

5.3.3 Die generierten Zustandsdiagramme

Die bisherigen Untersuchungen waren rein quantitativer Art. Eine sehr wichtige qualitative Frage ist, wie die generierten Zustandsdiagramme aussehen und ob sich aus ihnen brauchbare Informationen ableiten lassen.
Die Antwort dieser Frage muss differenziert ausfallen. Für manche Klassen ergeben sich tatsächlich brauchbare und aufschlussreiche Zustandsdiagramme; für andere Klassen sind die Diagramme nicht lesbar. Häufig offenbaren sie auch ein unbeabsichtigtes Verhalten.
Überraschend viele Diagramme scheitern daran, dass beim Konstruktor keine Nachbedingung angegeben ist und somit sämtliche Zustände Anfangszustände sind. Dadurch enthält das

Diagramm viele Zustände, die das Objekt in Wirklichkeit niemals einnimmt und die deshalb im korrekten Diagramm als unerreichbare Zustände entfernt worden wären. Oft beobachtet man in einem solchen Diagramm viele Verklemmungen (Deadlocks), also Zustände, aus denen keine Transition herausführt. – Wie schon in Abschnitt 5.2 geschildert sind Schreibfehler in Zusicherungen eine weitere Ursache für fehlerhafte Zustandsdiagramme.
Viele dieser Diagramme lassen sich sehr einfach in lesbare, aussagekräftige Diagramme verwandeln, indem man die Zusicherungen in der Klasse korrigiert. Das Programm kann sehr dabei helfen, solche Fehler zu finden.
Viele Klassen liefern jetzt schon aussagestarke Zustandsdiagramme. Als Beispiel sei hier das generierte Diagramm der Klasse de.jwam.handling.toolconstruction.request.Request angegeben. Folgendes Diagramm erzeugte das Programm (grafisch unverändert):



Dabei ist vielleicht nicht auf den ersten Blick ersichtlich, dass die Transition `cancel()` von rechts oben nach links unten führt und `setHandled()` in die entgegengesetzte Richtung.
Hier offenbaren sich auch konzeptionelle Fehler in den Zusicherungen, denn es ist fraglich, ob ein Aufruf von `setHandled` oder `cancel` überhaupt noch möglich sein soll, wenn entweder `isHandled()` oder `isCancelled()` gilt. Aus den Zuständen rechts und unten sollte also gar keine Transition mehr herausführen.

5.4 Erwägung des Nutzens

Zum Abschluss wollen wir noch der Frage nachgehen, inwieweit die generierten Zustandsdiagramme in der Praxis von Nutzen sind. Die bisherigen Überlegungen und die Untersuchungen am JWAM-Rahmenwerk führten zu folgenden Ergebnissen:

1. **Fehlerhafte Zusicherungen können aufgedeckt werden.** Abschnitt 5.2 hat gezeigt, dass die meisten inkorrekten Zusicherungen erkannt werden können, wenn das erzeugte Diagramm untersucht wird. Das Programm class2statechart ersetzt damit zumindest

teilweise ein Prüfwerkzeug, das die Zusicherungen auf Stimmigkeit überprüft. Da bislang bei JWAM überhaupt kein derartiges Werkzeug eingesetzt wurde, ist class2statechart ein erster Ansatz die Zusicherungen automatisch zu analysieren.

2. **Das Verhalten des Zustandsdiagramms kann überprüft werden.** Während sich der erste Punkt vorwiegend auf syntaktische Fehler bei den Zusicherungen bezog, kann natürlich auch die Semantik der Zusicherungen verifiziert werden. Beispielsweise kann man entdecken, dass ein Methodenaufruf in einem Zustand möglich ist, wo er eigentlich verboten sein sollte. Wenn das Diagramm überschaubar ist (insbesondere wenn die syntaktischen Zusicherungsfehler beseitigt wurden), lassen sich solche semantischen Fehler leicht aufdecken. Dazu zählt auch die Überprüfung, ob alle im Diagramm eingezeichneten Zustände tatsächlich vom Objekt eingenommen werden können. Ist dies nicht der Fall, so ist das ein klares Zeichen dafür, dass bei der Formulierung der Zusicherungen Fehler gemacht wurden.
3. **Bei Änderungen an der Klasse kann das neue Zustandsdiagramm mit dem alten verglichen werden.** Dabei kann das Verhalten der Diagramme auf Kompatibilität geprüft werden. Will man das alte Verhalten beibehalten (bzw. lediglich erweitern), so zeigt ein Vergleich der Diagramme, ob dies geglückt ist. – Auch dieser Punkt bezieht sich letztendlich auf die Korrektheit der Zusicherungen, weil schließlich geprüft wird, ob die neu formulierten Zusicherungen richtig bzw. kompatibel zu den alten sind.

Der Nutzen des entwickelten Verfahrens und des Programms ist mit einem Wort die Überprüfung der Zusicherungen auf Korrektheit. Das ist insofern nicht weiter verwunderlich, als dass das gesamte Verfahren auf den Zusicherungen beruht; die Zusicherungen sind das Einzige von der Klasse, was verwendet wird. Die Fehlersuche gestaltet sich sehr einfach, denn die meisten Fehler offenbaren sich sehr schnell, wenn das generierte Diagramm angezeigt wird. Man kann in mehreren Schritten zuerst syntaktische und dann semantische Fehler korrigieren, was ohne größeren Aufwand möglich sein sollte.
Grafisch noch weiter ausgebessert könnte das Programm auch eine Unterstützung beim täglichen Programmieren bieten. Sehr bequem wäre eine Einbindung in die Entwicklungsumgebung, sodass man sich jederzeit per Knopfdruck das Zustandsdiagramm der gerade bearbeiteten Klasse anzeigen lassen kann. Dies könnte dazu führen, dass sich die Programmierer über das Zustandsverhalten ihrer Klassen intensivere Gedanken machen.

6. Ausblick

Das bisher vorgestellte Verfahren liefert in den meisten Fällen brauchbare Zustandsdiagramme. Es wurden jedoch nicht alle Aspekte berücksichtigt, die prinzipiell möglich sind, weder im Hinblick auf die Zusicherungen noch im Hinblick auf die Ausdrucksmöglichkeiten von Zustandsdiagrammen nach der UML.
In diesem letzten Kapitel sollen daher einige zusätzliche Aspekte untersucht werden. Dabei werden jeweils Ideen angegeben, wie sie in das Verfahren integriert werden könnten, ohne jedoch diese Integration im Detail zu vollziehen.

6.1 Vererbung

Bislang blieb Vererbung unberücksichtigt. Die Zustandsdiagramme wurden erzeugt ohne die Zustandsdiagramme der Oberklassen zu berücksichtigen.
Observer, die sich in Oberklassen befinden, werden hingegen sehr wohl berücksichtigt. Es wird ja nirgends geprüft, wo sich die Observer befinden, die in den Zusicherungen genannt werden.
Allerdings werden die Zustandsdiagramme der Oberklassen nicht in den Generierungsprozess einbezogen. Das wäre aber notwendig, wenn nicht ein Teil des Verhaltens verloren gehen soll. Man müsste sämtliche Methoden, die in der unteren Klasse (deren Zustandsdiagramm generiert werden soll) sichtbar sind, einbeziehen, deren Zustands-Zusicherungen bestimmen und auf dieser Basis das Diagramm erzeugen. Betrachtet man die Methoden der Oberklassen nicht, so erhält man unter Umständen zu wenig Zustände, da möglicherweise einige Zustände nur über Methoden aus Oberklassen erreichbar sind. Außerdem könnten die Oberklassen noch zusätzliche Observer haben.
Eine weitere Frage ist, ob die Zustandsdiagramme einer Klassenhierarchie ebenfalls eine Hierarchie bilden (d. h. ob das Zustandsdiagramm der direkten Oberklasse erweitert wird). Hierzu muss beachtet werden, wie sich die Zusicherungen im Hinblick auf Vererbung verhalten sollen. Im Vertragsmodell von Meyer ist vorgesehen, dass Vorbedingungen in Unterklassen nicht verschärft und Nachbedingungen nicht abgeschwächt werden dürfen.[33] Dabei ist es möglich, dass Transitionen, die in der Oberklasse vorhanden sind, in der Unterklasse wegfallen. Sind z. B. in der Oberklasse die Zusicherungen `@require a` und `@ensure !b` vorhanden, so existiert eine Transition von `a&&b` nach `a&&!b`. Wird die Nachbedingung in der Unterklasse zu `@ensure !b&&!a` verschärft, so entfällt diese Transition. Das Zustandsdiagramm der Unterklasse ist also keine Erweiterung des Diagramms der Oberklasse.

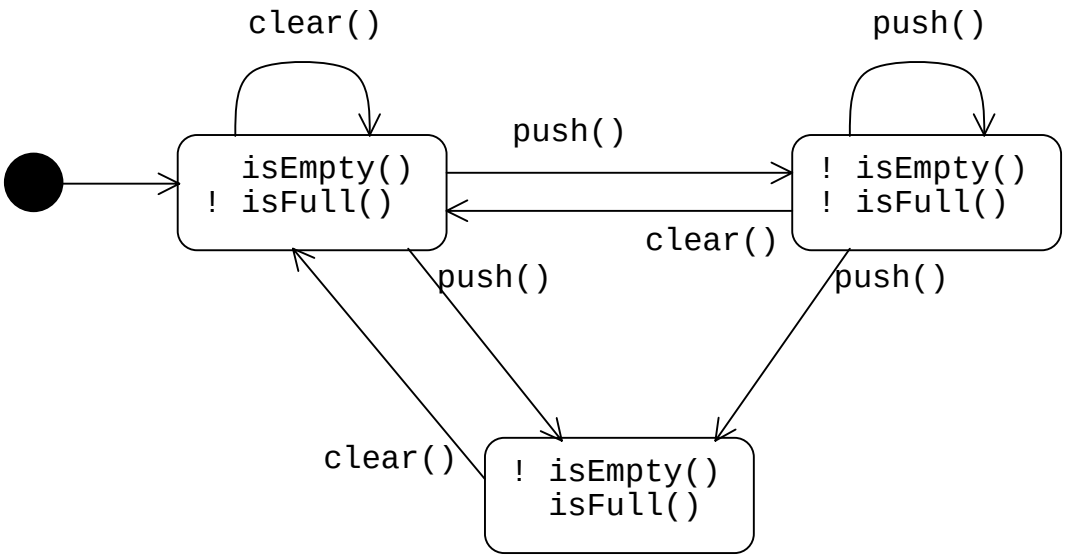
6.2 Unterzustände

Das besprochene Verfahren generiert ausschließlich flache Zustandsdiagramme. Sie sind also äquivalent zu endlichen Automaten. Die UML erlaubt jedoch wesentlich kompaktere Darstellungen von Zustandsdiagrammen, indem beispielsweise Zustände zusammengefasst und Unterzustände gebildet werden können.
Dass das bei unseren generierten Diagrammen durchaus möglich ist, soll an folgendem Beispiel verdeutlicht werden. Wir betrachten wiederum eine Stack-Klasse, wobei diesmal die Elemente nicht einzeln entfernt werden können. Die beiden zustandsändernden Methoden sind `push` zum Ablegen eines Elements und `clear` zum Löschen des gesamten Stacks.
Die Schnittstelle sehe wie folgt aus:

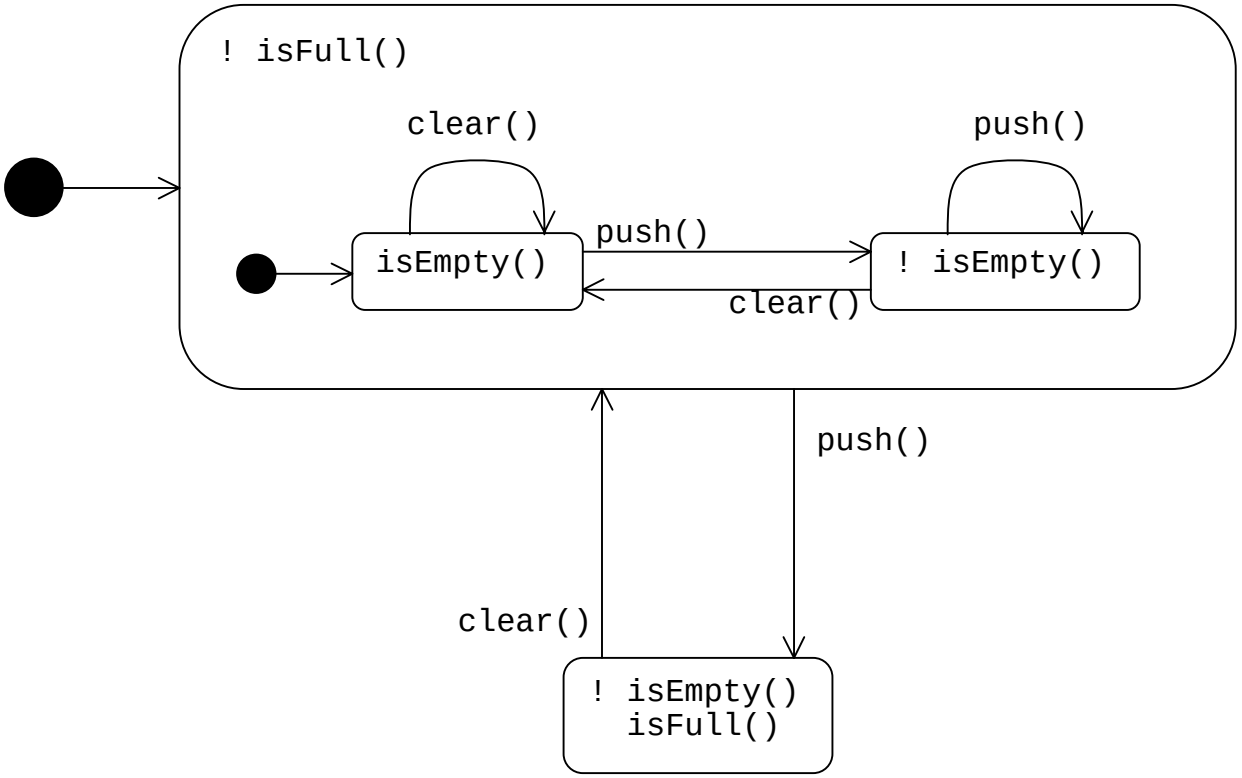
[33] Da wir die Zusicherungen lediglich in Kommentaren angeben, kann dies nur per Konvention geregelt werden.

```
public class Stack
{

     /**
      * @ensure isEmpty() && ! isFull()
      */
     public Stack();

     /**
      * @require ! isFull()
      * @ensure ! isEmpty()
      * @canchange isFull()
      */
     public void push(Object x);

     /**
      * @ensure isEmpty() && ! isFull()
      */
     public void clear();

}
```

Das Zustandsdiagramm dieser Klasse sieht folgendermaßen aus:



Nun ist es möglich, die oberen beiden Zustände zusammenzufassen, denn in beiden Zuständen gilt `!isFull()` und aus beiden Zuständen führt eine durch `push` ausgelöste Transition zum unteren Zustand. Aus dem neuen Zustand, der die beiden oberen zusammenfasst, kann also eine einzige Transition `push()` herausführen. Die eingehenden Transitionen in den neuen Zustand werden vom unteren Zustand durch `clear` und vom Anfangszustand durch den Konstruktor ausgelöst. Beide führen zum linken oberen Zustand, was durch einen Anfangszustand innerhalb des neuen Zustands geregelt werden kann.
Diese Überlegungen führen zu folgendem Zustandsdiagramm mit ähnlichem Verhalten:



Der große Zustand bekommt die Bezeichnung `!isFull()`. Dieser Ausdruck gilt in allen Unterzuständen.
Dieses Beispiel funktioniert nur deshalb, weil für die beiden Zustände, in denen `!isFull()` gilt, Folgendes zutrifft:

- Sämtliche Transitionen, die nach außen führen, führen zum selben Zustand, nämlich `!isEmpty()&&isFull()`.
- Sämtliche Transitionen, die von außen kommen, führen zum selben Zustand, nämlich `isEmpty()&&!isFull()`.

Dabei ist jeweils entscheidend, dass alle Transitionen einer Methode zwischen den Unterzuständen und den außerhalb liegenden Zuständen dasselbe Verhalten zeigen. Bei einer genaueren Untersuchung zu einer systematischen Zusammenfassung von Zuständen wäre zu klären, welche Kriterien es gibt um Zustände zu Unterzuständen zusammenzufassen. Weiterhin wäre zu prüfen, ob es auch möglich ist, orthogonale Zustände zu generieren, also parallele Unterzustände.

6.3 Old-Zusicherungen

In Abschnitt 3.3.4 wurde erläutert, warum wir Zusicherungen, in denen die Old-Schreibweise verwendet wird, nicht betrachtet haben. Wie müssten wir nun vorgehen, wenn wir das Schlüsselwort `old` doch zuließen? In Parameter- und Rückgabewert-Nachbedingungen dürfen sie bisher bereits stehen, da diese nicht für den Generierungsprozess verwendet werden. Interessant ist der Fall, dass die Old-Schreibweise in Zustands-Nachbedingungen verwendet wird.
Hierbei müssen wir mehrere Fälle unterscheiden. Die Methode `push` könnte z. B. zusichern:

```
@ensure size = old size + 1
```

Sehen wir `size` als Observer an, so können wir aus dieser Zusicherung schließen, dass die Belegung von `size` nach Ausführung der Methode in der Werteteilmenge liegt, in der sie vorher lag, oder in der Werteteilmenge, die die nächstgrößere Belegung beschreibt. Enthält die ursprüngliche Werteteilmenge nur ein Element, so liegt die Belegung hinterher in jedem Fall in der nächstgrößeren Teilmenge.
Eine derartige Zuordnung setzt allerdings schon in diesem einfachen Fall viel Wissen über arithmetische Ausdrücke voraus. Es ist sehr fraglich, ob bei komplexeren Ausdrücken eine sinnvolle Automatisierung möglich ist.
Mit der Old-Schreibweise lassen sich auch Ausdrücke formulieren, die nicht sinnvoll sind:

```
@ensure old hasNext()
```

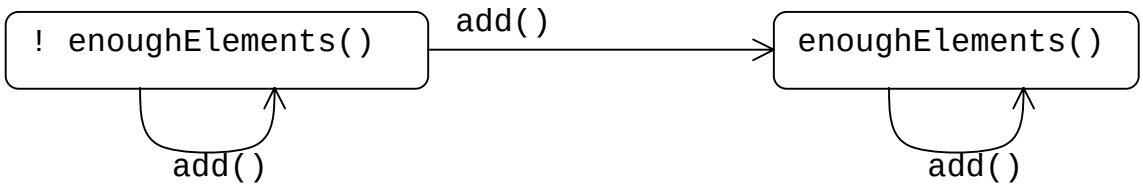
Was soll diese Zusicherung ausdrücken? Wörtlich genommen sagt sie aus, dass vor Ausführung der Methode `hasNext()` galt. Damit diese Nachbedingung auch tatsächlich gilt, muss die Methode folgende Vorbedingung besitzen:

```
@require hasNext()
```

Nun wird die Nachbedingung aber zur Tautologie, da sie exakt dasselbe aussagt. Sie ist also in dieser Form überflüssig.
Folgende Verwendung in der Zustands-Nachbedingung einer Methode `add` kann dagegen sinnvoll sein:

```
@ensure old ! enoughElements() ||
                ( old enoughElements() && enoughElements() )
```

Man beachte, dass `old` stärker bindet als die Und-Verknüpfung.
Diese Zusicherung ist deshalb sinnvoll, weil sie eine Disjunktion von Mintermen darstellt, wobei jeder Minterm denselben Observer in der Old-Schreibweise enthält und die Belegungen dieses Observers den relevanten Werteteilmengen entsprechen.[34] Damit lassen sich Transitionen beschreiben, deren Zielzustände von den Ursprungszuständen abhängen. Im obigen Beispiel heißt das, dass der Zielzustand beliebig ist, wenn vorher `!enoughElements()` galt; wenn das nicht der Fall war, gilt hinterher weiterhin `enoughElements()`.
Das Zustandsdiagramm sieht also wie folgt aus:



Wollten wir diese Art von Old-Zusicherungen in das Verfahren integrieren, so müssten wir die mathematische Modellierung der Nachbedingungen erweitern. Die Nachbedingungen müssten dann in Abhängigkeit von der jeweiligen Vorbedingung modelliert werden. Die Bestimmung der Transitionen wäre etwas komplizierter, aber prinzipiell möglich.

[34] Genau genommen sind nur diejenigen Werteteilmengen erforderlich, die in der Vorbedingung eingenommen werden können.

6.4 Vom Zustandsdiagramm zu den Zusicherungen

Diese Arbeit ist darauf ausgelegt, auf der Basis der Zusicherungen ein Zustandsdiagramm zu generieren. Interessant wäre aber auch die Untersuchung des umgekehrten Wegs: Gegeben ein Zustandsdiagramm, kann man daraus die passenden Zusicherungen ableiten? Auch dies könnte eine Hilfe beim Entwurf von Klassen sein: Man beschreibt das Verhalten der Klasse mithilfe eines Zustandsdiagramms und lässt dann die Zusicherungen der einzelnen Methoden automatisch bestimmen.
Die erste wichtige Frage hierbei wäre, ob sich überhaupt für alle möglichen Zustandsdiagramme Zusicherungen generieren lassen. Wir beschränken uns dabei selbstverständlich auf solche Zustandsdiagramme, wie sie in dieser Arbeit verwendet wurden, also Zustände mit Observer-Belegungen und Methodenaufrufe an den Transitionen.
Wenn wir Old-Zusicherungen ausschließen (was ja in unserem Verfahren geschehen ist), so lassen sich nicht für alle Zustandsdiagramme passende Zusicherungen erzeugen. Das Zustandsdiagramm im vorigen Abschnitt beispielsweise kann nicht durch Zustands-Zusicherungen ausgedrückt werden. Die Vorbedingung von `add` müsste `true` sein, da `add` in jedem Zustand aufgerufen werden kann. Es lässt sich aber keine Zustands-Nachbedingung angeben, die das gewünschte Verhalten nachbildet, wie durch Ausprobieren der Möglichkeiten leicht festgestellt werden kann.
Erst mit Old-Zusicherungen ist das Verhalten beschreibbar (wie im vorigen Abschnitt geschehen). Lässt man Old-Zusicherungen generell zu, so ist in der Tat jedes Zustandsdiagramm auf Zusicherungen abbildbar. Dazu geht man für jede Methode wie folgt vor:

- Die Vorbedingung ist die Disjunktion der Zustände, aus denen Transitionen herausführen, die mit der entsprechenden Methode beschriftet sind.
- Für die Nachbedingung nimmt man die Observer-Belegungen aus jedem Zustand der Vorbedingung in der Old-Schreibweise, und-verknüpft sie mit der Disjunktion der Zielzustände der ausgehenden Transitionen und bildet von all diesen Termen die Disjunktion.

Auf diese Weise erhält man Zusicherungen, die dem Diagramm exakt entsprechen. Es wäre allerdings noch zu klären, wie sich die Old-Zusicherungen in Zusicherungen umformen lassen, die nicht mehr in der Old-Schreibweise stehen, soweit das möglich ist.

Literaturverzeichnis

[BRJ 00] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*. Reading, Massachusetts: Addison-Wesley, 2000

[CHB 92] D. Coleman, F. Hayes, S. Bear*: Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*. In: IEEE Transactions on Software Engineering 18(1), S. 9-18, Januar 1992

[GJSB 00] J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java Language Specification*. Boston, Massachusetts [u. a.]: Addison-Wesley, Second Edition, 2000

[Harel 87] D. Harel: *Statecharts: A visual formalism for complex systems*. In: Science of Computer Programming 8, S. 231-274, 1987

[HG 96] D. Harel, E. Gery: *Executable Object Modeling with Statecharts*. In: 18th International Conference on Software Engineering, S. 246-257, IEEE Computer Society Press, 1996. Auch in: Computer 30(7), S. 31-42, Juli 1997

[LL 98] A. Laue, M. Liedtke: *Eine Einführung in Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, November 1998

[Meyer 92] B. Meyer: *Design by Contract*. In: D. Mandrioli [Hrsg.]: Advances in Object-Oriented Software Engineering. New York; London: Prentice-Hall, 1992

[Meyer 97] B. Meyer: *Object-Oriented Software Construction*. Upper Saddle River, New Jersey: Prentice-Hall, Second Edition, 1997

[Oestereich 98] B. Oestereich: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*. München; Wien: Oldenbourg, 4. Auflage, 1998

[Rumbaugh 95] J. Rumbaugh: *OMT : The dynamic model*. Journal of Object-Oriented Programming 7(9), S. 6-12, Februar 1995

[Schöning 01] U. Schöning: *Theoretische Informatik – kurzgefasst*. Heidelberg; Berlin: Spektrum Akademischer Verlag, 4. Auflage, 2001

[Turau 96] V. Turau: *Algorithmische Graphentheorie*. Bonn; Paris; Reading, Massachusetts: Addison-Wesley, 1996

[Züllighoven 98] H. Züllighoven et al.*: Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. Heidelberg: dpunkt-Verlag, 1998